CS350: Operating Systems
Lecture 7: Virtual memory HW

Ali Mashtizadeh

University of Waterloo
Want processes to co-exist

<table>
<thead>
<tr>
<th></th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>0x9000</td>
</tr>
<tr>
<td>gcc</td>
<td>0x7000</td>
</tr>
<tr>
<td>bochs/pintos</td>
<td>0x4000</td>
</tr>
<tr>
<td>emacs</td>
<td>0x3000</td>
</tr>
<tr>
<td></td>
<td>0x0000</td>
</tr>
</tbody>
</table>

- Consider multiprogramming on physical memory
  - What happens if emacs needs to expand?
  - If emacs needs more memory than is on the machine?
  - If emacs has an error and writes to address 0x7100?
  - When does gcc have to know it will run at 0x4000?
  - What if emacs isn’t using its memory?
Issues in sharing physical memory

• **Protection**
  ▶ A bug in one process can corrupt memory in another
  ▶ Must somehow prevent process A from trashing B’s memory
  ▶ Also prevent A from even observing B’s memory (ssh-agent)

• **Transparency**
  ▶ A process shouldn’t require particular physical memory bits
  ▶ Yes processes often require large amounts of contiguous memory (for stack, large data structures, etc.)

• **Resource exhaustion**
  ▶ Programmers typically assume machine has “enough” memory
  ▶ Sum of sizes of all processes often greater than physical memory
Virtual memory goals

- Give each program its own “virtual” address space
  - At run time, Memory-Management Unit relocates each load, store to actual memory... App doesn’t see physical memory
- Also enforce protection
  - Prevent one app from messing with another’s memory
- And allow programs to see more memory than exists
  - Somehow relocate some memory accesses to disk
Virtual memory advantages

- Can re-locate program while running
  - Run partially in memory, partially on disk
- Most of a process’s memory may be idle (80/20 rule).
  
  - Write idle parts to disk until needed
  - Let other processes use memory of idle part
  - Like CPU virtualization: when process not using CPU, switch (Not using a memory region? switch it to another process)

- Challenge: VM = extra layer, could be slow
Idea 1: load-time linking

- **Linker** patches addresses of symbols like `printf`
- Idea: link when process executed, not at compile time
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- Problems?
Idea 1: load-time linking

- **Linker** patches addresses of symbols like `printf`.
- **Idea**: link when process executed, not at compile time
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- **Problems**?
  - How to enforce protection
  - How to move once already in memory (Consider: data pointers)
  - What if no contiguous free region fits program?
Idea 2: base + bound register

- Two special privileged registers: base and bound
- On each load/store:
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address $< \text{bound}$, else trap to kernel
- How to move process in memory?
- What happens on context switch?
Idea 2: base + bound register

• Two special privileged registers: base and bound

• On each load/store:
  ▶ Physical address = virtual address + base
  ▶ Check $0 \leq \text{virtual address} < \text{bound}$, else trap to kernel

• How to move process in memory?
  ▶ Change base register

• What happens on context switch?
Idea 2: base + bound register

- Two special privileged registers: base and bound
- On each load/store:
  - Physical address = virtual address + base
  - Check $0 \leq \text{virtual address} < \text{bound}$, else trap to kernel
- How to move process in memory?
  - Change base register
- What happens on context switch?
  - OS must re-load base and bound register
• Programs load/store to **virtual** (or **logical**) addresses
• Actual memory uses **physical** (or **real**) addresses
• VM Hardware is Memory Management Unit (**MMU**)
Address space

Virtual Address View

Physical Address View

MMU

OS
Base+bound trade-offs

• Advantages
  ▶ Cheap in terms of hardware: only two registers
  ▶ Cheap in terms of cycles: do add and compare in parallel
  ▶ Examples: Cray-1 used this scheme

• Disadvantages
  ▶ Growing a process is expensive or impossible
  ▶ No way to share code or data (E.g., two copies of bochs)

• One solution: Multiple segments
  ▶ E.g., separate code, stack, data segments
  ▶ Possibly multiple data segments
1 Segmentation

2 Paging

3 Intel x86: Hardware MMU

4 MIPS: Software Managed MMU
- Let processes have many base/bound regs
  - Address space built from many segments
  - Can share/protect memory at segment granularity
- Must specify segment as part of virtual address
Each process has a segment table

Each VA indicates a segment and offset:
- Top bits of addr select segment, low bits select offset (PDP-10)
- Or segment selected by instruction or operand (means you need wider “far” pointers to specify segment)
• 2-bit segment number (1st digit), 12 bit offset (last 3)
  ▶ Where is 0x0240? 0x1108? 0x265c? 0x3002? 0x1600?
Segmentation trade-offs

- **Advantages**
  - Multiple segments per process
  - Allows sharing! (how?)
  - Don’t need entire process in memory

- **Disadvantages**
  - Requires translation hardware, which could limit performance
  - Segments not completely transparent to program (e.g., default segment faster or uses shorter instruction)
  - $n$ byte segment needs $n$ contiguous bytes of physical memory
  - Makes fragmentation a real problem.
• **Fragmentation** → Inability to use free memory

• Over time:
  ▶ Variable-sized pieces = many small holes (external fragmentation)
  ▶ Fixed-sized pieces = no external holes, but force internal waste (internal fragmentation)
Outline

1. Segmentation

2. Paging

3. Intel x86: Hardware MMU

4. MIPS: Software Managed MMU
Paging

- Divide memory up into small pages
- Map virtual pages to physical pages
  - Each process has separate mapping
- Allow OS to gain control on certain operations
  - Read-only pages trap to OS on write
  - Invalid pages trap to OS on read or write
  - OS can change mapping and resume application
- Other features sometimes found:
  - Hardware can set “accessed” and “dirty” bits
  - Control page execute permission separately from read/write
  - Control caching or memory consistency of page
Paging trade-offs

- Eliminates external fragmentation
- Simplifies allocation, free, and backing storage (swap)
- Average internal fragmentation of .5 pages per “segment”
• Allocate any physical page to any process
• Can store idle virtual pages on disk
Paging data structures

- Pages are fixed size, e.g., 4K
  - Least significant 12 ($\log_2 4K$) bits of address are page offset
  - Most significant bits are page number

- Each process has a page table
  - Maps virtual page numbers (VPNs) to physical page numbers (PPNs)
  - Also includes bits for protection, validity, etc.

- On memory access: Translate VPN to PPN, then add offset
Example: Paging on PDP-11

- 64K virtual memory, 8K pages
  - Separate address space for instructions & data
  - I.e., can’t read your own instructions with a load

- Entire page table stored in registers
  - 8 Instruction page translation registers
  - 8 Data page translations

- Swap 16 machine registers on each context switch
MMU Types

- Memory Management Units (MMU) come in two flavors
  - Hardware Managed
    - Hardware reloads TLB with pages from a page tables
    - Typically hardware page tables are Radix Trees
    - Requires complex hardware
    - Examples: x86, ARM64, IBM POWER9+
  - Software Managed
    - Simpler hardware and asks software to reload pages
    - Requires fast exception handling and optimized software
    - Enables more flexibility in the TLB (e.g. variable page sizes)
    - Examples: MIPS, Sun SPARC, DEC Alpha, ARM and POWER
Outline

1. Segmentation
2. Paging
3. Intel x86: Hardware MMU
4. MIPS: Software Managed MMU
x86 Paging

- Paging enabled by bits in a control register (%cr0)
  - Only privileged OS code can manipulate control registers
- Normally 4KB pages
- %cr3: points to 4KB page directory
- Page directory: 1024 PDEs (page directory entries)
  - Each contains physical address of a page table
- Page table: 1024 PTEs (page table entries)
  - Each contains physical address of virtual 4K page
  - Page table covers 4 MB of Virtual mem
- See intel manual for detailed explanation
  - Volume 2 of *AMD64 Architecture docs*
  - Volume 3A of *Intel Pentium Manual*
x86 page translation

- **CR3 (PDBR)**
- **Page Directory**
- **Directory Entry**
- **Page Table**
- **Page-Table Entry**
- **Physical Address**
- **Linear Address**
- **Directory**
- **Table**
- **Offset**

**1024 PDE × 1024 PTE = 2^{20} Pages**

32 bits aligned onto a 4-KByte boundary
## x86 page directory entry

### Page-Directory Entry (4-KByte Page Table)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Page-Table Base Address</td>
</tr>
<tr>
<td>12-11</td>
<td>Avail</td>
</tr>
<tr>
<td>9</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>P</td>
</tr>
<tr>
<td>7</td>
<td>S</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>P</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>W</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>W</td>
</tr>
</tbody>
</table>

- **Available for system programmer’s use**
- **Global page (Ignored)**
- **Page size (0 indicates 4 KBytes)**
- **Reserved (set to 0)**
- **Accessed**
- **Cache disabled**
- **Write-through**
- **User/Supervisor**
- **Read/Write**
- **Present**
Page-Table Entry (4-KByte Page)

Available for system programmer’s use
Global Page
Page Table Attribute Index
Dirty
Accessed
Cache Disabled
Write-Through
User/Supervisor
Read/Write
Present
x86 hardware segmentation

- x86 architecture *also* supports segmentation
  - Segment register base + pointer val = *linear address*
  - Page translation happens on linear addresses
- Two levels of protection and translation check
  - Segmentation model has four privilege levels (CPL 0–3)
  - Paging only two, so 0–2 = kernel, 3 = user
- Why do you want *both* paging and segmentation?
x86 hardware segmentation

- x86 architecture *also* supports segmentation
  - Segment register base + pointer val = *linear address*
  - Page translation happens on linear addresses
- Two levels of protection and translation check
  - Segmentation model has four privilege levels (CPL 0–3)
  - Paging only two, so 0–2 = kernel, 3 = user
- Why do you want *both* paging and segmentation?
- Short answer: You don’t – just adds overhead
  - Most OSes use “flat mode” – set base = 0, bounds = 0xffffffff in all segment registers, then forget about it
  - x86-64 architecture removes much segmentation support
- Long answer: Has some fringe/incidental uses
  - VMware runs guest OS in CPL 1 to trap stack faults
  - OpenBSD used CS limit for W\(^\wedge\)X when no PTE NX bit
Making paging fast

- x86 PTs require 3 memory references per load/store
  - Look up page table address in page directory
  - Look up PPN in page table
  - Actually access physical page corresponding to virtual address

- For speed, CPU caches recently used translations
  - Called a translation lookaside buffer or TLB
  - Typical: 64-2K entries, 4-way to fully associative, 95% hit rate
  - Each TLB entry maps a VPN → PPN + protection information

- On each memory reference
  - Check TLB, if entry present get physical address fast
  - If not, walk page tables, insert in TLB for next time
    (Must evict some entry)
TLB details

- TLB operates at CPU pipeline speed → small, fast
- Complication: what to do when switch address space?
  - Flush TLB on context switch (e.g., old x86)
  - Tag each entry with associated process’s ID (e.g., MIPS)
- In general, OS must manually keep TLB valid
- E.g., x86 `invlpg` instruction
  - Invalidates a page translation in TLB
  - Must execute after changing a possibly used page table entry
  - Otherwise, hardware will miss page table change
- More Complex on a multiprocessor (TLB shootdown)
Where does the OS live?

• In its own address space?
  ▶ Can’t do this on most hardware (e.g., syscall instruction won’t switch address spaces)
  ▶ Also would make it harder to parse syscall arguments passed as pointers

• So in the same address space as process
  ▶ Use protection bits to prohibit user code from writing kernel

• Typically all kernel text, most data at same VA in every address space
  ▶ On x86, must manually set up page tables for this
  ▶ Usually just map kernel in contiguous virtual memory when boot loader puts kernel into contiguous physical memory
  ▶ Some hardware puts physical memory (kernel-only) somewhere in virtual address space
Outline

1. Segmentation

2. Paging

3. Intel x86: Hardware MMU

4. MIPS: Software Managed MMU
Very different MMU: MIPS

- Hardware has 64-entry TLB
  - References to addresses not in TLB trap to kernel
- Each TLB entry has the following fields:
  Virtual page, Pid, Page frame, NC, D, V, Global
- Kernel itself unpaged
  - All of physical memory contiguously mapped in high VM
  - Kernel uses these pseudo-physical addresses
- User TLB fault hander very efficient
  - Two hardware registers reserved for it
  - utlb miss handler can itself fault—allow paged page tables
- OS is free to choose page table format!
MIPS Memory Layout

FFFF  FFFF
kseg2: Paged Kernel

C000  0000
kseg1: Phys. Uncached

BFFF  FFFF
kseg0: Phys. Cached

A000  0000

9FFF  FFFF

8000  0000

7FFF  FFFF

Kernel Memory

useg: Paged User

User Memory

0000  0000
MIPS Translation Lookaside Buffer

- TLB Entries: 64 - 64-bit entries containing:
  - PID: Process ID (tagged TLB)
  - N: No Cache - disables caching for memory mapped I/O
  - D: Writeable - makes the page writeable
  - V: Valid
  - G: Global - ignores the PID during lookups

<table>
<thead>
<tr>
<th>Frame Number (VPN)</th>
<th>PID</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 62 61 60 59 58</td>
<td>N</td>
</tr>
<tr>
<td>57 56 55 54 53 52</td>
<td>D</td>
</tr>
<tr>
<td>51 50 49 48 47 46</td>
<td>V</td>
</tr>
<tr>
<td>45 44 43 42 41 40</td>
<td>G</td>
</tr>
<tr>
<td>39 38 37 36 35 34</td>
<td></td>
</tr>
<tr>
<td>33 32</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Page Number (PPN)</th>
<th>N</th>
<th>D</th>
<th>V</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22 21 20 19 18 17 16 15 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 12 11 10 9 8 7 6 5 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Page Sizes: Multiples of 4 from 4 kiB–16 MiB
  - 4 kiB, 16 kiB, 64 kiB, 256 kiB, 1 MiB, 4 MiB, 16 MiB
TLB PID and Global Bit

- Process ID (PID) allows multiple processes to coexist
  - We don’t need to flush the TLB on context switch
  - By setting the process ID
  - Only flush TLB entries when reusing a PID
  - Current PID is stored in c0_entryhi

- Global bit
  - Used for pages shared across all address spaces in kseg2 or useg
  - Ensures the TLB ignores the PID field
  - Typically in most hardware a TLB flush doesn’t flush global pages
TLB Instructions

- MIPS co-processor 0 (COP0) provides the TLB functionality

- Four instructions:
  - `tlbwr`: TLB write a random slot
  - `tlbwi`: TLB write a specific slot
  - `tlbr`: TLB read a specific slot
  - `tlbp`: Probe the slot containing an address

- For each of these instructions you must load the following registers
  - `c0_entryhi`: high bits of TLB entry
  - `c0_entrylo`: low bits of TLB entry
  - `c0_index`: TLB Index
Hardware Lookup Exceptions

• TLB Exceptions:
  ▶ UTLB Miss: Generated when the accessing useg without matching TLB entry
  ▶ TLB Miss: Generated when the accessing kseg2 without matching entry
  ▶ TLB Mod: Generated when writing to read-only page

• UTLB handler is separate from general exception handler
  ▶ UTLBs are very frequent and require a hand optimized path
  ▶ 64 entry TLB with 4 kiB pages covers 256 kiB of memory
  ▶ Modern machines have workloads with far more memory
  ▶ Require more entries (expensive hardware) or larger pages
Hardware Lookup Algorithm

- If most significant bit (MSB) is 1 and in user mode → address error exception.
- If no VPN match → TLB miss exception if MSB is 1, otherwise UTLB miss.
- If PID mismatches and global bit not set → generate a TLB miss or UTLB miss.
- If valid bit not set → TLB miss.
- Write to read-only page → TLB mod exception.
- If N bit is set directly access device memory (disable cache)
OS/161 Assembly Wrappers

- `tlb_random`: Write random TLB entry
- `tlb_write`: Write specific TLB entry
- `tlb_read`: Read specific TLB entry
- `tlb_probe`: Lookup TLB entry

Currently the OS implements segments using paging hardware

In a later assignment you will implement a Radix tree (like x86)
Example Memory Layout: user/testbin/sort

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFF FFFF</td>
<td>Stack</td>
</tr>
<tr>
<td>xxxx xxxx</td>
<td></td>
</tr>
<tr>
<td>1012 00B0</td>
<td>Data</td>
</tr>
<tr>
<td>1000 0000</td>
<td></td>
</tr>
<tr>
<td>0040 1A0C</td>
<td>Text + R/O Data</td>
</tr>
<tr>
<td>0040 0000</td>
<td></td>
</tr>
</tbody>
</table>
Paging in day-to-day use

- Paging Examples
  - Demand paging
  - Growing the stack
  - BSS page allocation
  - Shared text
  - Shared libraries
  - Shared memory
  - Copy-on-write (fork, mmap, etc.)

- Next time: detailed discussion on operating system side