Producteurs-consommateurs

PCO

5 - Prodcons

Résumé du document

Definition

Table des matières

1. Problématique

Lors-ce que nous souhaitons que deux threads puissent se transmettre des données, nous aurons deux types de threads:

- un thread producteur
- un thread consommateur

Il existe plusieurs variations:

- taille du tampon
  - tampon simple
  - tampon multiple
- nombre de threads
  - 1 producteur et 1 consommateur
  - N producteurs et M consommateurs

1.1. Contraintes

1. Les éléments contenus dans le tampon ne sont consommées qu'une seule fois
2. Les éléments du tampon sont consommés selon leur ordre de production
3. Il n'y a pas d'écrasement prématuré des éléments du tampon, autrement dit, si le tampon est plein, une tâche productrice doit attendre la libération d'un élément du tampon

- Les deux types d'action concurrentes:
  - **Production** : attendre que le tampon soit libre puis déposer du contenu
  - **Consommation** : attendre du contenu dans le tampon puis le prélever

1.2. Buffer abstrait

- Toutes nos implémentations dériveront d'une classe template abstraite

```
template<typename T>
class AbstractBuffer {
public:
  virtual void put(T item) = 0;
  virtual T get() = 0;
};
```

2. Tampon simple

- fonctionne à base de sémaphores
- exploitant un maximum les capacités des sémaphores
- deux sémaphores permettent de gérer l'attente des producteurs et celle des consommateurs
- un sémaphore pour faire attendre les consommateurs
  - `waitFull` vaut 1 si le tampon est plein
- un sémaphore pour faire attendre les producteurs
  - `waitEmpty` vaut 1 si le tampon est vide
- `waitFull` + `waitEmpty` = 1

2.1. Exemple de tampon simple

```
template<typename T> class Buffer1a : public AbstractBuffer<T> {
public:
  Buffer1a() : waitEmpty(1) {}

  virtual ~Buffer1a() {}

  void put(T item) override {
    waitEmpty.acquire();
    element = item;
    waitFull.release();
  }

  T get() override {
    T item;
    waitFull.acquire();
    item = element;
    waitEmpty.release();
    return item;
  }

protected:
  T element;
  PcoSemaphore waitEmpty, waitFull;
};
```

3. Tampon de taille N

- le tampon partagé contient N éléments
- problème à résoudre
  - synchronisation des tâches
  - gestion du temps
- le tampon est une liste circulaire
  - un pointeur `writePointer` pour l'écriture
    - initialisé à 0
  - un pointeur `readPointer` pour la lecture
    - initialisé à 0

3.1. Exemple

Voici un exemple qui prend en compte la protection de l'accès multiple à la section critique modifiant les variables `readPointer` ainsi que `writePointer`. Pour cela il faut rajouter un sémaphore initalisé à `1`.

```
#include <PcoSemaphore>

template<typename T>
class BufferNa : public AbstractBuffer<T> {
protected:
    std::vector<T> elements;
    int writePointer;
    int readPointer;
    int bufferSize;
    PcoSemaphore mutex, waitNotFull, waitNotEmpty;

public:
    BufferNa(unsigned int size) : elements(size), writePointer(0),
        readPointer(0), bufferSize(size),
        mutex(1), waitNotFull(size), waitNotEmpty(0) {}

    virtual ~BufferNa() {}

    void put(T item) override {
        waitNotFull.acquire();  // Attente que le tampon ne soit pas plein
        mutex.acquire();  // Protection de la section critique
        elements[writePointer] = item;
        writePointer = (writePointer + 1) % bufferSize;
        waitNotEmpty.release();  // Libère l'attente pour un consommateur
        mutex.release();  // Libère la protection
    }

    T get() override {
        T item;
        waitNotEmpty.acquire();  // Attente que le tampon ne soit pas vide
        mutex.acquire();  // Protection de la section critique
        item = elements[readPointer];
        readPointer = (readPointer + 1) % bufferSize;
        waitNotFull.release();  // Libère l'attente pour un producteur
        mutex.release();  // Libère la protection
        return item;
    }
};
```

3.2. Exemple optimisé

Dans cet exemple, nous avons une version optimisé en mettant en place un passage de `mutex` entre la méthode `put` et `get`.

```
#include "abstractbuffer.h"
#include <pcosynchro/pcosemaphore.h>

template<typename T> class BufferN : public AbstractBuffer<T> {
protected:
    std::vector<T> elements;
    int writePointer, readPointer, nbElements, bufferSize;
    PcoSemaphore mutex, waitProd, waitConso;
    unsigned nbWaitingProd, nbWaitingConso;

public:

    BufferN(unsigned int size) : elements(size), writePointer(0),
                                 readPointer(0), nbElements(0),
                                 bufferSize(size),
                                 mutex(1), waitProd(0),waitConso(0),
                                 nbWaitingProd(0), nbWaitingConso(0) {
       }

    virtual ~BufferN() {}

    virtual void put(T item) {}

    virtual T get(void) {}
};
```

```
virtual void put(T item) {
  mutex.acquire();
  if (nbElements == bufferSize) {
      nbWaitingProd += 1;
      mutex.release();
      waitProd.acquire();
  }
  elements[writePointer] = item;
  writePointer = (writePointer + 1)
                 % bufferSize;
  nbElements ++;
  if (nbWaitingConso > 0) {
      nbWaitingConso -= 1;
      waitConso.release();
  }
  else {
      mutex.release();
  }
}
```

```
virtual T get(void) {
  T item;
  mutex.acquire();
  if (nbElements == 0) {
      nbWaitingConso += 1;
      mutex.release();
      waitConso.acquire();
  }
  item = elements[readPointer];
  readPointer = (readPointer + 1)
                % bufferSize;
  nbElements --;
  if (nbWaitingProd > 0) {
      nbWaitingProd -= 1;
      waitProd.release();
  }
  else {
      mutex.release();
  }
  return item;
}
```

Dans cet exemple on peut voir que la méthode `put`dans le cas ou des consommateurs attendent, on libère un consommateur et on ne libère pas le mutex. Cela s'applique aussi pour la méthode `get` dans le cas ou des producteurs attendent. Cela permet de réduire le **vérouillage / déverrouillage du mutex**.