

Programmation concurrente

Accès concurrent à des structures de données de taille bornée
Cas du buffer de type producteurs-consommateurs

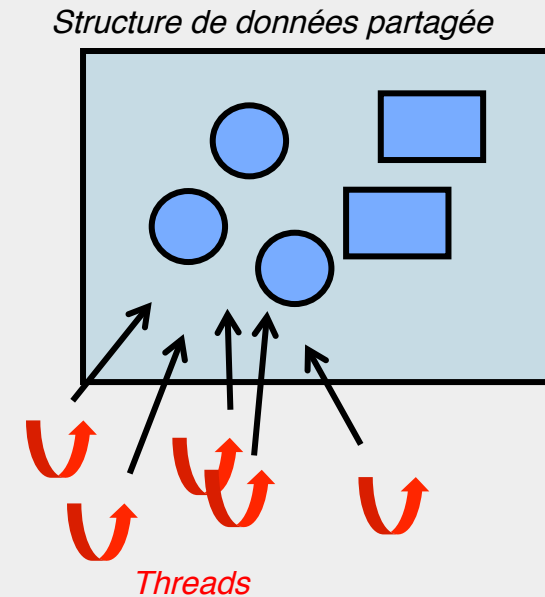
Polytech/INFO 4, 2022-2023

Fabienne Boyer
UFR IM2AG, LIG, Université Grenoble Alpes
Fabienne.Boyer@imag.fr



Où en est t'on

- **Processus**
- **Threads**
- **Notion d'exclusion mutuelle**
- **Outils**
 - ◆ Verrous
 - ◆ Moniteurs
 - ◆ Sémaphores
- **Méthodologie de programmation**
 - ◆ Solutions directes
 - ◆ Solutions FIFO
 - ◆ Solutions à base de priorités
 - ◆ Gestion des interblocages
- **Problèmes typiques étudiés**
 - ◆ Allocation de ressource
 - ◆ Rendez-vous
 - ◆ Lecteurs-Rédacteurs
 - ◆ Producteurs-Consommateurs
 - ◆ Philosophes



Partage de structures de données de taille bornée

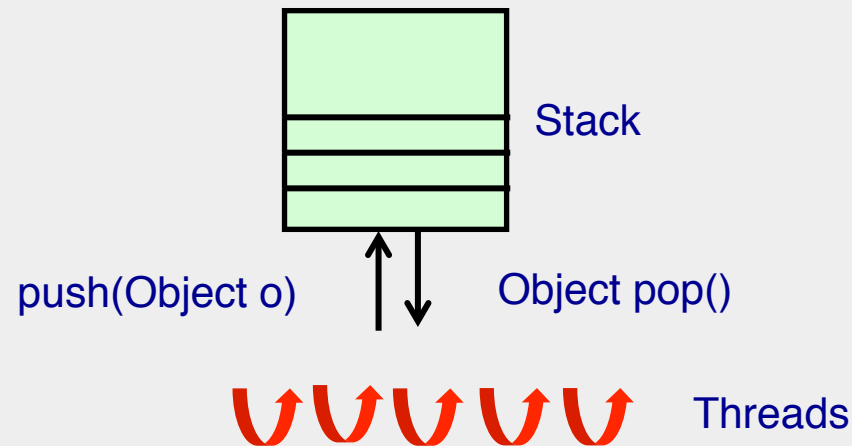
■ Exemples

- ◆ Tableau, Buffer, Pile, ..

■ Taille bornée ?

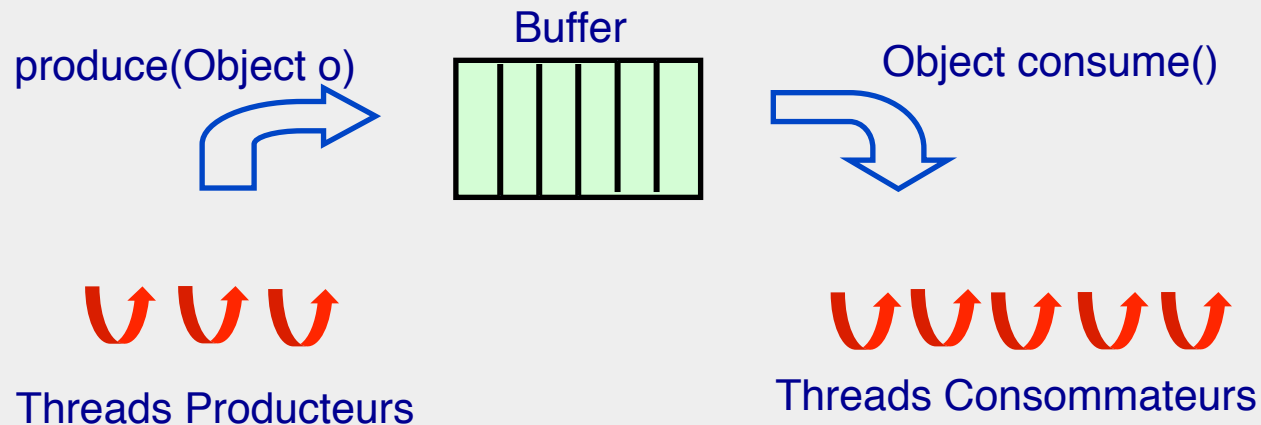
- ◆ Choix d'implémentation, borne la place mémoire utilisée
- ◆ Gestion de la saturation de place
 - ❖ Par exception
 - ❖ Par blocage : un thread qui veut déposer une donnée sera bloqué tant que la structure est pleine
- ◆ Le blocage met en place une auto-régulation : les threads qui ne peuvent pas déposer une donnée sont bloqués et ralentissent donc leur exécution

Partage d'une pile de taille bornée



- Garantir la cohérence de la pile
 - push place l'élément en tête de pile
 - pop retourne l'élément en tête de pile
- pop() bloquant la pile est vide
- push() bloquant si la pile est pleine

Partage d'un buffer Producteur-Consommateur

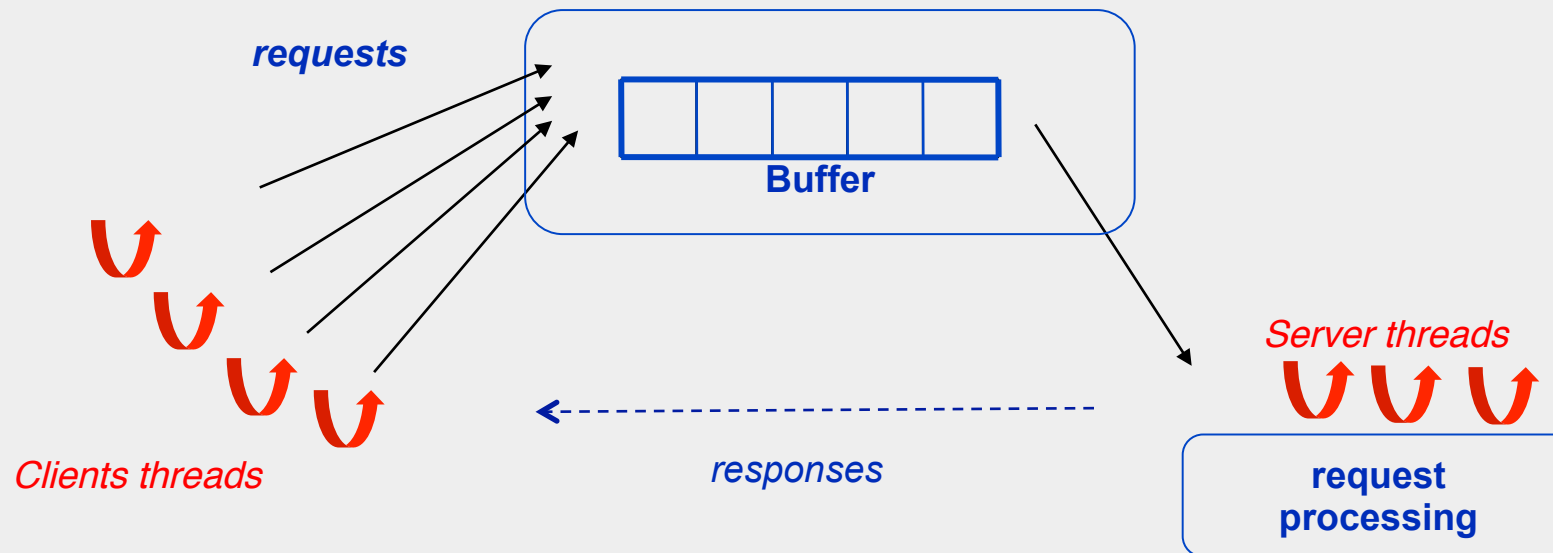


- Garantir la cohérence du buffer
 - Un élément n'est consommé qu'une fois
 - Les éléments sont consommés dans l'ordre dans lequel ils ont été produits
- produce() bloquant le buffer est plein
- consume() bloquant si le buffer est vide
- Variantes: MPMC, MPSC, SPMC, SPSC (Multiple/Single Producers, Multiple/Single Consumers)

Usage typique d'un buffer producteurs-consommateurs

■ Architectures client-serveur

- ◆ Serveurs GUI (pompes à événements, MPSC)
- ◆ Serveurs Web (MPMC)
- ◆ ..



Implémentation d'un buffer de type Producteur / Consommateur

■ Basée moniteur

- ◆ On utilise des variables (ex: *int nempty, int nfull*) pour évaluer les conditions de dépôt et de retrait

■ Basée sémaphores

- ◆ On utilise des sémaphores pour représenter les conditions de dépôt et de retrait (*Semaphore notEmpty, notFull*)
- ◆ On protège les manipulation des données partagées (*Semaphore mutex*)

Implémentation basée sémaphore d'un buffer Producteur / Consommateur

■ Hypothèses

- ◆ MPMC
- ◆ Buffer d'1 entrée

■ Données partagées

```
Msg buffer[] = new Msg[1];  
// production condition (not full entries)  
Semaphore notFull = new Semaphore(1);  
// consommation condition (not empty entries)  
Semaphore notEmpty = new Semaphore(0);
```


Implémentation basée sémaphore d'un buffer Producteur / Consommateur à 1 entrée

```
produce (Msg msg) {  
    //wait until buffer not full  
    notFull.P();  
  
    buffer[0] = msg;  
  
    // wakeup some waiting process  
    notEmpty.V();  
}
```

```
Msg Consume {  
    // wait until buffer not empty  
    notEmpty.P();  
  
    Msg msg = buffer[0];  
  
    // wakeup some waiting process  
    notFull.V();  
    return msg;  
}
```

Implémentation basée sémaphore d'un buffer Producteur /consommateur

■ Gestion d'un buffer de N cases ($N \geq 1$)

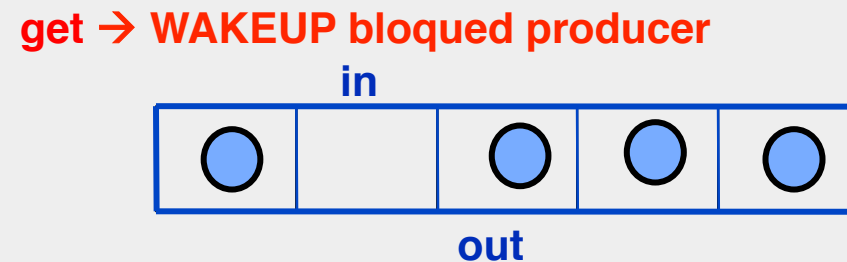
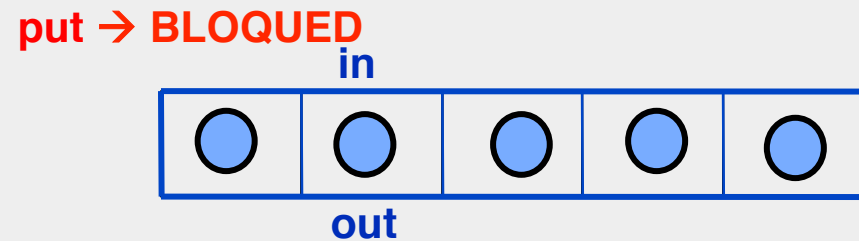
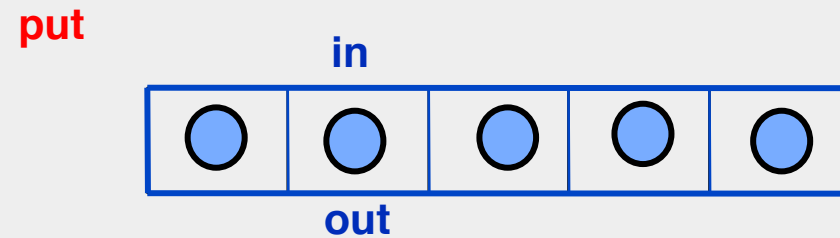
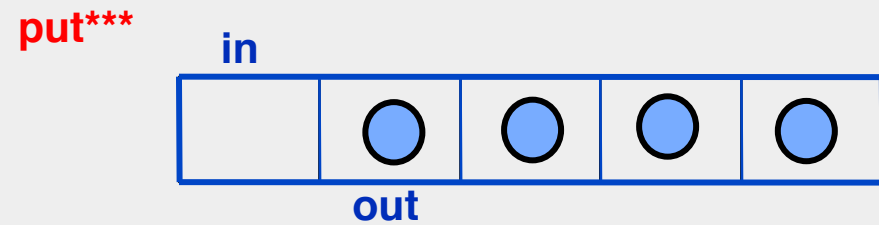
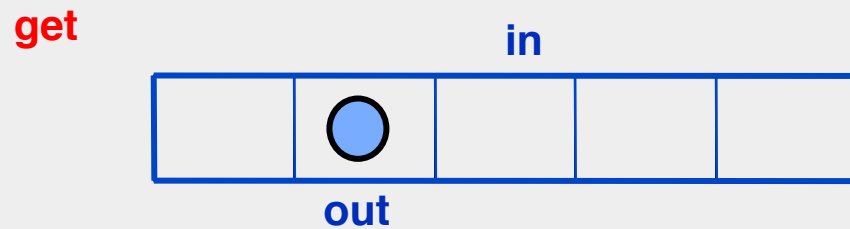
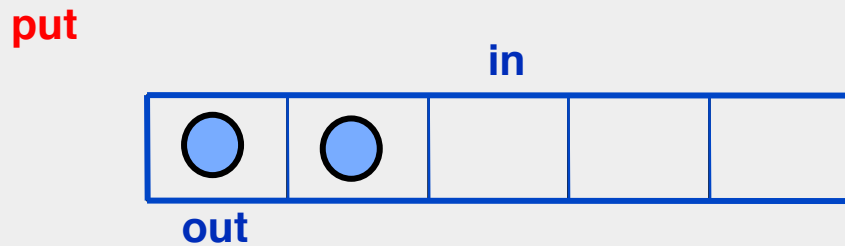
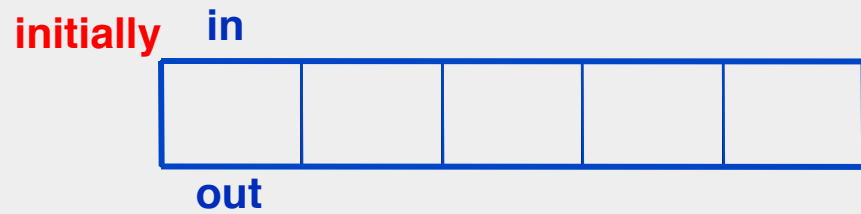
Données partagées

```
int bufferSz;  
Msg buffer[];  
Semaphore notFull;  
Semaphore notEmpty;  
Semaphore mutex;  
int in = 0, out = 0;
```

Initialisation

```
public ProdCons(int bufferSz) {  
    this.bufferSz = bufferSz;  
    buffer = new Msg[bufferSz];  
    notFull = new Semaphore(bufferSz);  
    notEmpty = new Semaphore(0);  
    mutex = new Semaphore(1);  
}
```

Scenario



Implémentation basée sémaphore d'un buffer Producteur /consommateur

```
Produce(Msg msg) {  
    //wait until buffer not full  
    notFull.P();  
    mutex.P();  
    buffer[in] = msg;  
    in = in + 1 % bufferSz;  
    mutex.V();  
    // one more not empty entry  
    notEmpty.V();  
}
```

```
Msg Consume() {  
    //wait until buffer not empty  
    notEmpty.P();  
    mutex.P();  
    Msg msg = buffer[out];  
    out = out + 1 % bufferSz;  
    mutex.V();  
    // one more not fullentry  
    notFull.V();  
}
```

Producteur / Consommateur

- **Solution correcte**
- **Mais pas optimisée (pas de parallélisme entre productions et consommations au moment du dépôt / retrait)**
 - Peut-on enlever le mutex ?
 - Attention aux opérations suivantes qui doivent rester exclusives
 - ❖ $in = in + 1 \% bufferSz$
 - ❖ $out = out + 1 \% bufferSz$

Buffer producteur/consommateur avec Java (Blocking Queues)

■ Package `java.util.concurrent`

◆ Interface `BlockingQueue`

◆ Classes

- ❖ [`ArrayBlockingQueue`](#) (basée sur l'usage d'un tableau borné)
- ❖ [`DelayQueue`](#) (élément peut être retiré au bout d'un délai)
- ❖ [`LinkedBlockingQueue`](#) (éléments ordonnés FIFO, borné)
- ❖ [`PriorityBlockingQueue`](#) (éléments ordonnés / priorité)
- ❖ [`ConcurrentLinkedQueue`](#) (file non bornée)
- ❖ [...](#)

Objectifs du mini-projet

- **Programmation de classes producteurs-consommateurs qui fournissent différentes politiques**
- **Usage des outils de base fournis par Java (moniteurs, sémaphores)**