UNIVERSITY OF PORTSMOUTH

Operating Systems and Internetworking

OSI – M30233 (OS Theme)

Week 3- Mutual Exclusion

Dr Tamer Elboghdadly

Mutual Exclusion

Mutual Exclusion

- Plan:
  - General introduction on *race conditions*, *critical sections*, and *mutual exclusion*.
  - Attempts at implementing mutual exclusion, culminating in Peterson's algorithm.
  - More <u>practical techniques for mutual exclusion</u> – *atomic instructions*, *semaphores*, and *monitors*.

An Example

- Recall the example from last week of a *counter variable*, updated once by *two* threads.
- More generally, each thread may update the counter repeatedly, doing other work in between:

```
repeat
   x = c
      c = x + 1
   <<do other work>>
forever;
```

- Here **x** is a <u>private</u> variable of the thread, and **c** is <u>shared</u> between both threads.

Critical Sections

- The set of steps - or algorithm - that must be executed to update a shared data structure is called a *critical section*.
- So a generalization of thread behaviour in our example is like this:

```
repeat
   <<critical section>>
   <<do other work>>
forever;
```

- Written naively like this, the programme as a whole very likely suffers a *race condition*.

OS Terminology

- ***Race condition*** - the outcome of two or more threads competing to modify shared data, where interference between threads results in erroneous results or a corrupted data structure.
- ***Critical section*** (or ***critical region***) – a code segment in a thread that updates or accesses shared data; only one involved thread should be in its critical section at any particular time.
- ***Mutual exclusion*** - methods to ensure only one thread does a particular activity, such as executing their critical section, at a time. i.e., all other threads are excluded from doing that activity.

[Watch videos on moodle]

UNIVERSITY OF PORTSMOUTH

Atomicity

- Mutual exclusion can be used to guarantee that critical sections execute *atomically*.
- In general an atom is an indivisible unit (e.g. of matter).
- In the context of concurrent programming, an atomic instruction, or statement, or code fragment is a piece of code that executes *as a whole, without interruption*, in such a way that no code in other threads can interfere with its execution.

Race Conditions are Evil

- Race conditions in concurrent programs are a particularly noxious kind of programming error.
- In their nature, they occur *non-deterministically*. Thus they give rise to *intermittent faults*.
  - A program might be tested thoroughly and never exhibit the race condition.
  - Then one day, out in the field, the program starts crashing, and nobody knows why.

Avoiding a Race Condition

- So, to avoid our race condition, we need to change our thread definition to something like this:

```
repeat
   … do something to begin mutual exclusion…
   <<critical section>>
   … do something to end mutual exclusion…
   <<do normal work>>
forever;
```

- The question is: *what should the "somethings" be?*

MUTUAL EXCLUSION (ME) USING SHARED VARIABLES

UNIVERSITY OF PORTSMOUTH

1-Naïve Attempt at ME (lock)

- Consider only two threads.
- A *new* shared Boolean variable, **lock**, initialized to false, specifies whether one thread is in its critical section:

```
repeat
   while(lock) do nothing; //means wait
   lock = true ;
   <<critical section>>
   lock = false ;
   <<do normal work>>
forever;
```

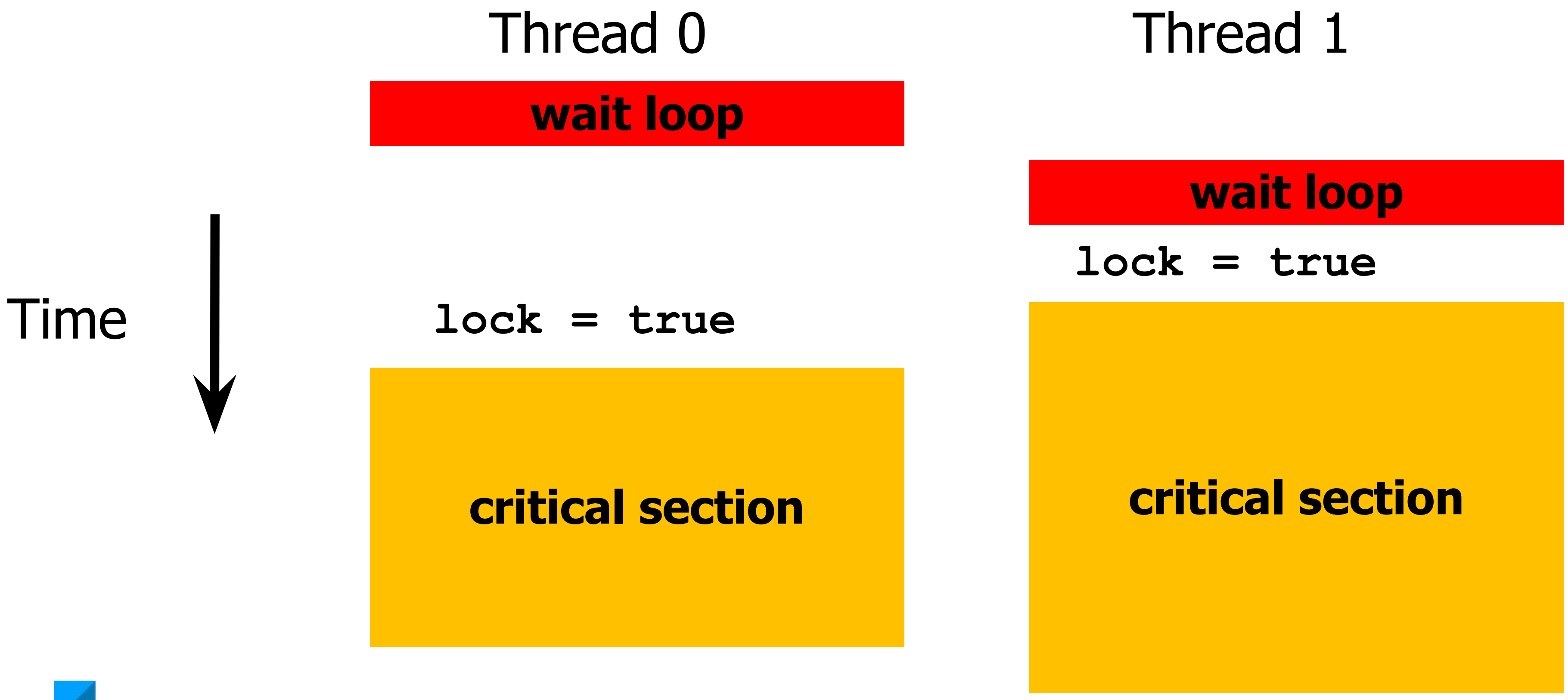
The Idea

- **lock** is initialized to false.
- When the first thread is ready to enter its critical section, its *wait loop* terminates immediately, lock is set true, and the critical section starts to execute.
- If a second thread wants to enter its critical section it will see lock is true, and its wait loop iterates until the *first* thread leaves its critical section and sets lock back to false.

Problem with Simple lock Variable

`lock = false`

Initially



Both threads in their critical sections!

The Problem

- This algorithm *does not* establish mutual exclusion
  - the two threads can be in their critical sections at the same time.
- If the second thread tests lock *in between* the while loop finishing in the first thread and that thread setting lock to true, the second thread also sees a false value for lock, and enters its critical section.
- We say that this solution does not guarantee *safety*.

A New Race Condition?

- In a sense what has happened is that our attempted algorithm for ME *itself* has a race condition – in the test and update of the new shared variable lock.
- Race conditions take many guises, and are often difficult to spot!

2-Another Attempt (turn)

- Again, consider only two threads.
- A *new* shared variable, **turn**, specifies whose turn it is to enter the critical section *next*.

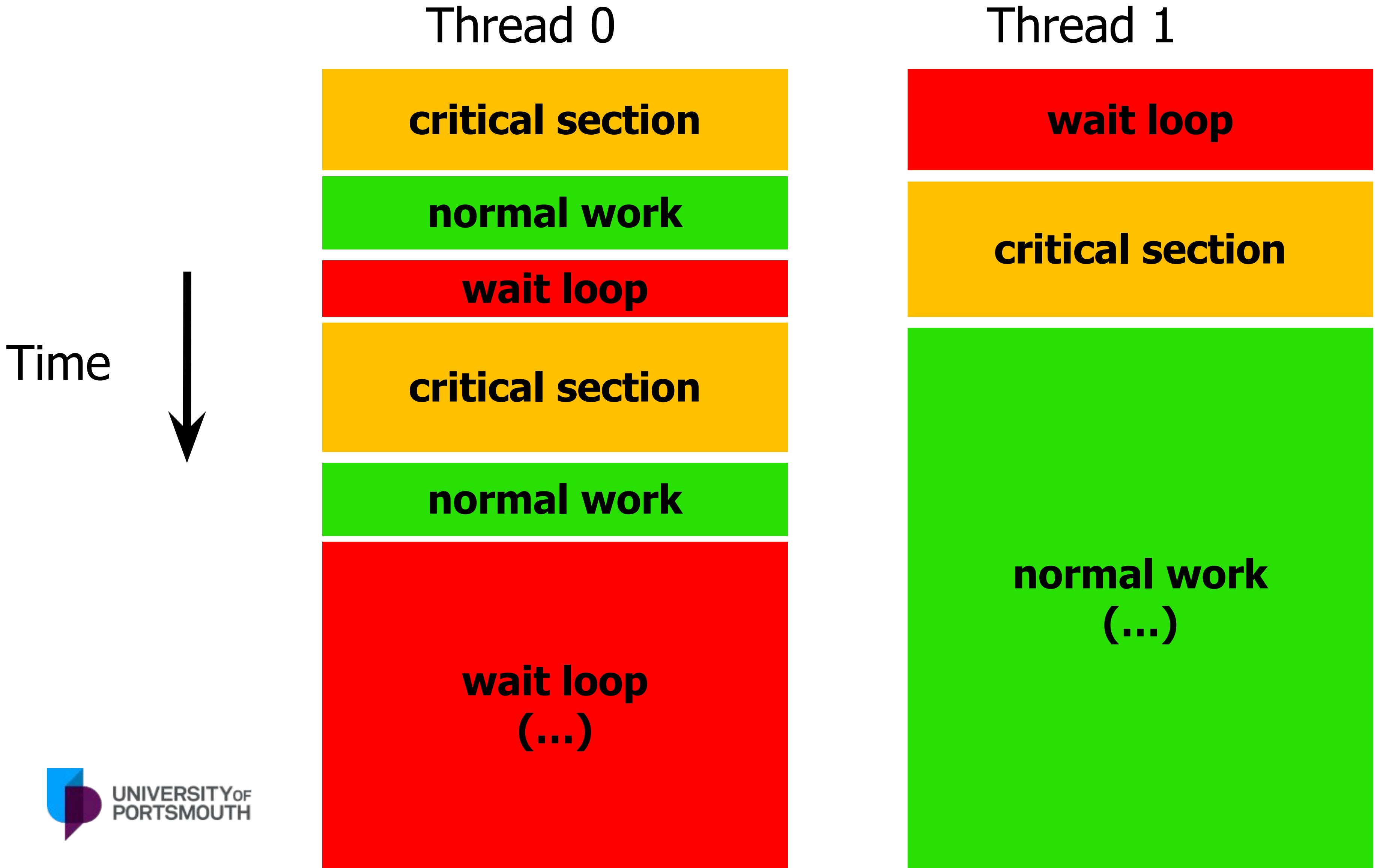
```
repeat
   while(turn != i) do nothing;
   <<critical section>>
   turn = j;
   <<do normal work>>
forever;
```

- **i** is identity of *this* thread; **j** is identity of other thread. Actual values will be 0 or 1

The Idea

- **turn** is initialized to, say, 0.
- Thread 0 is allowed to execute its critical section first.
  - If thread 1 tries to enter its own critical section before thread 0 has finished, it *waits in a loop*, doing nothing.
- When thread 0 leaves the critical section, **turn** is set to 1.
  - If thread 1 was waiting in a loop, the loop now ends;
  - Roles are now swapped; thread 0 cannot re-enter its critical region until thread 1 has had a turn.

Problem with Alternating Turns



Thread 0 must now wait indefinitely for Thread 1!

The Problem

- This algorithm *does* establish mutual exclusion
  - the two threads cannot be in their critical sections at the same time.
- But… it forces a strict 0, 1, 0, 1,… ordering of access to the shared data structure.
  - Suppose it is thread 1’s turn to enter the critical region, but thread 1 chooses to stay in the section **<<do other work>>** indefinitely.
  - Thread 1 may be *blocked forever*.
- We say that this solution guarantees *safety*, but not *progress*.

3- A Different Approach (interested)

- This time *two* shared variables, labeled **interested[0]** and **interested[1]**.
  - They are *Boolean* variables – either **true** or **false**.
  - **interested[i]** = **true** means thread **i** wants to enter its critical section.
- Thread **i**:

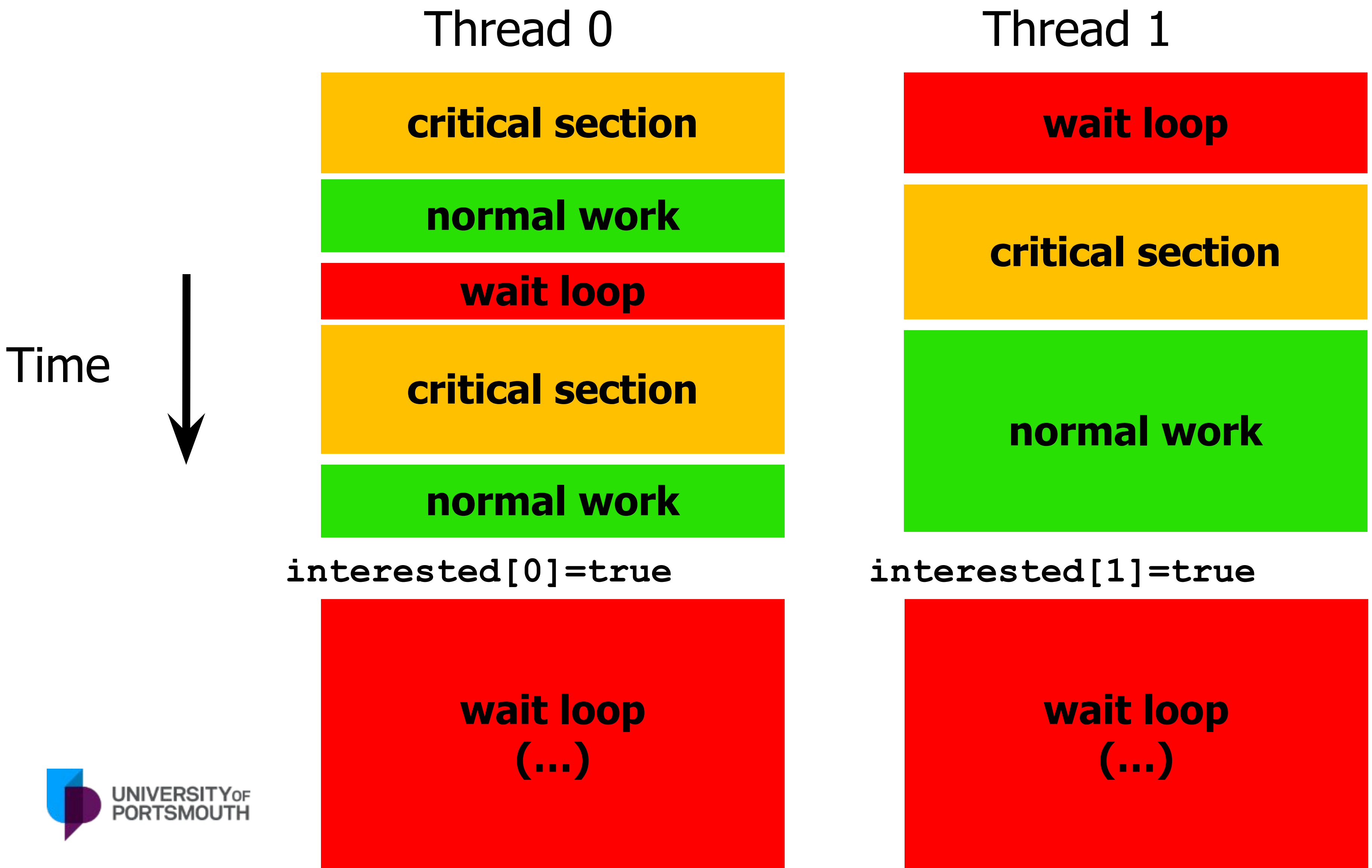
```
repeat
   interested[i] = true;
   while interested[j] do nothing;
   <<critical section>>
   interested[i] = false;
   <<do normal work>>
forever;
```

UNIVERSITY OF PORTSMOUTH

The Idea

- **interested[0]** and **interested[1]** are both initialized to **false**.
- A thread sets its **interested** variable when it wants to enter its critical region.
  - If the other thread has already set its own **interested** variable, it then waits in a loop until that thread has finished with the critical section
- When we leave our critical section, unset our **interested** variable, so the other thread can have access.

Problem with "Interested" Flags



Both threads now wait indefinitely!

The Problem

- Again this *does* establish mutual exclusion.
- But… what if the order of execution is such that the two threads reach their

  interested[i] = true;

  statements immediately after one another, *and before the other tests whether or not to loop*.
  - Both threads now loop (block) forever!
- Again, this solution guarantees *safety*, but not *progress*.
  - Because of a race condition in access to the variables that are supposed to implement mutual exclusion!

4- One Last Try (Peterson's algorithm)

- Combine the last two attempts – **but instead of switching turns after exiting the critical section, yield turn to the other thread *before* entering it**:

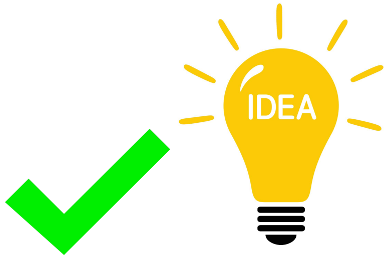
```
repeat
   interested[i] = true;
   turn = j;  /* "be nice" */
   while (interested[j] and turn=j) do nothing; //waiting
   <<critical section>>
   interested[i] = false;
   <<do normal work>>
forever;
```

UNIVERSITY OF PORTSMOUTH

The Problem

- Actually, there isn't one – it works!
  - The real problem is seeing why (see next slide)
- This is ***Peterson's algorithm*** for mutual exclusion.
  - Wasn't discovered until 1981.
  - Surprisingly late for such a fundamental and simple-*looking* algorithm.

The Idea



- If thread **i** tries to enter its critical section while **j** is already in *its* critical section, **interested[j]** will be true. **i** sets **turn** to **j**. So **i** waits until **j** unsets its interested flag.
- In general, if **i** reaches the wait loop while **j** is “interested”, it turns out that the *first* thread to set **turn** to *the other thread’s identity* actually gets to execute *its* critical region first.
  - If **i** was first, it may block *temporarily;* but **j** will set turn to **i** before hitting its own wait loop. Then **i** can go ahead and **j** is forced to wait.

PRACTICAL APPROACHES

Practical Mutual Exclusion

- Peterson's Algorithm is enlightening, but not particularly useful in practice
  - No easy generalization to more than **two threads**.
  - Relies on *busy waiting* – threads wait by looping. This can be *very* wasteful of CPU cycles.
- More realistic solutions
  - In truly parallel systems – specialized "atomic" instructions
  - In multitasking systems – incorporate support for synchronization into thread or process scheduling algorithms.

1- Hardware Support: Test and Set

- One kind of *atomic* instruction sometimes provided by hardware: *Test and Set Lock* (TSL)†.
  - Test and modify the content of a word *atomically*.
- Single *atomic* instruction that behaves *like*:

```
Boolean TestAndSet (Boolean &lock)
{
   Boolean initial = lock;
   lock = true;
   return initial;
}
```

†On X86, TSL can be implemented using for example the XCHG instruction.

Mutual Exclusion With TSL

- Using TSL, can do critical sections with the following approach.
  - Shared data: Boolean lock = false ; // initially
  - Thread definition:

```
repeat
   while (TestAndSet(lock)) do nothing ;
   << critical section >>
   lock = false;
   << do normal work >>
forever;
```

UNIVERSITY OF PORTSMOUTH

Comparison

Wrong!

```
repeat
   while(lock) do nothing;
   lock = true ;
   <<critical section>>
   lock = false ;
   <<do normal work>>
forever;
```

Correct!

```
repeat
    while (TestAndSet(lock)) ;
    << critical section >>
    lock = false;
    << do normal work >>
forever;
```

Programming Abstractions

- Parallel computers can use TSL and similar instructions to implement mutual exclusion and other kinds of synchronization.
- Still depends on *busy waiting*, and not appropriate in multi-tasking environments (because it wastes computer cycles).
- Need higher-level abstractions for synchronization, that can be implemented *either* by low-level instructions like TSL, *or* OS scheduling algorithms.

2- OS Support: Semaphores

- A semaphore *S* is an <u>integer variable</u> that may be accessed only using two operations *P* and *V*, with pseudocode *essentially* like atomic:

  P(S) : S = S - 1; // decrease S

  V(S) : S = S + 1; // increase S

  Remember
  V for increase

  but, with the special property that *a semaphore's value never goes below zero!*

- A thread that tries to decrease a *zero* semaphore with *P* must *wait* (block) until another thread raises the semaphore using *V*. [Watch videos on moodle]

Mutual Exclusion With Semaphore

- Shared variable:
  - S: semaphore = 1;
- Thread i, for i=1,2,3, … :

```
repeat
   P(S);
   <<critical section>>
   V(S);
   <<do normal stuff>>
forever;
```

Evaluation of Semaphores

- Semaphores provided one of the first high-level synchronization abstractions.
  - Can be implemented efficiently on multiprocessors or in multi-tasking operating systems.
- Problems:
  - Programming with semaphore is error prone …

3- Java Synchronized Methods

- Java and several other modern programming languages use a version of the *monitor* idea.
- Certain methods in a Java class can be declared to be *synchronized*.
- If two threads try to call synchronized methods *on the same object* at the same time, one blocks until the other has completed.
  - For *static* methods: if two threads try to call synchronized methods *in the same class*, one blocks.

UNIVERSITY OF PORTSMOUTH

Mutual Exclusion With Java

- Shared variable:
  - object with one or more synchronized methods
- Thread i, for i=1,2,3, … :

```
repeat
   object.mySynchronizedMethod()
   <<do normal stuff>>
forever;
```

Declaring a Synchronized Method

```
class MyClass {

    synchronized void mySynchronizedMethod(){
        <<critical section>>
    }
    …
}
```

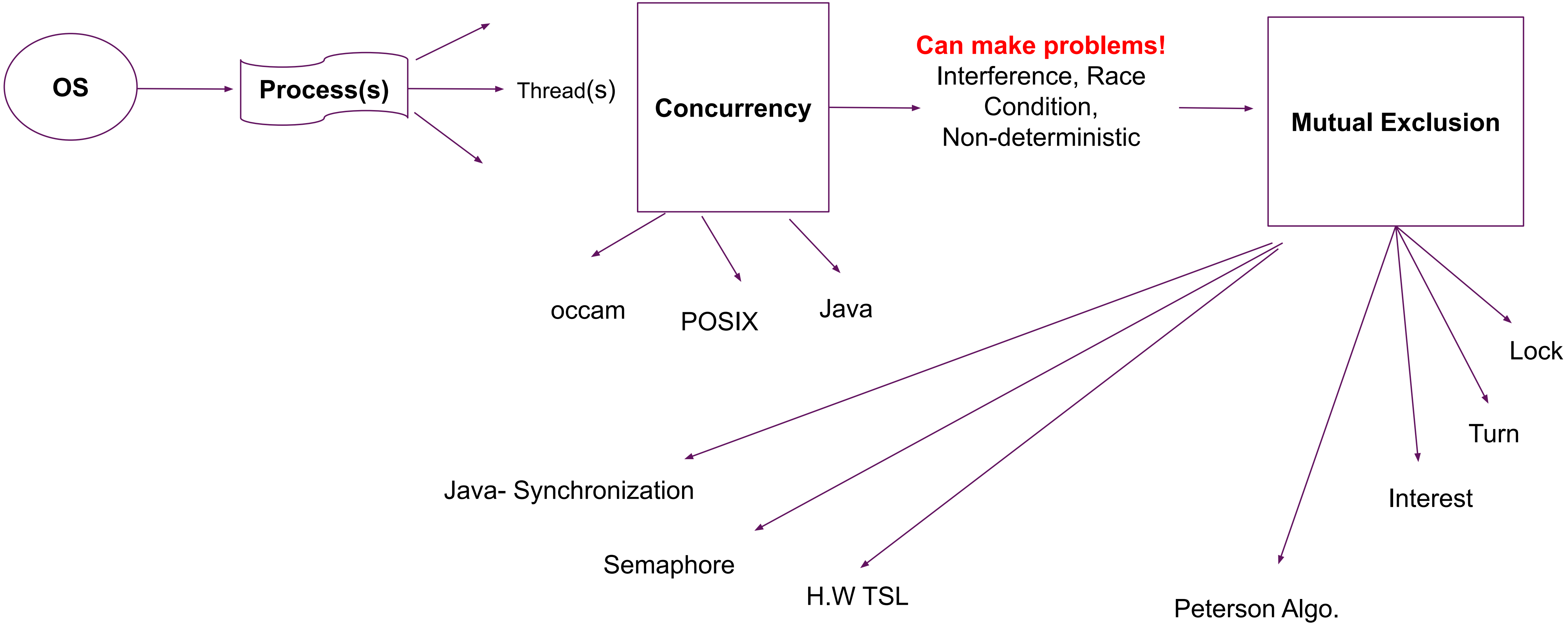
Sharable Java Counter

```
public class Counter {

    int count ;

    synchronized void increment() {
        int x = count ;
        count = x + 1 ;
    }
}
```

Summary

- Motivated the study of *mutual exclusion* by considering *race conditions* that may emerge in concurrent programmes.
- Developed *Peterson's algorithm*, a theoretically interesting protocol for mutual exclusion.
- Briefly described several more practical approaches to mutual exclusion, culminating with Java-style monitors.
- Next lecture: *General Synchronization, and Deadlock*

Recap



UNIVERSITY OF PORTSMOUTH

Further Reading

- Andrew S. Tanenbaum, "*Modern Operating Systems*", 4th Edition, Pearson, 2014 (MOS)
- More advanced material on concurrency:
  - Gadi Taubenfeld, *"Synchronization Algorithms and Concurrent Programming"* , Pearson, 2006
  - Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell, "*Pthreads Programming*",O'Reilly, 1996.
  - Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea "*Java Concurrency in Practice*", Addison-Wesley, 2006

APPENDIX: A NOTE RELATING TO THIS WEEK'S LAB

UNIVERSITY OF PORTSMOUTH

Minimum count from “fast race”

- In last week’s lecture we saw that for two threads both doing a single update of a counter, the minimum final value of the counter was 1 (whereas we might have expected 2).
- In some of the examples in this week’s lab script, two threads *each* try to update a counter 1 billion times. What is the *minimum* value for the final count in this case?
  - If you said 1 billion, that would have also been my first guess, but it is dramatically wrong!

UNIVERSITY OF PORTSMOUTH

Extreme worst case scheduling

Thread A		c	Thread B	
x = c (0)	//Starting 1	0	y = c (0)	
		1	c = y+1 (1)	
		2	y = c (1)	
			c = y+1 (2)	
		:	…	
		999,999,999	c = y+1 (999,999,999)	2
3				
c = x+1 (1)		1	y = c (1)	4
x = c (1)				
c = x+1 (2)		2		
…		:		
5 c = x+1 (1,000,000,000)		1,000,000,000		
		2	c = y+1 (2)	6

Remarks

- So the worst schedule is:
  - A reads c = 0 ;
  - B updates c 999,999,999 times to 999,999,999
  - A completes it's first update to c = 1
  - B reads c = 1
  - A updates c 999,999,999 times to 1,000,000,000
  - B completes it's last update to c = 2
- It looks as if, in general, if two threads without mutex attempt $N \geq 2$ increments, final value of c is anywhere between 2 and $2N$.

1
2
3
4
5
6

UNIVERSITY OF PORTSMOUTH

Questions?