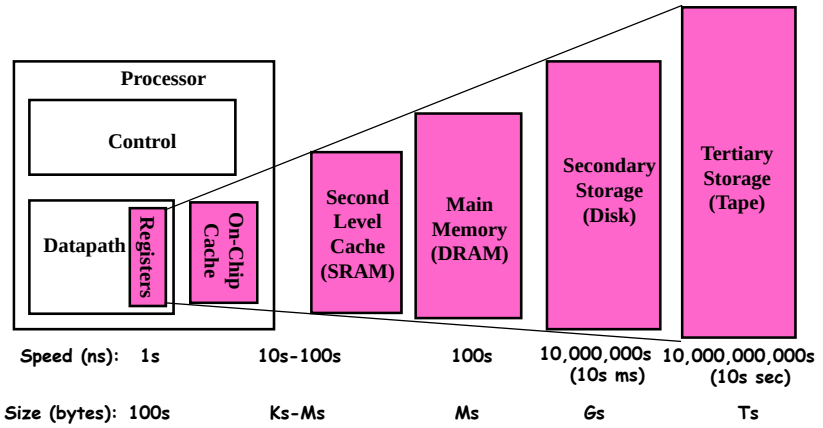
CS162
Operating Systems and Systems Programming
Lecture 14

Caching and Demand Paging

October 20, 2010
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: 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

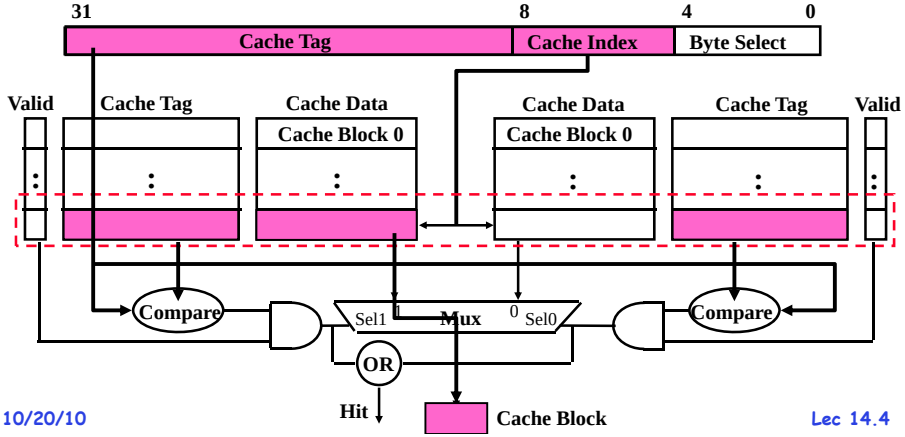


Review: A Summary on Sources of Cache Misses

- Compulsory (cold start): first reference to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: When running "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 same cache location
  - Solutions: increase cache size, or increase associativity
- Two others:
  - Coherence (Invalidation): other process (e.g., I/O) updates memory
  - Policy: Due to non-optimal replacement policy

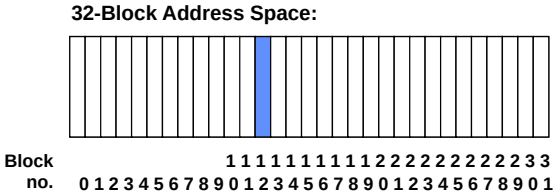
Review: Set Associative Cache

- 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

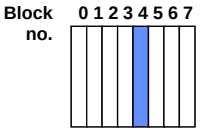


Review: Where does a Block Get Placed in a Cache?

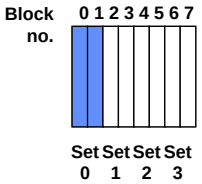
- Example: Block 12 placed in 8 block cache



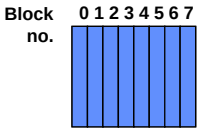
Direct mapped: block 12 can go only into block 4 (12 mod 8)



Set associative: block 12 can go anywhere in set 0 (12 mod 4)



Fully associative: block 12 can go anywhere



Goals for Today

- Finish discussion of Caching/TLBs
- Concept of Paging to Disk
- Page Faults and TLB Faults
- Precise Interrupts
- Page Replacement Policies

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

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)

	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

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

Review: Cache performance

- Miss-oriented Approach to Memory Access:

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty \right) \times CycleTime$$

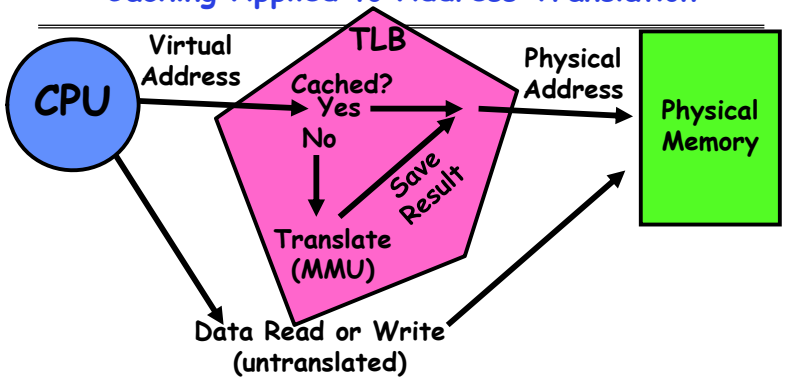
- Separating out Memory component entirely
  - AMAT = Average Memory Access Time

$$\begin{aligned} CPUtime &= IC \times \left(CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime \\ AMAT &= HitRate \times HitTime + MissRate \times MissTime \\ &= HitTime + MissRate \times MissPenalty \\ &= Frac_{Inst} \times (HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}) + \\ &\quad Frac_{Data} \times (HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data}) \end{aligned}$$

- AMAT for Second-Level Cache

$$\begin{aligned} AMAT_{1st} &= HitTime_{1st} + MissRate_{1st} \times MissPenalty_{1st} \\ &= HitTime_{1st} + MissRate_{1st} \times AMAT_{2nd} \\ &= HitTime_{1st} + MissRate_{1st} \times (HitTime_{2st} + MissRate_{2st} \times MissPenalty_{2st}) \end{aligned}$$

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

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

What happens on a Context Switch?

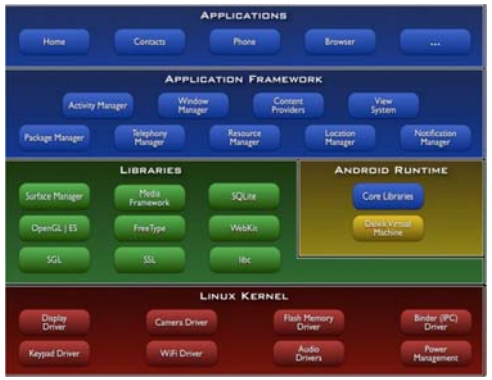
- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate TLB: simple but might be expensive
    - » What if switching frequently between processes?
  - Include ProcessID in TLB
    - » This 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!

Administrative

- Exam not graded yet
  - Will get solutions up early next week at latest
  - Will talk about Problem 3 in section on Friday
- Project 1 reports almost graded
  - Hoping to get them to you soon
- Project 2
  - Code due Tuesday (10/26)
  - Look at the lecture schedule to keep up with due dates!

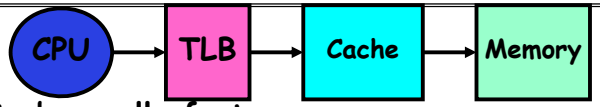
Mobile devices are the future

- Android is the popular operating system from Google
  - For Mobile devices
    - » Phones
    - » Ebook Readers (i.e. B&N)
    - » Tablets
  - Linux version 2.6.x
  - Java virtual machine and runtime system
  - Lots of media extensions
    - » WebKit for browsing
    - » Media Libraries
    - » Cellular Networking



- Mobile Systems are the hottest new software stack
  - Ubiquitous Computing
  - Worldwide, more than 1 billion new cell phones purchased/year for last few years
    - » Compare: worldwide number PCs purchased/year ~ 250M

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!
  - This argues that cost of Conflict (Miss Penalty) 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

TLB organization: include protection

- How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity
- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info
- Example for MIPS R3000:

Virtual Address	Physical Address	Dirty	Ref	Valid	Access	ASID
0xFA00	0x0003	Y	N	Y	R/W	34
0x0040	0x0010	N	Y	Y	R	0
0x0041	0x0011	N	Y	Y	R	0

- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a "TLB Slice"
- When does TLB lookup occur?
  - Before cache lookup?
  - In parallel with cache lookup?

Example: R3000 pipeline includes TLB "stages"

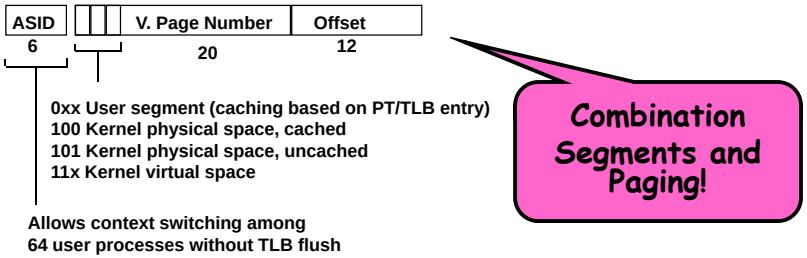
MIPS R3000 Pipeline

Inst Fetch	Dcd/ Reg	ALU / E.A	Memory	Write Reg
TLB	I-Cache	RF	Operation	WB
		E.A.	TLB	D-Cache

TLB

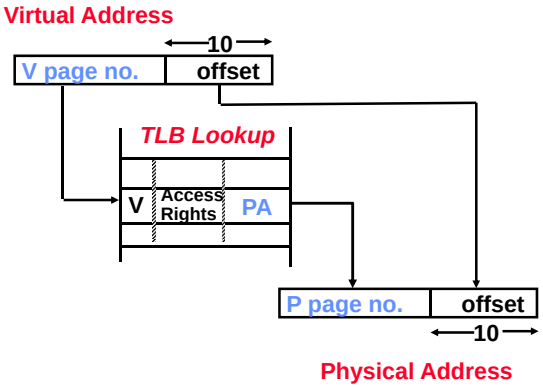
64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space



Reducing translation time further

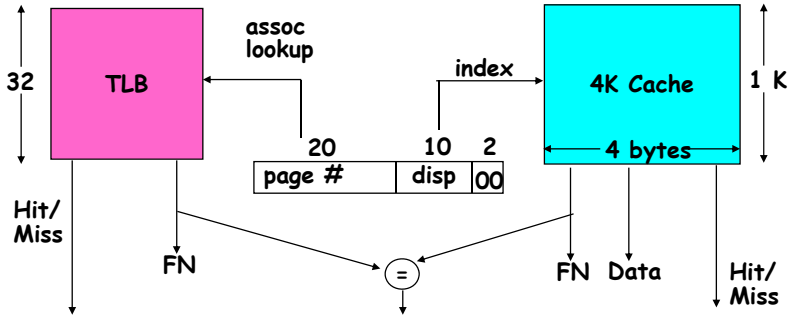
- As described, TLB lookup is in serial with cache lookup:



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

Overlapping TLB & Cache Access

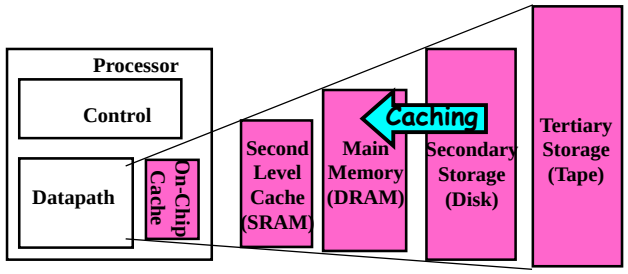
- Here is how this might work with a 4K cache:



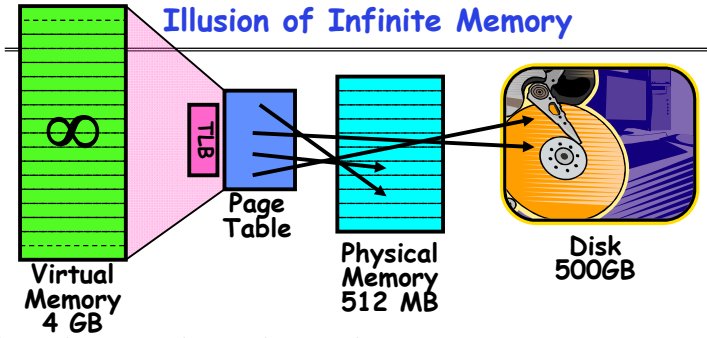
- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252
- Another option: Virtual Caches
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

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



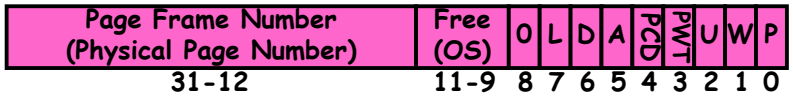
- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - » More programs fit into memory, allowing more concurrency
- Principle: Transparent Level of Indirection (page table)
  - 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

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!

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:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"



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
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

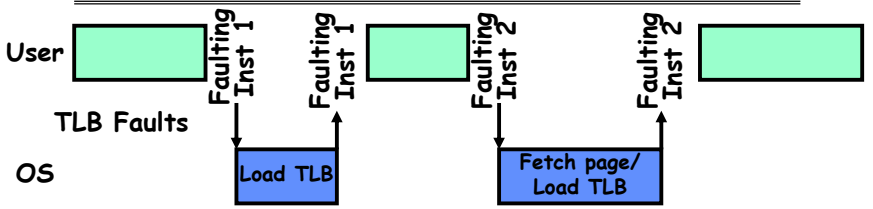
Demand Paging Mechanisms

- 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

Software-Loaded TLB

- MIPS/Nachos TLB is loaded by software
  - High TLB hit rate⇒ok to trap to software to fill the TLB, even if slower
  - Simpler hardware and added flexibility: software can maintain translation tables in whatever convenient format
- How can a process run without access to page table?
  - Fast path (TLB hit with valid=1):
    - » Translation to physical page done by hardware
  - Slow path (TLB hit with valid=0 or TLB miss)
    - » Hardware receives a "TLB Fault"
  - What does OS do on a TLB Fault?
    - » Traverse page table to find appropriate PTE
    - » If valid=1, load page table entry into TLB, continue thread
    - » If valid=0, perform "Page Fault" detailed previously
    - » Continue thread
- Everything is transparent to the user process:
  - It doesn't know about paging to/from disk
  - It doesn't even know about software TLB handling

Transparent Exceptions



- How to transparently restart faulting instructions?
  - Could we just skip it?
    - » 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?

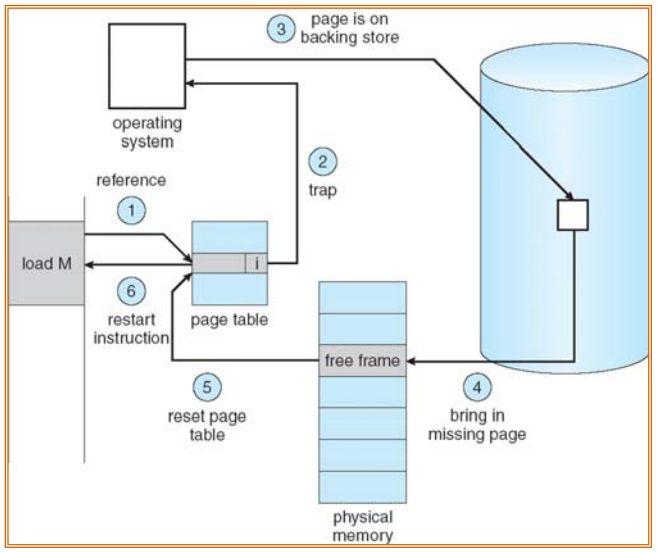
Consider weird things that can happen

- 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

Steps in Handling a Page Fault



Demand Paging Example

- 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 = Probably of hit
  - Then, we can compute EAT as follows:

    EAT = 200ns + p x 8 ms
    = 200ns + p x 8,000,000ns
- 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%?
  - 200ns x 1.1 < EAT ⇒ p < 2.5 x 10^{-6}
  - This is about 1 page fault in 400000!

What Factors Lead to Misses?

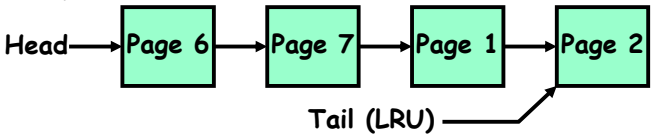
- Compulsory Misses:
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    - » Prefetching: loading them into memory before needed
    - » Need to predict future somehow! More later.
- Capacity Misses:
  - Not enough memory. Must somehow increase size.
  - Can we do this?
    - » One option: Increase amount of DRAM (not quick fix!)
    - » Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- Conflict Misses:
  - Technically, conflict misses don't exist in virtual memory, since it is a "fully-associative" cache
- Policy Misses:
  - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
  - How to fix? Better replacement policy

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, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable – makes it hard to make real-time guarantees

Replacement Policies (Con't)

- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- How to implement LRU? Use a list!



  - On each use, remove page from list and place at head
  - LRU page is at tail
- Problems with this scheme for paging?
  - Need to know immediately when each page used so that can change position in list…
  - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

  - FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away

Example: MIN

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

  - MIN: 5 faults
  - Where will D be brought in? Look for page not referenced farthest in future.
- What will LRU do?
  - Same decisions as MIN here, but won't always be true!

When will LRU perform badly?

- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

  - Every reference is a page fault!
- MIN Does much better:

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

Summary

- TLB is cache on translations
  - Fully associative to reduce conflicts
  - Can be overlapped with cache access
- Demand Paging:
  - Treat memory as cache on disk
  - Cache miss ⇒ get page from disk
- Transparent Level of Indirection
  - User program is unaware of activities of OS behind scenes
  - Data can be moved without affecting application correctness
- Software-loaded TLB
  - Fast Path: handled in hardware (TLB hit with valid=1)
  - Slow Path: Trap to software to scan page table
- Precise Exception specifies a single instruction for which:
  - All previous instructions have completed (committed state)
  - No following instructions nor actual instruction have started
- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: replace page that will be used farthest in future
  - LRU: Replace page that hasn't be used for the longest time