Système d'Exploitation
Compte rendu du projet ProdCons : Développement d'une application de concurrence basé sur le mécanisme thread de Java.

BESNIER Benjamin, MOLION Enzo

Multithreaded programming - theory and practice.

Enseignant : Fabienne BOYER
Année universitaire 2017-2018

Table des matières

Introduction

Nous tachons dans le présent compte rendu de mettre en évidence les parties intéressantes ou problématiques de notre application et du déroulement à la fois de son développement et de son test.
Celui-ci permettra au lecteur de se concentrer sur les éléments centraux du projet et d'en comprendre les subtilités et choix de programmation rapidement. Il aura également servi à ses auteurs de "journal de bord".

Tests

Nous préparons plusieurs configurations de test afin de stresser notre système de plusieurs manières afin de vérifier sa robustesse en un maximum de situations.
Ceci se fait notamment via l'écriture de différents fichiers de configurations (`.xml`) du système.
Nous établissons les jeux de test suivant :
Nous commençons par en établir un destiné aux objectifs 1, 2 et 3.

Numéro du test	Description et objectif(s)
0	Cas simple. Capacité supérieure au nombre de messages. Nombre de consommateurs supérieur au nombre de producteurs. Un message par producteur.
1	Consommation de plusieurs message par consommateur. Capacité supérieure au nombre de messages. Nombre de producteurs supérieur au nombre de consommateurs.
2	Tampon saturé avec un message par producteur. Capacité inférieure au nombre de messages. Des producteurs vont donc être endormis. Un message par producteur. Nombre de consommateurs supérieur au nombre de producteurs.
3	Tampon saturé avec plusieurs messages par producteurs. Capacité inférieure au nombre de messages. Des producteurs vont donc être endormis. Plusieurs messages par producteur. Nombre de consommateurs supérieur au nombre de producteurs.
4	Tampon saturé avec un message par producteur et peu de consommateurs. Capacité inférieure au nombre de messages. Des producteurs vont donc être endormis. Un message par producteur. Nombre de consommateurs inférieur au nombre de producteurs.

5	Tampon saturé avec plusieurs messages par producteur et peu de consommateurs. Capacité inférieure au nombre de messages. Des producteurs vont donc être endormis. Plusieurs messages par producteur. Nombre de consommateurs inférieur au nombre de producteurs.
6	Tampon saturé avec temps moyen de consommation long. Capacité inférieure au nombre de messages. Temps de consommation largement supérieur au temps de production.

Numéro	0	1	2	3	4	5	6
Nombre producteurs	2	5	7	2	7	5	5
Nombre consommateurs	5	2	15	6	2	1	5
Capacité du buffer	50	50	5	5	5	5	5
Temps moyen de production	10	10	10	10	10	10	10
Temps moyen consommation	10	10	10	10	10	10	100
Nombre moyen de production par producteur	1	3	1	4	1	6	6

: Valeur quelconque.
: La déviation associée doit être 0.

Pour l'objectif 4, les message étant en plusieurs exemplaires dans le tampon, nous étoffons le jeu de test en placant le paramètre Nombre d'exemplaire par message à une autre valeur que 1 avec déviation 0. Nous fixons ainsi 10±3 exemplaires pour ces tests supplémentaires.

Nous précisons que nous avons bien évidemment automatisé la génération de fichier de test, via un programme C nommé `testGeneration`[1].
Nous précisons également que nous avons distribué ce script à l'ensemble de la promotion.
Celui-ci s'utilise de la manière suivante :

```
$ testGeneration <nom du fichier généré> <liste de (clé, valeur) xml>
```

[1] BESNIER Benjamin GitHub *ProdCons-SE* repository, *testGeneration.c* file <https://github.com/Benjam73/ProdCons-SE/blob/master/testGeneration.c>

Par exemple, pour générer le fichier `xml` suivant :

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd" >
<properties>
	<comment>
		Cette configuration ...
	</comment>
	<entry key="nbProd">5</entry>
	<entry key="nbCons">2</entry>
	<entry key="nbBuffer">50</entry>
	<entry key="tempsMoyenProduction">10</entry>
	<entry key="deviationTempsMoyenProduction">1</entry>
	<entry key="tempsMoyenConsommation">10</entry>
	<entry key="deviationTempsMoyenConsommation">1</entry>
	<entry key="nombreMoyenDeProduction">3</entry>
	<entry key="deviationNombreMoyenDeProduction">1</entry>
	<entry key="nombreMoyenNbExemplaire">5</entry>
	<entry key="deviationNombreMoyenNbExemplaire">3</entry>
</properties>
```

On a utilisé le script comme ceci :

```
./testGeneration testX.xml nbProd 5 nbCons 2 nbBuffer 50 tempsMoyenProduction 10 deviationTempsMoyenProduction 1
tempsMoyenConsommation 10 deviationTempsMoyenConsommation 1 nombreMoyenDeProduction 3
deviationNombreMoyenDeProduction 1 nombreMoyenNbExemplaire 5 deviationNombreMoyenNbExemplaire 3
```

Nous ajoutons que nous avons inséré des impressions de valeur dans la console (`System.out.println()`) au cours de l'exécution de l'application afin de pouvoir contrôler facilement le bon chargement des options de celle-ci mais surtout de pouvoir repérer aisément des schémas (*patterns*) de comportement problématique de l'application (production et consommation systématiquement dans le même ordre, blocage simultané de tous les `Thread`, …).
Ainsi, nous affichons :
- le nombre de `Producteur` et de `Consommateur` pour cette instance de l'application, le nombre de `Message` à produire pour chacun des `Producteur`,
- à chaque appel de la méthode `put()`, le `Producteur` responsable de cet appel et la place restante dans le tampon (au moment de l'appel), son identifiant puis "waits" s'il s'endort,
- à chaque appel de la méthode `get()`, le `Consommateur` responsable de cet appel et le nombre de `Message` restants dans le tampon (au moment de l'appel), son identifiant puis "waits" s'il s'endort,
- à chaque traitement d'un `Message` par un `Consommateur`, son identifiant (caractérisé par son identifiant unique, le numéro de production par son `Producteur` et le numéro de consommation par son `Consommateur`), cela permet de s'assurer de l'ordre total puis par `Producteur` et enfin par `Consommateur`,
- "dies" lorsqu'un `Producteur` ou `Consommateur` arrive à la fin de sa tâche.

Nous permettons d'inhiber ces impressions via le passage de `true` à `false` de la variable statique `enabled` de la classe common.Debugger implémentée par nos soins[2].

Objectif 1

Réalisez les différentes classes nécessaires pour faire fonctionner le système de production/consommation décrit ci-dessus en appliquant dans un premier temps la solution directe (`wait()`/`notify()` de java).

On établit, afin de mettre en place la solution directe, le tableau de garde action de la classe ProdCons.
En supposant :
- que l'on ait implémenté les méthodes `enAttente()` et `taille()` donnant respectivement le nombre de messages disponibles dans le tampon (`ProdCons`) et la capacité maximale du tampon,
- que l'on ait un `Message` nommé `message`,
- que l'on ait un `ProdCons` doté d'un `Queue<Message>` nommé `queue`,
on obtient le tableau suivant :

Méthode	Garde	Action
`get`	`enAttente()` > 0	`message = queue.poll`
`put`	`enAttente()` < `taille()`	`queue.add(message)`

À partir de ce tableau, nous obtenons l'implémentation suivante :

```
public synchronized void put(_Producteur arg0, Message arg1) throws
Exception, InterruptedException {
	while (!(enAttente() < taille())) {
		wait();
	}
	queue.add(arg1);
	notifyAll();
}
```

[2] Stackoverflow website, *Outputting debug information in console* question, *Science_Fiction* answer <https://stackoverflow.com/a/11923310>

Il est ici nécessaire d'utiliser la méthode Java `notifyAll()`[3] et non `notify()`[4] puisqu'un ajout de `Message` provoque une situation de tampon non-vide, il est ainsi nécessaire de réveiller les `Consommateur` précédemment bloqués par manque de `Message` dans le tampon.

```
public synchronized Message get(_Consommateur arg0) throws Exception,
InterruptedException {
	while (!(enAttente() > 0)) {
		wait();
	}
	Message resultingMessage = queue.poll();
	if (resultingMessage == null) {
		throw new Exception("Couldn't poll message");
	}
	notifyAll();
	return resultingMessage;
}
```

Nous avions dans un premier temps pensé à utiliser la méthode Java `notify()` à la place de `notifyAll()`. Cependant, nous somme revenu sur cette décision puisque nous avons soulevé un cas problématique dans le cas d'une telle méthode :
Admettons que nous ayons :
- n Producteur, tous endormis (du fait d'un `wait()`, par exemple),
- 2 Consommateur c_1 et c_2 sur lesquels est appelée la méthode `get()`,
- 1 `Message` restant dans le tampon.

Alors si le c_1 récupère le dernier `Message` via la méthode get(), c_2 s'endort via la méthode `wait()`. En ce cas, si l'appel à la méthode `notify()` par c_1 réveille c_2, celui-ci n'ayant pas de `Message` à récupérer dans le tampon il s'endormira à nouveau (`wait()`), ce qui pose deux problèmes :
- tous les `Thread` sont endormis, personne ne peut donc faire appel à `notify()` ou `notifyAll()` pour réveiller quelqu'un.
- tous les `Thread` sont endormis alors que les `Producteur` ont encore des messages à placer dans le tampon.

Afin de nous assurer que "l'application ne se termine que lorsque tous les producteurs ont effectué leurs productions et que tous les messages produits ont été consommés et traités par des consommateurs", nous implémentons la méthode `run()` de la classe `Consommateur` de la manière suivante :

```
public void run() {
	while ((this.getBuffer().enAttente() != 0) || (simulator.hasProducer())) {
		int timeToConsume = randomConsumptionDuration();
```

[3] JavaSE 7 Documentation, *Object* class *notify* method subsection <https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notify()>
[4] JavaSE 7 Documentation, *Object* class *notifyAll* method subsection <https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notifyAll()>

```
                try {
                        sleep(timeToConsume);
                        newMessageConsumed();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

`simulator` étant le `TestProdCons` dans lequel a été créé le consommateur.

Afin de nous assurer que les messages étaient biens consommés dans l'ordre de leur production, nous implémentons une méthode `setMessageId()` qui "marque" le `MessageX` avec son ordre d'ajout au tampon, ce qui permet d'afficher celui-ci au moment de la récupération du `Message` par le `Consommateur`.

Tests :

Numéro test	0	1	2	3	4	5	6
Résultat	✔	✔	✔	✔	✔	✔	✔

✔ : Réussite du test (résultats escomptés)
✘ : Échec du test (Erreur ou résultats incorrects)

Objectif 2

Refaites une version où vous mettrez en place une forme optimisée à l'aide de vos propres sémaphores.

Conformément aux instructions données en travaux dirigés mardi 12 décembre, nous n'implantons pas une classe Sémaphore mais utilisons celle présente dans `java.util.concurrent.Semaphore`[5].
Nous obtenons l'implémentation suivante :

```
Semaphore fifoProducer;
Semaphore fifoConsumer;
Semaphore mutex;

// [...]

public ProdCons(Integer capacity) {
        // [...]
        fifoProducer = new Semaphore(capacity, true);
        fifoConsumer = new Semaphore(0, true);
```

[5] JavaSE 7 Documentation, *Semaphore* class
<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html>

```
        mutex = new Semaphore(1, true);
}

public Message get(_Consommateur arg0) throws Exception,
InterruptedException {
        Message resultingMessage;
        fifoConsumer.acquire();
        mutex.acquire();
        // [...]
        mutex.release();
        fifoProducer.release();
        return resultingMessage;
}

public void put(_Producteur arg0, Message arg1) throws Exception,
InterruptedException {
        fifoProducer.acquire();
        mutex.acquire();
        // [...]
        mutex.release();
        fifoConsumer.release();
}
```

L'instantiation des sémaphores requiert un paramètre supplémentaire par rapport à celle vue en cours (avec uniquement un entier donnant le nombre de ressources initialement disponibles). Il s'agit d'un booléen passé ici à true et qui impose au sémaphore un comportement *premier entré, premier sorti* (FIFO).
Nous en créons trois :
- un nommé `fifoProducer` initialisé à la capacité du tampon et permettant le blocage des `Producteur` une fois le tampon rempli. Il est acquis par un `Producteur` lors d'une dépose de `Message` et libéré par un `Consommateur` lors d'un retrait de `Message`,
- un nommé `fifoConsumer` initialisé à 0 et permettant le blocage des `Consommateur` avant qu'un `Producteur` n'en libère un par l'insertion d'un `Message` dans le tampon. Le sémaphore est acquis par un `Consommateur` lors d'un retrait de `Message` et libéré par un `Producteur` lors d'une dépose de `Message`,
- un nommé mutex et servant à éviter le vol de cycle : il est initialisé à 1 ressource disponible créant ainsi une file d'attente *un-par-un* (sas d'accès à une place) au niveau du tampon.

Tests :

Numéro test	0	1	2	3	4	5	6
Résultat	✔	✔	✔	✔	✔	✔	✔

Objectif 3

Afin contrôler votre application, nous avons réalisé deux classes `Observateur` et `Controleur`. Vous devrez, sur la base de l'objectif n°2, insérer dans votre programme, aux endroits adéquats, les appels aux différentes primitives de la classe `Observateur`.

Nous insérons les appels aux primitives "aux endroits adéquats".
Nous remarquons que nous avions déjà implanté des méthodes similaires (réalisant de l'impression en lieu et place de la vérification) afin de tester notre système.

Tests :

Numéro test	0	1	2	3	4	5	6
Résultat	✔	✔	✔	✔	✔	✔	✔

Objectif 4

On souhaite spécialiser le comportement de la Production/Consommation de sorte que les `Producteur` déposent dans le tampon un `Message` en plusieurs exemplaires.
Réalisez une nouvelle version de l'application de telle sorte qu'elle fonctionne selon ce protocole, vous ferez en sorte de le faire en réutilisant au mieux ce que vous avez déjà fait.

Afin de réaliser ceci, nous ajoutons un champ `compyNumber` étant un entier initialisé lors de l'instanciation du `MessageX`. Celui-ci donne à tout moment pour un `MessageX` son nombre d'exemplaires dans le tampon. Il est décrémenté lorsqu'un exemplaire en est retiré et il permet de retirer le `MessageX` lorsque sa valeur est évaluée à 1.

Tests :

Numéro test	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Résultat	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Objectif 5

A l'aide des classes fournies dans la bibliothè `java.util.concurrent`, reprenez l'objectif n°3 et mettez en place une solution qui implante la technique des `Condition` en prenant comme politique une priorité au processus exécutant le `signal`.

La seule classe impactée par cette évolution technique est `ProdCons`. Nous implantons l'interface `Condition`[6] par la classe `ReentrantLock`[7] via l'interface `Lock`[8].
La synchronisation se fait via les primitives `lock()` et `unlock()` sur un `Lock` baptisé `lock` et le réveil des `Producteur` et `Consommateur` via deux `Condition` sur `lock` : `bufferNotEmpty` et `bufferNotFull`.

Tests :

Numéro test	0	1	2	3	4	5	6
Résultat	✔	✔	✔	✔	✔	✔	✔

Objectif 6

Spécifiez et réalisez un mécanisme d'observation qui permette par le test de montrer le respect des propriétés du protocole pour l'objectif n°3.

Nous implémentons la classe `MyObservateur` de manière à contrôler toutes les conditions de correction du programme.
Une fois ceci fait, cette étape du projet consiste simplement à instancier `MyObservateur` et à réaliser ses appels de primitives de manière analogue à `Observateur` (voir Objectif 3).

Tests :

Numéro test	0	1	2	3	4	5	6
Résultat	✔	✔	✔	✔	✔	✔	✔

[6] JavaSE 7 Documentation, *Condition* interface
<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html>
[7] JavaSE 7 Documentation, *ReentrantLock* class
<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html>
[8] JavaSE 7 Documentation, *Lock* interface
<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html>

Conclusion

Nous incluons, afin de détailler encore plus le code réalisé une documentation établie à l'aide de Javadoc.
Nous estimons notre charge de travail pour effectuer tout ceci à environ 16h par personne[9].
De plus, ce projet nous a surtout permis de nous assurer de nos acquis, tant en terme de programmation concurrente qu'en Java pur mais également en gestion de projet (répartition des tâches, utilisation d'un logiciel de gestion de version, élaboration d'un rapport et d'un support de soutenance).

[9] Wikipedia website, *Loi de Brooks* article <https://fr.wikipedia.org/wiki/Loi_de_Brooks>