Projet Java Producteurs/Consommateurs

Cours Applications Concurrentes, *Polytech-INFO4*

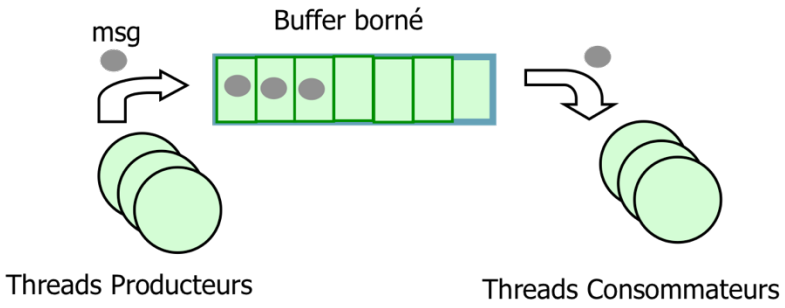
F. Boyer, Université Grenoble Alpes, année 2025-2026

Titre	Mise en œuvre d'un buffer pour producteurs / consommateurs
Organisation	Binôme
Téléchargement	Voir dans le document
Evaluation	Présentation de votre solution– analyse du code
Temps conseillé	7-10h par personne
Outils nécessaires	JDK, IDE (eclipse ou équivalent)
Date de rendu	Voir avec l'enseignant
Contenu du rendu	L'ensemble des sources (<files>.java) que vous avez réalisés, dans un paquet compressé nommé **NOM1_NOM2_PC.zip** (ou .tar)
Contact / pb	Fabienne.Boyer@imag.fr

Buffers de production-consommation

Un buffer de communication de type *producteur-consommateur* permet à des threads d'échanger des messages via un tampon borné, ce tampon étant généralement implanté avec un tableau de taille fixe (comme illustré dans la figure ci-après).
Ce type de structure de données est très utilisé dans les applications concurrentes. Notamment, les serveurs applicatifs tels que les serveurs Web multi-threadés utilisent un tel buffer : les requêtes client arrivant du réseau sont déposées dans ce buffer, elles sont ensuite consommées par des threads qui vont les traiter en parallèle.

Les threads qui déposent des messages sont appelés des *producteurs*, et ceux qui consomment les messages des *consommateurs*. Le nombre de producteurs peut être très différent du nombre de consommateurs. De manière générale, on a 4 schémas possibles : MPMC (multiple producteurs et multiples consommateurs), SPMC (un producteur et de multiples consommateurs), MPSC (de multiples producteurs et un seul consommateur), SPSC (un seul producteur et un seul consommateur). On trouve des implantations de buffers de production-consommation optimisées spécifiquement pour ces différents cas. Dans ce projet, on se place dans le cas général MPMC.



La mise en œuvre d'un buffer producteurs-consommateurs doit être à la fois performante (du point de vue tu temps d'exécution) et cohérente. Comme les threads producteurs et consommateurs s'exécutent en parallèle, la manipulation du buffer de messages est donc synchronisée. Cette synchronisation garantit les invariants suivants :

- Un message n'est consommé qu'une seule fois.
- Les messages sont consommés dans l'ordre dans lequel ils ont été produits.

Objectif général du TP

Ce TP comprend deux objectifs :

- Programmer une classe ***ProdConsBuffer*** mettant en œuvre buffer de communication de type *producteur-consommateur.*

- Programmer une application de test (**classe TestProdCons**) qui crée un ensemble de threads de type producteurs et consommateurs qui utilisent un buffer de type *ProdConsBuffer*, avec des variations dans les nombres de messages produits et dans les temps de production/consommation des messages échangés.

Attention, vous devrez respecter les spécifications que l'on vous donne au niveau des classes et interfaces, car votre programme doit être contrôlable par un automate programmé.

Vous trouverez ci-après l'interface du buffer de production-consommation à respecter initialement. Vous êtes libres de définir *Message* comme une interface ou une classe.

```
public interface IProdConsBuffer {
	/**
	* Put the message m in the buffer
	**/
	public void put(Message m) throws InterruptedException;

	/**
	* Retrieve a message from the buffer,
	* following a FIFO order (if M1 was put before M2, M1
	* is retrieved before M2)
	**
	public Message get() throws InterruptedException;

	/**
	* Returns the number of messages currently available in
	* the buffer
	**/
	public int nmsg();

	/**
	* Returns the total number of messages that have
	* been put in the buffer since its creation
	**/
	public int totmsg();
}
```

Vous aurez à réaliser différentes implémentations de l'interface *IProdConsBuffer*, ciblant différents objectifs.

Pour chaque objectif, l'ensemble des classes seront placées dans un package de nom ***prodcons.v***<#>, *# variant de 1 à k.* Attention, quelque-soit l'objectif, votre implémentation de la classe *IProdConsBuffer* doit utiliser un tableau de taille bornée (vous ne devez pas utiliser une structure de type *ArrayList* ou équivalent).

L'application de test (Classe TestProdCons)

La classe *TestProdCons* est la classe principale pour l'application de test. Cette classe crée un buffer de production-consommation, puis crée un ensemble de threads Producteurs et un ensemble de threads Consommateurs qui vont manipuler ce buffer en parallèle. Vous définirez le comportement des threads Producteur dans une classe dédiée (Producer), et de même pour les consommateurs (Consumer).

Toutes les options de configuration d'une exécution seront obtenues en utilisant la classe *Properties* de Java appliquée sur un fichier d'options au format XML stocké dans le package *prodcons*. Ce fichier se nomme par défaut *options.xml*. Vous pourrez vous inspirer du code suivant pour récupérer les options de configuration:

```
  …
  Properties properties = new Properties();
   properties.loadFromXML(
        TestProdCons.class.getClassLoader().getResourceAsStream("options.xml"));

  int nProd = Integer.parseInt(properties.getProperty("nProd"));
  int nCons = Integer.parseInt(properties.getProperty("nCons"));
  …
}
```

Le fichier d'option ayant la structure suivante :

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
      <entry key="nProd">15</entry>
      <entry key="nCons">10</entry>
      <entry key="bufSz">1</entry>
      <entry key="prodTime">10</entry>
      <entry key="consTime">10</entry>
      <entry key="minProd">5</entry>
      <entry key="maxProd">500</entry>
/properties>
```

Le paramètre *nProd* (resp. *nCons*) indique le nombre de threads producteurs (resp. consommateurs). *bufSz* indique la taille du buffer de production-consommation (on rappelle que celui-ci est implanté avec un tableau).

Chaque Consommateur devra passer son temps dans une boucle infinie dans laquelle il consomme un message et le traite. Comme nous sommes dans une application de test, nous allons simuler le traitement du message par un appel à *sleep*(). La durée moyenne de consommation est donnée dans le paramètre *consTime*.
Chaque Producteur devra produire **un nombre aléatoire** de messages, compris entre *minProd* et *maxProd*. Comme pour les consommateurs, un producteur suivra une durée moyenne pour produire un message (*prodTime*).

Tout Producteur (respectivement tout Consommateur) produit (respectivement consomme) un seul message à la fois.

Pour faciliter l'analyse de vos exécutions, essayez de placer dans un message les informations permettant de vérifier les propriétés du buffer de production-consommation. En particulier, si vous voulez conserver dans un message d'identité du thread producteur, pensez à utiliser la méthode *getId()* définie sur la classe *Thread*, qui retourne un entier identifiant le thread courant de manière unique dans la JVM.

Travail à réaliser

Objectif 1 – Solution directe

Réalisez la classe *ProdConsBuffer* qui implémente l'interface *IProdConsBuffer* en appliquant dans un premier temps le principe de la **solution directe** vue en TD (wait/notify de java). Pour cela, définissez les variables caractérisant le probléme (*nfull, nempty*), le tableau de gardes-actions, et dérivez en la solution directe.

Opération	Pre-action	Garde	Post-action
Produce(Message m)		..	..
Message Consume()		..	..

Réalisez l'application de test (classe *TestProdCons* et les classes associées – *Producer*, *Consumer*, ..). Essayez de mixer le démarrage des threads producteurs et consommateurs (en particulier, de ne pas démarrer tous les producteurs puis tous les consommateurs ou l'inverse, mais au contraire de faire en sorte que les threads démarrent aléatoirement). Vous prendrez aussi soin de garantir que les processus commutent souvent afin d'avoir une réelle concurrence.

Vous mettrez en place des tests qui permettent de s'assurer des propriétés attendues du programme.

Objectif 2 – Terminaison

Etendez la solution directe pour faire en sorte que l'application termine automatiquement lorsque ***tous les messages produits ont été consommés et traités***. Attention, pour satisfaire

cette condition de terminaison, vous devez réfléchir en termes de synchronisation entre vos threads.

Objectif 3 – Solution basée sémaphores

Réalisez une version de la classe ProdConsBuffer où vous utilisez la classe Semaphore fournie par Java pour gérer la synchronisation des producteurs et des consommateurs. Vous prendrez soin de favoriser le parallélisme au maximum entre les différents threads (producteurs et consommateurs).

Objectif 4 – Solution basée sur les Locks et Conditions de Java [optionnel]

Réalisez une version de la classe ProdConsBuffer dans laquelle vous utilisez les sections critiques conditionnelles de Java (Classe ReentrantLock et interface Condition, package *java.util.concurrent*).

Objectif 5 – Multi-consommation

Etendez le comportement de la classe *ProdConsBuffer* telle que définie à l'objectif 1, de sorte qu'un consommateur puisse retirer ***k*** **messages consécutifs** dans le buffer (*k* pouvant être supérieur à la taille du buffer). Vous étendrez l'interface *IProdConsBuffer* comme montré ci-dessous et vous mettrez en oeuvre une solution simple qui fournisse cette fonctionnalité.

```
public interface IProdConsBuffer {

	/**
	* Put the message m in the prodcons buffer
	**/
	public void put(Message m) throws InterruptedException;

	/**
	* Retrieve a message from the prodcons buffer, following a fifo order
	**/
	public Message get() throws InterruptedException;

	/**
	* Retrieve n consecutive messages from the prodcons buffer
	**/
	public Message[] get(int k) throws InterruptedException;


	…

}
```

Vous utiliserez l'application de tests pour valider cette nouvelle fonctionnalité.

Objectif 6 – Multi-exemplaires synchrone

Transformez le comportement de la classe *ProdConsBuffer* de sorte que les producteurs puissent déposer un message en *n* exemplaires, et que la production et consommation de ces exemplaires soient synchrones. Pour cela il faut respecter les règles suivantes:

- un message déposé en *n* exemplaires doit être retiré par *n* consommateurs avant de disparaître du buffer,
- **un producteur ne peut poursuivre son activité que lorsque tous les exemplaires du message ont été retirés**.
- **un consommateur ne peut poursuivre son activité que lorsque tous les exemplaires du message ont été retirés**.

Autrement dit, un producteur qui dépose un message en 3 exemplaires sera bloqué jusqu'à ce que 3 consommateurs viennent consommer le message. Les deux premiers consommateurs seront également bloqués, c'est le dernier consommateur qui vient consommer le message qui engendrera le déblocage global.

Pour fournir cette fonctionnalité, vous transformerez l'interface *IProdConsBuffer* comme suit et vous utiliserez l'application de tests pour valider cette nouvelle fonctionnalité. Si vous le souhaitez, vous pouvez étendre la classe Message pour mettre en œuvre cette fonctionnalité.

```
public interface IProdConsBuffer {


	/**
	* Put n instances of the message m in the prodcons buffer
	* The current thread is blocked until all
	* instances of the message have been consumed
	* Any consumer of m is also blocked until all the instances of
	* the message have been consumed
	**/
	public void put(Message m, int n) throws InterruptedException;


	/**
	* Retrieve a message from the prodcons buffer, following a fifo order
	**/
	public Message get() throws InterruptedException;

	…

}
```

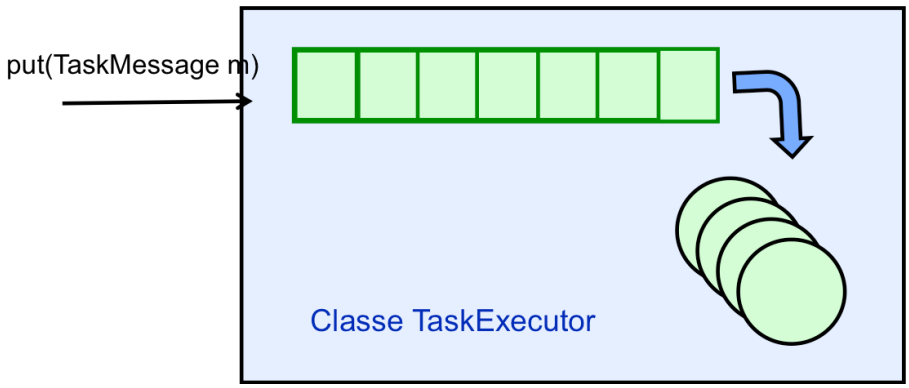
Objectif additionnel plus ouvert [optionnel]

Dans cet objectif, on s'intéresse à mettre en œuvre un buffer permettant à des threads producteurs de transmettre des *tâches à exécuter* à des threads consommateurs. Autrement dit, les threads producteurs déposent des messages qui contiennent des tâches à exécuter, ces tâches étant exécutées par des threads consommateurs. Tout thread Consommateur passe donc son temps dans une boucle dans laquelle il récupère un message depuis le buffer, puis exécute la tâche définie dans ce message.

Réalisez une première mise en oeuvre de cet objectif et vérifiez que les tâches déposées par les threads producteurs sont bien exécutées et dans le bon ordre par les threads consommateurs. Pour cette mise en oeuvre, vous pourrez spécialiser la classe ou l'interface *Message*, par exemple en faisant en sorte que celle-ci étende l'interface *Runnable* ; la méthode *run()* de cette classe correspondant à la tâche à exécuter.
.

Dans un deuxième temps, on s'intéresse à mettre en œuvre une classe *TaskExecutor* qui fournit une sémantique de plus haut niveau que la classe *ProdConsBuffer* : la classe *TaskExecutor* permet à des threads de déposer des tâches à exécuter et prend totalement en charge l'exécution de ces tâches. La différence principale avec la classe précédente est que la classe *TaskExecutor* encapsule la gestion des threads consommateurs qui exécutent les tâches déposées dans le buffer.

Une telle classe pourrait être utilisée pour l'encryption d'un répertoire. En créant une instance de la classe *TaskExecutor* puis en déposant une tâche par fichier à encrypter, on obtient automatiquement un traitement parallélisé de cette opération d'encryption.



Plusieurs versions de la classe *TaskExecutor* peuvent être envisagées mettant en oeuvre différentes politiques de gestion des threads consommateurs. Une première politique est de créer un thread consommateur pour chaque tâche déposée dans le buffer. Cette politique garantit que les tâches sont exécutées rapidement, mais peut engendrer un coût très important en termes de créations et commutations de threads.

Nous vous proposons de mettre en œuvre une autre politique dans laquelle :

- Lorsqu'une tâche est déposée, si aucun thread consommateur n'est disponible pour exécuter la tâche, un nouveau thread consommateur est créé.
- Lorsqu'un thread consommateur est inactif depuis plus de 3 secondes, il est automatiquement détruit.

Additionnellement, on s'autorisera à borner le nombre de threads consommateurs créés de sorte que ce nombre reste en dessous d'un maximum donné en argument du constructeur de la classe *TaskExecutor*.

Réalisez la mise en œuvre de la classe *TaskExecutor*, et testez cette classe au travers de simulations.