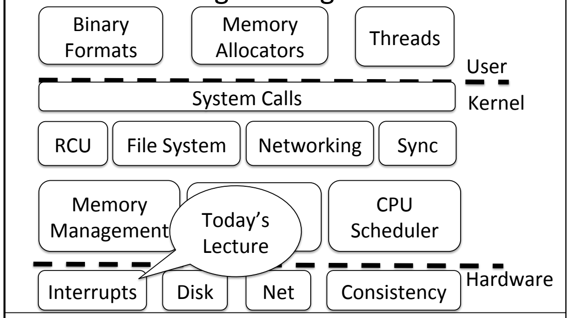
x86 Memory Protection and Translation

Don Porter

Logical Diagram



Today's Lecture: Focus on Hardware ABI

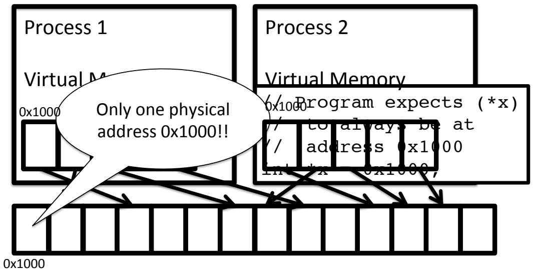
Lecture Goal

- Understand the hardware tools available on a modern x86 processor for manipulating and protecting memory
- Lab 2: You will program this hardware
- Apologies: Material can be a bit dry, but important
  - Plus, slides will be good reference
- But, cool tech tricks:
  - How does thread-local storage (TLS) work?
  - An actual (and tough) Microsoft interview question

Undergrad Review

- What is:
  - Virtual memory?
  - Segmentation?
  - Paging?

Memory Mapping



```
// Program expects (*x)
// to always be at
// address 0x1000
int *x = 0x1000;
```

Two System Goals

1) Provide an abstraction of contiguous, isolated virtual memory to a program

2) Prevent illegal operations

- Prevent access to other application or OS memory
- Detect failures early (e.g., segfault on address 0)
- More recently, prevent exploits that try to execute program data

Outline

- x86 processor modes
- x86 segmentation
- x86 page tables
- Advanced Features
- Interesting applications/problems

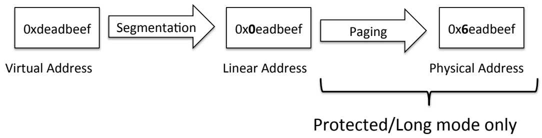
x86 Processor Modes

- Real mode – walks and talks like a really old x86 chip
  - State at boot
  - 20-bit address space, direct physical memory access
    - 1 MB of usable memory
  - Segmentation available (no paging)
- Protected mode – Standard 32-bit x86 mode
  - Segmentation and paging
  - Privilege levels (separate user and kernel)

x86 Processor Modes

- Long mode – 64-bit mode (aka amd64, x86_64, etc.)
  - Very similar to 32-bit mode (protected mode), but bigger
  - Restrict segmentation use
  - Garbage collect deprecated instructions
    - Chips can still run in protected mode with old instructions
- Even more obscure modes we won't discuss today

Translation Overview



- Segmentation cannot be disabled!
  - But can be a no-op (aka flat mode)

x86 Segmentation

- A segment has:
  - Base address (linear address)
  - Length
  - Type (code, data, etc).

Programming model

- Segments for: code, data, stack, "extra"
  - A program can have up to 6 total segments
  - Segments identified by registers: cs, ds, ss, es, fs, gs
- Prefix all memory accesses with desired segment:
  - mov eax, ds:0x80 (load offset 0x80 from data into eax)
  - jmp cs:0xab8 (jump execution to code offset 0xab8)
  - mov ss:0x40, ecx (move ecx to stack offset 0x40)

Segmented Programming Pseudo-example

```
// global int x = 1
int y; // stack
if (x) {
      y = 1;
      printf ("Boo");
} else
      y = 0;
```

```
ds:x = 1; // data

ss:y; // stack

if (ds:x) {

      ss:y = 1;

      cs:printf
         (ds:"Boo");

} else

      ss:y = 0;
```

Segments would be used in assembly, not C

Programming, cont.

- This is cumbersome, so infer code, data and stack segments by instruction type:
  - Control-flow instructions use code segment (jump, call)
  - Stack management (push/pop) uses stack
  - Most loads/stores use data segment
- Extra segments (es, fs, gs) must be used explicitly

Segment management

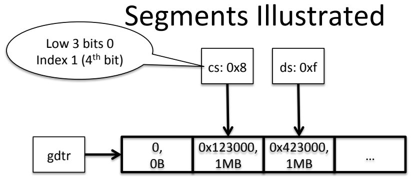
- For safety (without paging), only the OS should define segments. Why?
- Two segment tables the OS creates in memory:
  - Global – any process can use these segments
  - Local – segment definitions for a specific process
- How does the hardware know where they are?
  - Dedicated registers: gdtr and ldtr
  - Privileged instructions: lgdt, lldt

Segment registers

Table Index (13 bits)	Global or Local Table? (1 bit)	Ring (2 bits)

- Set by the OS on fork, context switch, etc.

Segments Illustrated



```
call cs:0xf150 ⟶ 0x123000 + 0xf150
                     = 0x123150
```

Sample Problem: (Old) JOS Bootloader

- Suppose my kernel is compiled to be in upper 256 MB of a 32-bit address space (i.e., 0xf0100000)
  - Common to put OS kernel at top of address space
- Bootloader starts in real mode (only 1MB of addressable physical memory)
- Bootloader loads kernel at 0x0010000
  - Can't address 0xf0100000

Booting problem

- Kernel needs to set up and manage its own page tables
  - Paging can translate 0xf0100000 to 0x00100000
- But what to do between the bootloader and kernel code that sets up paging?

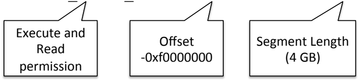
Segmentation to the Rescue!

- kern/entry.S:
  - What is this code doing?

```
mygdt:
 SEG_NULL                                  # null seg
 SEG(STA_X|STA_R, -KERNBASE, 0xffffffff) # code seg
 SEG(STA_W, -KERNBASE, 0xffffffff)       # data seg
```

JOS ex 1, cont.

```
SEG(STA_X|STA_R, -KERNBASE, 0xffffffff) # code seg
```

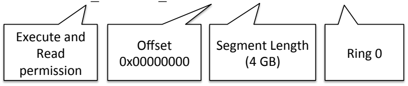


```
jmp 0xf01000db8   # virtual addr. (implicit cs seg)
      ⇩
jmp (0xf01000db8 + -0xf0000000)
      ⇩
jmp 0x001000db8   # linear addr.
```

Flat segmentation

- The above trick is used for booting. We eventually want to use paging.
- How can we make segmentation a no-op?
- From kern/pmap.c:

```
// 0x8 - kernel code segment
[GD_KT >> 3] = SEG(STA_X | STA_R, 0x0, 0xffffffff, 0),
```



Outline

- x86 processor modes
- x86 segmentation
- x86 page tables
- Advanced Features
- Interesting applications/problems

Paging Model

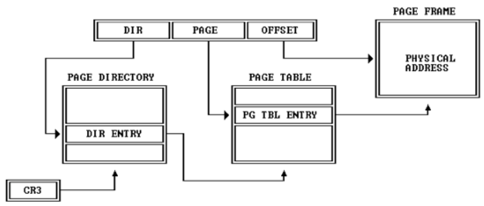
- 32 (or 64) bit address space.
- Arbitrary mapping of linear to physical pages
- Pages are most commonly 4 KB
  - Newer processors also support page sizes of 2 MB and 1 GB

How it works

- OS creates a page table
  - Any old page with entries formatted properly
  - Hardware interprets entries
- cr3 register points to the current page table
  - Only ring0 can change cr3

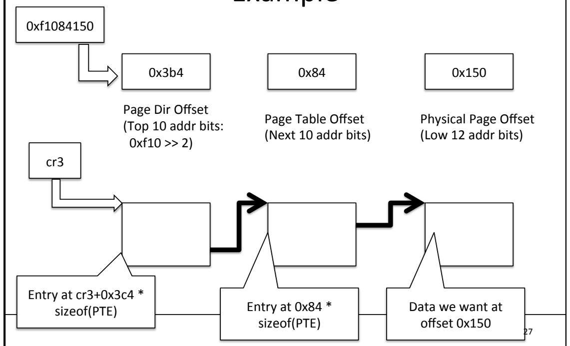
Translation Overview

Figure 5-9. Page Translation



From Intel 80386 Reference Programmer's Manual

Example



Page Dir Offset
(Top 10 addr bits:
0xf10 >> 2)

Page Table Offset
(Next 10 addr bits)

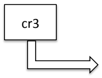
Physical Page Offset
(Low 12 addr bits)

Entry at cr3+0x3c4 * sizeof(PTE)

Entry at 0x84 * sizeof(PTE)

Data we want at offset 0x150

Page Table Entries



Physical Address Upper (20 bits)	Flags (12 bits)
0x00384	**PTE_W|PTE_P|PTE_U**
0	**0**
0x28370	**PTE_W|PTE_P**
0	**0**
0	**0**
0	**0**
0	**0**
0	**0**

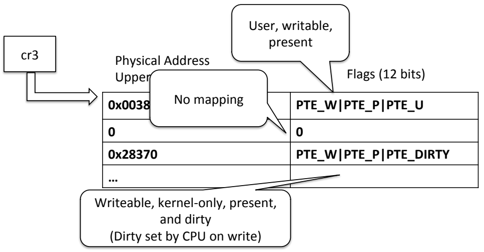
Page Table Entries

- Top 20 bits are the physical address of the mapped page
  - Why 20 bits?
  - 4k page size == 12 bits of offset
- Lower 12 bits for flags

Page flags

- 3 for OS to use however it likes
- 4 reserved by Intel, just in case
- 3 for OS to CPU metadata
  - User/vs kernel page,
  - Write permission,
  - Present bit (so we can swap out pages)
- 2 for CPU to OS metadata
  - Dirty (page was written), Accessed (page was read)

Page Table Entries



Back of the envelope

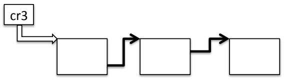
- If a page is 4K and an entry is 4 bytes, how many entries per page?
  - 1k
- How large of an address space can 1 page represent?
  - 1k entries * 1page/entry * 4K/page = 4MB
- How large can we get with a second level of translation?
  - 1k tables/dir * 1k entries/table * 4k/page = 4 GB
  - Nice that it works out that way!

Challenge questions

- What is the space overhead of paging?
  - I.e., how much memory goes to page tables for a 4 GB address space?
- What is the optimal number of levels for a 64 bit page table?
- When would you use a 2 MB or 1 GB page size?

TLB Entries

- The CPU caches address translations in the TLB
  - Translation Lookaside Buffer



Virt	Phys
0xf0231000	0x1000
0x00b31000	0x1f000
0xb0002000	0xc1000
–	–

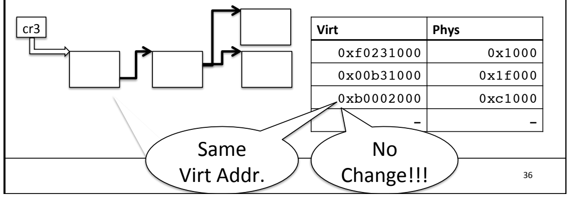
Page Traversal is **Slow** Table Lookup is **Fast**

TLB Entries

- The CPU caches address translations in the TLB
- Translation Lookaside BufferThe TLB is not coherent with memory, meaning:
  - **If you change a PTE, you need to manually invalidate cached values**
  - See the tlb_invalidate() function in JOS

TLB Entries

- The TLB is not coherent with memory, meaning:
  - **If you change a PTE, you need to manually invalidate cached values**
  - See the tlb_invalidate() function in JOS



Virt	Phys
0xf0231000	0x1000
0x00b31000	0x1f000
0xb0002000	0xc1000
–	–

Outline

- x86 processor modes
- x86 segmentation
- x86 page tables
- Advanced Features
- Interesting applications/problems

Physical Address Extension (PAE)

- Period with 32-bit machines + >4GB RAM (2000's)
- Essentially, an early deployment of a 64-bit page table format
- Any given process can only address 4GB
  - Including OS!
- Page tables themselves can address >4GB of physical pages

No execute (NX) bit

- Many security holes arise from bad input
  - Tricks program to jump to unintended address
  - That happens to be on heap or stack
  - And contains bits that form malware
- Idea: execute protection can catch these
  - Feels a bit like code segment, no?
- Bit 63 in 64-bit page tables (or 32 bit + PAE)

Nested page tables

- Paging tough for early Virtual Machine implementations
  - Can't trust a guest OS to correctly modify pages
- So, add another layer of paging between host-physical and guest-physical

And now the fun stuff…

Thread-Local Storage (TLS)

```
// Global
__thread int tid;
…
printf ("my thread id is %d\n", tid);
```

Identical code gets different value in each thread

Thread-local storage (TLS)

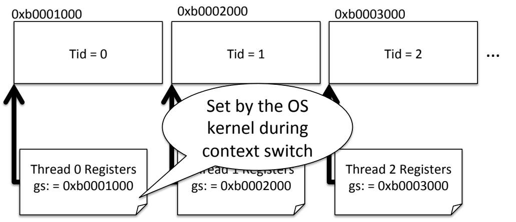
- Convenient abstraction for per-thread variables
- Code just refers to a variable name, accesses private instance
- Example: Windows stores the thread ID (and other info) in a thread environment block (TEB)
  - Same code in any thread to access
  - No notion of a thread offset or id
- How to do this?

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)

TLS Illustration



```
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

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

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

Conclusion

- Lab 2 will be fun

Housekeeping

- Reminder: sign up for course mailing list
  - Read the whole thing before posting
  - If you have an issue, please post if resolved (and how!)
- Checkpoint your VM before changing things
  - Instructions to follow soon
  - You break it, you buy it
- I'll update enrollment tomorrow