Session 4 : Threads, Concurrency

sourcemind

Recommended Reference

Java Tutorials: Concurrency

https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Processes

- A program is an algorithm expressed in a programming language.
- A process is a running instance of a program with all system resources allocated by the operating system to that instance of the program.
- In Unix, ps command lists running processes.
  - ✗ Unique process ID
  - ✗ Program counter
  - ✗ Executable code
  - ✗ Address space
  - ✗ System resources
  - ✗ etc.

Multitasking

- Multitasking is the ability of an operating system to execute multiple tasks (or processes) at once.
- True multitasking on a single-CPU computer is not possible.
- CPU time is divided between among all running processes.
- The switching of the CPU among processes is called a context switch.
- Save process state, stop it, load another process state, run, save, stop, ...
- A context switch is rather an expensive task.

Multiprocessing

- Multiprocessing is the ability of a computer to use more than one processor simultaneously.
- Parallel processing is the ability of a system to simultaneously execute the same task on multiple processors.
- Processes are generally not allowed to access another process address space.
  - ✗ Process 1: Work-processing application
  - ✗ Process 2: Skype
  - ✗ Process 3: Browser

Multi-threading

- What if two processes need to share memory?
  - ✗ Task 1: Play YouTube video
  - ✗ Task 2: Listen to mouse and keyboard events
- Each unit of execution (task) within a process is called a thread.
- Every process has at least one thread.
- A process can create multiple threads, if needed.
- All threads within a process share all resources including the address space.

Multi-threading

- Threads share the resources of their process.
- Each thread within a process operates independent of the other threads within the same process.
- Each thread has:
  - ✗ A program counter
  - ✗ A stack
- Context-switching among threads is less expensive
- On a multi-CPU computer, threads might run on different CPUs: True concurrency

```
Process = memory address space + resources + threads(1..N)
```

Viewing Java application threads

- For viewing Java processes, we used jps command.
- Viewing threads:
  - ✗ Download VisualVM. (**https://visualvm.github.io/index.html**)
  - ✗ Connect to a running JVM process.
  - ✗ Go to Threads tab.

sourcemind

Creating threads

- Java provides `Thread` class.
- Instantiate a thread:

```
Thread aThread = new Thread();
```

- Creating a thread object does not start it.
- You should request to start the thread. It does not guarantee when to start, it just schedules it to receive the CPU time:

```
aThread.start();
```

- At some point in time, this thread got the CPU time and started executing. What code does a thread in Java start executing when it gets the CPU time?
- A `run()` method should be provided to a thread that contains the code it should run.

Specifying a `run()` method

- There are two methods to specify a code for a thread to run:
  - ✗ Inheriting from the Thread Class

```
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Hello Java threads!");
    }
}
```

  - ✗ Implementing the Runnable interface

```
Runnable runnable = () -> {
        System.out.println("Hello Java threads!");
}

Thread aThread = new Thread(runnable);
```

Exercise 1

- Write a program that creates two threads. Each threads counts from 1 to 100 and prints them to the console.
- Each thread writes the result with a different prefix.
- In which order are the threads executed?
- Run the program multiple times.

Exercise 2

- Change the threads to ever-running tasks so that you can inspect them using VisualVM tool.
- While the program is running, start VisualVM and connect to your program.
- Find the threads and explain what you see in the Threads tab.
- Hint: You can use Thread.setName() method to give a name to the threads you create.

Daemon threads

- A Java program always has one main thread, created by JVM.
- Other threads are created by application source code.
- When will the following program stop execution ?

```
public static void main(String[] args) throws InterruptedException {

        Runnable runnable = () -> {
            while (true);
        };

        Thread aThread = new Thread(runnable);
        aThread.start();
        System.out.println("Main thread finished");
    }
```

Daemon threads

- When the main thread finishes, there might still be other unfinished threads.
- The program will not finish execution until all threads are finished execution.
- Unless, some threads are marked as daemon threads.
- A daemon thread is a thread that does not prevent the JVM from exiting when the program finishes, but the thread is still running.

```
Thread aThread = new Thread(() -> {while (true);});
aThread.setDaemon(true);
aThread.start();
```

Daemon threads `Thread.join()`

- When thread A calls the `join()` method on thread B, it causes thread A to go to waiting mode until the thread B is terminated.

```
public static void main(String[]args) {
    Thread aThread = new Thread(() -> {while (true);});
    aThread.setDaemon(true);
    aThread.start();
    aThread.join();
}
```

- Main thread will wait until aThread is terminated, even if aThread is a daemon thread.
- Another variant is Thread.join(long millis) that waits for a maximum given period of time.

sourcemind

Thread states

- A thread can be in different states:
  - ✗ `NEW`, created but not yet started
  - ✗ `RUNNABLE`, being executed right now
  - ✗ `BLOCKED`, waiting for a monitor lock
  - ✗ `WAITING`, indefinitely waiting for another thread to perform an action
  - ✗ `TIMED_WAITING`, waiting for another thread to perform an action up to a specified time
  - ✗ `TERMINATED`, exited
- Defined in `enum Thread.State`

Java Memory Model

- RAM, Heap, ThreadStack
- CPU Cache Memory
- CPU Registers
- CPU
- Local variables are stored in ThreadStack only
- Shared variables are stored in Heap

Two counter threads example

Write a program with two threads:

- Each thread counts from 1 to 1 million
- Collect the result in a shared count variable

Reader and writer threads example

Write a program with two threads:

- Start with a shared counter (object) between two threads
- Thread 1 constantly increases and decreases the counter
- Thread 2 monitors the counter: Should be zero
- Java code ThreadsExample2

Homework 4: Registeration website

In a single project, create:

- A Javalin web application that accepts HTTP requests and each request registers the user for an even
  - ✗ There are 10 open slots available at the moment. Therefore, only 10 people should be able to register.
  - ✗ Hitting the endpoint http://localhost/order must return:
    - ➢ Success; if the current request is within the first 10 requets
    - ➢ Fail: otherwise (no open slots anymore)
- A command line application that simulates registration orders
  - ✗ Model each user as a thread (implement using the HttpClient in the run() method).
  - ✗ Each thread makes registeration request.
  - ✗ Create and start more user threads than there are open slots (more than 10).

How many orders are accepted and rejected? Run the program multiple times.

Use **synchronized** mechanism, (or any other mechanism that you can find) to ensure that registered users actually get an open spot in the event.

Race Conditions

A **race condition** may happen when:

- At least two threads share an object, and
- At least two threads access the shared object in order to make an update
- Example: Repeat the following program multiple times and verify / explain the results.

```
class Counter {
    int value = 1;
    void increment() {
        value++;
    }
}

// main():

Counter c = new Counter();

Runnable task = () -> {
    c.increment();
};

for (int i = 0; i < 10; i++) {
    new Thread(task).start();
}

System.out.println(c.value);
```

sourcemind

Race Conditions

Two types:

- Read-modify-write (e.g. `counter++`)
  - ✗ A `counter++` is made of three operations
- Check-then-act (e.g. `Singleton`, `Map`)

Synchronization

- Every object in java has an **intrinsic lock** that only one thread can use at a time
- The synchronized keyword is used to use the intrinstic lock for the threads
- When entering the synchronized block, the method **acquires** the lock and when exiting the block, it **releases** the lock

```
public synchronized void safeMethod() {
   // Only one method can reach here
}
```

sourcemind

Intrinsic Lock

There are variants to the intrinsic lock and `synchronized` keyword:

- `synchronized` on method, equivalent to synchronized(this) but at the method level
- `synchronized(this)` can be used at any block-level code and not necessarily at the method level
- `synchronized(object)` in order to define an object reference and lock different **unrelated** blocks of code

Blocked threads

```
public synchronized void safeMethod() {
    while (true); // do nothing
}

Thread a = new Thread(() -> {safeMethod();});
Thread b = new Thread(() -> {safeMethod();});
a.start();
b.start();
```

- What are the states of the threads a and b?
- Which one will run?

Using different locks

- You can use any object with the synchronized keyword as a lock
- **Best practice**: Use a dedicated object as a lock

```
Object lock1 = new Object();
Object lock2 = new Object();

public void testMethod1() {
    synchronized(lock1) {
        // lock1 is acquired by a thread
    }
}

public void testMethod2() {
    synchronized(lock2) {
        // lock2 is acquired by another thread
    }
}
```

Deadlock

- Thread A executed methodA
- Thread B executed methodB

```
void methodA() {
    synchronized (lock1) {
        System.out.println("method a started");
        methodB();
        System.out.println("method a finished");
    }
}

void methodB() {
    synchronized (lock2) {
        System.out.println("method b started");
        synchronized (lock1) {
            System.out.println("processing in method b");
        }
        System.out.println("method b finished");
    }
}
```

Visibility

- Try the following program:

```java
boolean ready = false;

Thread thread1 = new Thread(() -> {
    System.out.println("Thread 1 started");
    while (!ready);
    System.out.println("Thread 1 complete");
});

Thread thread2 = new Thread(() -> {
    System.out.println("Thread 2 started");
    ready = true;
    System.out.println("Thread 2 complete");
});

thread1.start();
Thread.sleep(1000);
thread2.start();
```

Visibility

- Main memory and CPU caches
- Visibility issue

```
+-----------+-----------+
|           |           |
|   CPU 1   |   CPU 2   |
|           |           |
+-----------+-----------+
|   Cache   |   Cache   |
+-----------+-----------+
|                       |
|      Main Memory      |
|                       |
|ready = false          |
+-----------------------+
```

- Java `volatile` keyword

```
volatile boolean ready = false;
```

sourcemind

Visibility vs. Race Condition

- Let's try to fix the visibility issue above using synchronized method
- Ensuring that only one thread at a time can access the ready variable, without using volatile

```java
boolean ready = false;

Thread thread1 = new Thread(() -> {
    System.out.println("Thread 1 started");
    while (true) {
        synchronized (this) {
            if (ready) break;
        }
    }
    System.out.println("Thread 1 complete");
});

Thread thread2 = new Thread(() -> {
    System.out.println("Thread 2 started");
    synchronized (this) {
        ready = true;
    }
    System.out.println("Thread 2 complete");
});

thread1.start();
Thread.sleep(1000);
thread2.start();
```

sourcemind

Visibility vs. Race Condition

- In general, only **writer threads** need to be locked. Locking the **reader threads** slows down the program.
- `volatile` is more efficient (and the correct way) to ensure visibility
- `synchronized` should be used to deal with race conditions
- To make things worse, imagine that there is one writer thread and many reader threads
  - ✗ The writer thread blocks readers
  - ✗ A reader thread blocks the writer thread
  - ✗ **A reader thread blocks all other reader threads**