Recall: 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 2\textsuperscript{nd}-level table
  - Even when exist, 2\textsuperscript{nd}-level tables can reside on disk if not in use
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):
    
    | Field  | Description                        |
    |--------|------------------------------------|
    | 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 $\text{Max}(\text{CPL},\text{RPL}) \leq \text{DPL}$ |
    | S      | System Segment (0: System, 1: code or data) |
    | Type   | Code, Data, Segment                 |

Segment Register

Segment selector [13 bits]  \( G \)  RPL
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!

48-bit Virtual Address:

- Virtual P1 index
- Virtual P2 index
- Virtual P3 index
- Virtual P4 index

PageTablePtr

8 bytes

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

Physical Address:
(40-50 bits)

Physical Page #
12bit Offset
### 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>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>
<td></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

Total size of page table $\approx$ number of pages used by program in physical memory. Hash more complex
<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</td>
<td>Large table size (~ virtual memory)</td>
</tr>
<tr>
<td></td>
<td>Fast and easy allocation</td>
<td>Internal fragmentation</td>
</tr>
<tr>
<td>Paged Segmentation</td>
<td>Table size ~ # of pages in virtual memory</td>
<td>Multiple memory references per page access</td>
</tr>
<tr>
<td></td>
<td>Fast and easy allocation</td>
<td></td>
</tr>
<tr>
<td>Multi-Level Paging</td>
<td>Table size ~ # of pages in physical memory</td>
<td>Hash function more complex</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No cache locality of page table</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
- On every reference (I-fetch, Load, Store) read physical frame or FAULT
  - Through the caches to the memory
  - Then read/write the physical location

**Potentially reaaaaaaaly slow**
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 =
  \[ (\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time}) \]
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
Recall: In Machine Structures (eg. 61C) …

- 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 HitRate + MissRate = 1

HitRate = 90% \(\Rightarrow\) AMAT = \((0.9 \times 1) + (0.1 \times 101)\) = 11.1 ns

HitRate = 99% \(\Rightarrow\) AMAT = \((0.99 \times 1) + (0.01 \times 101)\) = 2.01 ns

\(\text{MissTime}_{L1}\) includes \(\text{HitTime}_{L1} + \text{MissPenalty}_{L1} \equiv \text{HitTime}_{L1} + \text{AMAT}_{L2}\)
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

![Diagram showing probability of reference and address space with temporal and spatial locality](image)
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
Caching Applied to Address Translation

- 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
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)
Recall: 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
Review: Set Associative Cache

- **N-way set associative**: N entries per Cache Index
  - N direct mapped caches operate 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

![Diagram of Cache Structure]

```
<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Cache Index</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>
```

**Valid Cache Tag**: Cache Block 0

**Compare**

**Sel1**

**Mux**

**OR**

**Hit**

**Cache Block**
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

![Cache Diagram]

- Cache Tag (27 bits long)
- Byte Select
- Cache Data
- Valid Bit
- Cache Tag

Ex: 0x01

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Valid Bit</th>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Byte 31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>** Byte 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 63</td>
</tr>
<tr>
<td></td>
<td></td>
<td>** Byte 33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 32</td>
</tr>
</tbody>
</table>

- Byte 0
- Byte 1
- Byte 32
- Byte 33
Where does a Block Get Placed in a Cache?

• Example: Block 12 placed in 8 block cache

32-Block Address Space:

- 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 Random</th>
<th>4-way LRU Random</th>
<th>8-way LRU Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>4.7%</td>
<td>4.4%</td>
</tr>
<tr>
<td></td>
<td>5.7%</td>
<td>5.3%</td>
<td>5.0%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>1.5%</td>
<td>1.4%</td>
</tr>
<tr>
<td></td>
<td>2.0%</td>
<td>1.7%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.13%</td>
<td>1.12%</td>
</tr>
<tr>
<td></td>
<td>1.17%</td>
<td>1.13%</td>
<td>1.12%</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
Physically-Indexed vs Virtually-Indexed Caches

- Physically-Indexed Caches
  - Address handed to cache *after translation*
  - Page Table holds *physical* addresses
  - Benefits:
    » Every piece of data has single place in cache
    » Cache can stay unchanged on context switch
  - Challenges:
    » TLB is in critical path of lookup!
- Pretty Common today (e.g. x86 processors)

- Virtually-Indexed Caches
  - Address handed to cache *before translation*
  - Page Table holds *virtual* addresses (one option)
  - Benefits:
    » TLB not in critical path of lookup, so can be faster
  - Challenges:
    » Same data could be mapped in multiple places of cache
    » May need to flush cache on context switch

- We will stick with Physically Addressed Caches for now!
What TLB Organization Makes Sense?

- Needs to be really fast
  - Critical path of memory access
    » In simplest view: before the cache
    » Thus, this adds to access time (reducing cache speed)
  - Seems to argue for Direct Mapped or Low Associativity

- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high! (PT traversal)
  - Cost of Conflict (Miss Time) is high
  - Hit Time – dictated by clock cycle

- Thrashing: continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    » First page of code, data, stack may map to same entry
    » Need 3-way associativity at least?
  - What if use high order bits as index?
    » TLB mostly unused for small programs
TLB organization: include protection

• How big does TLB actually have to be?
  – Usually small: 128-512 entries (larger now)
  – Not very big, can support higher associativity

• Small TLBs usually organized as fully-associative cache
  – Lookup is by Virtual Address
  – Returns Physical Address + other info

• What happens when fully-associative is too slow?
  – Put a small (4-16 entry) direct-mapped cache in front
  – Called a “TLB Slice”

• Example for MIPS R3000:

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access</th>
<th>ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFA00</td>
<td>0x0003</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>R/W</td>
<td>34</td>
</tr>
<tr>
<td>0x0040</td>
<td>0x0010</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>0x0041</td>
<td>0x0011</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>
Example: R3000 pipeline includes TLB “stages”

### MIPS R3000 Pipeline

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Dcd/ Reg</th>
<th>ALU / E.A</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>Operation</td>
<td>WB</td>
</tr>
<tr>
<td>E.A.</td>
<td>TLB</td>
<td>D-Cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TLB

- 64 entry, on-chip, fully associative, software TLB fault handler

### Virtual Address Space

<table>
<thead>
<tr>
<th>ASID</th>
<th>V. Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

- 0xx User segment (caching based on PT/TLB entry)
- 100 Kernel physical space, cached
- 101 Kernel physical space, uncached
- 11x Kernel virtual space

Allows context switching among
- 64 user processes without TLB flush
As described, TLB lookup is in serial with cache lookup
- Consequently, speed of TLB can impact speed of access to cache

Machines with TLBs go one step further: overlap TLB lookup with cache access
- Works because offset available early
- Offset in virtual address exactly covers the “cache index” and “byte select”
- Thus can select the cached byte(s) in parallel to perform address translation

Reducing translation time for physically-indexed caches
• Here is how this might work with a 4K cache:

  - What if cache size is increased to 8KB?
    – Overlap not complete
    – Need to do something else. See CS152/252

• Another option: Virtual Caches would make this faster
  – Tags in cache are virtual addresses
  – Translation only happens on cache misses
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!
  – Called “TLB Consistency”
• Aside: with Virtually-Indexed cache, need to flush cache!
  – Remember, everyone has their own version of the address “0”!
Putting Everything Together: Address Translation

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset
- Page TablePtr

Page Table (1st level)
- Page Table (2nd level)

Physical Address:
- Physical Page #
- Offset

Physical Memory:
Putting Everything Together: TLB

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (2nd level)

TLB:

Physical Address:
- Physical Page #
- Offset

Physical Memory:

Offset
Putting Everything Together: Cache

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Address:
- Physical Page #
- Offset

Physical Memory:

Virtual Address:
- PageTablePtr

Putting Everything Together: Cache

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Address:
- Physical Page #
- Offset

Physical Memory:

Virtual Address:
- PageTablePtr

Putting Everything Together: Cache

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Address:
- Physical Page #
- Offset

Physical Memory:

Virtual Address:
- PageTablePtr

Putting Everything Together: Cache

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Address:
- Physical Page #
- Offset

Physical Memory:

Virtual Address:
- PageTablePtr

Putting Everything Together: Cache

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Address:
- Physical Page #
- Offset

Physical Memory:

Virtual Address:
- PageTablePtr

Putting Everything Together: Cache

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Address:
- Physical Page #
- Offset

Physical Memory:

Virtual Address:
- PageTablePtr

Putting Everything Together: Cache

Virtual Address:
- Virtual P1 index
- Virtual P2 index
- Offset

Page Table (1st level)

Page Table (2nd level)

TLB:

Physical Address:
- Physical Page #
- Offset

Physical Memory:

Virtual Address:
- PageTablePtr
Page Fault

• The Virtual-to-Physical Translation fails
  – PTE marked invalid, Priv. Level Violation, Access violation, or does not exist
  – Causes an Fault / Trap
    » Not an interrupt because synchronous to instruction execution
  – May occur on instruction fetch or data access
  – Protection violations typically terminate the instruction

• Other Page Faults engage operating system to fix the situation and retry the instruction
  – Allocate an additional stack page, or
  – Make the page accessible - Copy on Write,
  – Bring page in from secondary storage to memory – demand paging

• Fundamental inversion of the hardware / software boundary
**Demand Paging**

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don’t use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user’s code to be in memory
- Solution: use main memory as “cache” for disk

![Diagram of Demand Paging](image-url)
Page Fault $\Rightarrow$ Demand Paging

Process $\rightarrow$ virtual address $\rightarrow$ MMU $\rightarrow$ physical address $\rightarrow$ PT $\rightarrow$ frame# $\rightarrow$ load page from disk

- retry
- exception
- page fault
- update PT entry
- scheduler
- Page Fault Handler
- Operating System

Load page from disk

MMU

PT

frame#

offset

physical address

page#

offset

frame#

Process

virtual address

instruction

exception

Page Fault Handler

scheduler
Demand Paging as Caching, …

• What “block size”? - 1 page (e.g, 4 KB)
• What “organization” ie. direct-mapped, set-assoc., fully-associative?
  – Fully associative since arbitrary virtual → physical mapping
• How do we locate a page?
  – First check TLB, then page-table traversal
• What is page replacement policy? (i.e. LRU, Random…)
  – This requires more explanation… (kinda LRU)
• What happens on a miss?
  – Go to lower level to fill miss (i.e. disk)
• What happens on a write? (write-through, write back)
  – Definitely write-back – need dirty bit!
Illusion of Infinite Memory

- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    » More programs fit into memory, allowing more concurrency
- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data
    » Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    » Performance issue, not correctness issue
Review: What is in a 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:
  - 2-level page table (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>P</th>
<th>D</th>
<th>A</th>
<th>PCD</th>
<th>PWT</th>
<th>U</th>
<th>W</th>
<th>P</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
Demand Paging Mechanisms

- PTE makes demand paging implementable
  - Valid $\implies$ Page in memory, PTE points at physical page
  - Not Valid $\implies$ Page not in memory; use info in PTE to find it on disk when necessary

- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    » Resulting trap is a “Page Fault”

- What does OS do on a Page Fault?:
  » Choose an old page to replace
  » If old page modified (“D=1”), write contents back to disk
  » Change its PTE and any cached TLB to be invalid
  » Load new page into memory from disk
  » Update page table entry, invalidate TLB for new entry
  » Continue thread from original faulting location

- TLB for new page will be loaded when thread continued!
- While pulling pages off disk for one process, OS runs another process from ready queue
  » Suspended process sits on wait queue
Origins of Paging

Disks provide most of the storage

Keep most of the address space on disk

Relatively small memory, for many processes

Actively swap pages to/from

Keep memory full of the frequently accesses pages

Many clients on dumb terminals running different programs
Very Different Situation Today

Powerful system
Huge memory
Huge disk
Single user
A Picture on one machine

- Memory stays about 75% used, 25% for dynamics
- A lot of it is shared 1.9 GB
Many Uses of Virtual Memory and “Demand Paging” …

• Extend the stack
  – Allocate a page and zero it
• Extend the heap (sbrk of old, today mmap)
• Process Fork
  – Create a copy of the page table
  – Entries refer to parent pages – NO-WRITE
  – Shared read-only pages remain shared
  – Copy page on write
• Exec
  – Only bring in parts of the binary in active use
  – Do this on demand
• MMAP to explicitly share region (or to access a file as RAM)
Classic: Loading an executable into memory

- .exe
  - lives on disk in the file system
  - contains contents of code & data segments, relocation entries and symbols
  - OS loads it into memory, initializes registers (and initial stack pointer)
  - program sets up stack and heap upon initialization:
    - 
    - `crt0` (C runtime init)
Create Virtual Address Space of the Process

- Utilized pages in the VAS are backed by a page block on disk
  - Called the backing store or swap file
  - Typically in an optimized block store, but can think of it like a file
Create Virtual Address Space of the Process

- User Page table maps entire VAS
- All the utilized regions are backed on disk
  - swapped into and out of memory as needed
- For every process
• User Page table maps entire VAS
  – Resident pages to the frame in memory they occupy
  – The portion of it that the HW needs to access must be resident in memory
Provide Backing Store for VAS

- User Page table maps entire VAS
- Resident pages mapped to memory frames
- For all other pages, OS must record where to find them on disk
• **FindBlock**(PID, page#) $\rightarrow$ disk_block
  – Some OSs utilize spare space in PTE for paged blocks
  – Like the PT, but purely software

• Where to store it?
  – In memory – can be compact representation if swap storage is contiguous on disk
  – Could use hash table (like Inverted PT)

• Usually want backing store for resident pages too

• May map code segment directly to on-disk image
  – Saves a copy of code to swap file

• May share code segment with multiple instances of the program
Provide Backing Store for VAS
On page Fault ...

disk (huge, TB)

stack

heap

data

code

vas 1

kernel

stack

heap

data

code

vas 2

kernel

stack

heap

data

code

pt 1

memory

user

page frames

user

pagetable

kernel

code
& data

active process & PT
On page Fault … find & start load

disk (huge, TB)

stack

heap

data

code

stack

heap

data

code

VAS 1

kernel

stack

heap

data

code

VAS 2

kernel

stack

heap

data

code

PT 1

memory

user

page frames

user

pagetable

kernel

code & data

active process & PT
On page Fault ... schedule other P or T
On page Fault … update PTE
Eventually reschedule faulting thread
Summary: Steps in Handling a Page Fault

1. Trap on page reference.
2. Bring in missing page from backing store.
3. Set page status to on backing store.
4. Update physical memory with the brought-in page.
5. Reset page table.
6. Restart instruction execution.
Summary (1/2)

• 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
Summary (2/2)

• “Translation Lookaside Buffer” (TLB)
  – Small number of PTEs and optional process IDs (< 512)
  – Often Fully Associative (Since conflict misses expensive)
  – On TLB miss, page table must be traversed and if located PTE is invalid, cause Page Fault
  – On change in page table, TLB entries must be invalidated

• Demand Paging: Treating the DRAM as a cache on disk
  – Page table tracks which pages are in memory
  – Any attempt to access a page that is not in memory generates a page fault, which causes OS to bring missing page into memory