Concurrent Programming (Part II)
Lecture 2: Understanding Concurrency

Kevin Bryson

K.Bryson@cs.ucl.ac.uk

Room: 8.04

Overview of Lecture

- In the previous lecture we gave an informal overview of concurrency including:
  - Parallelism leading to real-time speed-up.
  - Distributed computing.
  - Problem domains that are naturally concurrent and would be difficult to model sequentially.
  - Efficient CPU usage with multi-tasking.

- In this lecture we provide a more formal definition of the concurrency abstraction before moving onto its concrete implementation in terms of Java Threads.

- A lot of the ‘Concurrency Abstraction’ ideas are from: “Principles of Concurrent and Distributed Programming” by M. Ben-Ari.

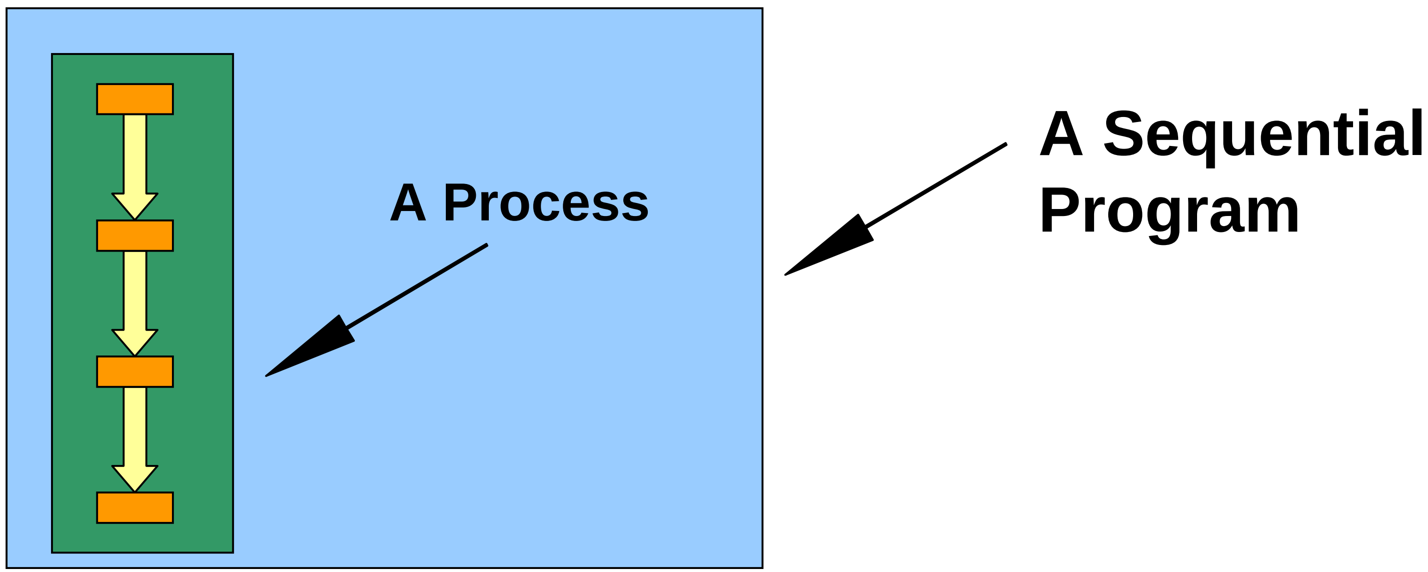
Definition of the Concurrency Abstraction

- The concurrency abstraction is based on having a number of ***processes***, each consisting of a ***totally ordered*** sequences of ***atomic actions***.
- The abstraction itself models the overall system by ***interleaving*** the processing of the ***atomic actions*** of the processes.
- We also abstract away the concept of time units …

- Clearly we need to define these terms.

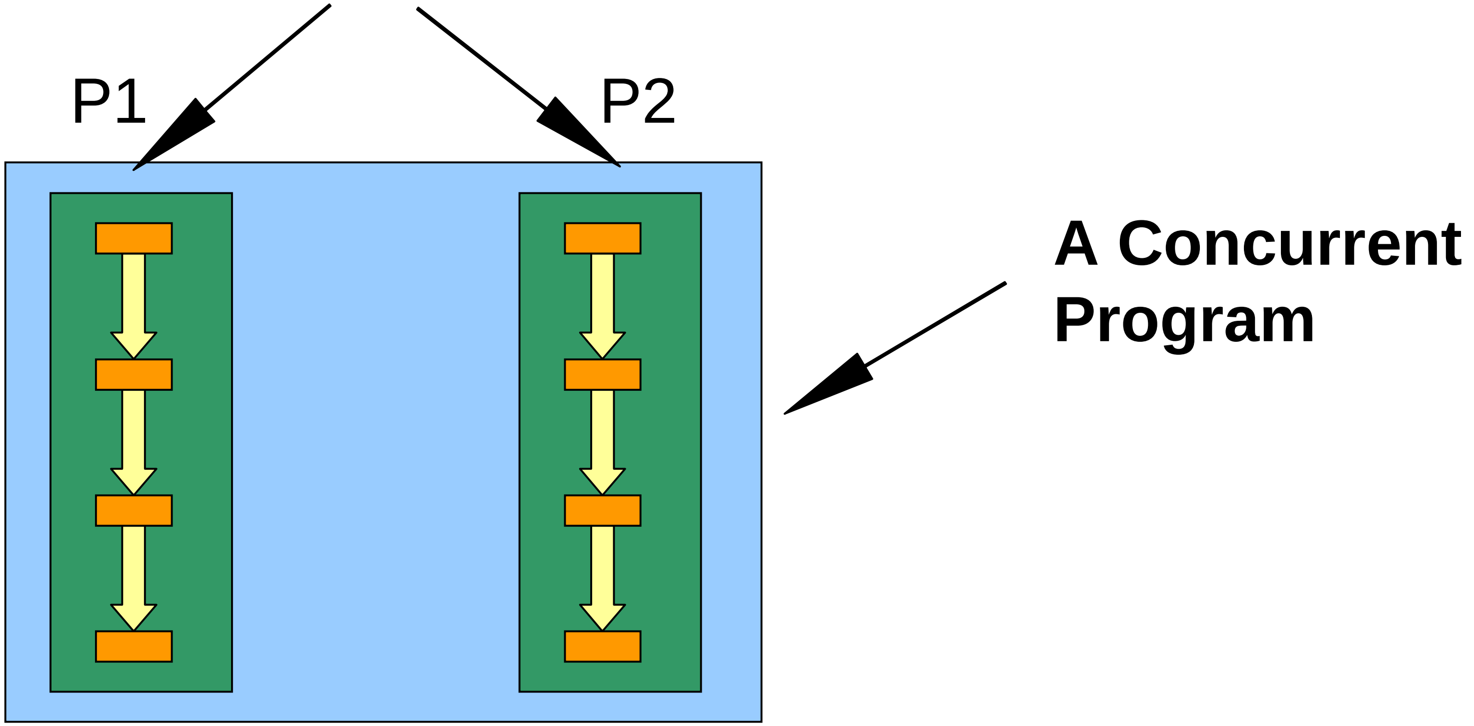
A Sequential Program

- Consisting of a single *process* carrying out a *totally ordered* sequence of *atomic actions*.
- Assuming a fixed input - program is deterministic.
- **NOTE –** we are using the term *process* in its *abstract form* as related to the concurrency abstraction – not in its concrete form related to operating systems.



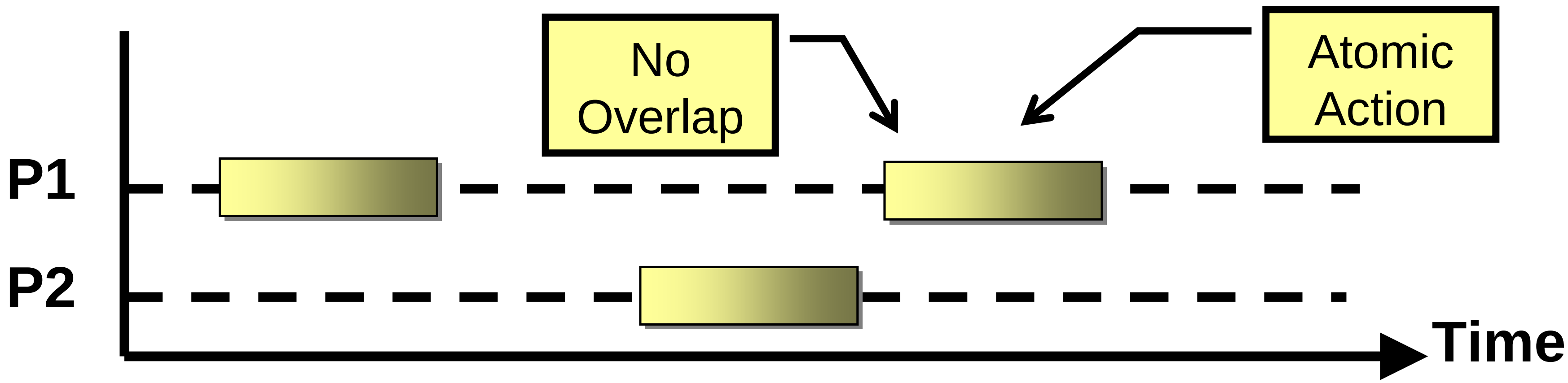
A Concurrent Program

- Consisting of two or more processes carrying out *atomic actions.*
- The term *Processes* is abstract – not concrete !



Atomic Actions and Interleaving

- In terms of the *concurrency abstraction*:
  - *atomic action,* as the name suggests, is an abstract term for an action that is not divisible in terms of processing.
  - *Interleaving* is an abstract term used to describe the **model** in which *atomic actions* from different *processes* are carried out.
  - *Atomic actions* do not get carried out simultaneously (at the same time) in the concurrency abstraction … they are *interleaved*.



A Concrete Example

- Assume we have common shared memory between the two processes – and that we have memory location N which currently contains the value 0.
- The processes have an *atomic instruction* (concrete atomic action) called **inc** that increments a memory location.

Process P1 carries out *atomic instruction:*

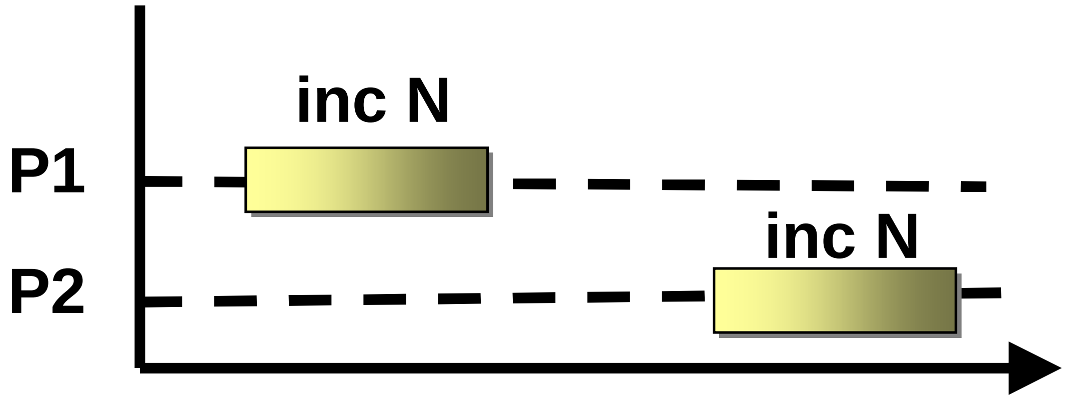
inc N

Process P2 carries out the *atomic instruction*:

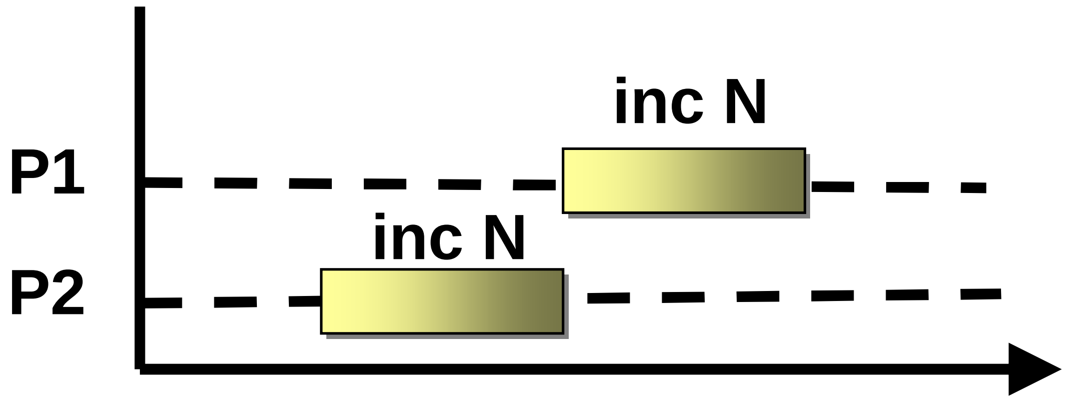
inc N

The actual HARDWARE makes these instructions atomic – the hardware only allows complete instructions to be carried out (and only a complete instruction at a time when using memory). It is important to realise that certain resources on a computer can be 'shared' (for instance a bus) and that it is left to the hardware to eliminate contention when different processes (e.g. devices) try to use these resources at the same time.

Two possible interleavings & results



Value of 2 at address N



Value of 2 at address N

All interleavings result in the desired behaviour – thus a good concurrent program.

But is this always the case ?

- Some of you looked at a MIPS processor last year.
- This is a RISC load/store design where values need to be loaded into ‘registers’ before operations can happen.
- Suppose our processor does not have an atomic instruction to increment a memory location.
- It has atomic instructions:
  - **`load A, M`** # Load register A with value at address M.
  - **`store A, M`** # Store value in register A at address M.
  - **`add A, 1`** # Add a value of 1 to register A.

RISC-based processor equivalent

- To do the same we have:

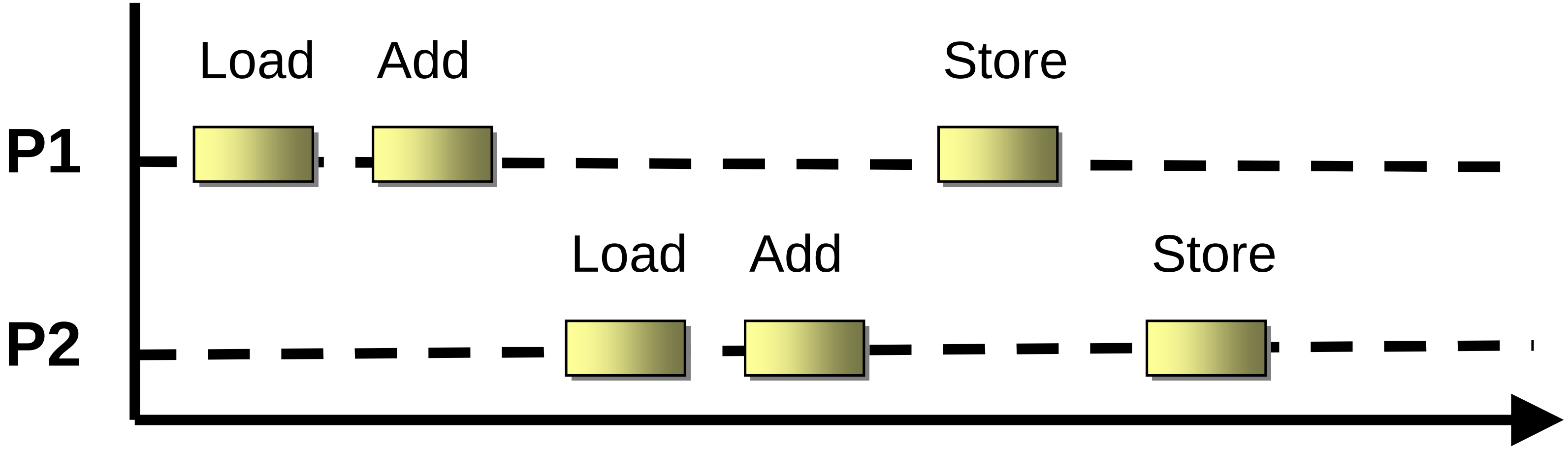
Process P1 carries out *atomic instruction:*

load A, M
add A, 1
store A, M

And process P2 carries out the *atomic instruction*:

load A, M
add A, 1
store A, M

One possible interleaving is now …



The result is a value of 1 rather than 2 at address N.

A CONCURRENT PROGRAM MUST WORK CORRECTLY UNDER ALL POSSIBLE INTERLEAVINGS OF THE PROCESSES' ATOMIC ACTIONS.

Sorting this concurrent program …

- One of the key features (amongst others) of a concurrent programming language is to define a mechanism which tells the computer what sections of code must appear atomic.
- These mechanisms are called ***synchronization facilities***.
- The sections of code are called ***critical sections.***

Process P1 carries out *atomic instruction:*

load A, M
add A, 1
store A, M

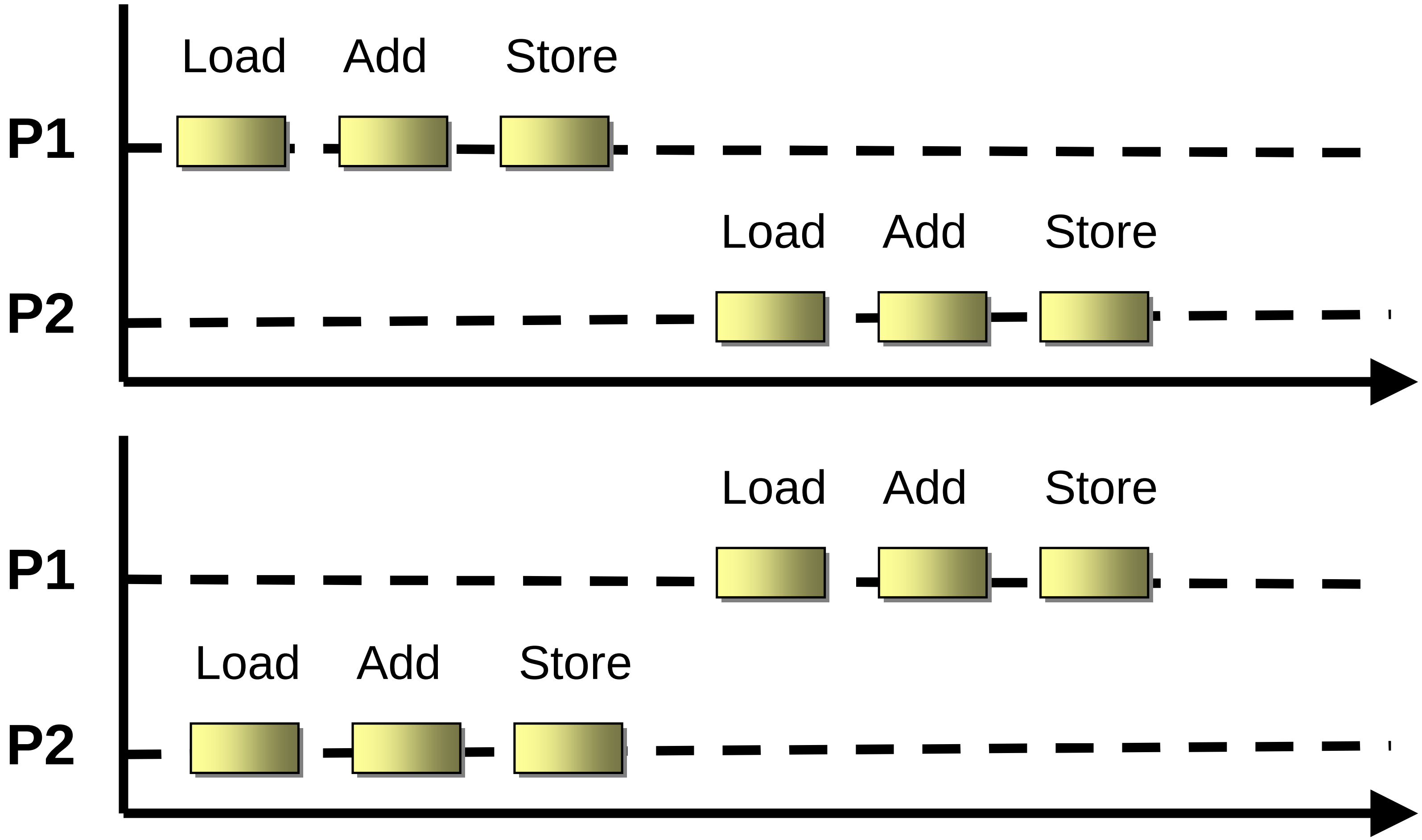
CRITICAL SECTION – MUST BE RUN AS ATOMIC UNIT

And process P2 carries out the *atomic instruction*:

load A, M
add A, 1
store A, M

CRITICAL SECTION – MUST BE RUN AS ATOMIC UNIT

We now only permit two interleavings …



We have synchronized our atomic actions so that we force the desired behaviour (an answer of 2).

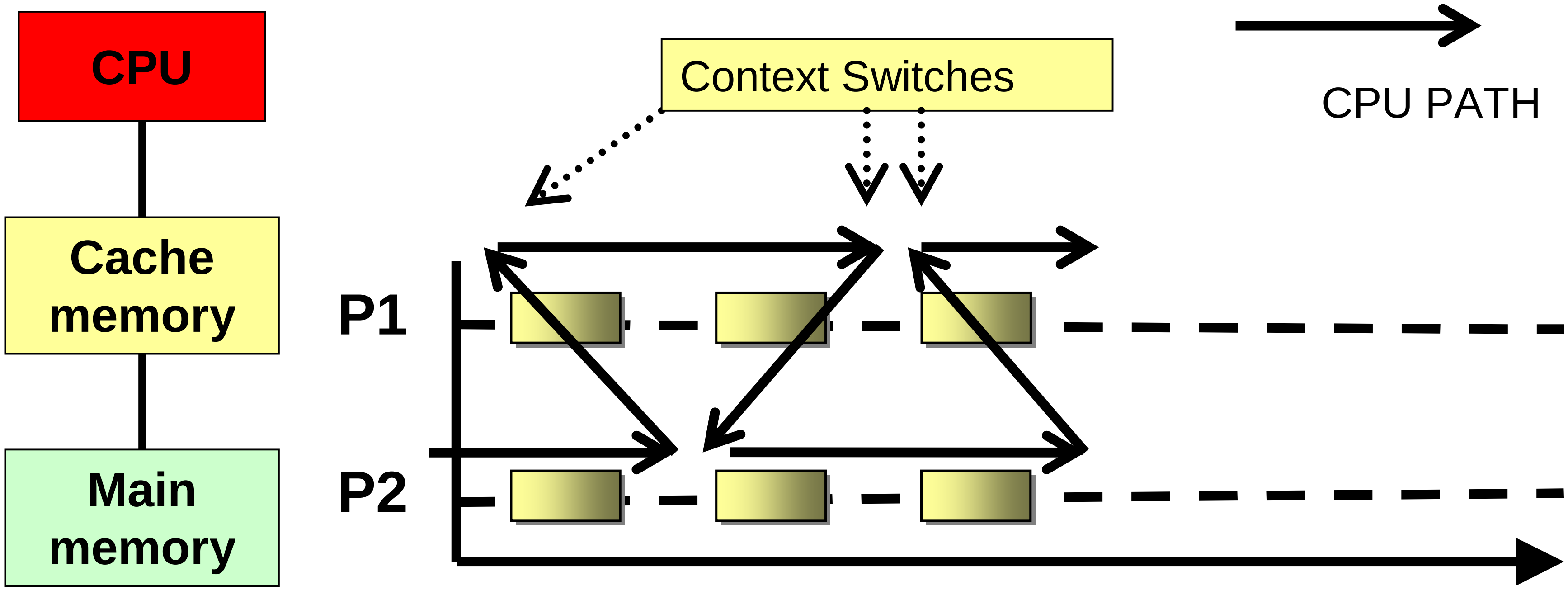
But is Java really affected by such low level considerations ?

- YES !
- All assignment of primitive types in Java are *specified* ("The Java Language Specification" by Gosling et al.) to be atomic EXCEPT …
- When you assign either longs or doubles.
- WHY ?
- A hardware consideration … these are 64-bit values and a lot of 16/32-bit hardware do not have atomic store instructions for 64-bit values and so Java permits long/double assignments to be non-atomic.

Does it really matter that long/double assignments in Java are not atomic?

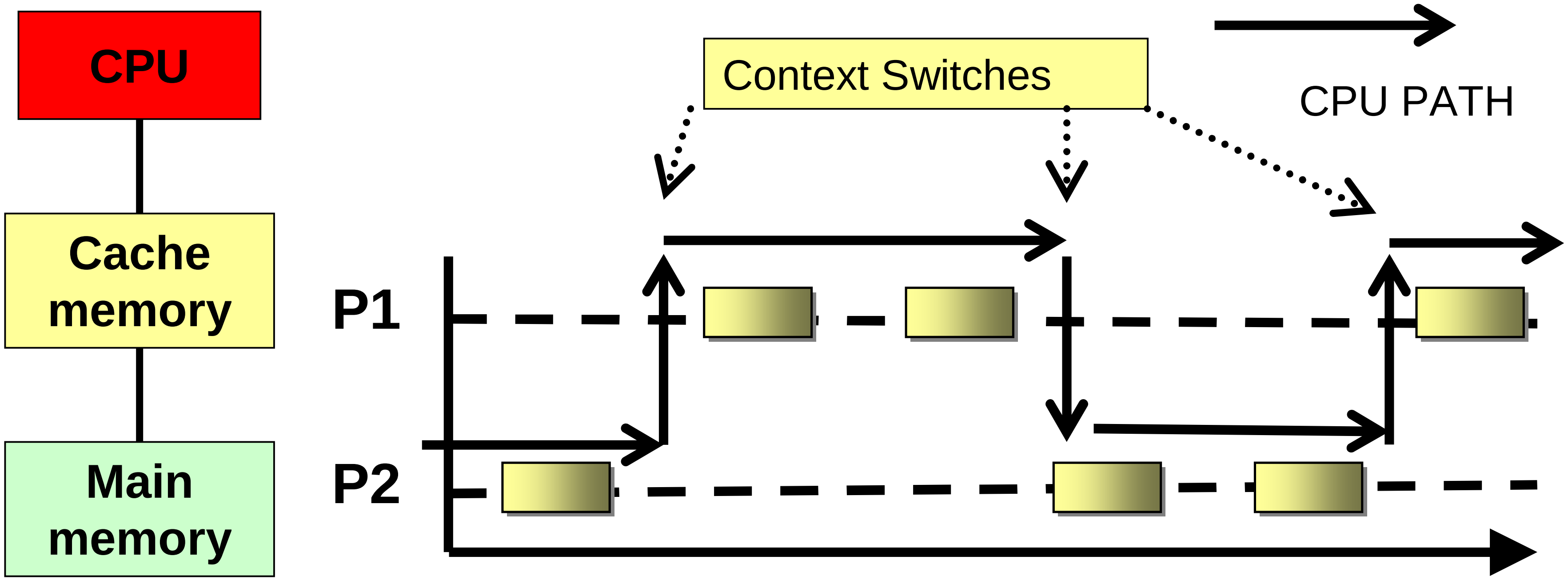
- YES !
- Only if you are writing concurrent software … then two processes/threads writing the value 3.14 and 10^-100 into a shared double variable may result in the variable having neither of these values!
  (A mixture of the first 32-bits of one value and the last 32-bits of the other!) … *as an exercise you could take the IEEE double precision format and see what possible numbers would result …*
- *MMM …*
- But does this abstraction of modelling things as interleaving atomic actions actually apply to proper computer systems?
- Again the answer is YES !

The case of multi-tasking on one CPU …
three atomic operations for two processes …



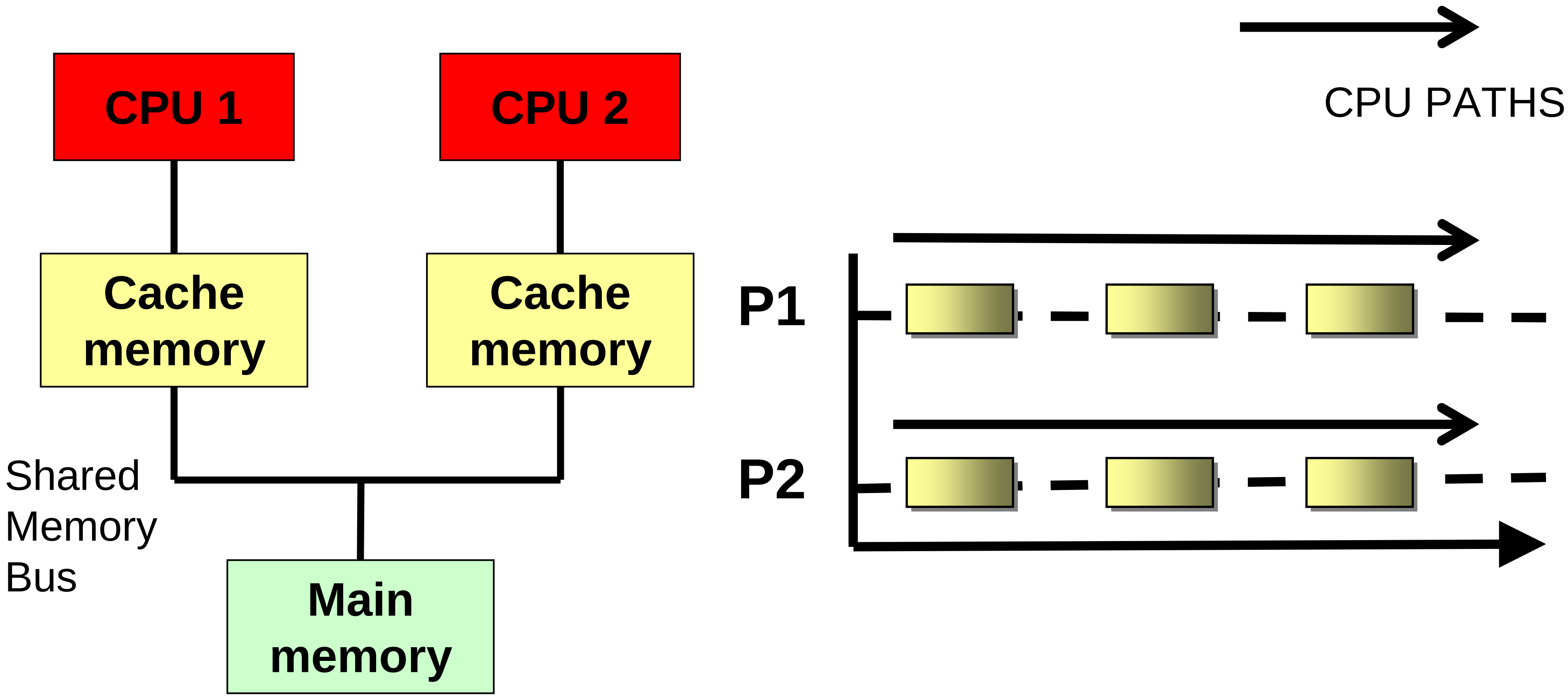
- The OS ‘context switches’ between the processes (stores all the CPU registers of one process before restoring all the registers of the other – including the Program Counter - and then continuing it’s operations).

Multi-tasking is accurately modelled by the concurrency abstraction ….



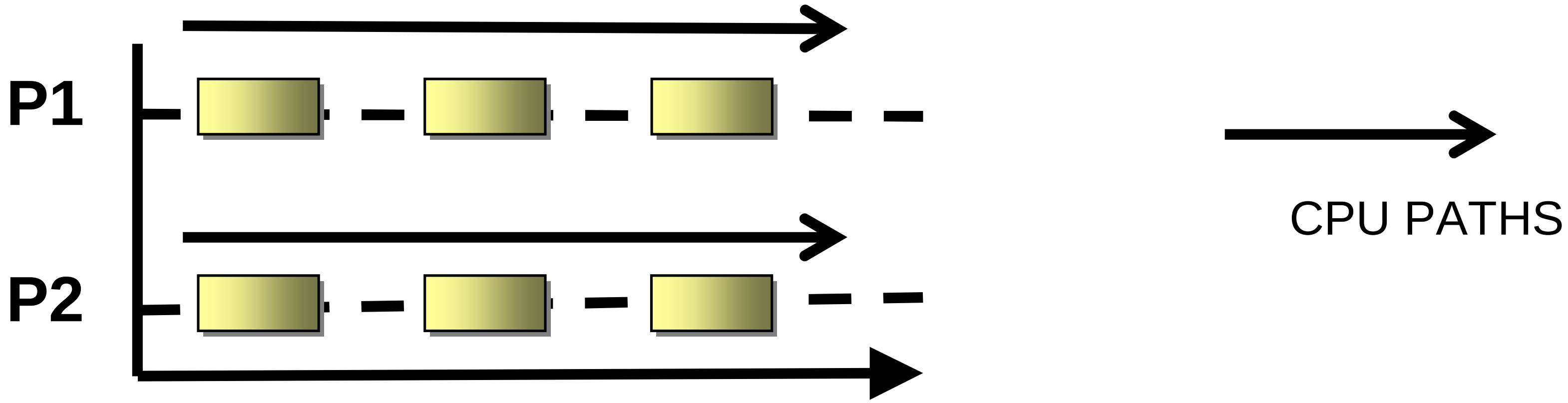
- The atomic operations/instructions/actions are serialized and non-overlapping.

The case of a multi-processor …

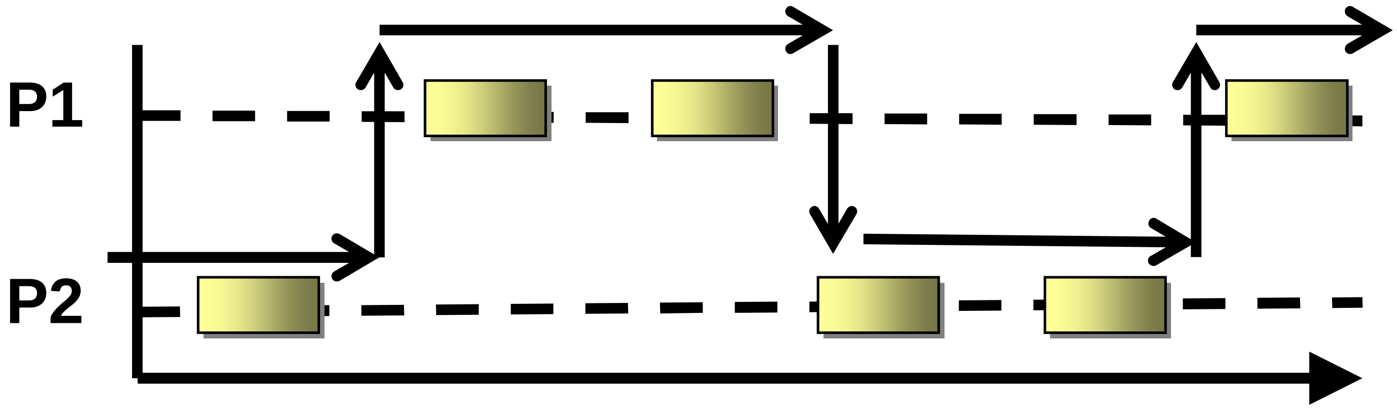


- Assume the two processes are being run on different processors with shared memory.

The case of a multi-processor …



- If the two processes *do not interact … then you will get the same final result if you just 'simulate' the two processors using a single processor (except it will take longer … but we are ignoring time in this abstraction!)*

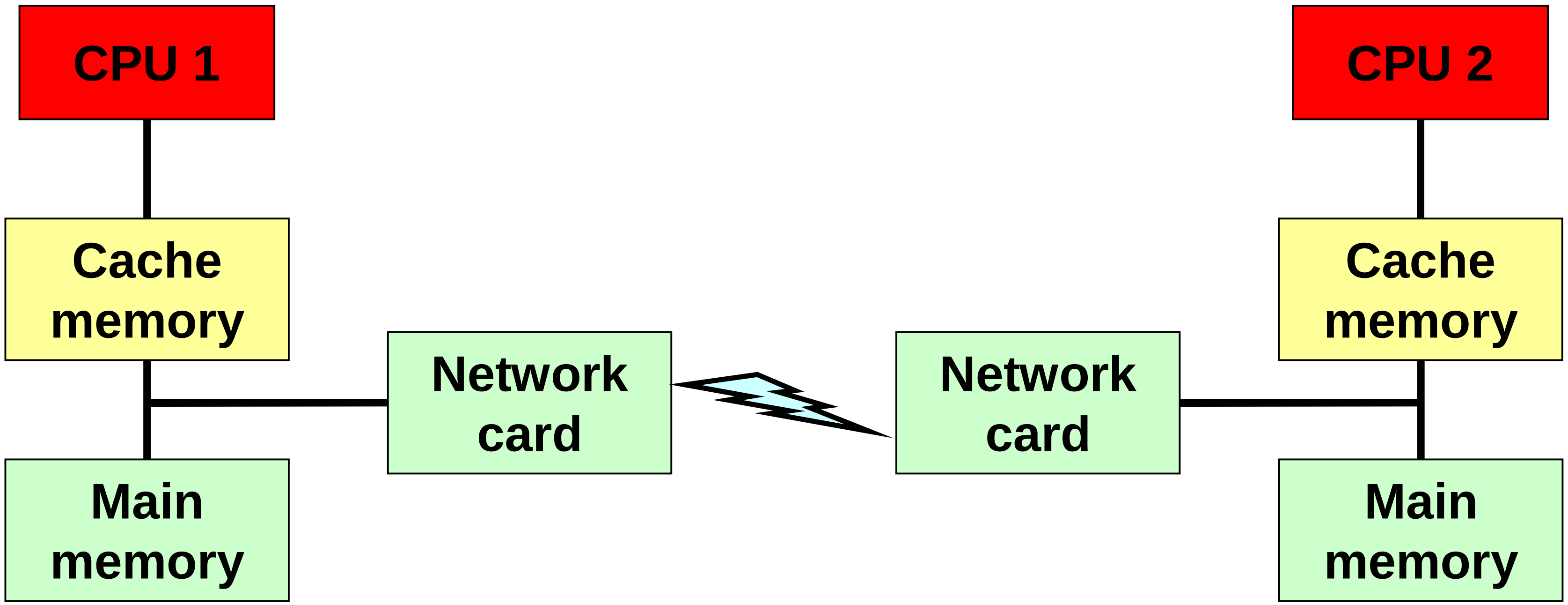


The case of a multi-processor …

- Even if the processes interact using shared memory – this interaction has to satisfy the atomic nature of the instructions (memory only allows one processor to read or write values at a time) – so this can still be simulated/modelled as if it was one processor interleaving the atomic actions.

- The same reasoning applies to distributed systems over networks – interaction between the processes is carried out by sending and receiving bytes which the network hardware essentially makes into ‘atomic actions’.

The case of distributed systems …



- Thus two computers running different programs in parallel that are interacting by message passing can be **modelled** as interleaved atomic instructions (there are some complications regarding 'fairness' here so that one process does not wait forever for a message from the other).

I’m worried about the fact that your ignoring the actual real time taken by actions …

- The concurrency abstraction ignores all aspects of time – each atomic action takes an arbitrary 1 unit of time.
- But surely we need to consider different actions taking different amounts of time?

- We cannot make any assumptions about the time things will take – upgrading a processor/memory/network card may completely revise how long certain actions take within a concurrent system.
- If the system works for all possible interleavings of the atomic actions … then it will work for all possible durations of the individual atomic actions.

- Programming real-time systems takes time into account … and then things get **much more** complex.

Lecture Summary

- We have defined a lot of terminology in this lecture and provided a definition for the concurrency abstraction.
- This abstraction allows us to think about what possible scenarios may result within a concurrent system. It allows us to not worry about whether things are really happening in parallel or other particular aspects of real time.
- In the next lecture we will apply some of this theory and actually program our first concurrent system in Java.