Address Mapping: Page Table

Virtual Address:

VPN offset

Page Table

<table>
<thead>
<tr>
<th>V</th>
<th>A.R.</th>
<th>P.P.A.</th>
</tr>
</thead>
</table>

Val Access Rights

Physical Page Address

Physical Memory Address

Page Table located in physical memory

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)

Page Table

<table>
<thead>
<tr>
<th>V</th>
<th>A.R.</th>
<th>P.P.N.</th>
</tr>
</thead>
</table>

Val Access Rights

Physical Page Number

P.T.E. (Page Table Entry)

- 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
- Stack
- Page Table
- Code
- Static
- User B:
- Virtual Memory
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>N-way Set Associative</td>
<td></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

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

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

On TLB miss, get page table entry from main memory

Typical TLB Format

<table>
<thead>
<tr>
<th></th>
<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>Processor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<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→PA mappings?
  • Because the miss penalty dominates the AMAT for VM.
  • High associativity → lower miss rates.
  • VPN→PPN mappings are fully associative

Virtual Memory Problem #1 Recap

• Slow:
  • Every memory access requires:
    - 1 access to PT to get VPN→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→PN mapping
  • TLB associativity

Virtual Memory Problem #2

• Page Table too big!
  • 4GB Virtual Memory ÷ 1 KB page
    ⇒ ~ 4 million Page Table Entries
    ⇒ 16 MB just for Page Table for 1 process
    ⇒ 8 processes ⇒ 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 3 Due Friday
- Proj 4 Out Soon

- HW 8? Probably, but it will be short

**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
- Update TLB
- Free mem?
- Victim to disk
- Load new page
- Update PT
- Update TLB
- Restart

**Where is the page fault?**

### Q1: Where block placed in upper level?

- Fully associative, direct mapped, 2-way set associative
- S.A. Mapping = Block Number Mod Number Sets

<table>
<thead>
<tr>
<th>Block</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
</tbody>
</table>

**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)

### Q4: What to do on a write hit?

- **Write-through**
  - write 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

### $&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)