Multi-Threading

Christophe Gravier, Frédérique Laforest, Julien Subercaze

Télécom Saint-Étienne
Université Jean Monnet

{prénom.nom}@univ-st-etienne.fr

FI2_INFO4 2017–2018

Outline

What you will learn today

Mandatory

- How to create and run thread
- Synchronization mechanisms
- Inter-thread communication

Optional

- Advanced synchronization

Outline

Processus & Threads

Definition : process

A process manages its own execution environment, particularly its reserved memory space. Generally process == application.

Definition : Thread

Threads or *lightweight* processes exist within a process. Threads share the memory space of their parent process.

In this course

- We talk only about threads. Processes are topics of OS course

Usage

Multi-thread applications take advantage of multi-core archictecture and are therefore potentially better performing. However, multi-thread applications are more complex to develop due to synchronization issues.

Usage

Multi-thread applications take advantage of multi-core archictecture and are therefore potentially better performing. However, multi-thread applications are more complex to develop due to synchronization issues.

When?

- Intensive computation
- I/O management
- Graphical User Interface

Usage

Multi-thread applications take advantage of multi-core archictecture and are therefore potentially better performing. However, multi-thread applications are more complex to develop due to synchronization issues.

When?

- Intensive computation
- I/O management
- Graphical User Interface

When not?

- Low computation
- Single core

A programmer has one problem

A programmer has one problem

He thought, "I know, I'll use threads."

A programmer has one problem

He thought, "I know, I'll use threads."

Now the programmer has two problems

How to create Threads

Threads in Java

One class :

- `Thread`

Two interfaces:

- `Runnable`: does not return a result (may modify some memory space)
- `Callable`: returns a result. Using `Future`

```
1  public class HelloRunnable implements Runnable {
2
3      public void run() {
4          System.out.println("Hello from a thread!");
5      }
6
7      public static void main(String args[]) {
8          (new Thread(new HelloRunnable())).start();
9      }
10
11 }
```

Exemple

```
1  package fr.tse.info4.lab10.example;
2
3  public class Example1Thread1 extends Thread {
4    int number;
5    public Example1Thread1(int number) {
6      super();
7      this.number = number;
8    }
9    @Override
10   public void run() {
11     int i = 0;
12     while (i < 5) {
13       System.out.println("Thread␣" + number);
14       i++;
15     }
16   }
17
18 }
```

Exemple - 2

```
1  package fr.tse.info4.lab10.example;
2
3  public class Eample1 {
4
5    public static void main(String[] args) {
6      Example1Thread1 thread1 = new Example1Thread1(1);
7      Example1Thread1 thread2 = new Example1Thread1(2);
8      new Thread(thread1).start();
9      new Thread(thread2).start();
10   }
11
12 }
```

Demonstration

Demonstration

Result

Different for each execution !

Demonstration

Result

Different for each execution !

Race condition - concurrency

This is the name given to this situation, where the final result depends from events and contexts that are not controlled by the program. Here the OS decides the order of execution, not our program.

Common issues with multi-threading - I

Data synchronisation

A thread modifies the state of a resource that is also modified by another thread. May lead to inconsistencies (today's first exercise). Similar issues as those encountered in database (dirty reads, ...).

Common issues with multi-threading - II

Deadlock

Two or more threads are blocked indefinitely

1. Alphonse and Gaston are very polite
2. When one greets a friend, he must bow and wait for his friend to stand back up.
3. If Alphonse and Gaston bow at the same time, they wait indefinitely for each other.

Common issues with multi-threading - II

Deadlock

Two or more threads are blocked indefinitely

1. Alphonse and Gaston are very polite
2. When one greets a friend, he must bow and wait for his friend to stand back up.
3. If Alphonse and Gaston bow at the same time, they wait indefinitely for each other.

Starvation

Situation in which a thread requires a shared resource to progress but this resource remain locked by other threads.

Common issues with multi-threading - II

Deadlock

Two or more threads are blocked indefinitely

1. Alphonse and Gaston are very polite
2. When one greets a friend, he must bow and wait for his friend to stand back up.
3. If Alphonse and Gaston bow at the same time, they wait indefinitely for each other.

Starvation

Situation in which a thread requires a shared resource to progress but this resource remain locked by other threads.

Livelock

Similar to deadlock, but the state of the application is changing

- A. et G. meet in a two lane corridor
- Each one wants to avoid the other by going the other lane. They always meet, alternatively on each lane.

Outline

Data Synchronisation

- volatile: data is always read from main memory
- Immutability: modification $\rightarrow$ new Object. String in Java.
- **Atomic Objects**

Inventory

Data Synchronisation

- volatile: data is always read from main memory
- Immutability: modification $\rightarrow$ new Object. String in Java.
- **Atomic Objects**

Synchronisation de threads

- **Join** - wait for the thread to finish
- Wait - gives up monitor and goes to sleep
- Notify - wakes the first thread that called wait()

Inventory

Data Synchronisation

- volatile: data is always read from main memory
- Immutability: modification $\rightarrow$ new Object. String in Java.
- **Atomic Objects**

Synchronisation de threads

- **Join** - wait for the thread to finish
- Wait - gives up monitor and goes to sleep
- Notify - wakes the first thread that called wait()

Code synchronization

- **synchronized**
- **Locks**

Atomic Objects

In the standard API

The `java.util.concurrent.atomic` package contains thread-safe versions of primitives.

`AtomicBoolean, AtomicLong, ...`

Methods - `AtomicLong`

- `long get()`
- `long incrementAndGet()`
- ...

join()

Wait for a thread to finish

```
1  package fr.tse.info4.lab10.example;
2  public class Example1 {
3
4    public static void main(String[] args) {
5      Thread thread1 =
6          new Thread(new Example1Thread1(1));
7      Thread thread2  =
8          new Thread(new Example1Thread1(2));
9      thread1.start();
10     thread1.join();
11     thread2.start();
12     thread2.join();
13     System.out.println("This␣is␣the␣end");
14   }
15
16 }
```

join()

Wait for a thread to finish

```
1  package fr.tse.info4.lab10.example;
2  public class Example1 {
3
4    public static void main(String[] args) {
5      Thread thread1 =
6          new Thread(new Example1Thread1(1));
7      Thread thread2  =
8          new Thread(new Example1Thread1(2));
9      thread1.start();
10     thread1.join();
11     thread2.start();
12     thread2.join();
13     System.out.println("This␣is␣the␣end");
14   }
15
16 }
```

Synchronized

`synchronized` method

Mutual exclusion. Only one thread can execute the method. Others are waiting.

Synchronized

`synchronized` method

Mutual exclusion. Only one thread can execute the method. Others are waiting.

Advantages/Drawbacks

+ Ease of use
- Performance, coarse grained synchronization

Synchronized

`synchronized` method

Mutual exclusion. Only one thread can execute the method. Others are waiting.

Advantages/Drawbacks

+ Ease of use
- Performance, coarse grained synchronization

`synchronized` block

allows to synchronize the critical part of the method. The rest of the method can be executed in parallel. Leads to better performance.

synchronized method

```
1  public class Counter{
2  private static int count = 0;
3
4   public static synchronized int getCount(){
5    return count;
6   }
7
8   public synchronized setCount(int count){
9     this.count = count;
10  }
11
12 }
```

synchronized block

```
1  public class Singleton{
2  private static volatile Singleton _instance;
3
4  public static Singleton getInstance(){
5
6      if(_instance == null){
7                  synchronized(Singleton.class){
8                    if(_instance == null)
9                    _instance = new Singleton();
10                 }
11
12     }
13     return _instance;
14
15 }
```

Locks

Locks - as in OS programming

`synchronized` uses locks behind the scene

Differents Locks

- `ReentrantLock` : Manage how thread acquire the lock. Param : `fairness`
- `ReadWriteLock` : Allow concurrent reads when not locked for writing. Usage: writer/readers situation.

The End