Caching in Operating Systems Design & Systems Programming

David E. Culler

CS162 – Operating Systems and Systems Programming

Lecture 19

October 13, 2014

Reading: A&D 9.6-7
HW 4 going out
Proj 2 out today

Objectives

- Recall and solidify understanding the concept and mechanics of caching.
- Understand how caching and caching effects pervade OS design.
- Put together all the mechanics around TLBs, Paging, and Memory caches
- Solidify understanding of Virtual Memory

Review: Memory Hierarchy

- 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

Speed (ns):	0.3	1	3	10-30	100	100,000 (0.1 ms)	10,000,000 (10 ms)
Size (bytes):	100Bs	10kBs	100kBs	MBs	GBs	100GBs	TBs

Examples

- vmstat –s
- top
- mac-os utility/activity

Where does caching arise in Operating Systems ?

Where does caching arise in Operating Systems ?

- 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?

Where does caching arise in Operating Systems ?

- Indirect - dealing with cache effects
- Process scheduling
  - which and how many processes are active ?
  - large memory footprints versus small ones ?
  - priorities ?
- Impact of thread scheduling on cache performance
  - rapid interleaving of threads (small quantum) may degrade cache performance
    - increase ave MAT !!!
- Designing operating system data structures for cache performance.
- All of these are much more pronounced with multiprocessors / multicores

MP $

Memory

bus

$	$	* * *	$
P	P		P

Working Set Model (Denning ~70)

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

Address

Time

Cache Behavior under WS model

1

0

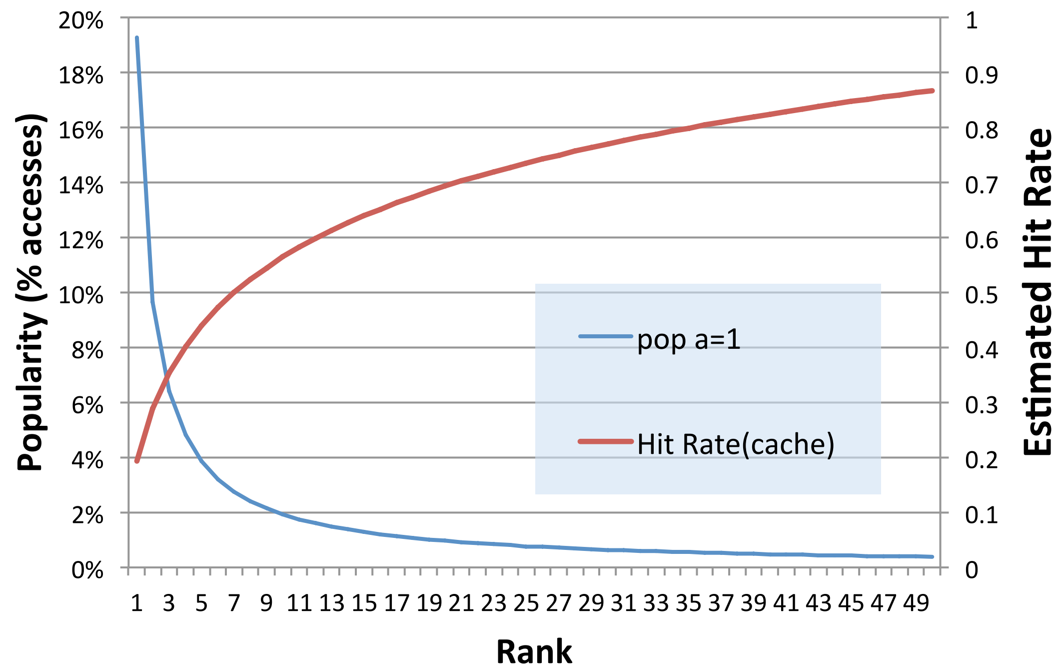
Hit Rate

new working set fits

Cache Size

- Amortized by fraction of time the WS 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 $\alpha 1/r^a$
- 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

Where does caching arise in Operating Systems ?

- Maintaining the *correctness* of various caches

- TLB consistent with PT across context switches ?
- Across updates to the PT ?
- Shared pages mapped into VAS of multiple processes ?

Going into detail on TLB

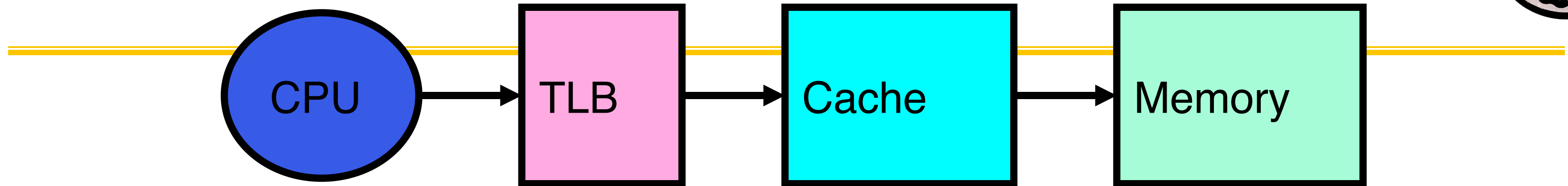
What Actually Happens on a TLB Miss?

- 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 (ala 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

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!

What TLB organization makes sense?



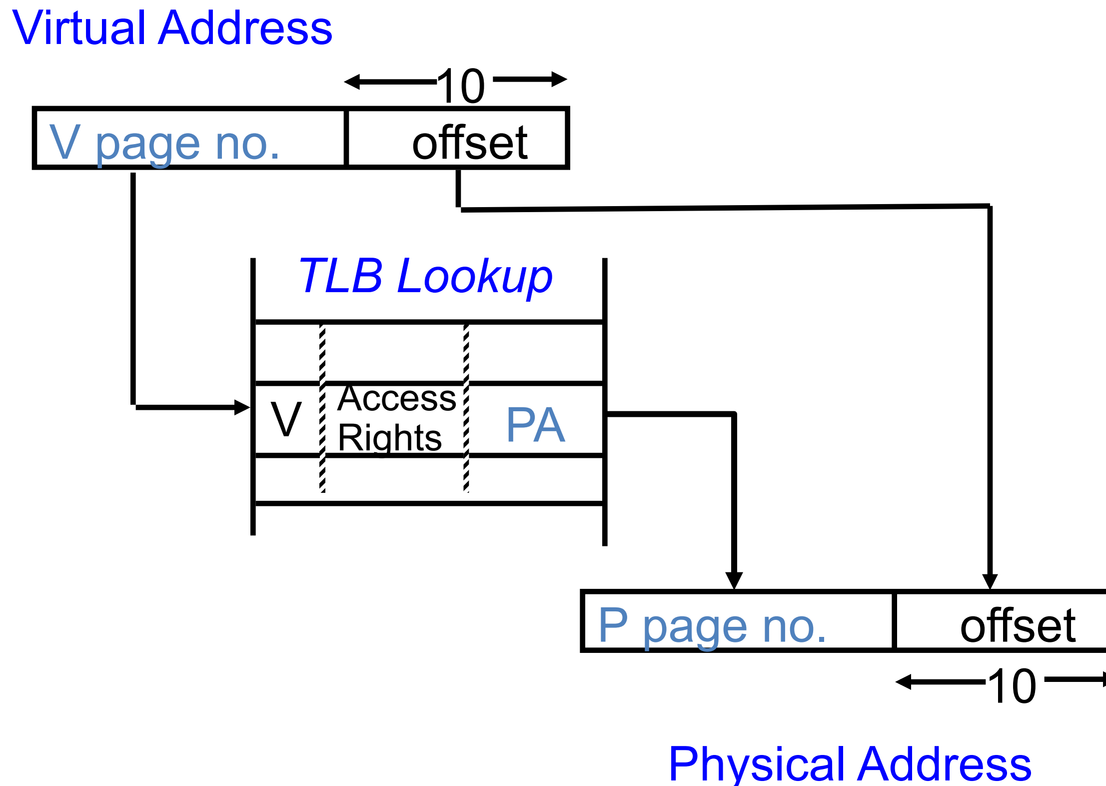
- Needs to be really fast
  - Critical path of memory access
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high!
  - This argues that cost of Conflict (Miss Time) is much higher than slightly increased cost 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

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
- 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 relative to memory cache access?
  - Before memory cache lookup?
  - In parallel with memory cache lookup?

Reducing translation time further

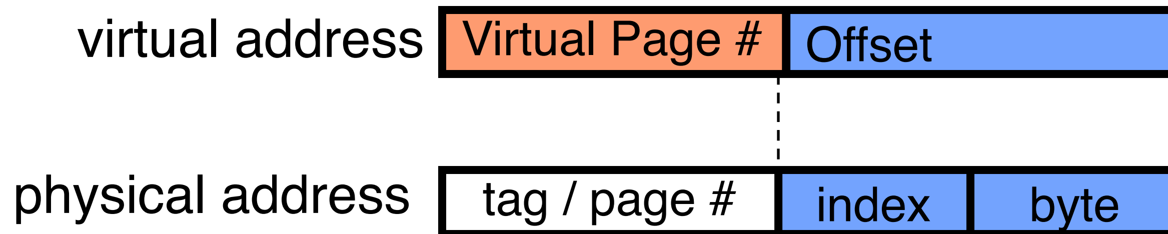
- As described, TLB lookup is in serial with cache lookup:



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

Overlapping TLB & Cache Access (1/2)

- 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



Overlapping TLB & Cache Access (1/2)

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

Putting Everything Together: Address Translation

Virtual Address:

Virtual P1 index	Virtual P2 index	Offset

PageTablePtr

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

PageTablePtr

Page Table (1st level)

Page Table (2nd level)

Physical Address:

Physical Page #	Offset

Physical Memory:

TLB:

...

Putting Everything Together: Cache

Virtual Address:

Virtual P1 index	Virtual P2 index	Offset

PageTablePtr

Page Table (1st level)

Page Table (2nd level)

TLB:

...

Physical Address:

Physical Page #	Offset

tag	index	byte

cache:

tag: block:

...

Physical Memory:

Admin: Projects

- Project 1
  - deep understanding of OS structure, threads, thread implementation, synchronization, scheduling, and interactions of scheduling and synchronization
  - work effectively in a team
    - effective teams work together with a plan

    => schedule three 1-hour joint work times per week
- Project 2
  - exe load and VAS creation provided for you
  - syscall processing, FORK+EXEC, file descriptors backing user file handles, ARGV
    - registers & stack frames
  - two development threads for team
    - but still need to work together

Virtual Memory – the disk level

Reacall: the most basic OS function

OS Basics: Loading

Loading an executable into memory

disk (huge)

info

data

code

exe

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

disk (huge)

info
data
code
exe

process VAS

kernel
stack
sbrk
heap
data
code

memory

user page frames
user pagetable
kernel code & data

- Utilized pages in the VAS are backed by a page block on disk
  - called the backing store
  - typically in an optimized block store, but can think of it like a file

Create Virtual Address Space of the Process

disk (huge, TB)

info
data
code
exe

stack
heap
data
code

process VAS (GBs)

kernel
stack
heap
data
code

memory

user page frames

user pagetable

kernel code & data

- 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

disk (huge, TB)

VAS – per process

PT

memory

info

data

code

exe

stack

heap

data

code

kernel

stack

heap

data

code

user page frames

user pagetable

kernel code & data

- 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

disk (huge, TB)

VAS – per process

memory

info
data
code
exe

stack
heap
data
code

kernel
stack
heap
data
code

user page frames

user pagetable

kernel code & data

- 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 is required to map non-resident pages to disk?

- FindBlock(PID, page#) => disk_block

- Like the PT, but purely software
- Where to store it?
- Usually want backing store for resident pages too.
- Could use hash table (like Inverted PT)

Provide Backing Store for VAS

disk (huge, TB)

info

data

exe

stack

heap

data

stack

heap

data

code

VAS 1

PT 1

kernel

stack

heap

data

code

VAS 2

PT 2

kernel

stack

heap

data

code

memory

user page frames

user pagetable

kernel code & data

On page Fault …

disk (huge, TB)

info
data
exe
stack
heap
data
stack
heap
data
code

VAS 1
PT 1
memory
kernel
stack
heap
data
code

VAS 2
PT 2
kernel
stack
heap
data
code

user page frames

user pagetable

kernel code & data

active process & PT

On page Fault … find & start load

disk (huge, TB)

info

data

exe

stack

heap

data

stack

heap

data

code

VAS 1

PT 1

memory

kernel

stack

heap

data

code

VAS 2

PT 2

kernel

stack

heap

data

code

user page frames

user pagetable

kernel code & data

active process & PT

On page Fault … schedule other P or T

disk (huge, TB)

info
data
exe
stack
heap
data
stack
heap
data
code

VAS 1
PT 1
kernel
stack
heap
data
code

VAS 2
PT 2
kernel
stack
heap
data
code

memory

user page frames

user pagetable

kernel code & data

active process & PT

On page Fault … update PTE

Eventually reschedule faulting thread

disk (huge, TB)

info

data

exe

stack

heap

data

stack

heap

data

code

VAS 1

kernel

stack

heap

data

code

PT 1

memory

user page frames

user pagetable

kernel code & data

VAS 2

kernel

stack

heap

data

code

PT 2

active process & PT

Where does the OS get the frame?

- Keeps a free list
- Unix runs a “reaper” if memory gets too full
- As a last resort, evict a dirty page first

How many frames per process?

- Like thread scheduling, need to "schedule" memory resources
  - allocation of frames per process
    - utilization? fairness? priority?
  - allocation of disk paging bandwith

Historical Perspective

- Mainframes and minicomputers (servers) were "always paging"
  - memory was limited
  - processor rates <> disk xfer rates were much closer
- When overloaded would THRASH
  - with good OS design still made progress
- Modern systems hardly every page
  - primarily a safety net + lots of untouched "stuff"
  - plus all the other advantages of managing a VAS

Summary

- Virtual address space for protection, efficient use of memory, AND multi-programming.
  – hardware checks & translates when present
  – OS handles EVERYTHING ELSE
- Conceptually memory is just a cache for blocks of VAS that live on disk
  – but can never access the disk directly
- Address translation provides the basis for sharing
  – shared blocks of disk AND shared pages in memory
- How else can we use this mechanism?
  – sharing ???
  – disks transfers on demand ???
  – accessing objects in blocks using load/store instructions