Lecteurs-rédacteurs

PCO

6 - Lectred

Résumé du document

Definition

Table des matières

1. Introduction

Dans le cadre de la gestion des accès aux ressources, il est possible de définir des priorités entre les lecteurs et les rédacteurs. Ces priorités permettent de définir qui a la priorité d'accès à une ressource en cas de conflit. Nous avons plusieurs contraintes à respecter :

- Plusieurs lecteurs peuvent accéder à une ressource en même temps
- Un seul rédacteur peut accéder à une ressource à la fois
- Un rédacteur ne peut pas accéder à une ressource si un lecteur est en train de la lire

Nous avons donc 4 types de solutions possibles :

1. Priorité aux lecteurs
2. Priorité aux lecteurs si lecture en cours
3. Priorité égale
4. Priorité aux rédacteurs

1.1. Exemple

Dans tous les exemple suivants, nous utiliserons une classe abstraite `AbstractReaderWriter` définie comme suit :

```
class AbstractReaderWriter {
public:
  AbstractReaderWriter() {}
  virtual ~AbstractReaderWriter() {}
  virtual void lockReading() = 0;
  virtual void lockWriting() = 0;
  virtual void unlockReading() = 0;
  virtual void unlockWriting() = 0;
};
```

2. Priorité aux lecteurs

2.1. Règles

- Un lecteur peut accéder si le nombre de rédacteur vaut 0
- Un rédacteur peut accéder si
  - Le nombre de rédacteur vaut 0
  - Et le nombre de lecteurs en cours de lecture vaut 0
  - Et le nombre de lecteurs en attente vaut 0

2.2. Implémentation

Pour implémenter cette solution, nous avons besoin de 4 variables :

- `nbReaders` : le nombre de lecteurs en cours de lecture
- `mutexNbreaders` : un mutex pour protéger l'accès à `nbReaders`
- `lockWrite` : qui permet au premier lecteur qui accède de bloquer les futurs rédacateurs ainsi que les lecteurs
- `lockWriters` : qui permet au rédacteur de bloquer les autres rédacteurs, de ce fait un seul rédacteur peut-être en attente sur `lockWrite` et ne brûlera pas la priorité à un lecteur

```
class ReaderWriterPrioReaders :   public AbstractReaderWriter {
protected:
  PcoSemaphore mutexNbReaders;
  PcoSemaphore lockWrite;
  PcoSemaphore lockWriters;
  int nbReaders;
public:
  ReaderWriterPrioReaders() :
    mutexNbReaders(1),
    lockWrite(1),
    lockWriters(1),
    nbReaders(0) {}

  void lockReading();
  void unlockReading();
  void lockWriting();
  void unlockWriting();
};
```

```
void lockReading() {
  mutexNbReaders.acquire();
  nbReaders++;
  if (nbReaders == 1) {
    lockWrite.acquire();
  }
  mutexNbReaders.release();
}

void unlockReading() {
  mutexNbReaders.acquire();
  nbReaders -= 1;
  if (nbReaders == 0) {
    lockWrite.release();
  }
  mutexNbReaders.release();
}
```

```
void lockWriting() {
  lockWriters.acquire();
  lockWrite.acquire();
}
void unlockWriting() {
  lockWrite.release();
  lockWriters.release();
}
```

3. Priorité aux lecteurs si lecture en cours

Pour offrir une priorité constante aux lecteurs, il suffit de partir de la solution priorités lecteurs et de ne plus utiliser de sémaphore `lockWriters` pour bloquer les rédacteurs. Les rédacteurs devront attendre que tous les lecteurs aient terminé leur lecture pour pouvoir accéder à la ressource.

```
class ReaderWriterPrioReading : public AbstractReaderWriter {
protected:
  PcoSemaphore mutexNbReaders;
  PcoSemaphore lockWrite;
  int nbReaders;
public:
  ReaderWriterPrioReading() :
    mutexNbReaders(1),
    lockWrite(1),
    nbReaders(0) {}

  void lockReading();
  void unlockReading();
  void lockWriting();
  void unlockWriting();
};
```

```
void lockReading() {
  mutexNbReaders.acquire();
  nbReaders++;
  if (nbReaders == 1) {
    lockWrite.acquire();
  }
  mutexNbReaders.release();
}

void unlockReading() {
  mutexNbReaders.acquire();
  nbReaders -= 1;
  if (nbReaders == 0) {
    lockWrite.release();
  }
  mutexNbReaders.release();
}
```

```
void lockWriting() {
  lockWrite.acquire();
}
void unlockWriting() {
  lockWrite.release();
}
```

4. Priorité égale

4.1. Règles

- Un lecteur peut accéder si le nombre de rédacteur vaut 0
- Un rédacteur peut accéder si
  - Le nombre de rédacteur vaut 0
  - Et le nombre de lecteurs en cours de lecture vaut 0

4.2. Implémentation

Pour implémenter cette solution, nous avons besoin de 4 variables :

- `nbReaders` : le nombre de lecteurs en cours de lecture
- `mutexNbreaders` : un mutex pour protéger l'accès à `nbReaders`
- `lockWrite` : qui permet au premier lecteur qui accède de bloquer les futurs rédacateurs
  - **Attention** `lockWrite` ne sera pas utilisé par les rédacteurs pour blqouer les lecteurs
- `fifo` : une file d'attente (sémaphore) pour tous les lecteurs et rédacteurs

```
class ReaderWriterEqual : public AbstractReaderWriter {
protected:
    PcoSemaphore mutexNbReaders;
    PcoSemaphore lockWrite;
    PcoSemaphore fifo;
    int nbReaders;

public:
    ReaderWriterEqual()
        : mutexNbReaders(1), lockWrite(1), fifo(1), nbReaders(0) {}

    void lockReading() {}
    void unlockReading() {}
    void lockWriting() {}
    void unlockWriting() {}
};
```

```
void lockReading() {
    fifo.acquire();
    mutexNbReaders.acquire();
    nbReaders++;
    if (nbReaders == 1) {
        lockWrite.acquire();
    }
    mutexNbReaders.release();
    fifo.release();
}

void unlockReading() {
    mutexNbReaders.acquire();
    nbReaders -= 1;
    if (nbReaders == 0) {
        lockWrite.release();
    }
    mutexNbReaders.release();
}
```

```
void lockWriting() {
    fifo.acquire();
    lockWrite.acquire();
}

void unlockWriting() {
    lockWrite.release();
    fifo.release();
}
```

5. Priorité aux rédacteurs

5.1. Règles

- Un lecteur peut accéder si
  - Le nombre de rédacteur vaut 0
  - Et le nombre de rédacteur en attente vaut 0
- Un rédacteur peut accéder si
  - Le nombre de rédacteur vaut 0
  - Et le nombre de lecteurs en cours de lecture vaut 0

5.2. Implémentation

Pour implémenter cette solution, nous avons besoin de 7 variables :

- `nbReaders` : le nombre de lecteurs en cours de lecture
- `mutexNbreaders` : un mutex pour protéger l'accès à `nbReaders`
- `nbWriters` : le nombre de rédacteurs en cours d'écriture
- `mutexNbWriters` : un mutex pour protéger l'accès à `nbWriters`
- `lockWrite` : qui permet au premier lecteur qui accède de bloquer les futurs rédacateurs
- `lockRead` : qui permet au premier rédacteur qui accède de bloquer les futurs lecteurs
- `lockReaders` : qui permet de favoriser les rédacteurs

```
class ReaderWriterPrioWriter : public AbstractReaderWriter {
protected:
    PcoSemaphore mutexNbReaders, mutexNbWriters;
    PcoSemaphore lockWrite, lockRead;
    PcoSemaphore lockReaders;
    int nbReaders, nbWriters;

public:
    ReaderWriterPrioWriter()
        : mutexNbReaders(1), mutexNbWriters(1),
          lockWrite(1), lockRead(1),
          lockReaders(1),
          nbReaders(0), nbWriters(0) {}

    void lockReading() {}
    void unlockReading() {}
    void lockWriting() {}
    void unlockWriting() {}
};
```

```
void lockReading() {
    lockReaders.acquire();
    lockRead.acquire();
    mutexNbReaders.acquire();
    nbReaders++;
    if (nbReaders == 1)
        lockWrite.acquire();
    mutexNbReaders.release();
    lockRead.release();
    lockReaders.release();
}

void unlockReading() {
    mutexNbReaders.acquire();
    nbReaders -= 1;
    if (nbReaders == 0)
        lockWrite.release();
    mutexNbReaders.release();
}
```

```
void lockWriting() {
    mutexNbWriters.acquire();
    nbWriters++;
    if (nbWriters == 1)
        lockRead.acquire();
    mutexNbWriters.release();
    lockWrite.acquire();
}

void unlockWriting() {
    lockWrite.release();
    mutexNbWriters.acquire();
    nbWriters -= 1;
    if (nbWriters == 0)
        lockRead.release();
    mutexNbWriters.release();
}
```