18-447 Lecture 17: Address Translation

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today
  - see the step-wise development of virtual memory
- Notices
  - Lab 3, due this week
  - Midterm 4/10 in class; covers Lectures 11~19
  - practice midterm-2 from S’16
- Readings
  - P&H Ch 5
2 Parts to Modern VM

- In a multi-tasking system, *virtual* memory provides each process with the *illusion* of a *large*, *private*, and *uniform* memory
  
  *Not orthogonal but disregard caches for now . . . . .

- **Ingredient A:** naming and protection
  - each process conveniently sees a large, contiguous address space without holes
  - each process’s memory is private, i.e., protected from access by other processes

- **Ingredient B:** demand paging
  - capacity of secondary storage (swap space on disk)
  - speed of primary storage (DRAM)

Address Translation

- **Large, private, and uniform** abstraction achieved through address translation
  - user process operates on “effective” addresses
  - HW translates from effective to “physical” address on every memory reference (EA → PA)

- Through address translation
  - control which physical locations (DRAM and/or swap disk) can be referred to by a process
  - allow dynamic relocation of physical backing store (DRAM vs. swap disk)

- Address translation HW and policies controlled by the OS and protected from users, i.e., privileged
Evolution of Memory Protection

- Earliest machines did not have the concept of protection and address translation
  - no need—single process, single user automatically “private and uniform”
  - programs operated on physical addresses directly cannot support multitasking protection

- Multitasking 101
  - give each process a non-overlapping, contiguous physical memory region
  - everything belonging to a process must fit in that region
  - how do you keep one process from reading or trashing another process’s code and data?

Base and Bound

- A process’s private memory region can be defined by
  - base: starting address of the region
  - bound: ending address of the region

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

- Privileged control registers

<table>
<thead>
<tr>
<th>base</th>
<th>bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>active process’s region</td>
<td></td>
</tr>
<tr>
<td>another process’s region</td>
<td></td>
</tr>
</tbody>
</table>
Base and Bound Registers

- Translation and protection check in hardware on every user memory reference
  - PA = EA + base
  - if (PA < 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 2 privilege levels such that the OS can see and modify certain states that the user cannot

  privileged instructions and state

Segmented Address Space

- Limitations of the base and bound scheme
  - large contiguous space is hard to come by after the system runs for a while—free space may be fragmented
  - how do two processes shared some memory regions but not others?

- A “base&bound” pair is a unit of protection
  ⇒ give user multiple memory “segments”
  - each segment is a contiguous region
  - each segment is defined by a base and bound pair

- Earliest use, separate code and data segments
  - 2 sets of base-and-bound reg’s for inst vs data memory
  - allow processes to share read-only code segments

  became 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
  - segments can have different sizes; not all SOs are meaningful
- Per-process segment translation table
  - maps SN to corresponding base and bound
  - separate mapping for each process
  - privileged structure if for protection

<table>
<thead>
<tr>
<th>SN</th>
<th>SO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>segment table</td>
</tr>
<tr>
<td></td>
<td>base, bound, rights</td>
</tr>
<tr>
<td>+,&lt;</td>
<td>PA, okay?</td>
</tr>
</tbody>
</table>

Access Protections

- In addition to naming, finer-grain access protection can be associated with each segment as extra bits in the segment table
- Generic options include
  - readable?
  - writeable?
  - executable?
  - misc. options such as cacheable? which level?
- For example
  - normal data pages ⇒ RW(!E)
  - static shared data pages ⇒ R(!W)(!E)
  - code pages ⇒ R(!W)E what about self modifying code?
  - illegal pages ⇒ (!R)(!W)(!E) why would one want this?
Aside: Another Use of Segments

- How to extend an old ISA to give new applications a larger address space while stay compatible with old?
- “User-managed” segmented addressing
  - old applications use identify mapping in table, \( SA = EA_{small} \)
  - new applications reload table at run time with “active segments” to touch different \( EA_{large} \) regions
  - a “non-linear” address space: unequal access to active vs inactive \( EA_{large} \)

![Diagram](image1)

Paged Address Space

- Divide PA and EA space into fixed size segments known as “page frames”, historically 4KByte
- EAs and PAs are interpreted as page number (PN) and page offset (PO)
  - page table translates EPN to PPN
  - EPO is the same as PPO, just concatenate to PPN to get PA

![Diagram](image2)
Fragmentation

- External Fragmentation
  - a system may have plenty of unallocated DRAM, but they are useless in a segmented system if they do not form a contiguous region of a sufficient size
  - paged memory eliminates external fragmentation

- Internal Fragmentation
  - with paged memory, a process is allocated an entire page (4KByte) even if it only needs 4 bytes
  - a smaller page size reduces likelihood for internal fragmentation
  - modern ISA are moving to larger page sizes (Mbytes) in addition to 4KBytes

Demand Paging

- Use main memory and “swap” disk as automatically managed levels in the memory hierarchies analogous to cache vs. main memory

- Early attempts
  - von Neumann already described manual memory hierarchies
  - Brookner’s interpretive coding, 1960
    - program interpreter that managed paging between a 40KByte main memory and a 640KByte drum
  - Atlas, 1962
    - hardware demand paging between a 32-page (512 word/page) main memory and 192 pages on drums
    - user program believes it has 192 pages
Similar Fundamentals

- Potentially $M = 2^m$ bytes of backing store, how to keep the most frequently used ones in $C$ bytes of fast DRAM where $C << M$
- Basic issues
  1. where to “cache” a virtual page in DRAM?
  2. how to find a virtual page in DRAM?
  3. when to bring a page into DRAM?
  4. which virtual 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 either in DRAM or on disk
  - total effective capacity $C+M$

Demand Paging vs. Caching

- Drastically different size and time scale
  ⇒ drastically different implementation

<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>a few cyc</td>
<td>a few 10s cyc</td>
<td>a few 100s cyc</td>
</tr>
<tr>
<td>miss penalty</td>
<td>0.1~10%</td>
<td>a few 100s cyc</td>
<td>10 msec</td>
</tr>
<tr>
<td>miss rate</td>
<td></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 cannot be independent variables!!
Terminology and Usage

- **Physical Address**: directly refers to specific locations on DRAM or on swap disk

- **Effective Address**: emitted by user instructions for data and code access
  
  most often associated with “protection”

- **Virtual Address**: refers to locations in a large, linear memory abstraction; not all regions of the virtual address space have physical backing
  
  most often associated with “demand paging”

EA, VA and PA (IBM Power view)

- **EA**: divided into $X$ fixed-size segments

- **EA**: divided into $X$ fixed-size segments

- **EA**: divided into $X$ fixed-size segments

- **VA**: divided into $Y$ segments ($Y > X$); also divided as $Z$ pages ($Z > Y$)

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

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

segmented EA: private, contiguous + sharing

demand paged VA: size of swap, speed of DRAM
EA, VA and PA (almost everyone else)

\[ Z \gg V, V \gg W \]

\[ Z \gg Y \]

EA, VA, and PA (almost synonymous)

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? \( 2^{52} \text{ entries} \times \sim 4 \text{ bytes} \approx 16 \times 10^{15} \text{ Bytes} \)

And that is for just one process!!?
How large is the page table?

- Don’t need to keep track of the entire VA space
  - total allocated VA space in a system is $2^{64}$ bytes x # processes, but most of which is not alive
  - system can’t possibly use more memory locations than physically exist (DRAM and swap disk)
- A clever page table should scale “linearly” with size of physical storage and not size of VA space
- Also cannot be too convoluted
  - a page table must be “walkable” by HW
  - a page table is accessed not infrequently
- Two basic schemes in use today
  - hierarchical page tables
  - hashed page tables