Lecture 13: Virtual Memory Management

CSC 469H1F
Fall 2006
Angela Demke Brown

Memory Management Requirements

• Relocation
  • Programmers don’t know what physical memory will be available when their programs run
  • Need some type of address translation

• Protection
  • A program’s memory should be protected from unwanted access by other processes, both intentional and accidental
  • Requires hardware support

• Sharing
  • Need ways to specify and control what sharing is allowed

• Logical/Physical Organization
  • Map between program structures and linear array of bytes
  • Manage transfers between disk and main memory

Topics

• Review virtual memory basics
• Large (64-bit) virtual address spaces
• Multiple Page Sizes
• Placement policy and cache effects
• NUMA multiprocessor memory management
• Distributed shared memory

Virtual address space

- process address space (A.S.) layout
  - logical or virtual A.S.
- CPU generates logical addresses in this space as program executes
  - Called virtual addresses
- Memory system must see physical (real) address
  - Translation is done by memory management unit (MMU)
- Physical memory must be allocated for each virtual location used by the program

Memory Management Activities

- Relocation
- Programmers don’t know what physical memory will be available when their programs run
  - Need some type of address translation

- Protection
- A program’s memory should be protected from unwanted access by other processes, both intentional and accidental
  - Requires hardware support

- Sharing
- Need ways to specify and control what sharing is allowed

- Logical/Physical Organization
- Map between program structures and linear array of bytes
  - Manage transfers between disk and main memory

Virtual address space

- process address space (A.S.) layout
  - logical or virtual A.S.
- CPU generates logical addresses in this space as program executes
  - Called virtual addresses
- Memory system must see physical (real) address
  - Translation is done by memory management unit (MMU)
- Physical memory must be allocated for each virtual location used by the program

Memory Management

- Relocation
  - Programmers don’t know what physical memory will be available when their programs run
  - Need some type of address translation

- Protection
  - A program’s memory should be protected from unwanted access by other processes, both intentional and accidental
  - Requires hardware support

- Sharing
  - Need ways to specify and control what sharing is allowed

- Logical/Physical Organization
  - Map between program structures and linear array of bytes
  - Manage transfers between disk and main memory
Paging

- Partition memory into equal, fixed-size chunks
  - called page frames or simply frames
- Divide processes' memory into chunks of the same size
  - These are called pages
- Any page can be assigned to any free page frame
  - No external fragmentation
  - Minimal internal fragmentation
- First seen in CTSS circa 1961
- Demand paging (automatic transfer to/from backing store) first used in the Atlas computer
  - Described in a 2-page CACM article, 1961

Atlas virtual memory

- Inverted page table (entry per physical page, records what virtual page is stored there)
  - Only 2048 entries, stored in registers, searched in parallel
  - Missing pages fetched on demand from drum into core
  - Victim also selected on demand

"Typical" Address Translation

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page number</td>
<td>Page frame</td>
</tr>
<tr>
<td>Offset</td>
<td>Offset</td>
</tr>
<tr>
<td>Page table</td>
<td>Physical Memory</td>
</tr>
</tbody>
</table>

Page tables - space limitations

- Memory required for page table can be large
  - Need one PTE per page
  - 32 bit virtual address space w/ 4K pages = 2^20 PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
- Solution 1: Hierarchical page tables

Physical Memory

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master page number</td>
<td>Secondary</td>
</tr>
<tr>
<td>Page table</td>
<td>Page frame</td>
</tr>
<tr>
<td>Offset</td>
<td>Offset</td>
</tr>
<tr>
<td>Physical Page Table</td>
<td>Secondary Page Table</td>
</tr>
</tbody>
</table>
64-bit address spaces

• Suppose we just extended the hierarchical page tables with more levels
  • 4K pages → 52 bits for page numbers
  • Maximum 1024 entries per level → 6 levels
    • Too much overhead
  • 16K pages → 48 bits for page numbers
  • Maximum 4096 entries per level → 4 levels
    • Better, but still a lot
• "A new page table for 64-bit address spaces", Talluri, Hill & Khalidi, SOSP ’95
  • Introduces clustered page tables, building on the concept of hashed page tables

Recall Hashed Page Tables

• Hash function maps virtual page number (VPN) to bucket in fixed-size hash table
• Search entries at that bucket for matching VPN

Clustered page tables

• Similar to hashed page tables
  • But each entry stores mapping information for several consecutive pages with a single key
  • Hashed page tables with subblocking

Hashed Page Tables

• Hash table should have 1 bucket per physical page to keep expected chain length short
  ✓ Overhead is fixed, good for sparse address spaces
  ✗ Overhead is high (200%, or 16 bytes for 8 bytes of mapping info)
    • Next field can be eliminated with fixed number of PTEs per bucket (PowerPC)
  • Want fixed, low overhead for both sparse and dense address spaces
Clustered Page Tables

- Better overhead than hashed page tables
  - Example: subblock factor 16, 16 64-bit words \(ightarrow\) 144 bytes per clustered PTE
  - Break even if 6 mappings used (same overhead as hashed PTE)
  - Roughly 1/3 less space if all mappings used
  - Can use more space if address space is very sparse
    - Use smaller subblock factor
- Smaller hash table or shorter chains \rightarrow\) more efficient access
  - But can be worse if PTEs span multiple cache lines

Page tables - time overhead

- Each virtual memory reference requires multiple physical memory references to complete
  - 1 per level in hierarchical tables + actual data access
- Solution: cache recently used translations in MMU
  - Translation lookaside buffer (TLB)
  - Fully associative cache (all entries looked up in parallel)
    - Indexed by virtual page numbers
    - Entries are PTEs (entries from page tables)
  - With PTE + offset, can directly calculate physical address

TLB performance

- TLB hit rates critical to performance
  - TLB reach == fraction of the virtual address space covered by the TLB
    - Depends on page size, number of TLB entries
  - TLB size is fixed (typically small, 2048 entries or less)
    - 4 KB page \rightarrow\) TLB reach is 2048*4K = 8 MB
    - 16 KB page \rightarrow\) TLB reach is 64 MB
    - Miniscule compared to data sets used by applications today
  - Just using a larger page size for everything is problematic
    - Internal fragmentation
  - Solution: support multiple page sizes

Superpage TLBs

- Superpages: page sizes are power-of-two multiples of the base page size
  - Must be aligned in both virtual and physical memory (e.g. 4 MB superpage must begin on a 4 MB address boundary in both spaces)
    - Superpage mapping for size of \(2^n\)
    - TLB entry (copy of PTE) includes page size
    - Supported by MMUs in many processors
      - MIPS, UltraSPARC, Alpha, PowerPC ...
      - Itanium II sizes: 4K, 8K, 16K, 64K, 256K, 1M, 4M, 16M, 64M, 256M, 1G, 4G
Subblock TLBs

- Subblocking associates multiple physical page numbers (PPN) with each TLB tag
  - TLB tag is Virtual page block number (VPBN)
  - Subblocks must be aligned in virtual address space, but each virtual page has a separate PPN so they need not be aligned in physical space
  - Supported by MIPS R4x00 processors (subblock factor of 2)
  - Increases size of TLB entry vs. superpages
- Partial subblocking blends ideas
  - TLB entry stores only one PPN, but multiple valid bits
  - subblocks must be aligned, but not all pages must be valid

Pentium Address Translation