Operating Systems And Concurrency: Threads

Dr Amanda Coles

Slides based on Chapter 4 of “Operating Systems Concepts” Silberschatz, Galvin and Gagne and

Chapters 3 and 4 of “Principles of Concurrent and Distributed Programming” M. Ben-Ari

With additional Java Examples

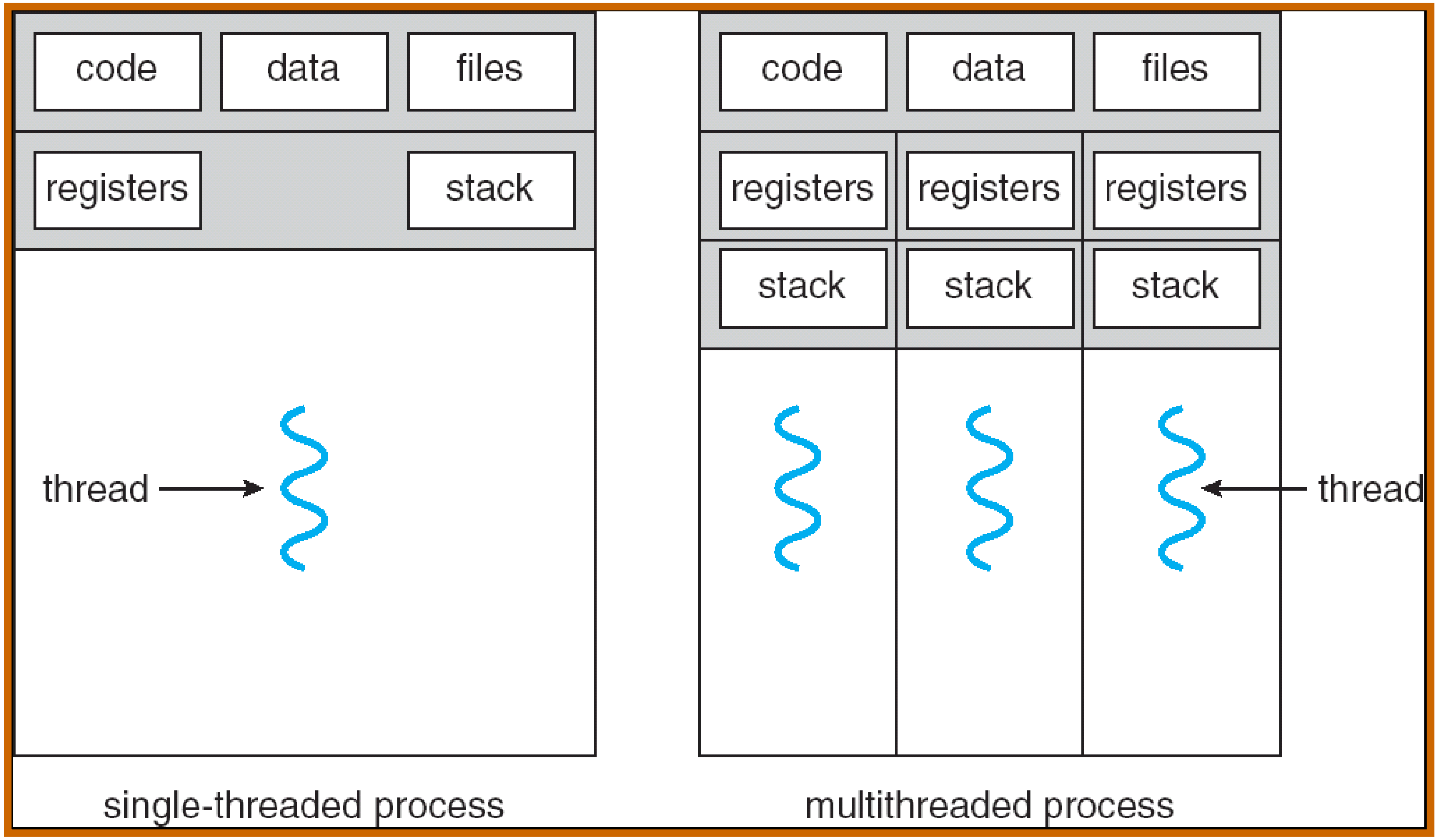
Overview

- Introduction to Threads; *(Operating Systems Concepts)*
- Basic Threading in Java;
- The Critical Section Problem;
- Solutions to the Critical Section Problem. *(Principles of Concurrent and Distributed Programming)*

What is a Thread?

- A thread, is a lightweight process**,** is a basic unit of CPU utilisation.
- It comprises a thread ID, a program counter, a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- A traditional process has a single thread of control.
- If the process has multiple threads of control, it can do more than one task at a time.

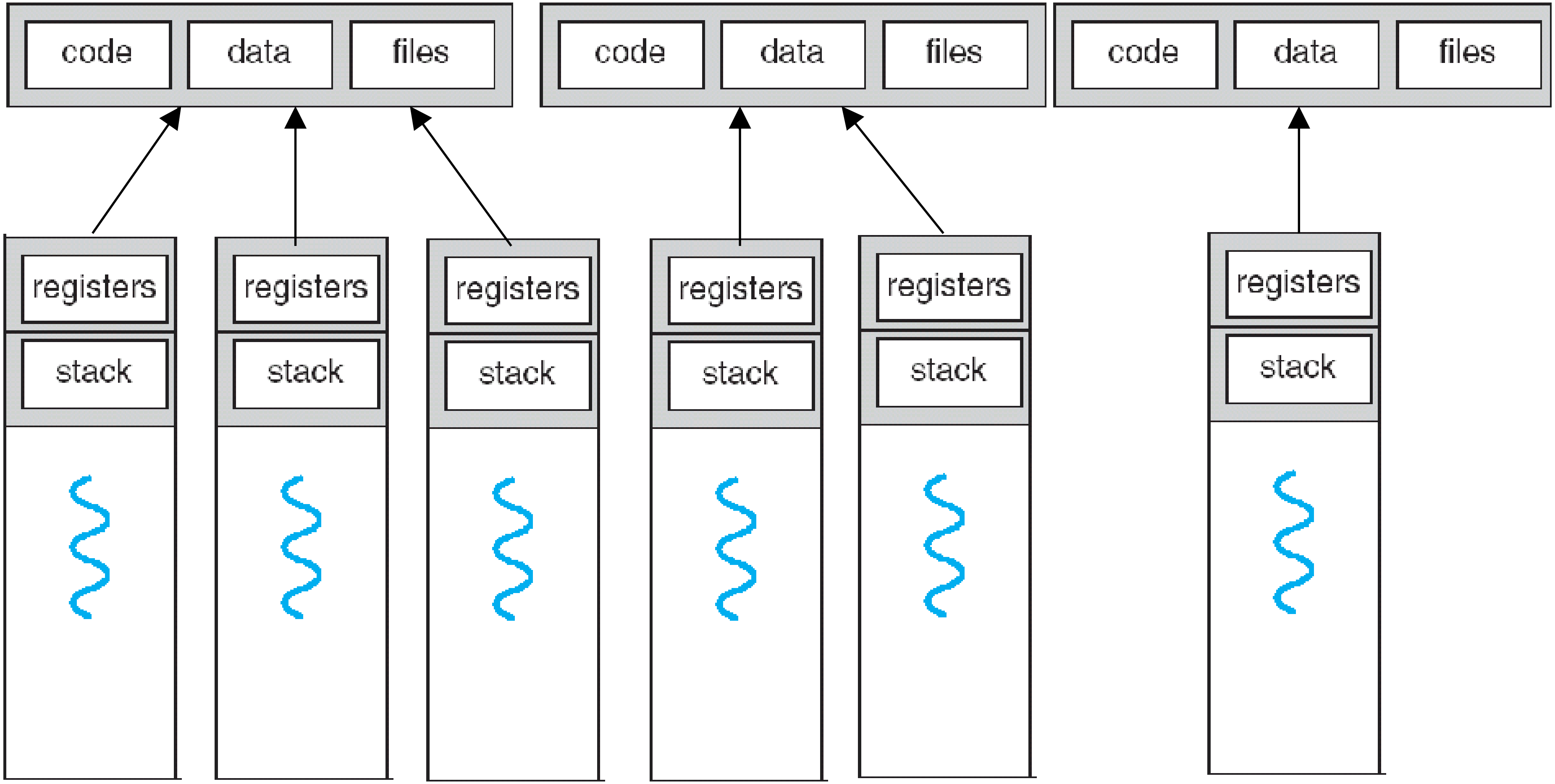
Threads and Processes



Processes and Threads

- **Process:**
  - Isolated with its own virtual address space
  - Contains process data like file handles
  - Lots of overhead
  - Every process has AT LEAST one thread
- **Threads:**
  - Shared virtual address space
  - Contains running state data
  - Less overhead
  - From the OS's point of view, this is what is scheduled to run on a CPU

Processes/Tasks in Linux/Windows



Linux and Windows do not distinguish between processes and threads, multi-threaded processes make many threads that share resources.

Multi-Threading

- Most of the software you run is multi-threaded;
- Examples:
  - A ***web browser*** might have one thread display images or text while another thread retrieves data from the network.
  - A ***word processor*** may have a thread for displaying graphics, another thread for reading keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.
- This is good because if one thread is blocked waiting for I/O (e.g. loading an image) the remainder of the threads can still execute and remain responsive.
- You don’t know it yet, but the Java programs you write with GUIs are multi-threaded automatically.

Multi-Threading 2

- A program might need to perform a similar task many times:
- Examples:
  - A web server needs to serve many connected clients at once:
    - Multi-threaded: create a new thread each time a client arrives to serve that client;
    - Sequential (Non-Threaded): Each client has to wait until the earlier ones have finished before connecting (bad!).
  - An expensive computation needs to be done many times with different parameters:
    - E.g.: protein folding method needs to run to check interactions between two proteins (thousands of pairs); or we need to calculate the colour of each pixel in a fractal.
    - Multi-Threaded: Create a thread to run the method with each type of parameter, run the threads in parallel on many CPUs (or even different machines).
    - Sequential: each one has to wait for the previous one to complete, even if we have 100 CPUs…

Could we do that with Processes?

- In theory yes:
  - Create a new process each time a client connects to server;
  - For every pair of proteins;
  - For every pixel (or row of pixels) to be calculated.

- ***But*** This has more overheads:
  - The creation of a thread is much cheaper (no memory space needs to be created).
  - Also since the same data, variables and code will be required to service each client there’s no point duplicating that.

Benefits of Multi-Threading

- Responsiveness: If one thread is blocked (e.g. waiting for I/O) the other threads in the process can still run, meaning the user can still interact with the program;

- Resource Sharing: Avoids duplication, allows threads to interact with each other easily;

- Efficiency: Threads are cheaper to create (no memory assignment) and to context switch.

- Multi-processor Utilisation: A single threaded process can only ever run on one CPU. Multiple threads can be distributed over multiple CPUs: more concurrency = faster execution time.

Could I do this all Myself?

- I could write a piece of code that appeared to be a single process:
  - Internally what it does is:
    - Run one thread for 8ms (e.g. background calculation).
    - Run another thread for 1ms (e.g. check for keyboard input).
    - Run another thread for 1ms (e.g. check/update GUI);
    - Or I could implement a scheduling algorithm myself.

- Now the OS thinks this is one thread and when my process is scheduled to run, it runs whichever is due.

- Or I could let the Operating System do the work for me

Overview

- Introduction to Threads;
- Basic Threading in Java;
- The Critical Section Problem;
- Solutions to the Critical Section Problem.

Making an Object a Thread

```
public class PrintP extends Thread {


  public void run(){
      for(int i = 0; i < 100; ++i){
          System.out.println(“P”);
      }
  }
}
```

```
public class PrintQ implements runnable
    extends SomethingElse {

  public void run(){
      for(int i = 0; i < 30; ++i){
          System.out.println(“Q”);
      }
  }
}
```

- Two ways of making an object able to run as a thread:
  - Extend the class Thread;
  - Implement runnable;
- Why two? Java does not allow multiple inheritance, so if something is already extending something else we can’t extend Thread.
- Both ways require the implementation of a method run():
  - This is the method that will be called when the thread is started, i.e. the code that determines what the thread will do.

Starting a Thread

```
public class myMain {
  public static void main(String[] args){
    Thread p = new PrintP();
    Thread q = new Thread(new PrintQ());
    p.start();
    q.start();
    for(int i = 0; i < 200; ++i){
      System.out.println(“R”);
    }
  }
}
```

(Extended Version)

```
Runnable qRun = new PrintQ();
Thread q = new Thread(qRun);
```

- Instantiate objects of the appropriate type:
- Thread has a method **`start()`** which tells the JVM to create a new thread and then call the **`run()`** method in that new thread;
  - Do not call **`run()`** directly: otherwise the creating a new thread part is missed out and the **`run()`** method runs just as any other method.
  - Slight Detour: similarly never call **`paint()`** always call **`repaint():`** repaint starts a new thread then calls **`paint()`**.
- Notice because **`PrintQ`** implements **`runnable`** and we need a **`Thread`** to call start, we have to use the constructor of `Thread` that takes a **`runnable`** object as an argument, to get a **`Thread`** object for **`PrintQ`**.

How Many Threads Are Running?

```
public class myMain {
  public static void main(String[] args){
    Thread p = new PrintP();
    Thread q = new Thread(new PrintQ());
    p.start();
    q.start();
    for(int i = 0; i < 200; ++i){
      System.out.println(“R”);
    }
  }
}
```

- How many threads are running:
  - No, not 2, 3!
  - The main method itself is in a thread (the original one for the program) that carries on executing whilst the other threads also execute.
- What is the output? Different every time, or at least it will be if the content of the methods is significant enough to so they don’t finish before context switching occurs:
  - The output is at the mercy of the scheduler: ***unpredictable***.

Output: 3 Runs

Threads execute so quickly that the chance of a context switch during execution is small.

```
p.start();
q.start();
for(int i = 0; i < 200; ++i){
   System.out.println(“R”);
}
```

```
RRQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
```

```
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
```

```
RRRRPPPPPPPPPPPPPPPPPPPPPRRRPPPPPPPPPPPRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRPPPPPPPPPPPPPPPPPP
PPRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
```

Wasting Some Time

```
public class PrintP extends Thread {

  public void run(){
     for(int i = 0; i < 100; ++i){
        System.out.println(“P”);
        //waste some time
     }
  }
}
```

```
int n = 0;
for(int j = 0; j < 1000000; ++j){
  ++n;
  --n;
}
```

```
PQPQPPQPPQPQPQPQPQPQPQPQPQPQQPQPQPQPQPQPQPQPQPQPQPQPQP
QPQPQPQPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
```

```
QPQPQQPQPQPQPQPQPQPQPQPQPQPQPQPQPQPQQQPQPPPQQPQPQPQQ
PQPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
```

Wasting Time More Graciously

```
public class PrintP extends Thread {
  public void run(){
     for(int i = 0; i < 100; ++i){
        System.out.println(“P”);
        try{
          Thread.sleep(30);
        } catch (InterruptedException e){
          e.printStackTrace();
        }
     }
  }
}
```

```
public class PrintQ implements runnable
   extends SomethingElse {
  public void run(){
    for(int i = 0; i < 30; ++i){
      System.out.println(“Q”);
    }
    try{
     Thread.sleep(50);
   } catch (InterruptedException e){
         e.printStackTrace();
   }
  }
}
```

- Threads can sleep: this stops their execution and allows other threads to run:
  - Why? Suppose a thread is checking a value waiting for it to change, we don’t want to hog the CPU (more later).
  - When threads sleep they enter a blocked state so context switches happen.

```
PRRRR…RRRRQRRR..RRRPPQPQPPQPPQPQPPQPQPPQPPQPPQPQP
PQPPQPQPPQPPQPQPPQPPQPQPPQPQPPQPPQPQPPQPPQPPQPQPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
```

Waiting for a Thread: `join()`

```
public static void main(String[] args){
  Thread p = new PrintP();
  Thread q = new Thread(new PrintQ());
  p.start();
  q.start();
  q.join(); //try catch needed around this
  for(int i = 0; i < 200; ++i){
    System.out.println(“R”);
  }
}
```

- Sometimes we might want to wait for a thread we start to finish before we continue.

q.join()

```
QPPQPPQPQPPQPPQPQPPQPPQPQPPQPPQPQPPQPPQPQPPQPPQPPQPQPPQPP
QPQPPQPPQPQPPQPPQPQPPQPPRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
```

p.join()

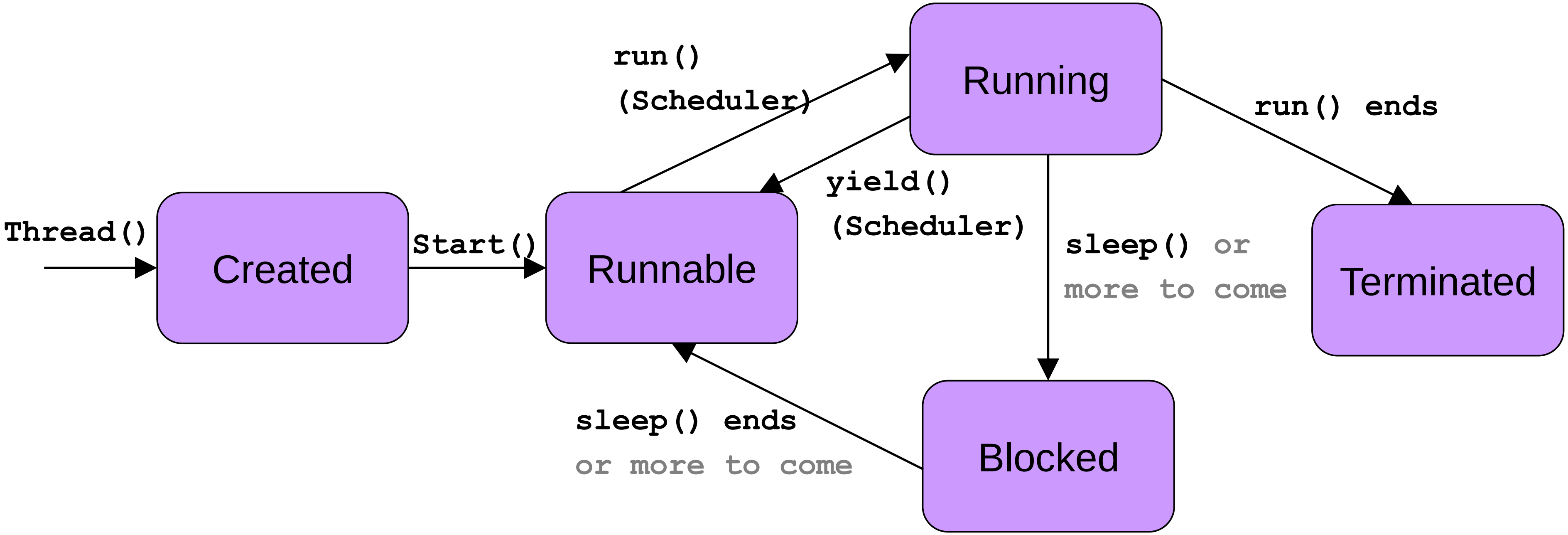
```
QPPQPPQPQPPQPPQPQPPQPPQPPQPQPPQPPQPQPPQPPQPQPPQPPQPQPPQP
PQPQPPQPPQPQPPQPPQPQPPQPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
```

What is this **`InterruptedException`**?

- In general there are two methods for cancelling (killing) threads:
  - ***Asynchronous Cancellation***: The thread is terminated immediately;
  - ***Deferred Cancellation***: The thread checks periodically to see whether it is to terminate.

- In Java a thread might be killed whilst it is sleeping:
  - In this case an **`InterruptedException`** is thrown:
    - When this is caught the thread can do whatever tidying up it needs to do when it is terminated;
    - Doesn’t need to check all the time whether it is to be terminated but can still clean up (e.g. close files).

Java Thread Lifecycle



Overview

- Introduction to Threads;
- Basic Threading in Java;
- The Critical Section Problem;
- Solutions to the Critical Section Problem.

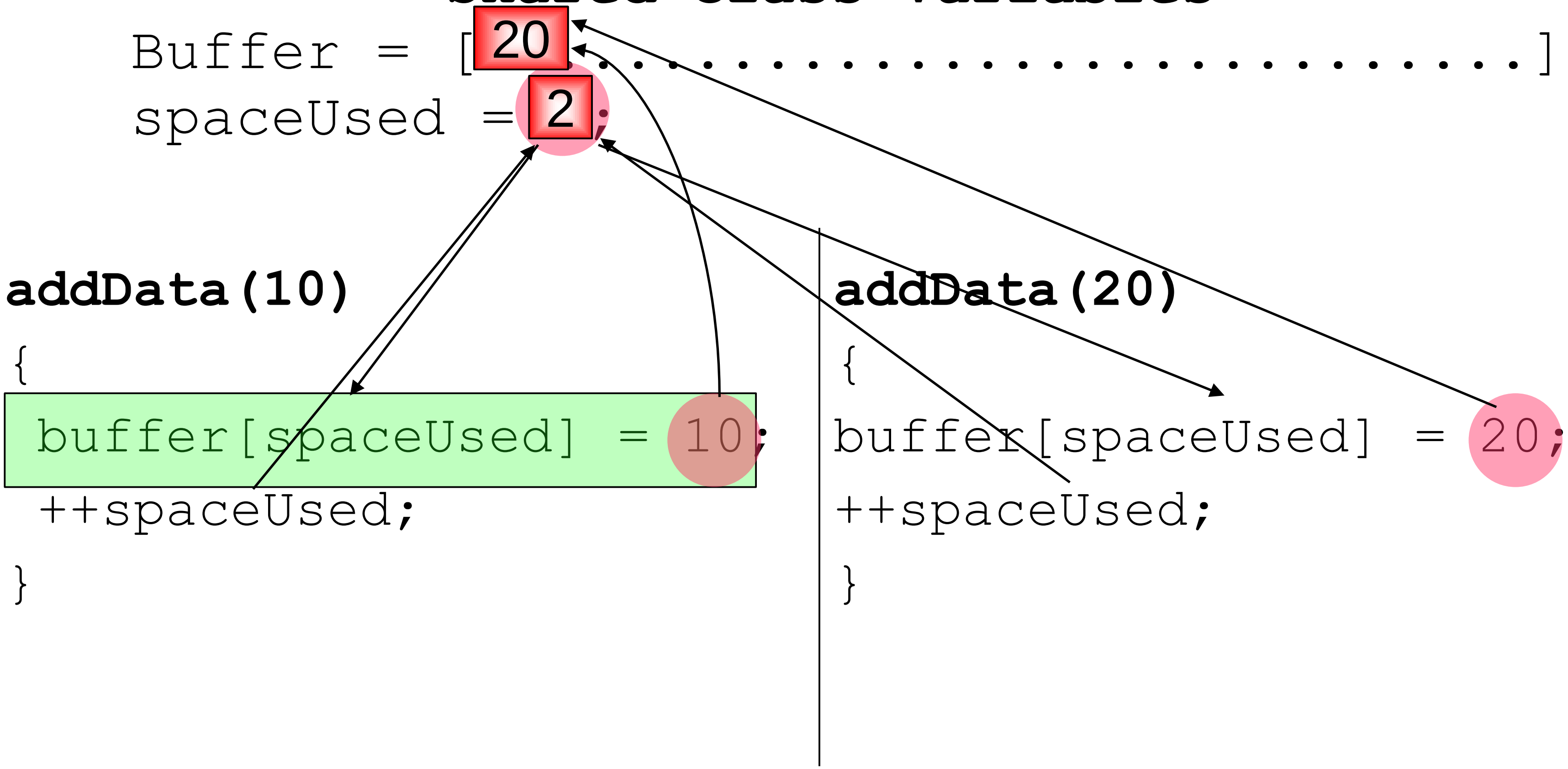
Shared Resources

```
public static void main(String[] args){
  int[] buffer = new int[10];
  Thread p = new pcThread(buffer);
  Thread q = new pcThread(buffer);
  p.start();
  q.start();
}
```

- These two threads now both use ***the same*** array to perform operations on.
  - Not a copy of the same array, but actually the same physical space in memory.
  - Can use this to pass data between them.

- Let’s have a look at what can happen.
  - Consider a method addData(int n) which puts n at the end of the buffer.

Shared Class Variables



```
Buffer = [........................]
spaceUsed = ;
```

```
addData(10)
{
 buffer[spaceUsed] = 10;
 ++spaceUsed;
}
```

```
addData(20)
{
buffer[spaceUsed] = 20;
++spaceUsed;
}
```

- We don’t know what order the threads will be interleaved:
  - Sometimes we will get a crash;
  - Sometimes we won’t;
  - Race condition.
- Makes bug fixing much harder.

It Gets Worse…

- Each line of Java in the previous slide gives rise to multiple machine code instructions (remember CPUs run machine code not Java…);
  - Let’s take the simplest one: ++spaceUsed;

```
mov R1, .spaceUsed    //put address of spaceUsed into Register R1
ldm R0, [R1]          //load to R0 from memory address in R1
add R0, R0, 1         //R0 = R0 + 1
stm R0, [R1]          //store R0 in memory address in R1
```

Interleaving

```
mov R1, .spaceUsed     //put address of spaceUsed into Register R1
ldm R0, [R1]           //load to R0 from memory address in R1
```

R1 .SpaceUsed R0 0

=> Context switch store R0,R1 in memory

```
mov R1, .spaceUsed     //put address of spaceUsed into Register R1
ldm R0, [R1]           //load to R0 from memory address in R1
```

R1 .SpaceUsed R0 0

=> Context switch store R0,R1 in memory load R0,R1.

```
add R0, R0, 1          //R0 = R0 + 1
stm R0, [R1]           //store R0 (1) in memory address in R1
```

R1 .SpaceUsed R0 1

=> Context switch store R0,R1 in memory load R0,R1.

```
add R0, R0, 1          //R0 = R0 + 1
stm R0, [R1]           //store R0 (1) in memory address in R1
```

R1 .SpaceUsed R0 1

=> Now spaceUsed = 1

Atomicity

- An atomic statement is a single statement that cannot be interrupted:
  - Cannot split the atom in CS (ignore physics!).

- Concurrency is the interleaving of atomic statements;
  - To save us having to work in machine code, we'll assume that assignment statements are atomic: the same theory applies at machine code level to allow use to make sure they are.
  - Each line in the pseudocode we see from now on we will assume is an atomic instruction.

- We still have problems, in the earlier example we saw interleaving of two lines of code that we assumed to be atomic causing problems.

Critical Section

```
addData(10)
{
 buffer[spaceUsed] = 10;
 ++spaceUsed;
}
```

```
addData(20)
{
buffer[spaceUsed] = 20;
++spaceUsed;
}
```

- A Critical Section a the part of a program which needs to be executed atomically (w.r.t. other related threads)
  - If two or more threads enter the critical section simultaneously then errors could occur.

Critical Sections in Java

- Java provides the keyword **`synchronized`** which can be used to mark a method as a critical section;

- If this keyword is used, then for each instance of the given object only one synchronised method can be run at once.

- More than one method can be marked as **`synchronized`** in which case none of these methods can be run at the same time.

- How does Java manage this? That's the topic of the next few lectures, but for now we'll get familiar with using it.

Synchronisation Example

```
public class Buffer {
    int[] buffer;
    int spaceUsed;

    public Buffer(int size){
        buffer = new int[size];
        spaceUsed = 0;
    }

    public synchronized void add(int toAdd){
      buffer[spaceUsed] = toAdd;
      ++spaceUsed;
    }

    public synchronized void printBuffer(){
        System.out.print(buffer[0]);
        for(int i = 1; i <= spaceUsed; ++i){
            System.out.print(", " + buffer[i]);
        }
        System.out.println();
   }
}
```

We are assuming that the buffer is sufficiently big that it won’t get full.

No two threads may be allowed to call add at once;

Also no thread can be allowed to call print whilst something else is adding.

We assume print is not called on an empty buffer

```
public class Producer extends Thread {
  int value;
  Buffer buffer;

  public Producer(Buffer toUse,
                     int toProduce){
     value = toProduce;
     buffer = toUse;
  }

  public void run(){
    for(int i = 0; i < 5; ++i){
      buffer.add(value);
      buffer.printBuffer(); //then optionally waste some time…
    }}}  //on one line to save slide space
```

```
public static void main(String[] args){
  Buffer buffer = new Buffer(10);
  Thread p = new Producer(buffer,1);
  Thread q = new Producer(buffer,2);
  p.start();
  q.start();
}
```

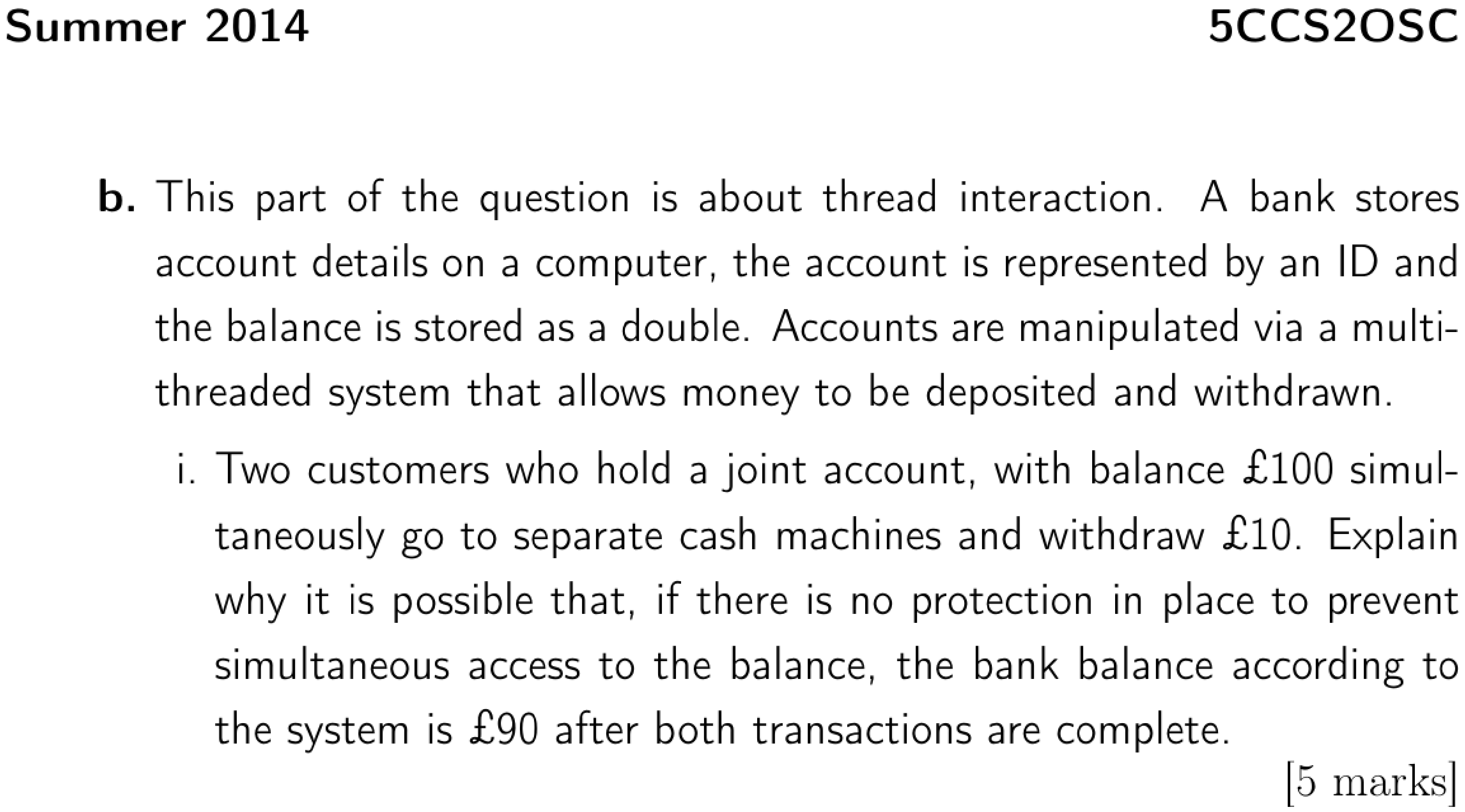
With and Without Synchronized

(and some time wasting)

```
[1]
[1, 2]
[1, 2, 1]
[1, 2, 1, 2]
[1, 2, 1, 2, 1]
[1, 2, 1, 2, 1, 2]
[1, 2, 1, 2, 1, 2, 1]
[1, 2, 1, 2, 1, 2, 1, 2]
[1, 2, 1, 2, 1, 2, 1, 2, 1]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

```
[1]
[1, 1]
[1, 1, 1]
[1, 1[1, 1, 1, 2, 1]
, 1, 2, 1]
[1, 1, 1, 2[1, 1, 1, 2]
, 1, 1, 2, 1, 1, 2]
[1, 1, 1, 2, 1, 1, 2, 2]
[1, 1, 1, 2, 1, 1, 2, 2, 2]
[1, 1, 1, 2, 1, 1, 2, 2, 2, 2]
```

- With **`synchronized`** (left) the output is sensible;
- Without **`synchronized`** (right) one thread prints in the middle of the other printing.
- Update errors are also possible, but didn’t occur this run.



Summer 2014 **5CCS2OSC**

b. This part of the question is about thread interaction. A bank stores account details on a computer, the account is represented by an ID and the balance is stored as a double. Accounts are manipulated via a multi-threaded system that allows money to be deposited and withdrawn.

i. Two customers who hold a joint account, with balance £100 simultaneously go to separate cash machines and withdraw £10. Explain why it is possible that, if there is no protection in place to prevent simultaneous access to the balance, the bank balance according to the system is £90 after both transactions are complete.

[5 marks]

May 2014, Q1 (50/150) marks

Overview

- Introduction to Threads;
- Basic Threading in Java;
- The Critical Section Problem;
- Solutions to the Critical Section Problem.
  - Properties of a solution;
  - Proposed solutions;
  - Proofs of (in)correctness.

Synchronising Critical Sections

Global Variables	
p	q
local variables	**local variables**
loop_forever	**loop_forever**
non-critical section	**non-critical section**
preprotocol	**preprotocol**
critical section	**critical section**
postprotocol	**postprotocol**

- **`preprotocol`**: check that it is okay to enter critical section;
- **`postprotocol`**: signal that critical section is complete.
- Together called a ***synchronisation mechanism***.

Properties of a Solution

- ***Mutual Exclusion***: Only one process can enter the critical section at once.

- ***Freedom from Deadlock***: If some processes are waiting to enter their critical section, eventually one must succeed;
  - If all processes are waiting then we have deadlock.

- ***Freedom from Starvation (Success in the Absence of Contention):*** If a process is waiting to enter its critical section it must do so eventually (regardless of whether others have finished executing).
  - Note that this does not guarantee ‘fairness’ if one process gets 10000 entries into its critical section before another enters the other process is not starved.

- *Starvation vs Deadlock: In deadlock processes must be waiting for a variable assignment that will never happen (there is no chance of escape); in starvation it can be that an assignment might not happen due to interleaving but could happen on a different one.*

There must be no interleaving that breaks these properties.

Checking these Properties

- Standard testing: running the program, doesn’t work, because each time we run the program we only see one interleaving and we need to show correctness for all interleavings.

- Testing can find some bugs, but it’s not guaranteed to find all bugs (unless all possible interleavings are exhaustively tested).

- We therefore need to prove correctness of programs considering all possible cases.

Assumptions

Global Variables	
p	q
local variables **loop_forever** **non-critical section** **preprotocol** **critical section** **postprotocol**	**local variables** **loop_forever** **non-critical section** **preprotocol** **critical section** **postprotocol**

- A process will complete its critical section:
  - It will not exit;
  - There is no infinite loop.
- A process may terminate during its non-critical section.
- The CPU scheduler will not starve a process.
- An atomic assignment operator ←.

First Attempt

integer turn ← 1	
p	q
`loop_forever`	`loop_forever`
`p1:   non-critical section`	`q1:   non-critical section`
`p2:   await turn = 1`	`q2:   await turn = 2`
`p3:   critical section`	`q3:   critical section`
`p4:   turn` ← 2	`q4:   turn` ← 1

- Await is akin to:
  - `while (turn != 2) {//might want to sleep!};`

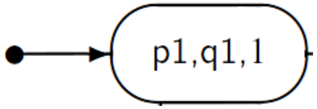
- Does this work?

First Attempt

integer turn ← 1	
p	q

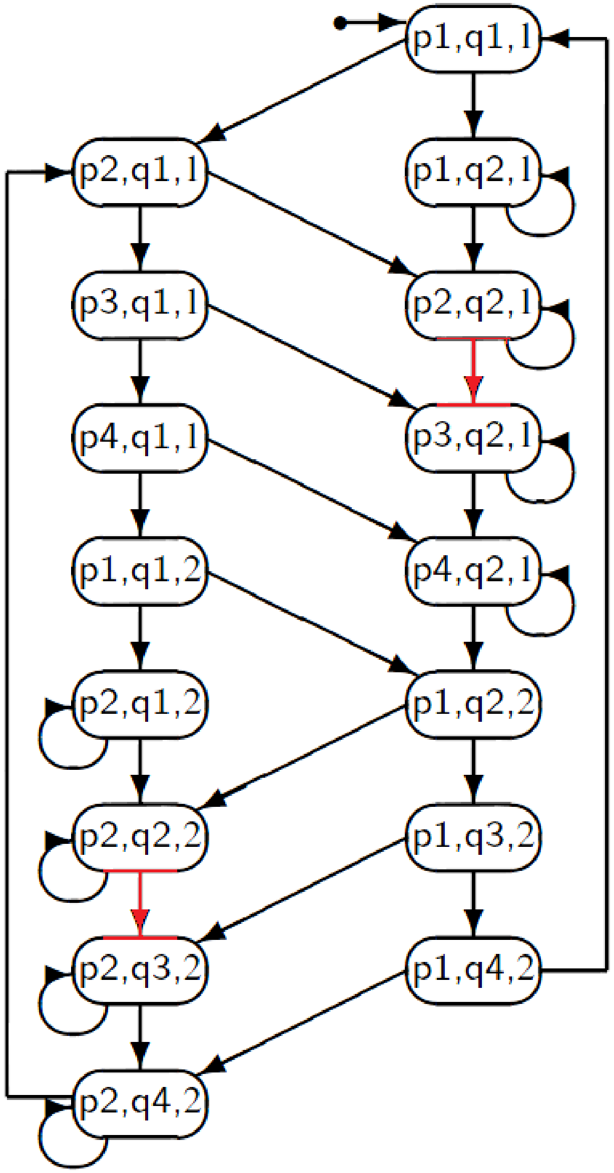
`loop_forever`
`p1:   non-critical section`
`p2:   await turn = 1`
`p3:   critical section`
`p4:   turn ← 2`
	`loop_forever`
	`q1:   non-critical section`
	`q2:   await turn = 2`
	`q3:   critical section`
	`q4:   turn ← 1`

State Diagram Begins:



Full State Diagram

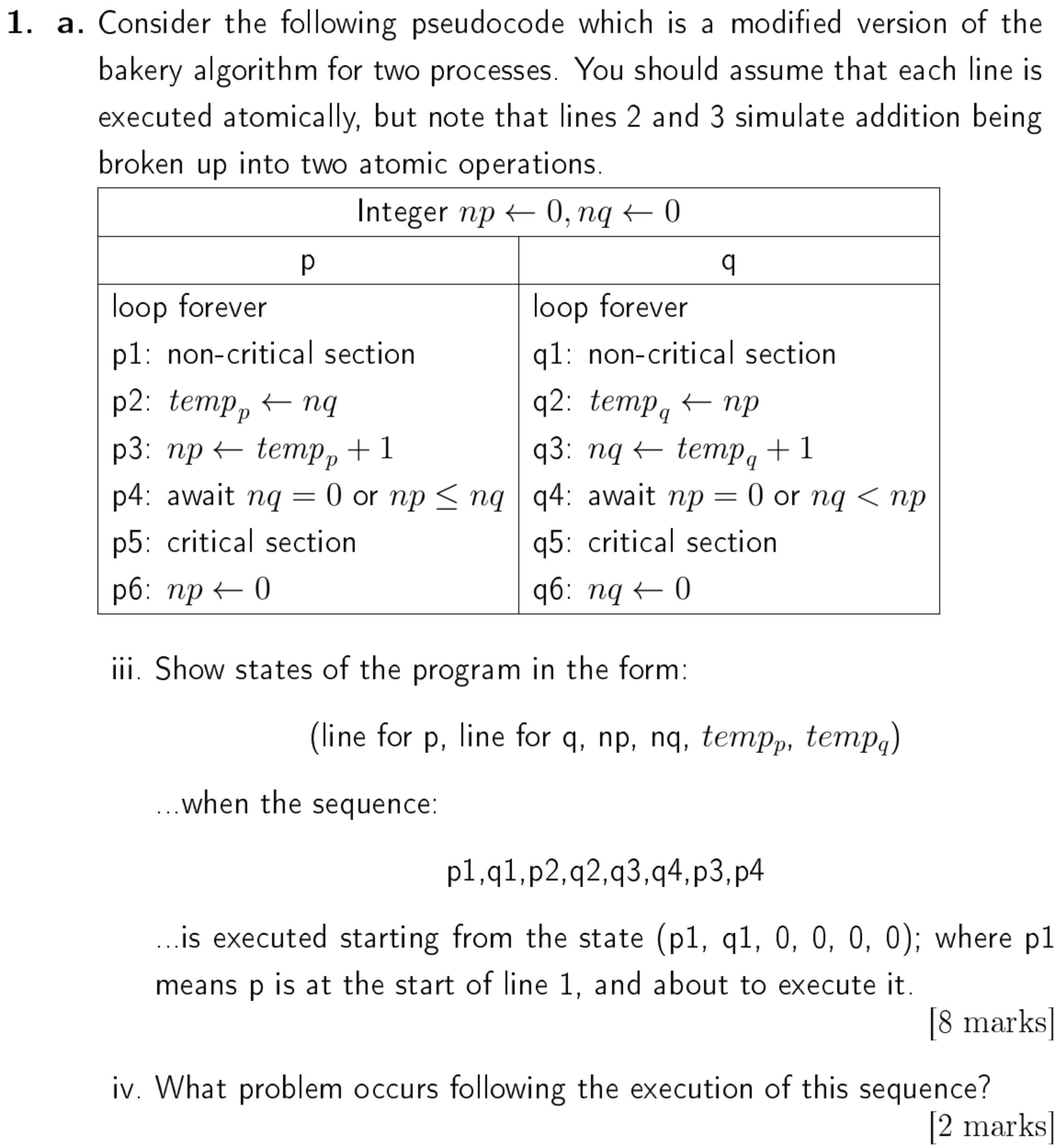
- Shows all execution traces.
- Never are we in p3,q3 therefore ***mutual exclusion*** holds!
- **Deadlock?** Well, the only place we can gets stuck is at await: p2 → p3 or q2 → q3 (red arrows).
- Deadlock will only happen in (p2,q2,_).
  - But if we're in those states then either turn = 1 or turn = 2.
    - turn = 1 Eventually p will be scheduled and progress.
    - turn = 2: eventually q will be scheduled and progress.



Freedom from Starvation (Success in the Absence of Contention)

integer turn ← 1	
p	q
loop_forever p1: non-critical section p2: await turn = 1 p3: critical section p4: turn ← 2	loop_forever q1: non-critical section q2: await turn = 2 q3: critical section q4: turn ← 1

- Suppose we have:
  - (p1,q1,1), (p2,q1,1), (p3,q1,1), (p4,q1,1), (p1,q1,**2**) then q exits (or has an infinite loop) during its non-critical section (which is permitted).
  - p1 cannot continue to execute.
- First attempt does not have this property!



1. **a.** Consider the following pseudocode which is a modified version of the bakery algorithm for two processes. You should assume that each line is executed atomically, but note that lines 2 and 3 simulate addition being broken up into two atomic operations.

Integer $np \leftarrow 0, nq \leftarrow 0$	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: $temp_p \leftarrow nq$	q2: $temp_q \leftarrow np$
p3: $np \leftarrow temp_p + 1$	q3: $nq \leftarrow temp_q + 1$
p4: await $nq = 0$ or $np \leq nq$	q4: await $np = 0$ or $nq < np$
p5: critical section	q5: critical section
p6: $np \leftarrow 0$	q6: $nq \leftarrow 0$

iii. Show states of the program in the form:

(line for p, line for q, np, nq, $temp_p$, $temp_q$)

...when the sequence:

p1,q1,p2,q2,q3,q4,p3,p4

...is executed starting from the state (p1, q1, 0, 0, 0, 0); where p1 means p is at the start of line 1, and about to execute it.

[8 marks]

iv. What problem occurs following the execution of this sequence?

[2 marks]

May 2015, Q1 (50/150) marks

Second Attempt

boolean wantp ← false, wantq ← false	
p	q
loop_forever	**loop_forever**
p0: non-critical section	**q0: non-critical section**
p1: await wantq = false	**q1: await wantp = false**
p2: wantp ← true	**q2: wantq ← true**
p3: critical section	**q3: critical section**
p4: wantp ← false	**q4: wantq ← false**

- First Attempt failed as both were relying on a single global variable.
- Let’s fix that by having two variables: one that says p wants to go into its critical section, and one that q does.
- To be fair, each will set this variable to false before their non-critical section.

Mutual Exclusion: Execution Trace

boolean wantp ← false, wantq ← false	
p	q
loop_forever	loop_forever
non-critical section	non-critical section
p1: await wantq = false	q1: await wantp = false
p2: wantp = true	q2: wantq = true
critical section	critical section
p3: wantp ← false	q3: wantq ← false

Process p	Process q	wantp	wantq
p1: await wantq = false		false	false
	q1: await wantp = false	false	false
p2: wantp ← true		true	false
	q2: wantq ← true	true	true
critical section		true	true
	critical section	true	true

Does Not Satisfy Mutual Exclusion

- Lesson: it’s a lot easier to prove something is incorrect than to prove that it is correct;
  - Prove incorrect: find ***one*** trace that is a counter example;
  - Prove correct: show that ***every*** trace is safe.

- Solution 2 didn’t work so let’s try another…

Third Attempt

boolean wantp ← false, wantq ← false	
p	q

`loop_forever`
`p1:  non-critical section`
`p2:  wantp ← true`
`p3:  await wantq = false`
`p4:  critical section`
`p5:  wantp ← false`
	`loop_forever`
	`q1:  non-critical section`
	`q2:  wantq ← true`
	`q3:  await wantp = false`
	`q4:  critical section`
	`q5:  wantq ← false`

- Mutual Exclusion didn’t work so let’s swap the order.
- Now let’s try to prove mutual exclusion.
- Informal proofs here: formal proofs in bonus material.
- Types of proof (jokes): http://school.maths.uwa.edu.au/~berwin/humour/invalid.proofs.html

Mutual Exclusion (Informal)

- State diagrams can get big and cumbersome to draw, so we’re going to prove this differently.

- For mutual exclusion to hold we must show that (p4, q4, _,_,) cannot be true.

- To get to that state one of the two must already be @4 (and stay there) the other must move from 3 to 4.
  - Processes are symmetric so it doesn’t matter which one.

- So let’s assume p is at p4: What is the value of *wantp*?
  - q cannot modify wantp.
  - The only lines that modify *wantp* are p2 and p5, if we are at p4 then the last one to execute was p2: wantp = true.

- For q to transition from q3 to q4 wantp = false
  - But if p is in p4 wantp = true;
  - Therefore q cannot transition from q3 to q4 if p = p4.

Freedom from Deadlock

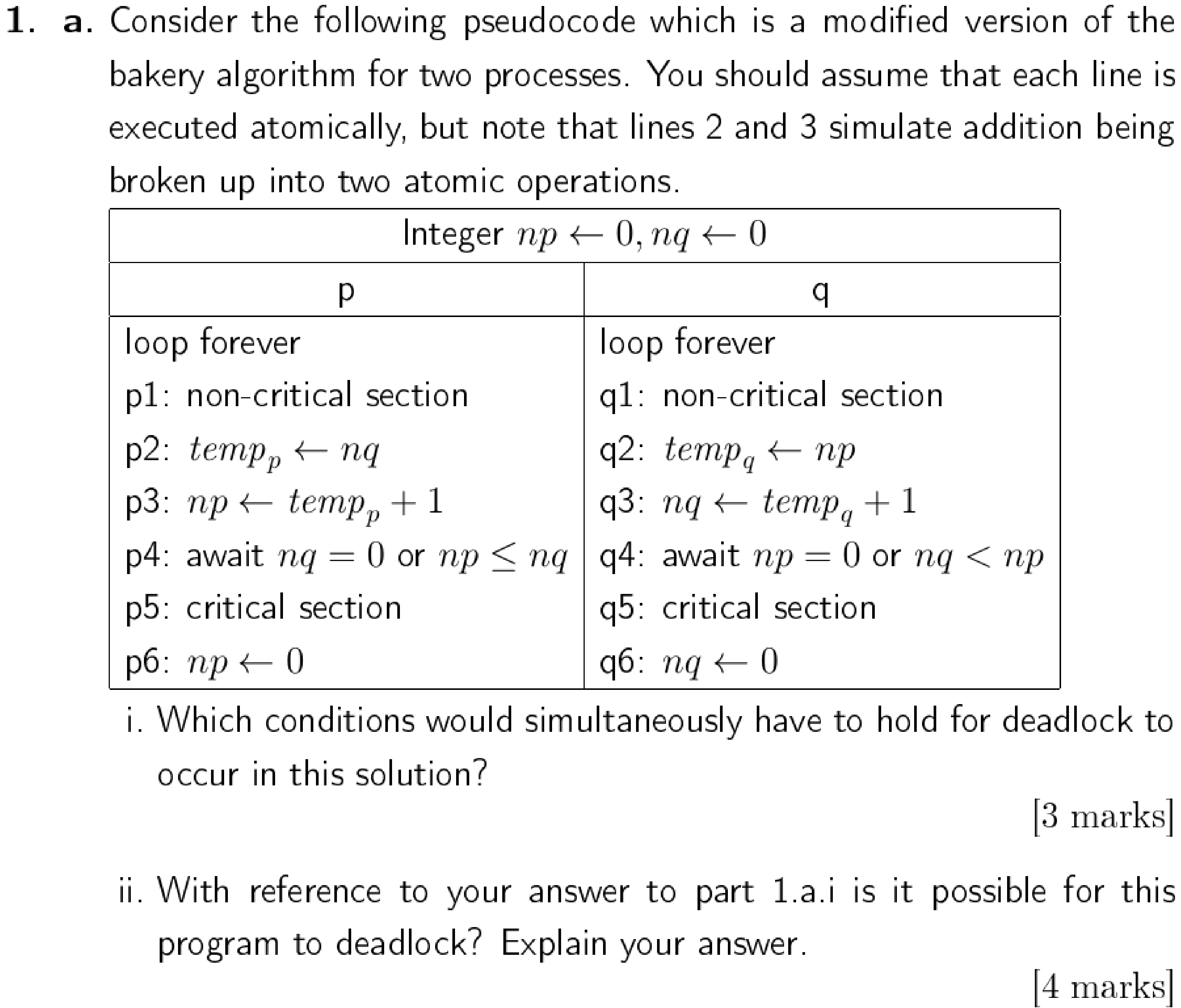
<table>
<tr><th colspan="2">boolean wantp ← false, wantq ← false</th></tr>
<tr><th>p</th><th>q</th></tr>
<tr><td>loop_forever</td><td>loop_forever</td></tr>
<tr><td>p1: non-critical section</td><td>q1: non-critical section</td></tr>
<tr><td>p2: wantp ← true</td><td>q2: wantq ← true</td></tr>
<tr><td>p3: await wantq = false</td><td>q3: await wantp = false</td></tr>
<tr><td>p4: critical section</td><td>q4: critical section</td></tr>
<tr><td>p5: wantp ← false</td><td>q5: wantq ← false</td></tr>
</table>

Process p	Process q	wantp	wantq
p1: non-critical section		false	false
	q1: non-critical section	false	false
p2: wantp ← true		true	false
	q2: wantq ← true	true	true
p3: await wantq = false		true	true
	q3: await wantp = false	true	true

Attempt 4

boolean wantp ← false, wantq ← false	
p	**q**
`loop_forever`	`loop_forever`
`p1:  non-critical section`	`q1:  non-critical section`
`p2:  wantp ← true`	`q2:  wantq ← true`
`P3:  while wantq`	`q3:  while wantp`
`p4:    wantp ← false`	`q4:    wantq ← false`
`P5:    wantp ← true`	`q5:    wantq ← true`
`p6:  critical section`	`q6:  critical section`
`p7:  wantp ← false`	`q7:  wantq ← false`

- This algorithm satisfies some but not all of the three properties.
- Exercise, which? Provide proofs.



1. a. Consider the following pseudocode which is a modified version of the bakery algorithm for two processes. You should assume that each line is executed atomically, but note that lines 2 and 3 simulate addition being broken up into two atomic operations.

Integer $np \leftarrow 0, nq \leftarrow 0$	
p	q
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: $temp_p \leftarrow nq$	q2: $temp_q \leftarrow np$
p3: $np \leftarrow temp_p + 1$	q3: $nq \leftarrow temp_q + 1$
p4: await $nq = 0$ or $np \leq nq$	q4: await $np = 0$ or $nq < np$
p5: critical section	q5: critical section
p6: $np \leftarrow 0$	q6: $nq \leftarrow 0$

i. Which conditions would simultaneously have to hold for deadlock to occur in this solution?

[3 marks]

ii. With reference to your answer to part 1.a.i is it possible for this program to deadlock? Explain your answer.

[4 marks]

May 2015, Q1 (50/150) marks

Attempt 5: Peterson’s Algorithm

boolean wantp ← false, wantq ← false, integer last ← 1	
p	q
`loop_forever`	`loop_forever`
`p1:  non-critical section`	`q1:  non-critical section`
`p2:  wantp ← true`	`q2:  wantq ← true`
`P3:  last ← 1`	`q3:  last ← 2`
`p4:  await wantq = false or`	`q4:  await wantp = false or`
`              last = 2`	`              last = 1`
`P5:  critical section`	`q5:  critical section`
`p6:  wantp ← false`	`q6:  wantq ← false`

- This algorithm satisfies all of the required properties.

Mutual Exclusion (Informal)

- For mutual exclusion to hold we must show that (p5, q5, _,_,_) cannot be true.

- To get to that state one of the two must already be @5 (and stay there) the other must move from 4 to 5.
  - Processes are symmetric so it doesn’t matter which one.

- So let’s assume p is at p5: What is the value of *wantp*?
  - Only p can change *wantp*: at lines p2 and p6.
  - Since we’re at p5 the last of these two to execute must be p2, therefore *wantp* = true.

- Also note that last = 1 or last = 2 (by inspection of the program: these are the only values ever assigned).

Mutual Exclusion II (Informal)

`p4: await wantq = false or last = 2`	`q4:  await wantp = false or last = 1`

- We know: (p5,q4,true,_,1 or 2);

- We want to show that q cannot transition from q4 to q5 :
  - *wantp* = true (from previous slide) so q can only proceed if last = 1.
  - The only line of code that can set last = 1 is p3.

- When p executed p4 and transitioned to p5 either:
  - *wantq = false:*
    - That means q was at q1 or q2 (inspection of the program);
    - Therefore q must execute q3 setting last =2 before q4;
    - Since p remains at p5 it cannot execute last =1;
    - Therefore last = 2 and q stops at q4;
  - last = 2
    - If last = 2 at p4 and p remains in p5, p3 has not been executed to set last = 1
    - therefore last = 2 and q stops at q4.

Deadlock (Informal)

- For deadlock to occur both processes must be stuck at line 4:
  - `await wantq = false or last = 2`
  - `await wantp = false or last = 1`
  - **Both `wantp` and `wantq` are true (set at p2 and q2) and not changed**
  - **Both looping would require last = 2 and last = 1.**
  - **Contradiction: an integer cannot take 2 values. Therefore no deadlock.**

Peterson’s Algorithm

<table>
<tr><th colspan="2">boolean wantp ← false, wantq ← false, integer last ← 1</th></tr>
<tr><th>p</th><th>q</th></tr>
<tr><td><code>loop_forever</code></td><td><code>loop_forever</code></td></tr>
<tr><td><code>p1: non-critical section</code></td><td><code>q1: non-critical section</code></td></tr>
<tr><td><code>p2: wantp ← true</code></td><td><code>q2: wantq ← true</code></td></tr>
<tr><td><code>P3: last ← 1</code></td><td><code>q3: last ← 2</code></td></tr>
<tr><td><code>p4: await wantq = false or</code></td><td><code>q4: await wantp = false or</code></td></tr>
<tr><td><code>   last = 2</code></td><td><code>   last = 1</code></td></tr>
<tr><td><code>P5: critical section</code></td><td><code>q5: critical section</code></td></tr>
<tr><td><code>p6: wantp ← false</code></td><td><code>q6: wantq ← false</code></td></tr>
</table>

Freedom from Starvation (Informal)

- Show: If p is at p4 then eventually it will execute p5.
- If p is at p4 it can proceed unless: wantq = true and last = 1
- Where could q be?
  - if *wantq* = true, q is at q*3* or q*4* or q*5* (inspection of the code).
  - ***If q is at q3****:*
    - *It will execute last = 2*
    - *It will then remain at q4 until p executes.*
  - ***If q is at q4****:*
    - *It will continue to q5 as last = 1*
    - *Proceed to q6 (non-termination in cs)*
    - *Execute q6 (set wantq to false);*
    - *Execute q1 and either:*
      - *Terminate, p can proceed as wantq is false; or*
    - *Execute q2 setting wantq to true.*
    - *Execute q3 setting last = 2;*
    - *It will then remain at q4 until p executes.*
  - ***If q is at q5:***
    - *Follow from step 2 for q4.*

Properties as Temporal Logic

- Mutual Exclusion:
  - $\square \neg(p5 \wedge q5)$

- Freedom from Deadlock:
  - $\square ((p4 \wedge q4) \rightarrow \lozenge (p5 \vee q5))$

- Freedom from Starvation:
  - $\square (p4 \rightarrow \lozenge p5)$

Model Checking

- Given:
  - a Model: a formula to verify;
  - A program;

- Show that all states satisfy the model.

- Build the state diagram from the initial state and use search to show that no violating state exists.
  - It’s much quicker to get a computer to build the state diagrams than to do it yourself.
  - Although for big complex programs the state space can be too big.

- Area of research: how to efficiently show correctness.

Overview

- Introduction to Threads;
- Basic Threading in Java;
- The Critical Section Problem;
- Solutions to the Critical Section Problem:
  - Properties of a solution;
  - Proposed solutions;
- Proofs of (in)correctness.