Programmation Répartie - Le multi-tâche en Java sur architecture à mémoire partagée

T.DUFAUD

UVSQ - IUT Vélizy - Informatique

INF3 S5 - 09 / 2025

Plan

Contexte de ce cours

Contexte d'exécution : multi thread

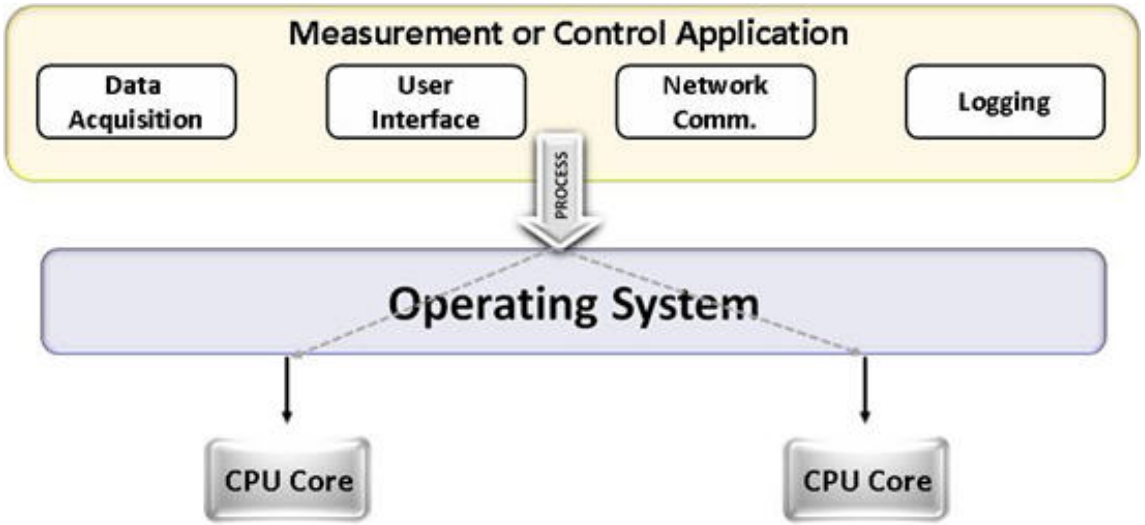


Figure: Multi thread : multi tâche au sein des applications grace au multi thread (Source : National Instrument, 2008)

Définitions

En TD

- Processus léger (ou Thread)
- Les threads partagent la même zone mémoire
- Chaque thread a son propre environnement d'éxécution et sa pile
- Il est caractérisé par son environnement (le processus), son état (actif, en attente, ...), son nom
- La classe **Thread** en Java

La classe Thread

- Depuis les premières versions de Java
- Le support des Thread est depuis la version 5 possible via l'API Concurrent
- l'API Concurrent enrichit les concepts de la classe Thread.
- On découvre d'abord les concepts avec Thread.

Les outils de base

- Création de processus
- Communication via le partage d'objet
- Synchronisation à l'aide de moniteurs

1. Implémentation de l'interface Runnable

- on souhaite définir un objet et son code a exécuter (méthode run())
- on crée un thread qui supporte cet objet

- On crée la classe Classe qui implémente l'interface Runnable
- On instancie un Thread en invoquant le constructeur Thread(Runnable target)

```
public class UnMobile extends JPanel implements Runnable
{
        public void run() { // code to run }
}
public class UneFenetre extends JFrame
{
        UneFenetre(){
                sonMobile = new UnMobile(LARG,HAUT);
                Thread laTache = new Thread (sonMobile);
                laTache.start();
        }
```

2. Héritage de la classe Thread

- on souhaite définir un objet et son code a exécuter (méthode run())
- L'abstraction de cet objet est une classe qui hérite de Thread

- On crée la classe Classe qui hérite de Thread
- On instancie un Thread en invoquant le constructeur Thread()

```
public class Task extends Thread
{
        public Task(){ super(); }

        public void run() {
                // code to run
        }

        static public void main(String argv[]){
                new Task.start();
        }
}
```

Cycle de vie d'un Thread

En exécution

- le processus est éxécuté par le(s) processeur(s)

Prêt à l'exécution

- le processus est prêt à être exécuté
  - méthode start() invoquée
  - fin du blocage

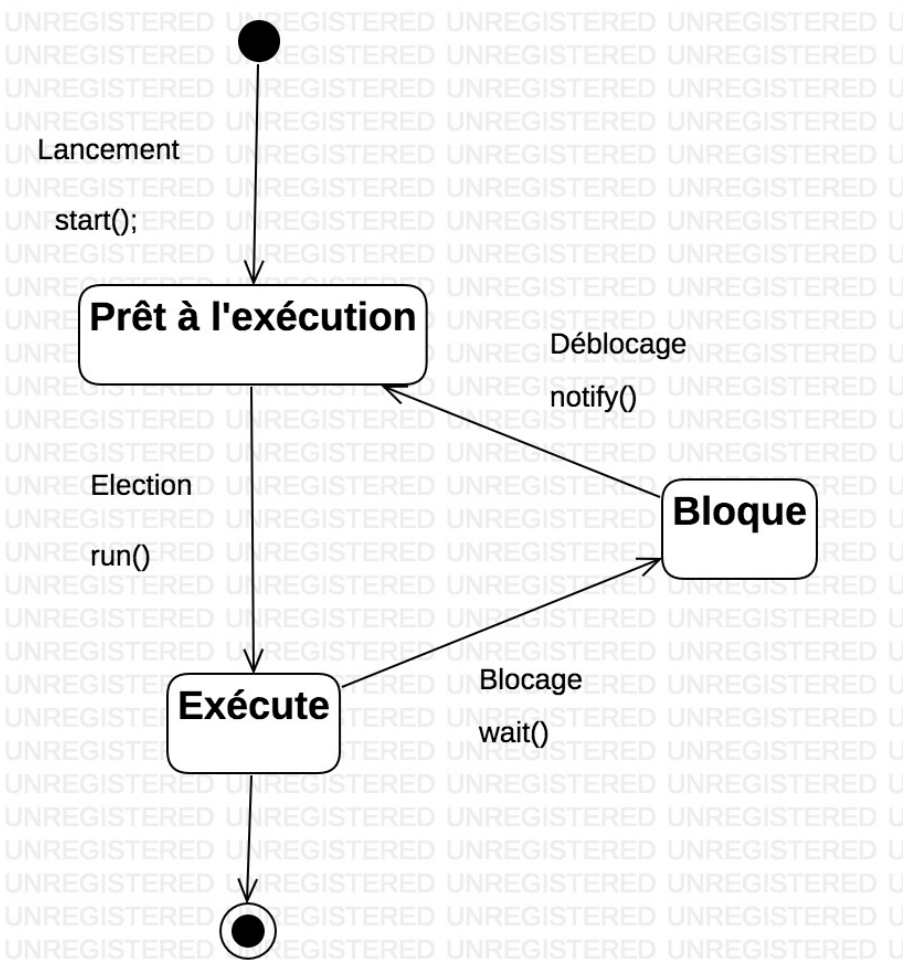
En attente

- le thread n'exécute aucun traitement et ne consomme aucune ressource CPU
- En attente d'une ressource pour être exécuté ou de la vérification d'une condition (wait())
- Attente d'un accès à une section critique (bloqué sur synchronisation (synchronized))
- Mise en sommeil (sleep())

Mort

- méthode run() terminée ou processus tué

Cycle de vie d'un Thread II



Gestion des tâches et des sections critiques

Section critique

Ressource critique

Une ressource critique est une ressource qui ne peut être utilisée que par un seul processus à la fois. Par exemple une zone mémoire, ou une imprimante.

Section critique

Portion de code dans laquelle ne s'exécute qu'un thread à la fois. Une section critique est utilisée lorsque plusieurs thread accèdent à une même ressource.

Exclusion mutuelle

Soit P1 et P2, deux processus qui accèdent à la même ressource critique RC. RC est dédiée à l'un ou l'autre pour la durée complète de l'exécution du processus. **L'accès à RC par P1 implique l'exclusion de P2 et réciproquement. P1 et P2 sont en exclusion mutuelle.**

Section critique

Exemple

Soit P1 et P2 deux processus qui produisent un fichier devant être imprimé sur une imprimante unique RC. L'accès à l'imprimante par P1 implique l'exclusion de P2. P1 et P2 sont en exclusion mutuelle. L'**imprimante** est une **ressource critique**. Le code correspondant à l'**impression** est une **section critique.**

Gestion des sections critiques

Principes

- A un instant *t* un,processus au plus peut se trouver en section critique.
- Si un processus est bloqué en dehors de la section critique, un autre processus doit pouvoir entrer en section critique.
- Si plusieurs processus sont bloqués en attente d'entrée dans une section critique et qu'aucun processus n'est en section critique, alors l'un des processus doit pouvoir y entrer au bout d'un temps fini
- la solution doit être la même pour tous

$\Rightarrow$ **Verrou MUTEX**

Le verrou MUTEX en Java

déclaration *synchronized*

- Un objet ou une méthode Java possède un verrou MUTEX
- déclenché par *synchronized*
- Une méthode ou un bloc d'instruction est alors protégé pendant l'exécution

```
synchronized (object) { // instructions }

// OR

synchronized method(arg1, arg2...){//instructions}
```

Cf: Exemple TP 2 - Affichage

Exclusion et moniteurs

Assurer qu'un objet ne subisse pas en même temps plusieurs séquences d'actions.

- Exclusion
  - Bloc synchronisé (verrou)
  - Méthode synchronisée (verrou)
  - Sémaphores (un ou plusieurs verrous)
- Moniteurs
  - Partage des données

Exclusion mutuelle avec synchronized

Bloc synchronisé

- Voir TP sur l'affichage de mots

Sémaphore

Accès à une ou plusieurs ressources

Objectif : pouvoir contrôler l'accès à une ou plusieurs ressources. Le nombre d'accès possible peut être supérieur à 1 (mais borné)

- Sémaphore binaire : ressource à accès unique
- sémaphore général ressource à accès multiple
- il comprend au moins un entier *n* qui peut caractériser le nombre de ressources disponibles et le nombre de processus en attente.

primitives pour la gestion des accès

- Wait() : permet l'accès à une ressource
- Signal() : permet la libération de la ressource

Sémaphore

Exemples

- TP Affichage - sémaphore binaire
- TP Mobile - un tiers de la fenêtre accessible à seulement p threads.

Concept de Moniteur

Un Moniteur est un objet de **synchronisation** qui permet :

- l'exclusion mutuelle entre opérations sur des données
- d'attendre qu'une condition soit validée pour permettre l'accès aux données

C'est donc une **structure de donnée avec des méthodes qui sont le point d'entrée des Threads**. Le moniteur possède des ressources et gère leur protection.

Règle

- à un instant donné 1 seul processus léger **T** peut être actif dans un moniteur
- les autres sont bloqués tant que **T** n'est pas sorti du moniteur

C'est donc une **structure de donnée avec des méthodes qui sont le point d'entrée des Threads**.

Moniteur

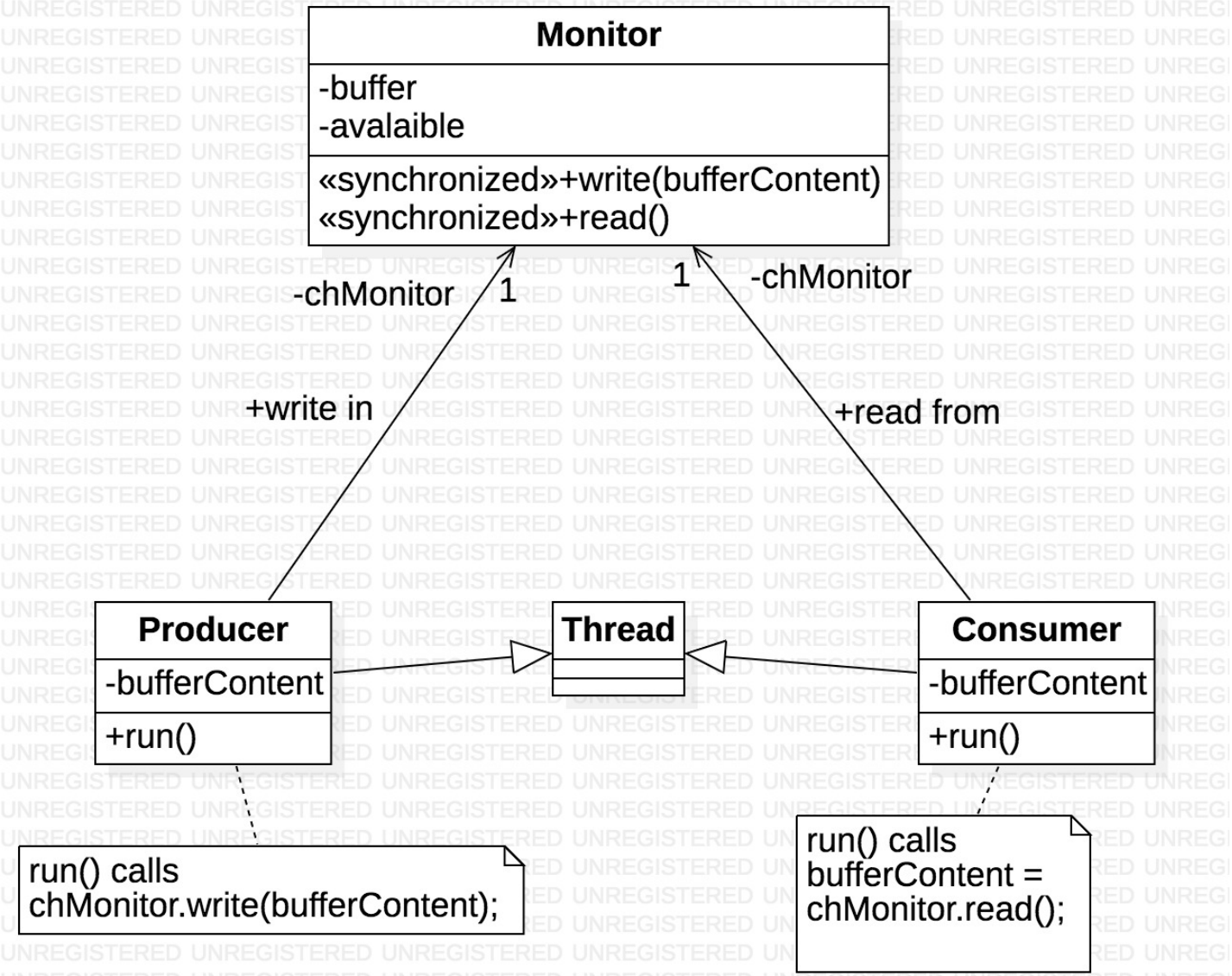
Design

- Soit une classe Monitor abstraction d'un moniteur
- Deux attributs au moins : la ressource partagée *buffer*, la variable de synchronisation *available*
- Des méthodes qui sont les procédures pour manipuler la ressource : *write* et *read*

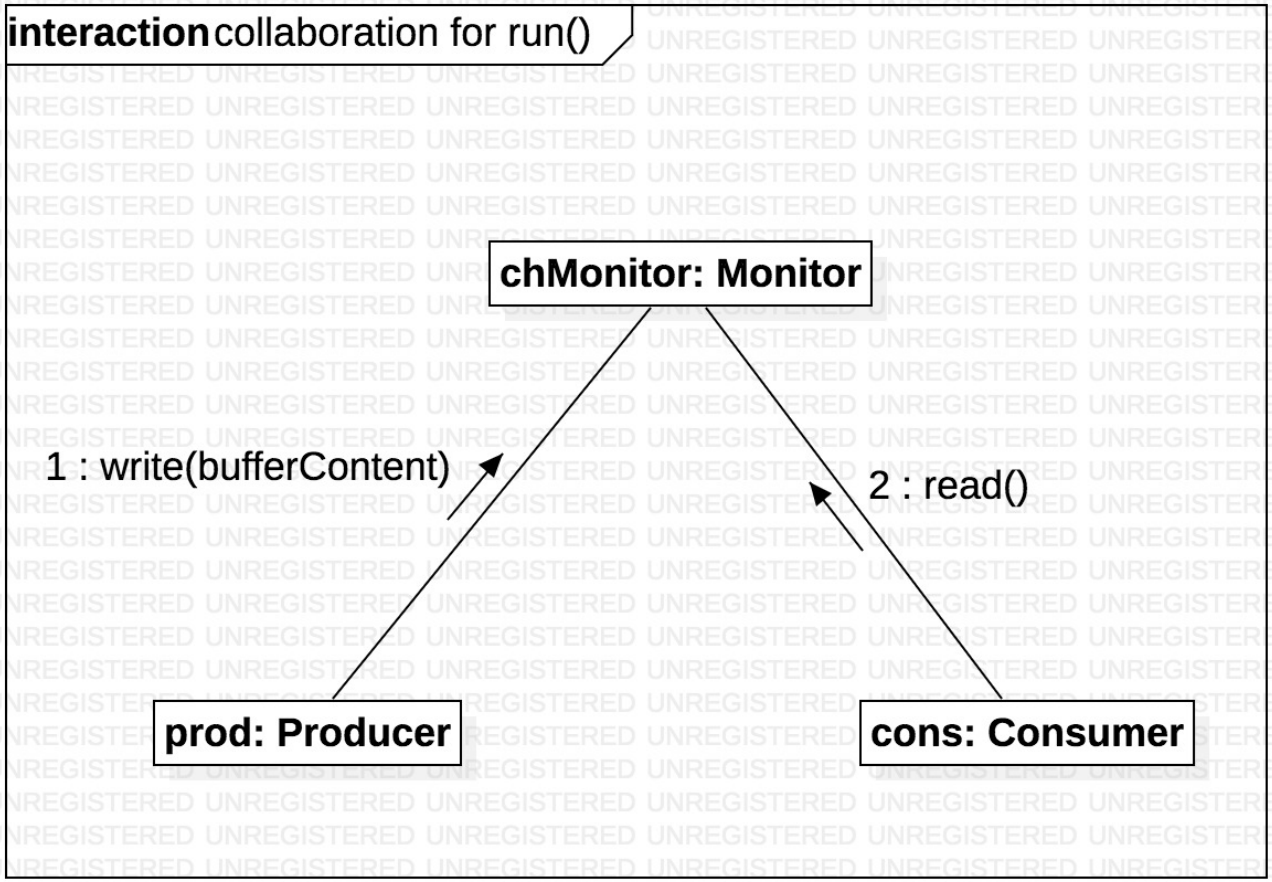
Exemple du modèle producteur/consommateur

- On crée un classe Producer abstraction d'un producteur qui écrit dans un buffer
- On crée une classe Consumer abstraction d'un consommateur qui écrit dans un buffer
- Pour gérer les accès au buffer on utilise la classe Monitor

Diagramme de classe modele Prod/Cons



Les appels des méthodes run()



TD 3 - la boîte aux lettres

- **TD3 Exercice 1**
- Adaptez le modèle proposé pour répondre à l'exercice.

API Concurrent : les files d'attentes avec concurrence d'accès

- Exemple classe BlockingQueue de l'API Concurrent
- Voir utilisation sur l'exemple de **la boulangerie** du blog de J. Paumard
- `http://blog.paumard.org/cours/java-api/chap05-concurrent-queues.html`
- **TD3 Exercice 2 :** Adaptez la boîte aux lettres en utilisant la classe BlockingQueue (plutôt que Monitor) comme dans l'exemple de la boulangerie

Les deadlocks et la "nouvelle" API (depuis Java 5...)

Deadlock

Lorsque deux thread s'attendent mutuellement on est en situation de deadlock.

- T1 dans le bloc B1 attend que le bloc B2 soit libre
- T2 dans le bloc B2 attend que le bloc B1 soit libre

$\Rightarrow$ deadlock : seule l'interruption d'un thread peut débloquer l'autre.

API Concurrent

Enrichit les concepts présentés.

- Cf site (Paumard).

$\Rightarrow$ deadlock : seule l'interruption d'un thread peut débloquer l'autre.

Références

- (Paumard) Java api chapitre 5 Programmation concurrente - `http://blog.paumard.org/cours/java-api/chap05-concurrent.html`
- (National Instrument, 2008) Différences entre le multithread et le multitâche pour les programmeurs, National Instrument, 2008, `http://www.ni.com/white-paper/6424/fr/`