Virtual Memory

Lecture 13
CDA 3103
07-02-2014
Review

- Cache design choices:
  - Size of cache: speed v. capacity
  - Block size (i.e., cache aspect ratio)
  - Write Policy (Write through v. write back)
  - Associativity choice of N (direct-mapped v. set v. fully associative)
  - Block replacement policy
  - 2nd level cache?
  - 3rd level cache?

- Use performance model to pick between choices, depending on programs, technology, budget, ...
Another View of the Memory Hierarchy

Thus far

Next: Virtual Memory

Upper Level

Faster

L2 Cache

Memory

Disk

Tape

Upper Level

Faster

Larger

Lower Level

Regs

Instr. Operands

Blocks

Pages

Files
Memory Hierarchy Requirements

- If Principle of Locality allows caches to offer (close to) speed of cache memory with size of DRAM memory, then recursively why not use at next level to give speed of DRAM memory, size of Disk memory?

- While we’re at it, what other things do we need from our memory system?
Memory Hierarchy Requirements

- Allow multiple processes to simultaneously occupy memory and provide protection – don’t let one program read/write memory from another

- Address space – give each program the illusion that it has its own private memory
  - Suppose code starts at address 0x40000000. But different processes have different code, both residing at the same address. So each program has a different view of memory.
Virtual Memory

- Next level in the memory hierarchy:
  - Provides program with illusion of a very large main memory:
  - Working set of “pages” reside in main memory - others reside on disk.
- Also allows OS to share memory, protect programs from each other
- Today, more important for protection vs. just another level of memory hierarchy
- Each process thinks it has all the memory to itself
- (Historically, it predates caches)
Virtual to Physical Address 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 gives virtual → physical mapping
Analogy

- Book title like *virtual address*
- Library of Congress call number like *physical address*
- Card catalogue like *page table*, mapping from book title to call #
- On card for book, in local library vs. in another branch like *valid bit* indicating in main memory vs. on disk
- On card, available for 2-hour in library use (vs. 2-week checkout) like *access rights*
Simple Example: Base and Bound

Reg

$base$

$base+$

$bound$

User C

User B

User A

OS

$\infty$

Enough space for User D, but discontinuous ("fragmentation problem")

• Want:
  • discontinuous mapping
  • Process size $>>$ mem
  • Addition not enough!

$\square$ use Indirection!
Mapping Virtual Memory to Physical Memory

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

64 MB
Paging Organization (assume 1 KB pages)

Physical Address

- 0
- 1024
- ... 
- 7168

Physical Memory

- page 0
- page 1
- ... 
- page 7

Page is unit of mapping

Page also unit of transfer from disk to physical memory

Virtual Address

- 0
- 1024
- ... 
- 31744

Virtual Memory

- page 0
- page 1
- ... 
- page 31
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 located in physical memory

Page Table Base Reg

index into page table

Page Table

<table>
<thead>
<tr>
<th>V</th>
<th>A.R.</th>
<th>P. P. A.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val-id</td>
<td>Access Rights</td>
<td>Physical Page Address</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Physical Memory Address

Physical Page Address
Address Translation

- Fixed-size pages (e.g., 4K)

Diagram showing virtual addresses, physical addresses, and disk addresses with address translation and page number translation.
**Page Table**

- A page table is an operating system structure which contains the mapping of virtual addresses to physical locations.
  - There are several different ways, all up to the operating system, to keep this data around.
- Each process running in the operating system has its own page table.
  - “State” of process is PC, all registers, plus page table.
  - OS changes page tables by changing contents of Page Table Base Register.
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!
- What about the memory hierarchy?
Page Table Entry (PTE) Format

- Contains either Physical Page Number or indication not in Main Memory
- OS maps to disk if Not Valid ($V = 0$)
- If valid, also check if have permission to use page: **Access Rights (A.R.)** may be Read Only, Read/Write, Executable
Translation Using a Page Table

Virtual address

Page table register

Virtual page number | Page offset

Physical page number | Page offset

Physical address

If 0 then page is not present in memory

Valid

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 20
Mapping Pages to Storage

Virtual page number

Page table
Physical page or disk address

Valid

Physical memory

Disk storage

1
1
1
1
1
1
1
0
1
1
1
0
1
1
0
1
1
0
1
1
0
1
**Question:** How many bits wide are the following fields?

- 16 KiB pages
- 40-bit virtual addresses
- 64 GiB physical memory

<table>
<thead>
<tr>
<th></th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>A)</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>B)</td>
<td>24</td>
<td>20</td>
</tr>
<tr>
<td>C)</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>D)</td>
<td><strong>26</strong></td>
<td><strong>22</strong></td>
</tr>
</tbody>
</table>

**Answer:** D) 26, 22
## Comparing the 2 levels of hierarchy

<table>
<thead>
<tr>
<th>Cache version</th>
<th>Virtual Memory vers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block or Line</td>
<td>Page</td>
</tr>
<tr>
<td>Miss</td>
<td>Page Fault</td>
</tr>
<tr>
<td>Block Size: 32-64B</td>
<td>Page Size: 4K-8KB</td>
</tr>
<tr>
<td>Placement:</td>
<td>Fully Associative</td>
</tr>
<tr>
<td>Direct Mapped,</td>
<td></td>
</tr>
<tr>
<td>N-way Set Associative</td>
<td></td>
</tr>
<tr>
<td>Replacement:</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td>LRU or Random (LRU)</td>
<td></td>
</tr>
<tr>
<td>Write Thru or Back</td>
<td>Write Back</td>
</tr>
</tbody>
</table>
Notes on Page Table

- Solves Fragmentation problem: all chunks same size, 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
Virtual Memory Problem #1

- Map every address → 1 indirection via Page Table in memory per virtual address → 1 virtual memory accesses = 2 physical memory accesses → SLOW!

- Observation: since locality in pages of data, there must be locality in virtual address translations of those pages

- Since small is fast, why not use a small cache of virtual to physical address translations to make translation fast?

- For historical reasons, cache is called a Translation Lookaside Buffer, or TLB
Retrieving Data from Memory

1) Access page table for address translation

2) Access correct physical address

Requires two accesses of physical memory!
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

On TLB miss, get page table entry from main memory
Where Are TLBs Located?

• Which should we check first: Cache or TLB?
  – Can cache hold requested data if corresponding page is not in physical memory? **No**
  – With TLB first, does cache receive VA or **PA**?

Notice that it is now the TLB that does translation, not the Page Table!
Address Translation Using TLB

Virtual Address

TLB Tag | TLB Index | Page Offset

VPN

TLB

Tag | PPN
(used just like in a cache)

... PA split two different ways!

Data Cache

Tag | Block Data

... Physical Address

PPN | Page Offset

Tag | Index | Offset

Note: TIO for VA & PA unrelated
Valid and Access Rights: Same usage as previously discussed for page tables

Dirty: Basically always use write-back, so indicates whether or not to write page to disk when replaced

Ref: Used to implement LRU
  – Set when page is accessed, cleared periodically by OS
  – If Ref = 1, then page was referenced recently

TLB Index: VPN mod (# TLB sets)

TLB Tag: VPN – TLB Index
Fast Translation Using a TLB

Virtual page number

TLB

Physical page address

Physcial memory

Page table

Valid Dirty Ref or disk address

Disk storage

Chapter 5 — Large and Fast: Exploiting Memory Hierarchy — 32
**Question:** How many bits wide are the following?

- 16 KiB pages
- 40-bit virtual addresses
- 64 GiB physical memory
- 2-way set associative TLB with 512 entries

<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>Ref</th>
<th>Access Rights</th>
<th>TLB Tag</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>XX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TLB Tag</th>
<th>TLB Index</th>
<th>TLB Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>A) 12</td>
<td>14</td>
<td>38</td>
</tr>
<tr>
<td>B) 18</td>
<td>8</td>
<td>45</td>
</tr>
<tr>
<td>C) 14</td>
<td>12</td>
<td>40</td>
</tr>
<tr>
<td>D) 17</td>
<td>9</td>
<td>43</td>
</tr>
</tbody>
</table>
Peer Instruction

1) Locality is important yet different for cache and virtual memory (VM): temporal locality for caches but spatial locality for VM

2) VM helps both with security and cost
1) Locality is important yet different for cache and virtual memory (VM): temporal locality for caches but spatial locality for VM.

1. No. Both for VM and cache

2) VM helps both with security and cost

2. Yes. Protection and a bit smaller memory

Peer Instruction Answer

12
a) FF
b) FT
c) TF
d) TT
TLB and Cache Interaction

- If cache tag uses physical address
  - Need to translate before cache lookup
- Alternative: use virtual address tag
  - Complications due to aliasing
    - Different virtual addresses for shared physical address
Data Fetch Scenarios

• Are the following scenarios for a single data access possible?
  – TLB Miss, Page Fault
  – TLB Hit, Page Table Hit
  – TLB Miss, Cache Hit
  – Page Table Hit, Cache Miss
  – Page Fault, Cache Hit

Yes
No
<table>
<thead>
<tr>
<th>TLB</th>
<th>Page Table</th>
<th>Cache</th>
<th>Possible? If so, under what circumstance?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit</td>
<td>Hit</td>
<td>Miss</td>
<td>Possible, although the page table is never really checked if TLB hits.</td>
</tr>
<tr>
<td>Miss</td>
<td>Hit</td>
<td>Hit</td>
<td>TLB misses, but entry found in page table; after retry, data is found in cache.</td>
</tr>
<tr>
<td>Miss</td>
<td>Hit</td>
<td>Miss</td>
<td>TLB misses, but entry found in page table; after retry, data misses in cache.</td>
</tr>
<tr>
<td>Miss</td>
<td>Miss</td>
<td>Miss</td>
<td>TLB misses and is followed by a page fault; after retry, data must miss in cache.</td>
</tr>
<tr>
<td>Hit</td>
<td>Miss</td>
<td>Miss</td>
<td>Impossible: cannot have a translation in TLB if page is not present in memory.</td>
</tr>
<tr>
<td>Hit</td>
<td>Miss</td>
<td>Hit</td>
<td>Impossible: cannot have a translation in TLB if page is not present in memory.</td>
</tr>
<tr>
<td>Miss</td>
<td>Miss</td>
<td>Hit</td>
<td>Impossible: data cannot be allowed in cache if the page is not in memory.</td>
</tr>
</tbody>
</table>

FIGURE 5.32  The possible combinations of events in the TLB, virtual memory system, and cache. Three of these combinations are impossible, and one is possible (TLB hit, virtual memory hit, cache miss) but never detected.
**Question:** A program tries to load a word at X that causes a TLB miss but not a page fault. Are the following statements TRUE or FALSE?

1) The page table does not contain a valid mapping for the virtual page corresponding to the address X

2) The word that the program is trying to load is present in physical memory

A) F F
B) F T
C) T F
D) T T
Aside: Context Switching

• How does a single processor run many programs at once?

• *Context switch*: Changing of internal state of processor (switching between processes)
  – Save register values (and PC) and change value in Page Table Base register

• What happens to the TLB?
  – Current entries are for different process
  – Set all entries to invalid on context switch
Virtual Memory Summary

• User program view:
  – Contiguous memory
  – Start from some set VA
  – “Infinitely” large
  – Is the only running program

• Reality:
  – Non-contiguous memory
  – Start wherever available memory is
  – Finite size
  – Many programs running simultaneously

• 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, Sharing

• Implementation:
  – Divide memory into chunks (pages)
  – OS controls page table that maps virtual into physical addresses
  – memory as a cache for disk
  – TLB is a cache for the page table