Introduction à la programmation concurrente

Producteurs-consommateurs

Yann Thoma, Fiorenzo Gamba

Février 2021

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

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License

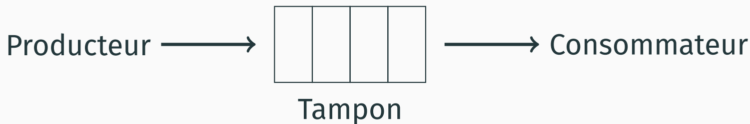
Introduction

Enoncé du problème

- Deux threads doivent se transmettre des données

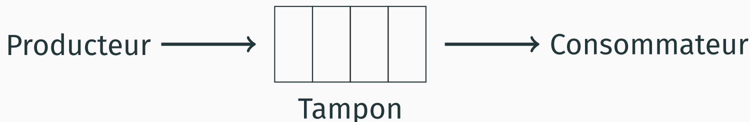
Enoncé du problème

- Deux threads doivent se transmettre des données
- Deux types de thread:
  - Un thread producteur
  - Un thread consommateur



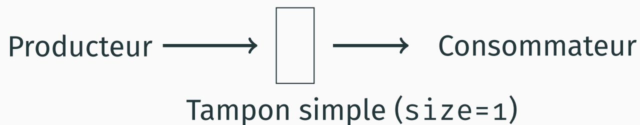
Enoncé du problème

- Deux threads doivent se transmettre des données
- Deux types de thread:
  - Un thread producteur
  - Un thread consommateur

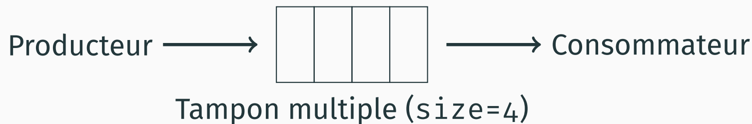


- Variations:
  - Taille du tampon:
    - Un tampon simple
    - Un tampon multiple
  - Nombre de threads:
    - 1 producteur et 1 consommateur
    - N producteurs et M consommateurs

Tampon simple



Tampon multiple



Contraintes

1. Les éléments contenus dans le tampon ne sont consommés qu'une seule fois;

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;

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 éléments du tampon, autrement dit, si le tampon est plein, une tâche productrice doit attendre la libération d'un élément du tampon.

Tâches concurrentes

- 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.

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 template 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 producerTask() {
  ITEM item;
  while (true) {
     // produire item
     buffer->put(item);
  }
}

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

int main(void) {
  buffer = new Buffer1<ITEM>();
  PcoThread prod(producerTask);
  PcoThread cons(consumerTask);
  prod.join();
  cons.join();
  return 0;
}
```

Tampon simple

Tampon simple: à base de sémaphores

- A base de sémaphores

Tampon simple: à base de sémaphores

- A base de sémaphores
- En exploitant un maximum les capacités des sémaphores

Tampon simple: à base de sémaphores

- 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

Tampon simple: à base de sémaphores

- 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

Tampon simple: à base de sémaphores

- 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` vaut 1 si le tampon est plein

Tampon simple: à base de sémaphores

- 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` vaut 1 si le tampon est plein
- Un sémaphore pour faire attendre les producteurs
  - `waitEmpty` vaut 1 si le tampon est vide

Tampon simple: à base de sémaphores

- 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` vaut 1 si le tampon est plein
- Un sémaphore pour faire attendre les producteurs
    - `waitEmpty` vaut 1 si le tampon est vide
- $\texttt{waitFull} + \texttt{waitEmpty} = 1$

Tampon simple: à base de sémaphores

```
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;
};
```

Tampon de taille N

Tampon de taille N

- Le tampon partagé contient N éléments

Tampon de taille N

- Le tampon partagé contient N éléments
- Problèmes à résoudre:
    - Synchronisation des tâches
    - Gestion du tampon

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

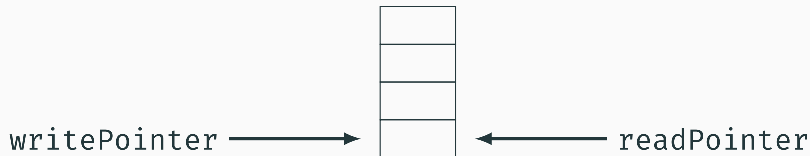
Fonctionnalité des tâches

Productrices

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

Consommatrices

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



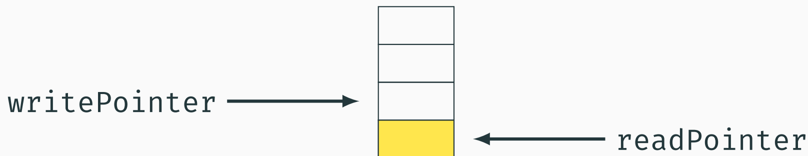
Fonctionnalité des tâches

Productrices

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

Consommatrices

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



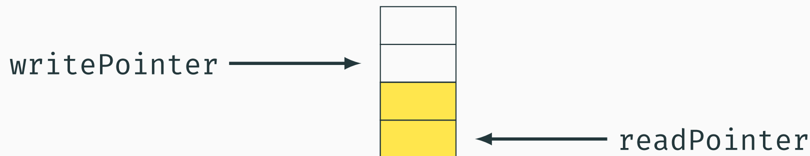
Fonctionnalité des tâches

Productrices

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

Consommatrices

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



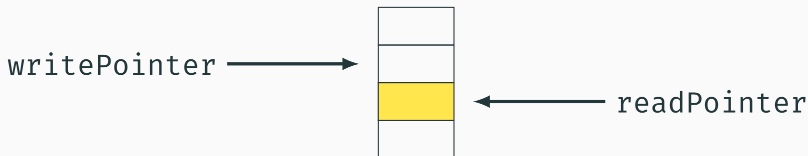
Fonctionnalité des tâches

Productrices

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

Consommatrices

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



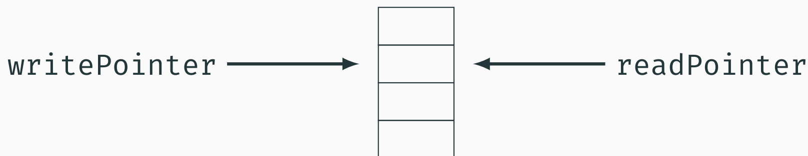
Fonctionnalité des tâches

Productrices

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

Consommatrices

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



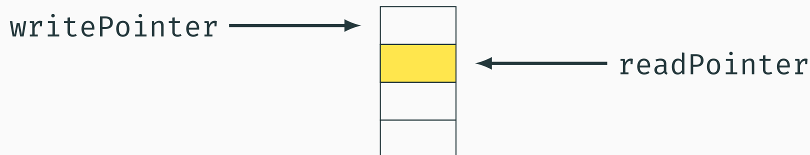
Fonctionnalité des tâches

Productrices

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

Consommatrices

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



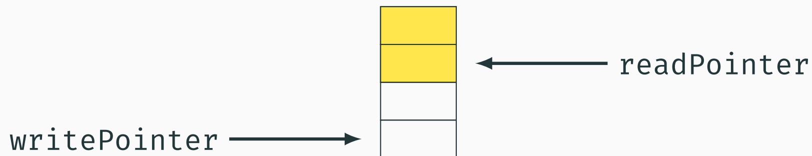
Fonctionnalité des tâches

Productrices

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

Consommatrices

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



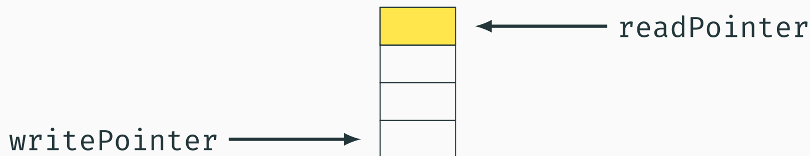
Fonctionnalité des tâches

Productrices

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

Consommatrices

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



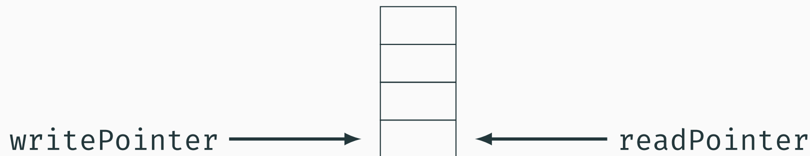
Fonctionnalité des tâches

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?

Tampon multiple: 1ère tentative

- 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: 1ère tentative

```
#include <pcosynchro/pcosemaphore>

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

protected:
    std::vector<T> elements;
    int writePointer;
    int readPointer;
    int bufferSize;
    PcoSemaphore waitNotFull, waitNotEmpty;

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

    virtual ~BufferNa() {}
```

Tampon multiple: 1er algorithme

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

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

Tampon multiple: 1er algorithme

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

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

- Quel est le problème (ou les limitations) de cette implémentation?

Tampon multiple: algorithme correct

Avec sémaphore supplémentaire pour protéger l'état!

```
#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() {}
```

Tampon multiple: algorithme correct

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

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

Tampon multiple: algorithme optimisé

Objectif: réduire le nombre d’opérations sémaphore, avec "passage de témoin"

Tampon multiple: algorithme optimisé

Objectif: réduire le nombre d’opérations sémaphore, avec "passage de témoin"

- Un sémaphore `waitProd` pour faire patienter les producteurs si le tampon est plein

Tampon multiple: algorithme optimisé

Objectif: réduire le nombre d'opérations sémaphore, avec "passage de témoin"

- Un sémaphore `waitProd` pour faire patienter les producteurs si le tampon est plein
- Une variable `nbWaitingProd` pour compter combien de producteurs sont en attente

Tampon multiple: algorithme optimisé

Objectif: réduire le nombre d'opérations sémaphore, avec "passage de témoin"

- 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

Tampon multiple: algorithme optimisé

Objectif: réduire le nombre d'opérations sémaphore, avec "passage de témoin"

- 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

Tampon multiple: algorithme optimisé

Objectif: réduire le nombre d'opérations sémaphore, avec "passage de témoin"

- 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

Tampon multiple: algorithme optimisé

Objectif: réduire le nombre d’opérations sémaphore, avec "passage de témoin"

- 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: algorithme optimisé

```cpp
#include <PcoSemaphore>
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() {}
```

Tampon multiple: algorithme optimisé

```
    void put(T item) override {
        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();
        }
    }
```

```
    T get() override {
        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

- Préférez-vous la première ou la deuxième solution?

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?

Exercice 1: inversion des instructions

Exercice 1: `mutex.release()` avant `waitX.release()`

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

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

•

Exercice 2

Dans cet 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`?
- Qu'en est-il de l'algorithme optimisé?

Code source

```
run:../code/5-prodcons/prodcons1
run:../code/5-prodcons/prodcons_buffer1_simple1
run:../code/5-prodcons/prodcons_bufferN_limited
run:../code/5-prodcons/prodcons_bufferN_correct
run:../code/5-prodcons/prodcons_bufferN1
```