Concurrency

COMP 2911, 18s2

Thread Basics

- A thread is a single sequence of execution within a process
- Multi-threading refers to multiple threads of control within a single process
- Process versus a thread
  - A process has a self-contained execution environment with its own dedicated memory space
  - Threads are light-weight compared to processes
  - Threads run within a single process, share the same address space and files of the process.
  - All Java programs have at least one thread, known as the main thread, which is created by the JVM at the program’s start, when the main() method is invoked with the main thread.

Creating and Starting Threads

- Create a thread in Java as:

```
Thread thread = new Thread();
```

- To start the Java thread you will call its start() method, like this:

```
thread.start();
```

- The example above, does not specify any code for the thread to execute
- The thread will stop again right away after it is started
- There are two ways to specify what code a thread should execute
  - Extend the Thread Class
  - Implement the interface Runnable

Extending the Thread class

- Create a subclass of Thread and override the run() method
- The `run()` method is executed, when you invoke `start()` on the thread

```
public class MyThread extends Thread {
    public void run() {
          System.out.println("MyThread running");
    }
}
```

- To create and start the above thread

```
MyThread aThread = new MyThread();
aThread.start();
```

Implementing the Runnable interface

- Create a class that implements `Runnable` and the runnable object can be executed by a `Thread`
- The `run()` method is executed, when you invoke `start()` on the thread

```
public class MyRunnable implements Runnable {
    public void run() {
        System.out.println(“MyRunnable running");
    }
}
```

- To execute the `run()` method, pass an instance of `Runnable` to a `Thread` in its constructor

```
Thread someThread = new Thread(new MyRunnable());
someThread.start();
```

- When the thread is started it will call the `run()` method of the MyRunnable instance instead of executing it's own `run()` method.

A Common Pitfall

- What is wrong in the code below?

```
public class MyRunnable implements Runnable {
@Override
public void run() {
  System.out.println("Runnable running with: " +
Thread.currentThread().getName());
}
public static void main(String[] args) {
  System.out.println(Thread.currentThread().getName());
  Thread aThread = new Thread(new MyRunnable());
  System.out.println(aThread.getName());
  aThread.run(); // Incorrect! Should be aThread.start();
 }
```

- It appears that the Runnable's `run()` method executed as expected, but it is NOT executed by the new thread you just created. Instead the `run()` method is executed by the thread that created the thread i.e. the Main thread
- You MUST invoke `someThread.start()` on the thread

A Simple Thread Program

```
public class Counter implements Runnable {

 private static int counter = 0;
 public Counter(int counter) {
        this.counter = counter; }

@Override
public void run() {
      for(int i=0; i<5; i++) {
        System.out.println (Thread.currentThread().getName() + ":"
                + counter++);
        }
      }
public static void main(String[] args) {
       Runnable r1 = new Counter(counter);
       Thread t1 = new Thread(r1);
       Thread t2 = new Thread(r1);
       t1.start();  t2.start();
} }
```

Critical Section & Race Condition

- A **race condition** is a special condition that may occur inside a critical section
- A **critical section** is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section
- When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, **the critical section is said to contain a race condition**

BoundedQueue class

```
public class BoundedQueue<E>
{
private Object[] elements;
private int head;
private int tail;
private int size;

public void add(E newValue) {
    elements[tail] = newValue;
    tail++;
    size++;
    if (tail == elements.length)
        tail = 0;
}
```

```
public boolean isFull() {
    return size == elements.length;
}

public E remove() {
    E r = elements[head];
    head++;
    size--;
    if (head == elements.length)
        head = 0;
    return r;
}

public boolean isEmpty() {
    return size == 0;
}
```

Producer Class

```
public class Producer implements Runnable
{
   private String greeting;
   private BoundedQueue<String> queue;
   private int greetingCount;

   public Producer(BoundedQueue<String> aQueue, int count)
   {
      queue = aQueue; greetingCount = count;
   }
   public void run() {
      try {
         int i = 1;
         while (i <= greetingCount) {
            if (!queue.isEmpty()) {
               String greeting = queue.remove();
               System.out.println(greeting);
               i++;
            }
     //... Rest of the code
```

Consumer Class

```
public class Consumer implements Runnable
{
   public Consumer(BoundedQueue<String> aQueue, int count)
   {  queue = aQueue;
      greetingCount = count;
   }

   public void run() {
      try
      {
         int i = 1;
         while (i <= greetingCount) {
            if (!queue.isEmpty()) {
               String greeting = queue.remove();
               System.out.println(greeting);
               i++;
            }
 //... Rest of the code
```

ThreadTester class

How could size be 11? (Race-Condition)

```
public class ThreadTester {

    BoundedQueue<String> = new BoundedQueue<String>(10);

    Runnable run1 = new Producer("Hello", queue);
    Runnable run2 = new Producer("Hello", queue);
    Runnable run3 = new Consumer("Goodbye", queue);

    Thread t1 = new Thread(run1);
    Thread t2 = new Thread(run2);
    Thread t3 = new Thread(run3);

    t1.start();
    t2.start();
    t3.start();
}
```

Lecture Demo: Race Condition

How can the queue be corrupted?

How could size be 11?

Using Java's original synchronization primitives

- Every Java object has an associated object lock
- Acquire and release this lock, simply tag the method with the synchronized keyword

```
// BoundedQueue class

public synchronized void add(E newValue)
{ ... }

public synchronized E remove()
{ ... }
```

```
// Insider producer

If (!queue. isFull()) {

     queue.add(...);

}
```

Why doesn't this work? – Queue corrupted?

- Solution: the test `isFull()` must be moved to `add()`

```
// BoundedQueue class

public synchronized void add(E newValue)
{  try  {
        while (queue is full)
              sleep(); // wait for more space
        ...
       }
```

This still doesn't work…

- If the thread sleeps after acquiring lock, no other thread can remove elements
- A consumer thread calls `remove()` but will be blocked until `add()` exits … Deadlock!!

Problems with Locks (1)

A Deadlocks arises when locking threads result in a situation where they cannot proceed and thus wait indefinitely for others to terminate e.g.,

- One thread acquires a lock on resource r1 and waits to acquire another on resource r2 and at the same time, another thread that has already acquired r2 is waiting to obtain a lock on r1
- Here neither thread can proceed until the other one releases the lock, which never happens – resulting in a deadlock

Problems with Locks (2)

- A Lock Starvation arises when threads have different priorities and a thread scheduler gives lock to high-priority threads.
- If there are many high-priority threads that want to obtain the lock and also hold the lock for long time periods, this could result in a situation where low-priority threads “starve” for a long time trying to obtain the lock

Revised bounded queue

```
public synchronized void
        add(E newValue) {
    while (isFull()) wait();
    elements[tail] = newValue;
    tail++;
    size++;
    if (tail ==
elements.length)
        tail = 0;
    notifyAll();
}
```

```
public synchronized E
        remove() {
    while (isEmpty()) wait();
    E r = elements[head];
    head++;
    size--;
    if (head == elements.length)
        head = 0;
    notifyAll();
}
```

- This does work, but can it be better?

Synchronisation using Object Locks

```
public void add(E newValue) {
   queueLock.lock();
   while (isFull()) sleep();
   elements[tail] = newValue;
   tail++;
   size++;
   if (tail == elements.length)
      tail = 0;
   queueLock.unlock();
}
```

```
public E remove() {
   queueLock.lock();
   while (isEmpty()) sleep();
   E r = elements[head];
   head++;
   size--;
   if(head ==
elements.length)
      head = 0;
   queueLock.unlock();
```

Java's new `Lock` interface, is preferred over using primitive `synchronised`…but the code above has still same issue deadlock

Reentrant Locks: Using Lock and Condition Objects

```
private Lock queueLock = new ReentrantLock();
private Condition spaceAvailableCond = queueLock.newCondition();

private Condition valueAvailableCond = queueLock.newCondition();
```

```
public void add(E newValue) {
    queueLock.lock();
    try {
      while (isFull())
         space.await();
      elements[tail] = newValue;
      tail++;
      size++;
      if (tail == elements.length)
          tail = 0;
       valueAvailableCond.signalAll();
    }
    finally {queueLock.unlock();}
}
```

```
public E remove() {
 queueLock.lock();
 try {
    while (isEmpty())
        value.await();
    E r = elements[head];
    head++;
    size--;
    if (head== lements.length)
         head = 0;
    spaceAvailableCond.signalAll();
 }
 finally { queueLock.unlock();}
}
```