Memory 2: Virtual Memory (Con’t), Caching and TLBs

October 12th, 2021
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu
Recall: General Address translation

• Consequently, two views of memory:
  – View from the CPU (what program sees, virtual memory)
  – View from memory (physical memory)
  – Translation box (Memory Management Unit or MMU) converts between the two views

• Translation ⇒ much easier to implement protection!
  – If task A cannot even gain access to task B's data, no way for A to adversely affect B
  – Extra benefit: every program can be linked/loaded into same region of user address space
Implementation of Multi-Segment Model

- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    » x86 Example: mov [es:bx],ax.
- What is “V/N” (valid / not valid)?
  - Can mark segments as invalid; requires check as well
Example: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Virtual Address Format

Virtual Address Space

Physical Address Space
Example: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Virtual Address Format

Virtual Address Space

Physical Address Space
Example: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Virtual Address Format

Virtual Address Space

Physical Address Space
Example: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Virtual Address Format:

```
0x0000 code
0x4000 data
0x8000 shared
0xC000 stack
```

Physical Address Space:

```
0x0000 stack
0x4000 code
0x4800 data
0x5C00 data
0xF000 shared
```

Space for Other Apps

Might be shared

Shared with Other Apps
Let’s simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get “la $a0, varx”
   Move 0x4050 → $a0, Move PC+4→PC
Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get “la $a0, varx”
   Move 0x4050 → $a0, Move PC+4→PC

2. Fetch 0x244. Translated to Physical=0x4244. Get “jal strlen”
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC
Let’s simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get “la $a0, varx”
   Move 0x4050 → $a0, Move PC+4 → PC

2. Fetch 0x244. Translated to Physical=0x4244. Get “jal strlen”
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC

3. Fetch 0x360. Translated to Physical=0x4360. Get “li $v0, 0”
   Move 0x0000 → $v0, Move PC+4 → PC
Let's simulate a bit of this code to see what happens (PC=0x0240):

1. Fetch 0x0240 (0000 0010 0100 0000). Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get “la $a0, varx”
   Move 0x4050 → $a0, Move PC+4→PC

2. Fetch 0x0244. Translated to Physical=0x4244. Get “jal strlen”
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC

3. Fetch 0x0360. Translated to Physical=0x4360. Get “li $v0, 0”
   Move 0x0000 → $v0, Move PC+4→PC

4. Fetch 0x0364. Translated to Physical=0x4364. Get “lb $t0, ($a0)”
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050 (0100 0000 0101 0000). Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850→$t0, Move PC+4→PC
Observations about Segmentation

• Translation on every instruction fetch, load or store
• Virtual address space has holes
  – Segmentation efficient for sparse address spaces
• When it is OK to address outside valid range?
  – This is how the stack (and heap?) allowed to grow
  – For instance, stack takes fault, system automatically increases size of stack
• Need protection mode in segment table
  – For example, code segment would be read-only
  – Data and stack would be read-write (stores allowed)
• What must be saved/restored on context switch?
  – Segment table stored in CPU, not in memory (small)
  – Might store all of processes memory onto disk when switched (called “swapping”)
What if not all segments fit in memory?

- Extreme form of Context Switch: **Swapping**
  - To make room for next process, some or all of the previous process is moved to disk
    - Likely need to send out complete segments
  - This greatly increases the cost of context-switching
- What might be a desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory
Problems with Segmentation

- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk

- **Fragmentation**: wasted space
  - **External**: free gaps between allocated chunks
  - **Internal**: don’t need all memory within allocated chunks
Recall: General Address Translation

Translation Map 1

Translation Map 2

Physical Address Space
Paging: Physical Memory in Fixed Size Chunks

- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    » Can use simple vector of bits to handle allocation:
      00110001110001101 ... 110010
    » Each bit represents page of physical memory
      1 ⇒ allocated, 0 ⇒ free

- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment
### How to Implement Simple Paging?

- **Page Table** (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc)
- **Virtual address mapping**
  - Offset from Virtual address copied to Physical Address
    - Example: 10 bit offset \( \Rightarrow \) 1024-byte pages
  - Virtual page # is all remaining bits
    - Example for 32-bits: \( 32-10 = 22 \) bits, i.e., 4 million entries
  - Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

#### Virtual Address:

<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td>V, R</td>
</tr>
<tr>
<td>page #1</td>
<td>V, R</td>
</tr>
<tr>
<td>page #2</td>
<td>V, R, W</td>
</tr>
<tr>
<td>page #3</td>
<td>V, R, W</td>
</tr>
<tr>
<td>page #4</td>
<td>N</td>
</tr>
<tr>
<td>page #5</td>
<td>V, R, W</td>
</tr>
</tbody>
</table>

#### PageTablePtr

<table>
<thead>
<tr>
<th>PageTableSize</th>
<th>Access Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageTablePtr</td>
<td>Access Error</td>
</tr>
<tr>
<td>PageTableSize</td>
<td>Access Error</td>
</tr>
</tbody>
</table>

#### Physical Address

<table>
<thead>
<tr>
<th>Physical Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>check Perm</td>
<td></td>
</tr>
</tbody>
</table>

#### Example:

- 10 bit offset
  - \( 2^{10} = 1024 \) byte pages
Simple Page Table Example

Example (4 byte pages)

Virtual Memory

0x00

0x04

0x06?

0x08

0x09?

ab

cd
ef

gh

ij

kl

0000 0000

0000 0100

0000 1000

0000 1001

0000 0110

0000 1101

Physical Memory

0x00

0x04

0x08

0x0C

0x10

0x05!

0x0E!

Page Table

0x00

0x04

0x08

0x0C

0x10

0x00

0x04

0x08

0x0C

0x10

0001 0000

0000 1100

0000 0100

0000 0101

0000 1110

0000 0110

0000 0101
What about Sharing?

Virtual Address (Process A):

<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td>V,R</td>
</tr>
<tr>
<td>page #1</td>
<td>V,R</td>
</tr>
<tr>
<td>page #2</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #3</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #4</td>
<td>N</td>
</tr>
<tr>
<td>page #5</td>
<td>V,R,W</td>
</tr>
</tbody>
</table>

Virtual Address (Process B):

<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td>V,R</td>
</tr>
<tr>
<td>page #1</td>
<td>N</td>
</tr>
<tr>
<td>page #2</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #3</td>
<td>N</td>
</tr>
<tr>
<td>page #4</td>
<td>V,R</td>
</tr>
<tr>
<td>page #5</td>
<td>V,R,W</td>
</tr>
</tbody>
</table>

PageTablePtrA

PageTablePtrB

Shared Page

This physical page appears in address space of both processes
Where is page sharing used?

- The “kernel region” of every process has the same page table entries
  - The process cannot access it at user level
  - But on U → K switch, kernel code can access it AS WELL AS the region for THIS user
    » What does the kernel need to do to access other user processes?
- Different processes running same binary!
  - Execute-only, but do not need to duplicate code segments
- User-level system libraries (execute only)
- Shared-memory segments between different processes
  - Can actually share objects directly between processes
    » Must map page into same place in address space!
  - This is a limited form of the sharing that threads have within a single process
Summary: Paging
What happens if stack grows to 1110 0000?
Summary: Paging

Virtual memory view

1111 1111
stack

1110 0000
heap

1000 0000
data

0100 0000
code

Page Table

Physical memory view

Allocate new pages where room!
How big do things get?

• 32-bit address space $\Rightarrow 2^{32}$ bytes (4 GB)
  – Note: “b” = bit, and “B” = byte
  – And for memory:
    » “K” (kilo) $= 2^{10} = 1024 \approx 10^3$ (But not quite!): Sometimes called “Ki” (Kibi)
    » “M” (mega) $= 2^{20} = (1024)^2 = 1,048,576 \approx 10^6$ (But not quite!): Sometimes called “Mi” (Mibi)
    » “G” (giga) $= 2^{30} = (1024)^3 = 1,073,741,824 \approx 10^9$ (But not quite!): Sometimes called “Gi” (Gibi)

• Typical page size: 4 KB
  – how many bits of the address is that? (remember $2^{10} = 1024$)
  – Ans – 4KB = $4 \times 2^{10} = 2^{12} \Rightarrow 12$ bits of the address

• So how big is the simple page table for each process?
  – $2^{32}/2^{12} = 2^{20}$ (that’s about a million entries) $\times$ 4 bytes each $\Rightarrow$ 4 MB
  – When 32-bit machines got started (vax 11/780, intel 80386), 16 MB was a LOT of memory

• How big is a simple page table on a 64-bit processor (x86_64)?
  – $2^{64}/2^{12} = 2^{52}$ (that’s $4.5 \times 10^{15}$ or 4.5 exa-entries) $\times$ 8 bytes each = $36 \times 10^{15}$ bytes or 36 exa-bytes!!! This is a ridiculous amount of memory!
  – This is really a lot of space – for only the page table!!!

• The address space is sparse, i.e. has holes that are not mapped to physical memory
  – So, most of this space is taken up by page tables mapped to nothing
Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit
- What provides protection here?
  - Translation (per process) and dual-mode!
  - Can’t let process alter its own page table!
- Analysis
  - Pros
    » Simple memory allocation
    » Easy to share
  - Con: What if address space is sparse?
    » E.g., on UNIX, code starts at 0, stack starts at $2^{31} - 1$
    » With 1K pages, need 2 million pages table entries!
  - Con: What if table really big?
    » Not all pages used all the time \( \Rightarrow \) would be nice to have working set of page table in memory
- Simple Page table is way too big!
  - Does it all need to be in memory?
  - How about multi-level paging?
  - or combining paging and segmentation
Summary

• Segment Mapping
  – Segment registers within processor
  – Segment ID associated with each access
    » Often comes from portion of virtual address
    » Can come from bits in instruction instead (x86)
  – Each segment contains base and limit information
    » Offset (rest of address) adjusted by adding base

• Page Tables
  – Memory divided into fixed-sized chunks of memory
  – Virtual page number from virtual address mapped through page table to physical page number
  – Offset of virtual address same as physical address
  – Large page tables can be placed into virtual memory

• Next Time: Multi-Level Tables
  – Virtual address mapped to series of tables
  – Permit sparse population of address space
How to Structure a Page Table

- Page Table is a map (function) from VPN to PPN

- Simple page table corresponds to a very large lookup table
  - VPN is index into table, each entry contains PPN

- What other map structures can you think of?
  - Trees?
  - Hash Tables?
Fix for sparse address space: The two-level page table

- Tree of Page Tables
  - “Magic” 10b-10b-12b pattern!
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register (i.e. CR3)
- Valid bits on Page Table Entries
  - Don’t need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use
Announcement

• Midterm scores released! Check Piazza for regrade policy.

• Project 2 Design Doc: due EOD Friday (10/15)

• Homework 3: due EOD Sunday (10/17)

• Fill out midsemester survey for extra credit (check Piazza for details)

• Next lecture: Nathan Pemberton (PhD student)
What is in a Page Table Entry (PTE)?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called “Directories”

<table>
<thead>
<tr>
<th>Page Frame Number (Physical Page Number)</th>
<th>Free (OS)</th>
<th>0</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>11-9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **P**: Present (same as “valid” bit in other architectures)
- **W**: Writeable
- **U**: User accessible
- **PWT**: Page write transparent: external cache write-through
- **PCD**: Page cache disabled (page cannot be cached)
- **A**: Accessed: page has been accessed recently
- **D**: Dirty (PTE only): page has been modified recently
- **PS**: Page Size: PS=1 → 4MB page (directory only). Bottom 22 bits of virtual address serve as offset
Examples of how to use a PTE

• How do we use the PTE?
  – Invalid PTE can imply different things:
    » Region of address space is actually invalid or
    » Page/directory is just somewhere else than memory
  – Validity checked first
    » OS can use other (say) 31 bits for location info
• Usage Example: Demand Paging
  – Keep only active pages in memory
  – Place others on disk and mark their PTEs invalid
• Usage Example: Copy on Write
  – UNIX fork gives copy of parent address space to child
    » Address spaces disconnected after child created
  – How to do this cheaply?
    » Make copy of parent’s page tables (point at same memory)
    » Mark entries in both sets of page tables as read-only
    » Page fault on write creates two copies
• Usage Example: Zero Fill On Demand
  – New data pages must carry no information (say be zeroed)
  – Mark PTEs as invalid; page fault on use gets zeroed page
  – Often, OS creates zeroed pages in background
Sharing with multilevel page tables

- Entire regions of the address space can be efficiently shared
Summary: Two-Level Paging
Summary: Two-Level Paging

Virtual memory view

- stack
- heap
- data
- code

Page Table (level 1)

- stack:
  - Page Table (level 2): null

Page Tables (level 2)

- stack:
  - null

Physical memory view

- stack:
  - Page Table (level 1): null

- heap:
  - Page Table (level 2): 0010101010101000 (0x80)

- data:
  - Page Table (level 2): 0010000000000000

- code:
  - Page Table (level 2): 0010000000000000

Page Tables (level 1)

- null
Multi-level Translation: Segments + Pages

- What about a tree of tables?
  - Lowest level page table ⇒ memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

Virtual Address:

<table>
<thead>
<tr>
<th>Virtual Seg #</th>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base0</td>
<td>Limit0</td>
<td>V</td>
</tr>
<tr>
<td>Base1</td>
<td>Limit1</td>
<td>V</td>
</tr>
<tr>
<td>Base2</td>
<td>Limit2</td>
<td>V</td>
</tr>
<tr>
<td>Base3</td>
<td>Limit3</td>
<td>N</td>
</tr>
<tr>
<td>Base4</td>
<td>Limit4</td>
<td>V</td>
</tr>
<tr>
<td>Base5</td>
<td>Limit5</td>
<td>N</td>
</tr>
<tr>
<td>Base6</td>
<td>Limit6</td>
<td>N</td>
</tr>
<tr>
<td>Base7</td>
<td>Limit7</td>
<td>V</td>
</tr>
</tbody>
</table>

- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)
### What about Sharing (Complete Segment)?

<table>
<thead>
<tr>
<th>Virtual Seg #</th>
<th>Virtual Page #</th>
<th>Offset</th>
<th>Process A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base0</td>
<td>Limit0</td>
<td>V</td>
<td>page #0</td>
</tr>
<tr>
<td>Base1</td>
<td>Limit1</td>
<td>V</td>
<td>page #1</td>
</tr>
<tr>
<td>Base2</td>
<td>Limit2</td>
<td>V</td>
<td>page #2</td>
</tr>
<tr>
<td>Base3</td>
<td>Limit3</td>
<td>N</td>
<td>page #3</td>
</tr>
<tr>
<td>Base4</td>
<td>Limit4</td>
<td>V</td>
<td>page #4</td>
</tr>
<tr>
<td>Base5</td>
<td>Limit5</td>
<td>N</td>
<td>page #5</td>
</tr>
<tr>
<td>Base6</td>
<td>Limit6</td>
<td>N</td>
<td>page #4</td>
</tr>
<tr>
<td>Base7</td>
<td>Limit7</td>
<td>V</td>
<td>page #5</td>
</tr>
</tbody>
</table>

**Virtual Page #**

- **Base0**: Limit0, V
- **Base1**: Limit1, V
- **Base2**: Limit2, V
- **Base3**: Limit3, N
- **Base4**: Limit4, V
- **Base5**: Limit5, N
- **Base6**: Limit6, N
- **Base7**: Limit7, V

**Offset**

- page #0: V, R
- page #1: V, R
- page #2: V, R, W
- page #3: V, R, W
- page #4: N
- page #5: V, R, W

### Shared Segment

<table>
<thead>
<tr>
<th>Virtual Seg #</th>
<th>Virtual Page #</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base0</td>
<td>Limit0</td>
<td>V</td>
</tr>
<tr>
<td>Base1</td>
<td>Limit1</td>
<td>V</td>
</tr>
<tr>
<td>Base2</td>
<td>Limit2</td>
<td>V</td>
</tr>
<tr>
<td>Base3</td>
<td>Limit3</td>
<td>N</td>
</tr>
<tr>
<td>Base4</td>
<td>Limit4</td>
<td>V</td>
</tr>
<tr>
<td>Base5</td>
<td>Limit5</td>
<td>N</td>
</tr>
<tr>
<td>Base6</td>
<td>Limit6</td>
<td>N</td>
</tr>
<tr>
<td>Base7</td>
<td>Limit7</td>
<td>V</td>
</tr>
</tbody>
</table>

**Virtual Page #**

- **Base0**: Limit0, V
- **Base1**: Limit1, V
- **Base2**: Limit2, V
- **Base3**: Limit3, N
- **Base4**: Limit4, V
- **Base5**: Limit5, N
- **Base6**: Limit6, N
- **Base7**: Limit7, V

**Offset**
Multi-level Translation Analysis

• Pros:
  – Only need to allocate as many page table entries as we need for application
    » In other words, sparse address spaces are easy
  – Easy memory allocation
  – Easy Sharing
    » Share at segment or page level (need additional reference counting)

• Cons:
  – One pointer per page (typically 4K – 16K pages today)
  – Page tables need to be contiguous
    » However, the 10b-10b-12b configuration keeps tables to exactly one page in size
  – Two (or more, if >2 levels) lookups per reference
    » Seems very expensive!
Recall: Dual-Mode Operation

• Can a process modify its own translation tables? **NO!**
  – If it could, could get access to all of physical memory (no protection!)
• To Assist with Protection, **Hardware** provides at least two modes (Dual-Mode Operation):
  – “Kernel” mode (or “supervisor” or “protected”)
  – “User” mode (Normal program mode)
  – Mode set with bit(s) in control register only accessible in Kernel mode
  – Kernel can easily switch to user mode; User program must invoke an exception of some sort to get back to kernel mode (more in moment)
• Note that x86 model actually has more modes:
  – Traditionally, four “rings” representing priority; most OSes use only two:
    » Ring 0 ⇒ Kernel mode, Ring 3 ⇒ User mode
    » Called “Current Privilege Level” or CPL
  – Newer processors have additional mode for hypervisor (“Ring -1”)
• Certain operations restricted to Kernel mode:
  – Modifying page table base, and segment descriptor tables
    » Have to transition into Kernel mode before you can change them!
  – Also, all page-table pages must be mapped only in kernel mode
X86 Segment Descriptors (32-bit Protected Mode)

- Segments are implicit in the instruction (e.g. code segments) or part of the instruction
  - There are 6 registers: SS, CS, DS, ES, FS, GS
- What is in a segment register?
  - A pointer to the actual segment description:
  - G/L selects between GDT and LDT tables (global vs local descriptor tables)
  - RPL: Requestor’s Privilege Level (RPL of CS ⇒ Current Privilege Level)
- Two registers: GDTR/LDTR hold pointers to global/local descriptor tables in memory
  - Descriptor format (64 bits):

  G: Granularity of segment [ Limit Size ] (0: 16bit, 1: 4KiB unit)
  DB: Default operand size (0: 16bit, 1: 32bit)
  A: Freely available for use by software
  P: Segment present
  DPL: Descriptor Privilege Level: Access requires Max(CPL,RPL)≤DPL
  S: System Segment (0: System, 1: code or data)
  Type: Code, Data, Segment
How are segments used?

• One set of global segments (GDT) for everyone, different set of local segments (LDT) for every process

• In legacy applications (16-bit mode):
  – Segments provide protection for different components of user programs
  – Separate segments for chunks of code, data, stacks
    » RPL of Code Segment ⇒ CPL (Current Privilege Level)
  – Limited to 64K segments

• Modern use in 32-bit Mode:
  – Even though there is full segment functionality, segments are set up as “flattened”, i.e. every segment is 4GB in size
  – One exception: Use of GS (or FS) as a pointer to “Thread Local Storage” (TLS)
    » A thread can make accesses to TLS like this:
      mov eax, gs(0x0)

• Modern use in 64-bit (“long”) mode
  – Most segments (SS, CS, DS, ES) have zero base and no length limits
  – Only FS and GS retain their functionality TLS
### X86_64: Four-level page table!

<table>
<thead>
<tr>
<th>48-bit Virtual Address:</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual P1 index</td>
<td>Virtual P2 index</td>
<td>Virtual P3 index</td>
<td>Virtual P4 index</td>
<td>Offset</td>
<td></td>
</tr>
</tbody>
</table>

- **PageTablePtr**
- **8 bytes**

4096-byte pages (12 bit offset)
Page tables also 4k bytes (pageable)

<table>
<thead>
<tr>
<th>Physical Address:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(40-50 bits)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Page #</th>
<th>12bit Offset</th>
</tr>
</thead>
</table>

### IA64: 64bit addresses: Six-level page table?!

<table>
<thead>
<tr>
<th>64bit Virtual Address:</th>
<th>7 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>9 bits</th>
<th>12 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Virtual P1 index</td>
<td>Virtual P2 index</td>
<td>Virtual P3 index</td>
<td>Virtual P4 index</td>
<td>Virtual P5 index</td>
<td>Virtual P6 index</td>
<td>Offset</td>
</tr>
</tbody>
</table>

No!

Too slow
Too many almost-empty tables
Alternative: Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    » Much of process space may be out on disk or not in use

- Answer: use a hash table
  - Called an “Inverted Page Table”
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
    » PowerPC, UltraSPARC, IA64

- Cons:
  - Complexity of managing hash chains: Often in hardware!
  - Poor cache locality of page table
# Address Translation Comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Segmentation</td>
<td>Fast context switching (segment map maintained by CPU)</td>
<td>External fragmentation</td>
</tr>
<tr>
<td>Paging (Single-Level)</td>
<td>No external fragmentation Fast and easy allocation</td>
<td>Large table size (~ virtual memory) Internal fragmentation</td>
</tr>
<tr>
<td>Paged Segmentation</td>
<td>Table size ~ # of pages in virtual memory Fast and easy allocation</td>
<td>Multiple memory references per page access</td>
</tr>
<tr>
<td>Multi-Level Paging</td>
<td>Table size ~ # of pages in physical memory Fast and easy allocation</td>
<td>Hash function more complex No cache locality of page table</td>
</tr>
<tr>
<td>Inverted Page Table</td>
<td>Table size ~ # of pages in physical memory</td>
<td></td>
</tr>
</tbody>
</table>
How is the Translation Accomplished?

- The MMU must translate virtual address to physical address on:
  - Every instruction fetch
  - Every load
  - Every store
- What does the MMU need to do to translate an address?
  - 1-level Page Table
    » Read PTE from memory, check valid, merge address
    » Set “accessed” bit in PTE, Set “dirty bit” on write
  - 2-level Page Table
    » Read and check first level
    » Read, check, and update PTE
  - N-level Page Table …
- MMU does page table Tree Traversal to translate each address
Where and What is the MMU?

- The processor requests READ Virtual-Address to memory system
  - Through the MMU to the cache (to the memory)
- Some time later, the memory system responds with the data stored at the physical address (resulting from virtual $\rightarrow$ physical) translation
  - Fast on a cache hit, slow on a miss
- So what is the MMU doing?
- On every reference (I-fetch, Load, Store) read (multiple levels of) page table entries to get physical frame or FAULT
  - Through the caches to the memory
  - Then read/write the physical location
Recall: CS61c Caching Concept

- **Cache**: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many techniques used today to make computers fast
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc...
- Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Important measure: Average Access time = 
  \[(\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time})\]
Caching is the key to memory system performance.

Average Memory Access Time (AMAT)

\[ \text{AMAT} = (\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time}) \]

Where \( \text{Hit Rate} + \text{Miss Rate} = 1 \)

- \( \text{Hit Rate} = 90\% \Rightarrow \text{AMAT} = (0.9 \times 1) + (0.1 \times 101) = 11.1 \text{ ns} \)
- \( \text{Hit Rate} = 99\% \Rightarrow \text{AMAT} = (0.99 \times 1) + (0.01 \times 101) = 2.01 \text{ ns} \)

\( \text{Miss Time}_{L1} \) includes \( \text{Hit Time}_{L1} \) + Miss Penalty\(_{L1} \equiv \text{Hit Time}_{L1} + \text{AMAT}_{L2} \)
Another Major Reason to Deal with Caching

- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access?
- Solution? Cache translations!
  - Translation Cache: TLB (“Translation Lookaside Buffer”)

Virtual Address: [Virtual Seg #] [Virtual Page #] [Offset]

<table>
<thead>
<tr>
<th>Virtual Page #</th>
<th>Access Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base0 Limit0</td>
<td>V</td>
</tr>
<tr>
<td>Base1 Limit1</td>
<td>V</td>
</tr>
<tr>
<td>Base2 Limit2</td>
<td>V</td>
</tr>
<tr>
<td>Base3 Limit3</td>
<td>N</td>
</tr>
<tr>
<td>Base4 Limit4</td>
<td>V</td>
</tr>
<tr>
<td>Base5 Limit5</td>
<td>N</td>
</tr>
<tr>
<td>Base6 Limit6</td>
<td>N</td>
</tr>
<tr>
<td>Base7 Limit7</td>
<td>V</td>
</tr>
</tbody>
</table>

Physical Address: [Physical Page #] [Offset]

Check Perm

Access Error
Why Does Caching Help? Locality!

- **Temporal Locality** (Locality in Time):
  - Keep recently accessed data items closer to processor

- **Spatial Locality** (Locality in Space):
  - Move contiguous blocks to the upper levels
Recall: Memory Hierarchy

- Caching: Take advantage of the principle of locality to:
  - Present the illusion of having as much memory as in the cheapest technology
  - Provide average speed similar to that offered by the fastest technology

![Diagram of memory hierarchy with a processor at the top, followed by core registers, L1 cache, L2 cache, L3 cache (shared), main memory (DRAM), secondary storage (SSD), secondary storage (disk). Speeds (ns): 0.3, 1, 3, 10-30, 100, 100,000 (0.1 ms), 10,000,000 (10 ms). Sizes (bytes): 100Bs, 10kBs, 100kBs, MBs, GBs, 100GBs, TBs.]

Address Translation needs to occur here
Page table lives here (perhaps cached)
How do we make Address Translation Fast?

- Cache results of recent translations!
  - Different from a traditional cache
  - Cache Page Table Entries using Virtual Page # as the key
Translation Look-Aside Buffer

• Record recent Virtual Page # to Physical Frame # translation
• If present, have the physical address without reading any of the page tables !!!
  – Even if the translation involved multiple levels
  – Caches the end-to-end result
• Was invented by Sir Maurice Wilkes – prior to caches
  – When you come up with a new concept, you get to name it!
  – People realized “if it’s good for page tables, why not the rest of the data in memory?”
• On a TLB miss, the page tables may be cached, so only go to memory when both miss
• Question is one of page locality: does it exist?
  – Instruction accesses spend a lot of time on the same page (since accesses sequential)
  – Stack accesses have definite locality of reference
  – Data accesses have less page locality, but still some…
• Can we have a TLB hierarchy?
  – Sure: multiple levels at different sizes/speeds
Summary

• Page Tables
  – Memory divided into fixed-sized chunks of memory
  – Virtual page number from virtual address mapped through page table to physical page number
  – Offset of virtual address same as physical address
  – Large page tables can be placed into virtual memory
• Multi-Level Tables
  – Virtual address mapped to series of tables
  – Permit sparse population of address space
• Inverted Page Table
  – Use of hash-table to hold translation entries
  – Size of page table ~ size of physical memory rather than size of virtual memory