COMS20012 Segmentation, Pages and Memory Protection

Joseph Hallett

What’s all this about?

- Many processes run at the same time in separate address spaces
  – Isn’t *virtual memory* neat?

- So how are we going to get the OS to deal with getting the right bit of memory into the right place at the right time?

- …and what can we do with with these mechanisms once we’ve got them going

Here be dragons!

- This stuff is super confusing, and we're going to focus on the mechanisms for the Intel x86 architecture which is especially weird, arcane and tricksy
- If in doubt, go read Volume 3, Chapters 3–5 of the *Intel 64 and IA-32 Architectures Software Developer's Manual* which gives details
- Get the broad ideas down… go into more detail if you need/want!

Segmentation

- We want different programs and tasks to run on a single processor without interfering with each other and the OS kernel…
- Code and stacks should all seem to start at the same addresses yet should really refer to different bits of memory…
- Oh and we might have more memory available than our CPU can simply address with an *n*-bit register…
- So how are we going to do this?

Segment Registers

- The X86 instruction set has a bunch of *segment registers*
  - As well as all the standard RAX/RBX/RIP/RBP registers…

- *CS* is the code segment
- *DS* is the data segment
- *SS* is the stack segment
- *ES* is the extra segment *(used for strings mostly)*
- *FS/GS* are general purpose segments

- Pointers are treated as an offset from one of these segments
- *Global Descriptor Table (GDT)* says where all these s
- Each segment can have a set of *access* permissions

And what happens with these all these segment registers?

(Basically, it all goes down a rabbit hole of tables and virtual address spaces…)

(let's focus on the left-hand side for now)

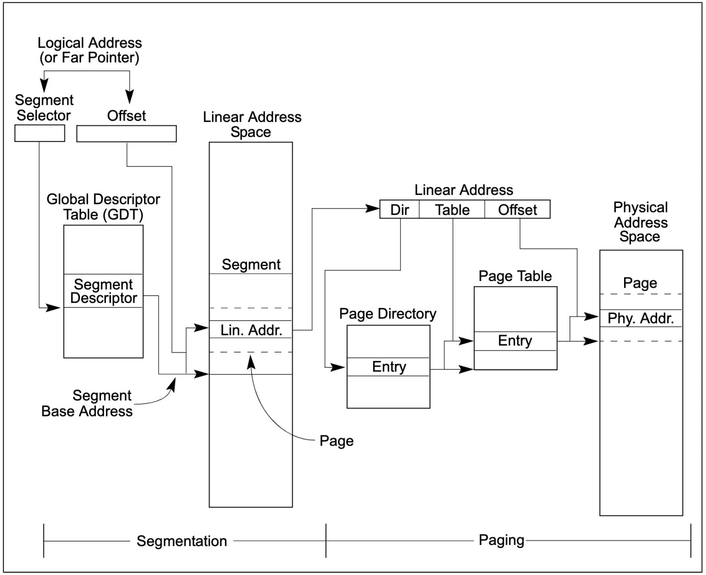

Figure 3-1. Segmentation and Paging

Flat Model

(Segments for code and data/stack… try and pretend it doesn't really exist)

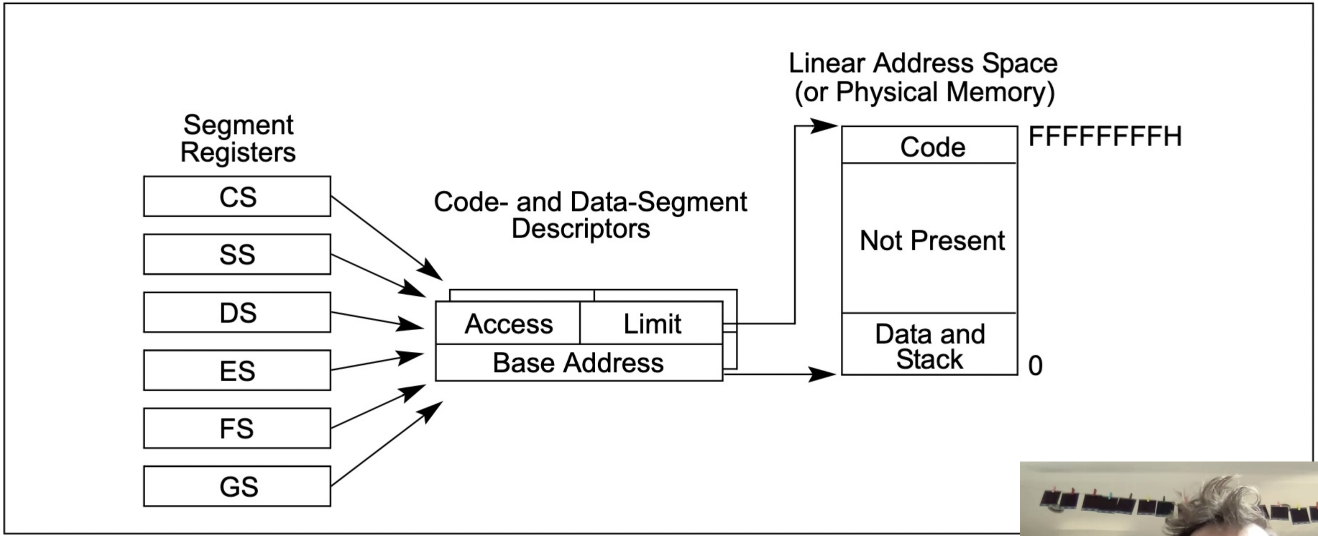

Figure 3-2. Flat Model

Protected Flat Model

(Separate segments for ring 0 and ring 3 code and data… maybe more if helpful)

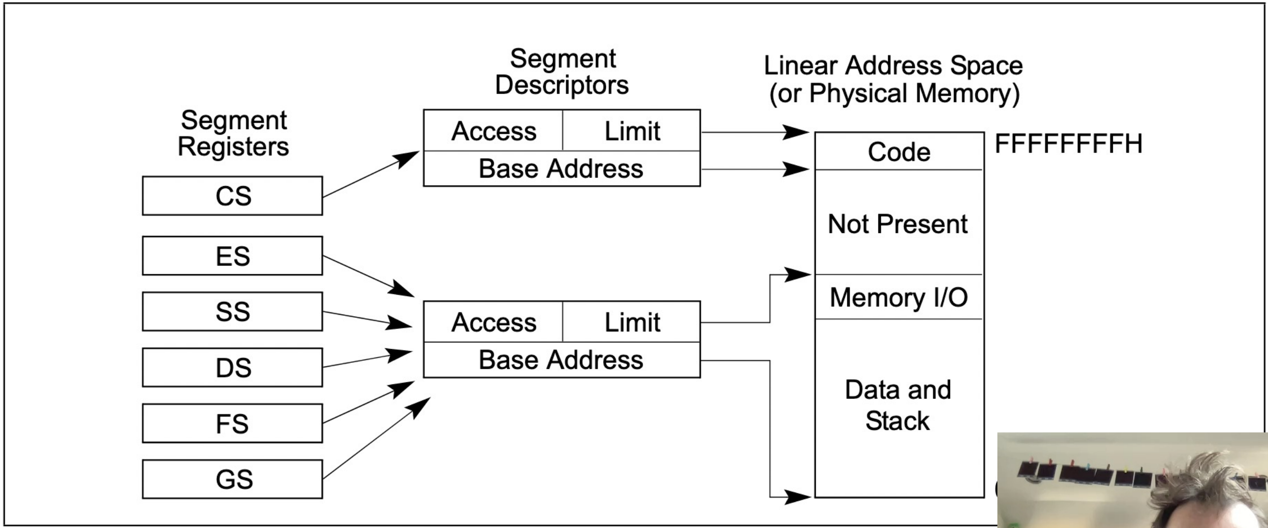

Figure 3-3. Protected Flat Model

Multi-segment model

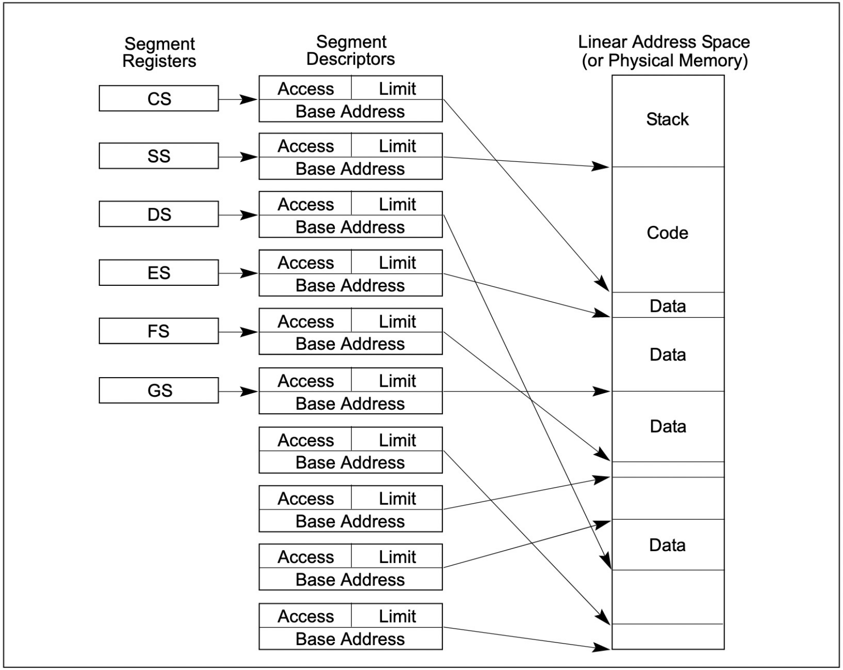

Figure 3-4. Multi-Segment Model

- Lots of segments for every program and process
- Everything isolated from each other

(This is the one we use)

Next problem!

So far segmentation lets us divide up linear memory (the big virtual address space) into smaller areas, per process/ring/whatever

- Limited scope for changing segments within userland
- Limited permissions
  - (read only, execute only, read and execute, read and write… *a few more*)

Sometimes we'd like even finer grained permissions…

- How are we going to do this then?

Paging

Segmentation splits up the virtual address space…

- …Which is then mapped onto page entries which map to physical address space
- …And a whole load more tables and permissions

Kernel is allowed to switch pages in and out of memory

- If a process tries to access memory in a different page a PAGEFAULT trap occurs and the kernel can decide what to do
- (*Probably* swap in the right page for the process and let it continue… which is slow)

Paging rules

- Pages are *at least* 4KB on every platform you will ever care about
- Pages cannot span different physical chips
  - The memory has to be physically contiguous

- Pages can configure how processes access physical memory
  - i.e. are writes batched or done directly/rollback
  - (See *Spectre and Meltdown* in COMSM0049 ;-))

- You can disable paging if you *really* want…
  - But then you lose virtual memory and have to deal with the physical address space, and it makes everything worse
  - (This is necessary for some low-level, BIOS/EFI level code)

Paging permissions

- For each page you can set a bunch of extra permission *if your CPU supports it*
  - Almost all architectures have some kind of permissions available

- The big one you need to know about is W^X
  - If you can *write* to memory you shouldn't be able to *execute* it
  - Stops a malicious user finding a buffer overflow, injecting a program, and trivially executing it…*ish*
  - But it also slows program loading down…

JITing and W^X

- Say I have a program that takes code written by users, and then compiles and runs it on the fly…
- Oh and the code can be dynamically generated and change at short notice
  - Like the JavaScript engine running JS in your web browser

- What has to happen to make this work?

JITing and W^X is slow

1. Get the code chunk you want to compile into memory
2. Compile it and write that code into memory
3. Stick a bit of code on the end to go fetch the next bit of code to compile and repeat
4. Run it

If we have W^X bits we need to make an extra system call between 3 and 4, and before 1 to ensure that the area of memory we're using to load our code into is writable or executable between every chunk.

2 extra syscalls per *Just In Time-compiled* chunk…

We’re done!

- Congratulations we’ve reached the end!

- We’re all new lecturers here…
- This was our first year running this unit (and the second year its ever run)…
- Thank you for baring with us!

- Hopefully you enjoyed it…
- Feel free to grab us and chat if you want to talk operating systems and security

Exam

We've set the exam…

- 8 short questions… 5 marks each
- 4 long questions… 15 marks each
- Equally distributed over the content and reading for the lectures
- Open book

Tips:

- If a question is worth 5 marks give me 5 things to tick
- If a question says *debate*, then make an argument both ways or at least say why the other way is wrong
- Remember that we want you to do well!