Recall: In Machine Structures (eg. 61C) …

- Caching is the key to memory system performance

![Diagram of memory system with processor, main memory (DRAM), second level cache (SRAM), and access times](image)

- Average Access time = (Hit Rate \times HitTime) + (Miss Rate \times MissTime)

- HitRate + MissRate = 1
- HitRate = 90\% \Rightarrow Average Access Time = 19\,\text{ns}
- HitRate = 99\% \Rightarrow Average Access Time = 10.9\,\text{ns}

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: What Actually Happens on a TLB Miss?

- 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
Transparent Exceptions: TLB/Page fault

- How to transparently restart faulting instructions?
  - (Consider load or store that gets TLB or Page fault)
  - Could we just skip faulting instruction?
    - No: need to perform load or store after reconnecting physical page
- Hardware must help out by saving:
  - Faulting instruction and partial state
    - Need to know which instruction caused fault
    - Is single PC sufficient to identify faulting position????
  - Processor State: sufficient to restart user thread
    - Save/restore registers, stack, etc
- What if an instruction has side-effects?

Consider weird things that can happen

- What if an instruction has side effects?
  - Options:
    - Unwind side-effects (easy to restart)
    - Finish off side-effects (messy!)
  - Example 1: mov (sp) +, 10
    - What if page fault occurs when write to stack pointer?
      - Did sp get incremented before or after the page fault?
  - Example 2: strcpy (r1), (r2)
    - Source and destination overlap: can't unwind in principle!
      - IBM S/370 and VAX solution: execute twice – once read-only
- What about "RISC" processors?
  - For instance delayed branches?
    - Example: bne somewhere
    - ld r1, (sp)
    - Precise exception state consists of two PCs: PC and nPC
  - Delayed exceptions:
    - Example: div r1, r2, r3
    - ld r1, (sp)
    - What if takes many cycles to discover divide by zero, but load has already caused page fault?

Precise Exceptions

- Precise ⇒ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - MIPS takes this position
- Imprecise ⇒ system software has to figure out what is where and put it all back together
- Performance goals often lead designers to forsake precise interrupts
  - system software developers, user, markets etc. usually wish they had not done this
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Recall: TLB Organization

- 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!
  - This argues that cost of Conflict (Miss Time) is much higher than slightly increased cost of access (Hit Time)
- 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
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

Reducing translation time further

- As described, TLB lookup is in serial with cache lookup:

```
Virtual Address

V page no.  offset
```

```
TLB Lookup

V Access Rights PA
```

```
Physical Address

P page no.  offset
```

- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early

Overlapping TLB & Cache Access

- Main idea:
  - 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

  virtual address: Virtual Page # Offset

  physical address: Tag / page # index byte

Overlapping TLB & Cache Access (1/2)

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

```
TLB
```

```
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
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

Overlapping TLB & Cache Access (2/2)
Putting Everything Together: Address Translation

Physical Address: Offset

Virtual Address: Offset

Page Table (1st level)

Page Table (2nd level)

Page TablePtr

Physical Memory:

Putting Everything Together: TLB

Physical Address: Offset

Page Table (1st level)

Page Table (2nd level)

Page TablePtr

Physical Memory:

Putting Everything Together: Cache

Virtual Address: Offset

Physical Address: Offset

Page Table (1st level)

Page Table (2nd level)

Page TablePtr

Physical Memory:

Next Up: What happens when ...

virtual address

physical address

MMU

Page Fault Handler:

update PT entry

load page from disk

scheduler
**Administrivia**

- Midterm #1 grades posted to gradescope and autograder

![Midterm #1 grades distribution graph]

- Midterm #1 is 11% of total course grade
  - If your grade is >2 std devs below mean, talk with prof or staff

- Project 2 has been released
  - Get started early as design doc is due Monday 3/28

**Where are all places that caching arises in OSes?**

- Direct use of caching techniques
  - paged virtual memory (mem as cache for disk)
  - TLB (cache of PTEs)
  - file systems (cache disk blocks in memory)
  - DNS (cache hostname => IP address translations)
  - Web proxies (cache recently accessed pages)

- Which pages to keep in memory?
  - All-important “Policy” aspect of virtual memory
  - Will spend a bit more time on this in a moment

**Impact of caches on Operating Systems**

- Indirect - dealing with cache effects (e.g., sync state across levels)
- Process scheduling
  - which and how many processes are active?
  - large memory footprints versus small ones?
  - priorities?
  - Shared pages mapped into VAS of multiple processes?
- Impact of thread scheduling on cache performance
  - rapid interleaving of threads (small quantum) may degrade cache performance
  - increase average memory access time (AMAT) !!!
- Designing operating system data structures for cache performance
- Maintaining the correctness of various caches
  - TLB consistency:
    - With PT across context switches?
    - Across updates to the PT?
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

- 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 one.

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.
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)
  – Support 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

Recall: What is in a Page Table Entry

• 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>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11-12</th>
</tr>
</thead>
<tbody>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Demand Paging is Caching

• Since Demand Paging is Caching, must ask:
  – What is block size?
    » 1 page
  – What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
    » Fully associative: arbitrary virtual→physical mapping
  – How do we find a page in the cache when look for it?
    » 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!

Demand Paging Mechanisms

• PTE helps us implement demand paging
  – 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
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

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
Create Virtual Address Space of the 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

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
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

- Eventually reschedule faulting thread
- disk (huge, TB)
- stack
- heap
- data
- code
- user
- page frames
- user pagetable
- kernel code & data
- active process & PT

Summary: Steps in Handling a Page Fault

1. Trap
2. Reset page table
3. Bring in missing page
4. Physical memory
5. Free frame
6. Page table
7. Loading system
8. Reference

Some following questions

- 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
  - 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
Summary

- A cache of translations called a "Translation Lookaside Buffer" (TLB)
  - Relatively small number of entries (< 512)
  - Fully Associative (Since conflict misses expensive)
  - TLB entries contain PTE and optional process ID

- On TLB miss, page table must be traversed
  - If located PTE is invalid, cause Page Fault

- On context switch/change in page table
  - TLB entries must be invalidated somehow

- TLB is logically in front of cache
  - Thus, needs to be overlapped with cache access to be really fast

- Precise Exception specifies a single instruction for which:
  - All previous instructions have completed (committed state)
  - No following instructions nor actual instruction have started