Lecture 19 — Concurrency and Threads

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- See Chapter 11 of the textbook
- `Thread` and `Runnable`
- `synchronized`, `wait()`, and `notify()`
- Race conditions and deadlocks

Multi Processing

- A core feature of operating systems is that they allow multiprocessing
- You can run a Java program and your text editor at the same time
- You can watch a lecture while browsing the internet (but you shouldn't)
- Each program can run as its own *process*
- If there is only a single core, the operating system will schedule these processes to make it appear as though they are concurrent
- If there are multiple cores, the operating system will distribute the processes across them
- A simple way to write concurrent Java is to run several programs at once (java X, java Y, ...)
- However, this is limiting. It is hard to make those programs talk to each other

Multi Threading

- A single process can have multiple threads of execution
- This is useful in several types of situations
- Imagine we are writing code that reads a large file. It would be bad if we had to halt the entire program to wait for the file to load
- Instead, we can have a thread that reads the file while another updates the GUI, or does something else
- Modern computers have lots of cores (the desktop I am writing this on has 16)
- We will want to write code that uses all these cores to maximize efficiency

Java Multi Threading

- Java supports multi-threaded code
- This is via the `Thread` class
- Each instance represents a new thread of execution
- If we create multiple threads, we can have multiple parts of our code executed simultaneously
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html`
- Typically we will want to extend this class and override the `run()` method

ThreadExample

```
class MyThread extends Thread {
    private int number;

    public MyThread(int number) {
        this.number = number;
    }

    @Override
    public void run() {
        System.out.println("MyThread (" + number + ") running");
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            MyThread t = new MyThread(i);
            t.start();
        }
    }
}
```

- Once we call `start()` the `run()` method executes concurrently in a new thread
- Run this program and notice how the threads are not executed in order

ThreadExample

```
class MyRunnable implements Runnable {
    private int number;

    public MyRunnable(int number) {
        this.number = number;
    }

    @Override
    public void run() {
        System.out.println("MyThread (" + number + ") running");
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            Thread t = new Thread(new MyRunnable(i));
            t.start();
        }
    }
}
```

- The `Runnable` interface can also be used
- `Thread` comes with a constructor taking a `Runnable`
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Runnable.html`

Java Multi Threading

- Once we start a thread, how do we wait until it has finished?
- This may happen if you depend on the result
- For example, if you create two threads to read from 2 different databases you may wait for a result from both to continue with the program
- `Thread` has a `join()` method
- When the thread finishes executing `run()`, it will join with the calling thread
- Calling `join()` waits for that to happen

JoinExample

```
public class JoinExample {
    public static void main(String[] args) {
        Thread[] threads = new Thread[10];
        for (int i = 0; i < 10; i++) {
            threads[i] = new MyThread(i);
            threads[i].start();
        }
        for (int i = 0; i < 10; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {}
        }
        System.out.println("All threads finished");
    }
}
```

- Notice that `join()` throws a checked exception
- This occurs when a thread's execution ends in an unexpected fashion

Java Multi Threading

- A *race condition* occurs when two threads try to do the same thing concurrently
- The simplest example is modification of some variable
- If two threads are trying to increment an integer by 1 at the same time, the result may not be the same as it being incremented twice
- Thread 1 reads the integer, computes its value plus 1, then saves the integer
- Thread 2 does the same
- The reads may happen at exactly the same time, so we may save the same result twice!

SpecialInt

```
class SpecialInt {
    private int value;

    public SpecialInt(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public void increment() {
        value++;
    }
}
```

- This class lets us pass around an `int` by reference

SpecialInt

```
class AddThread extends Thread {
    private SpecialInt specialInt;

    public AddThread(SpecialInt specialInt) {
        this.specialInt = specialInt;
    }

    public void run() {
        for (int i = 0; i < 1000000; i++) {
            specialInt.increment();
        }
    }
}
```

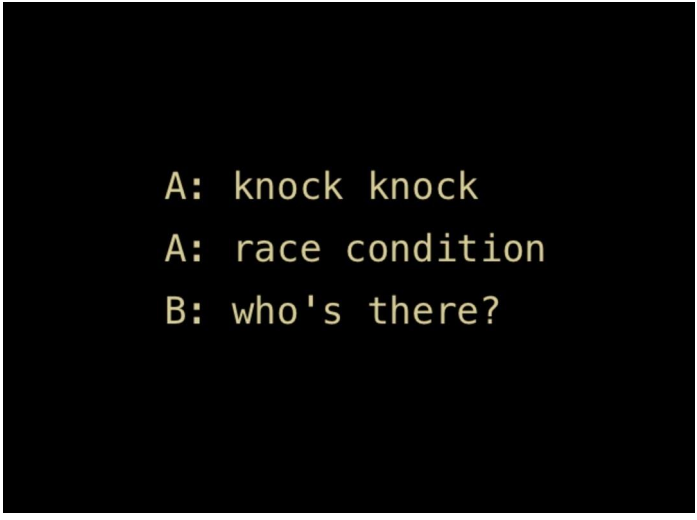
- This thread increments an integer many times

SpecialInt

```
public class RaceCondition {
    public static void main(String[] args) {
        SpecialInt specialInt = new SpecialInt(0);
        Thread t1 = new AddThread(specialInt);
        Thread t2 = new AddThread(specialInt);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e){}
        System.out.println(specialInt.getValue());
    }
}
```

- This program leads to lots of race conditions
- Run the program and see that the output is not 2000000

Mid-lecture Break



synchronized statements

```
synchronized (object) {
    // ... code ...
}
```

- Java offers the `synchronized` statement
- When a thread enters the block, it attempts to get a lock on the object
- The thread will wait until the object is available if it is locked
- First, it locks the object
- Then, it executes the block
- Then, the object is unlocked

SyncStatement

```
class AddThread extends Thread {
    private SpecialInt specialInt;

    public AddThread(SpecialInt specialInt) {
        this.specialInt = specialInt;
    }

    public void run() {
        for (int i = 0; i < 1000000; i++) {
            synchronized (specialInt) {
                specialInt.increment();
            }
        }
    }
}
```

- We can modify `AddThread` to avoid the race condition
- If we run `SyncStatement.java`, it now outputs 2000000

synchronized statement

```
class SpecialInt {
    private int value;

    public SpecialInt(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public synchronized void increment() {
        value++;
    }
}
```

- This would also work
- A `synchronized` method locks its object
- Equivalent to `synchronized (this) { ... }`

Deadlock

- Imagine a situation where there are two threads A and B
- A has a lock on object o_1 and wants to get a lock on object o_2
- B has a lock on object o_2 and wants to get a lock on object o_1
- They will be waiting forever!
- This is called a *deadlock*
- It's like when two people are so polite that they're both waiting for the other one to go first
- Let's see an example

AddBothThread

```
class AddBothThread extends Thread {
    private SpecialInt specialInt1, specialInt2;

    public AddBothThread(SpecialInt specialInt1, SpecialInt specialInt2) {
        this.specialInt1 = specialInt1;
        this.specialInt2 = specialInt2;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println("Before lock 1 " + this.getId());
            synchronized (specialInt1) {
                System.out.println("Before lock 2 " + this.getId());
                synchronized (specialInt2) {
                    specialInt1.increment();
                    specialInt2.increment();
                }
                System.out.println("After lock 2 " + this.getId());
            }
            System.out.println("After lock 1 " + this.getId());
        }
    }

}
```

- This thread increments two `SpecialInts` 100 times each

AddBothThread

```
public class Deadlock {
    public static void main(String[] args) {
        SpecialInt specialInt1 = new SpecialInt(0);
        SpecialInt specialInt2 = new SpecialInt(0);
        Thread t1 = new AddBothThread(specialInt1, specialInt2);
        Thread t2 = new AddBothThread(specialInt2, specialInt1);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e){}
        System.out.println(specialInt1.getValue());
        System.out.println(specialInt2.getValue());
    }
}
```

- There is a deadlock caused by this code
- Can you see why?

AddBothThread

```
public class Deadlock {
    public static void main(String[] args) {
        SpecialInt specialInt1 = new SpecialInt(0);
        SpecialInt specialInt2 = new SpecialInt(0);
        Thread t1 = new AddBothThread(specialInt1, specialInt2);
        Thread t2 = new AddBothThread(specialInt2, specialInt1); // Oops!
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e){}
        System.out.println(specialInt1.getValue());
        System.out.println(specialInt2.getValue());
    }
}
```

- `t1` gets a lock on 1, `t2` gets a lock on 2
- `t1` tries to get a lock on 2, `t2` tries to get a lock on 1
- We can fix the deadlock by making this change:
  `AddBothThread(specialInt1, specialInt2)`

Inter-thread Communication

- In addition to synchronization, Java has inter-thread communication
- Suppose we have a thread that needs to wait until an object is updated before it can continue
- It needs to release a lock so that the object can be updated
- This is where `wait()` and `notify()` come in
- These are methods of `Object` and be used for this kind of inter-thread communication

The `wait()` method

- The `wait()` method is used to put the current thread into a "waiting" state
- It must be called from a synchronized context (i.e., from inside a `synchronized` block or method)
- Once `wait()` is called, the thread releases the lock on the object and waits until another thread calls `notify()` on the object

The `notify()` and `notifyAll()` methods

- `notify()` wakes up a single thread that is waiting on the object's lock
- If multiple threads are waiting, one is picked arbitrarily
- `notifyAll()` wakes up all threads that are waiting on the object's lock (the highest priority one will get the lock first)
- Like `wait()`, these methods must also be called from a synchronized context
- Let's see an example

ProduceConsume

```
class ProduceConsume {
    private String sharedResource;

    public synchronized void produce(String value) throws InterruptedException {
        while (sharedResource != null) {
            wait();  // wait for the consumer to consume the resource
        }
        sharedResource = value;
        System.out.println("Produced: " + value);
        notify();  // notify the consumer that the resource is ready
    }

    public synchronized void consume() throws InterruptedException {
        while (sharedResource == null) {
            wait();  // wait for the producer to produce the resource
        }
        System.out.println("Consumed: " + sharedResource);
        sharedResource = null;
        notify();  // notify the producer that the resource has been consumed
    }
}
```

Produce

```
class Produce extends Thread {
    private ProduceConsume produceConsume;

    public Produce(ProduceConsume produceConsume) {
        this.produceConsume = produceConsume;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                produceConsume.produce("value " + i);
            } catch (InterruptedException e) {}
        }
    }
}
```

Consume

```
class Consume extends Thread {
    private ProduceConsume produceConsume;

    public Consume(ProduceConsume produceConsume) {
        this.produceConsume = produceConsume;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                produceConsume.consume();
            } catch (InterruptedException e) {}
        }
    }
}
```

WaitNotify

```
public class WaitNotify {
    public static void main(String[] args) {
        ProduceConsume produceConsume = new ProduceConsume();
        Thread t1 = new Produce(produceConsume);
        Thread t2 = new Consume(produceConsume);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e){}
    }
}
```