



Caching and Demand Paging

<http://inst.eecs.berkeley.edu/~cs162>

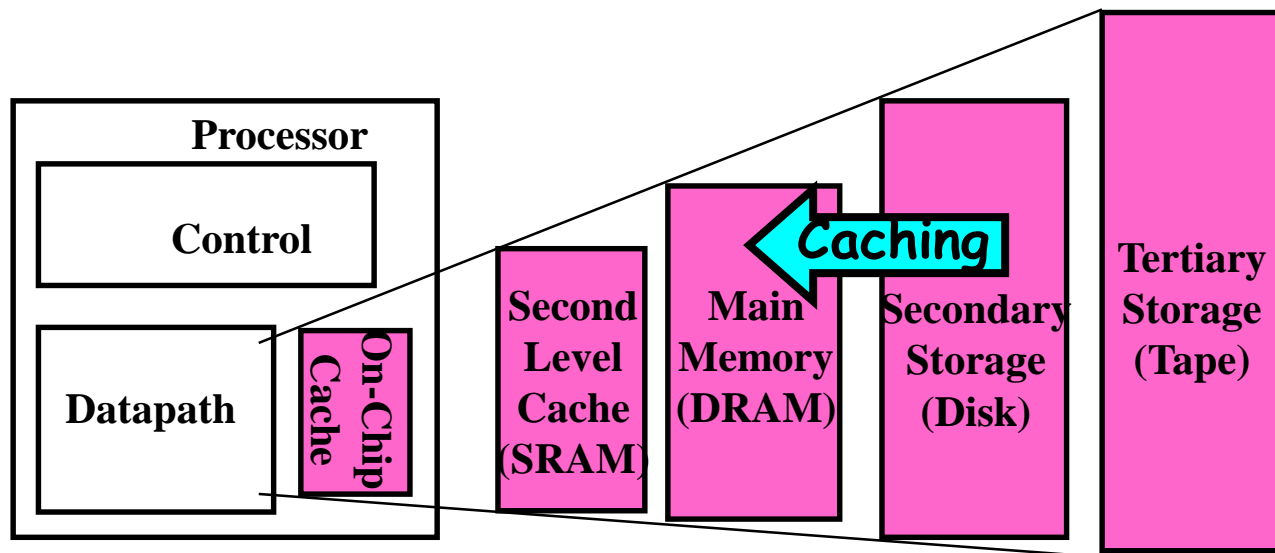
Goals for Today

- Concept of Paging to Disk
- Page Faults and TLB Faults
- Precise Interrupts
- Page Replacement Policies

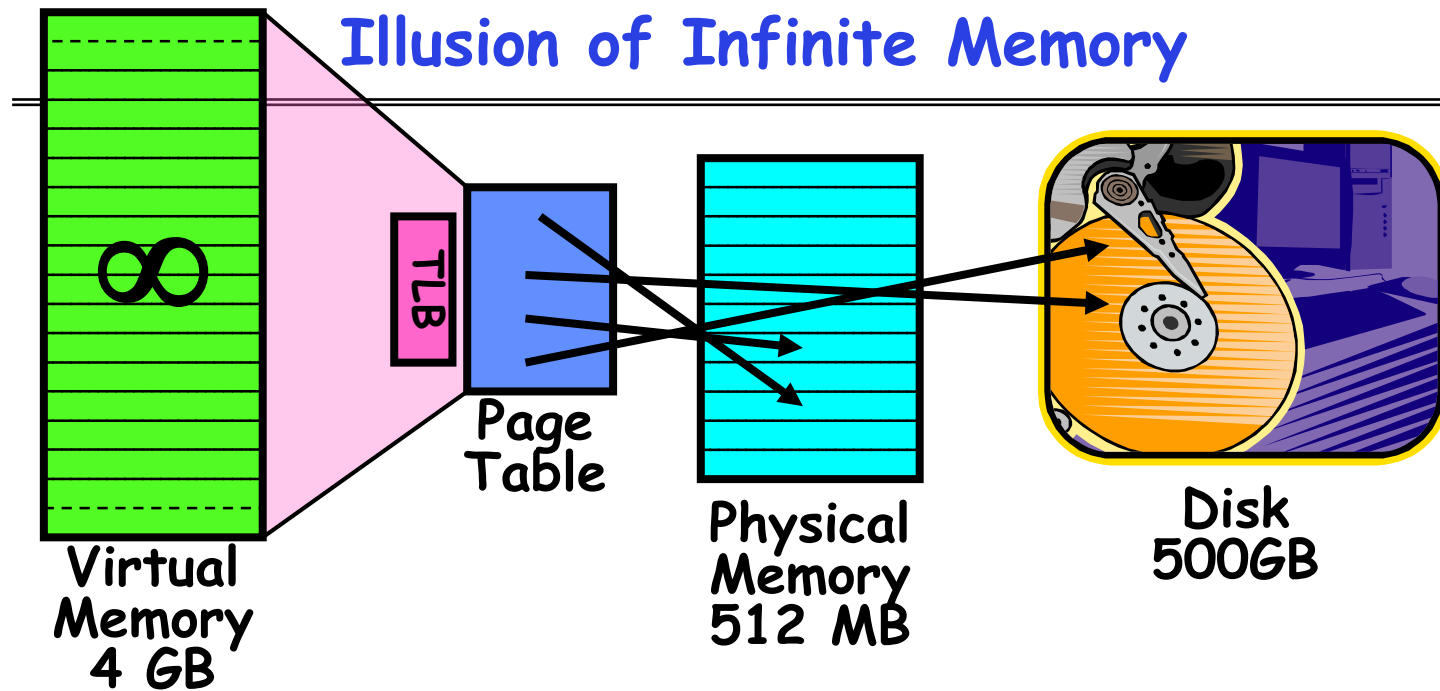
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatoiwicz.

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 \Rightarrow
 - 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"

Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

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 \Rightarrow 4MB page (directory only).

Bottom 22 bits of virtual address serve as offset

Demand Paging Mechanisms

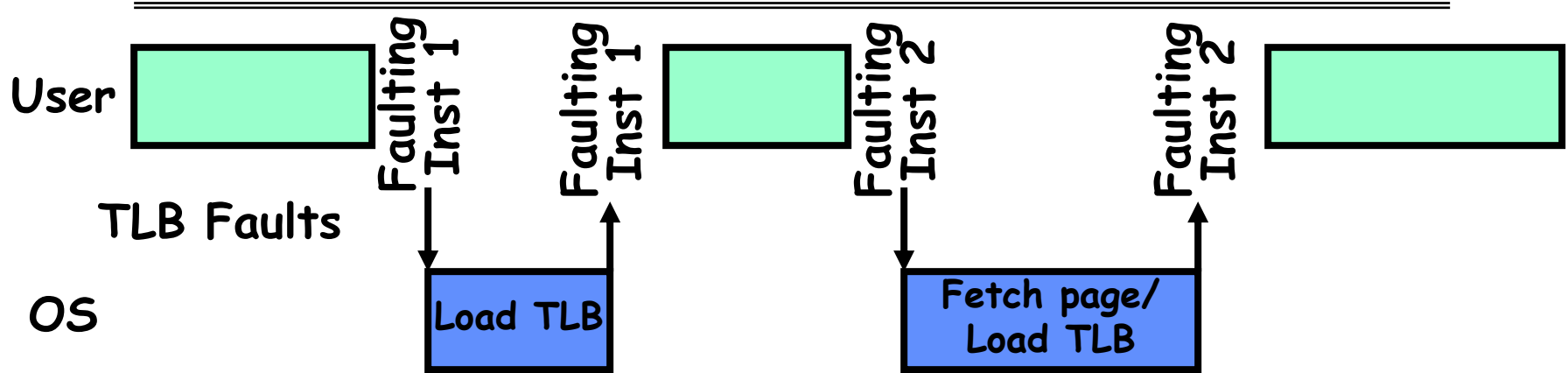
- PTE helps us implement demand paging
 - Valid \Rightarrow Page in memory, PTE points at physical page
 - Not Valid \Rightarrow Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
 - Memory Management Unit (MMU) traps to OS
 - » Resulting 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

cache

Software-Loaded TLB

- MIPS/Nachos TLB is loaded by software
 - High TLB hit rate \Rightarrow 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

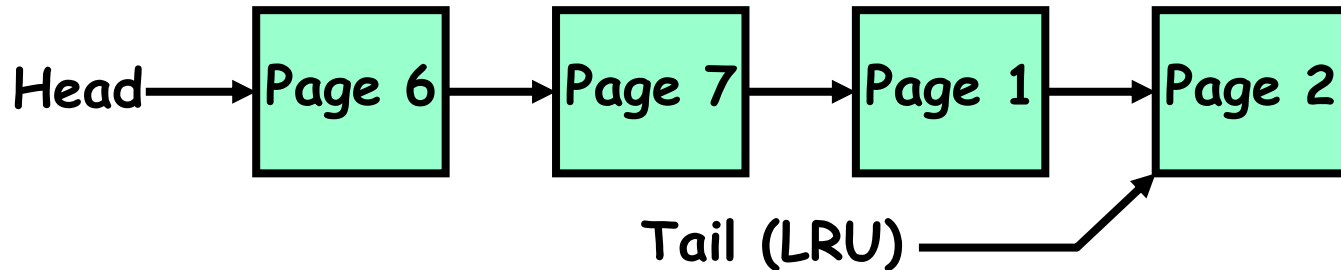
- Precise \Rightarrow state of the machine is preserved as if program executed up to the offending instruction
 - All previous instructions **completed**
 - Offending instruction and all following instructions act **as if they have not even started**
 - Same system code will work on different implementations
 - Difficult in the presence of pipelining, out-of-order execution, ...
 - **MIPS takes this position**
- Imprecise \Rightarrow 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**

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
- What about MIN?
 - Replace page that won't be used for the longest time
 - Great, but can't really know future...
 - Makes good comparison case, however
- What about RANDOM? 
 - Pick random page for every replacement
 - Typical solution for TLB's. Simple hardware
 - Pretty unpredictable - makes it hard to make real-time guarantees
- What about FIFO?
 - 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

Replacement Policies (Con't)

- What about LRU?
 - 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)

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 \Rightarrow 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