Virtual Memory Motivations

Motivations
- Programming burdens of small main memory
  - Programmers explicitly load/unload overlays
- Sharing of memory among multiple programs
  - Efficient and safe

Virtual Memory (VM) creates an illustration
- Extremely large address space
- Much faster than disk (as fast as main memory)
Virtual Memory Idea

- Use main memory as a “cache” for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
Virtual vs. Physical Address

- CPU and OS translate virtual addresses to physical addresses
  - VM “block” is called a page (e.g., 4KB size)
  - VM translation “miss” is called a page fault

![Diagram of virtual vs. physical addresses]
Address Translation

- Fixed-size pages (e.g., 4KB)
Page Tables

- Stores the VA=>PA mapping information
  - Page table is stored in **physical memory**
  - Array of page table entries (PTEs), indexed by virtual page number
  - Page table register points to page table in memory

- If a page is present in memory
  - PTE stores the physical page number
  - Plus other status bits (referenced, dirty, ...)

- If a page is not present in memory
  - PTE can refer to location in swap space on disk
Translation Using a Page Table

Virtual address

31 30 29 28 27 ...................... 15 14 13 12 11 10 9 8 ............ 3 2 1 0

Virtual page number  Page offset

Page table register

Valid

Physical page number

If 0 then page is not present in memory

Physical page number  Page offset

Physical address
Example: Page Table Size

- 32-bit virtual address
- 4KB page size
- 4 bytes per page table entry (PTE)
- \# of PTEs: \(2^{32}/2^{12} = 2^{20} = 1\text{ million}\)
- Page table size: 1M x 4B = 4MB
- One page table per process
Mapping Pages to Storage

- Swap space: the space on disk created by OS for all the pages of a process
- Memory data is a subset of data in swap space
Page Fault Penalty

- On page fault, the page is fetched from disk
  - Takes millions of clock cycles
  - Handled by OS code
- Try to minimize page fault rate
  - Large page size
  - Fully associative placement
  - Smart replacement algorithms
  - Write-back instead of write-through
Replacement

- To reduce page fault rate
  - Prefer least-recently used (LRU) replacement
  - Reference bit in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently
Writes

- Disk writes take millions of cycles
  - Write through is impractical
  - Use write-back
  - Dirty bit in PTE is set when page is written
  - Flush cache!
Fast Translation Using a TLB

- Problem: address translation may require an extra memory references
  - One to access the PTE in memory
  - Then the actual memory access

- However, access to page tables has good locality
  - So use a fast cache of PTEs within the CPU
  - Called a Translation Look-aside Buffer (TLB)
  - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
  - Misses could be handled by hardware or software
Fast Translation Using a TLB

### TLB

- **Virtual page number**
- **Valid Dirty Ref**
- **Tag**
- **Physical page address**

### Page table

- **Valid Dirty Ref**
- **Physical page or disk address**

### Physical memory

- **Disk storage**
TLB Miss Handler

- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler

- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction
Page Fault Handler

- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
  - If dirty, write to disk first
  - Flush cache if needed
- Read page into memory
- Update page table
- Restart from faulting instruction
TLB Miss, Page Fault, Cache Miss

- A memory reference can encounter 3 types of misses: TLB miss, page fault, and cache miss.
- Best case: hit in TLB & hit in cache.
- Otherwise: 7 combinations.

<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>
Memory Protection

- Different tasks can share parts of their virtual address spaces
  - Need to protect against errant access
  - Requires OS assistance
- Hardware support for OS protection
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., syscall in MIPS)
The Memory Hierarchy

- Common principles apply at all levels of the memory hierarchy
  - Based on notions of caching

- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy
Block Placement

- Determined by associativity
  - Direct mapped (1-way associative)
    - One choice for placement
  - n-way set associative
    - n choices within a set
  - Fully associative
    - Any location

- Higher associativity reduces miss rate
  - Increases complexity, cost, and access time
### Finding a Block

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Location method</th>
<th>Tag comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct mapped</td>
<td>Index</td>
<td>1</td>
</tr>
<tr>
<td>n-way set associative</td>
<td>Set index, then search entries within the set</td>
<td>n</td>
</tr>
<tr>
<td>Fully associative</td>
<td>Search all entries</td>
<td>#entries</td>
</tr>
<tr>
<td></td>
<td>Full lookup table</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Virtual memory
- Full table lookup makes full associativity feasible
- Benefit in reduced miss rate
Replacement

- Choice of entry to replace on a miss
  - Least recently used (LRU)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU, easier to implement

- Virtual memory
  - LRU approximation with hardware support
Write Policy

- **Write-through**
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer

- **Write-back**
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state

- **Virtual memory**
  - Only write-back is feasible, given disk write latency
## Multilevel On-Chip Caches

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>ARM Cortex-A8</th>
<th>Intel Nehalem</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache organization</td>
<td>Split instruction and data caches</td>
<td>Split instruction and data caches</td>
</tr>
<tr>
<td>L1 cache size</td>
<td>32 KiB each for instructions/data</td>
<td>32 KiB each for instructions/data per core</td>
</tr>
<tr>
<td>L1 cache associativity</td>
<td>4-way (I), 4-way (D) set associative</td>
<td>4-way (I), 8-way (D) set associative</td>
</tr>
<tr>
<td>L1 replacement</td>
<td>Random</td>
<td>Approximated LRU</td>
</tr>
<tr>
<td>L1 block size</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L1 write policy</td>
<td>Write-back, Write-allocate(?)</td>
<td>Write-back, No-write-allocate</td>
</tr>
<tr>
<td>L1 hit time (load-use)</td>
<td>1 clock cycle</td>
<td>4 clock cycles, pipelined</td>
</tr>
<tr>
<td>L2 cache organization</td>
<td>Unified (instruction and data)</td>
<td>Unified (instruction and data) per core</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>128 KiB to 1 MiB</td>
<td>256 KiB (0.25 MiB)</td>
</tr>
<tr>
<td>L2 cache associativity</td>
<td>8-way set associative</td>
<td>8-way set associative</td>
</tr>
<tr>
<td>L2 replacement</td>
<td>Random(?)</td>
<td>Approximated LRU</td>
</tr>
<tr>
<td>L2 block size</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L2 write policy</td>
<td>Write-back, Write-allocate (?)</td>
<td>Write-back, Write-allocate</td>
</tr>
<tr>
<td>L2 hit time</td>
<td>11 clock cycles</td>
<td>10 clock cycles</td>
</tr>
<tr>
<td>L3 cache organization</td>
<td>–</td>
<td>Unified (instruction and data)</td>
</tr>
<tr>
<td>L3 cache size</td>
<td>–</td>
<td>8 MiB, shared</td>
</tr>
<tr>
<td>L3 cache associativity</td>
<td>–</td>
<td>16-way set associative</td>
</tr>
<tr>
<td>L3 replacement</td>
<td>–</td>
<td>Approximated LRU</td>
</tr>
<tr>
<td>L3 block size</td>
<td>–</td>
<td>64 bytes</td>
</tr>
<tr>
<td>L3 write policy</td>
<td>–</td>
<td>Write-back, Write-allocate</td>
</tr>
<tr>
<td>L3 hit time</td>
<td>–</td>
<td>35 clock cycles</td>
</tr>
</tbody>
</table>
# 2-Level TLB Organization

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>ARM Cortex-A8</th>
<th>Intel Core i7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual address</td>
<td>32 bits</td>
<td>48 bits</td>
</tr>
<tr>
<td>Physical address</td>
<td>32 bits</td>
<td>44 bits</td>
</tr>
<tr>
<td>Page size</td>
<td>Variable: 4, 16, 64 KiB, 1, 16 MiB</td>
<td>Variable: 4 KiB, 2/4 MiB</td>
</tr>
<tr>
<td>TLB organization</td>
<td>1 TLB for instructions and 1 TLB for data</td>
<td>1 TLB for instructions and 1 TLB for data per core</td>
</tr>
<tr>
<td></td>
<td>Both TLBs are fully associative, with 32 entries, round robin replacement</td>
<td>Both L1 TLBs are four-way set associative, LRU replacement</td>
</tr>
<tr>
<td></td>
<td>TLB misses handled in hardware</td>
<td>L1 I-TLB has 128 entries for small pages, 7 per thread for large pages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L1 D-TLB has 64 entries for small pages, 32 for large pages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The L2 TLB is four-way set associative, LRU replacement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The L2 TLB has 512 entries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TLB misses handled in hardware</td>
</tr>
</tbody>
</table>
Memory Hierarchy Summary

- Fast memories are small, large memories are slow
  - We really want fast, large memories 😞
  - Caching gives this illusion 😊

- Principle of locality
  - Programs use a small part of their memory space frequently

- Memory hierarchy
  - L1 cache ↔ L2 cache ↔ ... ↔ DRAM memory ↔ disk

- Memory system design is critical for multiprocessors