Polytech'Grenoble

RICM4, année 2018-2019

F. Boyer, N. de Palma

Projet de Programmation concurrente

Titre	Mise en œuvre buffer producteurs et consommateurs
Organisation	Binôme
Téléchargement	Voir dans le document
Evaluation	Présentation de votre solution– analyse des choix de conception et du code
Temps conseillé	7-10h par binôme
Outils nécessaires	JDK > à la version 1.7, IDE (eclipse ou équivalent)
Date de rendu	Au plus tard Mardi 18 décembre minuit
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.dfr, Noel.depalma@imag.fr

1 Rappels

On vous conseille d'utiliser un environnement de développement tel Eclipse pour conduire ce travail, cependant voici quelques rappels d'utilisation de Java.

- Compiler un fichier <X>.java : javac -classpath ProdCons.jar:$CLASSPATH <X>.java
- Exécuter le programme : java -classpath ProdCons.jar:$CLASSPATH TestProdCons
- Produire la documentation dans le directory Docs : javadoc -d Docs *.java
- Produire le fichier jar : jar cvf Simulation.jar <files>.class

2 Objectif général

Ce TP constitue une initiation à la programmation **concurrente**. Il consiste en deux objectifs :

- programmer une classe ***(classe ProdConsBuffer)*** mettant en œuvre buffer de communication de type *producteur-consommateur,* permettant à des threads d'échanger des messages via un tampon borné implanté avec un tableau (voir figure ci-après).
- 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, car votre programme doit être contrôlable par un automate programmé.

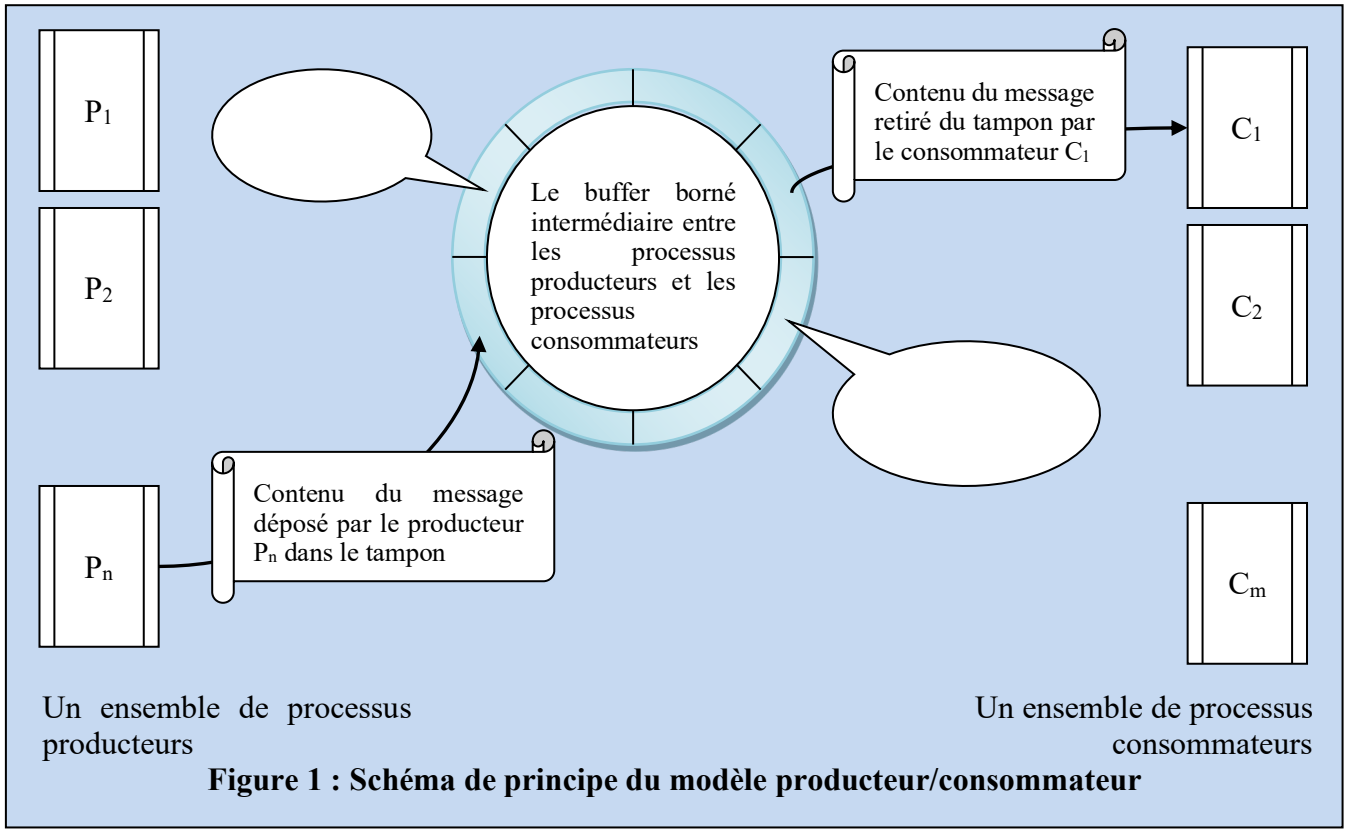

Figure 1 : Schéma de principe du modèle producteur/consommateur

2.1 Le buffer de production-consommation (classe ProdConsBuffer)

Vous trouverez ci-après l'interface du buffer de production-consommation à respecter.

```
public interface IProdConsBuffer {

	/**
	* put 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;

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

}
```

Vous aurez dans ce projet 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 **jus.poc.prodcons.v?**

Vous êtes libres de définir *Message* comme une interface ou une classe.

2.2 L'application de test (Classe TestProdCons)

La classe *TestProdCons* est la classe principale pour l'application de test. Tous les paramètres généraux d'une exécution seront obtenus en utilisant la classe *Properties* appliquée sur un fichier d'options au format xml stocké dans le package « jus.poc.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(file));

  int nbP = Integer.parseInt(properties.getProperty("nbP"));
  int nbC = Integer.parseInt(properties.getProperty("nbC"));
   …
}
```

Le fichier d'option aura 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="nbP">1</entry>
      <entry key="nbC">10</entry>
      <entry key="BufSz">1</entry>
      <entry key="ProdTime">10</entry>
      <entry key="ConsTime">10</entry>
      <entry key="Mavg">5</entry></properties>
/properties>
```

Le parametre *nbP* (resp. *nbC*) 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 Producteur devra produire **un nombre aléatoire** de messages, respectant la moyenne *Mavg*. Chaque Consommateur devra consommer des messages et les traiter. Ce traitement n'est pas précisé mais peut se réduire à l'impression du contenu du message, à une statistique de consommation ou …

Les Producteurs (respectivement les Consommateurs) produisent (respectivement consomment) un seul message à la fois et ils réalisent cette opération dans un délai qui respecte une loi probabiliste donnée par une moyenne (*ProdTime* pour les producteurs, *ConsTime* pour les concommateurs).

Concernant la terminaison de l'application, vous devez faire en sorte **que celle-ci se termine seulement 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.

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 courante.

3 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 (wait/notify de java).

Réaliser l'application de test (classe *TestProdCons* et classes associées – *Producer*, *Consumer*, ..). Vous prendrez soin 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 effectivement de façon régulière afin d'avoir une réelle concurrence.

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

Objectif 2 – solution basée sémaphores

Refaites une version où vous utilisez des sémaphores (vous utiliserez la classe fournie par Java) pour mettre en œuvre la classe *ProdConsBuffer*.

Vous utiliserez l'application de tests pour comparer les performances de cette version avec la précédente.

Objectif 3 – multi-exemplaires

On veut étendre le comportement de la classe *ProdConsBuffer* de sorte que les producteurs puissent déposer un message en N exemplaires, N étant une caractéristique spécifique au message. On souhaite respecter les règles suivantes:

- un message déposé en N exemplaires doit être retiré par N consommateurs avant de disparaître du buffer,
- le producteur du message et les consommateurs ne peuvent poursuivre leur activité que lorsque tous les exemplaires du message ont été retirés.

Vous étendrez l'interface *IProdConsBuffer* et vous utiliserez l'application de tests pour valider cette nouvelle fonctionnalité pour la classe *ProdConsBuffer*.

Objectif 4– multi-consommation (Bonus)

On souhaite étendre le comportement de la classe *ProdConsBuffer* de sorte qu'un consommateur puisse retirer *n* messages consécutifs dans le buffer (n pouvant être supérieur à la taille du buffer). Vous étendrez l'interface *IProdConsBuffer* et vous mettrez en oeuvre une solution simple qui fournisse cette fonctionnalité.

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