

Lecteurs-rédacteurs

F. Gluck, *V. Pilloux*

Version 0.35

Introduction

- Le problème des lecteurs-rédacteurs se présente lorsqu'une ressource (structure de donnée, base de donnée, système de fichiers, etc.) est modifiée de manière concurrente par plusieurs threads.
- Lorsque la ressource est modifiée (écrite), aucun thread ne doit pouvoir la lire afin d'éviter tout risque d'incohérence.
- Le but est de permettre une compétition cohérente entre deux types de threads, les lecteurs et les rédacteurs, en respectant les contraintes de synchronisation suivantes :
 - un nombre arbitraire de lecteurs peuvent accéder simultanément à la ressource ;
 - tant qu'au moins un lecteur accède à la ressource, aucun rédacteur ne peut y accéder ;
 - tout rédacteur possède un accès exclusif à la ressource (*i.e.* seulement un seul peut y accéder).

Modèle

- Chaque lecteur effectue continuellement l'opération :

```
read()
```

- Chaque rédacteur effectue continuellement l'opération :

```
write()
```

Lecteurs-rédacteurs

```
readers = 0  
mutex = mutex()  
available = sem_init(1)
```

- `readers` indique le nombre de lecteurs accédant à la ressource
- `mutex` protégeant le compteur
- `available` à 1 si la ressource est libre (read/write), 0 sinon

- Thread rédacteur :

```
write()
```

Lecteurs-rédacteurs

```
readers = 0  
mutex = mutex()  
available = sem_init(1)
```

- `readers` indique le nombre de lecteurs accédant à la ressource
- `mutex` protégeant le compteur
- `available` à 1 si la ressource est libre (read/write), 0 sinon

- Thread rédacteur :

```
wait(available)  
  
write()  
  
post(available)
```

Lecteurs-rédacteurs

```
readers = 0  
mutex = mutex()  
available = sem_init(1)
```

- Thread lecteur :

```
read()
```

- `readers` indique le nombre de lecteurs accédant à la ressource
- `mutex` protégeant le compteur
- `available` à 1 si la ressource est libre (read/write), 0 sinon

- Thread rédacteur :

```
wait(available)  
  
write()  
  
post(available)
```

Lecteurs-rédacteurs

```
readers = 0
mutex = mutex()
available = sem_init(1)
```

- Thread lecteur :

```
lock(mutex)
readers++
if (readers == 1)
    wait(available)
unlock(mutex)

read()

lock(mutex)
readers--
if (readers == 0)
    post(available)
unlock(mutex)
```

- `readers` indique le nombre de lecteurs accédant à la ressource
- `mutex` protégeant le compteur
- `available` à 1 si la ressource est libre (read/write), 0 sinon

- Thread rédacteur :

```
wait(available)

write()

post(available)
```

➤ Cette solution est-elle exempte de problème ?

Problème ?

- La solution précédente est-elle exempte de problème ?
- Non ! Problème de **famine** :
 - si un rédacteur veut accéder à la ressource alors qu'au moins un lecteur y accède encore, il sera bloqué;
 - Le rédacteur pourrait rester bloqué longtemps si de nouveaux lecteurs arrivent avant que le dernier n'ait fini.
- Comme les threads continuent leur progression, il ne s'agit pas d'un deadlock.
- Il se peut qu'un tel programme fonctionne de manière satisfaisante tant que la charge de la machine est faible car les rédacteurs ont encore l'opportunité d'y accéder.
- Par contre, lorsque la charge augmente la situation peut se détériorer rapidement en causant une famine des rédacteurs.

Problème ?

- La solution précédente est-elle exempte de problème ?
- Non ! Problème de famine :
 - si un rédacteur veut accéder à la ressource alors qu'un moins un lecteur y accède encore, il sera bloqué;
 - il restera bloqué à la ressource tant que le dernier n'ait fini.
- Comme les threads s'agit pas d'un deadlock.
- Il se peut qu'un tel programme fonctionne de manière satisfaisante tant que la charge de la machine est faible car les rédacteurs ont encore l'opportunité d'y accéder.
- Par contre, lorsque la charge augmente la situation peut se détériorer rapidement en causant une famine des rédacteurs.

Comment empêcher une famine des rédacteurs ?

Lecteurs-rédacteurs sans famine

```
readers = 0
mutex = mutex()
available = sem_init(1)
turnstile = sem_init(1)
```

- `readers` indique le nombre de lecteurs accédant à la ressource
- `mutex` protégeant le compteur
- `available` à 1 si la ressource est libre (read/write), 0 sinon
- `turnstile` est un tourniquet pour les lecteurs et un mutex pour les rédacteurs

Lecteurs-rédacteurs sans famine

```
readers = 0
mutex = mutex()
available = sem_init(1)
turnstile = sem_init(1)
```

- Thread lecteur :

```
lock(mutex)
readers++
if (readers == 1)
    wait(available)
unlock(mutex)
read()
lock(mutex)
readers--
if (readers == 0)
    post(available)
unlock(mutex)
```

- `readers` indique le nombre de lecteurs accédant à la ressource
- `mutex` protégeant le compteur
- `available` à 1 si la ressource est libre (read/write), 0 sinon
- `turnstile` est un tourniquet pour les lecteurs et un mutex pour les rédacteurs

Lecteurs-rédacteurs sans famine

```
readers = 0
mutex = mutex()
available = sem_init(1)
turnstile = sem_init(1)
```

- Thread lecteur :

```
wait(turnstile)
post(turnstile)

lock(mutex)
readers++
if (readers == 1)
    wait(available)
unlock(mutex)
read()
lock(mutex)
readers--
if (readers == 0)
    post(available)
unlock(mutex)
```

- `readers` indique le nombre de lecteurs accédant à la ressource
- `mutex` protégeant le compteur
- `available` à 1 si la ressource est libre (read/write), 0 sinon
- `turnstile` est un tourniquet pour les lecteurs et un mutex pour les rédacteurs

Lecteurs-rédacteurs sans famine

```
readers = 0
mutex = mutex()
available = sem_init(1)
turnstile = sem_init(1)
```

- Thread lecteur :

```
wait(turnstile)
post(turnstile)

lock(mutex)
readers++
if (readers == 1)
    wait(available)
unlock(mutex)
read()
lock(mutex)
readers--
if (readers == 0)
    post(available)
unlock(mutex)
```

- `readers` indique le nombre de lecteurs accédant à la ressource
- `mutex` protégeant le compteur
- `available` à 1 si la ressource est libre (read/write), 0 sinon
- `turnstile` est un tourniquet pour les lecteurs et un mutex pour les rédacteurs

- Thread rédacteur :

Lecteurs-rédacteurs sans famine

```
readers = 0
mutex = mutex()
available = sem_init(1)
turnstile = sem_init(1)
```

- Thread lecteur :

```
wait(turnstile)
post(turnstile)

lock(mutex)
readers++
if (readers == 1)
    wait(available)
unlock(mutex)
read()
lock(mutex)
readers--
if (readers == 0)
    post(available)
unlock(mutex)
```

- `readers` indique le nombre de lecteurs accédant à la ressource
- `mutex` protégeant le compteur
- `available` à 1 si la ressource est libre (read/write), 0 sinon
- `turnstile` est un tourniquet pour les lecteurs et un mutex pour les rédacteurs

- Thread rédacteur :

```
wait(turnstile)
wait(available)
post(turnstile)
write()
post(available)
```

➤ Cette solution est-elle exempte de problème ?

Problème ?

- La solution précédente est-elle exempte de problème ?
 1. Si un rédacteur veut prendre la ressource alors qu'elle est déjà prise par un lecteur ou plus, il va bloquer sur `wait(available)`. Le tourniquet sera par contre bloqué, empêchant tout nouveau lecteur de venir.
 2. Lorsque le dernier lecteur libère la ressource, il signale `available`, débloquent un rédacteur en attente. Le rédacteur entre donc en section critique car aucun des threads lecteurs en attente ne peut passer le tourniquet.
 3. Lorsque le rédacteur libère la ressource, il signale `turnstile` qui débloquent un des threads en attente (lecteur ou rédacteur). Il est donc possible qu'un lecteur s'empare de la ressource alors que plusieurs rédacteurs sont déjà en attente.
- Selon le type d'application, il peut être préférable de donner la priorité aux rédacteurs. Généralement, l'ordonnanceur décide quel thread débloquent (FIFO, priorités, etc.); **changer la priorité d'un thread est la manière la plus simple pour résoudre ce type de problème (par ex: priorité plus élevée aux rédacteurs)**. La primitive `int pthread_setschedprio(pthread_t thread, int prio)` peut être utilisée à cet effet.