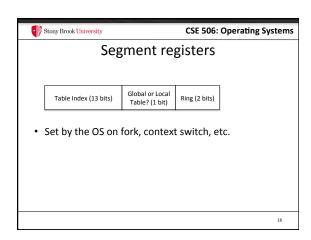
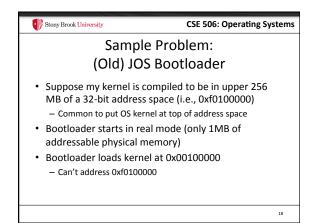


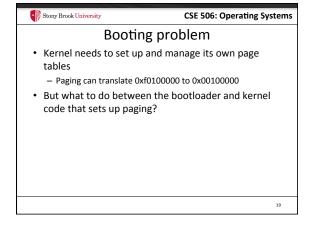
Segment management

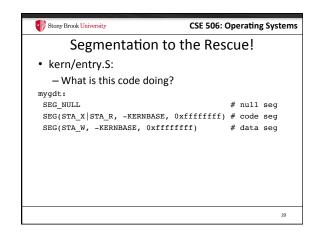
For safety (without paging), only the OS should define segments. Why?

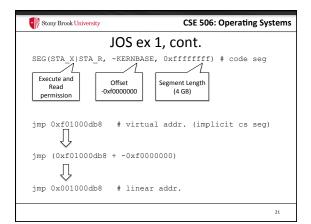
Two segment tables the OS creates in memory:
Global – any process can use these segments
Local – segment definitions for a specific process
How does the hardware know where they are?
Dedicated registers: gdtr and ldtr
Privileged instructions: lgdt, lldt

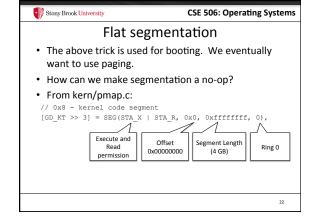


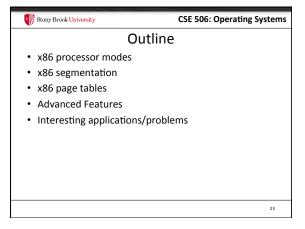


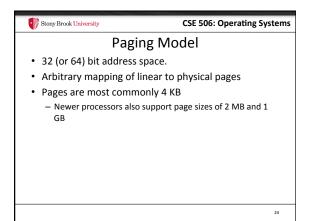


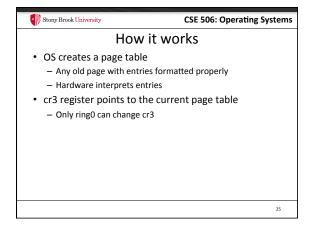


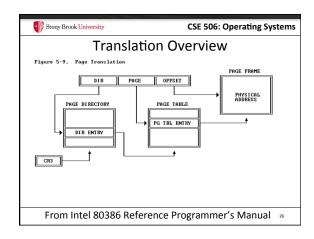


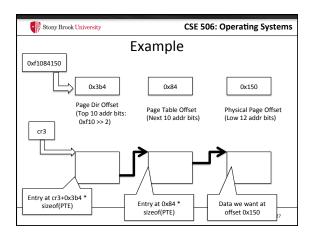


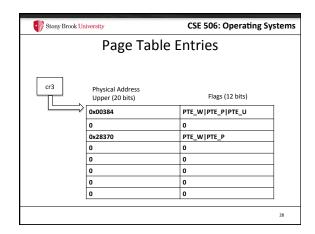


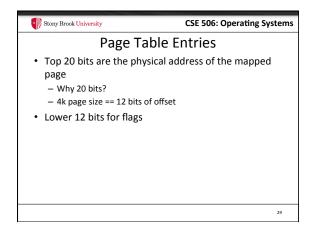


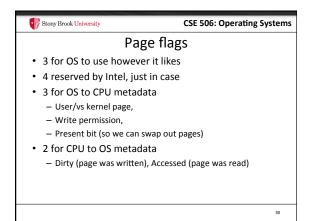


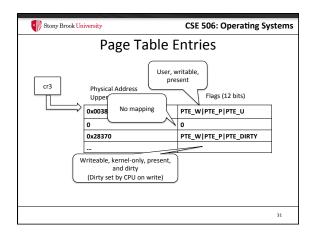


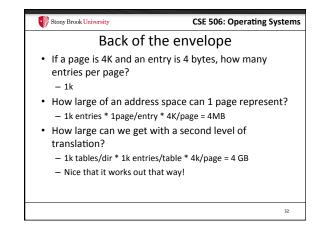


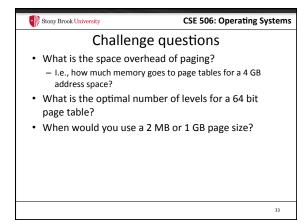


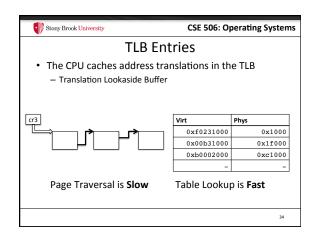


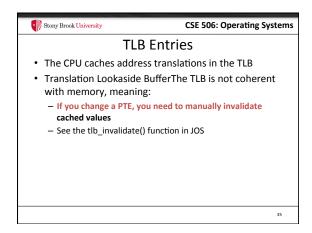


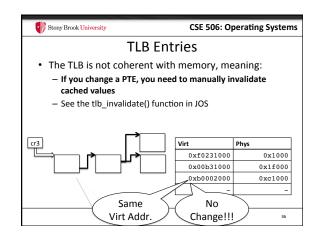


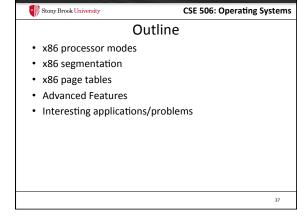


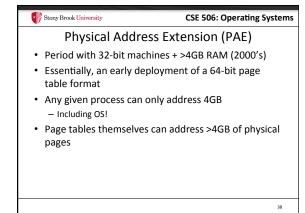


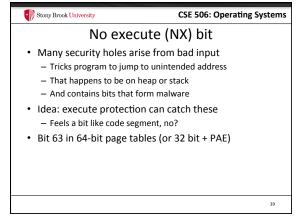


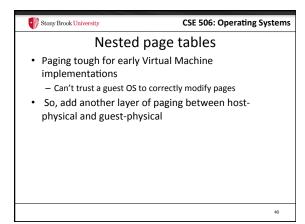








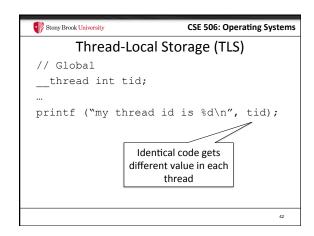




Stony Brook University

CSE 506: Operating Systems

And now the fun stuff...





- Convenient abstraction for per-thread variables
- Code just refers to a variable name, accesses private instance
- Example: Windows stores the thread ID (and other info) in a thread environment block (TEB)
  - Same code in any thread to access
  - No notion of a thread offset or id
- How to do this?

43

CSE 506: Operating Systems

TLS implementation

• Map a few pages per thread into a segment

• Use an "extra" segmentation register

— Usually gs

— Windows TEB in fs

• Any thread accesses first byte of TLS like this:

mov eax, gs:(0x0)

CSE 506: Operating Systems

TLS Illustration

Oxb0001000

Oxb0002000

Tid = 0

Tid = 1

Set by the OS kernel during context switch
Thread 0 Registers gs:= Oxb0001000

Thread 0 Registers gs:= Oxb0001000

printf ("My thread id is %d\n", gs:tid);

Viva segmentation!

• My undergrad OS course treated segmentation as a historical artifact

- Yet still widely (ab)used

- Also used for sandboxing in vx32, Native Client

- Used to implement early versions of VMware

• Counterpoint: TLS hack is just compensating for lack of general-purpose registers

• Either way, all but fs and gs are deprecated in x64

CSE 506: Operating Systems

Microsoft interview question

Suppose I am on a low-memory x86 system (<4MB). I don't care about swapping or addressing more than 4MB.

How can I keep paging space overhead at one page?

Recall that the CPU requires 2 levels of addr. translation

CSE 506: Operating Systems

Solution sketch

A 4MB address space will only use the low 22 bits of the address space.

So the first level translation will always hit entry 0

Map the page table's physical address at entry 0

First translation will "loop" back to the page table

Then use page table normally for 4MB space

Assumes correct programs will not read address 0

Getting null pointers early is nice

Challenge: Refine the solution to still get null pointer exceptions

8

