Csci 211 Computer System Architecture
– Review on Virtual Memory

Xiuzhen Cheng
cheng@gwu.edu

The Five Classic Components of a Computer

Memory Hierarchy Requirements

Today’s Topics:
- Virtual Memory
- TLB

An Another View of the Memory Hierarchy

Memory Hierarchy Requirements

Virtual Memory

- Called “Virtual Memory”
- Also allows OS to share memory, protect programs from each other
- Today, more important for protection vs. just another level of memory hierarchy
- Historically, it predates caches
What is virtual memory?
- Virtual memory => treat the main memory as a cache for the disk
- Motivations:
  - Allow efficient and safe sharing of memory among multiple programs
  - Compiler assigns virtual address space to each program
  - Virtual memory maps virtual address spaces to physical spaces such that no two programs have overlapping physical address space
  - Remove the programming burdens of a small, limited amount of main memory
  - Allow the size of a user program exceed the size of primary memory
- Virtual memory automatically manages the two levels of memory hierarchy represented by the main memory and the secondary storage

Virtual to Physical Addr. Translation
- Each program operates in its own virtual address space; ~only program running
- Each is protected from the other
- OS can decide where each goes in memory
- Hardware (HW) provides virtual => physical mapping

Mapping Virtual Memory to Physical Memory
- Divide into equal sized chunks (about 4 KB - 8 KB)
- Any chunk of Virtual Memory assigned to any chuck of Physical Memory

Virtual Memory Mapping Function
- Cannot have simple function to predict arbitrary mapping
- Use table lookup of mappings
  - Use table lookup ("Page Table") for mappings: Page number is index
- Virtual Memory Mapping Function
  - Physical Offset = Virtual Offset
  - Physical Page Number = PageTable(Virtual Page Number)
  - (P.P.N. also called "Page Frame")
Address Mapping: Page Table

- Virtual Address: page no., offset
- Page Table Base Reg: index into page table
- Page Table: V A.R. P. P. A. Val Access Rights Physical Page Address
- Physical Memory Address

Page Table located in physical memory

Requirements revisited

- Remember the motivation for VM:
- Sharing memory with protection
  - Different physical pages can be allocated to different processes (sharing)
  - A process can only touch pages in its own page table (protection)
- Separate address spaces
  - Since programs work only with virtual addresses, different programs can have different data/code at the same address!

Page Table Entry (PTE) Format

- Contains either Physical Page Number or indication not in Main Memory
- OS maps to disk if Not Valid (V = 0)

<table>
<thead>
<tr>
<th>Page Table Entry (PTE) Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>Val Access Rights Page Number</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>P.T.E.</td>
</tr>
</tbody>
</table>

- If valid, also check if have permission to use page:
- Access Rights (A.R.) may be Read Only, Read/Write, Executable

Comparing the 2 levels of hierarchy

- Cache Version: Virtual Memory vers.
- Block or Line: Page
- Miss: Page Fault
- Block Size: 32-64B
- Replacement: Least Recently Used (LRU)
- Page Size: 4K-8KB
- Placement: Fully Associative
- Direct Mapped, N-way Set Associative
- Write Thru or Back Write Back
Notes on Page Table

- Solves Fragmentation problem: all chunks same size, and aligned, so all holes can be used
- OS must reserve "Swap Space" on disk for each process
- To grow a process, ask Operating System
  - If unused pages, OS uses them first
  - If not, OS swaps some old pages to disk
  - (Least Recently Used to pick pages to swap)
- Each process has own Page Table
- Will add details, but Page Table is essence of Virtual Memory

Page Table Summary

- Map virtual page number to physical page number
  - Full table indexed by virtual page number
- Minimize page fault
  - Fully associative placement of pages in main memory
- Reside in main memory
- Page fault can be handled in software, why?
- Page size is larger than cache block size, why?
- Each process has its own page block size
- Page table base register:
  - The page table starting address of the active process will be loaded to the page table register

Virtual Memory Problem

- Virtual memory seems to be really slow:
  - We have to access memory on every access – even cache hits!
  - Worse, if translation not completely in memory, may need to go to disk before hitting in cache!
- Solution: Caching! (surprise!) – cache the translation
  - Keep track of most common translations and place them in a "Translation Lookaside Buffer" (TLB)

Translation Look-Aside Buffers (TLBs)

- TLBs usually small, typically 128 - 256 entries
- Like any other cache, the TLB can be direct mapped, set associative, or fully associative

Typical TLB Format

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access Rights</th>
</tr>
</thead>
</table>

- TLB just a cache on the page table mappings
- TLB access time comparable to cache (much less than main memory access time)
- Dirty: since use write back, need to know whether or not to write page to disk when replaced
- Ref: Used to help calculate LRU on replacement
  - Cleared by OS periodically, then checked to see if page was referenced

What if not in TLB?

- Option 1: Hardware checks page table and loads new Page Table Entry into TLB
- Option 2: Hardware traps to OS, up to OS to decide what to do
  - MIPS follows Option 2: Hardware knows nothing about page table
Page Fault: What happens when you miss?

- Page fault means that page is not resident in memory.
- Hardware must detect situation.
- Hardware cannot remedy the situation.
- Therefore, hardware must trap to the operating system so that it can remedy the situation.
  - Pick a page to discard (possibly writing it to disk).
  - Start loading the page in from disk.
  - Schedule some other process to run.

Later (when page has come back from disk):
- Update the page table.
- Resume to program so HW will retry and succeed!

What if the data is on disk?
- We load the page off the disk into a free block of memory, using a DMA transfer.
- Meanwhile we switch to some other process waiting to be run.
- When the DMA is complete, we get an interrupt and update the process’s page table.
- So when we switch back to the task, the desired data will be in memory.

What if we don’t have enough memory?
- We chose some other page belonging to a program and transfer it onto the disk if it is dirty.
  - If clean (disk copy is up-to-date), just overwrite that data in memory.
  - We chose the page to evict based on replacement policy (e.g., LRU).
- And update that program’s page table to reflect the fact that its memory moved somewhere else.
- If continuously swap between disk and memory, called Thrashing.

And in conclusion…
- Manage memory to disk? Treat as cache.
  - Included protection as bonus, now critical.
  - Use Page Table of mappings for each user vs. tag/data in cache.
  - TLB is cache of Virtual➔Physical addr trans.
- Virtual Memory allows protected sharing of memory between processes.
- Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well.

Address Translation & 3 Concept tests

Virtual Address

Data Cache

Physical Address

4 Qs for any Memory Hierarchy

Q1: Where can a block be placed?
- One place (direct mapped)
- A few places (set associative)
- Any place (fully associative)

Q2: How is a block found?
- Indexing (as in a direct-mapped cache)
- Limited search (as in a set-associative cache)
- Full search (as in a fully associative cache)
- Separate lookup table (as in a page table)

Q3: Which block is replaced on a miss?
- Least recently used (LRU)
- Random

Q4: How are writes handled?
- Write through (Level never inconsistent w/lower)
- Write back (Could be “dirty”, must have dirty bit)
Q1: Where block placed in upper level

- Block 12 placed in 8 block cache:
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets

<table>
<thead>
<tr>
<th>Block no.</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully associative: block 12 can go anywhere</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Direct mapped: block 12 can go only into block 4 (12 mod 8)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Set associative: block 12 can go anywhere in set 0 (12 mod 4)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Q2: How is a block found in upper level?

- Direct indexing (using index and block offset), tag compares, or combination
- Increasing associativity shrinks index, expands tag

Q3: Which block replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

<table>
<thead>
<tr>
<th>Miss Rates</th>
<th>Associativity:</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU Ran</td>
<td>LRU Ran</td>
<td>LRU Ran</td>
<td>LRU Ran</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2% 5.7%</td>
<td>4.7% 5.3%</td>
<td>4.4% 5.9%</td>
<td></td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9% 2.0%</td>
<td>1.5% 1.7%</td>
<td>1.4% 1.5%</td>
<td></td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15% 1.17%</td>
<td>1.13% 1.13%</td>
<td>1.12% 1.12%</td>
<td></td>
</tr>
</tbody>
</table>

Q4: What to do on a write hit?

- Write-through
  - update the word in cache block and corresponding word in memory
- Write-back
  - update word in cache block
  - allow memory word to be “stale”
    => add ‘dirty’ bit to each line indicating memory be updated when block is replaced

 Performance trade-offs?
- WT: read misses cannot result in writes
- WB: no writes of repeated writes

Three Advantages of Virtual Memory

1) Translation:
   - Program can be given consistent view of memory, even though physical memory is scrambled
   - Makes multiple processes reasonable
   - Only the most important part of program (“Working Set”) must be in physical memory
   - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later

2) Protection:
   - Different processes protected from each other
   - Different pages can be given special behavior
     - (Read Only, Invisible to user programs, etc).
   - Kernel data protected from User programs
   - Very important for protection from malicious programs ⇒ Far more “viruses” under Microsoft Windows
   - Special Mode in processor (“Kernel more”) allows processor to change page table/TLB

3) Sharing:
   - Can map same physical page to multiple users (“Shared memory”)
Why Translation Lookaside Buffer (TLB)?

- Paging is most popular implementation of virtual memory
- Every paged virtual memory access must be checked against Entry of Page Table in memory to provide protection
- Cache of Page Table Entries (TLB) makes address translation possible without memory access in common case to make fast

Bonus slide: Virtual Memory Overview (1/4)

- User program view of memory:
  - Contiguous
  - Start from some set address
  - Infinitely large
  - Is the only running program
- Reality:
  - Non-contiguous
  - Start wherever available memory is
  - Finite size
  - Many programs running at a time

Bonus slide: Virtual Memory Overview (2/4)

- Virtual memory provides:
  - Illusion of contiguous memory
  - All programs starting at same set address
  - Illusion of ~ infinite memory
  - (2^32 or 2^64 bytes)
  - Protection

Bonus slide: Virtual Memory Overview (3/4)

- Implementation:
  - Divide memory into “chunks” (pages)
  - Operating system controls page table that maps virtual addresses into physical addresses
  - Think of memory as a cache for disk
  - TLB is a cache for the page table

Bonus slide: Virtual Memory Overview (4/4)

- Let’s say we’re fetching some data:
  - Check TLB (input: VPN, output: PPN)
    - Hit: fetch translation
    - Miss: check page table (in memory)
      - Page table hit: fetch translation
      - Page table miss: page fault, fetch page from disk to memory, return translation to TLB
  - Check cache (input: PPN, output: data)
    - Hit: return value
    - Miss: fetch value from memory

The MIPS R2000 TLB
Implementing Protection with VM

- Multiple processes share a single main memory!
  - How to prevent one process from reading/writing over another’s data?
    - Write access bit in page table
    - Non-overlapping page tables
    - OS controls the page table mappings
    - Page tables reside in OS’s address space
  - How to share information?
    - Controlled by OS
    - Multi virtual addresses map to the same page table

Handling Page Fault and TLB Miss

- TLB Miss
- Page Fault
  - By exception handling mechanism
  - Page fault happens during the clock cycle used to access memory
  - EPC is used to store the address of the instruction that causes the exception
    - How to find the virtual address of the memory unit that holds the data when data page fault happens?
    - Prevent the completion of the faulting instruction – no writing!!
  - Cause register provide page fault reasons
  - OS does the following
    - Find the location of the referenced page on disk
    - Choose a physical page to replace. What about dirty pages?
    - Read the referenced page from disk

And in Conclusion...

- Virtual memory to Physical Memory Translation too slow?
  - Add a cache of Virtual to Physical Address Translations, called a TLB
- Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well
- Virtual Memory allows protected sharing of memory between processes with less swapping to disk

Questions?