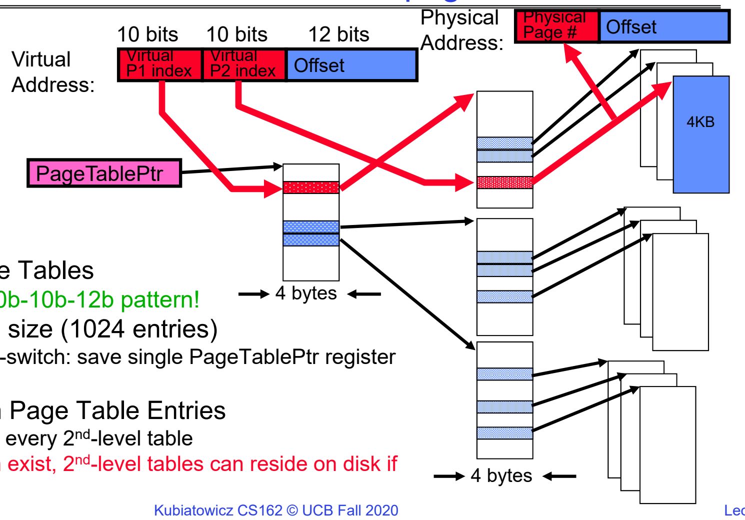
CS162
Operating Systems and Systems Programming
Lecture 15

Memory 3: Caching and TLBs (Con't), Demand Paging

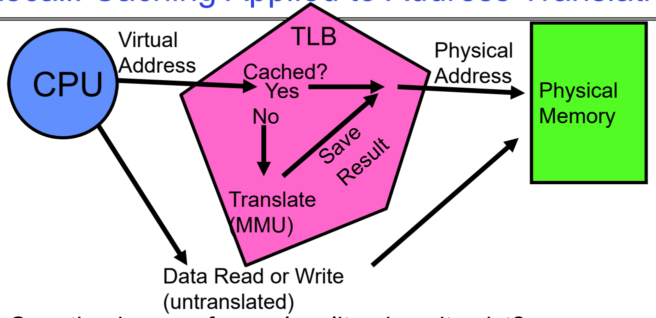
October 19th, 2020
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

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

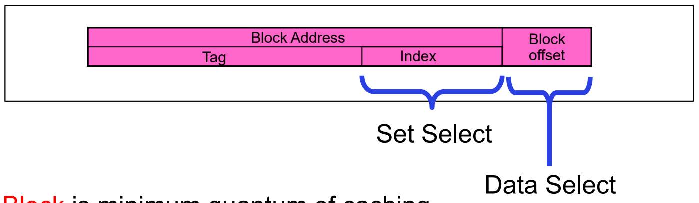


- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  - Sure: multiple 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

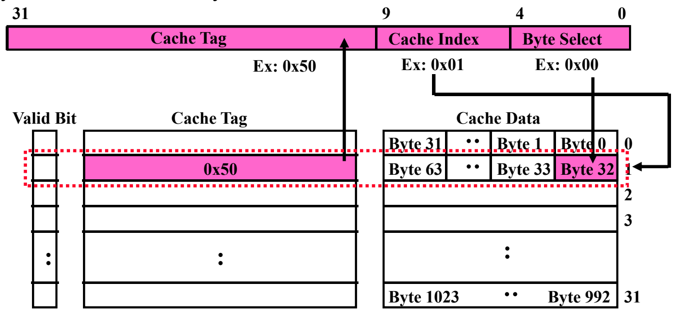
How is a Block found in a Cache?



- 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

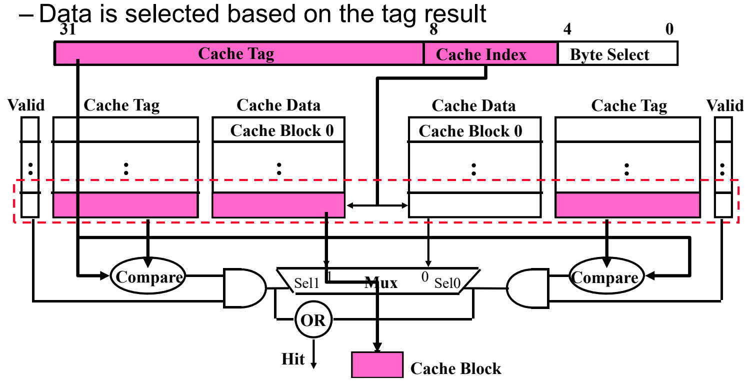
Review: Direct Mapped Cache

- 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



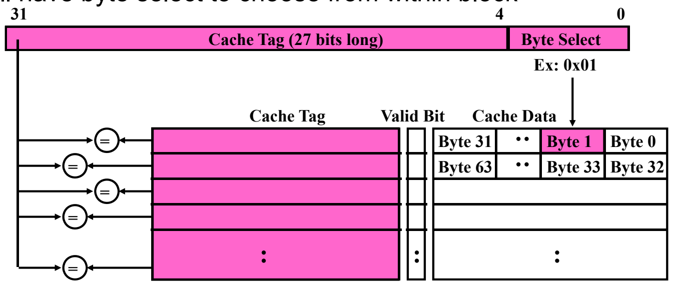
Review: Set Associative Cache

- 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



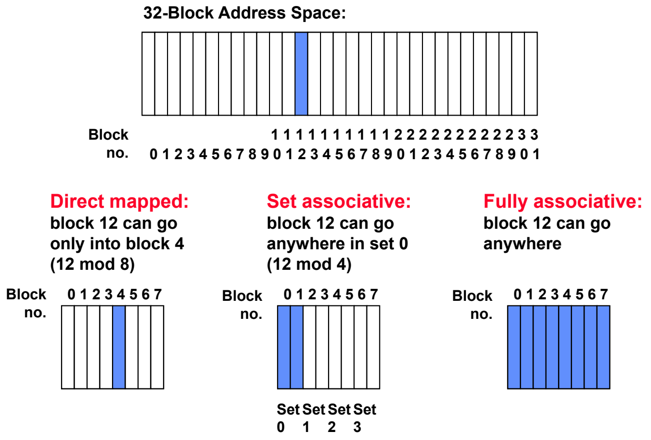
Review: Fully Associative Cache

- 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



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:

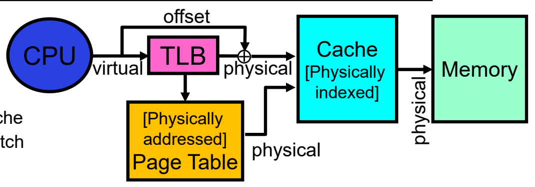
	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

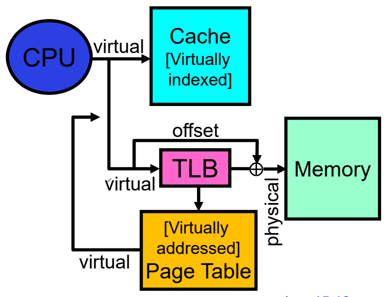
Review: What happens on a write?

- 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?

$$\begin{tikzcd} \text{CPU} \arrow[r] & \text{TLB} \arrow[r] & \text{Cache [Physically indexed]} \arrow[r] & \text{Memory} \end{tikzcd}$$

- 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

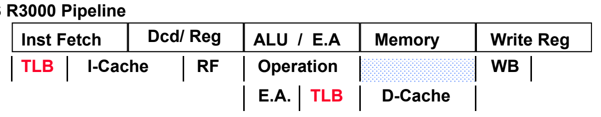
TLB organization: include protection

- 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:

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

Example: R3000 pipeline includes TLB "stages"

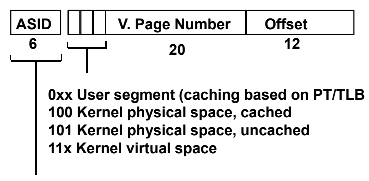
MIPS R3000 Pipeline



TLB

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

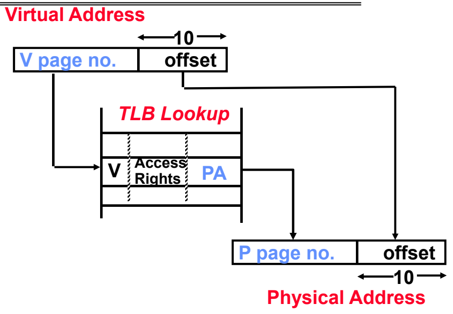
Virtual Address Space



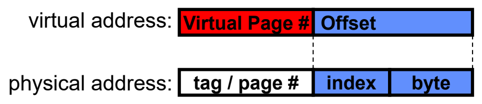
0xx User segment (caching based on PT/TLB entry)
100 Kernel physical space, cached
101 Kernel physical space, uncached
11x Kernel virtual space

Allows context switching among
64 user processes without TLB flush

Reducing translation time for physically-indexed caches

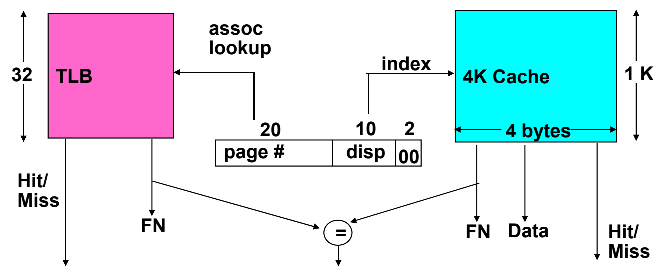


- As described, TLB lookup is in serial with cache lookup
  - Consequently, speed of TLB can impact speed of access to cache
- Machines with TLBs go one step further: overlap TLB lookup with cache access
  - Works because offset available early
  - Offset in virtual address exactly covers the "cache index" and "byte select"
  - Thus can select the cached byte(s) in parallel to perform address translation



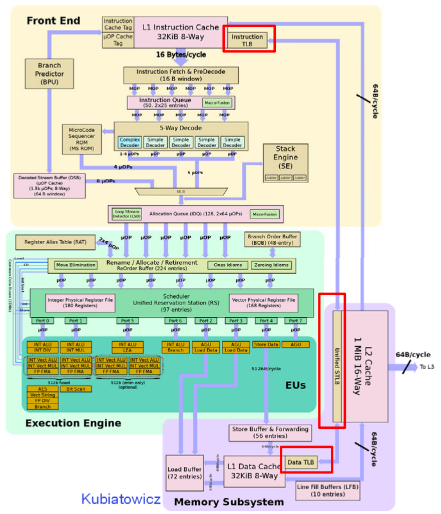
Overlapping TLB & Cache Access

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



- 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)


Front End
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode
Instruction Queue
5-Way Decode
MicroCode Sequencer ROM
Stack Engine (SE)
Decoded Stream Buffer (DSB)
Allocation Queue (IDQ)
Register Alias Table (RAT)
Branch Order Buffer (ROB)
Rename / Allocate / Retirement
Scheduler
Unified Reservation Station (RS)
Integer Physical Register File
Vector Physical Register File
EUs
Execution Engine
Unified STLB
L2 Cache 1 MiB 16-Way
64B/cycle
To L3
Store Buffer & Forwarding
Load Buffer
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB)
Memory Subsystem

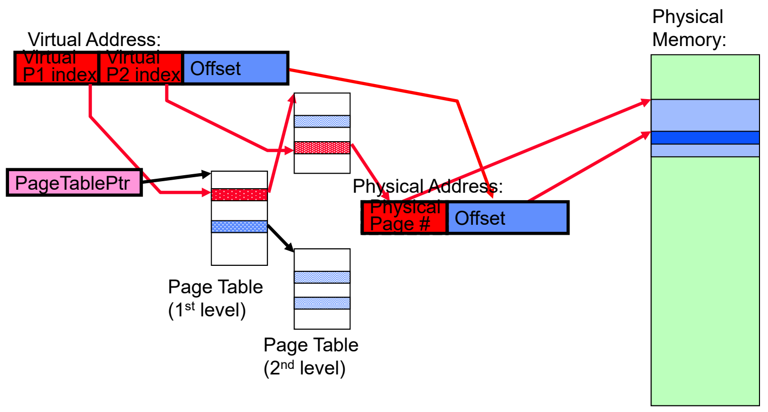
Current Example: Memory Hierarchy

- 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:

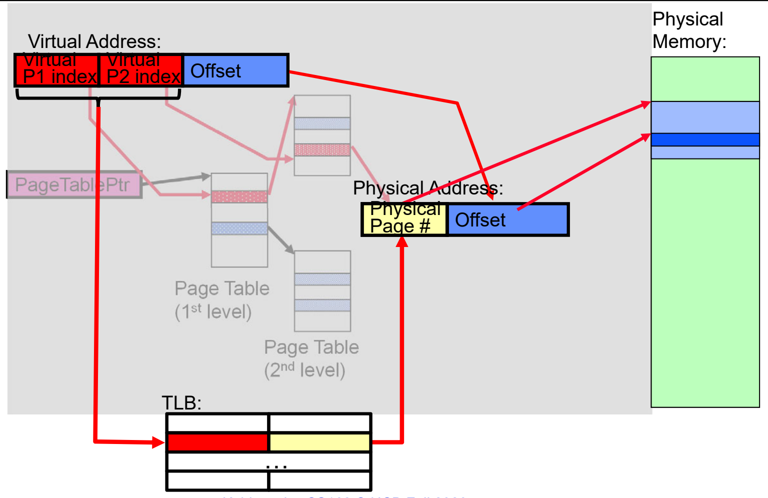
What happens on a Context Switch?

- 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!
  - Rember, everyone has their own version of the address “0”!

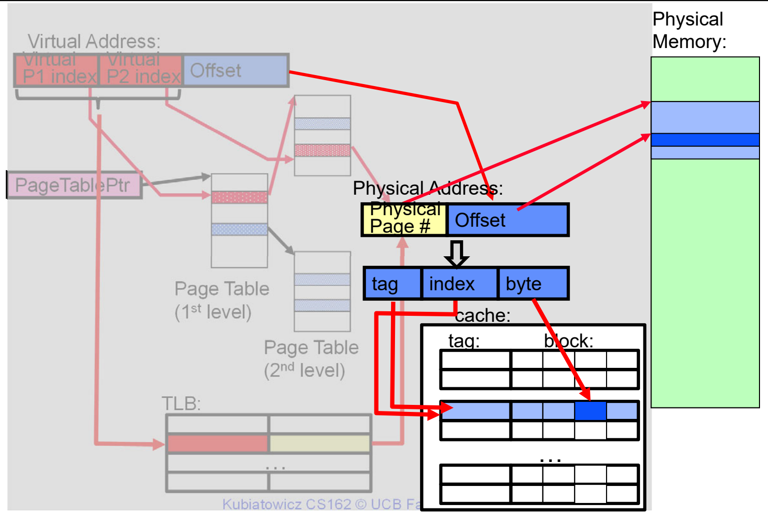
Putting Everything Together: Address Translation



Putting Everything Together: TLB



Putting Everything Together: Cache

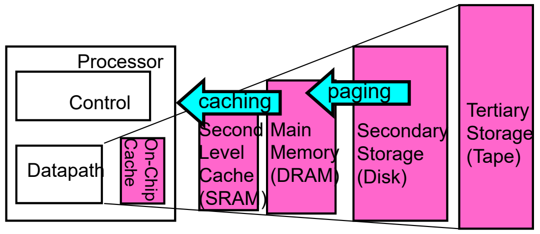


Page Fault

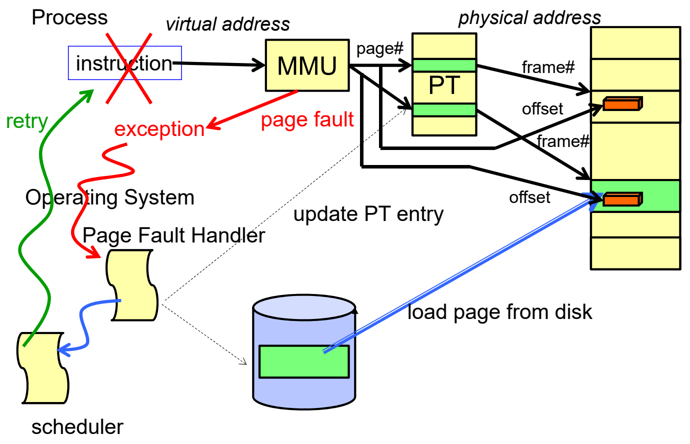
- The Virtual-to-Physical Translation fails
  - PTE marked invalid, Priv. Level Violation, Access violation, or does not exist
  - Causes an Fault / Trap
    - » Not an interrupt because synchronous to instruction execution
  - May occur on instruction fetch or data access
  - Protection violations typically 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



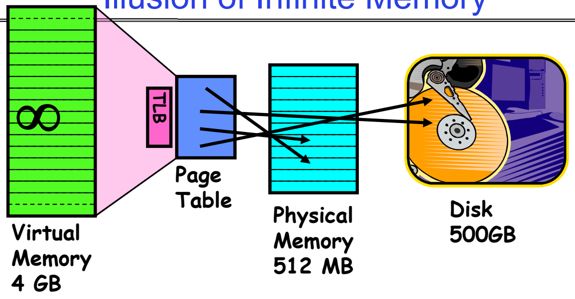
Page Fault ⇒ Demand Paging



Demand Paging as Caching, …

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

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"

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

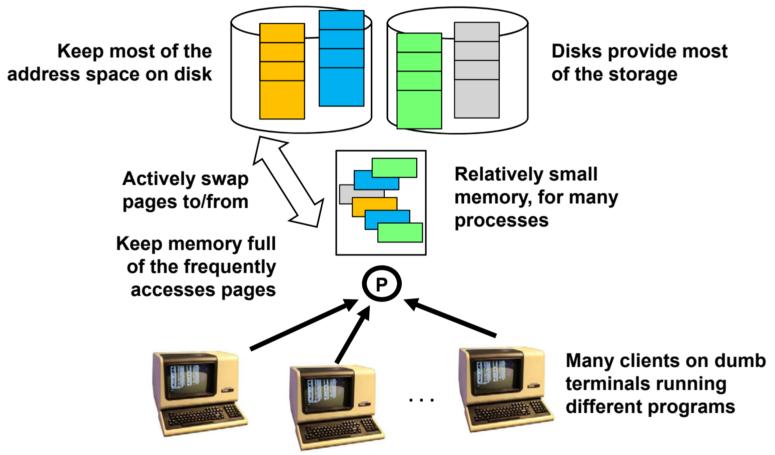
P:	Present (same as "valid" bit in other architectures)
W:	Writeable
U:	User accessible
PWT:	Page write transparent: external cache write-through
PCD:	Page cache disabled (page cannot be cached)
A:	Accessed: page has been accessed recently
D:	Dirty (PTE only): page has been modified recently
PS:	Page Size: PS=1⇒4MB page (directory only). Bottom 22 bits of virtual address serve as offset

Demand Paging Mechanisms

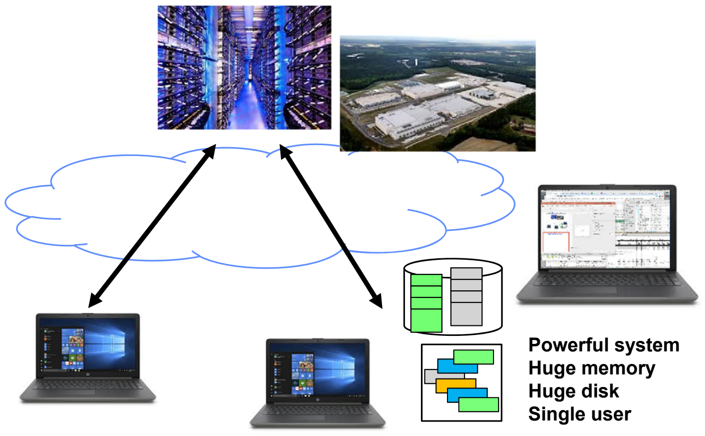
- PTE makes demand paging implementatable
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap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

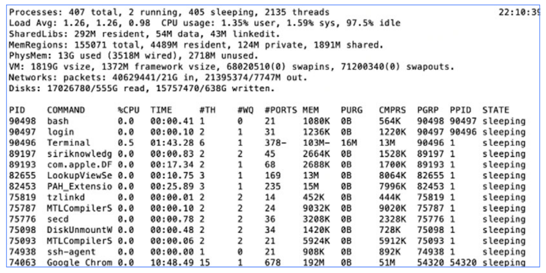
Origins of Paging


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

Very Different Situation Today



A Picture on one machine



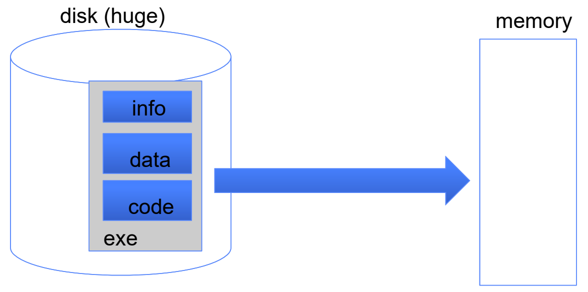
```
Processes: 407 total, 2 running, 405 sleeping, 2135 threads                          22:10:39
Load Avg: 1.26, 1.26, 0.98  CPU usage: 1.35% user, 1.59% sys, 97.5% idle
SharedLibs: 292M resident, 54M data, 43M linkedit.
MemRegions: 155071 total, 4489M resident, 124M private, 1891M shared.
PhysMem: 13G used (3518M wired), 2718M unused.
VM: 1819G vsize, 1372M framework vsize, 68020510(0) swapins, 71200340(0) swapouts.
Networks: packets: 40629441/21G in, 21395374/7747M out.
Disks: 17026780/555G read, 15757470/638G written.

PID    COMMAND      %CPU TIME     #TH    #WQ  #PORTS MEM    PURG   CMPRS  PGRP  PPID  STATE
90498  bash         0.0  00:00.41 1      0    21     1080K  0B     564K   90498 90497 sleeping
90497  login        0.0  00:00.10 2      1    31     1236K  0B     1220K  90497 90496 sleeping
90496  Terminal     0.5  01:43.28 6      1    378-   103M-  16M    13M    90496 1     sleeping
89197  siriknowledg 0.0  00:00.83 2      2    45     2664K  0B     1528K  89197 1     sleeping
89193  com.apple.DF 0.0  00:17.34 2      1    68     2688K  0B     1700K  89193 1     sleeping
82655  LookupViewSe 0.0  00:10.75 3      1    169    13M    0B     8064K  82655 1     sleeping
82453  PAH_Extensio 0.0  00:25.89 3      1    235    15M    0B     7996K  82453 1     sleeping
75819  tzlinkd      0.0  00:00.01 2      2    14     452K   0B     444K   75819 1     sleeping
75787  MTLCompilerS 0.0  00:00.10 2      2    24     9032K  0B     9020K  75787 1     sleeping
75776  secd         0.0  00:00.78 2      2    36     3208K  0B     2328K  75776 1     sleeping
75098  DiskUnmountW 0.0  00:00.48 2      2    34     1420K  0B     728K   75098 1     sleeping
75093  MTLCompilerS 0.0  00:00.06 2      2    21     5924K  0B     5912K  75093 1     sleeping
74938  ssh-agent    0.0  00:00.00 1      0    21     908K   0B     892K   74938 1     sleeping
74063  Google Chrom 0.0  10:48.49 15     1    678    192M   0B     51M    54320 54320 sleeping
```

- 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” …

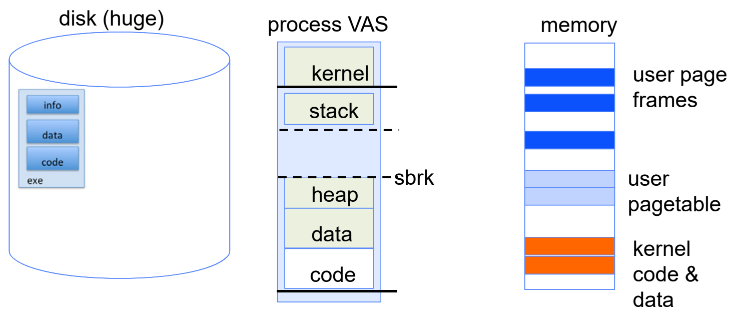
- Extend the stack
  – Allocate a page and zero it
- Extend the heap (sbrk of old, today mmap)
- Process Fork
  – Create a copy of the page table
  – Entries refer to parent pages – NO-WRITE
  – Shared read-only pages remain shared
  – Copy page on write
- Exec
  – Only bring in parts of the binary in active use
  – Do this on demand
- MMAP to explicitly share region (or to access a file as RAM)

Classic: Loading an executable into memory



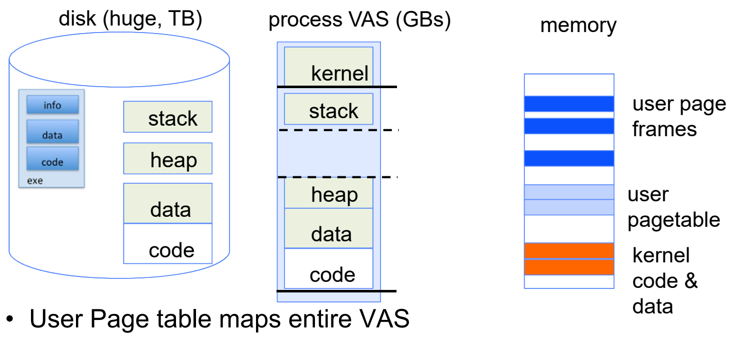
- .exe
  – lives on disk in the file system
  – contains contents of code & data segments, relocation entries and symbols
  – OS loads it into memory, initializes registers (and initial stack pointer)
  – program sets up stack and heap upon initialization: `crt0` (C runtime init)

Create Virtual Address Space of the Process



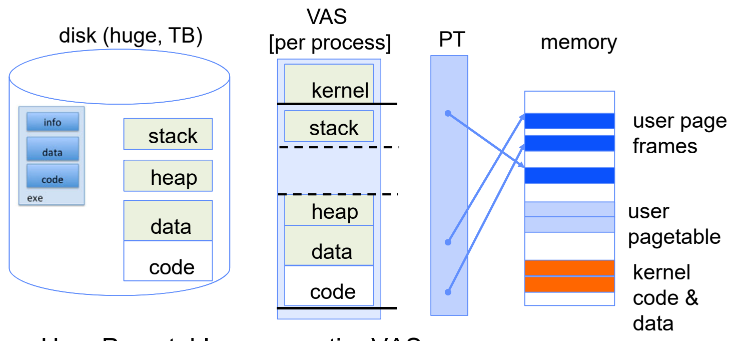
- Utilized pages in the VAS are backed by a page block on disk
  - Called the backing store or swap file
  - Typically in an optimized block store, but can think of it like a file

Create Virtual Address Space of the Process



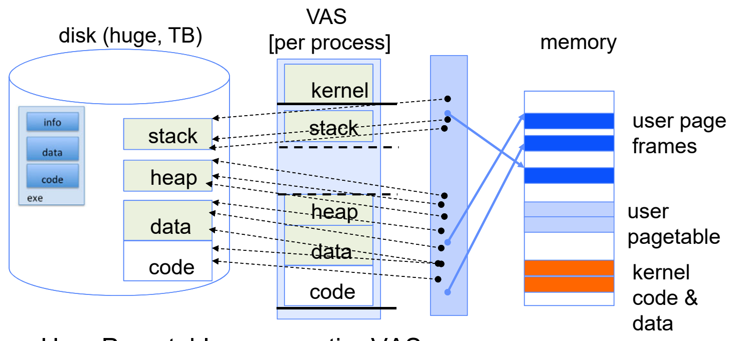
- User Page table maps entire VAS
- All the utilized regions are backed on disk
  - swapped into and out of memory as needed
- For *every* process

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

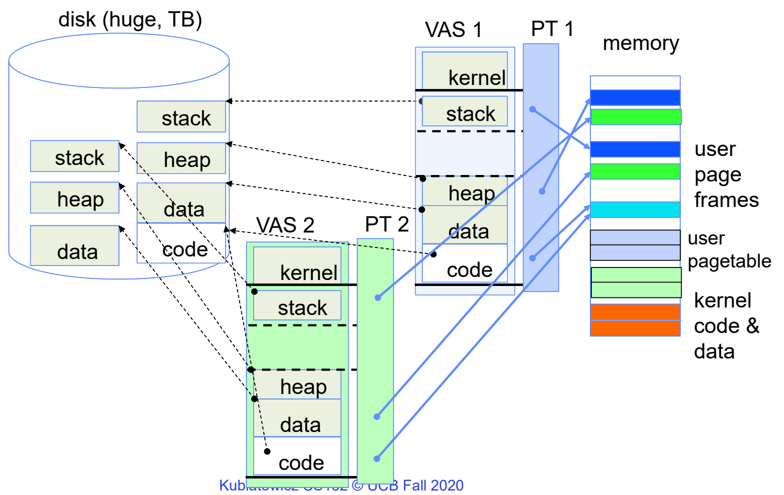


- User Page table maps entire VAS
- Resident pages mapped to memory frames
- For all other pages, OS must record where to find them on disk

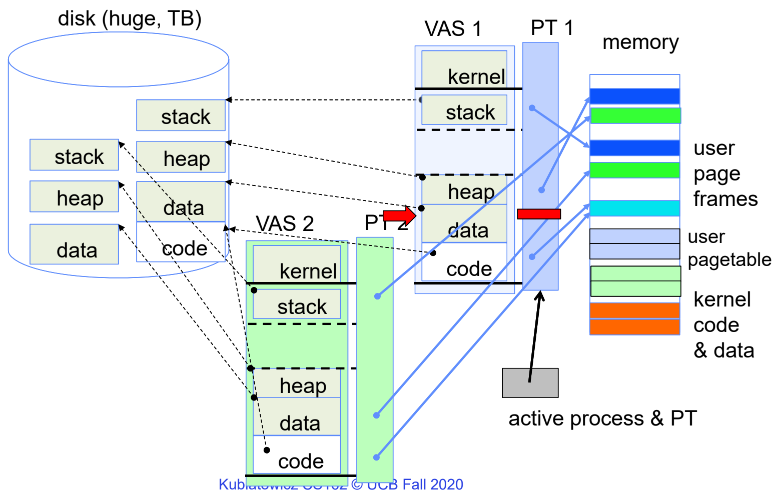
What Data Structure Maps Non-Resident Pages to Disk?

- `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

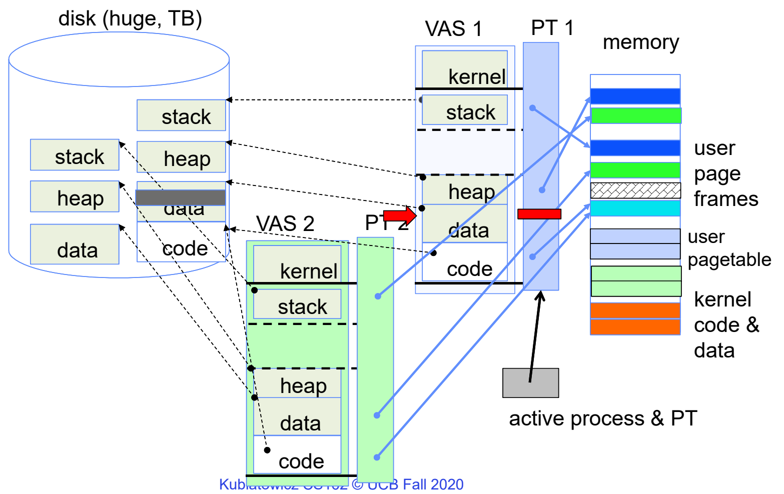
Provide Backing Store for VAS



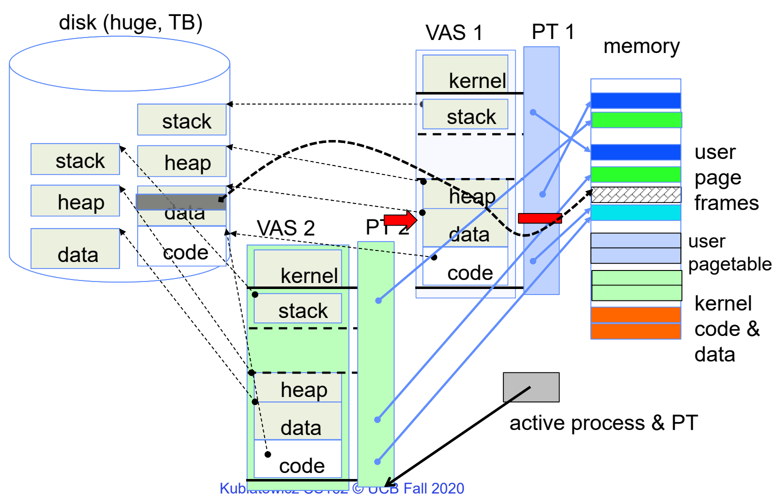
On page Fault …



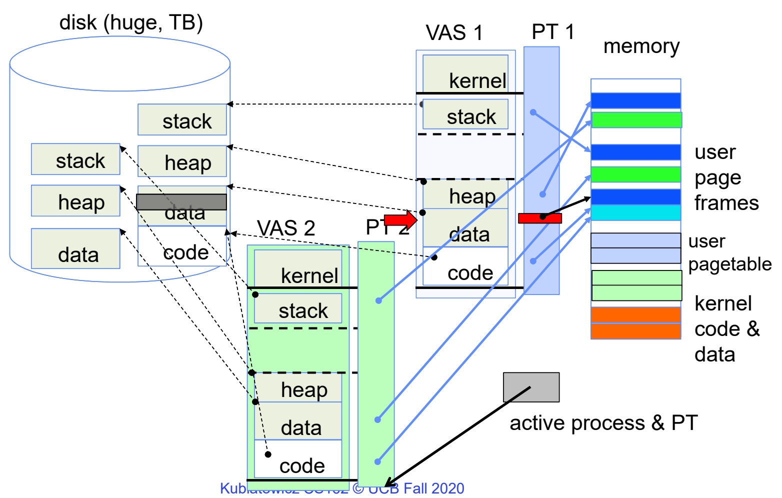
On page Fault … find & start load



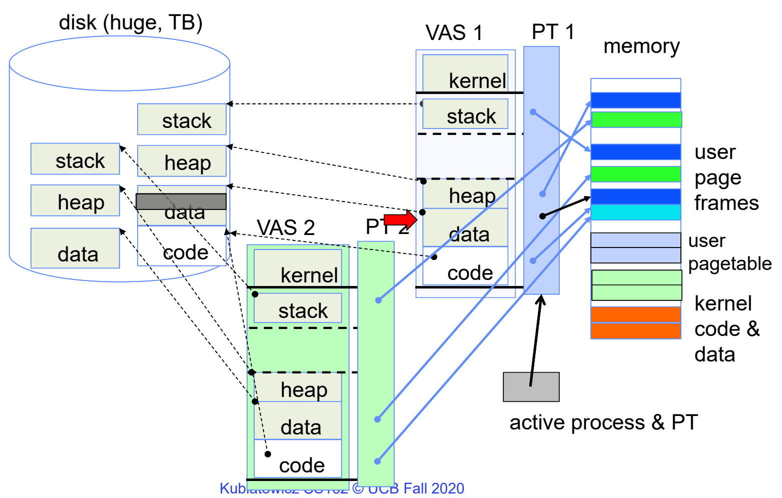
On page Fault … schedule other P or T



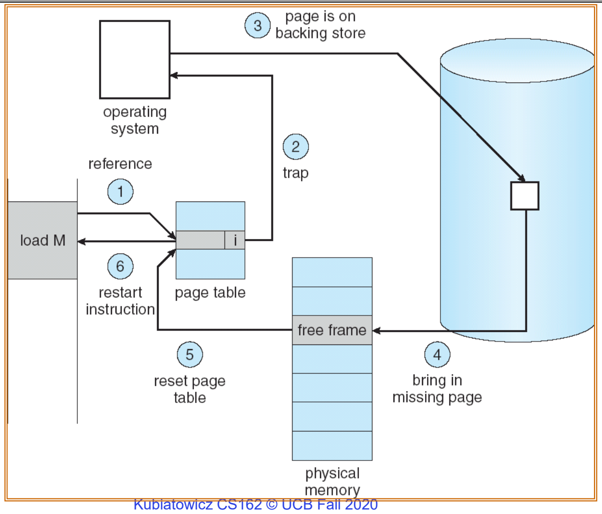
On page Fault … update PTE



Eventually reschedule faulting thread



Summary: Steps in Handling a Page Fault

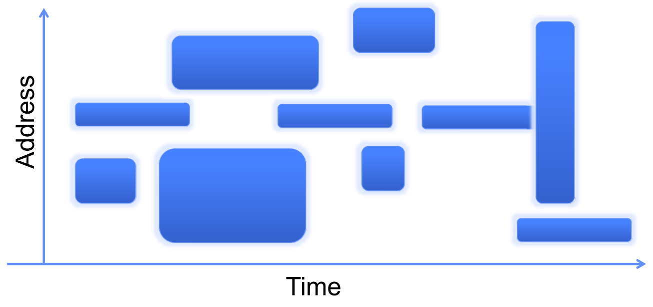


Some questions we need to answer!

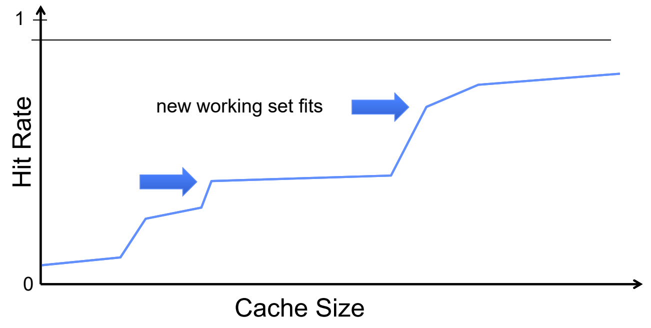
- During a page fault, where does the OS get a free frame?
  - Keeps a free list
  - Unix runs a “reaper” if memory gets too full
    - » Schedule dirty pages to be written back on disk
    - » Zero (clean) pages which haven’t been accessed in a while
  - As a last resort, evict a dirty page first
-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

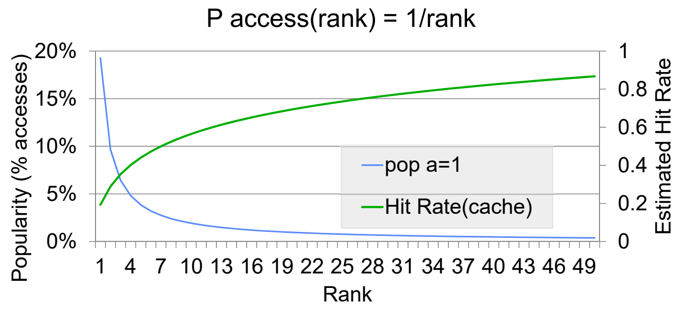


Cache Behavior under WS model



- 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



- Likelihood of accessing item of rank r is α $1/r^a$
- 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 = 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%?
  - EAT < 200ns x 1.1 $\Rightarrow$ 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