Computer Fundamentals
Lecture 4

Dr Robert Harle

Michaelmas 2013

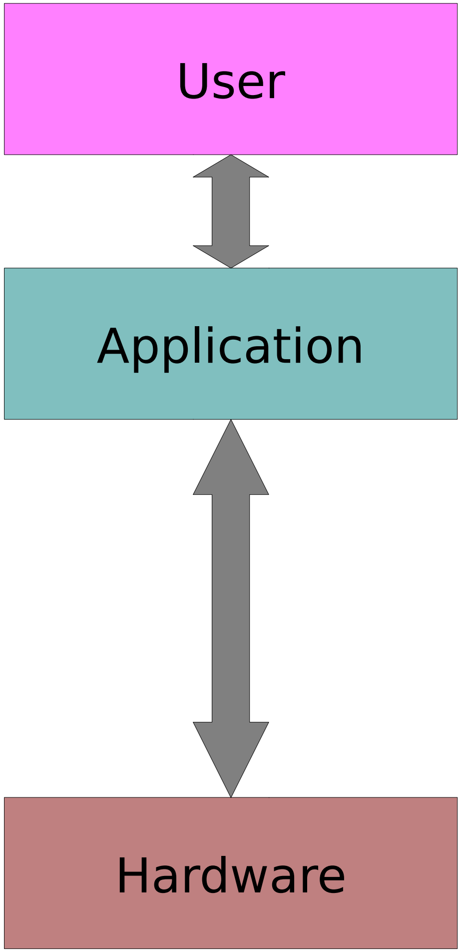
This Week

- The roles of the O/S (kernel, timeslicing, scheduling)
- The notion of threads
- Concurrency problems
- Multi-core processors
- Virtual machines

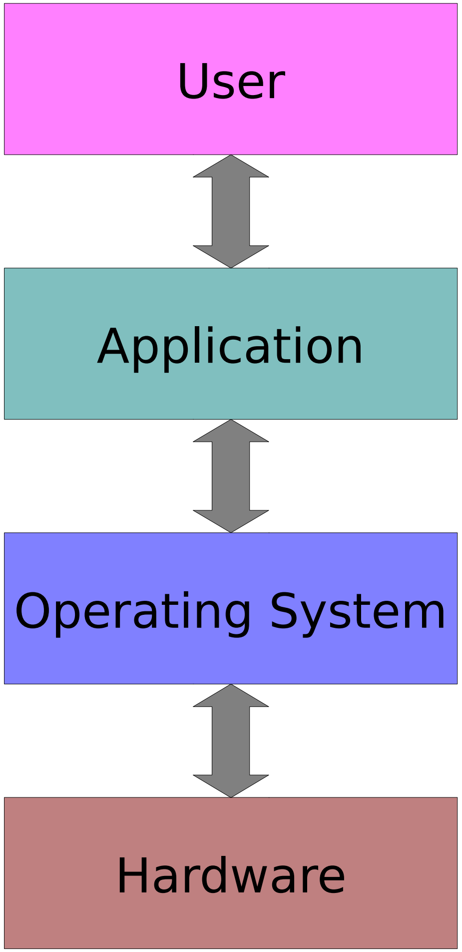
The Origins of the OS

- A lot of the initial computer programs covered the same ground – they all needed routines to handle, say, floating point numbers, differential equations, etc.
  - Therefore systems soon shipped with libraries: built-in chunks of programs that could be used by other programs rather than re-invented.
- Then we started to add new peripherals (screens, keyboards, etc).
  - To avoid having to write the control code ("drivers") for each peripheral in each program the libraries expanded to include this functionality
- Then we needed multiple simultaneous apps and users
  - Need something to control access to resources...

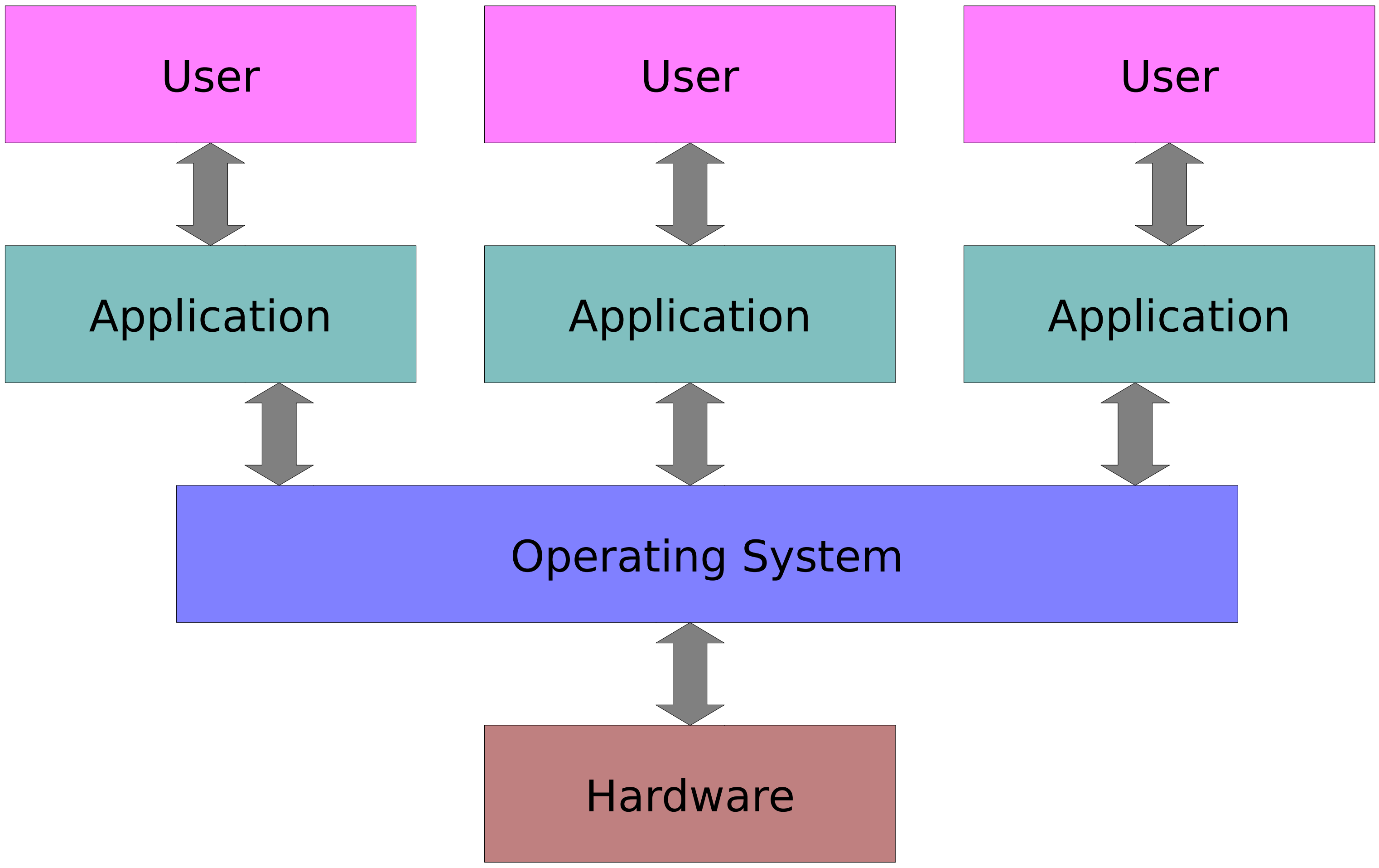
Operating System



Operating System



Operating System



OS Functions

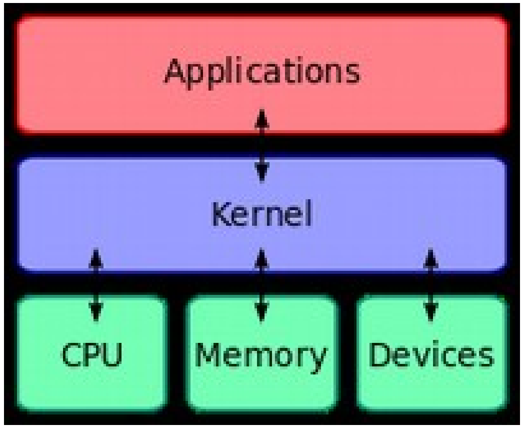
- Abstracts hardware (allows you to write code to e.g. access HDD and takes care of the different HDDs for you)
- Schedules processes (necessary for multitasking: see later)
- Allocates main memory (to individual processes)
- Provides library of useful functions (e.g. get system time, load file, etc)
- Enforces security
- May provide libraries to create a GUI

Platforms

- Almost all significant programs make use of the library functions in an OS (e.g. to draw a window)
- Our machine code needs not only a specific instruction set, but also the relevant operating system (with its libraries) installed
- So software is typically compiled for a specific platform: a (architecture, OS) pair
  - x86/Windows
  - ARM/Windows
  - x86/Linux
  - ARM/iOS
  - X86/OSX

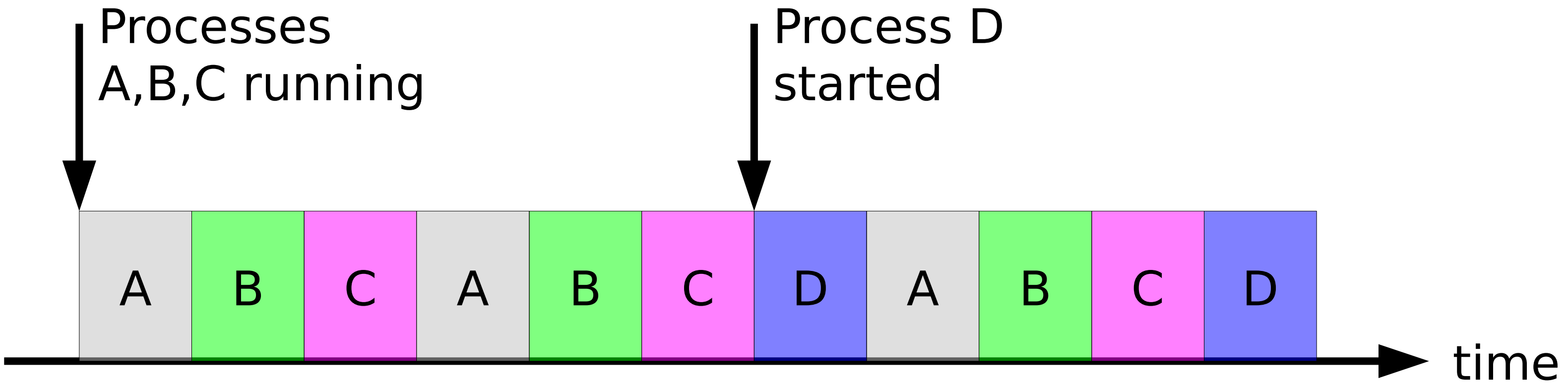
The Kernel

- The kernel is the part of the OS that runs the system
  - Just software
  - Handles process scheduling (see later)
  - Access to hardware
  - Memory management
- Very complex software – when it breaks... game over.



Multitasking by Time-slicing

- Modern OSes allow us to run many programs at once ("multitask"). Or so it seems. In reality a CPU time-slices:
  - Each running program (or "process") gets a certain slot of time on the CPU
  - We rotate between the running processes with each timeslot
  - This is all handled by the OS, which schedules the processes. It is invisible to the running program.

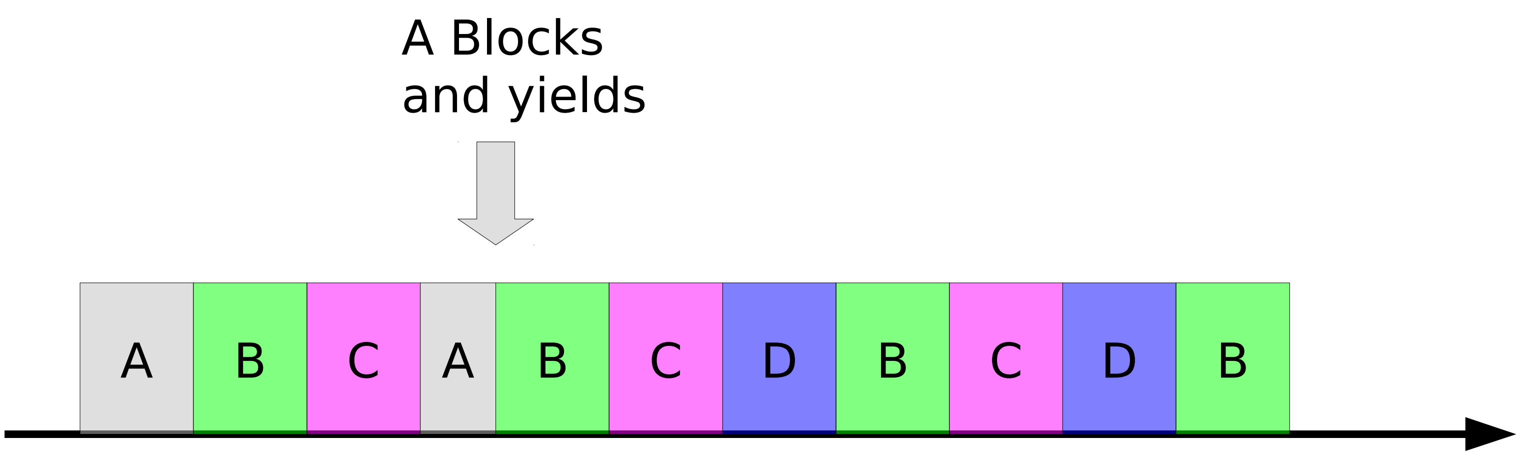


Context Switching

- Every time the OS decides to switch the running task, it has to perform a **context switch**
- It saves all the program's context (the program counter, register values, etc) to (main) memory
- It loads in the context for the next program
- Obviously there is a time cost associated with doing this...

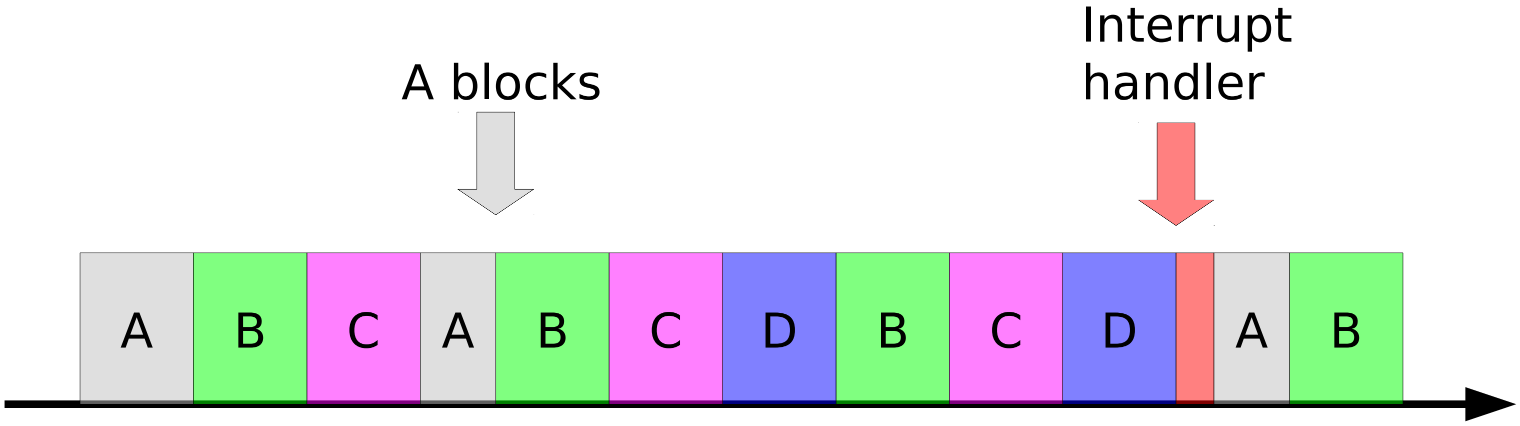
Relinquishing a Timeslot

- Sometimes a process is stuck waiting for something to happen (e.g. data to be read from disk)
- The process is "blocked"
  - Should release (yield) its timeslot
  - How can we know when to unblock it?



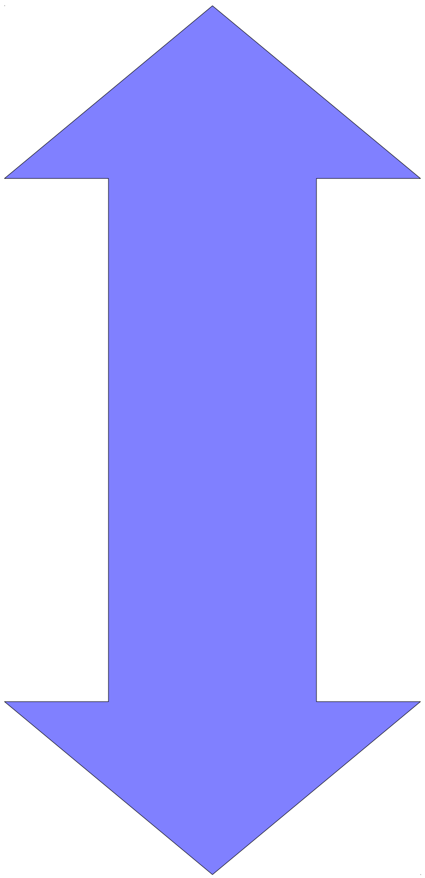
Interrupts

- Modern systems support interrupts
- Just signals that something has happened. An interrupt handler is associated with each interrupt
- E.g. HDD raises an interrupt to say it's done getting data → scheduler unblocks the process



Choosing a Timeslot Size

Longer



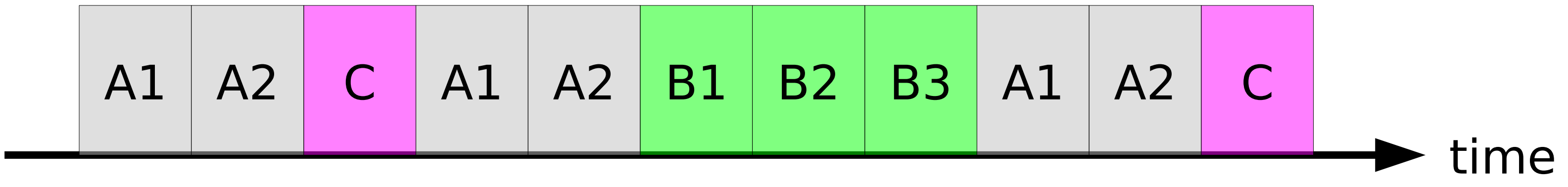
Shorter

- The computer is more efficient: it spends more time doing useful stuff and less time context switching
- The illusion of running multiple programs simultaneously is broken

- Appears more responsive
- More time context switching means the overall efficiency drops

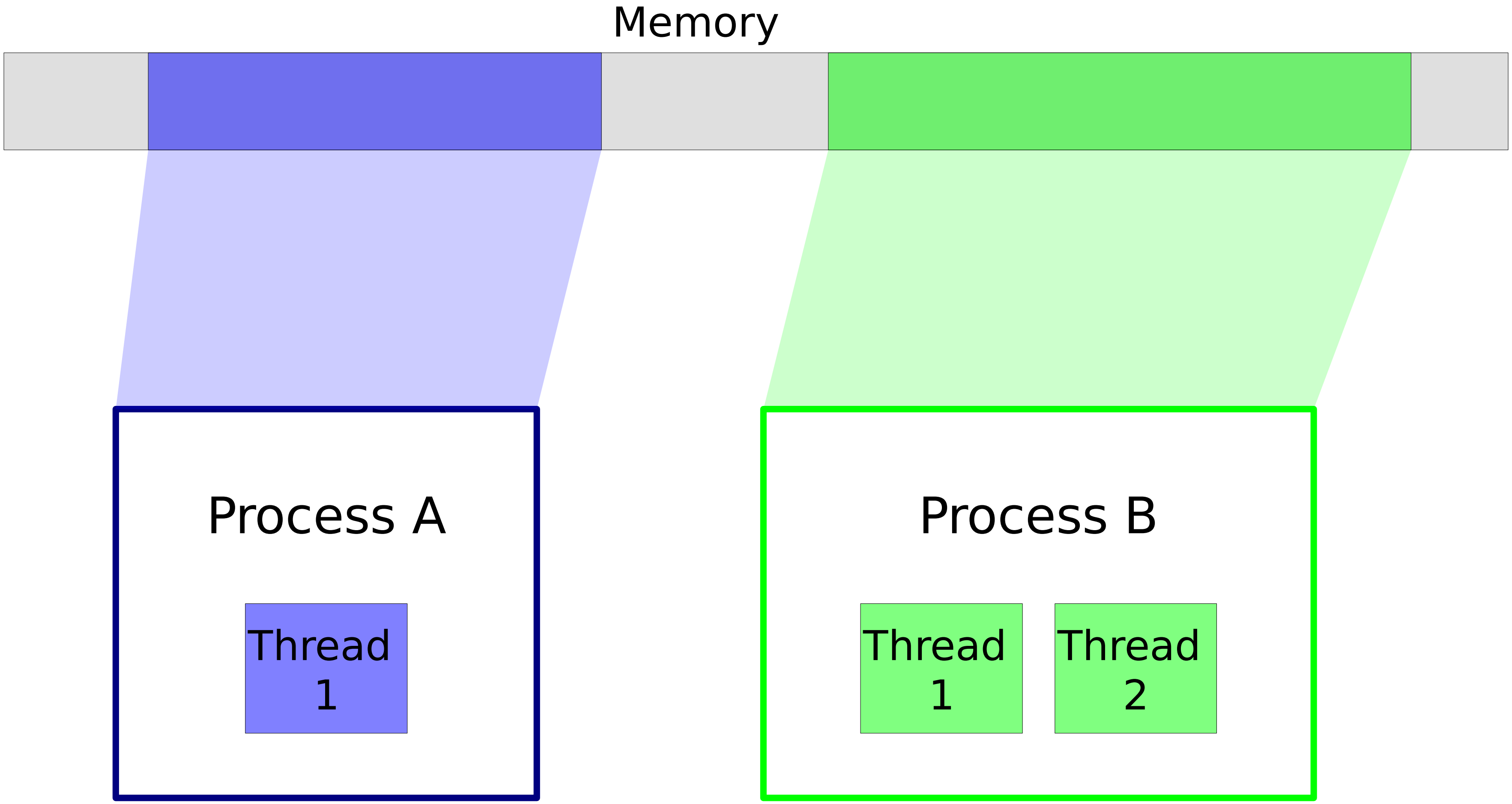
Threads

- Sometimes a program need to do background tasks whilst still performing a foreground task
- E.g. run an intensive computation but still process mouse events in case the user hits cancel.
- Processes have threads: effectively sub processes that run and are scheduled independently



Processes vs Threads

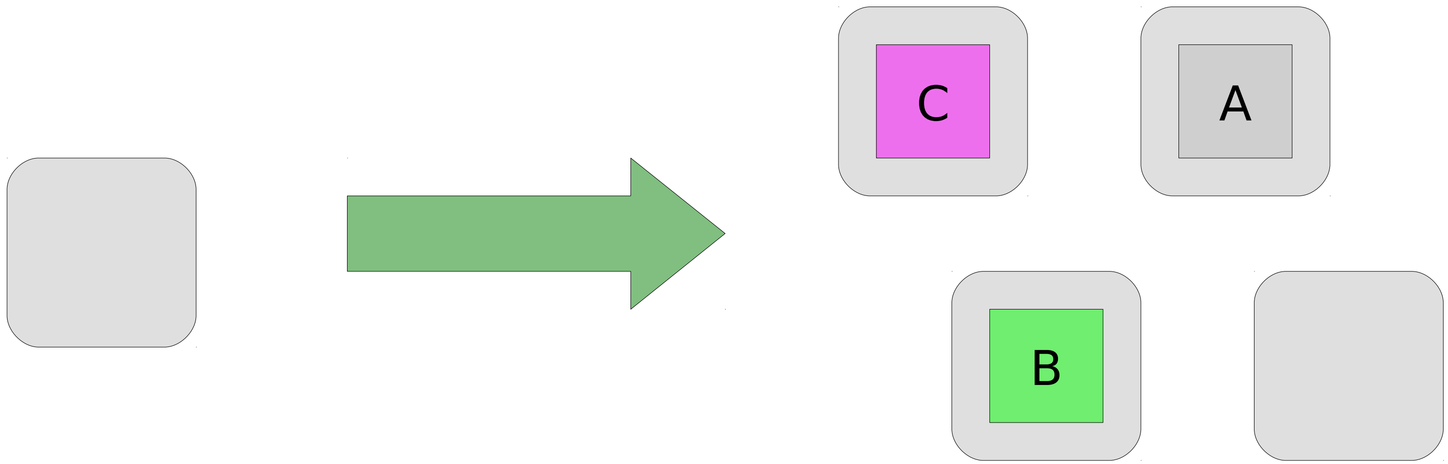
- Threads run independently but share memory



Multiple CPUs

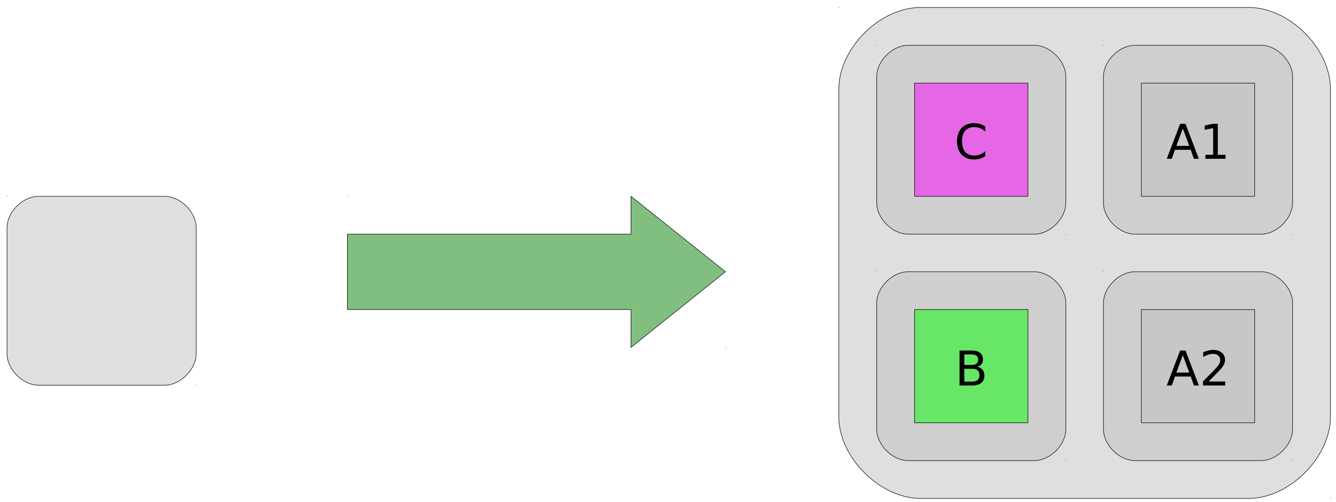
- Ten years ago, each generation of CPUs packed more in and ran faster. But:
  - The more you pack stuff in, the hotter it gets
  - The faster you run it, the hotter it gets
  - And we got down to physical limits anyway!!

- Some systems had multiple CPUs to get speed up



Multicore CPUs

- Modern system contain chips with multiple cores: multiple CPUs in a single package
- Connections shorter → faster
- Lower power



The New Challenge

- Two cores run completely independently, so a single machine really *can* run two or more applications simultaneously
- BUT the real interest is how we write reliable programs that use **more** than one core or thread
  - This is hard because they use the same resources, and they can then interfere with each other
  - Those sticking around for IB CST will start to look at such **concurrency** issues in far more detail. We will just look at...

Race Conditions

c=5

Main memory

Thread 1

```
c = c + 1;
```

Thread 2

```
c = c - 1;
```

Race Conditions

c=5

Main memory

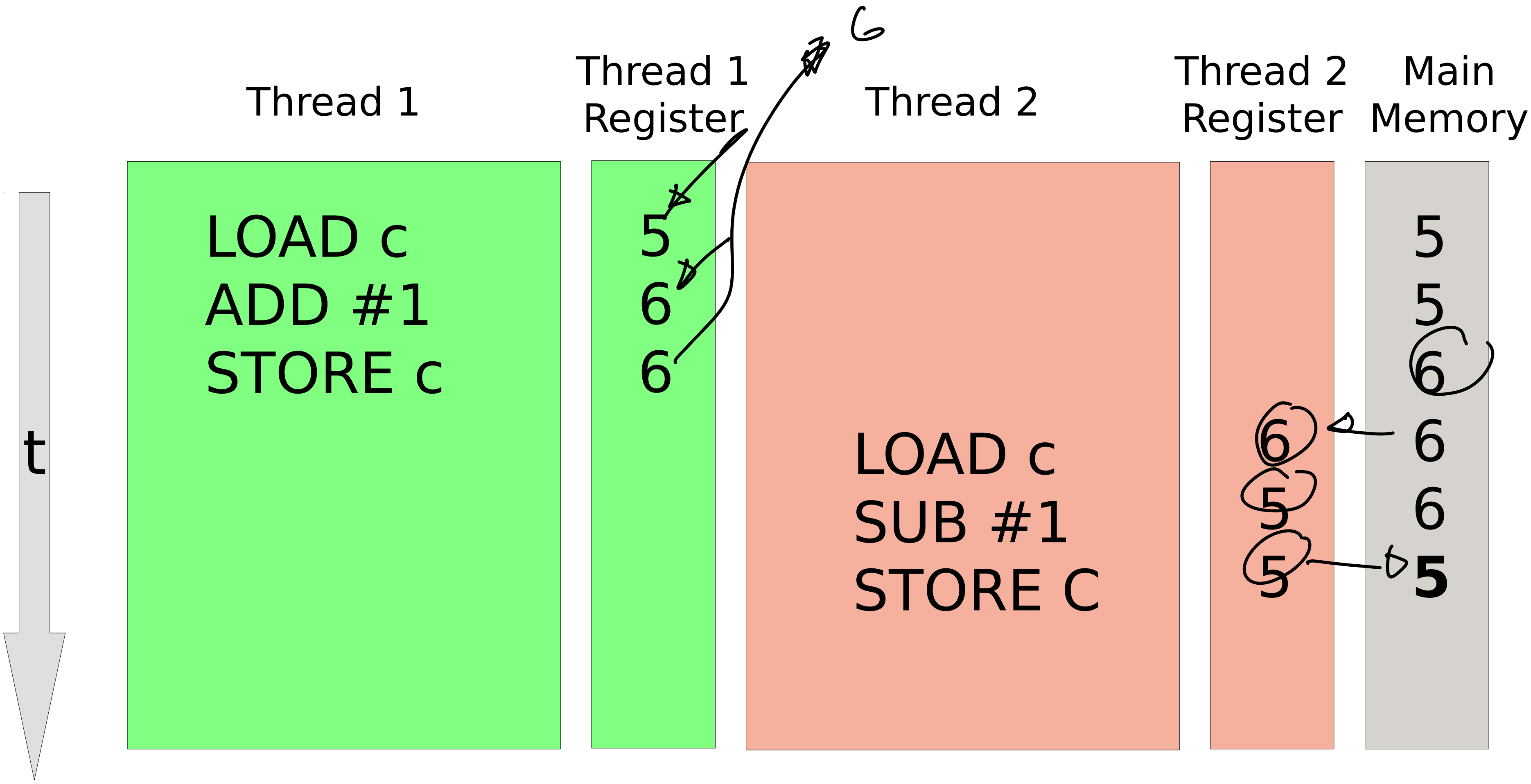
Thread 1

```
LOAD c
ADD #1
STORE c
```

Thread 2

```
LOAD c
SUB #1
STORE C
```

Race Conditions



Race Conditions

t	Thread 1	Thread 1 Register	Thread 2	Thread 2 Register	Main Memory
	LOAD c	5			5
			LOAD c	5	5
			SUB #1	4	5
			STORE C	4	4
	ADD #1	6			4
	STORE c	6			**6**

Race Conditions

t

Thread 1	Thread 1 Register	Thread 2	Thread 2 Register	Main Memory
		LOAD c	5	5
LOAD c	5			5
ADD #1	6			5
STORE c	6			6
		SUB #1	4	6
		STORE C	4	**4**

Race Conditions

- When we have two or more threads sharing a piece memory the result can depend on the order of execution
- → “Race condition”

- Hard to detect (non-deterministic)
- Hard to debug
- Generally just hard

Solving Race Conditions

```
LOAD c
ADD #1
STORE c
```

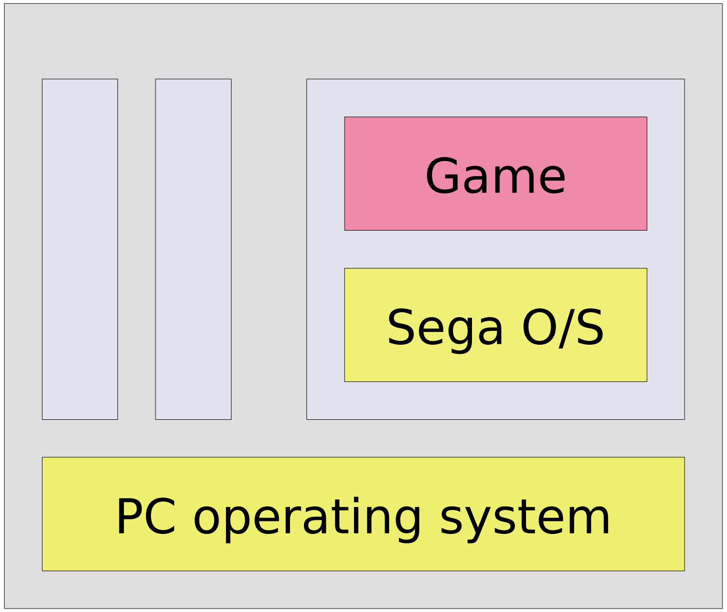
- Risky sets of operations like this one must be made atomic
- i.e. no context switching once the code block is started
- *Not trivial* → much of CST IB devoted to this

Aside: The Value of Immutability

- If something is immutable, the race conditions go away since you can only read it → remember this for OOP

Virtual Machines

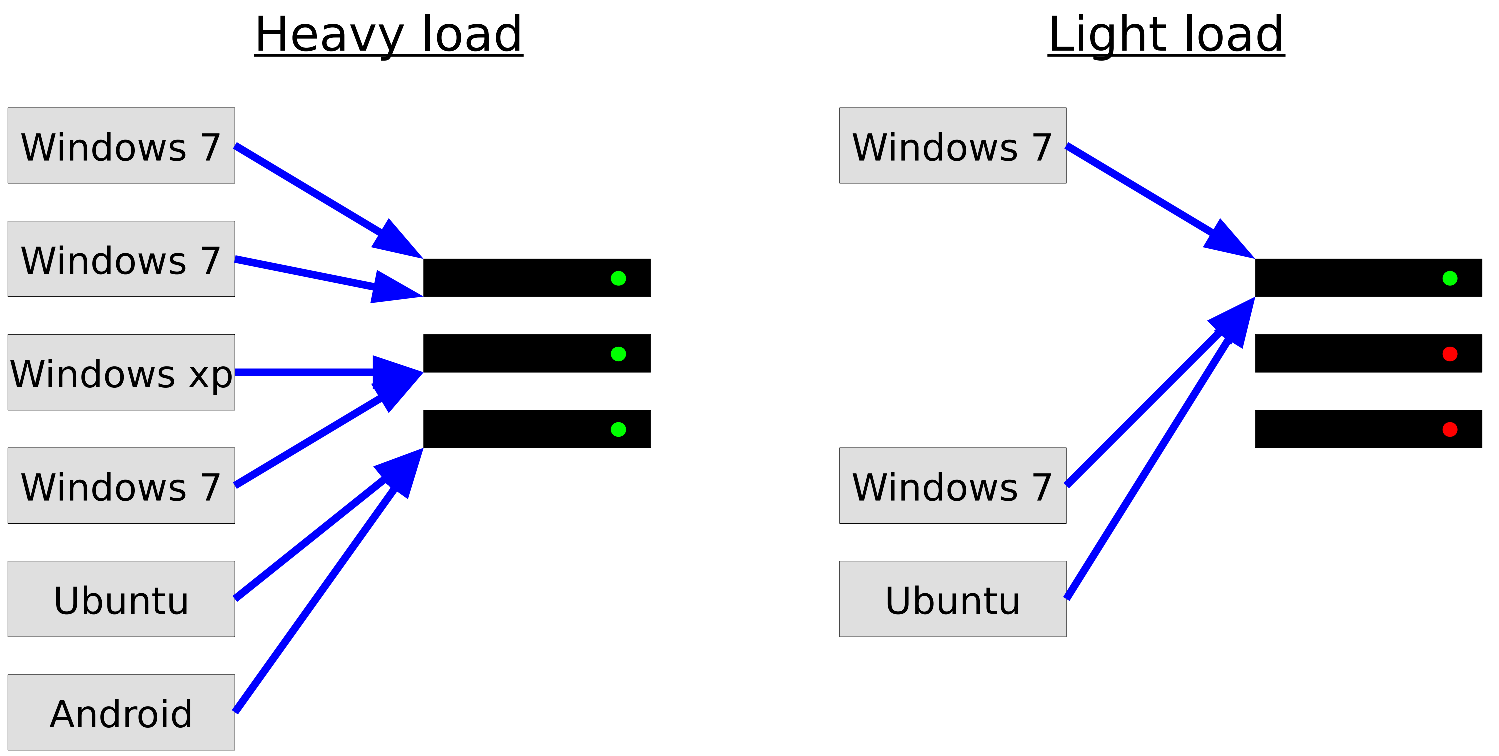
- Go back 20 years and emulators were all the rage: programs on architecture X that simulated architecture Y so that programs for Y could run on X
- Essentially interpreters, except they had to recreate the entire system. So, for example, they had to run the operating system on which to run the program.



- Now computers are so fast we can run multiple *virtual machines* on them
- Allows us to run multiple operating systems simultaneously!

Virtualisation

- Virtualisation is the new big thing in business. Essentially the same idea: emulate entire systems on some host server
- But because they are virtual, you can swap them between servers by copying state
- And can dynamically load your server room!



Thanks for coming

- These optional lectures were an experiment. I'd appreciate any feedback you have.
  - Should they be repeated next year?
  - Was the level about right? Too fast? Too slow?
  - Was there anything else you'd like to have had covered? Anything you'd not bother with?