Introduction à la programmation concurrente

Producteurs-consommateurs

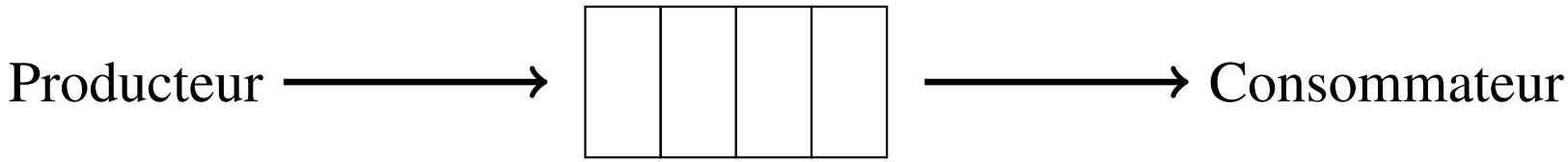
Yann Thoma, Jonas Chapuis

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

Février 2018

Enoncé du problème

- Deux threads doivent se transmettre des données au travers d'un tampon
- Un thread producteur
- Un thread consommateur



- Cadre:
  - Taille du tampon:
    - Un tampon simple
    - Un tampon multiple
  - Nombre de threads:
    - 1 producteur et 1 consommateur
    - N producteurs et M consommateurs

Contraintes

1. Les éléments contenus dans le tampon ne sont consommés 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 tampons, autrement dit, si le tampon est plein, une productrice doit attendre la libération d'un élément du buffer.

Contraintes (synchronisation)

- Soit
  - $Produit(t)$, le nombre d'éléments introduits dans le tampon jusqu'au temps t,
  - $Consommé(t)$, le nombre d'éléments retirés du tampon jusqu'à t,
  - N, la capacité du tampon,
- alors $\forall t \geq 0 : 0 \leq Produit(t) - Consommé(t) \leq N$.
- Les actions des tâches deviennent alors
  - Productrice : attendre que $Produit(t) - Consommé(t) < N$, puis déposer l'item produit;
  - Consommatrice : attendre que $Produit(t) - Consommé(t) > 0$, puis prélever l'item produit.

Pseudo-code

Producteur

```
Depose(item) {
    Tant que le tampon est plein:
          Attendre
    Déposer l'item
    Signaler ceci au consommateur
}
```

Consommateur

```
item Preleve() {
    Tant que le tampon est vide:
        Attendre
    Retirer l'item
    Signaler ceci au producteur
}
```

Buffer abstrait

- Toutes nos implémentations dériveront d'une classe abstraite

Classe abstraite

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

Tâches productrices-consommatrices

```
static AbstractBuffer<ITEM> *buffer;

void Producer::run() {
  ITEM item;
  while (true) {
     // produire item
     buffer->put(item);
  }
}

void Consumer::run() {
  ITEM item;
  while (true) {
     item = buffer->get();
     // consommer item
  }
}

int main(void) {
  Producer prod;
  Consumer cons;
  buffer = new Buffer1<ITEM>();
  prod.start();
  cons.start();
  prod.wait();
  cons.wait();
  return 0;
}
```

Tampon simple: 1er algorithme

- A base de sémaphores
- En 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` compte le nombre de tampons pleins
- Un sémaphore pour faire attendre les producteurs
  - `waitEmpty` compte le nombre de tampons vides
- `waitFull` + `waitEmpty` = 1

Tampon simple: 1er algorithme

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

    virtual ~Buffer1a() {}

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

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

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

Tampon simple: 2ème algorithme

- A base de sémaphores
- En exploitant des variables pour représenter l'état du système
- Une variable d'état: `Empty` ou `Full`
- Un sémaphore pour la protéger
- Un sémaphore pour faire attendre les consommateurs
- Un sémaphore pour faire attendre les producteurs
- Une variable pour le nombre de consommateurs en attente
- Une variable pour le nombre de producteurs en attente

Tampon simple: 2ème algorithme

```
template<typename T>
class Buffer1 : public AbstractBuffer<T> {

protected:
    T element;
    QSemaphore mutex, waitEmpty, waitFull;
    enum {Full, Empty} state;
    unsigned nbWaitingProd, nbWaitingCons;

public:

    Buffer1() : mutex(1),        <-- On ouvre le verrou
        state(Empty),
        nbWaitingProd(0),
        nbWaitingCons(0)
    {}

    virtual ~Buffer1() {}
```

Tampon simple: 2ème algorithme

```
virtual void put(T item) {
    mutex.acquire();
    if (state == Full) {
        nbWaitingProd += 1;
        mutex.release();
        waitEmpty.acquire();
    }
    element = item;
    if (nbWaitingCons > 0) {
        nbWaitingCons -= 1;
        waitFull.release();
    }
    else {
        state = Full;
        mutex.release();
    }
}
```

```
    virtual T get(void) {
        T item;
        mutex.acquire();
        if (state == Empty) {
            nbWaitingCons += 1;
            mutex.release();
            waitFull.acquire();
        }
        item = element;
        if (nbWaitingProd > 0) {
            nbWaitingProd -= 1;
            waitEmpty.release();
        }
        else {
            state = Empty;
            mutex.release();
        }
        return item;
    }
};
```

Tampon de taille N

- Le tampon partagé contient N éléments
- Problèmes à résoudre:
  - Synchronisation des tâches
  - Gestion du tampon
- Le tampon est une liste circulaire
  - Un pointeur `writePointer` pour l'écriture
    - initialisé à 0
  - Un pointeur `readPointer` pour la lecture
    - initialisé à 0

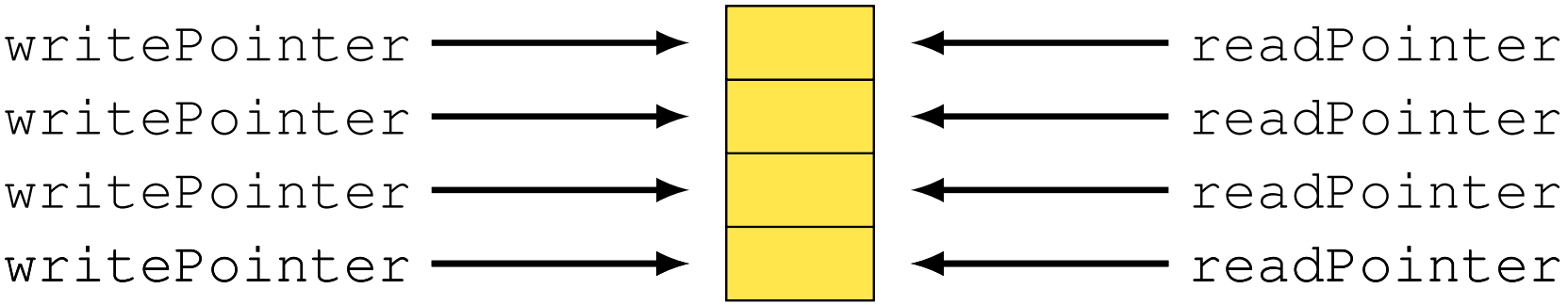
Fonctionnalité des tâches

Productrices

```
elements[writePointer] = item;
writePointer = (writePointer + 1) % BUFFER_SIZE;
```

Consommatrices

```
item = elements[readPointer];
readPointer = (readPointer + 1) % BUFFER_SIZE;
```



Question

- Si `writePointer` = `ReadPointer`, le tampon est:
  1. Vide?
  2. Plein?

Solution

- 1er algorithme: Les sémaphores devraient permettre de gérer ceci
- 2ème algorithme:
  - On introduit un compteur de cases libres `nbFree`
  - `nbFree`=0 $\Rightarrow$ tampon plein
  - `nbFree`=`BUFFER_SIZE` $\Rightarrow$ tampon vide

Tampon multiple: 1er algorithme

- Un sémaphore `waitNotFull` pour faire patienter les producteurs si le tampon est plein
- Un sémaphore `waitNotEmpty` pour faire patienter les consommateurs si le tampon est vide
- Un pointeur de lecture et un pointeur d'écriture

Tampon multiple: 1er algorithme

```
#include <QSemaphore>

template<typename T> class BufferNa : public AbstractBuffer<T> {

protected:
    T *elements;
    int writePointer;
    int readPointer;
    int bufferSize;
    QSemaphore waitNotFull, waitNotEmpty;

public:
    BufferNa(unsigned int size) : waitNotFull(size) {
        if ((elements = new T[size]) != 0) {
            writePointer = readPointer = 0;
            bufferSize = size;
            return;
        }
        throw NoInitTamponN;
    }

    virtual ~BufferNa() {}
```

Tampon multiple: 1er algorithme

```
    virtual void put(T item) {
        waitNotFull.acquire();
        elements[writePointer] = item;
        writePointer = (writePointer + 1) % bufferSize;
        waitNotEmpty.release();
    }

    virtual T get(void) {
        T item;
        waitNotEmpty.acquire();
        item = elements[readPointer];
        readPointer = (readPointer + 1) % bufferSize;
        waitNotFull.release();
        return item;
    }
};
```

- Quels sont les limitations de cette implémentation?

Tampon multiple: 1er algorithme (version correcte)

- Un sémaphore `waitNotFull` pour faire patienter les producteurs si le tampon est plein
- Un sémaphore `waitNotEmpty` pour faire patienter les consommateurs si le tampon est vide

New Un sémaphore `mutex` pour protéger les accès aux variables

- Un pointeur de lecture et un pointeur d'écriture

Tampon multiple: 1er algorithme

```
#include <QSemaphore>

template<typename T> class BufferNa : public AbstractBuffer<T> {

protected:
    T *elements;
    int writePointer;
    int readPointer;
    int bufferSize;
    QSemaphore mutex, waitNotFull, waitNotEmpty;

public:
    BufferNa(unsigned int size) : mutex(1), waitNotFull(size) {
        if ((elements = new T[bufferSize]) != 0) {
            writePointer = readPointer = 0;
            bufferSize = size;
            return;
        }
        throw NoInitTamponN;
    }

    virtual ~BufferNa() {}
```

Tampon multiple: 1er algorithme

```
    virtual void put(T item) {
        waitNotFull.acquire();
        mutex.acquire();
        elements[writePointer] = item;
        writePointer = (writePointer + 1) % bufferSize;
        waitNotEmpty.release();
        mutex.release();
    }

    virtual T get(void) {
        T item;
        waitNotEmpty.acquire();
        mutex.acquire();
        item = elements[readPointer];
        readPointer = (readPointer + 1) % bufferSize;
        waitNotFull.release();
        mutex.release();
        return item;
    }
};
```

Tampon multiple: 2ème algorithme

- Un sémaphore `waitProd` pour faire patienter les producteurs si le tampon est plein
- Une variable `nbWaitingProd` pour compter combien de producteurs sont en attente
- Un sémaphore `waitConso` pour faire patienter les consommateurs si le tampon est vide
- Une variable `nbWaitingConso` pour compter combien de consommateurs sont en attente
- Un sémaphore `mutex` pour protéger les accès aux variables
- Un pointeur de lecture, un pointeur d'écriture et un compteur d'éléments

Tampon multiple: 2ème algorithme

```
#include <QSemaphore>
template<typename T> class BufferN : public AbstractBuffer<T> {
protected:
    T *elements;
    int writePointer, readPointer, nbElements, bufferSize;
    QSemaphore mutex, waitProd, waitConso;
    unsigned nbWaitingProd, nbWaitingConso;

public:

    BufferN(unsigned int size) {
        mutex.release();
        if ((elements = new T[bufferSize]) != 0) {
            writePointer = readPointer = nbElements = 0;
            nbWaitingProd = nbWaitingConso = 0;
            bufferSize = size;
            return;
        }
        // Exception
        throw NoInitTamponN;
    }

    virtual ~BufferN() {}
```

Tampon multiple: 2ème algorithme

```
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;
    }
};
```

Comparaison

- Lequel des deux derniers algorithmes est le plus performant?

Exercice

1. Les 2 dernières opérations réalisées par les fonctions `put` et `get` de l'avant-dernier algorithme sont respectivement
   - `waitNotEmpty.release();`
   - `mutex.release();`
   - et
   - `waitNotFull.release();`
   - `mutex.release();`

   L'ordre de ces opérations peut-il être inversé?
2. Dans ce même algorithme, les tâches productrices et consommatrices se partagent un sémaphore `mutex` qui réalise l'exclusion mutuelle entre les productrices et les consommatrices. Est-il nécessaire de faire l'exclusion mutuelle entre toutes les tâches ou peut-on simplement réaliser une exclusion mutuelle entre les productrices et une autre entre les consommatrices? Autrement dit, peut-on introduire 2 sémaphores à la place de `mutex`? Et pour le dernier algorithme?

Code source

```
http://reds.heig-vd.ch/share/cours/PCO/cours/code/5-prodcons/prodcons1.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/5-prodcons/prodcons_buffer1_simple1.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/5-prodcons/prodcons_buffer1_simple2.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/5-prodcons/prodcons_bufferN_limited.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/5-prodcons/prodcons_bufferN_correct.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/5-prodcons/prodcons_bufferN1.tar.gz
```