Université d'Évry
Val d'Essonne

Master Informatique / MIAGE

Systèmes et Applications Répartis (SAR)

TD 1 : Threads, Workflow et Synchronisation

Exercice 1 (Workflow et Synchronisation)

Soit un système distribué composé des trois processus concurrents suivants :

- P1 = { A ; D ; F }
- P2 = { B ; X ; E }
- P3 = { C ; G }

Nous voulons réaliser le système distribué en respectant le Workflow représenté par la figure 1.

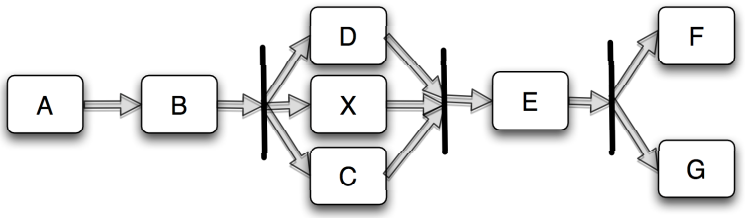

FIGURE 1 – Le workflow des trois processus

Les manières de réaliser un tel système respectant le workflow de la figure 1 sont nombreuses, nous favorisons ici la solution à base de contrôleur vue en cours. Voici le contrôleur proposé :

```
1  public class JobController {
2      boolean done=false;
3      synchronized public void jobDone(){
4          done=true;  this.notify();
5      }
6      public synchronized void isJobDone(){
7          if(!done){
8              try { wait();
9              }catch (InterruptedException e){e.printStackTrace();}
10         }
11     }
12 }
```

Question 1. Étant données trois instances de processus, respectivement de type P1, P2 et P3, de combien d'instances de contrôleurs avez-vous besoin pour réaliser le système distribué qui respecte le workflow de la figure 1 ? Expliquez.

Question 2. Combien de séquences possibles peut avoir ce système distribué ? Donnez quelques exemples de séquences.

Question 3. Donnez le code des trois classes des processus P1, P2 et P3 permettant de réaliser le système distribué respectant le workflow. Pour les actions, par exemple pour `A`, utilisez `System.out.println("A")`.

Question 4. Réaliser le système distribué (Application Java) : écrire une méthode `main` permettant de créer (instancier) et de lancer les instances des trois processus du système distribué.

Exercice 2 (Linda)

`Linda` est un langage de coordination où plusieurs processus interagissent via une mémoire partagée appelée `TupleSpace`. Un `TupleSpace` est un multi-ensemble de tuples (un ensemble où la duplication de tuples est permise). Un tuple est une liste ordonnée de valeurs, par exemple $< 25, "toto" >$. Les processus interagissent en effectuant trois actions :

- **Écriture** : notée $ECR(t)$ avec t un tuple. Cette action permet d'ajouter le tuple t dans le ***TupleSpace***.
- **Extraction** : L'action d'extraction se fait en utilisant des patrons de tuple appelés "*templates*" et notés $\widehat{t}$. Les templates sont des tuples composés de valeurs et de variables et jouent le rôle de requêtes. L'action d'extraction, $EXT(\widehat{t})$ renvoie un tuple t du `TupleSpace` qui "*match*" le template et le supprime du `TupleSpace`. Le *matching* se fait selon les règles suivantes *(i)* il faut que le tuple ait une taille égale à celle du template et *(ii)* il faut que chaque couple de valeurs partageant la même position dans le tuple et dans le template soient égaux.
- **Lecture** : L'action de lecture, $LEC(\widehat{t})$, est similaire à celle de l'extraction à la différence près que le tuple qui a *matché* le template n'est pas supprimé du `TupleSpace` par cette action.

t	$\widehat{t}$	*match*?
<"a">	$< x_1, x_2 >$	Faux
<"a">	<"a">	Vrai
<"a", 3 >	$< x_1, x_2 >$	Vrai (x_1 ="a" et $x_2 = 3$)
<"a",3>	<"b", x >	Faux
<"a",3>	<"a", x >	Vrai ($x = 3$)

TABLE 1 – Exemples de fonctionnement du *matching*

Les actions de lecture et d'extraction sont des actions bloquantes. Le processus exécutant

une telle action sur un `TupleSpace` qui ne contient aucun tuple satisfaisant le template reste bloqué jusqu'à ce qu'un tuple satisfaisant le template soit ajouté. Bien évidement, si un tuple qui vient d'être ajouté satisfait plusieurs processus bloqués sur une action de lecture ou d'extraction, alors un seul de ces processus obtiendra le tuple et les autres resteront bloqués.
On désigne par P_i les processus. Chaque processus est une suite d'actions d'écriture, d'extraction et/ou de lecture (ici désignés par des a). La grammaire des processus est la suivante :

$$P ::= a.P | \mathbf{0} \text{ où } \mathbf{0} \text{ désigne le processus terminé.}$$

On désigne par $T = \{t_1...t_n\}$ le `TupleSpace`. Un programme `Linda` L est composé d'un `TupleSpace` et d'un ensemble de processus s'exécutant en parallèle. L est noté $L = (T, P_1||...P_n)$ où $||$ représente le constructeur de parallélisme.

$$R1 = a.P \xrightarrow{a} P$$

$$R2 = \frac{P_i \xrightarrow{ECR(t)} P_i'}{(T,P_1||..||P_i||..) \xrightarrow{ECR_i(t)} (T'=T\cup\{t\},P_1||..||P_i'||..)}$$

$$R3 = \frac{P_i \xrightarrow{EXT(\hat{t})} P_i' \wedge \exists t \in T, match(t,\hat{t})=vrai}{(T,P_1||..||P_i||..) \xrightarrow{EXT_i(\hat{t})} (T'=T\setminus\{t\},P_1||..||P_i'||..)}$$

$$R4 = \frac{P_i \xrightarrow{LEC(\hat{t})} P_i' \wedge \exists t \in T, match(t,\hat{t})=vrai}{(T,P1||..||P_i||..) \xrightarrow{LEC_i(\hat{t})} (T,P1||..||P_i'||..)}$$

TABLE 2 – Sémantique opérationnelle de `Linda`

Le tableau 2 représente l'ensemble des règles de la sémantique opérationnelle d'un programme ***Linda*** :

- La règle $R1$ spécifie que si un processus est défini par une action a suivie d'un autre processus P, alors ce processus peut exécuter a et se réduire à P.
- La règle $R2$ spécifie que si l'un des processus du programme `Linda` peut écrire un tuple alors tout le programme `Linda` peut exécuter une action d'écriture indicée par le numéro du processus pour se transformer en un autre programme `Linda` où le `TupleSpace` est augmenté par le tuple écrit et où le processus qui a exécuté l'action est remplacé par sa rédution.
- La règle $R3$ spécifie qu'à chaque fois qu'un processus veut extraire un tuple satisfaisant un template et que le `TupleSpace` contient un tuple qui satisfait le template demandé alors le programme `Linda` exécute une action d'extraction indicée par le numéro du processus pour se transformer en un autre programme `Linda` où le tuple *matché* est supprimé du `Tuplespace` et le processus est remplacé par sa réduction.
- La règle $R4$ spécifie qu'à chaque fois qu'un processus veut lire un tuple satisfaisant le template et que le `TupleSpace` contient un tuple qui satisfait cette template alors le programme `Linda` exécute une action de lecture indicée par le numéro du processus pour se transformer en un autre programme `Linda` où le `TupleSpace` reste inchangé et le processus concerné est remplacé par sa réduction.

Soit le programme `Linda` $L = (\{t1\}, P_1||P_2||P_3)$ avec :

— $P_1 = EXT(\widehat{t_1}).ECR(t_3).\mathbf{0}$
— $P_2 = EXT(\widehat{t_1}).\mathbf{0}$
— $P_3 = EXT(\widehat{t_3}).ECR(t_1).\mathbf{0}$

Question 1. Dessinez le système de transition des états du programme (en appliquant les règles) en indiquant à chaque état du programme la valeur du `TupleSpace`. Notez bien que pour cette question $match(t_i, \widehat{t_j})$ est $vrai$ si $i = j$ et $faux$ sinon.

Question 2. Est-ce que L se bloque ? Caractérisez le blocage d'un programme `Linda` dans le cas général.

`Linda` sous java

On veut maintenant programmer le `TupleSpace` en java, **JTupleSpace**. Pour simplifier, les **Tuples** et **Templates** seront des classes d'objets Java de type liste (**JTuple**), ce qui nous permet d'utiliser la classe **ArrayList** pour les représenter et les méthodes *equals(Object o)*, héritée de la class **Object**, pour comparer leurs contenus. Les processus seront représentés par des **Thread**s.
Attention, le `TupleSpace` étant un objet partagé par un ensemble de processus, il est sujet à des accès concurrents.

Question 3. Indiquez les propriétés de sûreté que le fonctionnement de la classe **JTupleSpace** doit garantir.

Le fonctionnement d'un programme ***Linda*** peut être assimilé à un problème de type *producteurs-consommateurs* à la différence que les consommateurs ne peuvent pas consommer n'importe quel produit (ici tuple) mais seulement les produits qui satisfont une condition (ici le matching avec le template).

Question 4. Le listing 1 contient le code incomplet de la classe **JTupleSpace**. Écrire la classe **JTuple** et complétez le code des méthodes *ECR*, *EXT* et *LEC*.

Listing 1 – Classe à compléter.

```
1  public class JTupleSpace {
2      private List<JTuple> multiEnsemble;
3
4      public JTupleSpace(JTuple... t){
5          multiEnsemble = new ArrayList<>(Arrays.asList(t));
6      }
7
8      public void ECR(JTuple t){ // à compléter
9      }
10
11     public JTuple EXT(JTuple t){ // à compléter
12     }
```

```
13
14      public JTuple LEC(JTuple t){ // à compléter
15      }
16
17      public String toString(){
18          return multiEnsemble.toString();
19      }
20  }
```

On veut utiliser le `TupleSpace` pour programmer le problème des philosophes. Le `TupleSpace` est la table et elle contient au départ n tuples représentant les n fourchettes (chaque fourchette est représentée par un objet Integer portant le numéro de la fourchette).
Programmez la classe **Philosophe** dont le comportement serait de prendre la fourchette qui porte son numéro moins 1, ensuite de prendre la fourchette qui porte son numéro modulo n (n est la taille du problème, i.e le nombre des philosophes) avant de les libérer.

Question 5. En utilisant les classes **JTuple**, **JTupleSpace**, **Philosophe** et **Integer** écrivez une méthode *main* pour un problème de taille trois ($n = 3$).