ECM2414 – Software Development

Concurrency I

Dr. Yulei Wu, T1:W2 - L1

Department of Computer Science
College of Engineering, Mathematics and Physical Sciences
University of Exeter

Today's lecture

- Concurrency
  - What is it? Why use it?
- Multi-threading
  - As separate from multi-tasking
- Creating, starting, and stopping a thread
- Issues with multi-threading

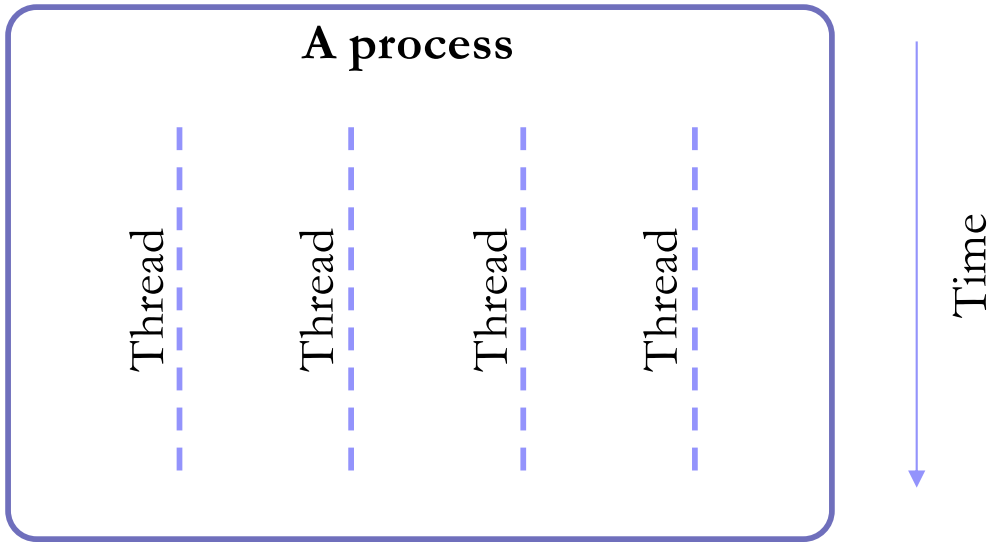
Concurrency

- To be able to perform multiple **processes** at the same time – including the potential interaction between them
  - Multiple tasks at the same time sometimes called ‘parallelism’

- Why are we interested in concurrency?
  - Allowing programs to interact with other systems/users
  - Multi-core systems

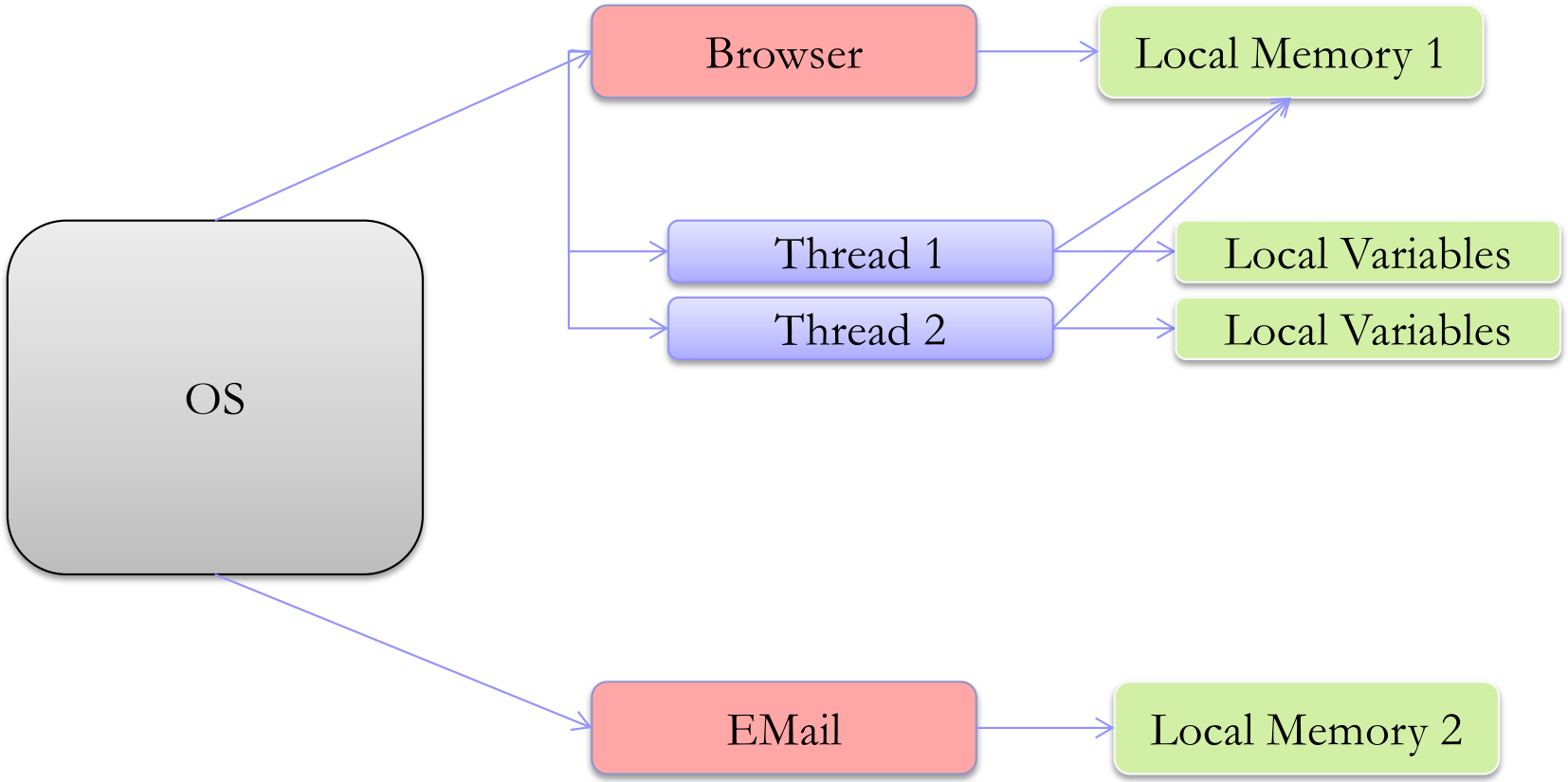
Threads

- Can be treated as lightweight processes.
- Threads exist within a process



- Why thread? Why not have separate programs?
  - Shared memory space

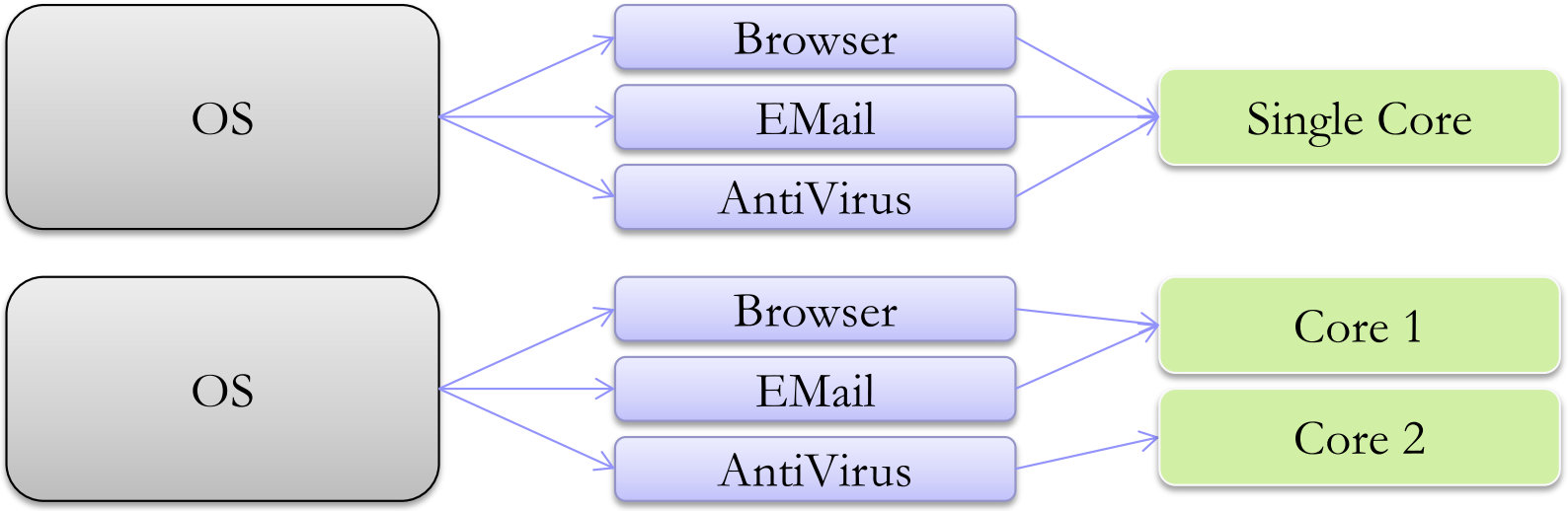
Multi-threading



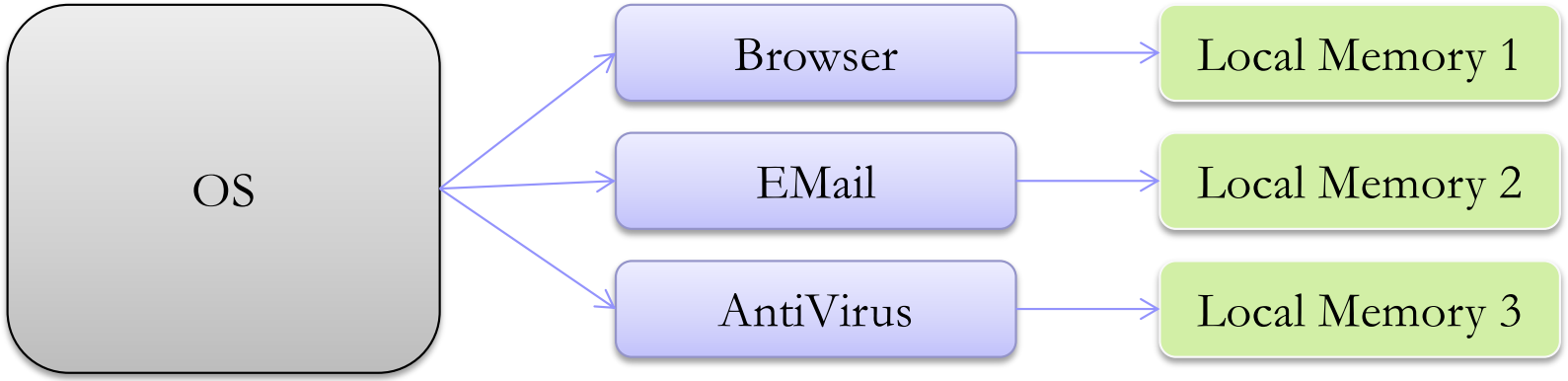
- The OS can run threads in parallel
  - Virtually, on a single processor
  - Actually on a multi-core processor

Multi-tasking

- Processing

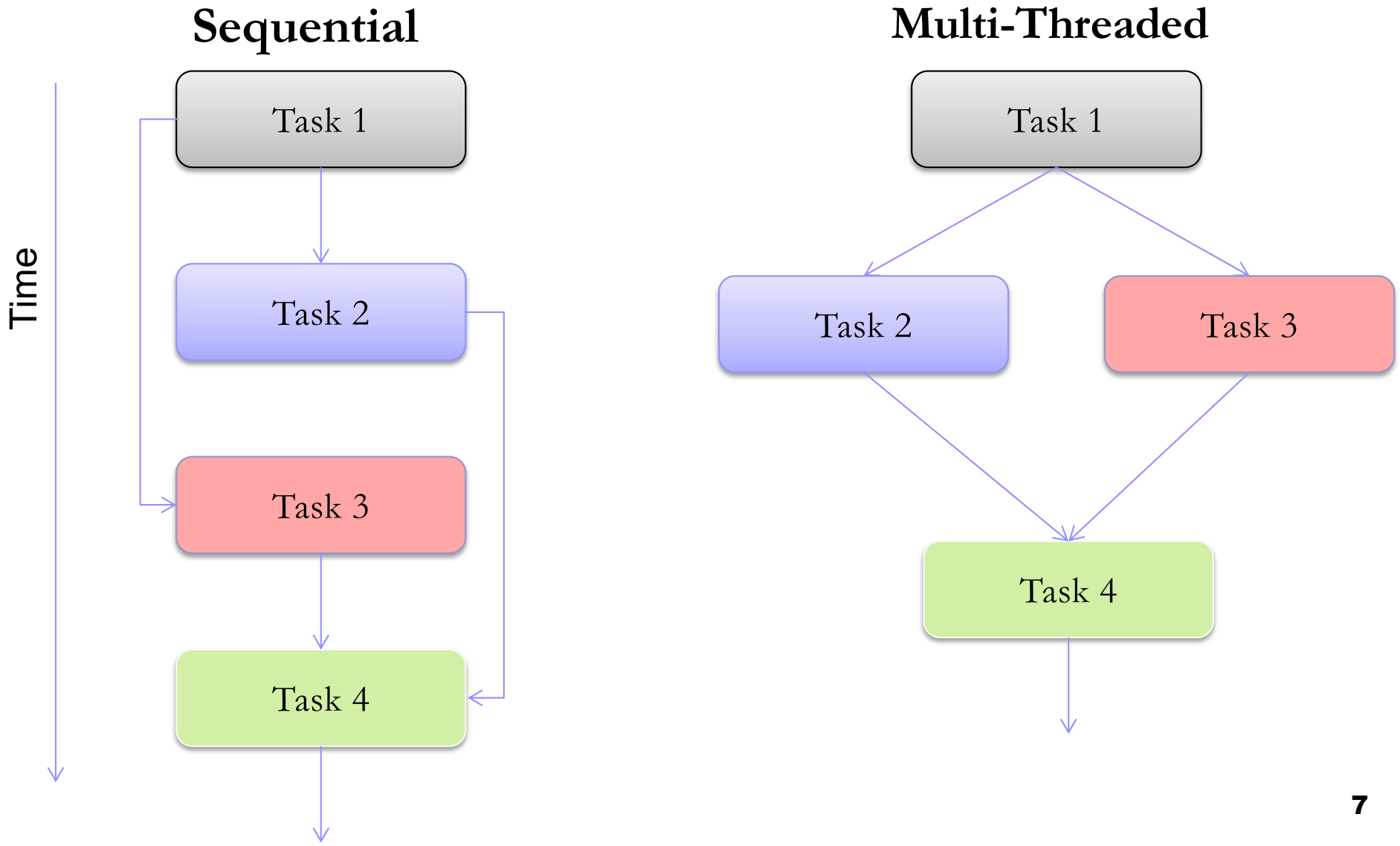


- Memory Space



New things to worry about - Nondeterminism

- What might be the problem here?



New things to worry about - Liveness

- Previously we've worried about safety, i.e. writing our code so that it performs ***correctly***

- With threads we now need to also worry about ***liveness***
  - Not just concerned that code doesn't perform correctly - i.e. that bad things don't happen
  - Also concerned code performs **correctly in time** - i.e. that good things do happen eventually
    - This second requirement means we need to concern ourselves what level of ***eventually*** is acceptable...

Threads in Java

- Can be created in two ways in Java
  - Using (and extending) the `Thread` class
  - Using the `Runnable` interface

```
public class HelloThread extends Thread {

   public void run () {
            System.out.println ("Hello from a thread!");
   }

   public static void main ( String args []) {
            (new HelloThread()).start();
   }

}
```

- Extending is simple to implement, but has drawbacks

- What might this be?

```
public class HelloRunnable implements Runnable {

   public void run () {
            System.out.println ("Hello from a thread!");
   }

   public static void main ( String args []) {
            (new Thread (new HelloRunnable ())).start();
   }

}
```

Thread lifecycle methods

```
Package java.lang
public class Thread implements Runnable {
        public void start();
        public void run();
        public void stop(); //deprecated (internal race condition)
        public void resume(); // deprecated
        public void suspend(); // deprecated
        public static void sleep(long millis);
        public static void sleep(long millis, int nanos);
        public boolean isAlive();
        public void interrupt();
        public boolean isInterrupted();
        public static boolean interrupted();
        public void join();
}
```

How do I stop a thread?

Setting a flag:

```
public void run () {
   while (!done ){
        //...
   }
}
```

- Loop will exit (and thread will stop) on the next iteration after done is set to true

Interrupting:

```
public void run() {
   while (!isInterrupted()){
        //...
   }
}
```

- As above, when `isInterrupted()` is called for this thread, it will stop
- Except that if the thread is sleeping /waiting will throw InterruptedException and immediately return from `run()`

Race conditions

- Data synchronisation between threads can be an issue
- When two threads try to access/change data at the same time it's known as a **race condition**
- Two volunteers for a game.

Race condition with threads

```
public class MyCounter {

        private int count = 0;

        public void addTwo() {
                count = count + 2;
        }

        public void subtractTwo() {
                count = count - 2;
        }

        public int countValue () {
                return count ;
        }

}
```

- Thread 1: (addTwo) Retrieve counter.
- Thread 2: (subtractTwo) Retrieve counter.

- Thread 1: Change retrieved value; counter value is 2.
- Thread 2: Change retrieved value; result is -2.

- Thread 1: Store result in counter (counter is 2).
- Thread 2: Store result in counter (counter is now -2).

Synchronisation

- Simply add the synchronize keyword to the methods in the class
  - When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the **same object block suspend execution** until the first thread is done with the object.

```
public class MyCounter {
        private int count = 0;
        public synchronized void addTwo() {
                count = count + 2;
        }
        public synchronized void subtractTwo() {
                count = count - 2;
        }
        public synchronized int countValue() {
                return count ;
        }
}
```

Statement synchronisation

- Also possible to synchronise a set of statements, not necessarily a whole method:

```
public void doSomeSynchronisedStuff() {

        // everything out here before is not synchronized
        synchronized (this){
                // everything in here is synchronised
                ...
        }
        // everything out here afterwards is not synchronised
either

}
```

Atomic actions

- In programming, atomic actions are self-contained, they cannot be stopped in the middle
  - They either happen or they don't

- Atomic access can help here:
  - Java primitive data types: byte, short, int, long, float, double, boolean, char
  - Reads and writes are atomic for most primitive variables (all types except long and double).
  - Reads and writes are atomic for *all* variables declared **volatile** (*including* long and double variables).

Summary

- Explored concurrency
  - Why concurrency is required
  - Multi-tasking vs multi-threading
- Threads as a mechanism for concurrency in Java
  - Liveness
  - Race conditions
  - Synchronization
  - Atomic access
- **Next time: more on threading**