Introduction à la programmation concurrente

Lecteurs-rédacteurs

Yann Thoma, Jonas Chapuis

Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Février 2018

Enoncé du problème

- Plusieurs threads doivent accéder à une ressource
- Threads lecteurs
  - Ne peuvent que lire les données
- Threads rédacteurs
  - Peuvent modifier les données

Contraintes du problème

1. Plusieurs lecteurs peuvent lire simultanément les données;
2. Les rédacteurs s'excluent mutuellement;
3. Les lecteurs et les rédacteurs s'excluent mutuellement.

Question

- Pourquoi pas une simple section critique pour protéger la ressource?

Types de solutions

1. Priorité aux lecteurs (famine possible des rédacteurs)
2. Priorité aux lecteurs si un lecteur a déjà accès à la ressource
3. Priorité aux rédacteurs (famine possible des lecteurs)
4. Accès aux données selon les ordres des arrivées. Toutes les demandes de lecteur qui se suivent sont satisfaites en même temps.

Classe abstraite

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

Toutes les sources de cette présentation sont regroupées dans un même projet accessible depuis le lien de cette page

Priorité aux lecteurs

- Les règles sont les suivantes:
- Un lecteur peut accéder à la ressource si:
  - Le nombre de rédacteurs en cours d'écriture vaut 0
- Un rédacteur peut accéder à la ressource si:
  - Le nombre de rédacteurs en cours d'écriture vaut 0
  - ET le nombre de lecteurs en cours de lecture vaut 0
  - ET le nombre de lecteurs en attente de la ressource vaut 0

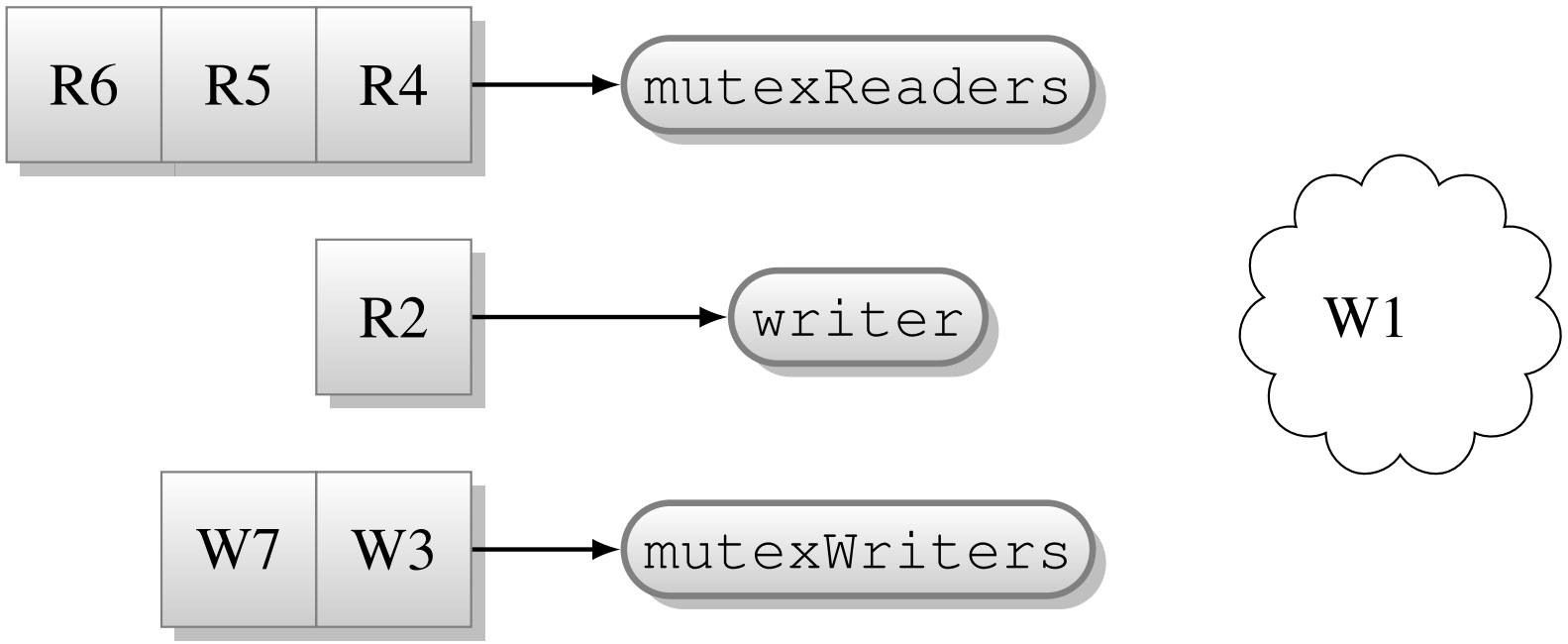
Sémaphores nécessaires

- Une variable `nbReaders`
- `mutexReaders`, qui est en charge de protéger l'accès à la variable `nbReaders`.
- `writer`, qui permet au premier lecteur qui accède la ressource de bloquer les futurs rédacteurs. Il permet également au rédacteur accédant la ressource de bloquer les lecteurs pendant l'écriture.
- `mutexWriters`, qui permet au rédacteur accédant la ressource de bloquer les autres rédacteurs. De ce fait, un seul rédacteur peut être en attente du sémaphore `writer`, empêchant ainsi un rédacteur de brûler la priorité à un lecteur.

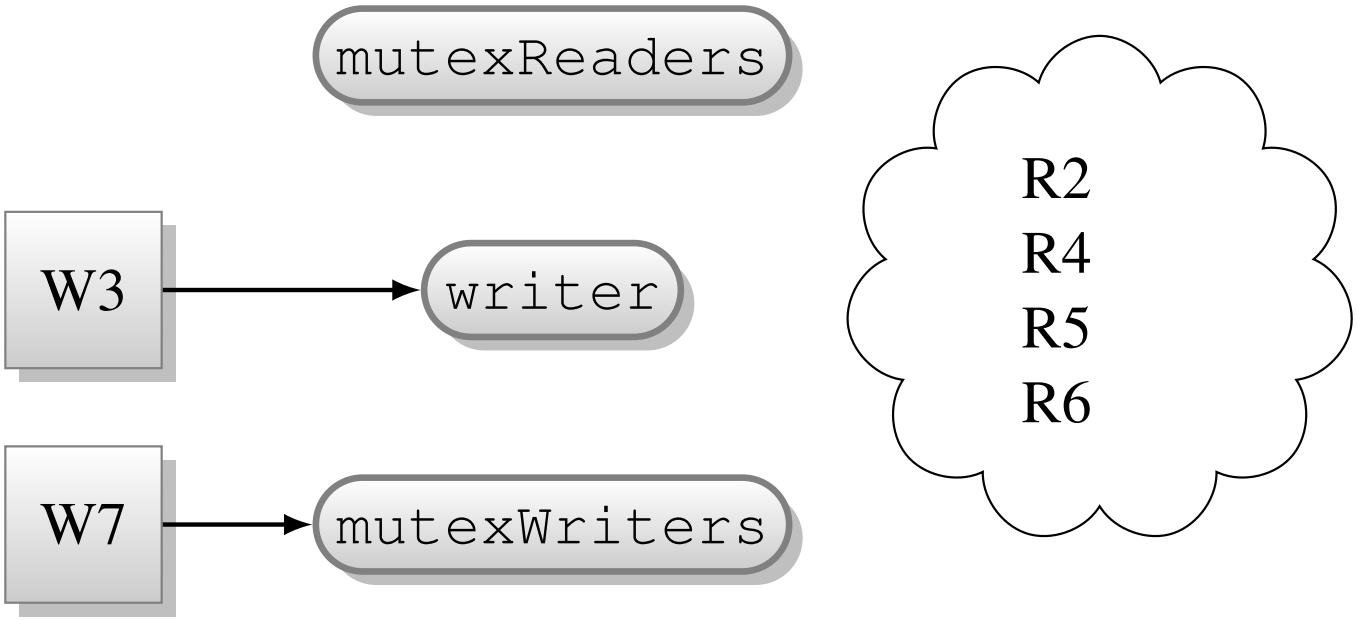
Algorithme possible

```
class ReaderWriterPrioReaders :
  public AbstractReaderWriter {
protected:
  QSemaphore mutexReaders;
  QSemaphore mutexWriters;
  QSemaphore writer;
  int nbReaders;

public:
  ReaderWriterPrioReaders() :
    mutexReaders(1),
    mutexWriters(1),
    writer(1),
    nbReaders(0){}

  void lockReading() {
    mutexReaders.acquire();
    nbReaders++;
    if (nbReaders==1) {
      writer.acquire();
    }
    mutexReaders.release();
  }

  void unlockReading() {
    mutexReaders.acquire();
    nbReaders -= 1;
    if (nbReaders==0) {
      writer.release();
    }
    mutexReaders.release();
  }

  void lockWriting() {
    mutexWriters.acquire();
    writer.acquire();
  }

  void unlockWriting() {
    writer.release();
    mutexWriters.release();
  }
};
```

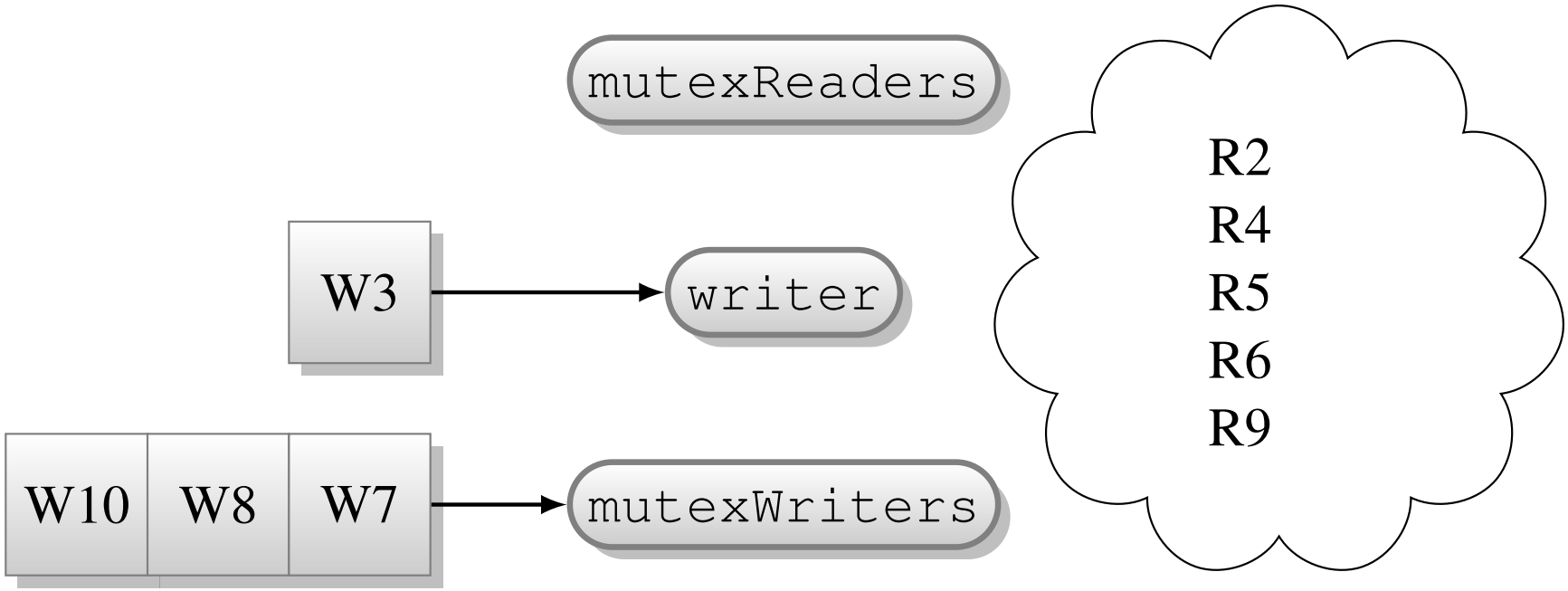
Exemple (1)



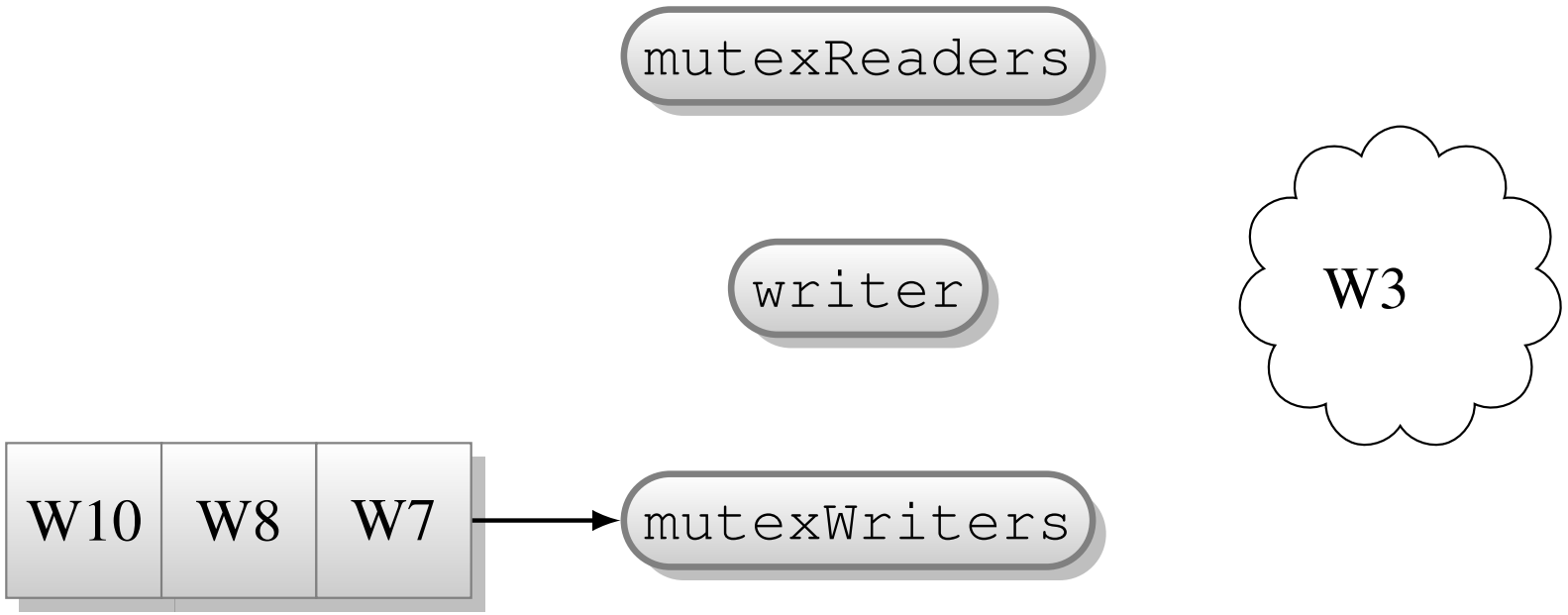
Exemple (2)



Exemple (3)



Exemple (4)



Priorité aux lecteurs

- Scénario:
  - Arrivée de $R1, R2, R3, W4, W5, R6, R7, W8$
  - Tous les lecteurs quittent l'accès à la ressource
  - Arrivée de $R9, W10, R11, R12$

Programme de test (1)

```
AbstractReaderWriter *resource;

void TaskWriter::run() {
  while(1) {
    resource->lockWriting();
    std::cout << "Task " << tid << ": écriture" << std::endl;
    resource->unlockWriting();
  }
}

void TaskReader::run() {
  while(1) {
    resource->lockReading();
    std::cout << "Task " << tid << ": écriture" << std::endl;
    resource->unlockReading();
  }
}
```

Programme de test (2)

```
#define NB_WRITERS   1
#define NB_READERS   3

int main (int argc, char *argv[]) {
  TaskWriter threadsWriter[NB_WRITERS];
  TaskReader threadsReader[NB_READERS];
  int t;
  int idLec[NB_READERS];
  int idRed[NB_WRITERS];
  resource = new ReaderWriterPrioReaders();
  for(t=0; t<NB_READERS; t++) {
    std::cout << "Creating the reader " << t << std::endl;
    threadsReader[t].tid = t;
    threadsReader[t].start();
  }
  for(t=0; t<NB_WRITERS; t++) {
    std::cout << "Creating the writer " << t << std::endl;
    threadsWriter[t].tid = t;
    threadsWriter[t].start();
  }
  for(t=0; t<NB_READERS; t++) {
    threadsReader.wait();
  }
  for(t=0; t<NB_WRITERS; t++) {
    threadsWriter.wait();
  }
  return 0;
}
```

Priorité aux lecteurs si lecture en cours

```
class ReaderWriterPrioReading : public AbstractReaderWriter {
protected:
  QSemaphore mutexReaders;
  QSemaphore writer;
  int nbReaders;
public:
  ReaderWriterPrioReading() :
    mutexReaders(1),
    writer(1),
    nbReaders(0){}
```

Priorité aux lecteurs si lecture en cours

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

  void unlockReading() {
    mutexReaders.acquire();
    nbReaders -= 1;
    if (nbReaders==0) {
      writer.release();
    }
    mutexReaders.release();
  }

  void lockWriting() {
    writer.acquire();
  }

  void unlockWriting() {
    writer.release();
  }
};
```

Priorité aux lecteurs si lecture en cours

- Scénario: $R1, R2, R3, W4, W5, R6, R7, W8, R9, R10$
- $R9$ arrive après que $R1, R2, R3, R6, R7$ ont terminé

Priorité aux lecteurs: algorithme général

- Algorithme selon un autre modèle

```
class ReaderWriterPrioReader2 :
      public AbstractReaderWriter {
protected:
  QSemaphore mutex;
  QSemaphore readerBlocker;
  QSemaphore writerBlocker;
  int nbReaders;
  int nbReadersWaiting;
  int nbWritersWaiting;
  bool oneWriter;
```

```
public:

  ReaderWriterPrioReader2() :
    mutex(1),
    nbReaders(0),
    nbReadersWaiting(0),
    nbWritersWaiting(0),
    oneWriter(false) {}
```

Algorithme général

```
void lockReading() {
  mutex.acquire();
  if (oneWriter) {
    nbReadersWaiting++;
    mutex.release();      // ouverture
    readerBlocker.acquire();
  }
  else {
    nbReaders++;
    mutex.release();
  }
}

void unlockReading() {
  mutex.acquire();
  nbReaders--;
  if (nbReaders==0) {
    if (nbWritersWaiting>0) {
      oneWriter=true;
      nbWritersWaiting--;
      writerBlocker.release();
    }
  }
  mutex.release();
}
```

```
  void lockWriting() {
    mutex.acquire();
    if (oneWriter || (nbReaders>0)
        || (nbReadersWaiting>0)) {
      nbWritersWaiting++;
      mutex.release();       // ouverture
      writerBlocker.acquire();
    }
    else {
      oneWriter=true;
      mutex.release();}}

  void unlockWriting() {
    mutex.acquire();
    oneWriter=false;
    if (nbReadersWaiting>0) {
      for(int i=0;i<nbReadersWaiting;i++)
        readerBlocker.release();
      nbReaders=nbReadersWaiting;
      nbReadersWaiting=0;
    }
    else {
      if (nbWritersWaiting>0) {
        oneWriter=true;
        nbWritersWaiting--;
        writerBlocker.release();
      }
    }
    mutex.release();
  }
};
```

Priorité égale

- Les règles sont les suivantes:
  - Un lecteur peut accéder à la ressource si:
    - Le nombre de rédacteurs en cours d'écriture vaut 0
  - Un rédacteur peut accéder à la ressource si:
    - Le nombre de rédacteurs en cours d'écriture vaut 0
    - ET le nombre de lecteurs en cours de lecture vaut 0

Sémaphores nécessaires

- Une variable `nbReaders`
- `mutex`, qui est en charge de protéger l'accès à la variable `nbReaders`.
- `writer`, qui permet au premier lecteur qui accède la ressource de bloquer les futurs rédacteurs. Il permet également au rédacteur accédant la ressource de bloquer les lecteurs pendant l'écriture.
- `fifo`, une file d'attente dans laquelle passent tous les lecteurs et rédacteurs.

Algorithme possible

```
class ReaderWriterEqual : public AbstractReaderWriter {
protected:
  QSemaphore mutex;
  QSemaphore fifo;
  QSemaphore writer;
  int nbReaders;

public:
  ReaderWriterEqual() :
    mutex(1),
    fifo(1),
    writer(1),
    nbReaders(0) {}

  void lockReading() {
    fifo.acquire();
    mutex.acquire();
    nbReaders++;
    if (nbReaders==1) {
      writer.acquire();
    }
    mutex.release();
    fifo.release();
  }

  void unlockReading() {
    mutex.acquire();
    nbReaders -= 1;
    if (nbReaders==0) {
      writer.release();
    }
    mutex.release();
  }

  void lockWriting() {
    fifo.acquire();
    writer.acquire();
  }

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

Priorité aux rédacteurs

- Les règles sont les suivantes:
  - Un lecteur peut accéder à la ressource si:
    - Le nombre de rédacteurs en cours d'écriture vaut 0
    - ET le nombre de rédacteurs en attente d'écriture vaut 0
  - Un rédacteur peut accéder à la ressource si:
    - Le nombre de rédacteurs en cours d'écriture vaut 0
    - ET le nombre de lecteurs en cours de lecture vaut 0

Sémaphores nécessaires

- Une variable `nbReaders`
- `mutexReaders`, qui permet de bloquer les lecteurs pendant que des écritures sont en cours.
- `mutexWriters`, qui permet de bloquer les rédacteurs pendant que des écritures ou des lectures sont en cours.
- `mutex`, qui est en charge de protéger l'accès à la variable `nbReaders`.
- `writer`, qui permet au premier lecteur qui accède la ressource de bloquer les potentiels rédacteurs.
- `reader`, qui permet au premier rédacteur arrivé de bloquer les potentiels futurs lecteurs.

Algorithme possible

```
class ReaderWriterPrioWriter : public AbstractReaderWriter {
protected:
  QSemaphore mutexReaders;
  QSemaphore mutexWriters;
  QSemaphore writer;
  QSemaphore reader;
  QSemaphore mutex;
  int nbReaders, nbWriters;

public:
  ReaderWriterPrioWriter() :
    mutexReaders(1),
    mutexWriters(1),
    writer(1),
    reader(1),
    mutex(1),
    nbReaders(0),
    nbWriters(0) {}

  void lockReading() {
    mutexReaders.acquire();
    reader.acquire();
    mutex.acquire();
    nbReaders++;
    if (nbReaders==1)
      writer.acquire();
    mutex.release();
    reader.release();
    mutexReaders.release();
  }

  void unlockReading() {
    mutex.acquire();
    nbReaders -= 1;
    if (nbReaders==0)
      writer.release();
    mutex.release();
  }

  void lockWriting() {
    mutexWriters.acquire();
    nbWriters++;
    if (nbWriters==1)
      reader.acquire();
    mutexWriters.release();
    writer.acquire();
  }

  void unlockWriting() {
    writer.release();
    mutexWriters.acquire();
    nbWriters -= 1;
    if (nbWriters==0)
      reader.release();
    mutexWriters.release();
  }
};
```

Exercices

- Reprendre l'algorithme général avec priorité aux lecteurs, et l'adapter pour la priorité aux rédacteurs.
- Mettre au point un programme capable de tester un des algorithmes présentés dans cette section.
- Une application est composée de threads de deux classes, A et B. Une ressource est partagée entre tous les threads, selon les contraintes suivantes:
  1. Les threads de classe A peuvent accéder concurrement à la ressource.
  2. Les threads de classe B peuvent accéder concurrement à la ressource.
  3. Les threads de différente classe ne peuvent accéder à la ressource au même instant.

  Proposer un algorithme permettant de gérer l'accès à la ressource, en s'inspirant des solutions du chapitre. Considérez une solution où la coalition est possible entre threads d'une même classe.

Code source

`http://reds.heig-vd.ch/share/cours/PCO/cours/code/6-lectred/lectred1.tar.gz`