SWEN90004
Modelling Complex Software Systems

Java threads; mutual exclusion

Nic Geard

Lecture Con.02

Semester 1, 2019

Overview

Threads in Java

Java calls a process a "thread".

There are two ways to create threads in Java. The first is to extend the `java.lang.Thread` class:

```
class MyThread extends Thread {
  public void run() {
    //insert process code here
    System.out.println("SWEN90004 thread");
    for (int i = 0; i < 10; i++) {
      System.out.println("\t thread " + i);
    }
  }
}
```

Threads in Java

Then, create an instance of this class:

```
public class UseMyThread {
  public static void main(String [] args)
  {
    Thread myThread = new MyThread();
    myThread.start();
  }
}
```

The first statement creates the thread.

The call to `start()` causes the new thread to call its `run()` method and execute it independently of the caller.

Threads in Java, second way

As Java does not support multiple inheritance, you may not always be able to extend class `Thread`.

The alternative (and usually recommended!) way to create a thread is to implement the `Runnable` interface:

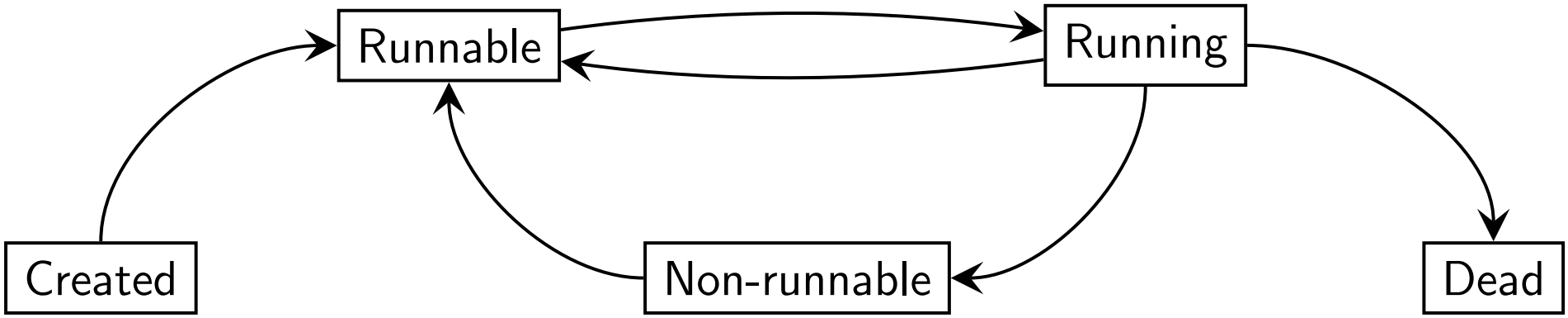
```
class MyRunnable implements Runnable {
  //we must implement the "run()" method
  public void run() {
    System.out.println("SWEN90004 runnable");
    for (int i = 0; i < 10; i++) {
      System.out.println("\t runnable " + i);
    }
  }
}
```

Threads in Java, second way

Then, create an instance of this using Thread:

```
public class UseMyRunnable {
  public static void main(String [] args)
  {
    Thread myRunnable =
      new Thread(new MyRunnable());
    myRunnable.start();
  }
}
```

Thread states



A thread that is alive is always in one of three states:

- running: it is currently executing;
- runnable: it is currently not executing but is ready to execute; or
- non-runnable: it is not running and is not ready to run—may be waiting on some input or shared data to become unlocked.

Java thread primitives

Calling `start()` causes the Java virtual machine to execute the `run()` method in a dedicated thread, concurrent with the calling code.

A thread stops executing when `run()` finishes.

Note: we don't call `run()` directly! (perhaps just when testing). Doing so executes the method in the current thread.

A thread can be suspended for a specified amount of time using `sleep(long milliseconds)`.

Java thread primitives

We can test whether a thread is running using the `isAlive()` method.

The method `yield()` causes the current thread to pause, going from "running" status to "runnable".

When it goes from "runnable" to "running" (ie, executes `run()` again) is a matter for a runtime system's scheduler to decide.

Calling `t.join()` suspends the caller until thread `t` has completed. (In this sense the two join together.)

Additional suspension states

To account for Java's concurrency primitives fully, we need to consider additional states that a thread can be in:

- Having called `sleep()`;
- having called `join()`;
- waiting for a lock to be released (having called `wait()` – *we will talk about this more next week*).

A thread can be interrupted through `Thread.interrupt()`.

If interrupted in one of the three states above, it will return to "runnable" state, and `sleep()`, `join()`, or `wait()` will throw an `InterruptedException`.

Interference example in Java

```
class Count extends Thread {
    static volatile int n = 0;

    public void run() {
      int temp;
      for (int i = 0; i < 10; i++) {
        temp = n;
        n = temp + 1;
      }
    }

    public static void main(String[] args) {
      Count p = new Count();
      Count q = new Count();
      p.start();
      q.start();
      try { p.join(); q.join(); }
      catch (InterruptedException e) { }
      System.out.println("The final value of n is " + n);
    }
}
```

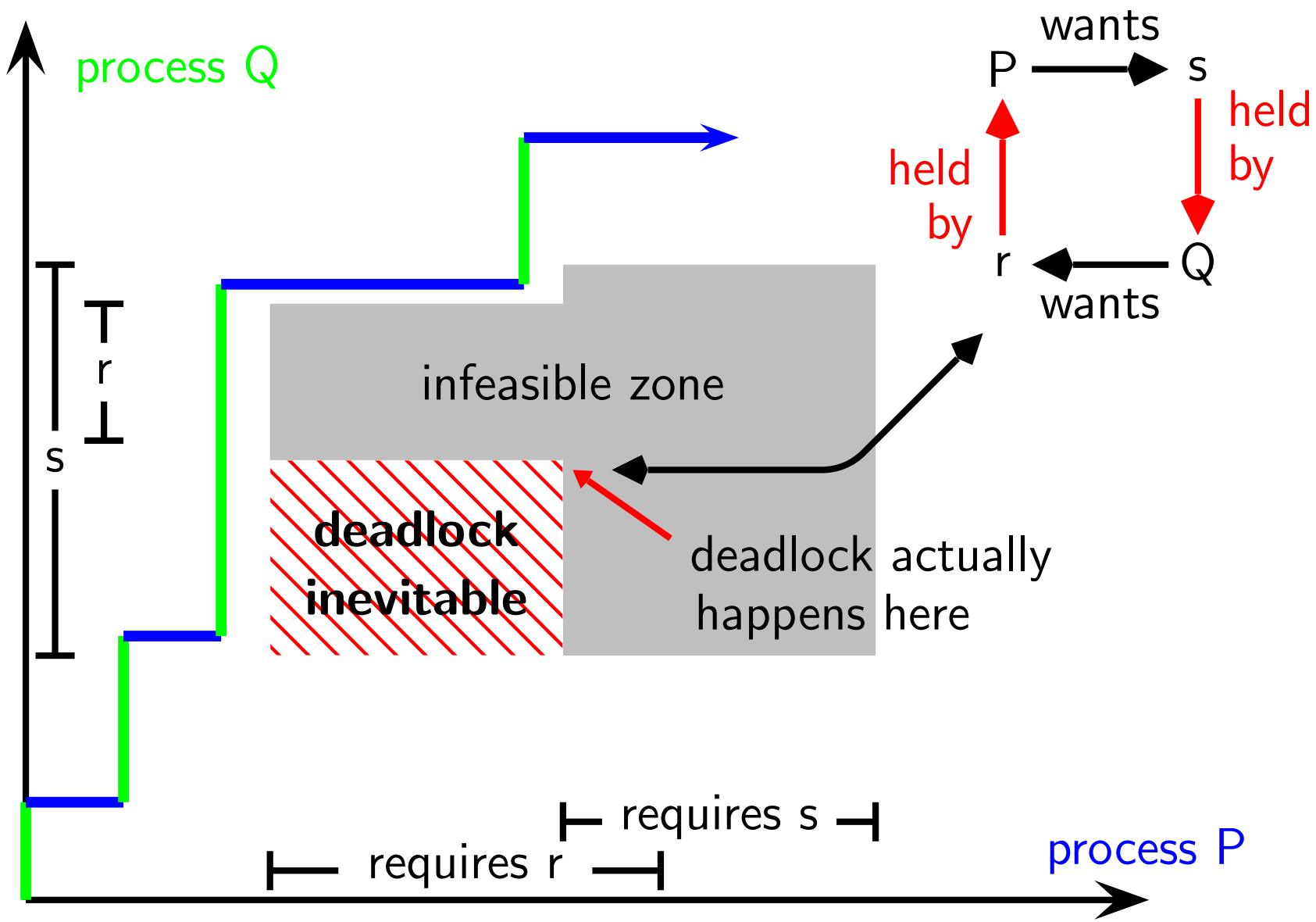
Deadlock

Deadlock is a situation where a set of processes are unable to make any further progress, because of mutually incompatible demands they make of shared resources.

Concurrent languages provide means for mutual exclusion and for processes to wait for required resources. They also use a principle of no preemption, that is, a process has to give up resources voluntarily.

We will return to these conditions later.

Potential deadlock (two processes)



The mutual exclusion (Mutex) problem

N processes (here, $N = 2$) are executing infinite loops, each alternating between a critical and a non-critical section. A process may halt in its non-critical section, but not in its critical section.

```
class P extends Thread {
  while (true) {
    non_critical_P();
    pre_protocol_P();
    critical_P();
    post_protocol_P();
  }
}
```

```
class Q extends Thread {
  while (true) {
    non_critical_Q();
    pre_protocol_Q();
    critical_Q();
    post_protocol_Q();
  }
}
```

Shared variables are only written to in the critical section, so in order to avoid race conditions, only one thread can be in its critical section at any time.

Desirable properties of a Mutex solution

There are a number of ways to solve this problem, but a solution should have the following properties:

- Mutual exclusion: only one process may be active in its critical section at a time.
- No deadlock: if one or more processes are trying to enter their critical section, one must eventually succeed.
- No starvation: if a process is trying to enter its critical section, it must eventually succeed.

Also desirable:

- Handles lack of contention: if only one process is trying to enter its critical section, it must succeed with minimal overhead.

Rules of the game

- No variable used in a protocol is also used in critical/non-critical sections, and vice versa.
- Load, store, and test of common memory variables are atomic operations.
- There must be progress through critical sections: once a process reaches its critical section, it must eventually reach the end of the section.
- We cannot assume progress through non-critical sections: while in such a section, a process might terminate or enter an infinite loop.

First attempt

```
static int turn = 1;
```

```
class P extends Thread {
  public void run() {
    while (true) {
p1:   non_critical_P();
p2:   while (turn != 1);
p3:   critical_P();
p4:   turn = 2;
    }
  }
}
```

```
class Q extends Thread {
  public void run() {
    while (true) {
q1:   non_critical_Q();
q2:   while (turn != 2);
q3:   critical_Q();
q4:   turn = 1;
    }
  }
}
```

Note that `while (! property);` is just a way of implementing an `await(property)` statement.

Properties of first attempt

Mutual exclusion: Yes. Each thread can only enter the critical section when it is its turn.

No deadlock: Yes. If the two processes are at `p2` and `q2` respectively, `turn` must be either 1 or 2, so one can enter. Alternatively, if the processes are at `p2` and `q3`/`q4`, `P` must go through the outer loop (`turn` must be 2) until `q4` is executed. If we assume `q4` is eventually executed, then the program is free from deadlock.

No starvation: No! Let `P` and `Q` be at `p1` and `q1` respectively. If process `Q` goes to `q2`, but process `P` continues executing its non-critical section indefinitely (or dies), `Q` will be starved.

Handles lack of contention: No! As above.

Second attempt

```
static int p = 0;
static int q = 0;
```

```
class P extends Thread {
  public void run() {
    while (true) {
p1:   non_critical_P();
p2:   while (q != 0);
p3:   p = 1;
p4:   critical_P();
p5:   p = 0;
    }
  }
}
```

```
class Q extends Thread {
  public void run() {
    while (true) {
q1:   non_critical_Q();
q2:   while (p != 0);
q3:   q = 1;
q4:   critical_Q();
q5:   q = 0;
    }
  }
}
```

Variable p is set to 1 when thread P *wants* to enter its critical section, and then set to 0 when it is done (same for q and Q).

Second attempt

```
static int p = 0;
static int q = 0;
```

```
class P extends Thread {
  public void run() {
    while (true) {
p1:   non_critical_P();
p2:   while (q != 0);
p3:   p = 1;
p4:   critical_P();
p5:   p = 0;
    }
  }
}
```

```
class Q extends Thread {
  public void run() {
    while (true) {
q1:   non_critical_Q();
q2:   while (p != 0);
q3:   q = 1;
q4:   critical_Q();
q5:   q = 0;
    }
  }
}
```

Mutual exclusion: No. Consider p1 $\rightarrow$ p2 $\rightarrow$ q1 $\rightarrow$ q2 : each process is now in its critical section.

Third attempt

```
static int p = 0;
static int q = 0;
```

```
class P extends Thread {
  public void run() {
    while (true) {
p1:   non_critical_P();
p2:   p = 1;
p3:   while (q != 0);
p4:   critical_P();
p5:   p = 0;
    }
  }
}
```

```
class Q extends Thread {
  public void run() {
    while (true) {
q1:   non_critical_Q();
q2:   q = 1;
q3:   while (p != 0);
q4:   critical_Q();
q5:   q = 0;
    }
  }
}
```

Each process now sets its *want* flag before waiting.

Properties of third attempt

Mutual exclusion: Yes. Once a process enters its inner while loop (at `p3`/`q3`), it must be assured that the other process cannot enter its inner loop. Because the other process sets its own flag prior to this, mutual exclusion is maintained.

No deadlock: No. Consider the interleaving `p1` $\rightarrow$ `q1` $\rightarrow$ `p2` $\rightarrow$ `q2`: each process waits for the other to set its flag, so deadlock occurs.

No starvation: No. There is deadlock; both processes are starved.

Handles lack of contention: Yes. If `P` is in its non-critical section, then it must be that `p == 0`, so `Q` can enter its critical section.

Fourth attempt

```
static int p = 0;
static int q = 0;
```

```
class P extends Thread {
  public void run() {
    while (true) {
p1:   non_critical_P();
p2:   p = 1;
p3:   while (q != 0) {
p4:     p = 0;
p5:     p = 1;
      }
p6:   critical_P();
p7:   p = 0;
    }
  }
}
```

```
class Q extends Thread {
  public void run() {
    while (true) {
q1:   non_critical_Q();
q2:   q = 1;
q3:   while (p != 0) {
q4:     q = 0;
q5:     q = 1;
      }
q6:   critical_Q();
q7:   q = 0;
    }
  }
}
```

Properties of fourth attempt

Mutual exclusion: Yes. See third attempt.

No deadlock: Yes. The inner loops force each process to set their flag to 0 for a brief period, removing the deadlock.

No starvation: No. Let `P` and `Q` be at `p2` and `q2` respectively. Then the interleaving `p3` $\rightarrow$ `p4` $\rightarrow$ `p5` $\rightarrow$ `q3` $\rightarrow$ `q4` $\rightarrow$ `q5` can occur, leaving the state unchanged. This can occur indefinitely, resulting in livelock. Livelock is similar to a deadlock, except the variables involved in the livelock are changing; so something is happening (not a deadlock), but what is happening is not useful.

Handles lack of contention: Yes. See third attempt.

Fifth attempt: Dekker's algorithm

```
static int turn = 1; static int p = 0; static int q = 0;
```

```
while (true) {
p0: non_critical_P();
p1: p = 1;
p2: while (q != 0) {
p3:   if (turn == 2) {
p4:     p = 0;
p5:     while (turn == 2);
p6:     p = 1;
      }
    }
p7: critical_P();
p8: turn = 2;
p9: p = 0;
}
```

```
while (true) {
q0: non_critical_Q();
q1: q = 1;
q2: while (p != 0) {
q3:   if (turn == 1) {
q4:     q = 0;
q5:     while (turn == 1);
q6:     q = 1;
      }
    }
q7: critical_Q();
q8: turn = 1;
q9: q = 0;
}
```

Properties of Dekker's algorithm

Mutual exclusion: Yes. As with the previous two attempts, `P` will only enter its critical section if `q != 0`, and *vice versa* for `Q`.

No deadlock: Yes. See the fourth attempt.

No starvation: Yes. If both processes want to enter their critical sections at the same time, the process that executed its critical section most recently is given a lower priority. Therefore, the livelock seen in the fourth attempt is not possible.

Handles lack of contention: Yes. See third attempt.

Unfortunately, Dekker's algorithm has proved hard to generalise to programs with more than two processes.

Exercise and recommended reading

One of the tutorial problems for next week is to apply each of these attempted solutions to Count.java – note that many of them will 'appear' to work reliably!

More formal analysis of mutex solution attempts is covered in

- M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice Hall, 2nd edition, 2006.