Virtual Memory
Reminder: Operating Systems

- **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
  - Exceptions to safely transition into supervisor mode
  - Virtual memory to abstract the storage resources of the machine

- **Last lecture**
- **Today**
Virtual Memory (VM) Systems

Illusion of a large, private, uniform store

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

- Demand Paging
  - Use main memory as a cache of disk
  - Enables running 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

- **Virtual address**
  - Address generated by the process
  - Specific to the process’s private address space

- **Physical address**
  - Address used to access physical (hardware) memory
  - Operating system specifies mapping of virtual addresses into physical addresses
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?
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 `<virtual page number, 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?

Cons of paging vs segmentation?

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

Translation:
- PPN = Mem[PT Base + VPN]
- PA = PPN + offset

All links represent physical addresses; no VA to PA translation

On process switch
- PT Base Reg := Kernel PT Base + new process ID

Accessing one data word or instruction requires two DRAM accesses!
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

16-entry Page Table

<table>
<thead>
<tr>
<th>VPN</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>


VA = 0x2C8, PA = _______

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)

\( \text{lw } 0x2C8(x0) \)

Dirty
Writable
Resident
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

Before Page Fault

<table>
<thead>
<tr>
<th>Page Table</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN 1</td>
<td>Disk</td>
</tr>
<tr>
<td>R=1</td>
<td></td>
</tr>
<tr>
<td>R=0</td>
<td></td>
</tr>
</tbody>
</table>

After Page Fault

<table>
<thead>
<tr>
<th>Page Table</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN 5</td>
<td>Disk</td>
</tr>
<tr>
<td>R=0</td>
<td></td>
</tr>
<tr>
<td>R=1</td>
<td></td>
</tr>
</tbody>
</table>

Nov 9, 2021

MIT 6.004 Fall 2021

L18-15
Translation Lookaside Buffer (TLB)

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

Solution: *Cache translations in TLB*

- TLB hit  ⇒  *Single-cycle translation*
- TLB miss  ⇒  *Access page table to refill TLB*

---

**Diagram:**

- **virtual address**
  - VPN
  - offset

- **physical address**
  - PPN
  - offset

- **VPN = virtual page number**
- **PPN = physical page number**

- **hit?**
- **fault?**

---

MIT 6.004 Fall 2021

L18-16
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. How many entries are there in the page table?
3. How many bits per entry in the page table? (Assume each entry has PPN, resident bit, dirty bit)
4. How many pages does page table take?
6. What is the physical address for virtual address 0x1804? What components are involved in the translation?
7. Same for 0x1080
8. Same for 0x0FC
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: Look up the page table (a.k.a. “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 → Protection Check
- permitted → Physical Address (to mem)
- denied
  → Page Table Lookup
    → miss → Page Fault (OS loads page)
    → hit → Update TLB

Page Table Lookup

the page is
- not in memory
- in memory

Protection Check

Resume process at faulting instruction

SEGFAULT
Using Caches with Virtual Memory

**Virtually-Addressed Cache**
- FAST: No virtual→physical translation on cache hits
- Problem: Must flush cache after context switch

**Physically-Addressed Cache**
- Avoids stale cache data after context switch
- SLOW: Virtual→physical translation before every cache access
Best of Both Worlds: Virtually-Indexed, Physically-Tagged Cache (VIPT)

Cache index comes entirely from address bits in page offset – don’t need to wait for TLB to start cache lookup!

OBSERVATION: If cache index bits are a subset of page offset bits, tag access in a physical cache can be done in parallel with TLB access. Tag from cache is compared with physical page address from TLB to determine hit/miss.

Problem: Limits # of bits of cache index → can only increase cache capacity by increasing associativity!
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

- TLBs make paging efficient by caching the page table
Thank you!

Next lecture: Pipelined Processors