Session 3: Operating Systems (part 2)

Dr John Fawcett

Resource virtualisation

In the last lecture, we saw how to multiple programs can avoid overwriting each other's memory. We called this virtual memory.

Virtualisation is the process of changing our hardware and software from using physical resources to mock-up resources that are controlled by the operating system.

What else do we need to virtualise?

- The CPU itself
- Hard disks
- USB ports
- Printers
- …

Virtualising the CPU

What do we want to do?

- Key concept: consider the CPU as a resource.
  - The CPU is not ‘in charge’ of the computer.
  - Each process can get time on the CPU to run its own instructions.
- Like most resources, we have more demand than supply!
- We need a *scheduler* to decide when processes may use the CPU, and the ratio of CPU time each process can have.

Another key concept is that virtualised resources should be independent. For example, a process that is using lots of hard disk time should be able to claim its fair share of the CPU time, the USB ports, and so on. In general, processes have a fair share of each resource which they can choose to use, or not use, but there should be no penalty for using two or more resources.

Process Control Blocks (PCBs)

We introduced the idea of processes in the last lecture, and we said that each process has its own translation tables for the paging system. Now we need to know whether a process is currently running on one of the CPUs, and perhaps some information for the CPU scheduler to make fair decisions.

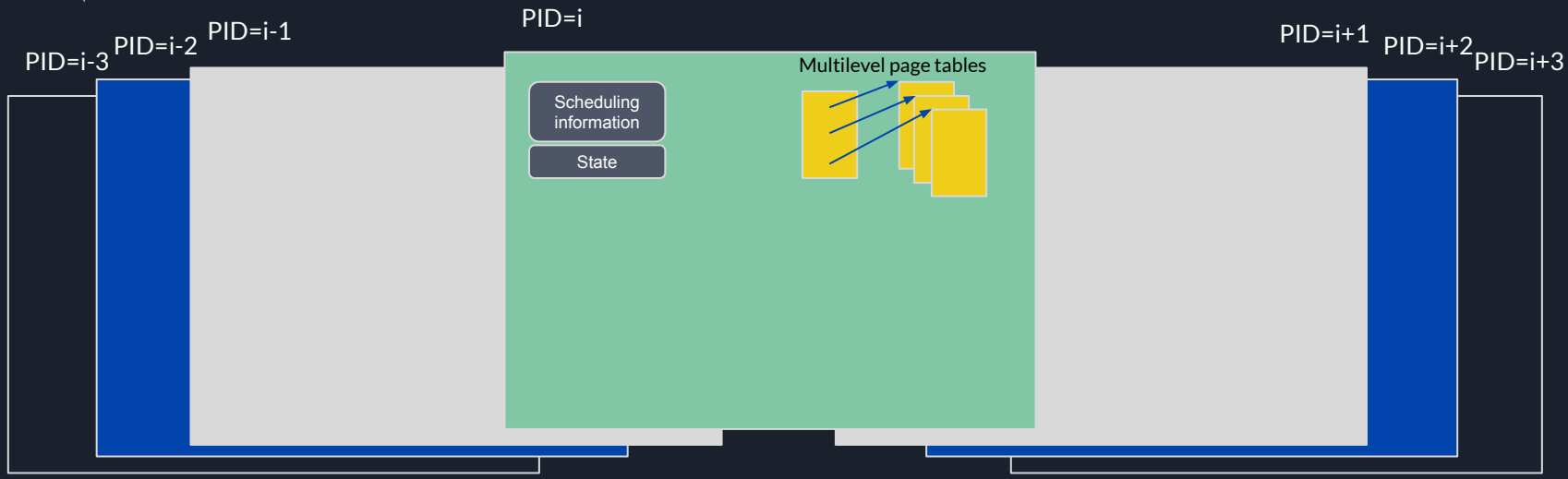
The operating system keeps all the information about a process in its *process control block*.

A PCB is a sequence of bytes, stored in physical memory. It encodes the translation tables, the state of the process, resource scheduling information, and lots more.

The operating system keeps an array of many thousands of PCBs, stored one after the other.

Each process has a unique ID which is actually just the array index for its PCB: very simple!

Process Control Blocks



What are processes?

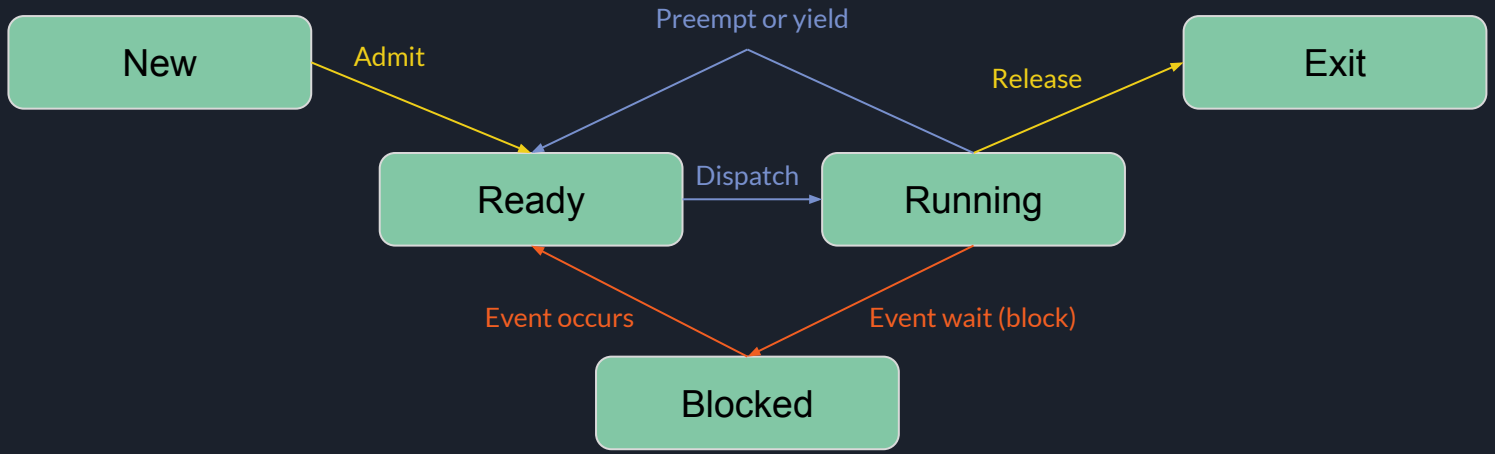
Unix model (traditional):

- Processes are the unit of
  - resource ownership
  - accounting
  - scheduling
- The kernel doesn't know about threads (which complicates how we exploit multicore processors).

Windows model:

- Processes are the unit of resource ownership and accounting
- Kernel threads are the unit of CPU scheduling

5 states of a Unix processes



The process control block stores which state a process is currently in.

New processes

There is no way to create a new process from scratch in Unix! When the operating system boots up, it creates one, special process (called *init*) to get things started.

We can only create a process by copying an existing one. This is called *forking* a process.

Any process can replace its own machine code and a common ‘trick’ is to replace yourself entirely with the machine code for a different program!

For example, if you use the ‘Start’ button to launch a new program, you are actually calling *fork* to create a second Start button, then the second Start button replaces itself with the machine code of the program you wanted to start.

This sounds crazy but is very secure: copies of your programs never gain Admin permissions!

Context switching

A context switch is how we change from running one process to another.

We might want to switch for a variety of reasons:

- The current process cannot continue until something else happens, e.g. hard disk access
- Something more important that was waiting for an event is now free to run
- The current process wants to let someone else have a go on the CPU
- The operating system wants to force the current process to let someone else have a go

The solution builds on interrupts so let's remind ourselves how interrupts work…

Interrupts

Interrupts are a new concept that we need to add to our CPU hardware and to our operating system to handle this problem. In fact, interrupts will solve several problems for us! For that reason, an interrupt comes with an integer which indicates what type of problem occurred.

Interrupts allow the CPU to tell the operating system that something has happened and that it needs the operating system to decide how to sort it out. This allows an operating system to handle the same circumstances in different ways at different times, so is much more flexible than building a single, unchangeable solution into the CPU hardware.

This is the *jump table* or *vector table*.

In physical memory beginning at address zero, the operating system puts a table of *interrupt service routines*: references to blocks of machine code that handle each type of problem. The integer is the row number so the hardware knows how to find the appropriate handler.

Handling interrupts [1/2]

When an interrupt occurs, the CPU saves the program counter (PC) in a special CPU register so it can remember where it was in the program it had been asked to run.

Then the CPU looks up the interrupt number in the jump table. The PC is set to the value in that table.

The CPU now starts executing instructions from the address found in the jump table. This machine code is trusted machine code provided by the operating system; it is not part of the program that we were trying to run.

In order to have full access to the physical memory, and any hardware devices that might be required to service the interrupt, the CPU hardware turns off the virtual address translation mechanism.

Handling interrupts [2/2]

Because the virtual to physical address translation system has been turned off, the machine code for the operating system's interrupt service routines have full access to the hardware. They are not confined to the address space of the program that was running.

The interrupt service routine does whatever is required to sort out the problem that triggered the interrupt.

Each interrupt service routine ends with a special instruction, RET in assembly code, which turns the translation mechanism on again and restores the PC to the value saved in that special register. This takes us back to where we were in the program that was running as if nothing had happened!

Hard disk requests

When a process needs to load some data from the disk, its own machine code cannot continue until that data has been loaded.

We need to:

1. Send a command to the hard disk to load the data we want
2. Remember where we were up to in this process's machine code
3. Run a different process until the hard disk finishes loading the data
4. Return to this process when the hard disk has finished

Sending commands to the hard disk

A program cannot talk to the real hard disk because it has been virtualised!

It talks to a mock-up hard disk, provided by the operating system.

If the real hard disk is busy, the operating systems puts this command on a queue; otherwise, it sends the command straight to the real hard disk.

That implements step 1!

Next: step 2, “Remember where we were up to in this process’s machine code”

Remembering where we were up

The process we have been running has got some information in the CPU that we would lose if we simply starting running a different process:

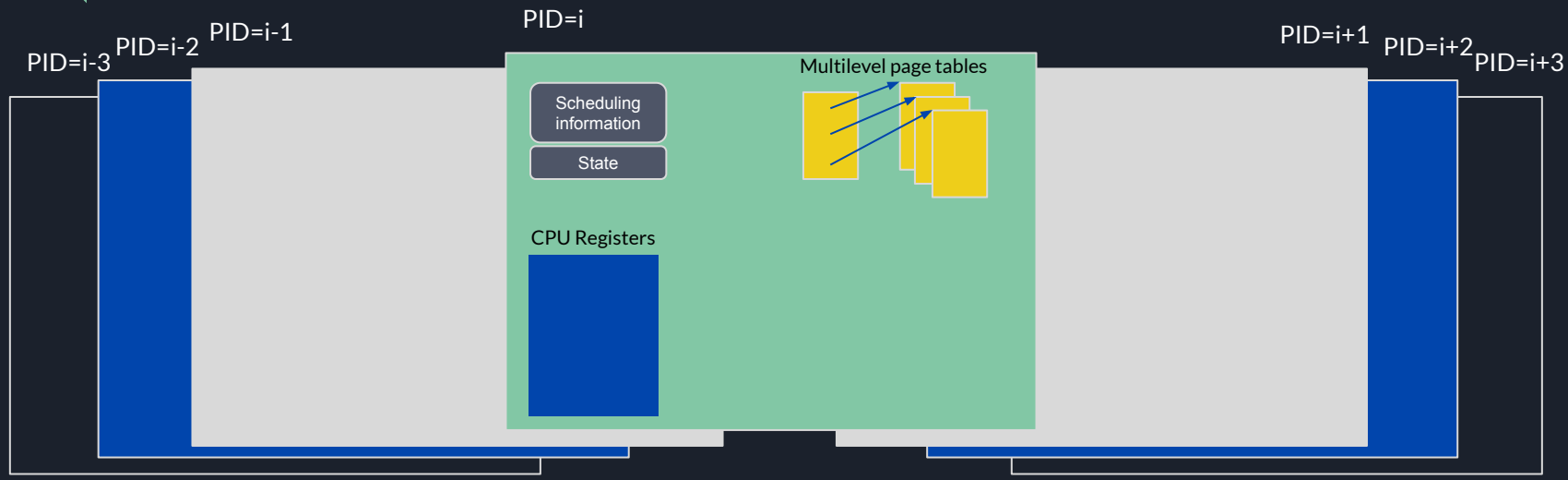
1. The numbers stored in the CPU registers
2. The program counter

We copy the registers and the program counter to the process's PCB! When we want to continue running this process from where we were, we can copy the values back into the CPU registers and program counter, and we'll be exactly where we were!

We set this process's state to BLOCKED. That implements step 2!

Next, step 3: "Run a different process until the hard disk finishes loading the data"

Process Control Blocks



Running a different process

Remember that, when we return from interrupts with a RET instruction, we will put the saved program counter back.

Suppose we change that program counter before we run RET.

After the operating system has sent a command to the disk, it…

- Runs the scheduling algorithm to decide which process to run next
- Copies the saved registers and program counter from that process's PCB into the CPU
- RET: the CPU thinks it's going back to where it was but we've changed where that is!

That implements step 3! Step 4, "Return to this process when the hard disk has finished" is very similar…

Returning to the blocked process

Whenever the disk finishes a command, it generates an interrupt.

The CPU hardware looks up the interrupt number in the jump table and runs the machine code it finds there.

This machine code is the operating system's interrupt service routine for the hard disk, which…

- Sends the next command from the queue to the disk, if there is one
- Updates the blocked process's state to READY
- Runs the scheduler to decide whether to continue with the current process or switch back. To switch back, we copy the registers and PC into this process's PCB and put back the registers and PC from the process that was waiting.
- Runs RET.

Reflections on the mechanism

Sounds very complicated.

Actually very simple!

It's very efficient because the operating system isn't wasting time checking to see if the hard disk has finished yet, nor doing anything else that is unnecessary.

The operating system wants to spend as little time on CPU as possible because, while it is running, the user's programs cannot be running. The user only cares about their programs!

Now we have a way to *context switch* between processes, we can think about how the operating system might force a process to give up the CPU – *preemption*.

Preempting processes

Some processes run for a very long time without stopping: weather simulations, colliding galaxy simulations, ray tracers; and some programs have bugs that make them go into infinite loops.

The operating system needs to be able to take these processes off CPU… but the operating system isn't running so how can it take control?

Interrupts help us, once again!

Timer interrupt

Modern CPUs have a programmable timer: a clock that ticks at a programmed frequency. Most operating systems set the timer to ‘tick’ every 1 ms.

A 4 GHz CPU will do 4 billion instructions per second, so 4 million instructions per millisecond.

When the timer ‘ticks’, it generates an interrupt. The CPU looks in the jump table again and runs the interrupt service routine it finds there.

The operating system’s interrupt service routine for the timer will subtract one from a counter that we set to 100 whenever we context switch. Whenever the counter reaches zero, we run the CPU scheduler algorithm and context switch again.

Any program that does not ‘block’ within 100ms will be paused automatically: *preemption*.

Yielding

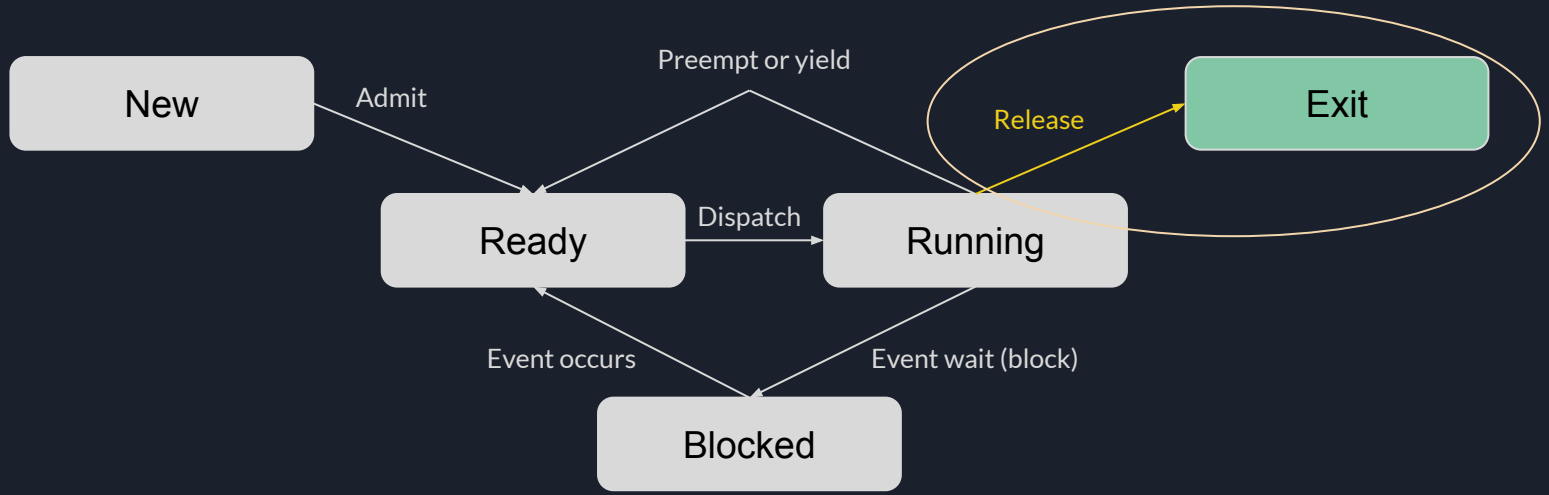
A process can also move from RUNNING to READY by *yielding*. Yielding is like preemption but the running process chooses to trigger a context switch, rather than the timer interrupt service routine forcing a context switch.

Well-behaved programs yield whenever they have finished their important work and their next task is less important and could happen later.

Co-operative multi-tasking operating systems rely on programs to call yield! They are much less reliable because buggy programs will forget to do it, but they do not need a hardware timer so were the first to be invented (Microsoft Windows 3.0) and are sometimes used in embedded systems.

yield runs the scheduler immediately and context switches if there's a more important process.

Release


New
Admit
Preempt or yield
Release
Exit
Ready
Dispatch
Running
Event occurs
Event wait (block)
Blocked

Now it’s time for you to have a go

Exercise sheet!