Page Table

- A page table: mapping function
  - 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
    - Historically, 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!

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

Paging/Virtual Memory Multiple Processes

User A:
- Virtual Memory
- Physical Memory
- 64 MB
- Static
- Code

User B:
- Virtual Memory
- Physical Memory
- Stack
- Page Table
- A Page Table
- B Page Table
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:</td>
<td>Page Size: 4K-8KB</td>
</tr>
<tr>
<td>Placement:</td>
<td>Fully Associative</td>
</tr>
<tr>
<td>Replacement:</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td></td>
<td>(LRU)</td>
</tr>
<tr>
<td>Write Thru or Back</td>
<td>Write Back</td>
</tr>
</tbody>
</table>

Notes on Page Table

- 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)
- Will add details, but Page Table is essence of Virtual Memory

- VM Problems and Solutions
  - TLB
  - Paged Page Tables

Translation Look-Aside Buffers (TLBs)

- TLBs usually small, typically 32 - 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>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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

What if the data is on disk?

- We load the page off the disk into a free block of memory, using a DMA (Direct Memory Access – very fast!) transfer
  - Meantime 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 choose 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

Question

- Why is the TLB so small yet so effective?
  - Because each entry corresponds to pagesize # of addresses

- Why does the TLB typically have high associativity? What is the “associativity” of VA\(\mapsto\)PA mappings?
  - Because the miss penalty dominates the AMAT for VM.
  - High associativity \(\Rightarrow\) lower miss rates.
    - VPN\(\mapsto\)PPN mappings are fully associative

Virtual Memory Problem #1 Recap

- Slow:
  - Every memory access requires:
    - 1 access to PT to get VPN\(\mapsto\)PPN translation
    - 1 access to MEM to get data at PA
- Solution:
  - Cache the Page Table
    - Make common case fast
    - PT cache called “TLB”
  - “block size” is just 1 VPN\(\mapsto\)PN mapping
  - TLB associativity

Virtual Memory Problem #2

- Page Table too big!
  - 4GB Virtual Memory ÷ 1 KB page
    \(\Rightarrow\) ~ 4 million Page Table Entries
    \(\Rightarrow\) 16 MB just for Page Table for 1 process, 8 processes \(\Rightarrow\) 256 MB for Page Tables!
- Spatial Locality to the rescue
  - Each page is 4 KB, lots of nearby references
  - But large page size wastes resources
- Pages in program’s working set will exhibit temporal and spatial locality.
  - So…
Solutions

- Page the Page Table itself!
  - Works, but must be careful with never-ending page faults
  - Pin some PT pages to memory
- 2-level page table
  - Solutions tradeoff in-memory PT size for slower TLB miss
    - Make TLB large enough, highly associative so rarely miss on address translation
    - CS 162 will go over more options and in greater depth

Page Table Shrink:

- Single Page Table
  - Page Number Offset
    - 20 bits 12 bits

- Multilevel Page Table
  - Super Page No. Page Number Offset
    - 10 bits 10 bits 12 bits

  - Only have second level page table for valid entries of super level page table
  - Book Exercises explore exact space savings

Administrivia

- Proj 4 Out, Due next week
- HW 78, soon
- Final, next Friday

Three Advantages of Virtual Memory

1) Translation:
   - Program can be given consistent view of memory, even though physical memory is scrambled (illusion of contiguous memory)
   - All programs starting at same set address
   - Illusion of ~ infinite memory (2^32 or 2^64 bytes)
   - 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

Cache, Proc and VM in IF (A Fine Slide)

- Fetch PC
- Trap os
- tlb hit?
- pt "hit"?
- Mem hit?
- WB if dirty
- Cache hit?
- Load into IR
- Cache full?
- Evict victim
- wb
- Write policy?
- WB if dirty
- Evict victim
- XXX
- Restarts
- Load new page
- Update PT
- Update TLB
- Restart
- Stack
- Heap
- Static
- Code
- Virtual Memory
- Physical Memory
- 64 MB
- 2nd Level Page Tables
- Super Page Table
- Code
- Static
- Heap
- Stack
Cache, Proc and VM in IF (A Fine Slide)

Fetch PC

Cache hit?

EXE; PC ← PC+4

Load into IR

Update TLB

Restart

Update PT

Load new page

Victim to disk

Victim?

Free mem?

n

Update TLB

Restart

Load block

Where is the page fault?

$&VM Review: 4 Qs for any Mem. Hierarchy

Q1: Where can a block be placed in the upper level? (Block placement)

Q2: How is a block found if it is in the upper level? (Block identification)

Q3: Which block should be replaced on a miss? (Block replacement)

Q4: What happens on a write? (Write strategy)

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 Mod Number Sets

Set Select

Data Select

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)

Miss Rates

<table>
<thead>
<tr>
<th>Size</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>Ran</td>
<td>LRU</td>
<td>Ran</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</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 that memory be updated when block is replaced
  • OS flushes cache before I/O !!!

• Performance trade-offs?
  • WT: read misses cannot result in writes
  • WB: no writes of repeated writes
**Peer Instruction (1/3)**

- 40-bit virtual address, 16 KB page

<table>
<thead>
<tr>
<th>Virtual Page Number (7 bits)</th>
<th>Page Offset (7 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 36-bit physical address

<table>
<thead>
<tr>
<th>Physical Page Number (7 bits)</th>
<th>Page Offset (7 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Number of bits in Virtual Page Number/Page offset, Physical Page Number/Page offset?

| 1: | 22/18 (VPN/PO), 22/14 (PPN/PO) |
| 2: | 24/16, 20/16 |
| 3: | 26/14, 22/14 |
| 4: | 26/14, 26/10 |
| 5: | 28/12, 24/12 |

**Peer Instruction (1/3) Answer**

- 40-bit virtual address, 16 KB (214 B)

<table>
<thead>
<tr>
<th>Virtual Page Number (26 bits)</th>
<th>Page Offset (14 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 36-bit virtual address, 16 KB (214 B)

<table>
<thead>
<tr>
<th>Physical Page Number (22 bits)</th>
<th>Page Offset (14 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Number of bits in Virtual Page Number/Page offset, Physical Page Number/Page offset?

| 1: | 22/18 (VPN/PO), 22/14 (PPN/PO) |
| 2: | 24/16, 20/16 |
| 3: | 26/14, 22/14 |
| 4: | 26/14, 26/10 |
| 5: | 28/12, 24/12 |

**Peer Instruction (2/3): 40b VA, 36b PA**

- 2-way set-assoc, TLB, 256 “slots”, 40b VA:

<table>
<thead>
<tr>
<th>TLB Tag (7 bits)</th>
<th>TLB Index (7 bits)</th>
<th>Page Offset (14 bits)</th>
</tr>
</thead>
</table>

- TLB Entry: Valid bit, Dirty bit, Access Control (say 2 bits), Virtual Page Number, Physical Page Number

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Access (2 bits)</th>
<th>TLB Tag (7 bits)</th>
<th>Physical Page No. (7 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Number of bits in TLB Tag/Index/Entry?

| 1: | 12 / 14 / 38 (TLB Tag/Index/Entry) |
| 2: | 14 / 12 / 40 |
| 3: | 18 / 8 / 44 |
| 4: | 18 / 8 / 58 |

**Peer Instruction (2/3) Answer**

- 2-way set-assoc data cache, 256 (2^8) “slots”, 2 TLB entries per slot => 8 bit index

<table>
<thead>
<tr>
<th>TLB Tag (18 bits)</th>
<th>TLB Index (8 bits)</th>
<th>Page Offset (14 bits)</th>
</tr>
</thead>
</table>

- TLB Entry: Valid bit, Dirty bit, Access Control (2 bits), Virtual Page Number, Physical Page Number

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Access (2 bits)</th>
<th>TLB Tag (18 bits)</th>
<th>Physical Page No. (22 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 1: | 12 / 14 / 38 (TLB Tag/Index/Entry) |
| 2: | 14 / 12 / 40 |
| 3: | 16 / 8 / 44 |
| 4: | 16 / 8 / 58 |

**Peer Instruction (3/3)**

- 2-way set-assoc, 64KB data cache, 64B block

<table>
<thead>
<tr>
<th>Cache Tag (7 bits)</th>
<th>Cache Index (9 bits)</th>
<th>Block Offset (6 bits)</th>
</tr>
</thead>
</table>

- Data Cache Entry: Valid bit, Dirty bit, Cache tag + 7 bits of Data

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Cache Tag (7 bits)</th>
<th>Cache Data (64 Bytes = 512 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Number of bits in Data cache Tag/Index/Offset/Entry?

| 1: | 12 / 9 / 14 / 87 (Tag/Index/Offset/Entry) |
| 2: | 20 / 10 / 6 / 86 |
| 3: | 20 / 10 / 6 / 534 |
| 4: | 21 / 9 / 6 / 87 |
| 5: | 21 / 9 / 6 / 535 |

**Peer Instruction (3/3) Answer**

- 2-way set-assoc data cache, 64K/1K (2^16) “slots”, 2 entries per slot => 9 bit index

<table>
<thead>
<tr>
<th>Cache Tag (21 bits)</th>
<th>Cache Index (9 bits)</th>
<th>Block Offset (6 bits)</th>
</tr>
</thead>
</table>

- Data Cache Entry: Valid bit, Dirty bit, Cache tag + 64 Bytes of Data

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Cache Tag (21 bits)</th>
<th>Cache Data (64 Bytes = 512 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 1: | 12 / 9 / 14 / 87 (Tag/Index/Offset/Entry) |
| 2: | 20 / 10 / 6 / 86 |
| 3: | 20 / 10 / 6 / 534 |
| 4: | 21 / 9 / 6 / 87 |
| 5: | 21 / 9 / 6 / 535 |