Virtual Memory

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
Goals of OS:
- Protection and privacy: Processes cannot access each other’s data
- Abstraction: Hide away details of underlying hardware
  - e.g., processes open and access files instead of issuing raw commands to hard drive
- Resource management: Controls how processes share hardware resources (CPU, memory, disk, etc.)

Key enabling technologies:
- User mode + supervisor mode
- Interrupts to safely transition into supervisor mode
- Virtual memory to abstract the storage resources of the machine

Reminder: Operating Systems
Virtual Memory (VM) Systems

*Illusion of a large, private, uniform store*

- **Protection & Privacy**
  - Each process has a private address space

- **Demand Paging**
  - Provides the ability to run programs larger than main memory
  - Hides differences in machine configuration

The price of VM is address translation on each memory reference
Names for Memory Locations

- **Machine language address**
  - As specified in machine code

- **Virtual address**
  - ISA specifies translation of machine code address into virtual address of program variable (sometimes called *effective* address)

- **Physical address**
  - Operating system specifies mapping of virtual address into name for a physical memory location
Each program’s data is allocated in a contiguous segment of physical memory.

- Physical address = Virtual Address + Segment Base
- Bound register provides safety and isolation
- Base and Bound registers should not be accessed by user programs (only accessible in supervisor mode)
Separate Segments for Code and Data

Pros of this separation?
- Prevents buggy program from overwriting code
- Multiple processes can share code segment
As processes start and end, storage is “fragmented”. Therefore, at some point segments have to be moved around to compact the free space.
Paged Memory Systems

- Divide physical memory in fixed-size blocks called **pages**
  - Typical page size: 4KB

- Interpret each virtual address as a pair
  \(<\text{virtual page number}, \text{offset}>\)

- Use a **page table** to translate from virtual to physical page numbers
  - Page table contains the physical page number (i.e., starting physical address) for each virtual page number
Private Address Space per Process

- Each process has a page table
- Page table has an entry for each process page

Page tables make it possible to store the pages of a program non-contiguously
Paging vs. Segmentation

Pros of paging vs segmentation?

Paging avoids fragmentation issues

Paging allows programs that use more virtual memory than the machine’s physical memory (demand paging)

Cons of paging vs segmentation?

Page tables are MUCH larger than base & bound regs!

...where do we store the page tables?
Suppose Page Tables reside in memory

- **Translation:**
  - $\text{PPN} = \text{Mem}[\text{PT Base} + \text{VPN}]$
  - $\text{PA} = \text{PPN} + \text{offset}$

- All links represent physical addresses; no VA to PA translation

- On process switch
  - $\text{PT Base Reg} := \text{Kernel PT Base} + \text{new process ID}$

Accessing one data word or instruction requires two DRAM accesses!
Paging Implementation Issues

- How to reduce memory access overhead
  - A good VM design must be fast and space-efficient

- What if all the pages can’t fit in DRAM?
  - What if the process page table can’t fit in DRAM?
  - What if the kernel page table can’t fit in DRAM?

Beyond the scope of 6.004
Demand Paging
Using main memory as a cache of disk

- All the pages of the processes may not fit in main memory. Therefore, DRAM is backed up by swap space on disk.

- Page Table Entry (PTE) contains:
  - A resident bit to indicate if the page exists in main memory
  - PPN (physical page number) for a memory-resident page
  - DPN (disk page number) for a page on the disk
  - Protection and usage bits

- Even if all pages fit in memory, demand paging allows bringing only what is needed from disk
  - When a process starts, all code and data are on disk; bring pages in as they are accessed
Example: Virtual $\rightarrow$ Physical Translation

### Setup:
- 256 bytes/page ($2^8$)
- 16 virtual pages ($2^4$)
- 8 physical pages ($2^3$)
- 12-bit VA (4 vpn, 8 offset)
- 11-bit PA (3 ppn, 8 offset)

**lw 0x2C8(x0)**

**VA = 0x2C8, PA = 0x4C8**

**VPN = 0x2**

$\rightarrow$ **PPN = 0x4**
Caching vs. Demand Paging

**Caching**
- cache entry
- cache block (~32 bytes)
- cache miss rate (1% to 20%)
- cache hit (~1 cycle)
- cache miss (~100 cycles)
- a miss is handled in *hardware*

**Demand paging**
- page frame
- page (~4K bytes)
- page miss rate (<0.001%)
- page hit (~100 cycles)
- page miss (~5M cycles)
- a miss is handled mostly in *software*
Page Faults

An access to a page that does not have a valid translation causes a page fault exception. OS page fault handler is invoked, handles miss:

- Choose a page to replace, write it back if dirty. Mark page as no longer resident
- Read page from disk into available physical page
- Update page table to show new page is resident
- Return control to program, which re-executes memory access
Translation Lookaside Buffer (TLB)

Problem: Address translation is very expensive! Each reference requires accessing page table

Solution: *Cache translations in TLB*

TLB hit \(\Rightarrow\) *Single-cycle Translation*
TLB miss \(\Rightarrow\) *Page Table Walk to refill*

\[
\begin{array}{ccc|c|c}
V & R & W & D & \text{tag} & \text{PPN} \\
\hline
\end{array}
\]

(VPN = virtual page number)

(VPPN = physical page number)

virtual address

physical address

November 15, 2018
TLB Designs

- Typically 32-128 entries, 4 to 8-way set-associative
  - Modern processors use a hierarchy of TLBs
    (e.g., 128-entry L1 TLB + 2K-entry L2 TLB)

- Switching processes is expensive because TLB has to be flushed
  - Alternatively, include process ID in TLB entries to avoid flushing

- Handling a TLB miss: “Walk” the page table. If the page is in memory, load the VPN→PPN translation in the TLB. Otherwise, cause a page fault
  - Page faults are always handled in software
  - But page walks are usually handled in hardware using a memory management unit (MMU)
    - RISC-V, x86 access page table in hardware
Address Translation: Putting it all together

Virtual Address → TLB Lookup

- hit: Page Table Walk
  - the page is ∈ memory → Update TLB
  - the page is ∉ memory → Page Fault (OS loads page)
- miss: Protection Check
  - permitted: Physical Address (to mem)
  - denied: SEGFAULT

Protection Check

Hardware or software
Example: TLB and Page Table

Suppose
• Virtual memory of $2^{32}$ bytes
• Physical memory of $2^{24}$ bytes
• Page size is $2^{10}$ (1 K) bytes
• 4-entry fully associative TLB

1. How many pages can be stored in physical memory at once? $2^{14}$
2. How many entries are there in the page table? $2^{22}$
3. How many bits per entry in the page table? (Assume each entry has PPN, resident bit, dirty bit) 16
4. How many pages does page table take? $2^{23}$ bytes = $2^{13}$ pages
5. What fraction of virtual memory can be resident? $1/2^8$
6. What is the physical address for virtual address 0x1804? What components are involved in the translation? [VPN=6] 0x804
7. Same for 0x1080 [VPN=4] 0x1480
8. Same for 0x0FC

[VPN=0] page fault

Page Table

<table>
<thead>
<tr>
<th>VPN</th>
<th>R</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

TLB

<table>
<thead>
<tr>
<th>VPN</th>
<th>R</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

November 15, 2018
Problem: Size of Linear Page Table

With 32-bit addresses, 4 KB pages & 4-byte PTEs:
- \(2^{20}\) PTEs, i.e, 4 MB page table per process
- We often have hundreds to thousands of processes per machine... use GBs of memory just for page tables?

Use larger pages?
- Internal fragmentation (not all memory in a page is used)
- Larger page fault penalty (more time to read from disk)

What about a 64-bit virtual address space?
- Even 1MB pages would require \(2^{44}\) 8-byte PTEs (35 TB!)

Solution: Use a sparse page table format (size proportional to amount of mapped pages)
Hierarchical Page Table

Virtual Address

Virtual Address (Processor Register)

Root of the Current Page Table

Level 1 Page Table

Level 2 Page Tables

Data Pages

Page in primary memory

Page in secondary memory

PTE of a nonexistent page

31 22 21 12 11 0

p1 p2 offset

10-bit 10-bit

L1 index L2 index

L1 index

L2 index

offset

offset
Summary

- Virtual memory benefits:
  - Protection and privacy: Private address space per process
  - Demand paging: Can use main memory as a cache of disk

- Segmentation: Each process address space is a contiguous block (a segment) in physical memory
  - Simple: Base and bound registers
  - Suffers from fragmentation, no demand paging

- Paging: Each process address space is stored on multiple fixed-size pages. A page table maps virtual to physical pages
  - Avoids fragmentation
  - Enables demand paging: pages can be in main memory or disk
  - Requires a page table access on each memory reference

- Making paging efficient
  - TLB: Cache the page table
  - Hierarchical page tables: Keep page table size small
Thank you!

Next lecture: I/O and implementation issues of Virtual Memory and Exceptions