UNIVERSITY OF PORTSMOUTH

Operating Systems and Internetworking

OSI – M30233 (OS Theme)

Week 2- Concurrency

Dr Tamer Elboghdadly

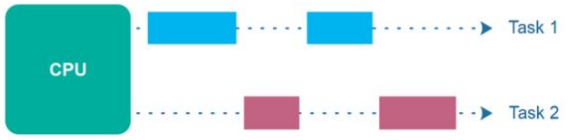
Concurrency

Plan of the Lecture

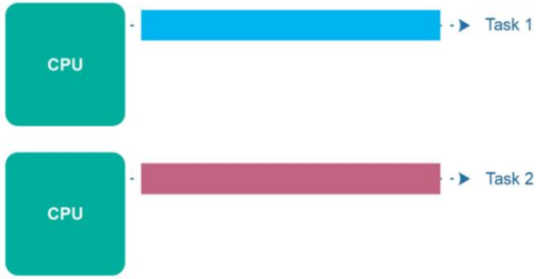
- Motivate the consideration of **concurrency** in operating systems and parallel systems.
- Basics of creating **threads** in various programming languages.
- Start to introduce the issues of concurrent programming – problems like **non-determinism**.

What is Concurrency?

- **In computer science:**
  - A *sequential system* is one where computations - or parts of a computation - are executed to completion, one after the other.
  - A *concurrent system* is one where two or more computations are executing – literally or effectively – “at the same time”.



- **A concurrent system is almost the same as a parallel system**
  - We sometimes reserve the latter for the case where computations are literally proceeding at the same time.

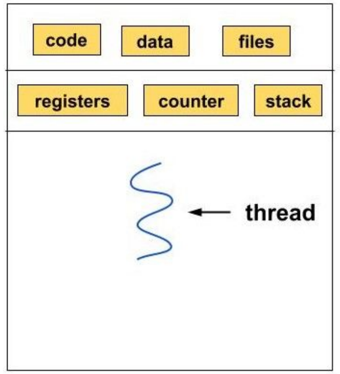


Scope of Concurrency

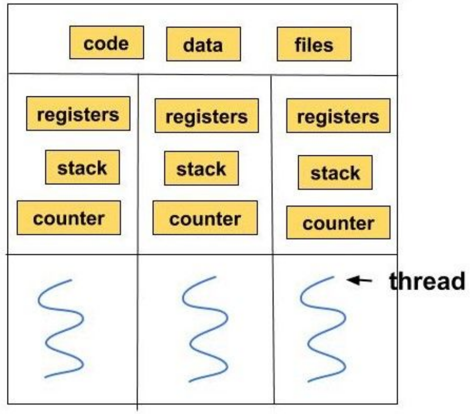
- Concurrency and associated issues arise in:
  - Multi-tasking operating systems, where many processes are running at once.
  - Individual applications like Web servers, that must be processing many “requests” at the same time.
  - Multicore processors where a single application is running across more than one core.
  - Parallel computers in general.
  - Distributed systems in general.
  - etc
- They are pervasive.

Processes and Threads

- A *thread* or *thread of control* is a specific sequence of instructions defined by some program, or by some section of a program.
  - Instruction sequences from one thread may run in parallel with, or be interleaved in an unpredictable way with, sequences from other threads.
- Any *process* has one or more threads.
  - Processes also have additional structure associated with them such as address spaces – processes will be discussed in detail in later lectures.
  - In most of this lecture we focus on threads themselves.



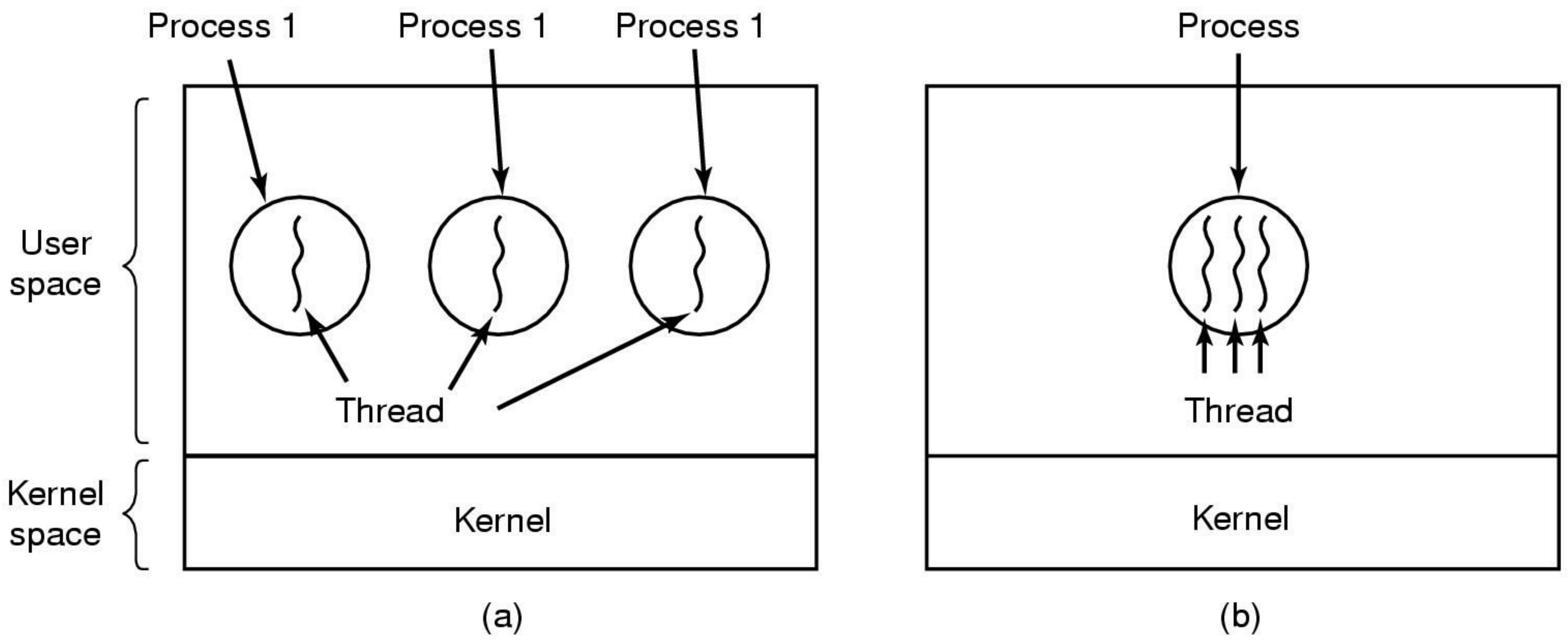
Single-threaded process



Multi-threaded process

Processes and Threads

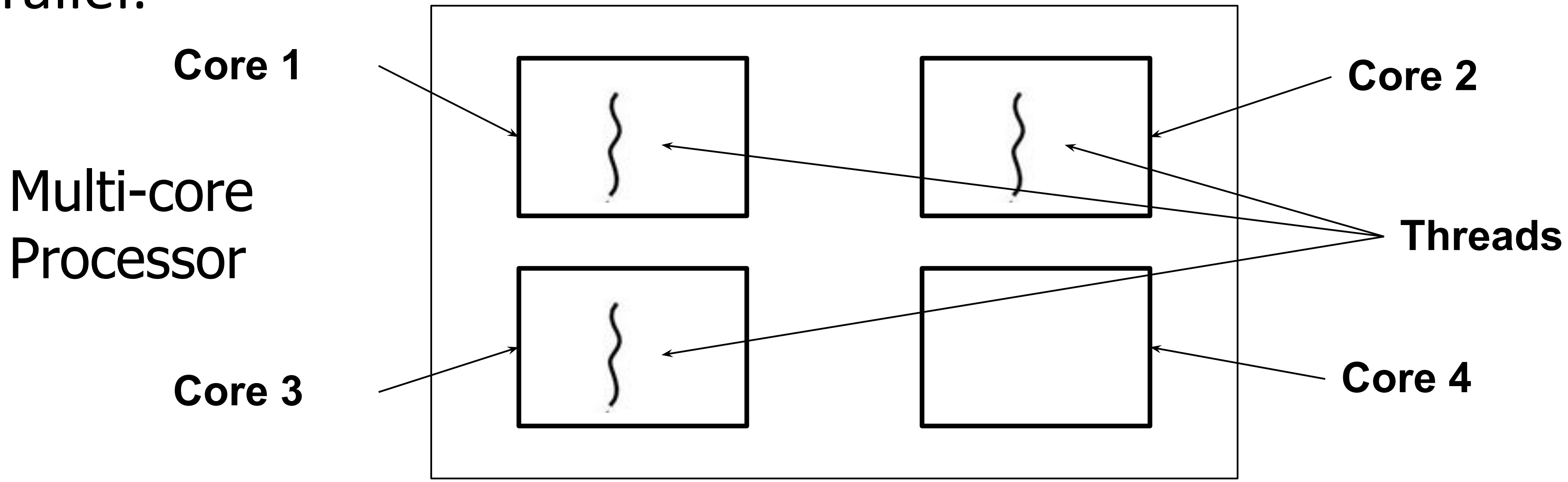
- Every process has at least one control flow within a single “address space”.
- A process may also have multiple control flows within the same address space†.



†Tanenbaum, MOS, Fig 2-11

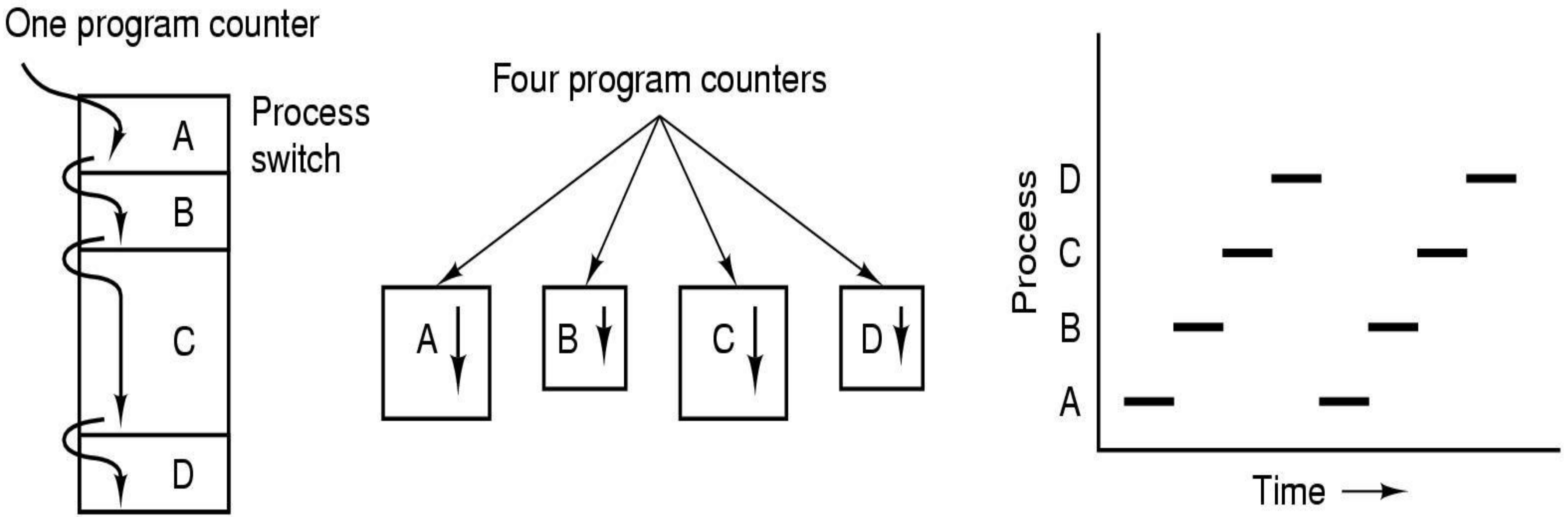
Parallelism vs Multitasking

- On a *multi-core processor*, threads may run on different cores, truly in parallel:



- But often, when we are discussing concurrency in operating systems, multiple threads *share the same* "core" by *multitasking*.

Multitasking



(a) (b) (c)

- Mechanisms for multitasking will be discussed in a later lecture.

PROGRAMMING WITH THREADS

Concurrent Programming

- How do we start to write concurrent program?
- Historically, many have promoted new programming languages with special “parallel” constructs.
- Today it is more common to use *thread libraries*.

Example from occam Language

- Popular in the UK in the 1980s thru' 90s†:

```
PAR
  SEQ
    x = 23
    print x
  SEQ
    y = 42
    print y
```

  - **PAR** means "do following in parallel", **SEQ** means "do following in sequence".
  - Code above runs blue and red threads *in parallel*.

†occam didn't really have a **print** statement – we used some license here!

UNIVERSITY OF PORTSMOUTH

POSIX Threads

- A low-level library for thread programming, often used from the *C* programming language†.
  - The code for a new thread is defined inside some C *function*.
  - A parent thread (e.g. “main program”) calls the library function **pthread_create**, passing it a *pointer to a function* with the code for the new thread.
  - A parent thread may create any number of threads, and children can create their own children, etc.

[POSIX stands for Portable Operating System Interface]

†C is used in implementing many OSs.

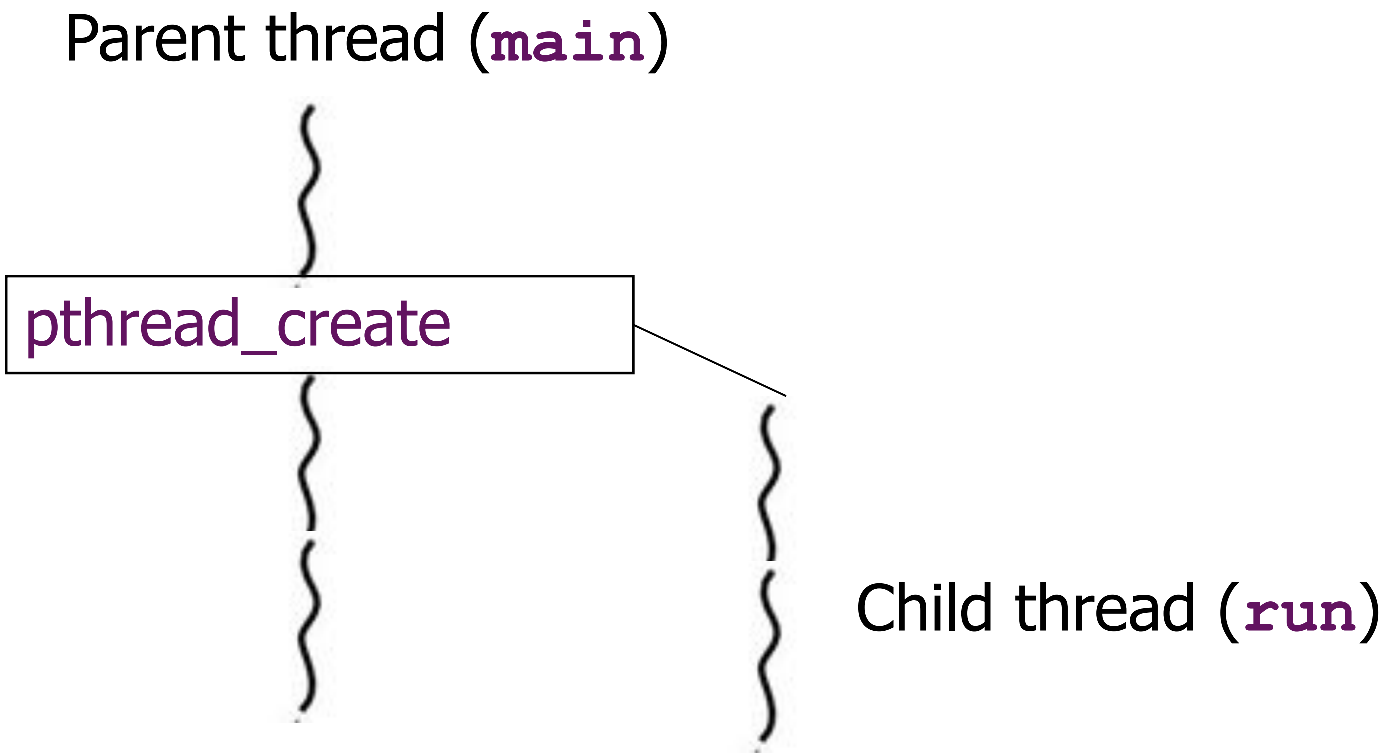
Creating a POSIX Thread

- Again, with some license:

```
int main(int argc, char* argv []) {
    pthread_t thread ;
    pthread_create(&thread, NULL, run, NULL) ;
    x = 23 ;
    print x ;
}

void* run(void *) {
    y = 42 ;
    print y ;
}
```

Visualization of Thread Creation



Concurrency in Java

- *Java* is a kind of half-way house between older, full-blown parallel languages like occam, and languages like C that "just" do threads using libraries.
- It doesn't contain explicit *parallel* constructs, but many features of the language have been carefully designed to *support* concurrency.
  - Modifiers like **synchronized**, **volatile** on declarations (see later lectures and labs)
  - **synchronized** construct
  - All carefully integrated with the *Java memory model*.

Java Threads

- Thread creation in Java is similar in style to POSIX threads, except it follows an object-oriented pattern.
- The “run” method is defined in a class extending **java.lang.Thread**.
- Create an object of this class, then call the **start** method to begin the thread.

Creating a Java Thread

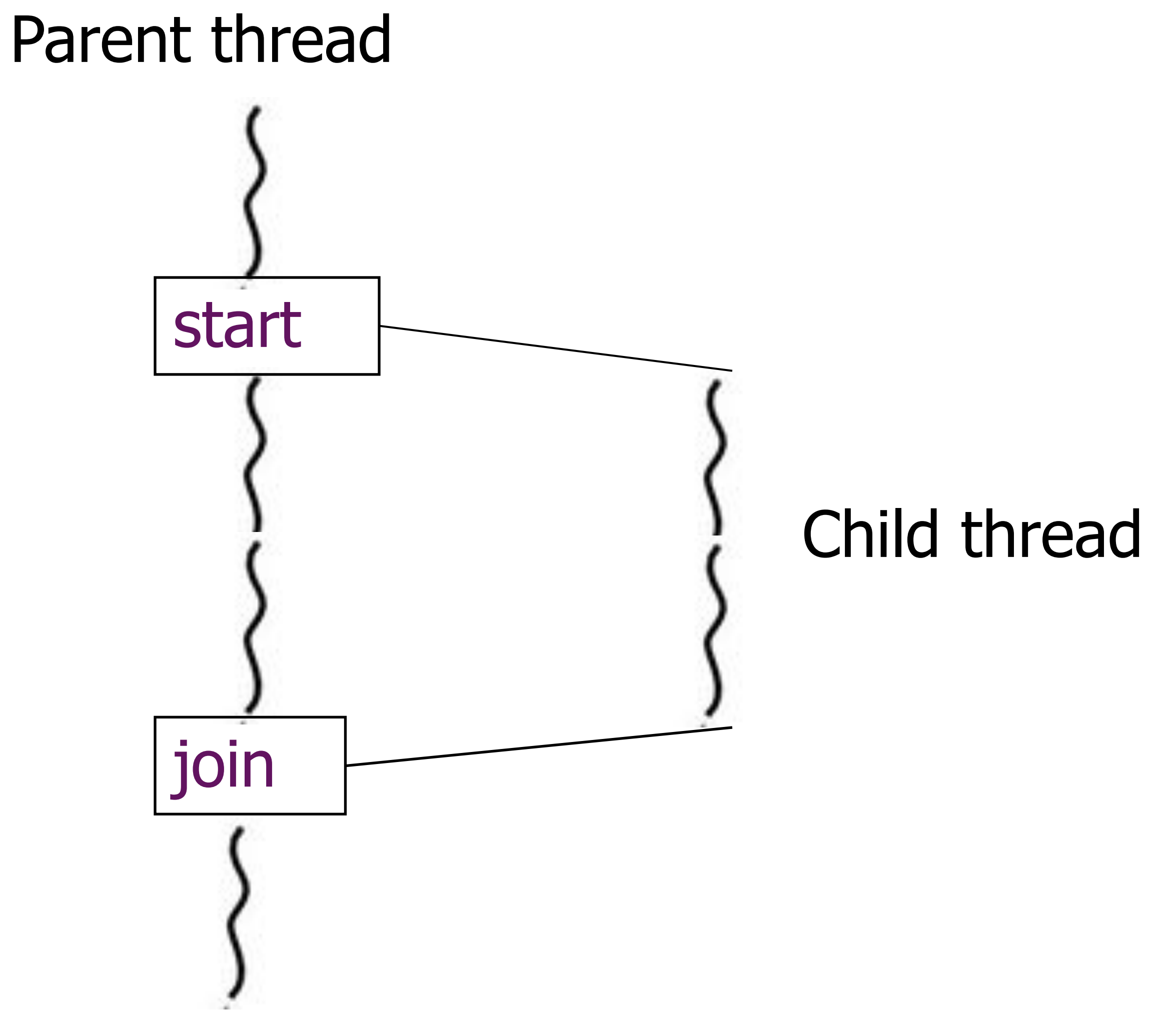
```
public static void main(String[] args) {
    MyThread thread = new MyThread();    //creating thread
    thread.start();                     //begin running thread B
    int x = 23;                          //thread A execute in parallel with thread B
    System.out.println (x);
    thread.join();
}

Public static class MyThread extends Thread {

    public void run() {                  //work for thread B
        int y = 42;
        System.out.println (y);
    }
}
```

[Try some real examples here]

Visualization of Java Thread



Join

- Call to `join` here is optional
- The **join** method waits (blocks) until the child thread has completed.
  - This is a simple example of *synchronization* between threads.
  - POSIX threads have an equivalent function called **pthread_join**.

NON-DETERMINISM

UNIVERSITY OF PORTSMOUTH

A Multi-threaded Program

- Start by considering this extremely trivial program (pseudocode) with two threads, *A* and *B*:

Thread *A*:

```
x = 23
print x
```

Thread *B*:

```
y = 42
print y
```

- Each thread assigns a value to a local variable, then prints it out.

Orders of Execution

- For sake of illustration, let’s make simplifying assumptions:
  - individual statements like "**x = 23**" and "**print y**" happen *instantaneously*
  - no two statements are executed at *exactly* the same time.
- This means that statements will actually be executed in a well-defined time order
  - But there are *many* possible orders of execution even for our trivial program.

UNIVERSITY OF PORTSMOUTH

Possible Orders

```
x = 23
print x
y = 42
print y
```

```
x = 23
y = 42
print x
print y
```

```
x = 23
y = 42
print y
print x
```

```
y = 42
x = 23
print x
print y
```

```
y = 42
x = 23
print y
print x
```

```
y = 42
print y
x = 23
print x
```

- In 1st, 2nd, and 4th orderings, program prints 23 then 42; in others it prints 42 then 23.

UNIVERSITY OF PORTSMOUTH

Lessons

- Trivially simple example illustrates two general features of concurrent programs:
  1. Even simple concurrent programs can have *many* possible orders of execution:
     - The number of possible orderings grows *exponentially* with program size.
     - This makes concurrent programs hard to design and debug, because there are many possibilities to consider.
  2. Concurrent programs are often ***non-deterministic*****:**
     - ***different orders of execution may lead to different outcomes.***

A Shared Counter

- Things become more complex when different threads share access to *the same* variable:

Thread *A*:

```
x = c
c = x + 1
```

Thread *B*:

```
y = c
c = y + 1
```

- Here each thread is trying to increase the value of variable **c** by 1.
  - "Local variables" **x** and **y** may be registers.

Possible Execution Orders

- What is the final value of `c` ?

```
x = c
c = x + 1
y = c
c = y + 1
```

```
x = c
y = c
c = x + 1
c = y + 1
```

```
x = c
y = c
c = y + 1
c = x + 1
```

```
y = c
x = c
c = x + 1
c = y + 1
```

```
y = c
x = c
c = y + 1
c = x + 1
```

```
y = c
c = y + 1
x = c
c = x + 1
```

- Assume the initial value of c is 0.

Two cases

```
x = c
c = x + 1
y = c
c = y + 1
```

c	x	y	
0	-	-	Initial state
0	0	-	
1	0	-	
1	0	1	
2	0	1	Final state

```
x = c
y = c
c = x + 1
c = y + 1
```

c	x	y	
0	-	-	Initial state
0	0	-	
0	0	0	
1	0	0	
1	0	0	Final state

UNIVERSITY OF PORTSMOUTH

Interference

- This is a more serious case of *non-determinism*.
  - The programmer may have *reasonably expected* that each thread increments the variable **c** by 1
  - the combined outcome of both threads would be to increment **c** by 2.
- This kind of unpredictable behaviour, when concurrent threads adversely affect one another's behaviour, is called *interference*.
  - Illustrated for a very simple "data structure" – a counter.
  - Similar more serious problems arise with shared access to more complex data structures (see appendix to this lecture).

UNIVERSITY OF PORTSMOUTH

Race Conditions

- You will also see situations like this referred to as ***race conditions***
  - because the outcome depends which thread gets to a particular point of its programme first.
- For our purposes, “interference” and “race conditions” are essentially the same thing
  - though race conditions also occur in distributed systems, without shared variables.

UNIVERSITY OF PORTSMOUTH

Avoiding Interference

- One way to avoid this kind of race condition between threads is just to make sure that threads *never have any variables in common*.
  - This is essentially what happens with *processes* – each process has a completely independent *address space*, and no variables are shared.
- But in the underlying operating system, which is responsible for *scheduling* processes, and in (say) multithreaded server programs, this solution is too restrictive.

UNIVERSITY OF PORTSMOUTH

Critical Sections

- In the counter example, results only correct if order of execution implies *no overlap* between execution of sections:

```
x = c
c = x + 1
```

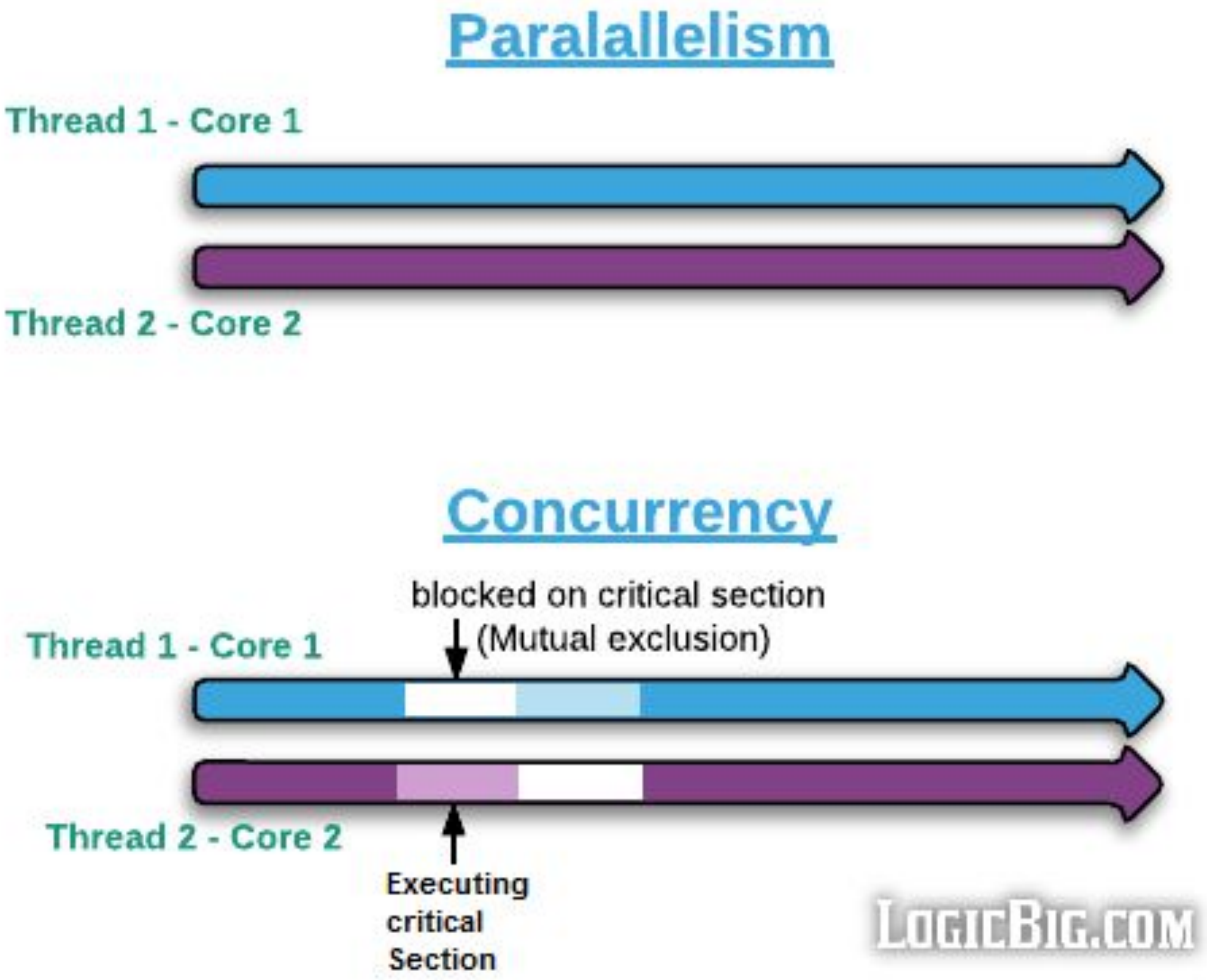
and

```
y = c
c = y + 1
```

- In general sections of code that must not overlap are called *critical sections*.
  - More usually, short sections of a whole program.

Mutual Exclusion

- Techniques to ensure that critical sections do *not* overlap during execution of a concurrent program are called *mutual exclusion*..
- Mutual exclusion is another simple example of *synchronization* between threads (like *join*, earlier).
- The next lecture covers mutual exclusion in detail



UNIVERSITY OF PORTSMOUTH

Summary

- We motivated the discussion of *concurrency*, in various contexts, some related to operating systems.
- Some basic problems of concurrency – including non-determinism and race conditions – were introduced.
- We described the creation of threads in various programming languages.
- We introduced Critical Sections in concurrency code which can make a race condition and can be resolved Mutual Exclusion.

- *Next lecture*: Mutual Exclusion

Further Reading

- Andrew S. Tanenbaum, “*Modern Operating Systems*”, 4^{th} 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

UNIVERSITY OF PORTSMOUTH

Questions?