Virtual Memory in x86

Nima Honarmand
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
  - No paging
  - No user mode; processor has only one protection level

- **Protected mode** – Standard 32-bit x86 mode
  - Combination of segmentation and paging
  - Privilege levels (separate user and kernel)
  - 32-bit virtual address
  - 32-bit physical address
    - 36-bit if **Physical Address Extension (PAE)** feature enabled
x86 Processor Modes

• **Long mode** – 64-bit mode (aka amd64, x86_64, etc.)
  • Very similar to 32-bit mode (protected mode), but bigger address space
    • 48-bit virtual address space
    • 52-bit physical address space
  • Restricted segmentation use

• Even more obscure modes we won’t discuss today

xv6 uses **protected mode w/o PAE** (i.e., 32-bit virtual and physical addresses)
Virt. & Phys. Addr. Spaces in x86

- Both RAM hand hardware devices (disk, NIC, etc.) connected to system bus
  - Mapped to different parts of the physical address space by the BIOS

- You can talk to a device by performing read/write operations on its physical addresses
  - Devices are free to interpret reads/writes in any way they want (driver knows)
Virt-to-Phys Translation in x86

- Segmentation cannot be disabled!
  - But can be made a no-op (a.k.a. flat mode)
Virt-to-Phys Translation in x86

- Every memory access has to go through this translation
  - Instruction fetches as well as data loads/stores

- Translation happens even in kernel mode
  - i.e., there is no variation of `mov` instruction, e.g., that would use physical addresses directly

- Even to talk to a device, its physical addresses have to be mapped somewhere in the page table, and kernel code should use the *corresponding virtual addresses*
x86 Segmentation

• A segment has:
  • Base address (linear address)
  • Segment 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

• Can 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)

• 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

• Two segment tables the OS creates in memory:
  • **GDT: Global Descriptor Table** – any process can use these segments
  • **LDT: Local Descriptor Table** – segment definitions for a specific process

• Each entry is called a **Segment Descriptor**
  • See the exact descriptor format in Intel or AMD manuals
  • What we care about for now is that it specifies segment base and length

• How does the hardware know where they are?
  • Dedicated registers: `gdtr` and `ldtr`
  • Privileged instructions to load the registers: `lgdt`, `lldt`
Segment (Selector) Registers

- cs, ds, ss, es, fs, gs

<table>
<thead>
<tr>
<th>Table Index</th>
<th>LDT or GDT?</th>
<th>RPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>(13 bits)</td>
<td>(1 bit)</td>
<td>(2 bits)</td>
</tr>
</tbody>
</table>

- “Table Index” is an index into either LDT or GDT

- RPL (Requestor Privilege Level): represents the privilege level (CPL) the processor is operating under at the time the selector is created

- To learn more about (complicated) details of privilege-level management in x86, read about DPL, CPL and RPL in either Intel or AMD architecture manuals
Segment (Selector) Registers

• Segment selectors are set by the OS on fork, context switch, interrupt, etc.

• On an interrupt, the interrupt handler should set all the segments selectors to kernel segments
  • But the CS needs to be set before the first kernel instruction is executed
  • Where to get it from?
  • Answer: IDT entry for the interrupt
Segment Management: Overall Picture

Source: AMD64 Architecture Programmer’s Manual (Volume 2)
Flat Segmentation

• Segments are relics of the ice age
  • We prefer to use paging for all address translations

• How can we make segmentation a no-op?
  • By setting the base address to 0, and length to max address space size (4GB in 32-bit x86)

• From vm.c:

```c
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
```
Task State Segment (TSS)

• On a user-to-kernel transfer (trap, exception, interrupt), the x86 processor dumps some data on the stack
  • \texttt{ss:esp, eflags, cs:eip}, and possibly an error code
  • Last few fields of \texttt{struct trapframe} in xv6

• But which stack? Should we keep using the user-mode stack?
  • Why not?
    • Because the user stack might not exist or might be full; remember user stack is completely under the user program’s control

• So, we need a different stack for the kernel mode

• But the processor needs to know the address of that stack before it can dump the data
  • TSS segment tells the processor where to find the kernel stack
Task State Segment (TSS)

• Another segment, just like code and data segment
  • A descriptor created in the GDT (cannot be in LDT)
  • Selected by special task register (\texttt{tr}) and loaded with \texttt{ltr}
  • Unlike others, the segment content has a hardware-specified layout

• Lots of fields for rarely-used features

• The fields we care about today:
  • Location of kernel stack (\texttt{ss} and \texttt{esp})
Page Tables
in 32-bit x86
32-bit Translation Overview

Source: AMD64 Architecture Programmer’s Manual (Volume 2)
32-bit PTE and PDEs

PDE in Protected-mode w/o PAE

PTE in Protected-mode w/o PAE

- P: present bit
- R/W: write permission?
- U/S: user-mode access?
- PWT, PCD, PAT: cache-related flags (ignore for now)
- A: Accessed, D: Dirty
- G: Global page? (for TLB management)
- AVL: available to OS to use in any way it wants
32-bit PTE and PDE flags

- 3 for OS to use however it likes (AVL)

- 7 for OS to CPU metadata
  - User vs. kernel page (U/S)
  - Write permission (R/W)
  - Present bit (P): page is present in memory
  - PWT, PCD, PAT, G

- 2 for CPU to OS metadata
  - Dirty (page was written), Accessed (page was read)

- In page directory entries, bit 7 indicates if it is a 4MB page
Address Space Organization

• Recall: In x86, all addresses used in instructions are virtual addresses and need to be translated
  • Including the instruction addresses
  • In all rings (ring 3 = user, ring 0 = kernel)
  • Including the very first instruction executed when transferring to kernel

→ To make OS designer’s life easier, most OSes map the kernel into the same (virtual address) in every process address space
Address Space Organization

- Kernel is mapped to the upper part of the virtual address space of every process
  - In xv6: at 0x80000000 (2GB)
  - In Linux/i386: at 0xC0000000 (3GB)
  - In all page tables, the upper mappings are the same
    = Kernel’s mappings

- Only the lower mappings (user part) differ across processes
Address Space Organization

• Why the 1-1 mapping region in the kernel space?

• Sometimes the kernel needs to access a location whose physical address it knows
  • For example, when it allocates a physical page, it fills it with 0
  • Say physical address is 0x00F00000

• But kernel is just instructions, and in x86, all instructions can only use virtual addresses
  • So kernel needs to have a virtual address mapped in the page table which will translate to 0x00F00000
  • How does the kernel find that virtual address?
  • By using the 1-1 mapping: just add the physical address to the beginning address of the 1-1 mapping region
xv6 code review

• Bootloader page table and segments
• Virtual address space layout
• Kernel page table and segments
• Why is kernel compiled to be execute from virtual address 0x80100000?
• TSS and kernel-mode stack
And now, some cool stuff...
Thread-Local Storage (TLS)

__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” segment register
  • Usually $gs$ or $fs$ to point to that range of virtual address
  • Each thread will use a different segment
  • When switching between threads should update $gs$ or $fs$

• Any thread accesses first byte of TLS like this:
  `mov eax, gs:(0x0)`
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