Memory, part 2
CS 350: Computer Organization & Assembler Language Programming
Lecture 18, Wed 4/3

Overview
- Memory management extremely important, very complicated
- Programs/CPU create logical addresses, which get sent to Memory Management Unit (MMU)
- MMU translates logical addresses to physical addresses and signals actual memory read/write.

Review History
- One program running, takes entire CPU and memory
- Multiple programs running, pieces of memory allocated to each one.
  - Move ("relocate") program to improve machine's memory usage
    - Requires hardware to keep track of location of program's memory.
  - Swap out memory for processes that are "running" but actually waiting (for I/O, e.g.)
  - Swap in (back into memory) when process gets restarted.
- Faster to swap only parts of a program's memory
  - Segmentation: memory broken up into small number of large pieces
  - Allocation is tricky
    - External fragmentation (unallocated machine memory broken up into less-usable small pieces)
    - Need to find right-sized pieces on memory request
    - Want to combine free spaces when memory given up

Paging
- Paging: Memory is broken up into large number of same-sized pieces
  - Think of segmentation with many same-sized segments
- Program gets allocated pages of memory as needed
- Addresses of physical memory pieces ("frames") are discontinuously located in physical memory
- Logical memory addresses is one contiguous space (or small number of large contiguous spaces).
- Allocation of pages can be customized depending on situation
  - E.g., for read-only library code, multiple processes can use the same physical frame
- External fragmentation: Not an issue anymore
- Page table maps page numbers to physical frame numbers
  - Logical address (page #, offset) becomes physical address (frame #, same offset)
(thanks to Silberstein OS book for illustration)
Very Small Paging Example

- Physical address space: 32 bytes = 8 frames \( \times \) 4 bytes/frame
- Physical address width: 5 bits
- Width of frame size = 2 (byte addressable)
- Width of frame number = 3 bits (8 frames)
- Logical address space: 16 bytes (= 4 pages)
- Logical address width: 4 bits (2 bits for page number, 2 for offset).

Logical memory

<table>
<thead>
<tr>
<th>Logical addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>abcd</td>
</tr>
<tr>
<td>4–7</td>
<td>efgf</td>
</tr>
<tr>
<td>8–11</td>
<td>ijkl</td>
</tr>
<tr>
<td>12–15</td>
<td>mnop</td>
</tr>
</tbody>
</table>

Page Table

<table>
<thead>
<tr>
<th>Page #</th>
<th>Frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Physical memory

<table>
<thead>
<tr>
<th>Physical addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>??</td>
</tr>
<tr>
<td>4–7</td>
<td>ijkl</td>
</tr>
<tr>
<td>8–11</td>
<td>mnop</td>
</tr>
<tr>
<td>12–15</td>
<td>??</td>
</tr>
<tr>
<td>16–19</td>
<td>??</td>
</tr>
<tr>
<td>20–23</td>
<td>abcd</td>
</tr>
<tr>
<td>24–27</td>
<td>efgf</td>
</tr>
<tr>
<td>28–31</td>
<td>??</td>
</tr>
</tbody>
</table>

Page Table Size Can Be Large

- Internal fragmentation \( \leq \) (frame size – 1 byte) \( \times \) nbr of segments
- ↑ Page size: ↑ Internal fragmentation, but also ↓ Page table length.
- Example 1: 4 GB (= 2 physical memory, 4 KB frame size)
  - \( 1024^2 \) (= \( 2^{20} \)) frames of memory
  - Page table entry \( \geq \) 20 bits (frame number is 20 bits long)
  - Need 20 bits / (8 bits/B) \( \times \) \( 1024^2 \) B memory = 2.5 MB memory minimum to list all frames
- Example 2: Logical address space = 32 GB, Page size = 4 KB
  - 32 GB / 4 KB page table entries = 8 M entries
  - If 4 B/table entry, then need 4 \( \times \) 8 MB = 32 MB of table
- Note: Each running process has its own page table
  - When we switch from one process to another, MMU has to get new table

Hierarchical Page Tables

- Break up page table into page-sized pieces so that entire table doesn't have to be contiguous in memory.
- Two-level hierarchical page table (“Page the page table”)
  - Use a separate page table to track the pages of page table entries.
• Continue Example 2: Logical address space = 32 GB, Page size = 4 KB
  • 8 M page table entries @ 4 B / entry = 32 MB of page table = 8 * 1024 pages
  • Outer page table has 8 * 1024 entries @ 4 B / entry = 32 KB space = 8 pages
• Logical address space = 32 GB = $2^5 \times 2^{20}$ implies logical address has 25 bits
• Inner page table has 8 M entries = $2^3 \times 2^{20}$ entries, so 23 bits to hold inner page table index
• Outer page table has 8 * 1024 entries = $2^5 \times 2^{10}$ so 15 bits to hold outer page table index
• Offset within page: pages are 4 KB = $2^2 \times 2^{10}$ B, so need 12 bits to hold offset
• Total length of logical address = $15 + 23 + 12 = 40$ bits
Alternatives to multi-level page tables

- Use hashing on page number to get corresponding frame number
- Use frame numbers as indexes (inverted table), use hashing on page number to get frame table entry number

Virtual Memory

- Not all program code needs to be in physical memory — some code/data is rarely used, some hasn't been/won't be used for a while.
- Allowing program to be partially loaded removes physical memory constraint.
- Virtual memory completes separation of logical and physical address spaces.

Demand paging

- Lazy pager: Bring page into memory only when needed (a.k.a. lazy swapper).
- No unnecessary I/O; use less memory
- When creating process, page table marks all unshared pages as invalid
- Page table lookup sees invalid page, generates page fault interrupt
Page Fault

- On page fault interrupt, see if page actually exists but is on disk.
  - (True error if page doesn't exist.)
- Page exists but not in memory: Get empty frame, schedule disk read to copy page into frame
- When read finishes, mark page as valid, restart instruction that caused the page fault.

![Diagram of logical memory, page table, and physical memory]
• **Instruction Restart**
  - Hardware support needed for instruction restart.
  - Nontrivial problem, especially if instruction accesses multiple memory locations
    - (Note MIPS instructions only deal with one memory address at a time.)
  - Need to handle/want to avoid partial execution of instruction

• **Total access time depends on page fault rate (probability of page fault)**
  - Without page fault: one complete memory access
  - With page fault: also need time to swap page in/out, overhead handling page fault, overhead restarting instruction).

• **Thrashing**: When a system spends most of its time swapping virtual memory in/out instead of doing useful work.

**Caching**

• Storing of data so future requests can be done faster
  - Can have hardware caches, software caches
  - Hardware caches are relatively small due to cost

• **Cache hit**: Request finds item in cache.; **Cache miss**: not in cache; **Hit rate**: probability of hit.
  - For cache to be useful, hit rate has to be high enough to overcome overhead of looking in cache, finding a miss, then getting data item, storing it in the cache (in addition to yielding requested value).
  - Decision to use cache / how to use cache depends on string of future access requests, so is unknown in perfect detail. If enough is known about requests to develop good cache policy, then caching is successful

• **Temporal locality**: Future accesses like this one need to be close in time

• **Spatial locality**: Can we predict probability of future use by nearness to past accesses?
  - **Locality of Reference**: In practice, for memory access, programs tend to go through phases where they access roughly the same pieces of data.
  - Within a phase, there is locality of reference and cache helps
  - When changing from one phase to another, cache winds up being cleaned out and refilled with new data. Hopefully future accesses make up for overhead cost.

• **Latency and Throughput**
  - **Latency** of memory reference (time for one reference): Smaller if we can access data from cache.
  - **Throughput** of references (number of references per unit time): Larger if we have smaller latencies that let us satisfy more requests per unit time.

• **Replacement Policy**: If new data has to be added to a full cache, which piece of data gets **evicted** (thrown out)? Perfect policy requires future knowledge.
  - **Least-Recently-Used (LRU)** policy: Throw out oldest unused item. If cache accesses are localized enough, this does well.
  - If current phase of program uses amount of data > size of cache, LRU can be bad: We repeatedly toss out an item only to need it fairly soon after.
• **Writing Policy**: When data is written to cache, eventually it must also be written to backing store (the memory we're caching).
  
  • **Write-through**: write is done to the cache and to the backing store at the same time.
  
  • **Write-back/write-behind**: Write only to cache, eventually write entire block of cached data to backing store. More complex to implement, especially if multiple users are accessing the backing store. *(Cache coherency problem.)*