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 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

Recall: 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

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 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

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:**

- 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% 5.7% 4.7% 5.3% 4.4% 5.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9% 2.0% 1.5% 1.7% 1.4% 1.5%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15% 1.17% 1.13% 1.13% 1.12% 1.12%</td>
<td></td>
<td></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!**
**Administrivia**

• **Midterm 2:** Coming up on Thursday 10/29  
  - Topics: up until Lecture 17: Scheduling, Deadlock, Address Translation, Virtual Memory, Caching, TLBs, Demand Paging, I/O  
  - Will REQUIRE you to have your zoom proctoring setup working  
    » You must have screen sharing, audio, and your camera working  
    » Make sure to get your setup debugged and ready!  
• **Review Session:** 10/27  
  - Details TBA  
• **Kubi Office Hours:** M/W 2:00-3:00  
  - Let me know if this doesn’t work…  
• **US Election coming up:** Don’t forget to Vote!  
  - Voting is one of the most important things you can do if you are allowed  
  - Don’t miss the opportunity!  
  - Be safe, of course

---

**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>E.A.</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
Reducing translation time for physically-indexed caches

- 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

---

Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:

  ![Diagram of TLB and 4K cache overlap](image)

- 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

---

Current Intel x86 (Skylake, Cascade Lake)

- Caches (all 64 B line size)
  - L1 I-Cache: 32 KiB/core, 8-way set assoc.
  - L1 D Cache: 32 KiB/core, 8-way set assoc., 4-5 cycles load-to-use, Write-back policy
  - L2 Cache: 1 MiB/core, 16-way set assoc., Inclusive, Write-back policy, 14 cycles latency
  - L3 Cache: 1.375 MiB/core, 11-way set assoc., shared across cores, Non-inclusive victim cache, Write-back policy, 50-70 cycles latency

- TLB
  - L1 ITLB, 128 entries; 8-way set assoc. for 4 KB pages
    - 8 entries per thread; fully associative, for 2 MiB / 4 MiB page
  - L1 DTLB 64 entries; 4-way set associative for 4 KB pages
    - 32 entries; 4-way set associative, 2 MiB / 4 MiB page translations:
    - 4 entries; 4-way associative, 1G page translations:
  - L2 STLB: 1536 entries; 12-way set assoc. 4 KiB + 2 MiB pages
    - 16 entries; 4-way set associative, 1 GiB page translations:
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"!
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

Demand Paging as Caching, …

- What “block size”? - 1 page (e.g, 4 KB)
- What “organization” ie. direct-mapped, set-associ., 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

Physical Memory: 512 MB
Disk: 500 GB
Virtual Memory: 4 GB

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 tabler (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

- PTE makes demand paging implementable
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ 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

Demand Paging Mechanisms

Origins of Paging

- Disks provide most of the storage
  - Relatively small memory, for many processes
  - Many clients on dumb terminals running different programs
  - Actively swap pages to/from
  - Keep memory full of the frequently accesses pages
  - Keep most of the address space on disk
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 (sbrc 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

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

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
What Data Structure Maps Non-Resident Pages to Disk?

- **FindBlock(PID, page#) → 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 …

On page Fault … find & start load
On page Fault … schedule other P or T

On page Fault … update PTE

Eventually reschedule faulting thread

Summary: Steps in Handling a Page Fault
Some questions we need to answer!

- During a page fault, where does the OS get a free frame?
  - Keeps a free list
  - Unix runs a “reaper” if memory gets too full
    » Schedule dirty pages to be written back on disk
    » Zero (clean) pages which haven’t been accessed in a while
  - As a last resort, evict a dirty page first

- How can we organize these mechanisms?
  - Work on the replacement policy

- How many page frames/process?
  - Like thread scheduling, need to “schedule” memory resources:
    » Utilization? fairness? priority?
  - Allocation of disk paging bandwidth

Working Set Model

- As a program executes it transitions through a sequence of “working sets” consisting of varying sized subsets of the address space

Cache Behavior under WS model

- Amortized by fraction of time the Working Set is active
- Transitions from one WS to the next
- Capacity, Conflict, Compulsory misses
- Applicable to memory caches and pages. Others?

Another model of Locality: Zipf

\[ P(\text{access}(\text{rank})) = \frac{1}{\text{rank}} \]

- Likelihood of accessing item of rank \( r \) is \( \alpha \frac{1}{r^a} \)
- Although rare to access items below the top few, there are so many that it yields a “heavy tailed” distribution
- Substantial value from even a tiny cache
- Substantial misses from even a very large cache
Demand Paging Cost Model

- Since Demand Paging like caching, can compute average access time ("Effective Access Time")
  - EAT = Hit Rate x Hit Time + Miss Rate x Miss Time
  - EAT = Hit Time + Miss Rate x Miss Penalty
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose p = Probability of miss, 1-p = Probability of hit
  - Then, we can compute EAT as follows:
    \[ EAT = 200\text{ns} + p \times 8\text{ ms} \]
    \[ = 200\text{ns} + p \times 8,000,000\text{ns} \]
- If one access out of 1,000 causes a page fault, then EAT = 8.2 μs:
  - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
  - EAT < 200ns x 1.1 ⇒ p < 2.5 x 10^{-6}
  - This is about 1 page fault in 400,000!

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - The cost of being wrong is high: must go to disk
    - Must keep important pages in memory, not toss them out
- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad – throws out heavily used pages instead of infrequently used
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable – makes it hard to make real-time guarantees
- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great (provably optimal), but can't really know future…
  - But past is a good predictor of the future …

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
- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past