Lecture Topics

- Today: The Memory Hierarchy
  (P&H 5.1-5.11)

- Next: Exam #2

Announcements

- Milestone #8 (due 4/6)
- Milestone #9 (due 4/13)
Exam #2

- Wednesday, 3/29 (3:00-4:20 PM)
- Location:
  - 226 Erickson Hall (last names A-D)
  - 1257 Anthony Hall (last names E-Z)
- 80 minutes, 18% of course grade
- Study suggestions on course website

Virtual Memory

- Memory management technique where *virtual addresses* (addresses as viewed by a process) are mapped to *physical addresses* (actual addresses in RAM).
- Only a subset of the *virtual address space* for a process needs to be in RAM for it to execute (the virtual addresses which are currently being used by the process).
Virtual Memory: Costs and Benefits

- Every virtual address must be translated to a physical address (takes time, needs hardware support).
- CPU utilization can be increased by having more processes resident in RAM.
- The logical address space of a process can be larger than the physical address space of the system.

Virtual Memory and Paging

- Most virtual memory systems use paging: keep a subset of the pages in primary storage (RAM) and the remainder in secondary storage (disk).
- When necessary, move pages from disk to RAM (and from RAM to disk).
- Page table must keep track of current location of each page (RAM or disk).
Page Table

- Assume 32-bit virtual addresses, 30-bit physical addresses, and 12-bit offsets
  - Page numbers: 20 bits
  - Frame numbers: 18 bits
- Virtual address space: $2^{32}$ bytes
  (viewed as $2^{20}$ pages)
- Physical address space: $2^{30}$ bytes
  (viewed as $2^{18}$ page frames)

Example
Example (2)

- Page table for each process is large (4 MB in example)
- Keep part of page table on disk (page table uses virtual memory, just like user data structures)
- Use multi-level page tables?
- Use inverted page tables?
Efficiency Considerations

- Too slow: with page table, every memory access takes twice as long (access page table in RAM, then access the desired address in RAM)

- TLB (Translation Look-aside Buffer): special cache of page table entries

- First memory access requires look-up in page table, subsequent accesses use TLB
TLB Processing

- Hit in TLB: map page number to frame (within 1 clock cycle)
- Miss in TLB, page in RAM: load page table entry into the TLB, then restart instruction (10-100 clock cycles)
- Miss in TLB, page not in RAM: page fault (millions of clock cycles)
Page Fault Processing

- Move process from Running to Blocked
- Choose page to replace (if all frames full)
- If victim page has been modified ("dirty"), copy victim page from RAM to disk
- Copy page from disk to RAM
- Update page table entry
- Move process from Blocked to Ready

Efficiency Considerations

- A page fault takes millions of clock cycles to process
- OS handles page faults (not hardware)
- TLB usually fully associative
- Algorithm to select victim page must be effective
Recap: Virtual Memory

- Provides illusion of very large main memory
  - sum of memory for all processes can be larger than physical memory
  - address space of one process can be larger than physical memory
- Allows main memory to be efficiently utilized
- Simplifies memory management (relocation, protection and sharing)
- Exploits locality of reference to keep average memory access time low

A virtual memory system has the following characteristics:

- Virtual address: 32 bits
- Physical address: 40 bits
- Size of one page: 8 kilobytes
- TLB organization: fully associative, 256 entries

a) Number of bits in 1 virtual page offset?
b) Number of bits in 1 virtual page number?
c) Number of bits in 1 physical page offset?
d) Number of bits in 1 physical page number?
e) Assume that a TLB entry includes a valid bit, a referenced bit, and a modified bit. Number of bits in a TLB entry?
A virtual memory system has the following characteristics:

Virtual address: 32 bits
Physical address: 40 bits
Size of one page: 8 kilobytes
TLB organization: fully associative, 256 entries

a) Number of bits in 1 virtual page offset?
   13 bits: 8 kilobytes = 2^13 bytes

b) Number of bits in 1 virtual page number?
   19 bits: 19 + 13 = 32 (virtual address size)

c) Number of bits in 1 physical page offset?
   13 bits: virtual page and physical page are same size

d) Number of bits in 1 physical page number?
   27 bits: 27 + 13 = 40 bits (physical address size)
A virtual memory system has the following characteristics:

- Virtual address: 32 bits
- Physical address: 40 bits
- Size of one page: 8 kilobytes
- TLB organization: fully associative, 256 entries

e) Assume that a TLB entry includes a valid bit, a referenced bit, and a modified bit. Number of bits in a TLB entry?

- 3 bits (V, R, M)
- + 19 bits (virtual page number)
- + 27 bits (physical frame number)
- = 49 bits

Linux Virtual Memory (32 bit)

Memory layout:
- 3 GB in user space
- 1 GB in kernel space

Loaded from file:
- machine language
- static data

Created during execution:
- stack
- heap
Linux Address Translation

Linux uses a two-level page table for 32-bit addresses.

- Page size: 4096 bytes
- Upper 10 bits used to identify page directory
- Middle 10 bits used to identify page table
- Lower 12 bits are the page offset
- Page table entries are 4 bytes, so 1024 PTEs per page (same with PDEs)
Linux Address Translation

Two-level page table efficient:

- virtual memory space is sparsely populated (large sections unused)
- many PDEs will be unused
- top-level table stored in one page
- each second-level table stored in one page
- page faults take two memory accesses, but relatively rare

Linux Address Translation

4-level page table for 64-bit addresses:
Virtual Memory and Cache

- All modern architectures support at least one level of cache (most support 2 or 3)
- The virtual memory system must incorporate the cache(s)
- Most caches use physical addresses (virtual addresses are mapped to physical addresses before reaching the cache)
Table Cache Possible? If so, under what circumstances?

<table>
<thead>
<tr>
<th>TLB</th>
<th>Page Table</th>
<th>Cache</th>
<th>Possible? If so, under what circumstances?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit</td>
<td>Hit</td>
<td>Hit</td>
<td>Yes: PTE in TLB, data in cache (best case)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note: page table not checked if hit in TLB</td>
</tr>
<tr>
<td>Hit</td>
<td>Hit</td>
<td>Miss</td>
<td>Yes: PTE in TLB, but data not in cache</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(must be copied from RAM)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note: page table not checked if hit in TLB</td>
</tr>
<tr>
<td>Hit</td>
<td>Miss</td>
<td>Hit</td>
<td>No: PTE cannot be in TLB if page is not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>present in RAM</td>
</tr>
<tr>
<td>Hit</td>
<td>Miss</td>
<td>Miss</td>
<td>No: PTE cannot be in TLB if page is not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>present in RAM</td>
</tr>
<tr>
<td>TLB</td>
<td>Page Table</td>
<td>Cache</td>
<td>Possible? If so, under what circumstances?</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>-------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>Miss</td>
<td>Hit</td>
<td>Hit</td>
<td>Yes: PTE not in TLB, but page is present in RAM; after retry, data is found in cache</td>
</tr>
<tr>
<td>Miss</td>
<td>Hit</td>
<td>Miss</td>
<td>Yes: PTE not in TLB, but page is present in RAM; after retry, miss in cache (data must be copied from RAM to cache)</td>
</tr>
<tr>
<td>Miss</td>
<td>Miss</td>
<td>Hit</td>
<td>No: data cannot be in cache if page is not present in RAM</td>
</tr>
<tr>
<td>Miss</td>
<td>Miss</td>
<td>Miss</td>
<td>Yes: miss in TLB, followed by page fault; after retry, miss in cache (data must be copied from RAM to cache)</td>
</tr>
</tbody>
</table>

**TLB Miss**

- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware (difficult for complex page table data structures)
  - Could be handled in software as an exception

- If page is not in memory (page fault)
  - OS copies the page from disk to RAM
  - OS updates the page table
  - OS restarts the faulting instruction
OS: TLB Miss Handler

- TLB miss indicates one of:
  - Page present, but PTE not in TLB
  - Page not present

- Exception handler:
  - Copy PTE from page table to TLB
  - Restart instruction
  - If page not present, page fault will occur

Example: Intrinsity FastMATH

- Virtual addresses: 32 bits
- Physical addresses: 32 bits
- TLB: 16 entries (fully associative)
- Instruction cache: 256 entries (direct mapped)
- Data cache: 256 entries (direct mapped)
- Cache line: 64 bytes (16 four-byte words)
Example: read from data cache

- Assume hit in TLB (if miss, physical address mapped by accessing page table)
- Cache index (8 bits) used to index into "list" of tags in data cache
- Assume hit in data cache (if miss, 64 bytes copied from RAM into data cache, 4 bytes forwarded to CPU)
- Tags and data split for efficient access
Example: access to data cache

- Virtual address mapped to physical address
- Read (load): find data (4 bytes) in data cache (hit) or RAM (miss)
- Write (store): check write access bit (allowed to write), then find data (4 bytes) in data cache (hit) or RAM (miss)
- Uses "write through": every write to data cache causes line to be copied to RAM
Recap: Memory Hierarchy

Take advantage of the principle of locality of reference to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology.

Memory Hierarchy: terminology

- Pairs of levels in the memory hierarchy:
  - Cache and RAM
  - RAM and Disk
- Block: unit of data which is transferred between levels (also called a line)
Memory Hierarchy: terminology

- Hit rate: fraction of accesses found in upper level
- Miss rate: fraction of accesses not found in upper level (1 – hit rate)
- Hit time: time to access upper level, determine if data is present or not
- Miss penalty: time to copy block from lower level to upper level, satisfy request

Memory: Common Framework

All levels fit into the same framework:

- Question #1: how is a block located?
- Question #2: where can a block be placed when there is a miss?
- Question #3: which block should be replaced when there is a miss?
- Question #4: how are writes handled?
Question #1 (block identification)

How is a specific block identified?

- Indexing (direct mapped cache)
- Limited search (set associative cache)
- Full search (fully associative cache)
- Table lookup (page table)

Question #2 (block placement)

When a block is copied from the lower level to the upper level, where can it be placed?

- One place (direct mapped)
- A few places (set associative)
- Any place (fully associative)
Question #3 (block replacement)

When a block is copied from the lower level, which block should it replace in the upper level?

- Only meaningful if there is a choice (set associative or fully associative)
- Optimal: block which is not needed for the longest time (not possible to implement)
- LRU: approximate optimal by looking at the blocks which were used in the past

Question #3

When a block is copied from the lower level, which block should it replace in the upper level?

- NUR: approximate LRU by looking at the blocks which have not been used recently (reset "used" bit periodically)
- Random: miss rate is only about 10% higher than LRU in studies of TLBs
Question #3

When a block is copied from the lower level, which block should it replace in the upper level?

- Caches (including TLBs) use NUR or FIFO or Random – the victim must be selected quickly by the hardware
- Virtual memory uses NUR – miss penalty is huge, so even small improvements in the miss rate are important

Question #4 (write policy)

How are writes handled? That is, what happens when a block in the upper level has been modified?

- Write-through: whenever a write occurs, both the upper and lower levels are updated
- Write-back: whenever a write occurs, only the upper level is updated immediately; the lower level is updated when the block is evicted from the upper level
Write-through

When a write occurs, the block in the upper level and in the lower level are both updated

- Misses are simpler and cheaper because they never require a block to be copied back to the lower level (already done earlier)
- Easier to implement than write-back
- Only practical for caches

Write-back

When a write occurs, a block in the upper level is updated; when that block is evicted from the upper level, copied to the lower level

- Misses are expensive because they may require a block to be copied back to the lower level before that block can be replaced (use buffer to reduce cost)
- Multiple writes to a single block only require one copy to the lower level
Causes of Misses

Why do misses occur in the memory hierarchy?

- Compulsory misses
- Capacity misses
- Conflict misses

Compulsory Misses

- The first access to a block will always cause a miss (the block cannot be in the upper level)
- Also known as cold-start misses
- Increasing the block size will reduce compulsory misses (fewer blocks, so fewer accesses to a block for the first time)
Capacity Misses

- Upper level not large enough to hold all of the blocks that the process is currently using
- Occur when a block is replaced, then retrieved again later
- Increasing the size of the upper level (so that it can hold more blocks) will reduce capacity misses

Conflict Misses

- Does not apply to fully associative
- Misses which occur when multiple blocks compete for the same location (direct mapped or set associative)
- Also known as collision misses
- Increasing the associativity of the upper level will reduce conflict misses