Paging: Smaller Tables

CMPU 334 – Operating Systems
Jason Waterman
Simple Memory System

- 14-bit virtual addresses, 12-bit physical address, 64 byte page size
- TLB: 16 entries, 4-way associative

<table>
<thead>
<tr>
<th>VPN</th>
<th>PFN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>

Page Table: first 16 out of 256 entries shown
Address Translation Examples

Virtual Address: \( \text{0x03D4} \)

Virtual Address: \( \text{0x0020} \)

9/30/2019 CMPU 334 -- Operating Systems
Address Translation Examples

Virtual Address: 0x03D4

- VPN ___
- TLBI ___
- TLBT ____
- TLB Hit? ____
- PFN: _____
- Physical Address: ____________

Virtual Address: 0x0020

- VPN ___
- TLBI ___
- TLBT ____
- TLB Hit? ____
- PFN: _____
- Physical Address: ____________
The story so far... Linear Page Tables

- We usually have one page table for every process in the system
  - Assume that 32-bit address space with 4KB pages and 4-byte page-table entry

\[
\text{Page table size} = \frac{2^{32}}{2^{12}} \times 4\text{Byte} = 4\text{MByte}
\]

Page table are too big and thus consume too much memory
Idea: Larger Page Size

- Assume that 32-bit address space with 16KB pages and 4-byte page-table entry

\[
\frac{2^{32}}{2^{16}} \times 4 = 1MB \quad \text{per page table}
\]

Big pages lead to internal fragmentation
Tiny Page Table Example

- Most of the Page Table is unused
- Filled with invalid entries

A 16KB Address Space with 1KB Pages

A Page Table For 16KB Address Space
Hybrid Approach: Paging and Segments

- Use base and bounds registers to reduce the memory overhead of page tables
  - Base holds the **physical address of the page table** of that segment
  - Bounds register indicates the end of the page table (i.e., how many pages in segment)

- Each process has **three** page tables associated with it
  - When process is running, the base register for each segment contains the physical address of a linear page table for that segment

- We still can have page table waste with a large but sparse heap
- Arbitrary sized page tables can lead to external fragmentation

<table>
<thead>
<tr>
<th>Seg value</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>unused segment</td>
</tr>
<tr>
<td>01</td>
<td>code</td>
</tr>
<tr>
<td>10</td>
<td>heap</td>
</tr>
<tr>
<td>11</td>
<td>stack</td>
</tr>
</tbody>
</table>
Multi-Level Page Table

- Chop up the page table into page-sized units
- If an entire page of page-table entries is invalid, don’t allocate that page of the page table
- To track whether a page of the page table is valid, use a new structure, called page directory

Linear (Left) And Multi-Level (Right) Page Tables
A Detailed Multi-Level Example

A 16-KB Address Space With 64-byte Pages

<table>
<thead>
<tr>
<th>Address space</th>
<th>16 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page size</td>
<td>64 byte</td>
</tr>
<tr>
<td>Virtual address</td>
<td>14 bit</td>
</tr>
<tr>
<td>Offset</td>
<td>6 bit</td>
</tr>
<tr>
<td>VPN</td>
<td>8 bit</td>
</tr>
<tr>
<td>Number of Page table entries</td>
<td>$2^8$ (256)</td>
</tr>
</tbody>
</table>

- Pages 0 and 1 for code
- Pages 4 and 5 for heap
- Pages 254 and 255 for the stack
- Rest of the address space is unallocated
A Detailed Multi-Level Example: Page Directory Index

- Divide the page table entries into pages
  - Assume 4 byte page table entry
  - Page size is 64 bytes
  - Each page can hold 16 page table entries
  - Need 16 pages to hold all the entries (256/16)

- Page Directory
  - One Page Directory Entry (PDE) for each page of the page table
  - Page Directory Index is 4-bits
  - If a PDE is valid, there is a Page Table in memory
A Detailed Multi-Level Example: Page Table Index

- The PDE is valid we use the Page Table Index to get the PTE
  - The Page Table Index is the offset into the Page Table Pointed to by the PDE
- If the PDE is not valid, we have an illegal memory access

<table>
<thead>
<tr>
<th>Page Directory (PFN) valid?</th>
<th>Page of PT (@PFN:100)</th>
<th>Page of PT (@PFN:101)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFN</td>
<td>PFN valid</td>
<td>prot</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>80</td>
<td>1</td>
<td>rw-</td>
</tr>
<tr>
<td>59</td>
<td>1</td>
<td>rw-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

14-bits Virtual address

13 12 11 10 9 8 7 6 5 4 3 2 1 0
A more realistic example

Virtual address 30 bit
Page size 512 byte
Offset 9 bit
VPN 21 bit
More than Two Level

• In some cases, more than two levels are needed

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>30 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page size</td>
<td>512 byte</td>
</tr>
<tr>
<td>Offset</td>
<td>9 bit</td>
</tr>
<tr>
<td>VPN</td>
<td>21 bit</td>
</tr>
<tr>
<td>Page entries per page</td>
<td>128 PTEs</td>
</tr>
<tr>
<td>Page Directory Entries</td>
<td>$2^{14}$ (16K)</td>
</tr>
</tbody>
</table>
More than Two Level : Page Directory

- If our page directory has \(2^{14}\) entries, it spans not one page but 128.
- To remedy this problem, we build a **further level** of the tree, by splitting the page directory itself into multiple pages of the page directory.
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True) // TLB Hit
   if (CanAccess(TlbEntry.ProtectBits) == True)
      Offset = VirtualAddress & OFFSET_MASK
      PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
      Register = AccessMemory(PhysAddr)
   else
      RaiseException(PROTECTION_FAULT)
else // TLB Miss
   // first, get page directory entry
   PDIndex = (VPN & PD_MASK) >> PD_SHIFT
   PDEAddr = PDBR + (PDIndex * sizeof(PDE))
   PDE = AccessMemory(PDEAddr)
   if (PDE.Valid == False)
      RaiseException(SEGMETATION_FAULT)
   else
      // PDE is valid: now fetch PTE from page table
      PTIndex = (VPN & PT_MASK) >> PT_SHIFT
      PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
      PTE = AccessMemory(PTEAddr)
      if (PTE.Valid == False)
         RaiseException(SEGMETATION_FAULT)
   else if (CanAccess(PTE.ProtectBits) == False)
      RaiseException(PROTECTION_FAULT)
   else
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
      RetryInstruction()
Intel Core i7 Memory System

Processor package

Core x4

- Registers
  - L1 d-cache: 32 KB, 8-way
  - L1 i-cache: 32 KB, 8-way
- Instruction fetch
  - L1 d-TLB: 64 entries, 4-way
  - L1 i-TLB: 128 entries, 4-way
- MMU (addr translation)
  - L2 unified TLB: 512 entries, 4-way
- L2 unified cache: 256 KB, 8-way
- L3 unified cache: 8 MB, 16-way (shared by all cores)

DDR3 Memory controller
- 3 x 64 bit @ 10.66 GB/s
- 32 GB/s total (shared by all cores)

Main memory

QuickPath interconnect
- 4 links @ 25.6 GB/s each

To other cores
To I/O bridge

Operating Systems

 CMPU 334 -- Operating Systems
End-to-end Core i7 Address Translation

- **CPU**
- **VPN**
- **VPO**
- **TLBT**
- **TLBI**
- **TLB (16 sets, 4 entries/set)**
- **L1 TLB (16 sets, 4 entries/set)**
- **VPN1**
- **VPN2**
- **VPN3**
- **VPN4**
- **CR3**
- **PTE**
- **Page tables**
- **PPN**
- **PPO**
- **Physical address (PA)**
- **L1, L3, and main memory**
- **Result**
- **L1 hit**
- **L1 miss**
- **L1 d-cache (64 sets, 8 lines/set)**
- **CT**
- **CI**
- **CO**

**Virtual address (VA)**

36

12

32

4

9

9

9

9

9

40

12

40

6

6

32/64

**Page tables**
Core i7 Level 1-3 Page Table Entries

| 63 | 62 | 52 | 51 | 12 | 11 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| XD | Unused | Page table physical base address | Unused | G | PS | A | CD | WT | U/S | R/W | P=1 |    |    |    |    |    |

Available for OS (page table location on disk) | P=0

Each entry references a child page table. Significant fields:

- **P**: Child page table present in physical memory (1) or not (0).
- **R/W**: Read-only or read-write access access permission for all reachable pages.
- **U/S**: user or supervisor (kernel) mode access permission for all reachable pages.
- **WT**: Write-through or write-back cache policy for the child page table.
- **A**: Reference bit (set by MMU on reads and writes, cleared by software).
- **PS**: Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

**Page table physical base address**: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

**XD**: Disable or enable instruction fetches from all pages reachable from this PTE.
Core i7 Level 4 Page Table Entries

<table>
<thead>
<tr>
<th>XD</th>
<th>Unused</th>
<th>Page physical base address</th>
<th>Unused</th>
<th>G</th>
<th>D</th>
<th>A</th>
<th>CD</th>
<th>WT</th>
<th>U/S</th>
<th>R/W</th>
<th>P=1</th>
</tr>
</thead>
</table>

Available for OS (page location on disk)  
P=0

Each entry references a 4K child page. Significant fields:

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for child page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

**Page physical base address**: 40 most significant bits of physical page address  
(forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.
Multi-level Page Tables: Advantage & Disadvantage

• Advantages
  • Only allocates page-table space in proportion to the amount of address space you are using
  • The OS can grab the next free page when it needs to allocate or grow a page table

• Disadvantages
  • Multi-level table is a small example of a time-space trade-off
  • Complexity
Inverted Page Tables

• Keeping a **single page** table that has an entry for each physical page of the system
  • PID, VPN, Protection bits

• The entry tells us which process is using this page, and which virtual page of that process maps to this physical page

• Steps in translation
  1. Hash PID and VPN to get HAT index
  2. Lookup a Physical Frame Number in the HAT
  3. Look in the inverted page table entry to see if the PID and VPN match
  4. If they don’t match, follow the pointer to the next link