Recall: Simple Paging

- Virtual Address:
  - Offset
  - PageTablePtr
  - PageTableSize
  - PageTableSize

- Physical Address:
  - Offset
  - Check Perm
  - Access Error

- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    » Permissions include: Valid bits, Read, Write, etc

- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    » Example: 10 bit offset ⇒ 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

Recall: Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit

- 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 page table entries!
  - Con: What if table really big?
    » Not all pages used all the time ⇒ would be nice to have working set of page table in memory

- How about multi-level paging or combining paging and segmentation?
Fix for sparse address space:
The two-level page table

Virtual Address: Physical Address:

Physical Page # Offset

10 bits 10 bits 12 bits

PageTablePtr

4 bytes (PTE)

4KB

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>L</th>
<th>D</th>
<th>A</th>
<th>C</th>
<th>S</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P: Present (same as “valid” bit in other architectures)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W: Writeable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U: User accessible</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PWT: Page write transparent: external cache write-through</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCD: Page cache disabled (page cannot be cached)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A: Accessed: page has been accessed recently</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D: Dirty (PTE only): page has been modified recently</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L: L=1→4MB page (directory only). Bottom 22 bits of virtual address serve as offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Summary: Two-Level Paging
### Summary: Two-Level Paging

<table>
<thead>
<tr>
<th>Virtual memory view</th>
<th>Stack (level 1)</th>
<th>Heap (level 2)</th>
<th>Physical memory view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>1000 0000</td>
<td>1000 0000</td>
<td>Stack</td>
</tr>
<tr>
<td>Heap</td>
<td>000 0000</td>
<td>000 0000</td>
<td>Heap</td>
</tr>
<tr>
<td>Code</td>
<td>000 0000</td>
<td>000 0000</td>
<td>Code</td>
</tr>
<tr>
<td>Data</td>
<td>000 0000</td>
<td>000 0000</td>
<td>Data</td>
</tr>
</tbody>
</table>

### 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:
    - Virtual Seg #
    - Virtual Page #
    - 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
  - Access: V,R
  - Physical Address: Physical Page #
  - Check Permissions
  - Access Error

### What about Sharing (Complete Segment)?

- Process A: Virtual Seg #
  - Base0 Limit0 V
  - Base1 Limit1 V
  - Base2 Limit2 V
  - Base3 Limit3 N
  - Base4 Limit4 V
  - Base5 Limit5 N
  - Base6 Limit6 N
  - Base7 Limit7 V

- Shared Segment

- Process B: Virtual Seg #
  - Base0 Limit0 V
  - Base1 Limit1 V
  - Base2 Limit2 V
  - Base3 Limit3 N
  - Base4 Limit4 V
  - Base5 Limit5 N
  - Base6 Limit6 N
  - Base7 Limit7 V

### 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, previous example keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    » Seems very expensive!
Administrivia

- Final Project 1 Report Due today!
- Project 2 Release on Thursday (3/14)
  - Happy PI day!
- Midterm 2: Thursday 4/4
  - Ok, this is a few weeks and after Spring Break
  - Will definitely include Scheduling material (lecture 10)
  - Up to and including some material from lecture 17
  - Probably try to have a Midterm review in early part of that week…. Stay tuned
- Note: A double-lock release always bad!
  - Solution to (3c) on exam doesn’t protect against a double release (i.e. release when lock is free).
    » Enhancement: Crash program (with error) if release on free lock
    » NOT: Fix it up so that a release on free lock ok!
  - Why? Because releasing without acquiring means messing up a different thread that has just acquired lock properly!

CS 162 Collaboration Policy

Explaining a concept to someone in another group
Discussing algorithms/testing strategies with other groups
Helping debug someone else’s code (in another group)
Searching online for generic algorithms (e.g., hash table)

Sharing code or test cases with another group
Copying OR reading another group’s code or test cases
Copying OR reading online code or test cases from from prior years

We compare all project submissions against prior year submissions and online solutions and will take actions (described on the course overview page) against offenders

X86 Segment Descriptors (32-bit Protected Mode)

- Segments are either implicit in the instruction (say for code segments) or actually 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:
    - A segment selector [13 bits]
    - G/L selects between GDT and LDT tables (global vs local descriptor tables)
    - Two registers: GDTR and LDTR hold pointers to the global and local descriptor tables in memory
      - Includes length of table (for < 2^13) entries
    - 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
      - S: System Segment (0: System, 1: code or data)
      - Type: Code, Data, Segment
Recall: 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
  - Limited to 64K segments
- Modern use in 32-bit Mode:
  - Segments “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:
      \[ \text{mov eax, gs(0x0)} \]
- 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 (for use in TLS)

X86_64: Four-level page table!

- 48-bit Virtual Address:
  - 9 bits
  - P1 index
  - 9 bits
  - P2 index
  - 9 bits
  - P3 index
  - 9 bits
  - P4 index
  - 12 bits
  - Offset

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

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

- 64bit Virtual Address:
  - 7 bits
  - P1 index
  - 9 bits
  - P2 index
  - 9 bits
  - P3 index
  - 9 bits
  - P4 index
  - 9 bits
  - P5 index
  - 12 bits
  - Offset

No!

Too slow
Too many almost-empty tables

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></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Segmentation</td>
<td>Fast context switching: Segment mapping maintained by CPU</td>
<td>External fragmentation</td>
</tr>
<tr>
<td>Paging (single-level page)</td>
<td>No external fragmentation, fast 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 easy allocation</td>
<td>Multiple memory references per page access</td>
</tr>
<tr>
<td>Two-level pages</td>
<td>Table size ~ # of pages in physical memory</td>
<td>Hash function more complex</td>
</tr>
<tr>
<td>Inverted Table</td>
<td>Table size ~ # of pages in physical memory</td>
<td>No cache locality of page table</td>
</tr>
</tbody>
</table>

### How is the Translation Accomplished?

- **What, exactly happens inside MMU?**
  - One possibility: **Hardware Tree Traversal**
    - For each virtual address traverses page table in hardware
    - Generates a "Page Fault" if it encounters invalid PTE
      - Fault handler will decide what to do
      - More on this next lecture
    - Pros: Relatively fast (but still many memory accesses!)
    - Cons: Inflexible, Complex hardware
  - Another possibility: **Software Tree Traversal**
    - Each traversal done in software (often invoked by "TLB Fault" more on this later)
    - Software may generate “Page Fault” later if PTE invalid
    - Pros: Very flexible
    - Cons: Every translation must invoke Fault!
  - In fact, need way to cache translations for either case!

### Recall: Dual-Mode Operation

- Can a process modify its own translation tables?
  - **NO!**
  - If it could, could get access to all of physical memory
  - Has to be restricted somehow
- 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
  - Newer processors have additional mode for hypervisor ("Ring -1")
- Certain operations restricted to Kernel mode:
  - Including modifying the page table (CR3 in x86), and GDT/LDT
  - Have to transition into Kernel mode before you can change them

### Recall: User→Kernel (System Call)

- Can’t let inmate (user) get out of padded cell on own
  - Would defeat purpose of protection!
  - So, how does the user program get back into kernel?
- System call: Voluntary procedure call into kernel
  - Hardware for controlled User→Kernel transition
  - Can any kernel routine be called?
    - **No!** Only specific ones.
  - System call ID encoded into system call instruction
    - Index forces well-defined interface with kernel
Recall: System Call Continued

• What are some system calls?
  – I/O: open, close, read, write, lseek
  – Files: delete, mkdir, rmdir, truncate, chown, chgrp, ..
  – Process: fork, exit, wait (like join)
  – Network: socket create, set options

• What happens at beginning of system call?
  – On entry to kernel, sets system to kernel mode
  – Handler address fetched from table/Handler started

• System call argument passing:
  – In registers (not very much can be passed)
  – Write into user memory, kernel copies into kernel mem
    » User addresses must be translated!
    » Kernel has different view of memory than user
  – Every argument must be explicitly checked!

Recall: User→Kernel
(Exceptions: Traps & Interrupts)

• A system call instruction causes a synchronous exception (or “trap”)
  – In fact, often called a software “trap” instruction
• Other sources of Synchronous Exceptions (“Trap”):
  – Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  – Segmentation Fault (address out of range)
  – Page Fault (for illusion of infinite-sized memory)

• Interrupts are Asynchronous Exceptions:
  – Examples: timer, disk ready, network, etc….
  – Interrupts can be disabled, traps cannot!

• On system call, exception, or interrupt:
  – Hardware enters kernel mode with interrupts disabled
  – Saves PC, then jumps to appropriate handler in kernel
  – Some processors (e.g. x86) also save registers, changes stack

  • Handler does any required state preservation not done by CPU:
    – Might save registers, other CPU state, and switches to kernel stack

How to get from Kernel→User

• What does the kernel do to create a new user process?
  – Allocate and initialize address-space control block
  – Read program off disk and store in memory
  – Allocate and initialize translation table
    » Point at code in memory so program can execute
    » Possibly point at statically initialized data
  – Run Program:
    » Set machine registers
    » Set hardware pointer to translation table (e.g. CR3 in x86)
    » Set processor status word for user mode
    » Jump to start of program

• How does kernel switch between processes?
  – Same saving/restoring of registers as before
  – Save/restore PSL (hardware pointer to translation table)

Closing thought:
Protection without Hardware

• Does protection require hardware support for translation and dual-mode behavior?
  – No: Normally use hardware, but anything you can do in hardware can also be done in software (possibly expensive)
• Protection via Strong Typing
  – Restrict programming language so that you can’t express program that would trash another program
  – Loader needs to make sure that program produced by valid compiler or all bets are off
  – Example languages: LISP, Ada, Modula-3 and Java
• Protection via software fault isolation:
  – Language independent approach: have compiler generate object code that provably can’t step out of bounds
    » Compiler puts in checks for every “dangerous” operation (loads, stores, etc). Again, need special loader.
    » Alternative, compiler generates “proof” that code cannot do certain things (Proof Carrying Code)
  – Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)
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 = (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Why Bother with Caching?

- 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
- 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”)
Memory Hierarchy of a Modern Computer System

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology

On-Chip
- Cache
- Registers
- Control
- Datapath

Second Level Cache (SRAM)
- Main Memory (DRAM)
- Secondary Storage (Disk)
- Tertiary Storage (Tape)

Speed (ns): 1s - 10s-100s - 100s - 10,000,000
Size (bytes): 100s - Ks-Ms - Ms-Gs - 10,000,000,000

A Summary on Sources of Cache Misses

- Compulsory (cold start or process migration, first reference): first access to a block
  - “Cold” fact of life: not a whole lot you can do about it
  - Note: If you are going to run “billions” of instruction, Compulsory Misses are insignificant

- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size

- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity

- Coherence (Invalidation): other process (e.g., I/O) updates memory

How is a Block found in a Cache?

- Block is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don’t have data select field

- Index Used to Lookup Candidates in Cache
  - Index identifies the set

- Tag used to identify actual copy
  - If no candidates match, then declare cache miss

Review: Direct Mapped Cache

- Direct Mapped $2^N$ byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = $2^M$)

- Example: 1 KB Direct Mapped Cache with 32 B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block

Valid Bit
Cache Data
Cache Index
Byte Select
Cache Tag

$0x50$
$0x01$
$0x00$
$0x00$

Byte 31
Byte 4
Byte 32
Byte 992
Byte 32
Byte 32
Review: Set Associative Cache

- **N-way set associative**: N entries per Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result

Example: Two-way set associative cache

- Cache Index selects a “set” from the cache
- Two tags in the set are compared to input in parallel
- Data is selected based on the tag result

Review: Fully Associative Cache

- **Fully Associative**: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block

Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache

32-Block Address Space:

<table>
<thead>
<tr>
<th>Block no.</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

- **Direct mapped**: block 12 can go only into block 4 (12 mod 8)
- **Set associative**: block 12 can go anywhere in set 0 (12 mod 4)
- **Fully associative**: block 12 can go anywhere

Which block should be replaced on a miss?

- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

Miss rates for a workload:

<table>
<thead>
<tr>
<th>Size</th>
<th>2-way LRU</th>
<th>4-way LRU</th>
<th>8-way LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>5.3%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.7%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>
### Review: What happens on a write?
- **Write through**: The information is written to both the block in the cache and to the block in the lower-level memory
- **Write back**: The information is written only to the block in the cache
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean or dirty?
- Pros and Cons of each?
  - **WT**:
    - **PRO**: read misses cannot result in writes
    - **CON**: Processor held up on writes unless writes buffered
  - **WB**:
    - **PRO**: repeated writes not sent to DRAM processor not held up on writes
    - **CON**: More complex
      Read miss may require writeback of dirty data

### Caching Applied to Address Translation

<table>
<thead>
<tr>
<th>CPU</th>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>Physical Memory</td>
<td></td>
</tr>
</tbody>
</table>

- **Translate (MMU)**
  - **Cached?**
    - Yes
      - **Save Result**
    - No
      - **Data Read or Write** (untranslated)

- **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

### What Actually Happens on a TLB Miss? (1/2)
- **Hardware traversed page tables**:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - If PTE valid, hardware fills TLB and processor never knows
    - If PTE marked as invalid, causes **Page Fault**, after which kernel decides what to do afterwards
- **Software traversed Page tables (like MIPS)**:
  - On TLB miss, processor receives TLB fault
    - Kernel traverses page table to find PTE
      - If PTE valid, fills TLB and returns from fault
      - If PTE marked as invalid, internally calls **Page Fault** handler
- **Most chip sets provide hardware traversal**
  - Modern operating systems tend to have more TLB faults since they use translation for many things
  - Examples:
    - shared segments
    - user-level portions of an operating system

### What happens on a Context Switch?
- **Need to do something, since TLBs map virtual addresses to physical addresses**
  - Address Space just changed, so TLB entries no longer valid!
- **Options?**
  - **Invalidate TLB**: simple but might be expensive
    - What if switching frequently between processes?
  - Include **ProcessID** in TLB
    - This is an architectural solution: needs hardware
- **What if translation tables change?**
  - For example, to move page from memory to disk or vice versa…
  - Must invalidate TLB entry!
    - Otherwise, might think that page is still in memory!
Summary (1/2)

- **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

Summary (2/2)

- **The Principle of Locality:**
  - Program likely to access a relatively small portion of the address space at any instant of time.
  - » **Temporal Locality**: Locality in Time
  - » **Spatial Locality**: Locality in Space
- **Three (+1) Major Categories of Cache Misses:**
  - **Compulsory Misses**: sad facts of life. Example: cold start misses.
  - **Conflict Misses**: increase cache size and/or associativity
  - **Capacity Misses**: increase cache size
  - **Coherence Misses**: Caused by external processors or I/O devices
- **Cache Organizations:**
  - **Direct Mapped**: single block per set
  - **Set associative**: more than one block per set
  - **Fully associative**: all entries equivalent