18-447 Lecture 17: Address Translation

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today
  – see “Virtual Memory” in digestible pieces

• Notices
  – Lab 3, due this week

• Readings
  – P&H Ch 5
2 Parts to Modern VM

• In a multi-tasking system, **virtual** memory supports the **illusion** of a **large**, **private**, and **uniform** memory space to each process

• Ingredient A: naming and protection
  – each process sees a large, contiguous address space without holes (**for convenience**)
  – each process’s memory is private, i.e., protected from access by other processes (**for sharing**)

• Ingredient B: demand paging (**for hierarchy**)
  – capacity of secondary storage (swap space on disk)
  – speed of primary storage (DRAM)
The Common Denominator: Address Translation

- Large, private, and uniform abstraction achieved through address translation
  - user process operates on effective address (EA)
  - HW translates from EA to physical address (PA) on every memory reference
- Through address translation
  - control which physical locations (DRAM and/or swap disk) can be referred to by a process
  - allow dynamic allocation and relocation of physical backing store (where in DRAM and/or swap disk)
- Address translation HW and policies controlled by the OS and protected from user
Evolution of Memory Protection

• No need for protection or translation early on
  – single process, single user at a time
  – access all locations directly with PA

• Multitasking 101
  – each process limited to a non-overlapping, contiguous physical memory region
  (space doesn’t start from addr 0 . . .)
  – everything must fit in the region
  – how to keep one process from reading or trashing another process’s code and data?
Base and Bound

- A process’s private memory region defined by
  - base: starting address of region
  - bound: size of region

- User process issue “effective” address (EA) between 0 and the size of its allocated region
Base and Bound Registers

• Translation and protection check in hardware on every user memory reference
  – $PA = EA + \text{base}$
  – if $(EA < \text{bound})$ then okay else violation

• When switching user processes, OS sets base and bound registers

• User processes cannot be allowed to modify base and bound registers themselves

Requires at least 2 privilege levels with protected instruction and state for OS only
Segmented Address Space

• Limitations of single base-and-bound region
  – hard to find large contiguous space after a while—free space become fragmented
  – can two processes shared some memory regions but not others?
• A “base-and-bound” pair is a unit of protection
  ⇒ give user multiple memory “segments”
  – each segment is a contiguous memory region
  – each segment is defined by a base and bound pair
• Earliest use, separate code and data segments
  – 2 sets of base/bound for code vs data
  – forked processes can share code segments
    more elaborate later: code, data, stack, etc.
Segmented Address Translation

- **EA** partitioned into segment number (**SN**) and segment offset (**SO**)
  - max segment size limited by the range of **SO**
  - active segment size set by **bound**
- Per-process segment translation table
  - map **SN** to corresponding **base** and **bound**
  - separate mapping for each process
  - privileged structure if used to enforce protection
Access Protection

• Per-segment access rights can be specified as protection bits in segment table entries

• Generic options include
  – **Readable**?
  – **Writeable**?
  – **Executable**?

• For example
  – normal data segment ⇒ **RW**(!**E**)
  – static shared data segment ⇒ **R**(!**W**(!**E**)
  – code segment ⇒ **R**(!**W**) **E**  
  – illegal segment ⇒ (!**R**)(!**W**(!**E**)

  *Illegal* segment

  also misc. options, such as “cacheable”

  *What for?*

  Access violation exception brings OS into play
Aside: Another (ab)use of segments

• Extend old ISA to give new applications a large address space while stay compatible with old

• “User-managed” segmented addressing $SA = EA_{small}$
  – old application use identity mapping in table; old applications unaware of segments
  – new application reloa...
Paged Address Space

• Divide PA and EA space into equal, fixed size segments known as “page frames”

  
  historically 4KByte pages

• EA and PA are interpreted as page number (PN) and page offset (PO)
  – page table translates EPN to PPN; EPO=PPO
  – PA={PPN,PO}

many small pages good for managing allocation
Fragmentation

• External fragmentation by segments
  – plenty of unallocated DRAM but none in contiguous region of a sufficient size
  – paged memory eliminates external fragmentation

• Internal fragmentation of pages
  – entire page (4KByte) is allocated; unused bytes go to waste
  – smaller page size reduces internal fragmentation
  – modern ISA moving to larger page sizes (Mbytes) in addition to 4KBytes

Segments and pages not meant for the same role
Demand Paging

• Use main memory and “swap” disk as automatically managed memory hierarchy levels analogous to cache vs. main memory

• Early attempts
  – von Neumann already described manual memory hierarchies
  – Brookner’s interpretive coding, 1960: *program interpreter managed paging between a 40KByte main memory and a 640KByte drum*
  – Atlas, 1962: *hardware managed paging between 32-page core memory and 192-page drum (512 word/page)*
Demand Paging: just like caching

• **M** bytes of storage, keep most frequently used **C** bytes in DRAM where **C << M**

• Same basic issues as before
  1. where to “cache” a page in DRAM?
  2. how to find a page in DRAM?
  3. when to bring a page into DRAM?
  4. which page to evict from DRAM to disk to free-up DRAM for new pages?

• Key conceptual difference: swap vs. cache
  – DRAM doesn’t hold copies of what is on disk
  – a page in **M** either in DRAM or on disk
  – address not bound to 1 location for all time
Demand Paging: not at all like caching

- Drastically different size and time scale leads to drastically different implementation choices

<table>
<thead>
<tr>
<th></th>
<th>L1 Cache</th>
<th>L2 Cache</th>
<th>Demand Paging</th>
</tr>
</thead>
<tbody>
<tr>
<td>capacity</td>
<td>10~100KByte</td>
<td>MByte</td>
<td>GByte</td>
</tr>
<tr>
<td>block size</td>
<td>~16 Byte</td>
<td>~128 Byte</td>
<td>4K~4M Byte</td>
</tr>
<tr>
<td>hit time</td>
<td>few cyc</td>
<td>few 10s cyc</td>
<td>few 100s cyc</td>
</tr>
<tr>
<td>miss penalty</td>
<td>few 10s cyc</td>
<td>few 100s cyc</td>
<td>10 msec</td>
</tr>
<tr>
<td>miss rate</td>
<td>0.1~10%</td>
<td>(?)</td>
<td>0.00001~0.001%</td>
</tr>
<tr>
<td>hit handling</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>miss handling</td>
<td>HW</td>
<td>HW</td>
<td>SW</td>
</tr>
</tbody>
</table>

Hit time, miss penalty and miss rate are not independent variables!!
Don’t use “VM” to mean everything

- Effective Address (EA): emitted by user instructions in a per-process space (protection)
- Physical Address (PA): corresponds to actual storage locations on DRAM or on swap disk
- Virtual Address (VA): refers to locations in a system-wide, large, linear address space; not all locations in VA space have physical backing (demand paging)
EA, VA and PA (IBM Power view)

- 64-bit EA_0 divided into X fixed-size segments
- 64-bit EA_1 divided into X fixed-size segments
- 80~90-bit VA divided into Y segments (Y>>X);
  also divided as Z pages (Z>Y)

segmented EA: private, contiguous + sharing

- Demand paged VA: size of swap, speed of DRAM

- 40~50-bit PA divided into W pages (Z>>W)

- Swap disk divided into V pages (Z>>V, V>W)
EA, VA and PA (almost everyone else)

- **EA**
  - **EA₀** with unique ASID=0
  - **EAᵢ** with unique ASID=i

- **VA** divided into N “address spaces” indexed by ASID; also divided as Z pages (Z>>N)

- **PA** divided into W pages (Z>>W)

- Swap disk divided into V pages (Z>>V, V>>W)

Easy to blur **EA** and **VA**

how do processes share pages?
Just one more thing:
How large is the page table?

- A page table holds mapping from VPN to PPN
- Suppose 64-bit VA and 40-bit PA, how large is the page table? \(2^{52}\) entries x ~4 bytes \(\approx 16 \times 10^{15}\) Bytes

And that is for just one process!!?
How large should it be?

• Don’t need to track entire VA space
  – total allocated VA space is $2^{64}$ bytes x # processes, but most of which not backed by storage
  – can’t use more memory locations than physically exist (DRAM and swap disk)

• A clever page table should scale linearly with physical storage size and not VA space size

• Table cannot be too convoluted
  – a page table should be “walkable” by HW FSM
  – a page table is accessed not infrequently

Two dominant schemes in use today:

* hierarchical page table and hashed page table