x86 Memory Protection and Translation
Don Porter

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

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
  - Permissions

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

```c
// global int x = 1   ds:x = 1; // data
int y; // stack     ss:y; // stack
if (x) {
    if (ds:x)
        y = 1;
    printf ("Boo");
    cs:printf (ds:"Boo");
} else
    y = 0;
```

Segments would be used in assembly, not C

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, llrd

Segment registers

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

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 0x00100000
  - 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:
    \[\text{SEQ\_NULL} \quad \text{# null seg}\]
    \[\text{SEQ(STA\_X}\mid\text{STA\_R, -KERNBASE, 0xffffffff)} \quad \text{# code seg}\]
    \[\text{SEQ(STA\_W, -KERNBASE, 0xffffffff)} \quad \text{# data seg}\]

JOS ex 1, cont.

\[\text{SEQ(STA\_X}\mid\text{STA\_R, -KERNBASE, 0xffffffff)} \quad \text{# code seg}\]

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:
    \[// 0x8 - kernel code segment\]
    \[[GD\_KT >> 3] = \text{SEQ(STA\_X}\mid\text{STA\_R, 0x0, 0xffffffff, 0)},\]

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

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

- User, writable, present
- Flags (12 bits)
- No mapping

Physical Address

Upper

0x00384

PTE_W|PTE_P|PTE_U

0

0

0x28370

PTE_W|PTE_P|PTE_DIRTY

Writeable, kernel-only, present, and dirty (Dirty set by CPU on write)

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

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
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)

```c
// 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

```
Tid = 0

Thread 0 Registers
gs = 0xb0001000

Tid = 1

Thread 1 Registers
gs = 0xb0002000

Tid = 2

Thread 2 Registers
gs = 0xb0003000

...  
```

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

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