Computer Fundamentals: Operating Systems, Concurrency

Dr Robert Harle

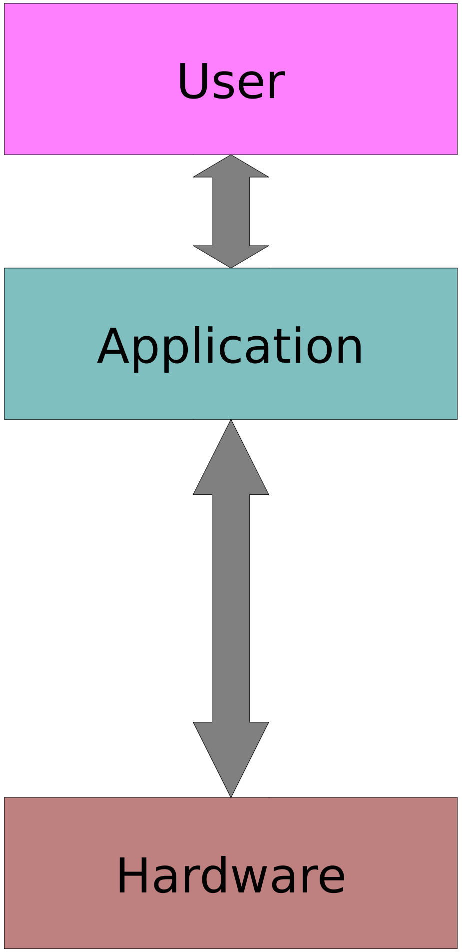
This Week

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

Traditionally

- A single program for a single user at a single time

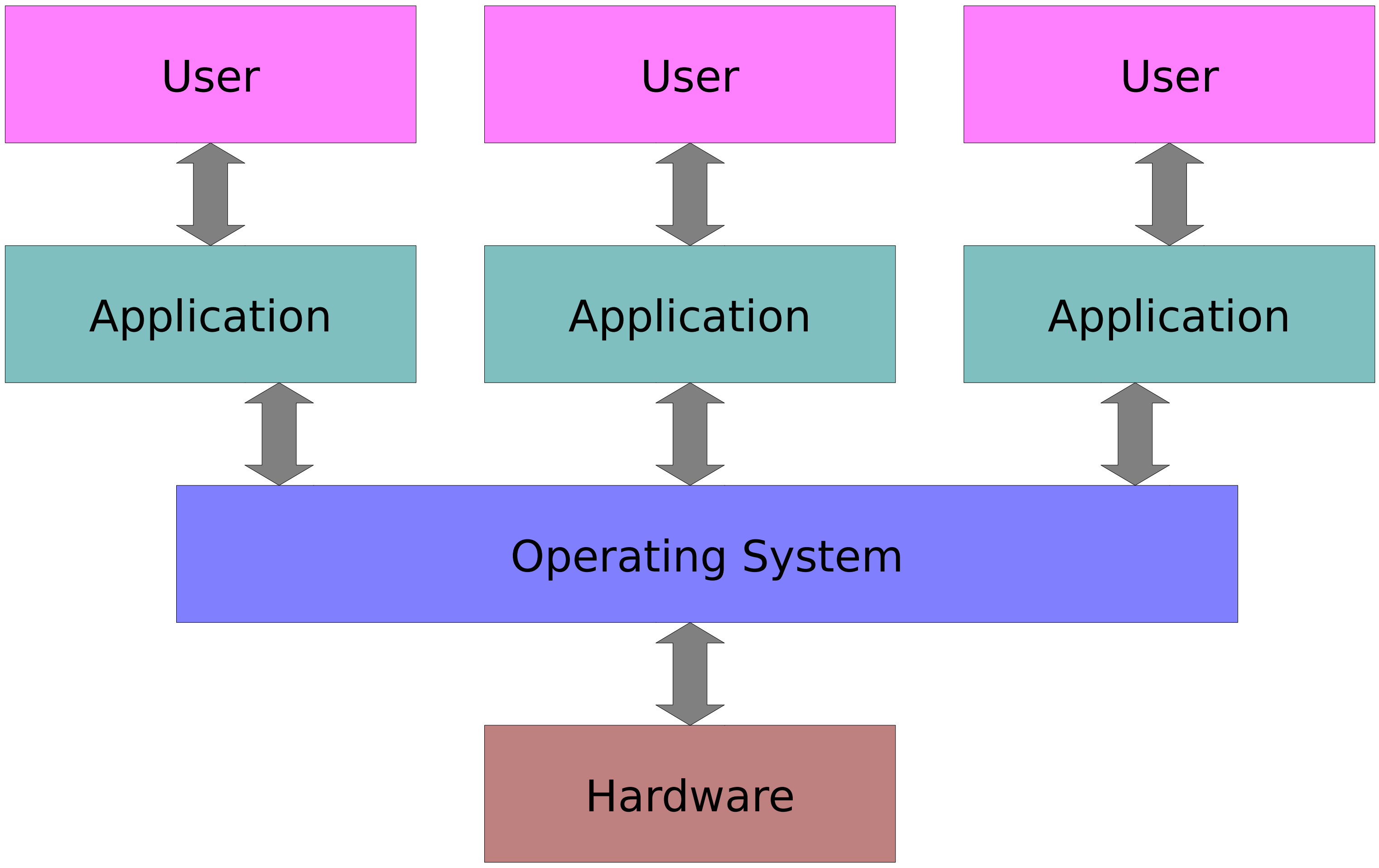
Operating System



Time sharing

- A single computer for multiple users each executing a single program

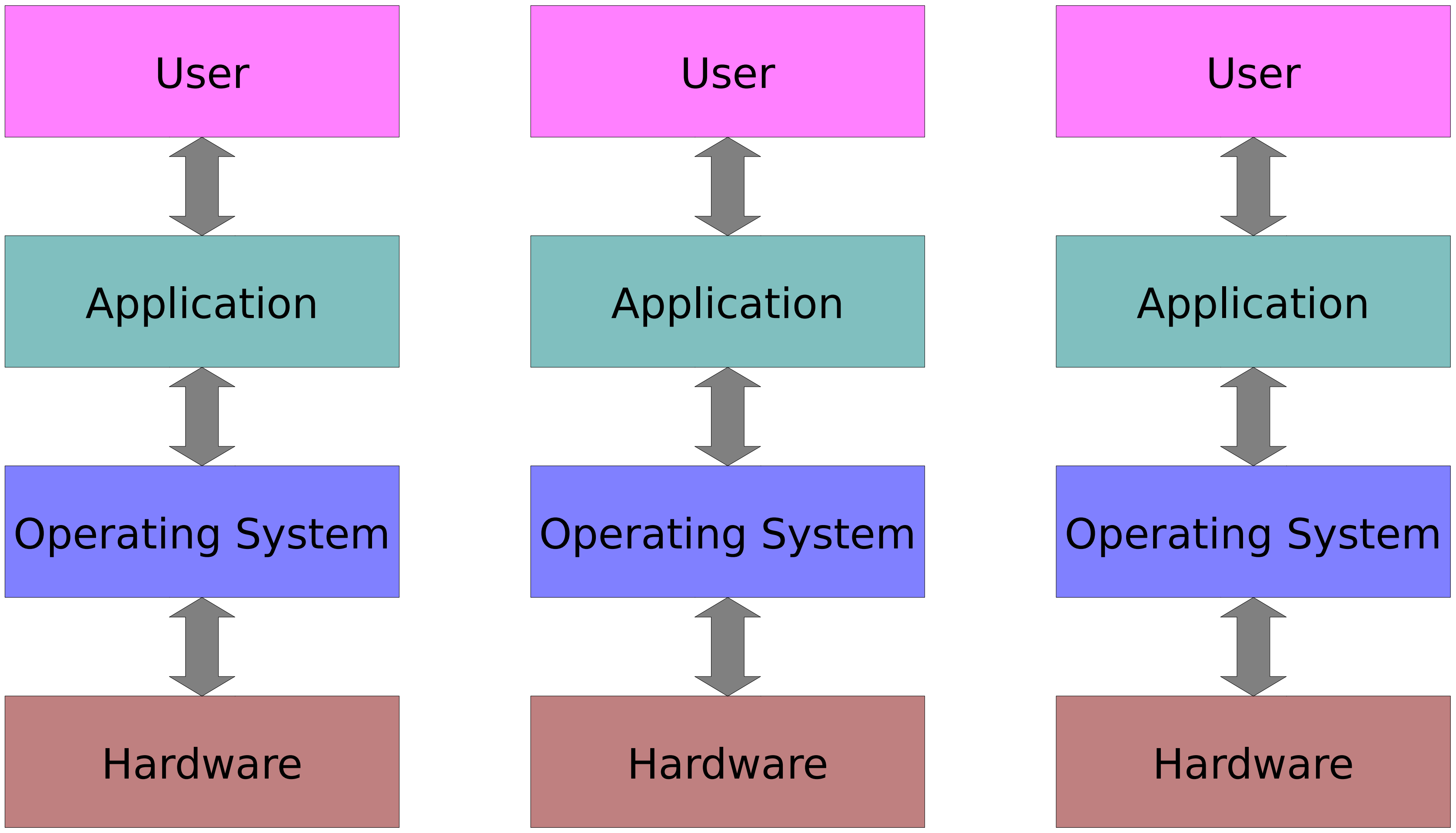
Operating System



Microprocessors (early 80s)

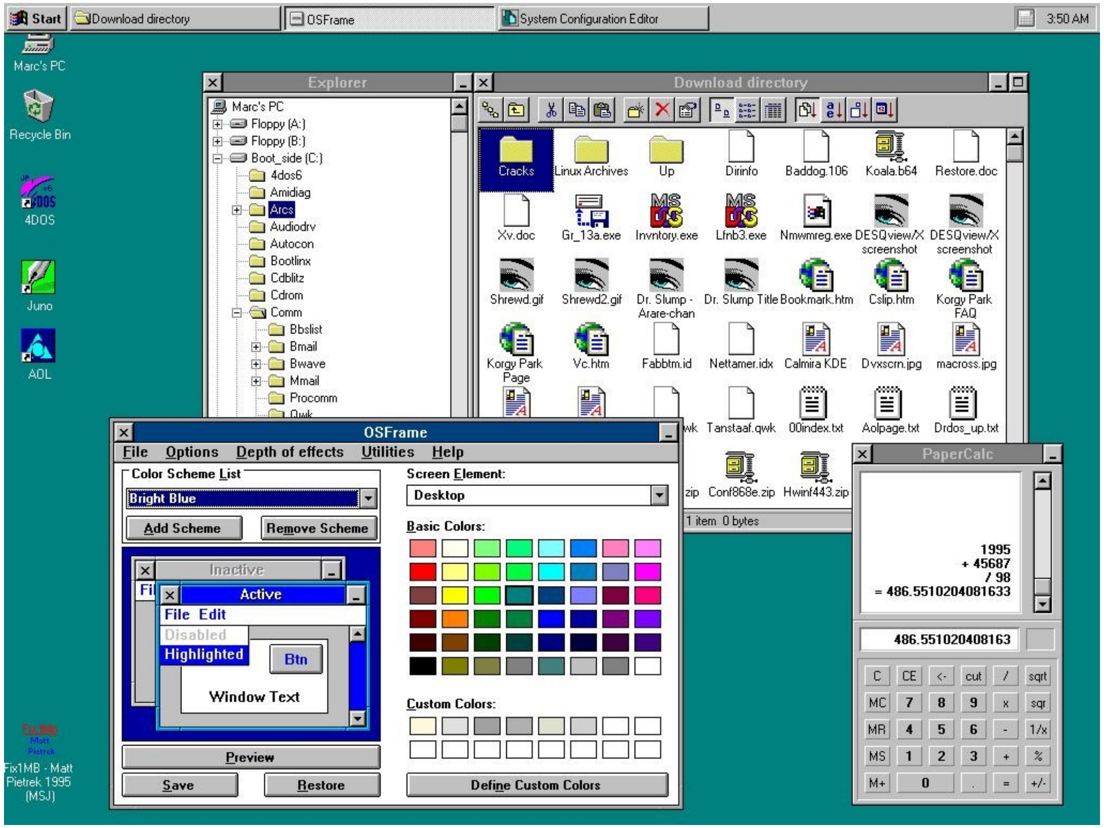
- A dedicated machine for each person running a single program

Operating System

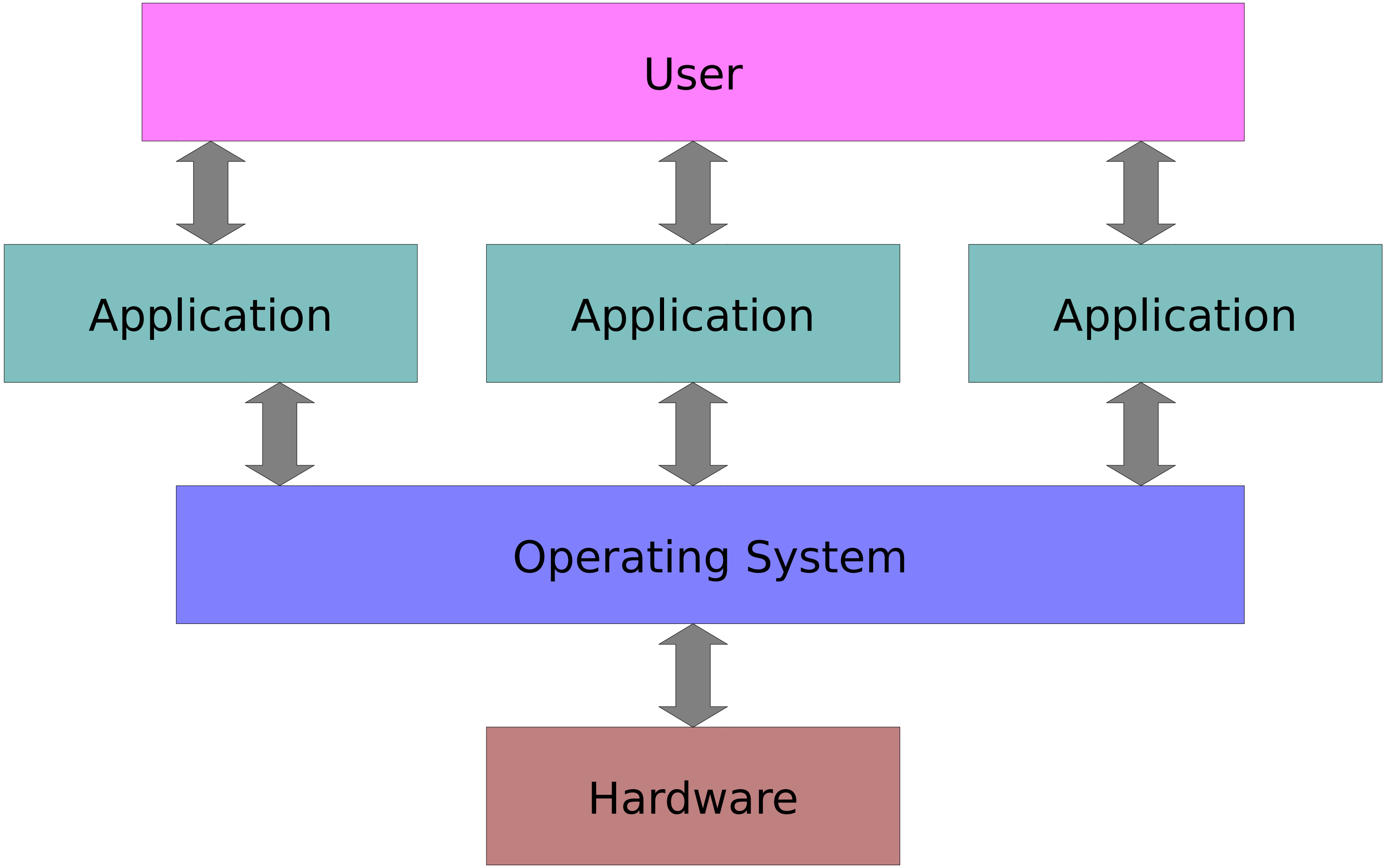


Multitasking (80s+)

- A dedicated machine for each person running multiple programs



Operating System

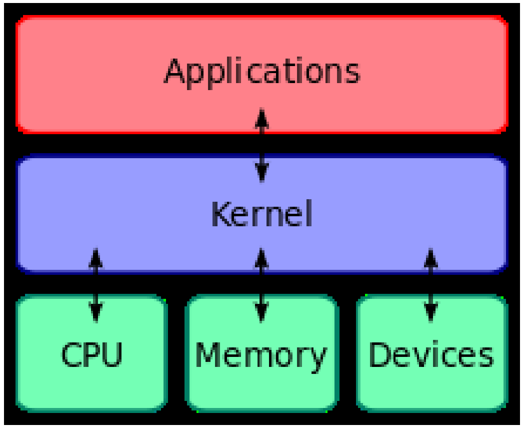


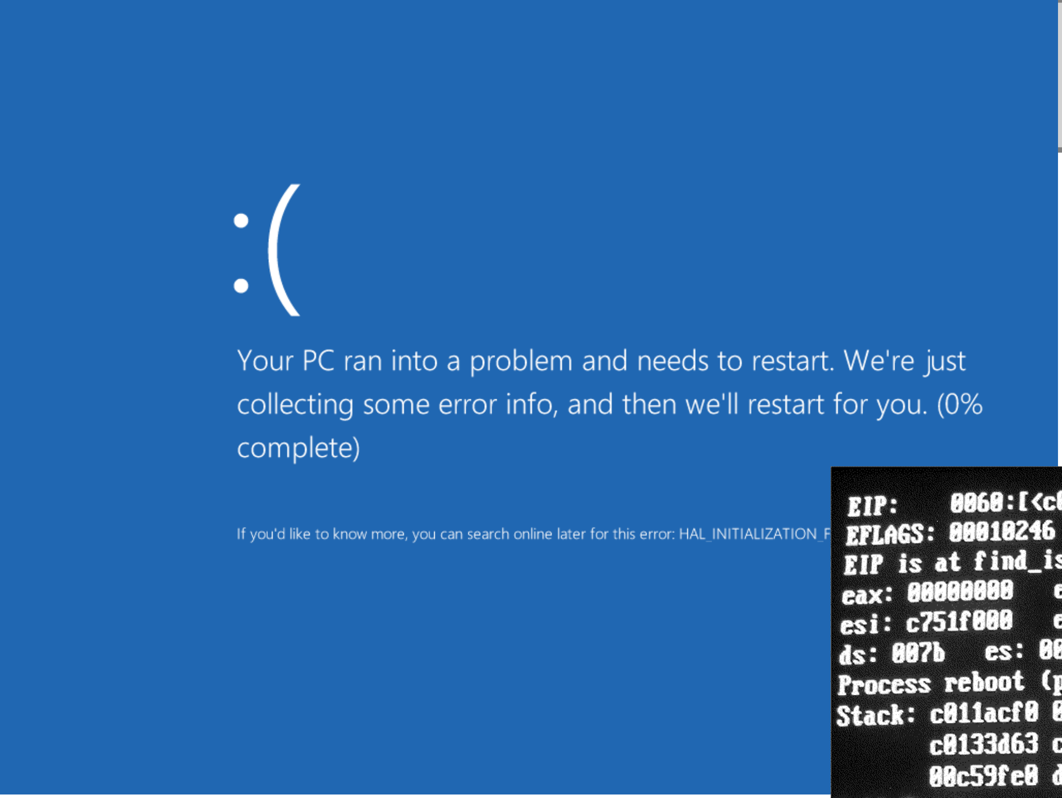
OS Functions

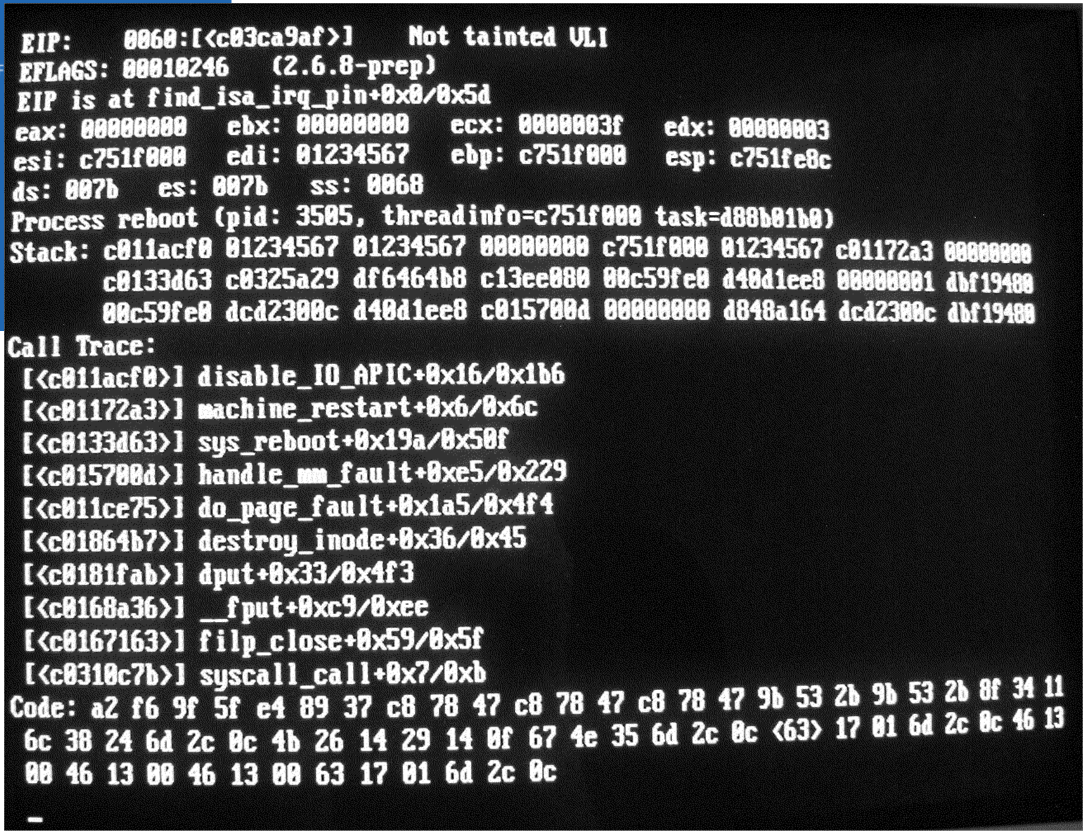
- A modern OS does a **lot**
- Abstracts hardware (allows you to write code to e.g. access HDD and takes care of the different HDDs for you)
- Schedules processes
- Allocates main memory (to individual processes)
- Provides library of useful functions (e.g. get system time, load file, etc)
- Enforces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

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.

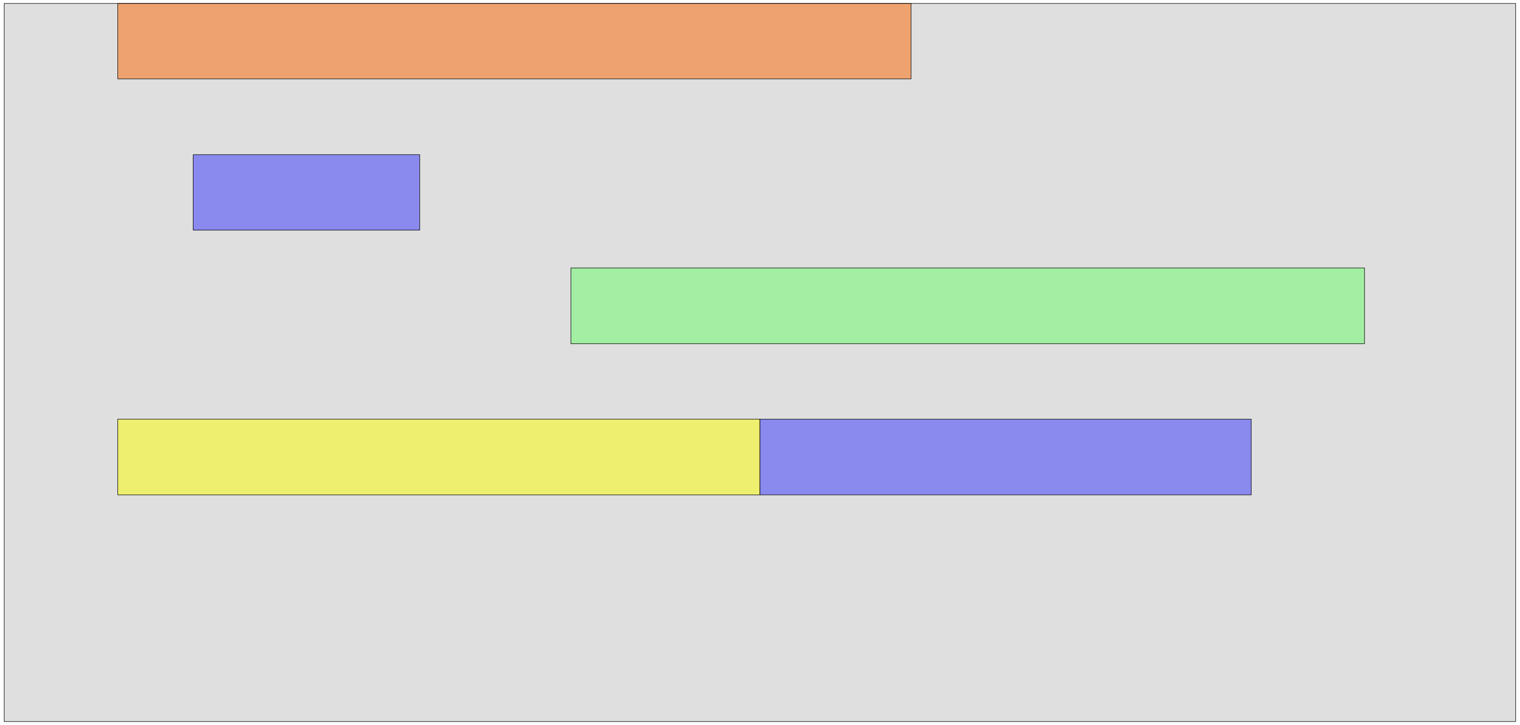



:(
Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (0% complete)
If you'd like to know more, you can search online later for this error: HAL_INITIALIZATION_F


EIP: 0060:[<c03ca9af>] Not tainted VLI
EFLAGS: 00010246 (2.6.8-prep)
EIP is at find_isa_irq_pin+0x0/0x5d
eax: 00000000 ebx: 00000000 ecx: 0000003f edx: 00000003
esi: c751f000 edi: 01234567 ebp: c751f000 esp: c751fe8c
ds: 007b es: 007b ss: 0068
Process reboot (pid: 3505, threadinfo=c751f000 task=d88b01b0)
Stack: c011acf0 01234567 01234567 00000000 c751f000 01234567 c01172a3 00000000
c0133d63 c0325a29 df6464b8 c13ee080 00c59fe0 d40d1ee8 00000001 dbf19480
00c59fe0 dcd2300c d40d1ee8 c015700d 00000000 d848a164 dcd2300c dbf19480
Call Trace:
[<c011acf0>] disable_IO_APIC+0x16/0x1b6
[<c01172a3>] machine_restart+0x6/0x6c
[<c0133d63>] sys_reboot+0x19a/0x50f
[<c015700d>] handle_mm_fault+0xe5/0x229
[<c011ce75>] do_page_fault+0x1a5/0x4f4
[<c01864b7>] destroy_inode+0x36/0x45
[<c0181fab>] dput+0x33/0x4f3
[<c0168a36>] __fput+0xc9/0xee
[<c0167163>] filp_close+0x59/0x5f
[<c0310c7b>] syscall_call+0x7/0xb
Code: a2 f6 9f 5f e4 89 37 c8 78 47 c8 78 47 c8 78 47 9b 53 2b 9b 53 2b 8f 34 11
6c 38 24 6d 2c 0c 4b 26 14 29 14 0f 67 4e 35 6d 2c 0c <63> 17 01 6d 2c 0c 46 13
00 46 13 00 46 13 00 63 17 01 6d 2c 0c

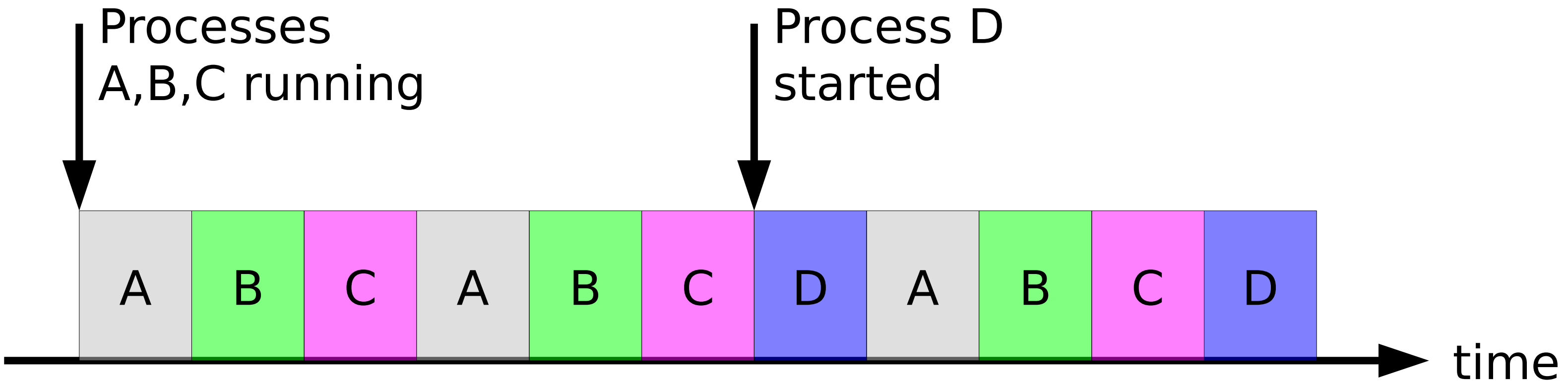
Memory Management

- The kernel allocates chunks of main memory to each process. It tries to prevent a program from accessing anything outside its allocation



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.



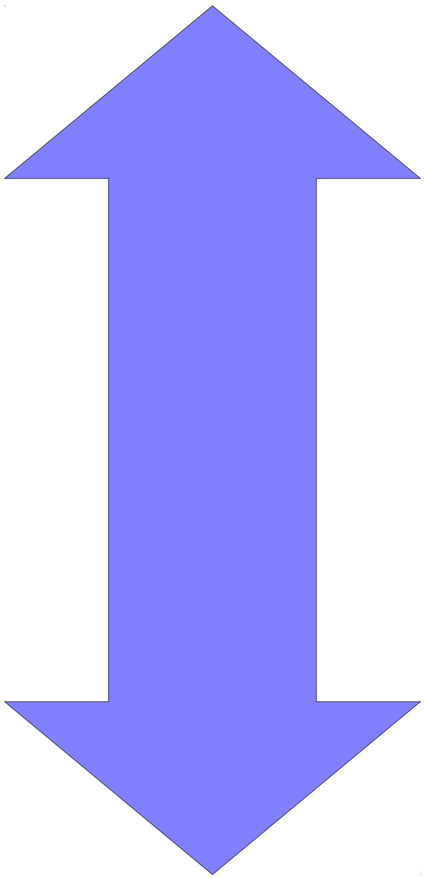
Context Switching

- 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 Fetch Execute stuff like 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...

Choosing a Timeslot Size

Longer

- 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

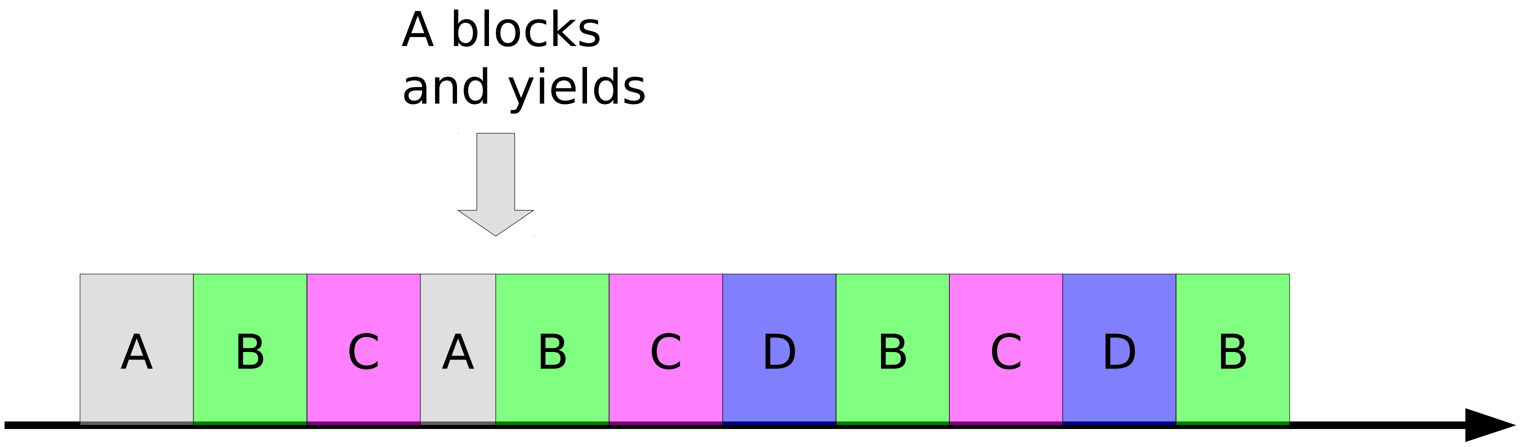


- Appears more responsive
- More time context switching means the overall efficiency drops

Shorter

Relinquishing a Timeslot Early

- 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?

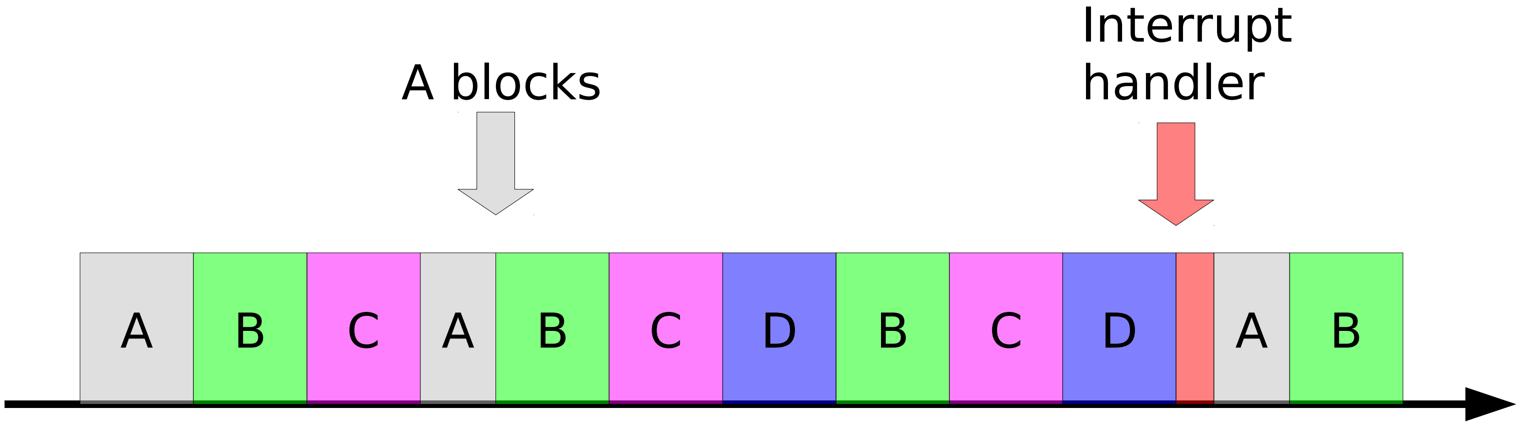


Poll

- We could periodically check ("poll") to see whether the data is there
- Essentially keep scheduling the process even though it will mostly be doing pointless checks
- Esay but obviously inefficient

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



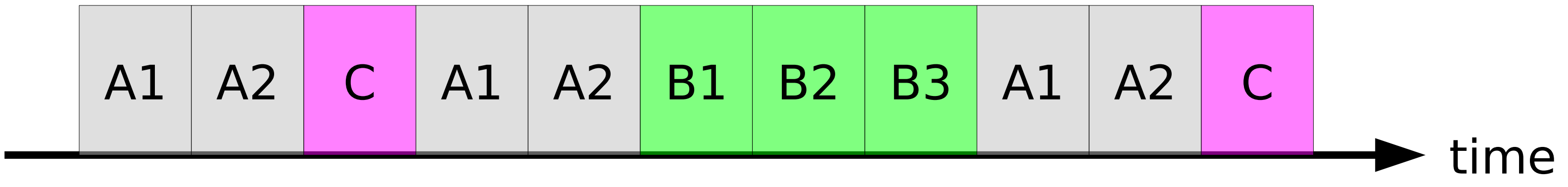
Platforms

- 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

Threads and Concurrency!

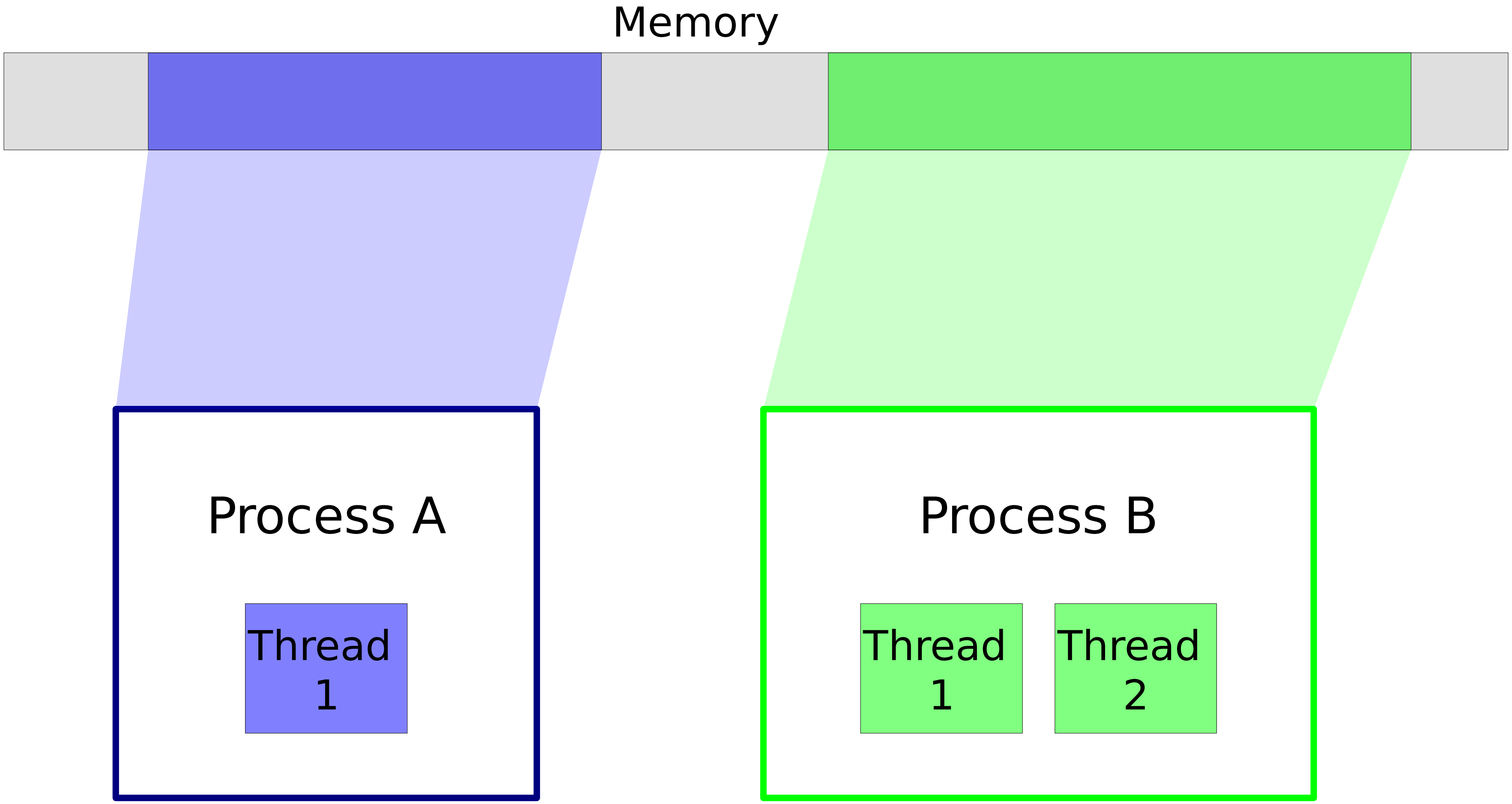
Threads

- Sometimes a program needs 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

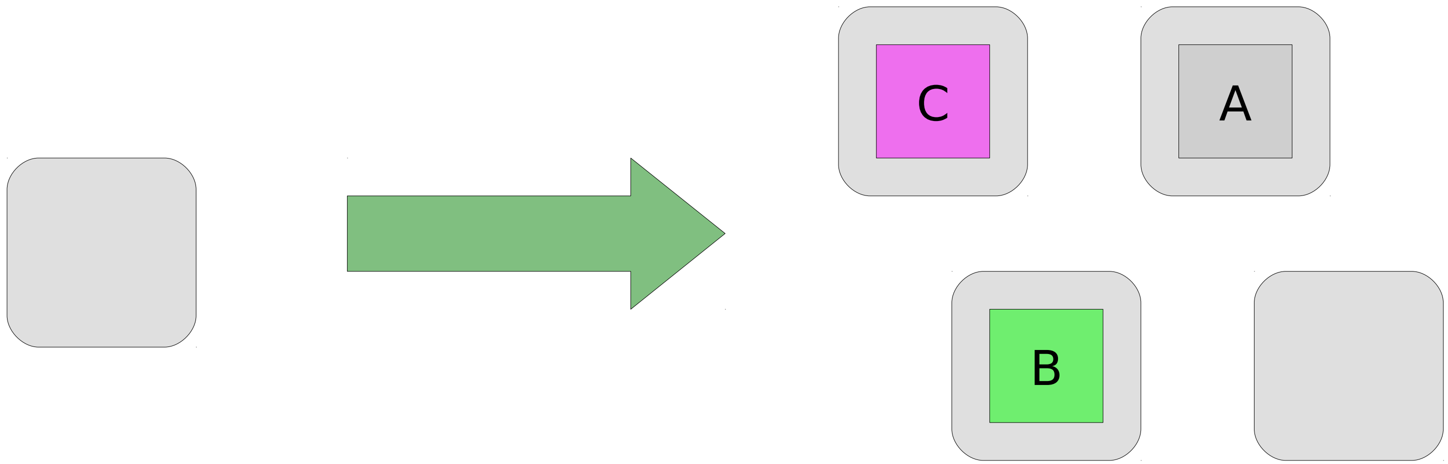
- Threads run independently but share memory



Multiple CPUs

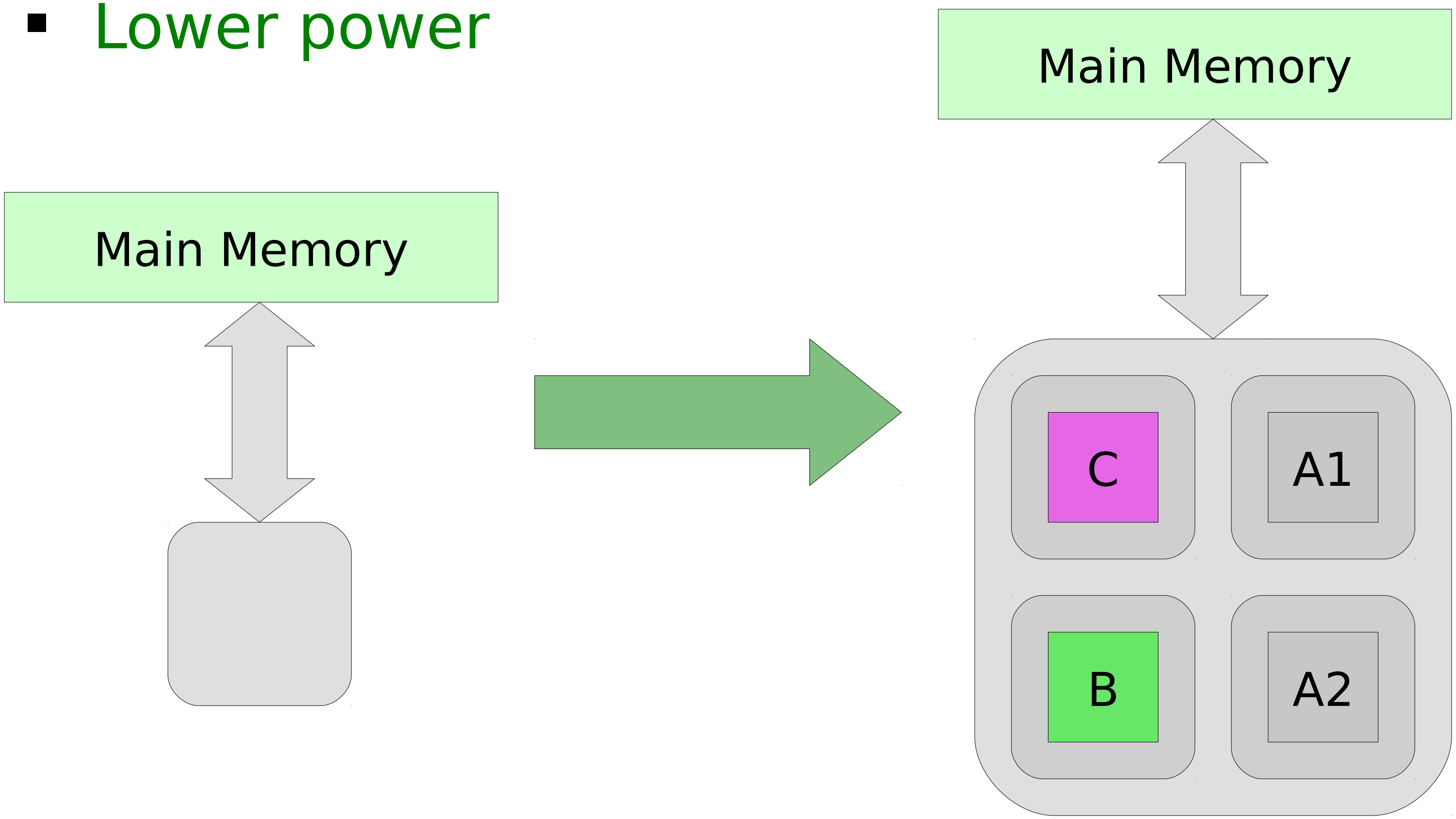
- Ten years ago, each generation of CPUs packed more in and ran faster. But:
  - The more you pack stuff in, the hotter it gets
  - The faster you run it, the hotter it gets
  - And we got down to physical limits anyway!!

- Some systems had multiple CPUs to get speed up



Multicore CPUs

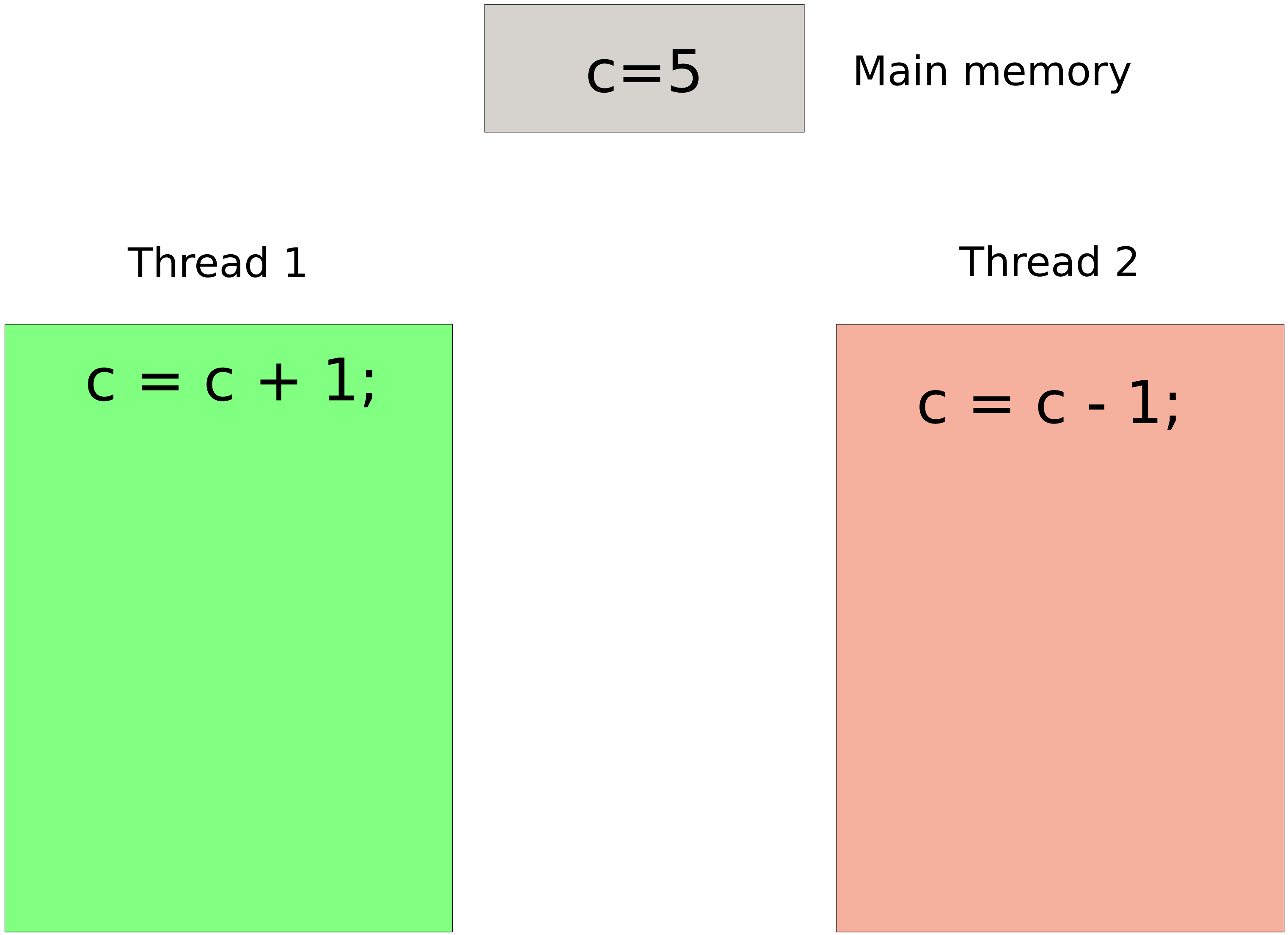
- Modern system contain chips with multiple cores: multiple CPUs in a single package
- Connections shorter → faster
- Lower power



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



Race Conditions

c=5

Main memory

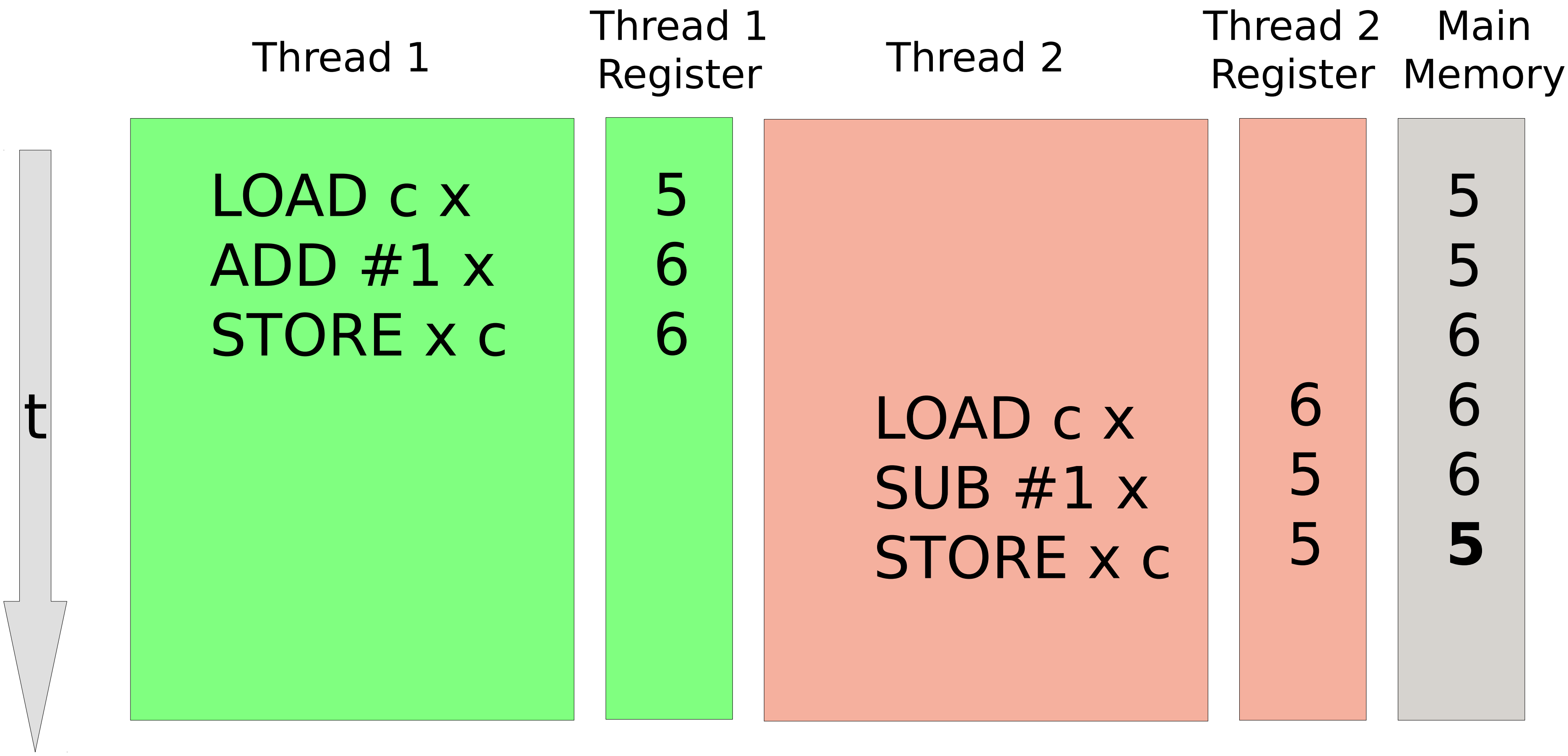
Thread 1

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

Thread 2

```
LOAD c x
SUB #1 x
STORE x c
```

Race Conditions



Race Conditions

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

Race Conditions

t

Thread 1	Thread 1 Register	Thread 2	Thread 2 Register	Main Memory
		LOAD c x	5	5
LOAD c x	5			5
ADD #1 x	6			5
STORE x c	6			6
		SUB #1 x	4	6
		STORE x 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 x
ADD #1 x
STORE x c
```

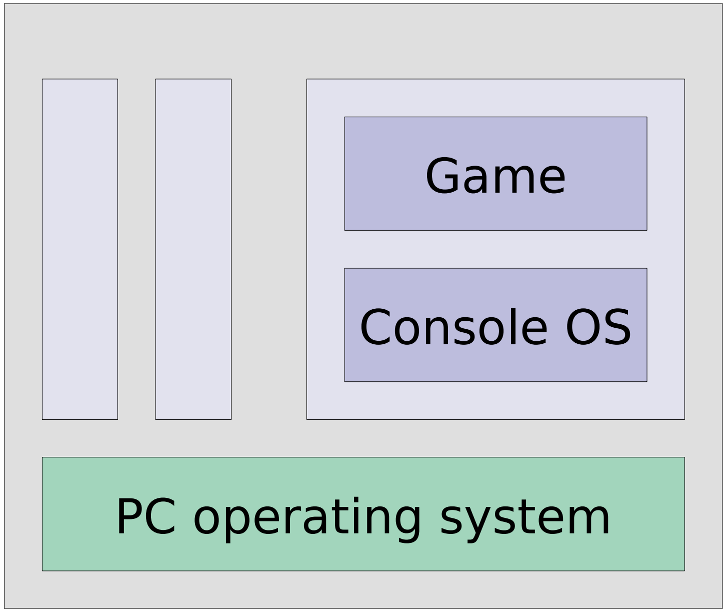
- Risky sets of operations like this must be made atomic
- i.e. no context switching once the code block is started
- *Not trivial* → much of CST IB devoted to this

Aside: The Value of Immutability

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

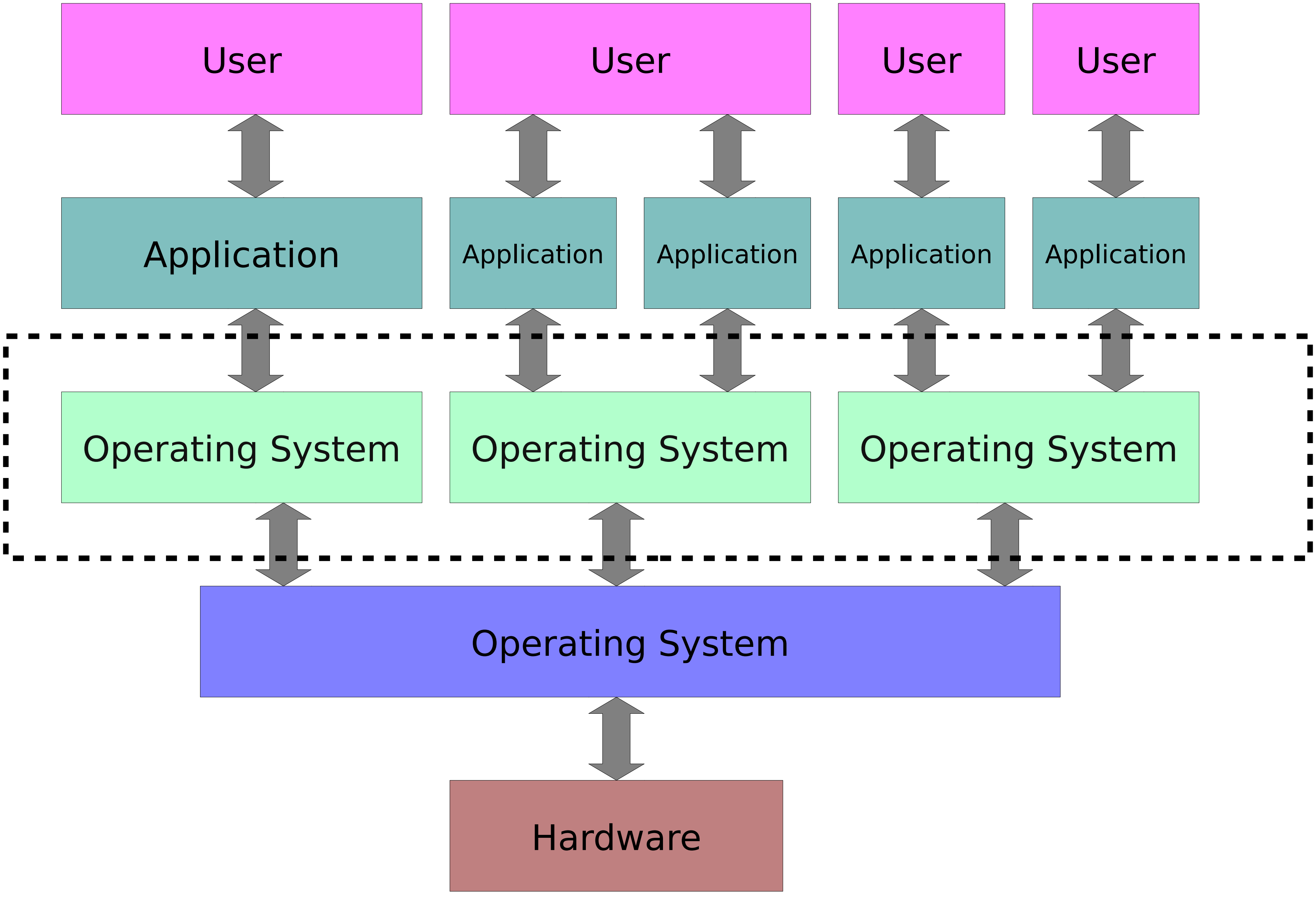
Emulation

- 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.



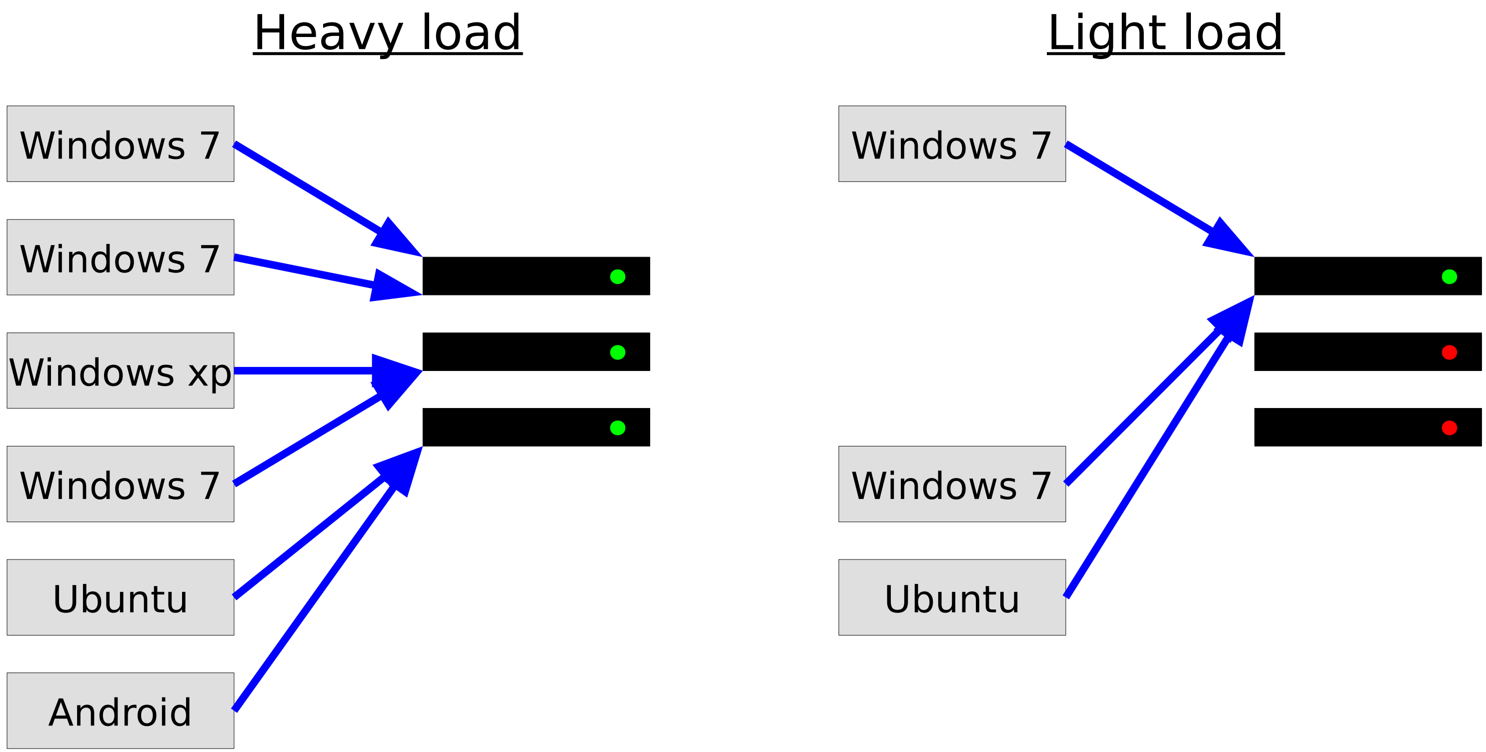
- Now computers are so fast we can run multiple *virtual machines* on them
- Allows us to run multiple operating systems simultaneously!

Virtualisation



Virtualisation

- This is time-sharing reinvented, with steroids
- Underpins the internet services we have today



So what have we learnt?

- Operating systems are complex pieces of software
- They are really a collection of management processes, each in charge of a different thing
- Multitasking is faked through timeslicing
- Multiple cores withn a CPU were introduced to boost performance on multitasking systems
- All this parallelism leads to lots of tricky concurrency issues that we're still trying to bottom out.