Session 2: Operating Systems (part 1)

Dr John Fawcett

Why do we need an Operating System?

In the last lecture, we saw how a CPU runs instructions and how we can build large programs using loops, conditional branches and unconditional jumps for subroutines.

What happens when we have lots of programs and we want to be able to pause one task to run another one?

- We need to be able to run more than one program on our computers at the same time.
- We need to stop one program overwriting the data of another one.
- We need to share resources – printers, the hard disk, the network.
- We need to share data with permissions.

Today's topic: running multiple programs

Problem: when we wrote our assembly code, we used *absolute addresses* in instructions such as LD R0 #4096. What if two or more programs use address 4096 at the same time? Each time one program stores data to that address, another program's data would be overwritten.

Solution 1: we could write our programs to know about each other and make sure we only use the same addresses when we mean to share data between programs.

- This is difficult to get right, and impossible if we have software developers all around the world who don't even know each other.

Solution 2: virtual memory

Virtual memory

- Give each program instance – known as a *process* – the impression that it is the only one running on the computer.
- Effectively, the operating system *kernel* creates the illusion that there is nothing else to worry about.
- Virtual memory is the application of this principle to the computer's memory
  - We will need to do the same for the hard disk, the printer, the network, etc.
- Each process runs in a *virtual address space*.
  - Works just like the memory we saw last time but the operating system changes where the data is stored in main memory so two programs do not overlap.
- We can do this in a few ways; we will look at splitting the memory into *pages*.

Pages in decimal

Decimal example:

- Suppose we have 1000 addresses (0..999).
- We divide into pages, each with 10 addresses. There will be 1000/10 = 100 pages.
- When we write down an address, e.g. 123, the rightmost digit (least significant) tells us which address within the page we are talking about and the other digits tell us which page we are talking about.
  - 123 ⇒ address 3 in page 12
  - 801 ⇒ address 1 in page 80
  - 077 ⇒ address 7 in page 07

We do the same thing for the computer, but in binary.

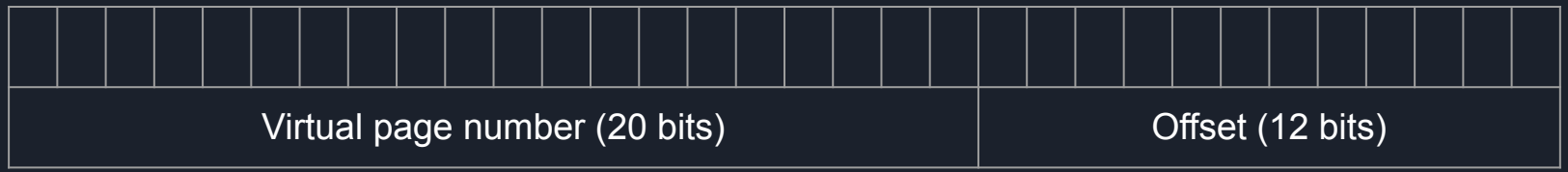
Pages in binary

We said that a 32-bit address gives us 2^{32} = 4,294,967,296 addresses, each holding 1 byte (for a byte-addressed memory).

We divide that into blocks, usually 2^{12} = 4096 bytes.

There will be 2^{32} / 2^{12} = 2^{20} pages (about one million, 1,048,576).

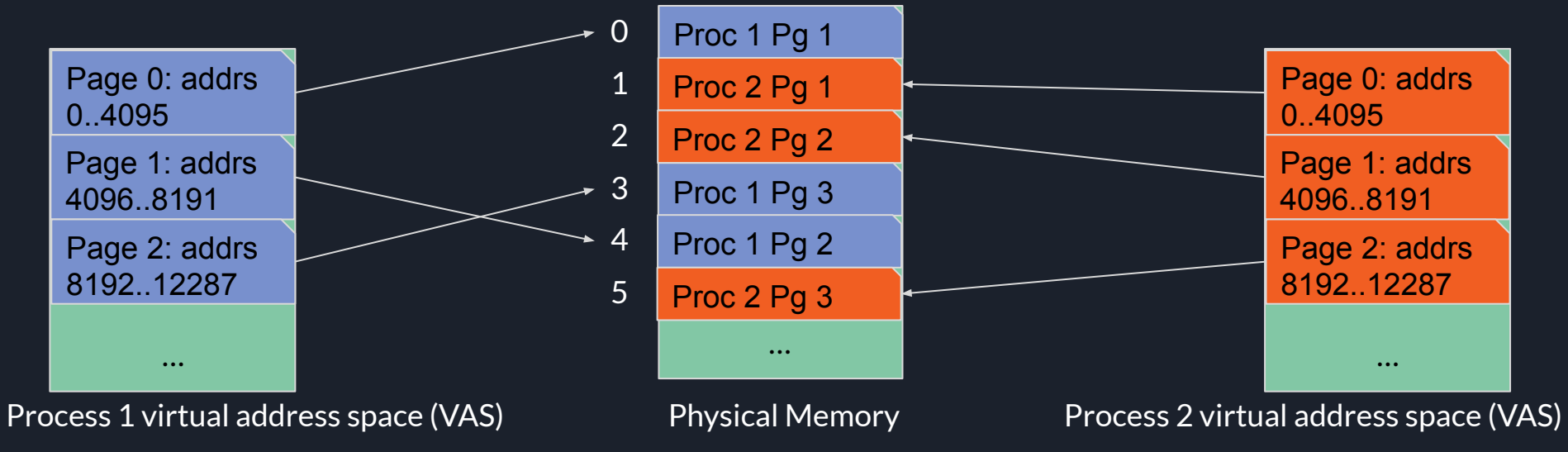
The least significant 12 bits of any address specify which address within a page we are talking about. The other 20 bits specify which page it is.



Preventing overwrites

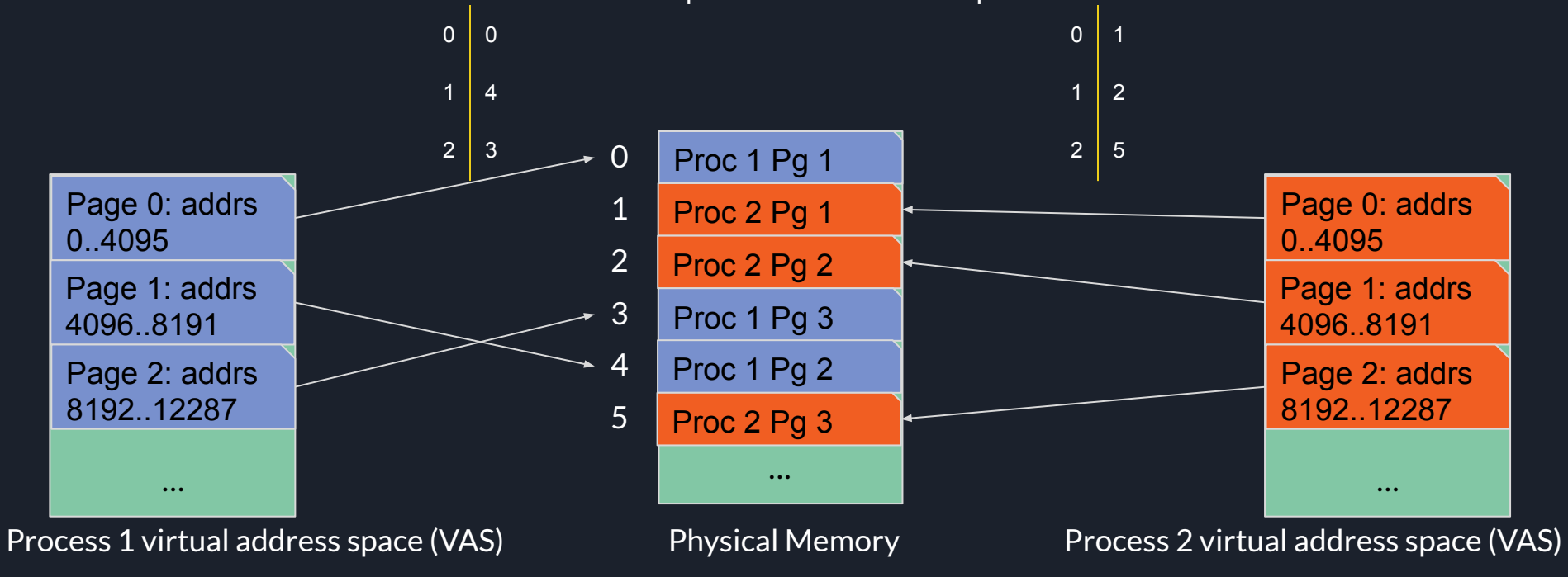
Each program thinks that it has the whole 2^{32} byte *address space* to itself.

The operating system records where pages really are in the physical memory and makes sure that we never put two pages in the same place.



Preventing overwrites

We need a translation table for each process. For these two processes we have…



Preventing overwrites

Let's see how to use our translation table. Suppose process 1 loads data from address 4096: LD R0 #4096. The address looks like this:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Virtual page number (20 bits) = 1																			Offset (12 bits) = 0										

We translate the virtual page number using our table: 1 ⇔ 4, we get 16384.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Physical frame number (20 bits) = 4 (*translated*)																			Offset (12 bits) = 0 (*same*)										

What about process 2?

If process 2 loads data from address 4096 with the same instruction, LD R0 #4096, the virtual address is exactly the same:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Virtual page number (20 bits) = 1																			Offset (12 bits) = 0										

But when we translate it using process 2's table: 1 ⇔ 2, we get 8192. Collision avoided!

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Physical frame number (20 bits) = 2 (*translated*)																			Offset (12 bits) = 0 (*same*)										

Where does translation happen?

- We have seen that when the CPU is running one sequence of instructions, that's all it's doing.
- When a particular program is running, the operating system cannot be running too.
- We cannot ask the operating system to translate these addresses.
- We could ask all application developers to use these tables to translate their addresses before they issue LD / ST instructions.
  - Would you trust them to get it right?
  - Leaves memory open to attack by malicious programs
- We need some new hardware support so the hardware automatically translates addresses
  - And, for security, there needs to be no way to turn off the translation system!

Operating system involvement

The operating system creates the translation tables when a process is started.

The operating system gives the translation table to a special unit inside the CPU called the *translation lookaside buffer* (TLB).

Whenever the process runs a LD or ST instruction, the CPU uses the TLB to translate the address before accessing memory.

Demand paging

Our idea turns out to work very well!

It's so easy for application programmers to use that lots of people around the world started writing programs. Now we have another problem: memory isn't big enough for all the programs we want to run at once!

Paging comes to the rescue again!

When we run out of memory in the DRAM, we can copy pages that aren't being used right now onto the hard disk. Now we can overwrite them in memory and keep the *working set* of pages for each process in memory, allowing us to have lots more programs open at the same time, without losing all the rest of their memory. Let's work out how to implement this…

Resident bits

We can keep one extra bit per entry in the translation tables: the R bit. R means Resident: a page is *resident* in memory if it is currently stored in memory (R=1), while R=0 indicates that the page has been moved out to the hard disk.

When a page is not resident, the bits allocated to the physical frame number are repurposed to tell us where on disk we saved the page.

If the running process tries to use an address that translates to a non-resident page, we have a problem!

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 has 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!

Page fault interrupt

When the TLB attempts to translate a virtual address but discovers that the corresponding translation table entry is non-resident, the TLB raises an interrupt with the integer corresponding to the code number for a page fault.

The CPU saves the program counter (PC) into the special backup register.

The CPU looks up the interrupt number and starts running the interrupt service routine.

The interrupt service routine needs to select a *victim page* to move out of DRAM onto the disk, in order to free up one page frame. Then it sends a command to the hard disk to copy the victim page from memory to the disk; then to copy the required page into memory, overwriting the victim; and finally we execute RET to take us back to where we were. The program continues as if nothing had happened!

Optimisations

1. Hard disks are slow compared to the CPU: if we waited for the hard disk, we would waste around 100,000 CPU clock cycles when we could have been running instructions!

   The operating system will usually decide to run a different program while it waits for the hard disk, keeping a note that it can return to this one when the hard disk has finished copying data.

2. If the victim page has not been written to since it was pulled in from the disk, there is no need to write it out again – it's the same as the copy on disk! By adding another *dirty bit* (D) to the translation table entries, we can keep track of whether a page needs to be copied out before it can be overwritten.

Security

Notice that each process can only access the areas of physical memory referred to by its own translation table. This prevents every program from reading or modifying the memory that is in use by every other program!

To keep this secure, we must ensure that no program can modify any of the translation tables! If they could, they could give themselves access to any area of physical memory they liked.

The kernel keeps the translation tables in physical memory but in places that are not the target of any process's virtual to physical mapping. This is sufficient to prevent any process from changing any translation table.

Multilevel paging

An annoying problem we have at the moment is that the translation tables are pretty large and could consume a lot of the physical memory.

We need to be able to push the translation tables for dormant processes out to the hard disk.

We already have a mechanism capable of doing this for pages of data – so we just need to make sure our page tables are 1 page in size!

We build a tree of tables: each process will have a single, top-level (“root”) table, known as a page directory. The entries in the page directory are the physical frame numbers where small pieces of the page table can be found. For 64-bit operating systems, there are often several levels, not just two.

Now it’s time for you to have a go

Exercise sheet!