COMP124

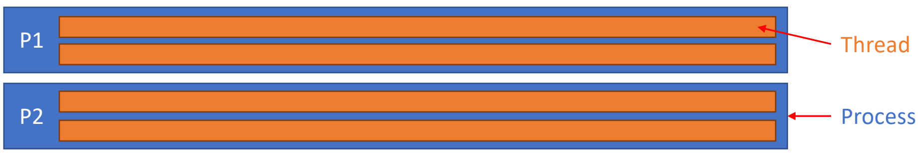
Computer Systems

21 | Threads | Concurrency | Race Conditions

Dr Stuart Thomason

Threads and Processes

- A thread is like a mini-process within another process
  - Threads are executed independently from (and in parallel with) rest of process
  - Each thread has its own instruction pointer and stack space
  - But it shares code and data (variables) with other threads in the same process



- In most systems, the kernel is not involved in context switching of individual threads
  - Handled at a higher level within the process that the threads belong to
  - Java threads are handled by the Java virtual machine (JVM) during its own time slice
  - These are known as user-level threads
  - Important mechanism for event-driven programs (especially user interfaces)

Threads

- A thread can be seen as a lightweight process within a normal heavyweight process
- It will usually serve a specific purpose that helps the main process to manage its tasks

- Consider a web browser
  - One thread for retrieving data from internet
  - Another thread to render the page within a window
  - Another thread to wait for keypresses and clicks in the user interface

- Consider a word processor
  - One thread handles the rendering of text on screen
  - Another handles keypresses and mouse clicks
  - Another performs spellchecking in the background
  - Another performs grammar checking

Thread Benefits

- Ease of programming – Easier to create one thread for each distinct activity instead of trying to do everything in one linear piece of code

- Responsiveness – A multithreaded application can continue to run even if one of the threads is blocked

- Resource sharing – Threads can share memory and resources of the process they belong to (but we need to handle concurrency properly)

- Economy – Threads are easier to create and switch between because they are allocated and managed within a process that already exists

- Parallel execution – Each thread can be physically running at the same time if the CPU has multiple cores (but concurrent programming brings other issues)

Concurrent Programming

- A single thread would perform all the steps in a calculation in a sequence (usually based on the mathematical order of precedence)

  $x = (-b + \sqrt{(b^2 - 4ac)}) / 2a$

- In a multi-threaded (or multi-core) system this can be split into two types of operation
  - Concurrent operations can be executed in parallel (independently)
  - Serial operations rely on the results of earlier operations

Concurrent	Serial
t1 = −b	t5 = t3 * c
t2 = b * b	t5 = t2 − t5
t3 = 4 * a	t5 = √t5
t4 = 2 * a	t5 = t1 + t5
	x = t5 / t4

- Processing time is reduced from 9 units to 6 units (however long an operation takes)
- A good compiler will identify concurrent instructions and assign to different CPU cores

Java Threads

- Each Java program runs in its own heavyweight process
  - The main program code runs in one lightweight thread
  - Java virtual machine (JVM) starts other threads for garbage collection, event handling, screen rendering, and so on

- Programmer can spawn new threads within their code by extending the Thread class

```
public class Thread extends Object implements Runnable
```

- The Runnable interface defines a run() method that the programmer must implement
  - This code does not start executing until the main program requests it
  - Create an instance of the thread and then call its start() method
- Threads are managed internally by the JVM
  - When JVM gets time on the CPU it decides which thread to run (including its own)
  - So it has its own internal thread scheduling algorithm (similar to round robin)

Java Thread Example

- Threads don’t start running until explicitly instructed to

```
class MyWorker extends Thread {
    public void run() {
        System.out.println(“I am a worker thread”);
    }
}

public class MyMain {
    public static void main (String[] args) {
        MyWorker runner = new MyWorker();
        runner.start();
        System.out.println(“I am the main thread”);
    }
}
```

- Think about which order the output will appear in – which line is printed first?

Thread Start

- The MyWorker class extends the Thread class
  - So the programmer must implement the Runnable interface
  - Define a run() method and put the code for the thread inside it

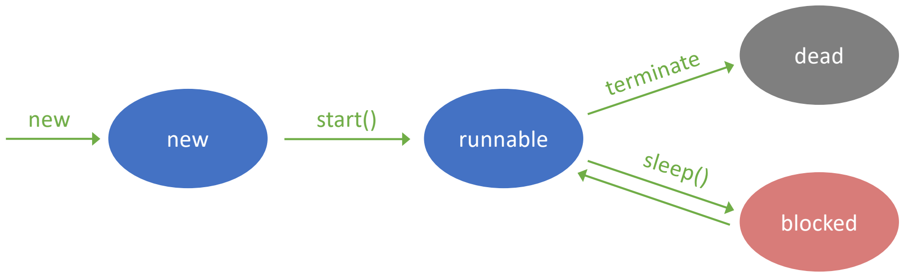
- In the main program…
  - Create a new object of type MyWorker
  - Call its start() method

- Within the JVM…
  - Memory is allocated for the new thread object
  - Its run() method is called and begins execution

- Now both the main program thread and the child thread are running in parallel

Java Thread States

- Each thread begins in the new state when its object is allocated memory
  - It moves to the runnable state when its start() method is called
    (but it might not run immediately depending on the JVM thread scheduler)
  - It moves to the blocked state when its sleep() method is called, or when it's waiting for I/O, or if it's waiting for another thread to finish
  - It moves to the dead state when it has terminated and is waiting for the JVM to clean up its memory via garbage collection



Synchronisation Problems

- Consider this thread object with a constructor that sets two internal character variables

```
class TwoChar extends Thread {
    private char out1, out2;

    public TwoChar(char first, char second) {
        out1 = first;
        out2 = second;
    }

    public void run() {
        System.out.print(out1);
        System.out.print(out2);
    }
}
```

- When the thread runs, it will output each character one after the other

Synchronisation Problems

- We can create and start multiple threads

```
public class ThreadExample {
    public static void main(String[] args) {
        TwoChar tc1 = new TwoChar('A', 'B');
        TwoChar tc2 = new TwoChar('1', '2');

        tc1.start();
        tc2.start();
    }
}
```

- We will get different results when we run this, depending on JVM thread scheduler

  AB12 A1B2 12AB 1A2B 1AB2 A12B

- We would probably see AB12 most often
  - The order that they appear in the program code
  - But there is no guarantee about the execution order of threads

Shared Variables

- Suppose we have an object that can be shared by multiple threads

```
class Something {
    private int num = 0;
    public void increase() {
        num = num + 1;
    }
}
```

- This object is passed into the constructor of two threads

```
Something thing = new Something();
MyThread t1 = new MyThread(thing);
MyThread t2 = new MyThread(thing);
```

- At some point during the program execution, both threads access the shared object

```
thing.increase();
```

- We cannot guarantee that both increases will happen (due to thread scheduling)

Race Conditions

- Consider what happens at the register (assembly) level when the increase happens

```
mov eax, num
inc eax
mov num, eax
```

- Works perfectly fine for most of the time, but occasionally the instructions are executed in a way that causes a problem
- If the starting value is 0, we would expect it to be 2 after both threads have executed

T1	T2	NUM
MOV (eax = 0)		0
INC (eax = 1)		0
	MOV (eax = 0)	0
	INC (eax = 1)	0
	MOV (eax = 1)	1
MOV (eax = 1)		1

- This is known as a race condition and is very hard to debug or even notice it happened

Critical Regions

- A race condition could happen whenever a variable is shared between multiple threads
  - Most of the time it won't happen because the threads are not interrupted
  - But we can't ignore the problem because sometimes it <u>will</u> happen

- Whenever a process or thread is accessing a shared resource, its code is said to be in a critical region (or critical section)

- We must ensure only one thread (or process) can access the resource at any one time
  - The programmer must somehow 'lock' the resource so it gets exclusive access
  - Other threads must wait for the resource to be unlocked

Semaphores

- The concept of semaphores (or tokens) comes from single track railways in the 1800s
  - A train can only enter a single track if the driver picks up a physical token
  - Only one token exists, so only one train can be on the track
  - Driver leaves token at the other end when they have passed the single track

- Programming semaphores were proposed by Dijkstra in the 1960s
  - Critical region of code is like a single track railway
  - Thread must acquire the semaphore before it can enter the region
  - Other threads must wait (block) until the semaphore becomes available

- Use of semaphores requires careful programming
  - Must consider how each shared resource will be used by each thread
  - Critical regions must be kept as small as possible to reduce thread blocking