CS162
Operating Systems and Systems Programming
Lecture 17

Memory 4: Demand Paging Policies

March 19th, 2024

Prof. John Kubiatowicz

http://cs162.eecs.Berkeley.edu

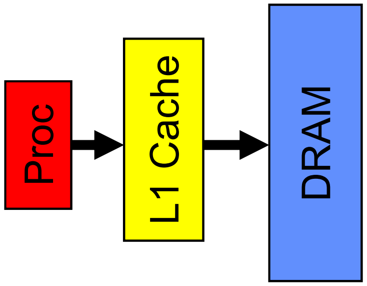
Recall 61C: Average Memory Access Time

- Used to compute access time probabilistically:

AMAT = Hit Rate$_{L1}$ x Hit Time$_{L1}$ + Miss Rate$_{L1}$ x Miss Time$_{L1}$

Hit Rate$_{L1}$ + Miss Rate$_{L1}$ = 1
Hit Time$_{L1}$ = Time to get value from L1 cache.
Miss Time$_{L1}$ = Hit Time$_{L1}$ + Miss Penalty$_{L1}$
Miss Penalty$_{L1}$ = AVG Time to get value from lower level (DRAM)

So, AMAT = Hit Time$_{L1}$ + Miss Rate$_{L1}$ x Miss Penalty$_{L1}$

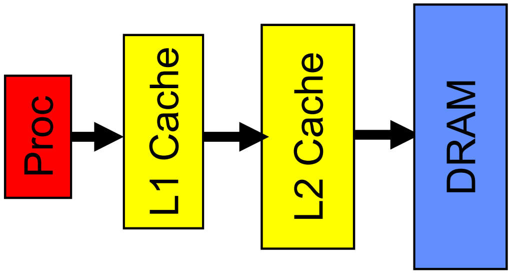


- What about more levels of hierarchy?

AMAT = Hit Time$_{L1}$ + Miss Rate$_{L1}$ x Miss Penalty$_{L1}$

Miss Penalty$_{L1}$ = AVG time to get value from lower level (L2)
= Hit Time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$

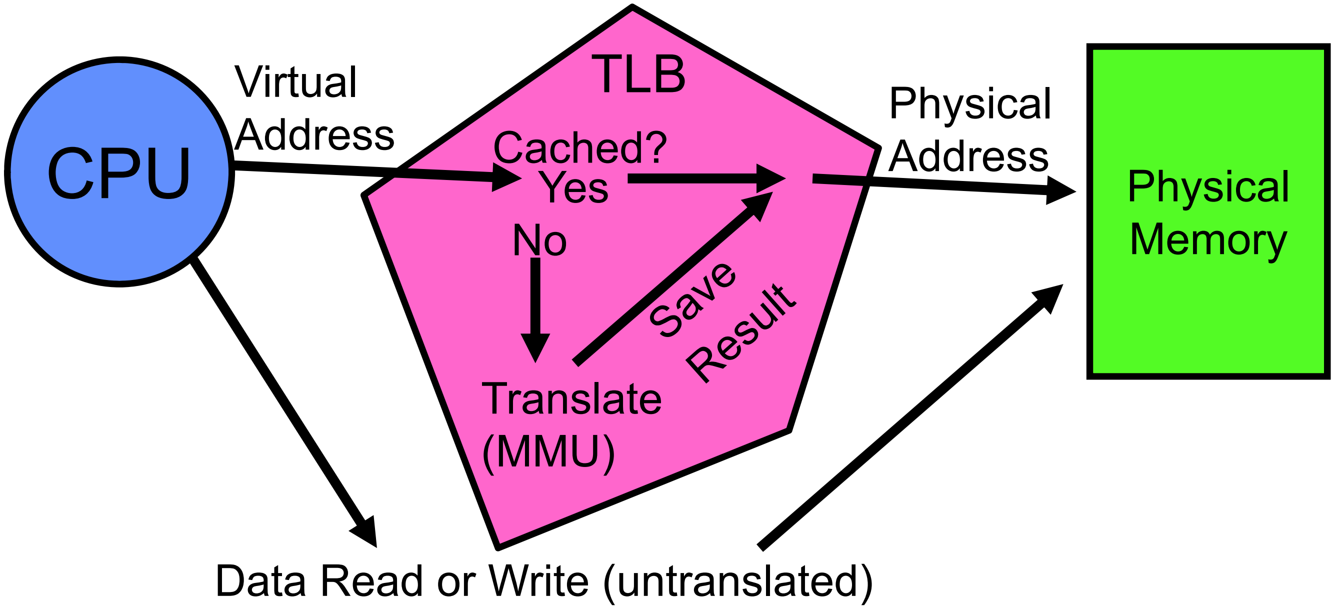
Miss Penalty$_{L2}$ = Average Time to fetch from below L2 (DRAM)



AMAT = Hit Time$_{L1}$ +
$\underline{\text{Miss Rate}_{L1}}$ x (Hit Time$_{L2}$ + $\underline{\text{Miss Rate}_{L2}}$ x Miss Penalty$_{L2}$)

- And so on … (can do this recursively for more levels!)

Recall: Caching Applied to Address Translation

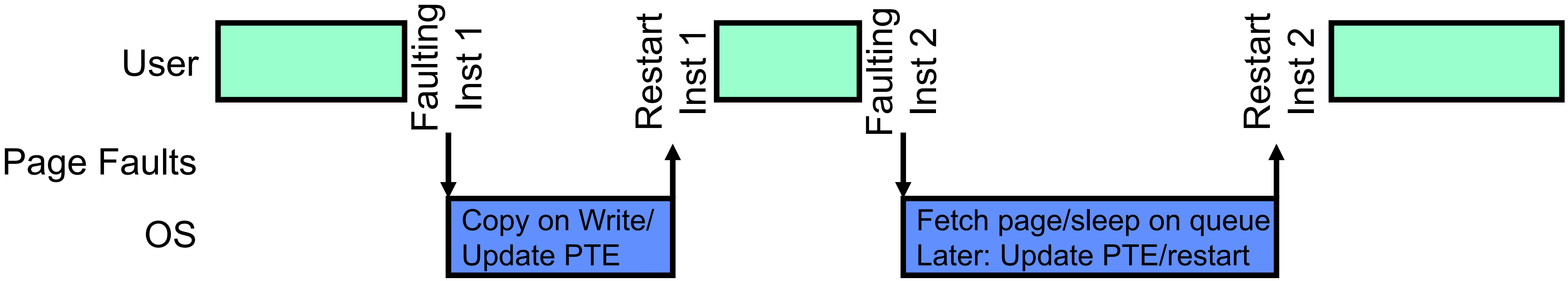


- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on same page (accesses are sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  - Sure: multiple levels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

What Actually Happens on a TLB Miss?

- Hardware traversed page tables (x86, many others):
  - 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: Page fault


User
Page Faults
OS
Faulting Inst 1
Restart Inst 1
Faulting Inst 2
Restart Inst 2
Copy on Write/ Update PTE
Fetch page/sleep on queue
Later: Update PTE/restart

- How to transparently restart faulting instructions?
  - (Consider load or store that gets 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?

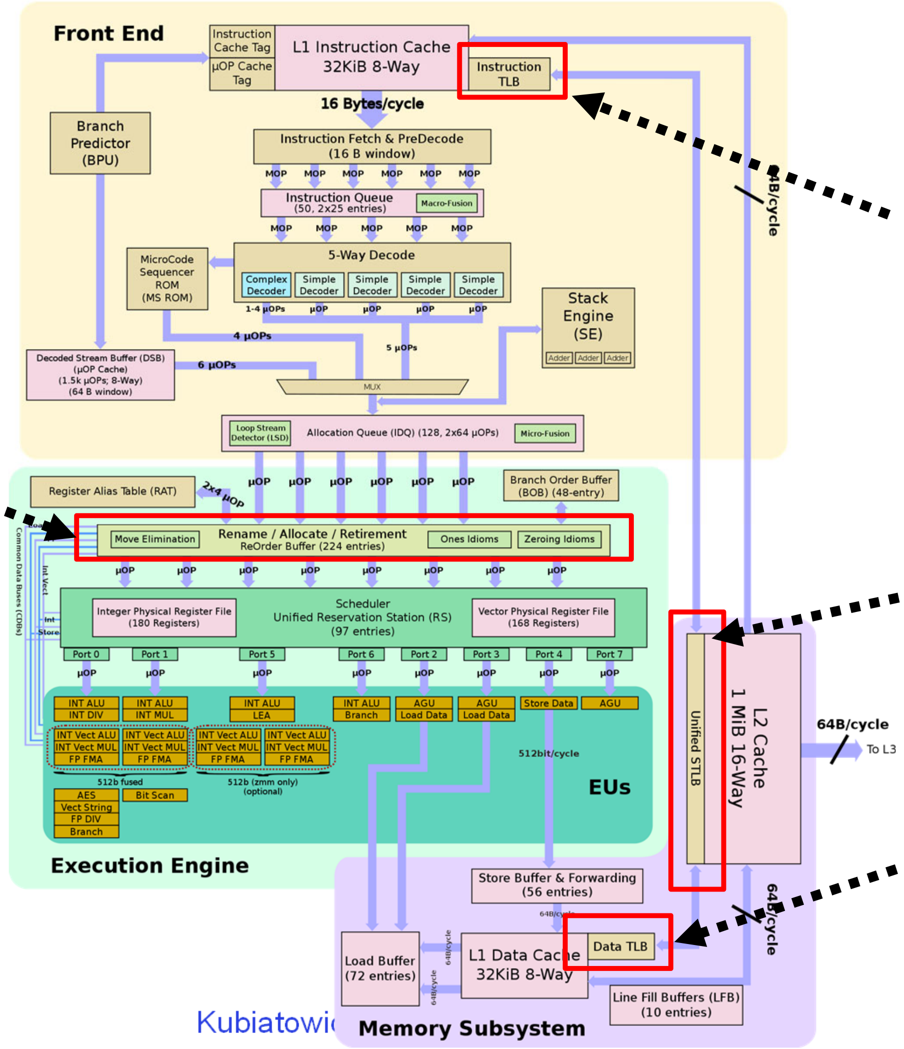
Consider weird things that can happen

- 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 writing 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)
      ```
    - » Restart after page fault: need 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, ...
  - x86 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

Recent Intel x86 (Skylake, Cascade Lake)


Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
1-4 µOPs
µOP
4 µOPs
5 µOPs
6 µOPs
Stack Engine (SE)
Adder
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
64B/cycle
Register Alias Table (RAT)
2x4 µOP
Branch Order Buffer (BOB) (48-entry)
Move Elimination
Rename / Allocate / Retirement
ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
Common Data Buses (CDBs)
Int Vect
Int
Store
Integer Physical Register File (180 Registers)
Scheduler
Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT MUL
LEA
Branch
AGU
Load Data
Store Data
INT Vect ALU
INT Vect MUL
FP FMA
512b fused
512b (zmm only) (optional)
AES
Vect String
FP DIV
Bit Scan
512bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 1 MiB 16-Way
64B/cycle
To L3
Store Buffer & Forwarding (56 entries)
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem

(L1) Instruction TLB

Reorder Buffer
(Precise Exceptions)

(L2) Unified TLB

(L1) Data TLB

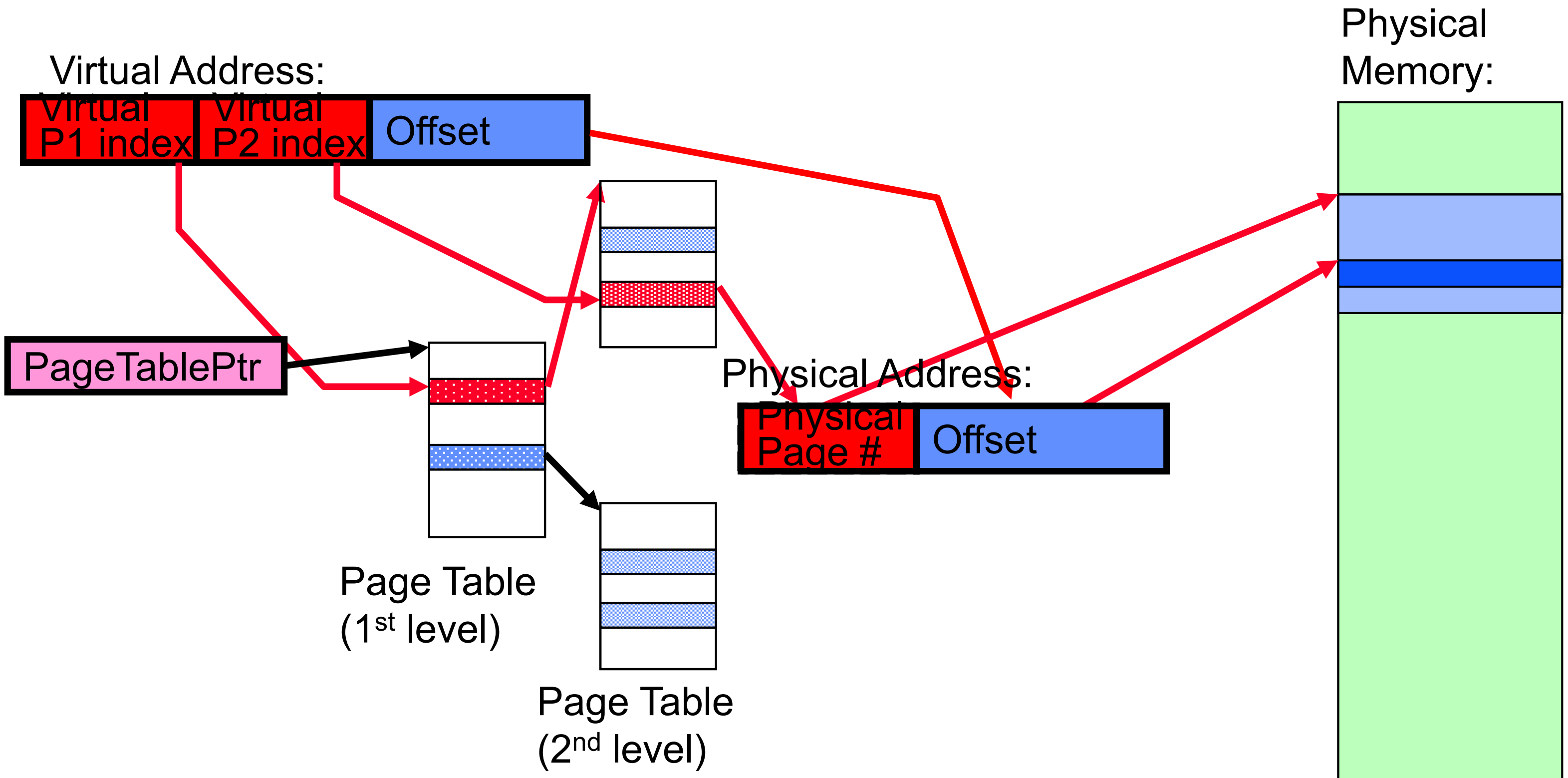
Recent Example: Memory Hierarchy

- 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:

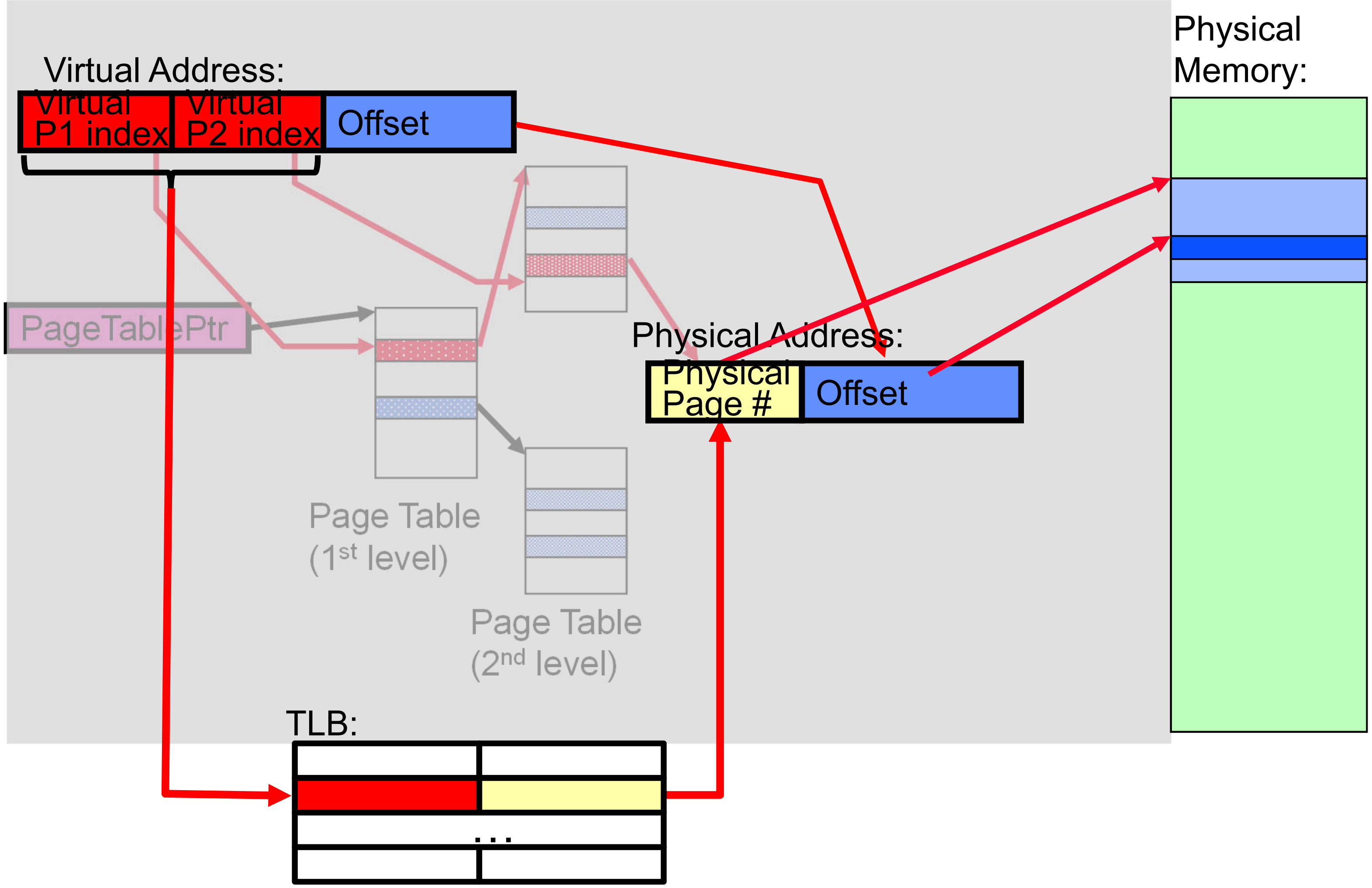
What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate (“Flush”) 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, Virtually-Tagged cache, need to flush cache!
  - Everyone has their own version of the address “0” and can’t distinguish them
  - This is one advantage of Virtually-Indexed, Physically-Tagged caches..

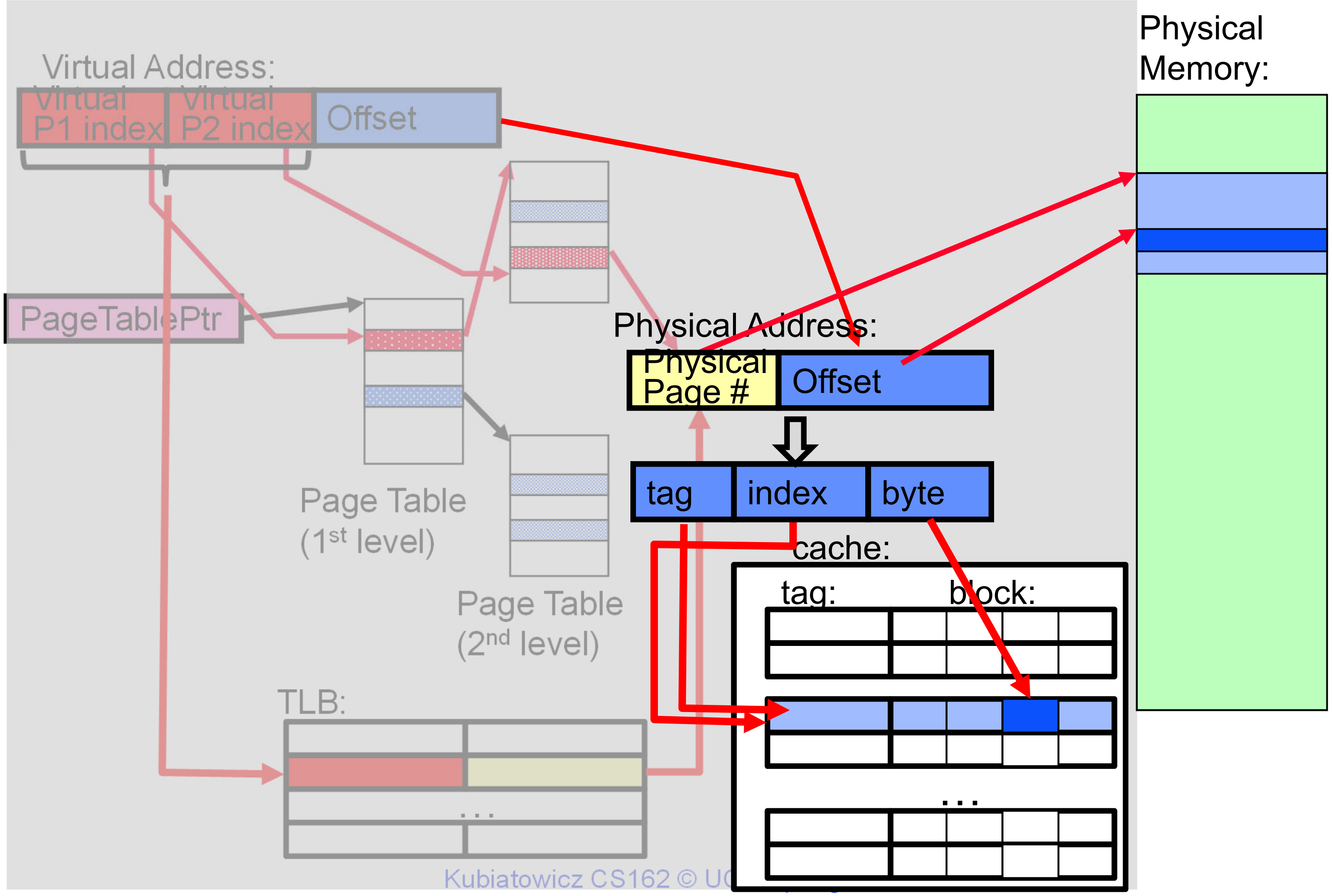
Putting Everything Together: Address Translation



Putting Everything Together: TLB



Putting Everything Together: Cache



Administrivia

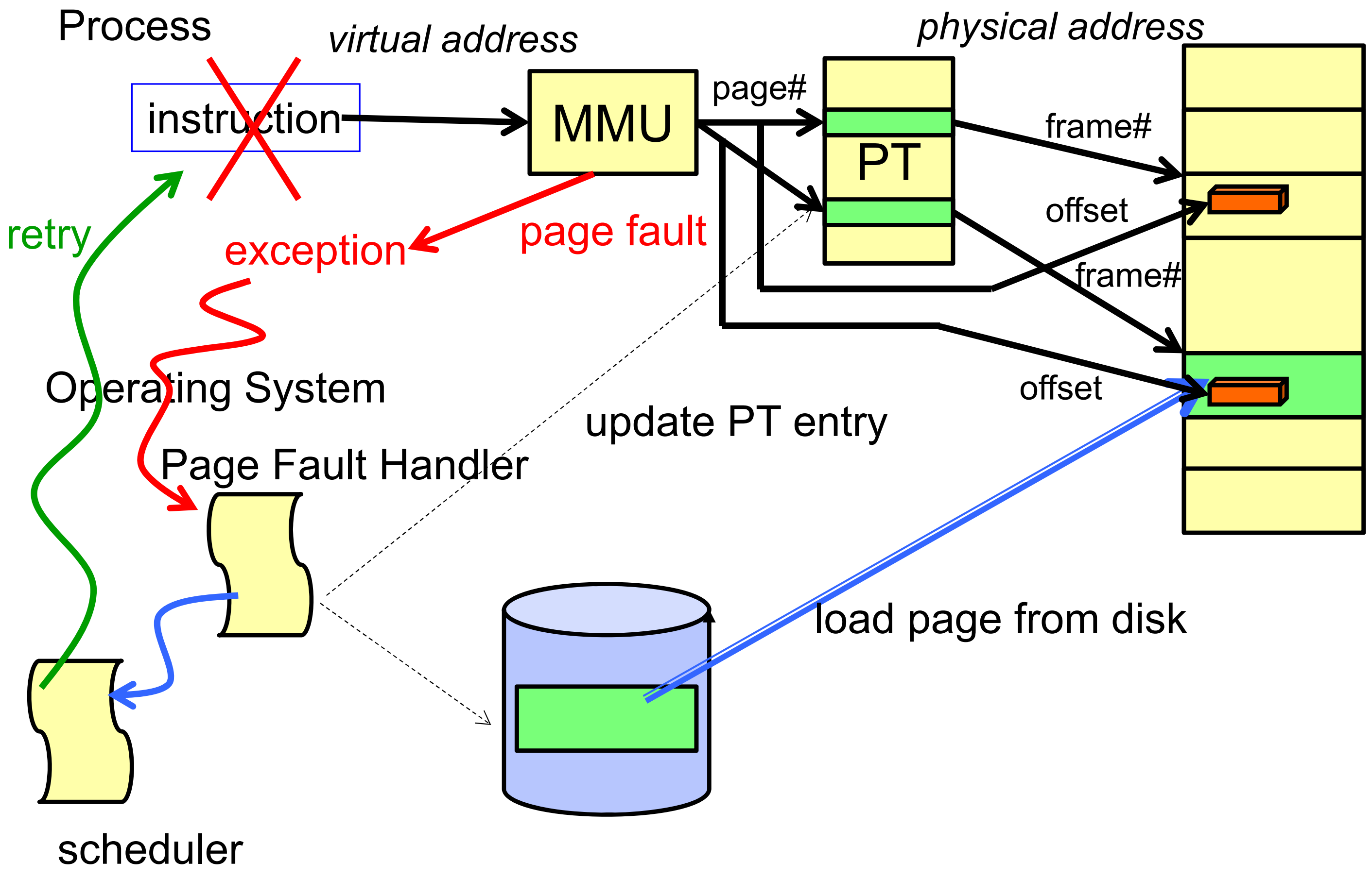
- Still grading exam!
  - Hopefully have it by early next week
- Project 2 in full swing
  - Stay on top of this one. Don’t wait until last moment to get pieces together
  - Decide how to your team is going divide up project 2
- Homework 4 also in full swing
- Make sure to fill out survey!
  - We really want to hear how you think we are doing
  - Also, will get a chance to suggest topics for the special topics lecture

Complete form quickly for attendance credit

Page Fault Handling

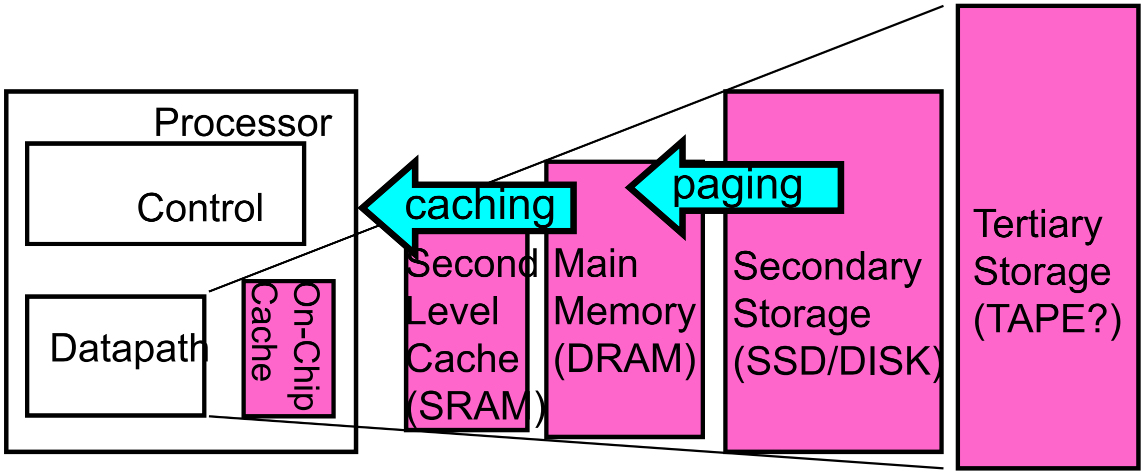
- The Virtual-to-Physical Translation fails
  - PTE marked invalid, Privilege Level Violation, Access violation, or does not exist
  - Causes an Fault / Trap
    - Not an interrupt because synchronous to instruction execution
  - May occur on instruction fetch or data access
  - Protection violations typically terminate the process
- 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
  - Need to execute software to allow hardware to proceed!

Page Fault ⇒ Demand Paging

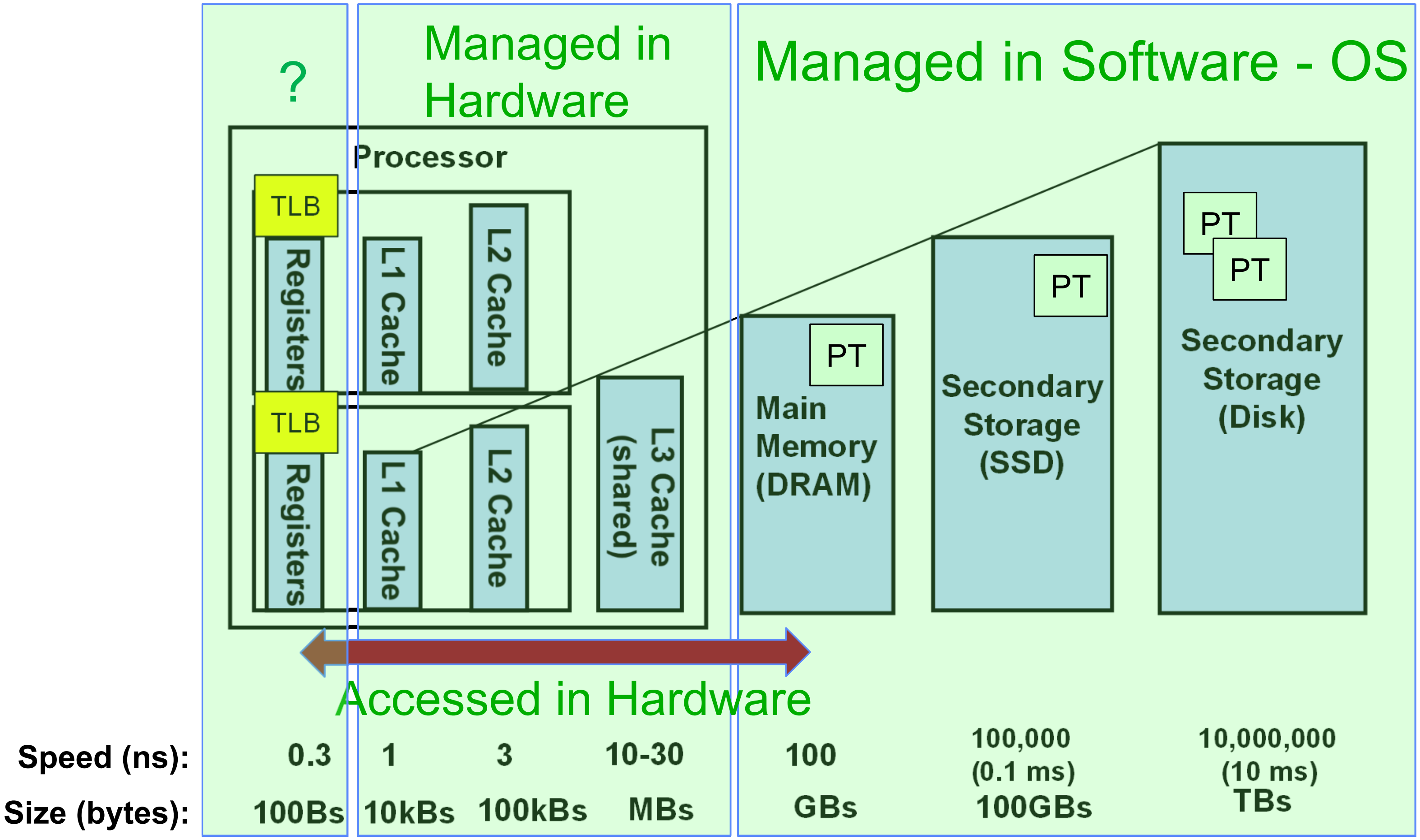


Demand Paging

- 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



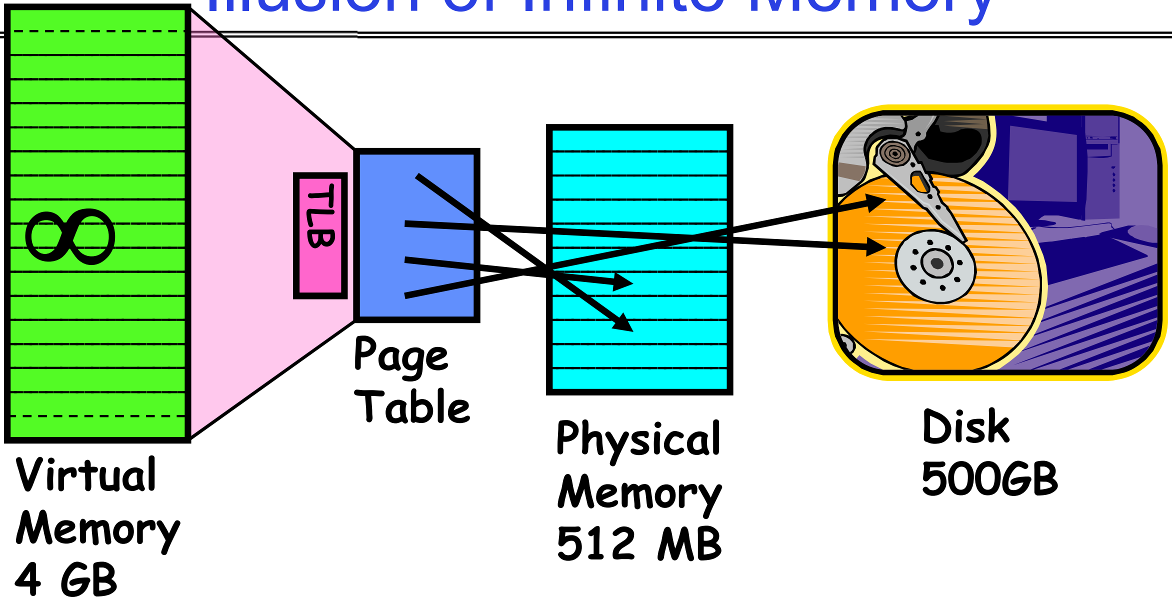
Management & Access to the Memory Hierarchy



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

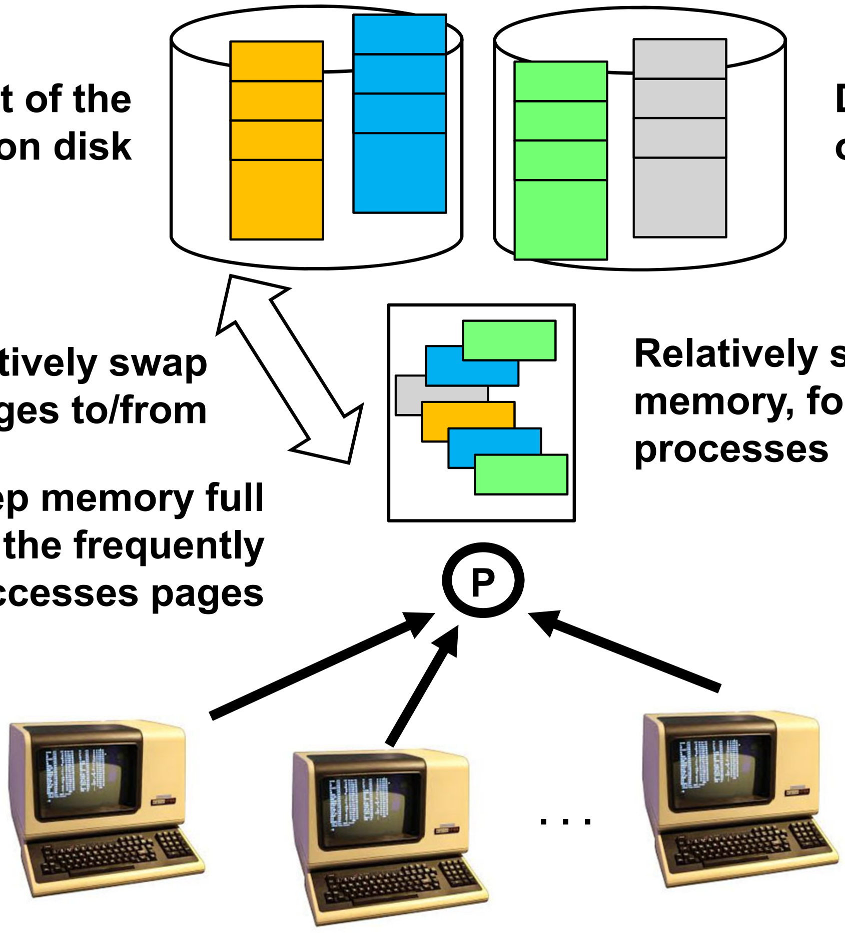
Demand Paging Mechanisms

- 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"

Cache

  - 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

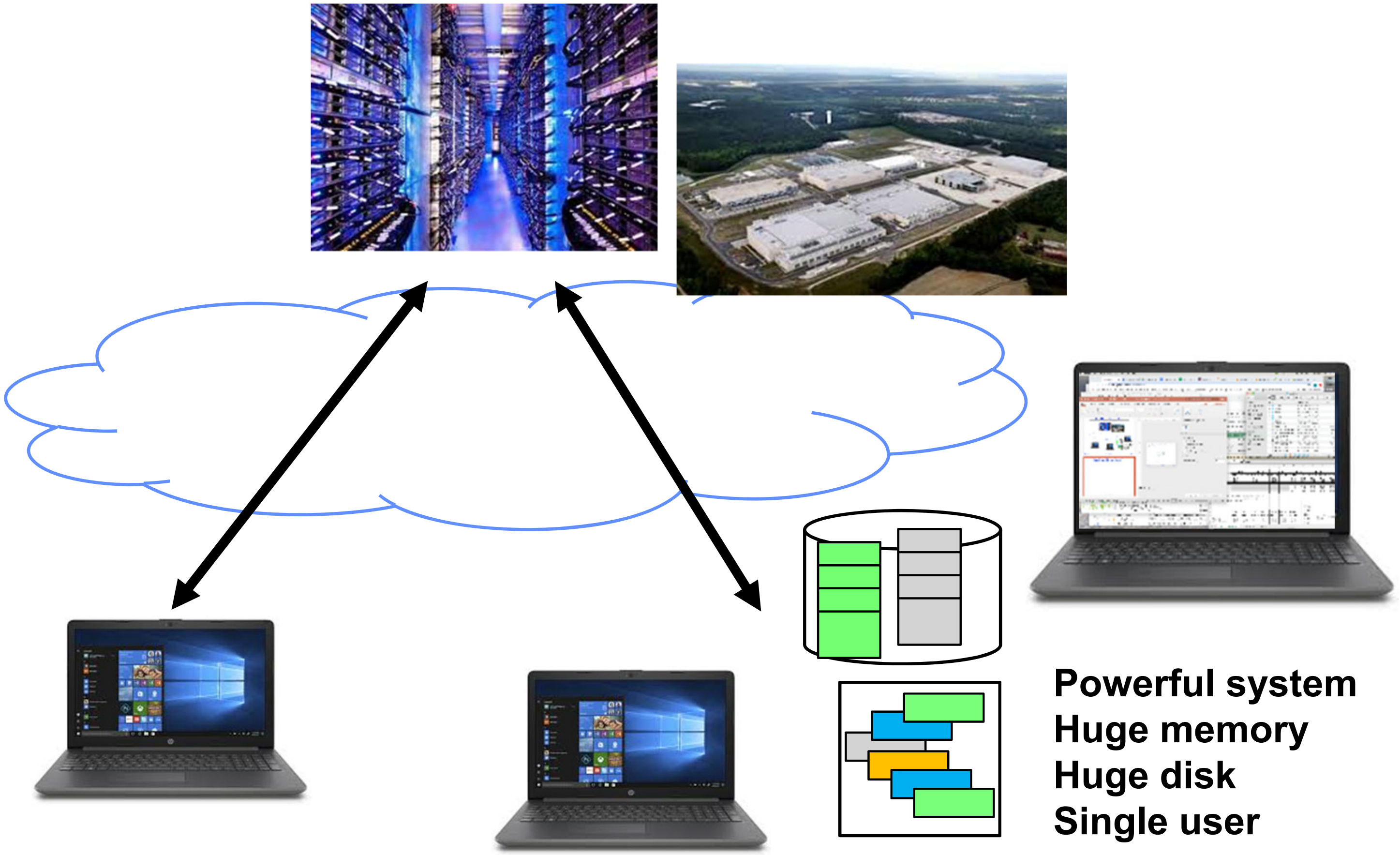
Actively swap pages to/from

Relatively small memory, for many processes

Keep memory full of the frequently accesses pages

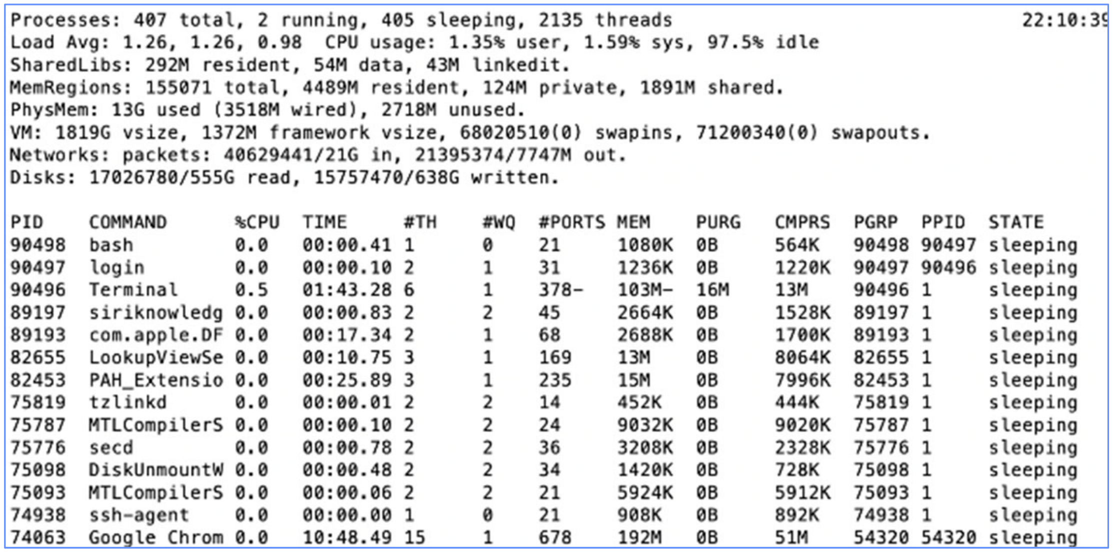
Many clients on dumb terminals running different programs

Very Different Situation Today



Powerful system
Huge memory
Huge disk
Single user

A Picture on one machine



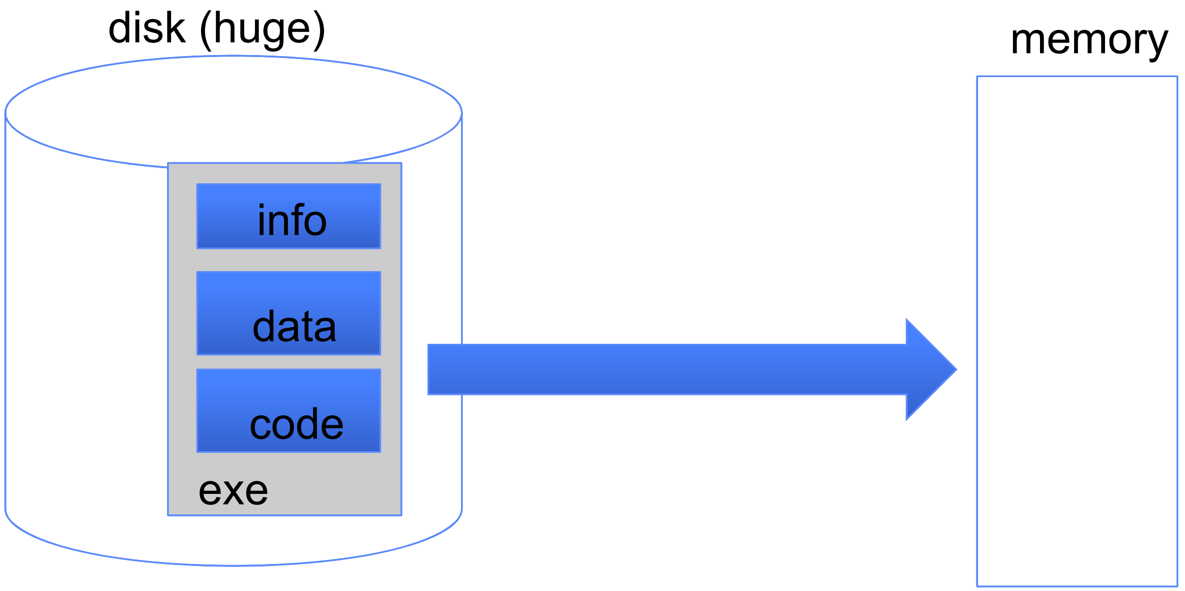
```
Processes: 407 total, 2 running, 405 sleeping, 2135 threads                    22:10:39
Load Avg: 1.26, 1.26, 0.98  CPU usage: 1.35% user, 1.59% sys, 97.5% idle
SharedLibs: 292M resident, 54M data, 43M linkedit.
MemRegions: 155071 total, 4489M resident, 124M private, 1891M shared.
PhysMem: 13G used (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" …

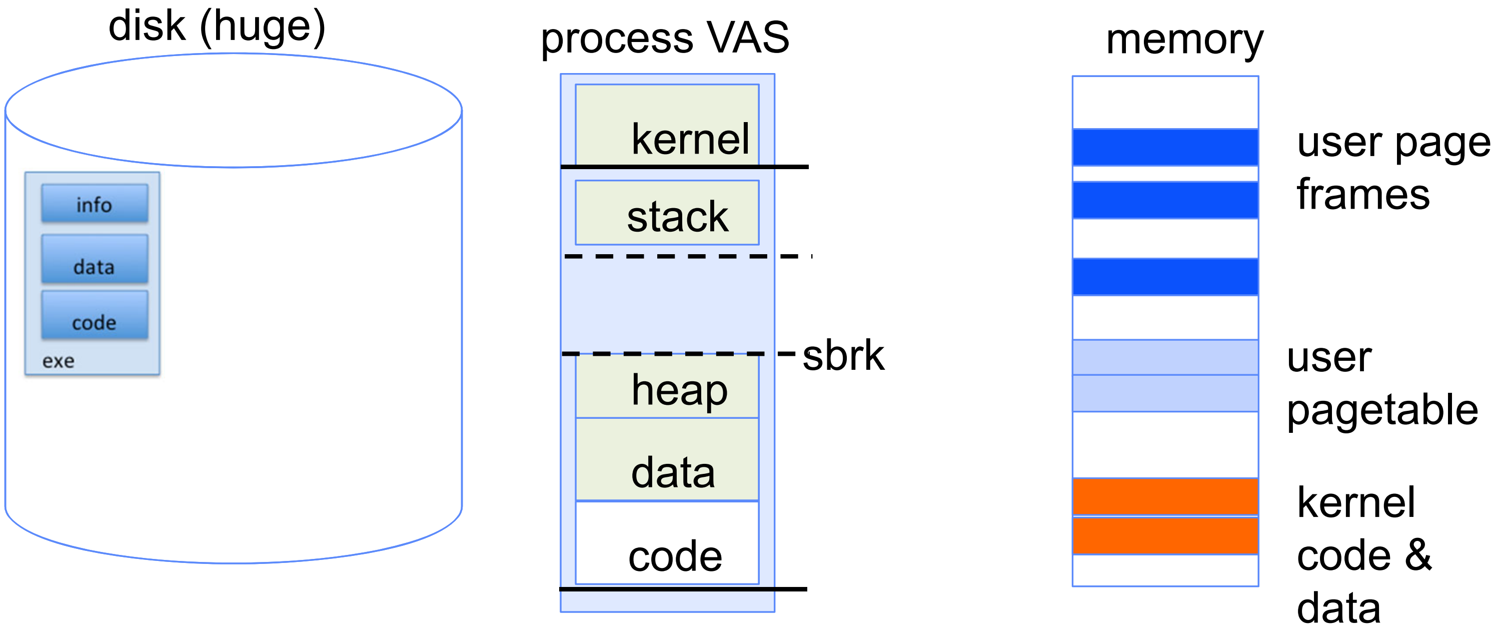
- Extend the stack
  - Allocate a page and zero it
- Extend the heap (sbrk of old, today mmap)
- Process Fork
  - Create a copy of the page table
  - Entries refer to parent pages – NO-WRITE
  - Shared read-only pages remain shared
  - Copy page on write
- Exec
  - Only bring in parts of the binary in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

Classic: Loading an executable into memory



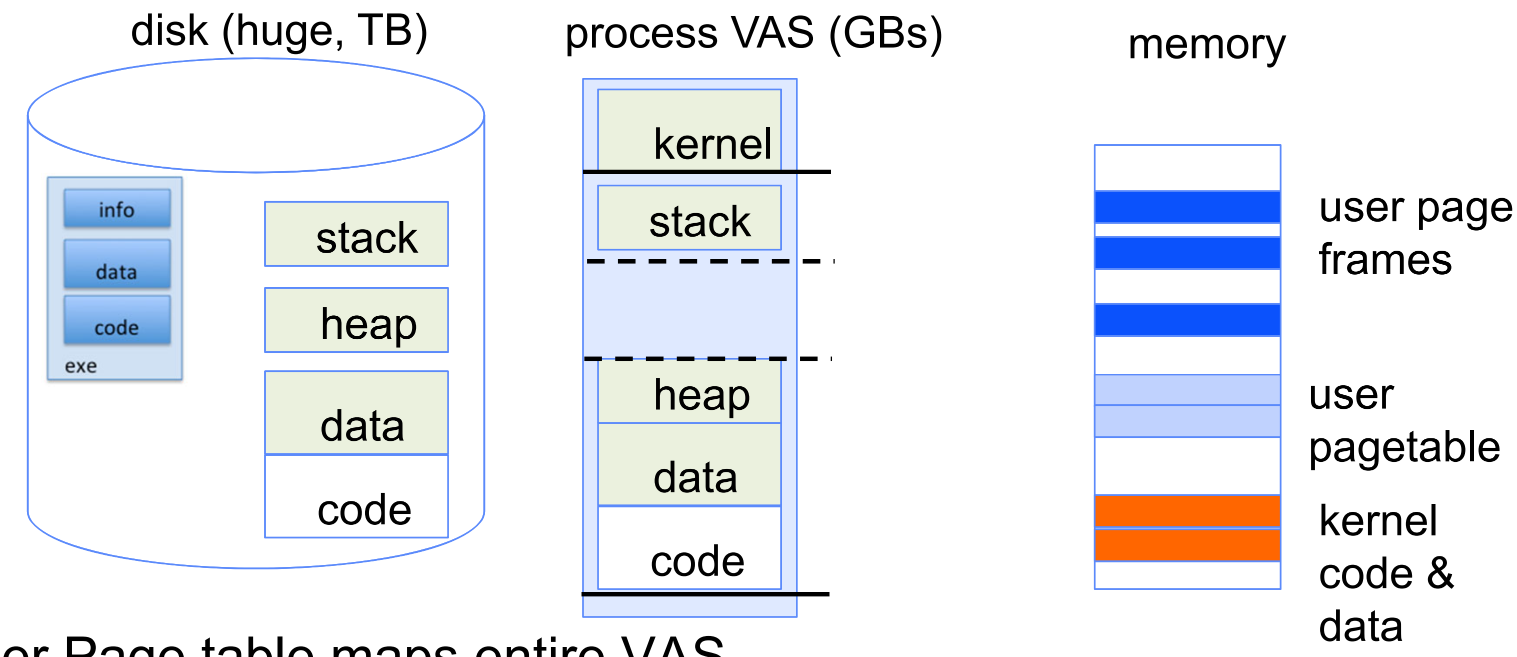
- .exe
  - lives on disk in the file system
  - contains contents of code & data segments, relocation entries and symbols
  - OS loads it into memory, initializes registers (and initial stack pointer)
  - program sets up stack and heap upon initialization: `crt0` (C runtime init)

Create Virtual Address Space of the Process



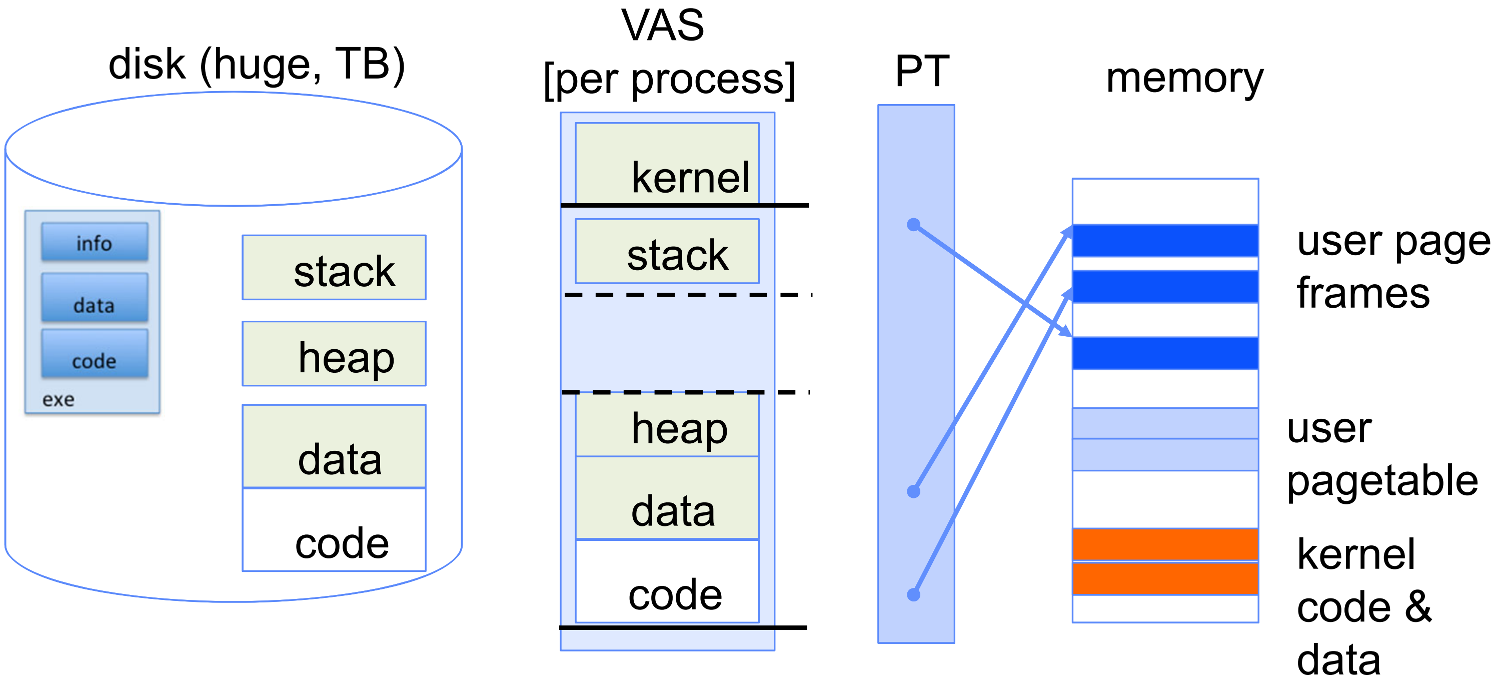
- Utilized pages in the VAS are backed by a page block on disk
  - Called the backing store or swap file
  - Typically in an optimized block store, but can think of it like a file

Create Virtual Address Space of the Process



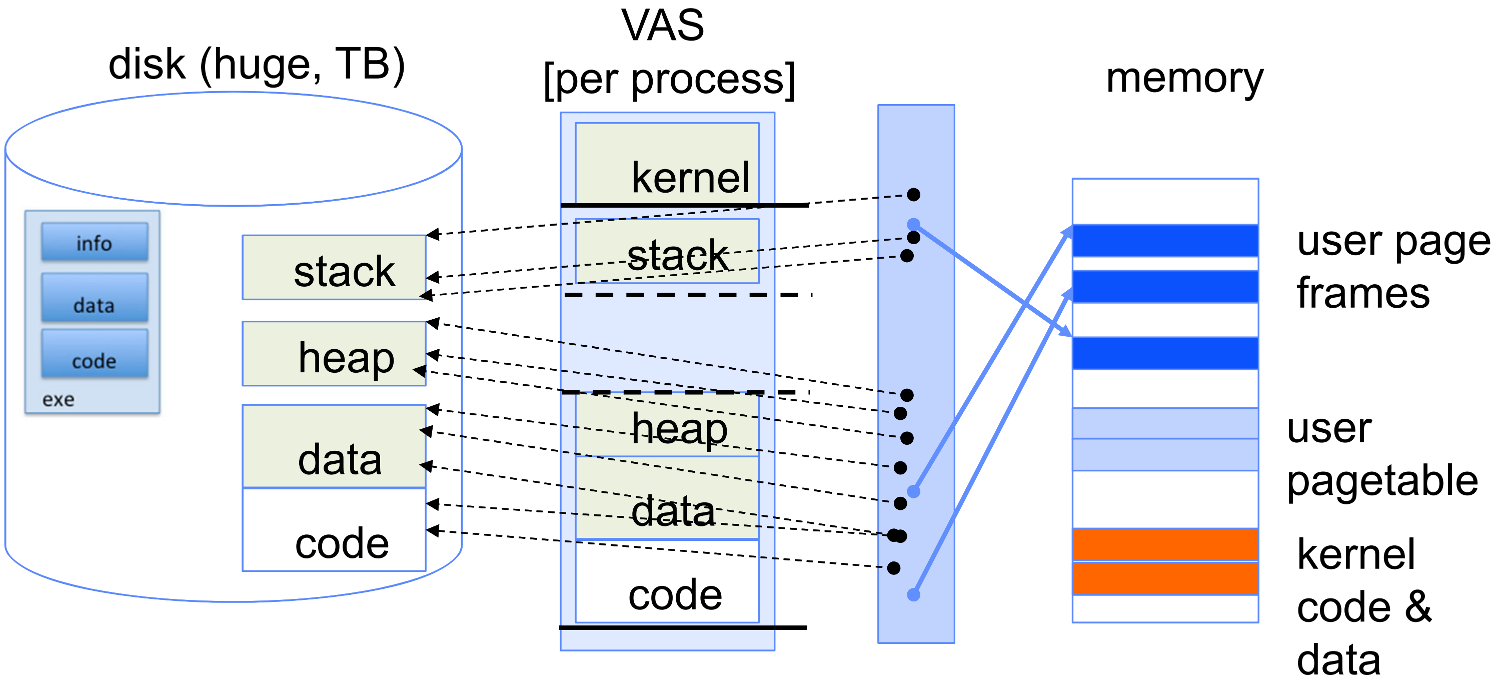
- User Page table maps entire VAS
- All the utilized regions are backed on disk
  - – swapped into and out of memory as needed
- For *every* process

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

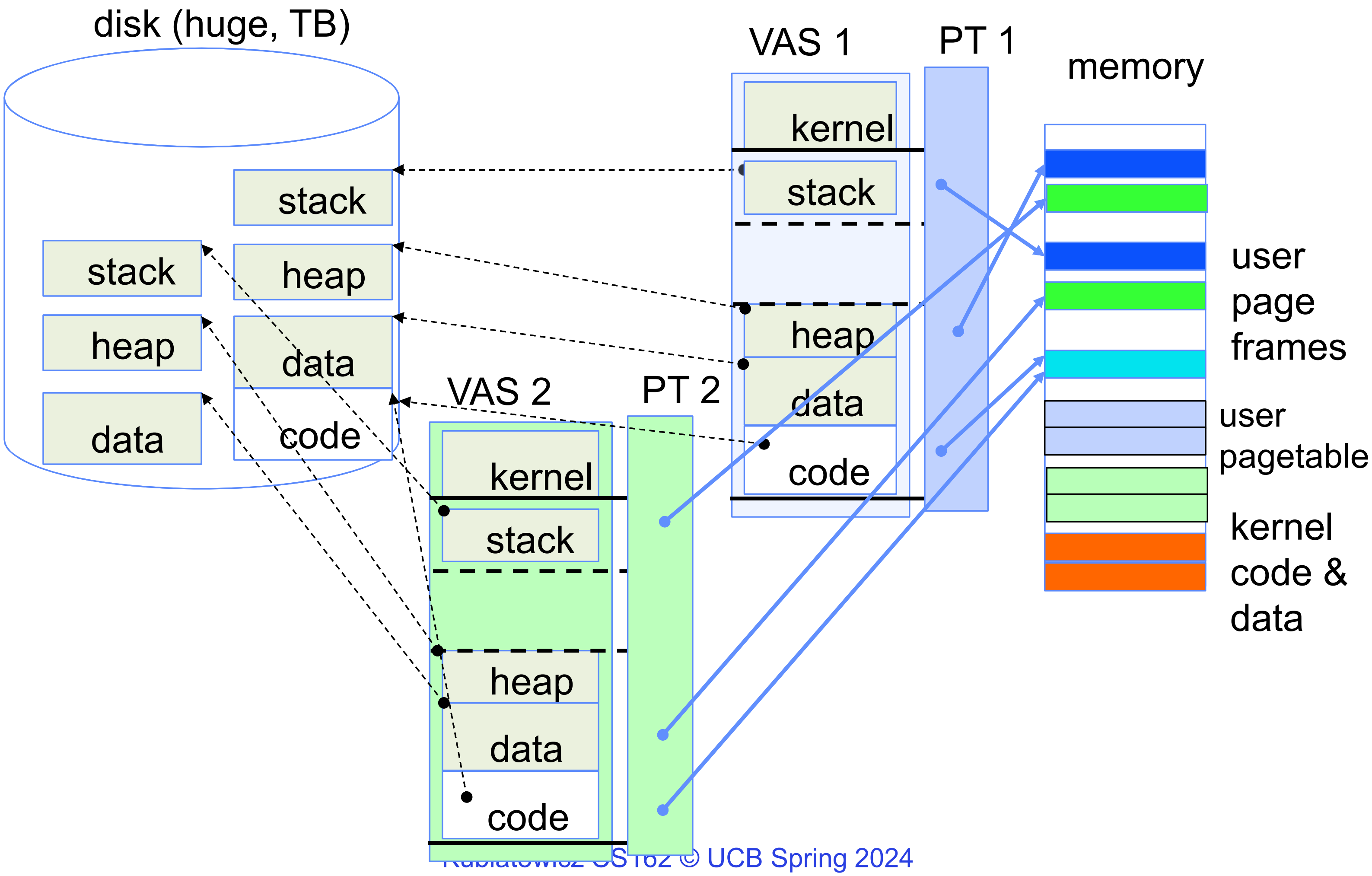


- User Page table maps entire VAS
- Resident pages mapped to memory frames
- For all other pages, OS must record where to find them on disk
  - – Many ways to do this, but might use remaining bits of PTE when P=0

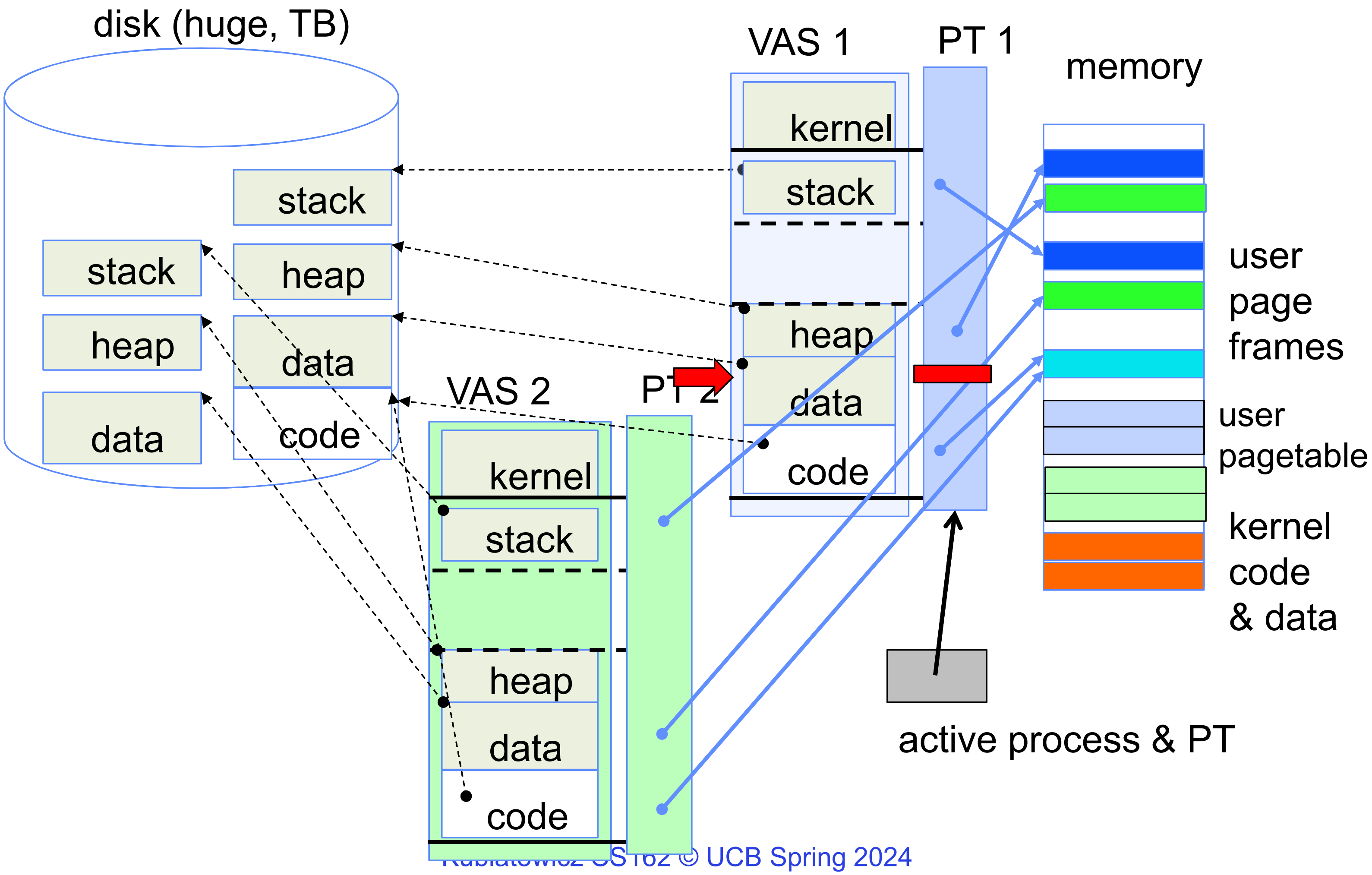
What Data Structure Maps Non-Resident Pages to Disk?

- `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

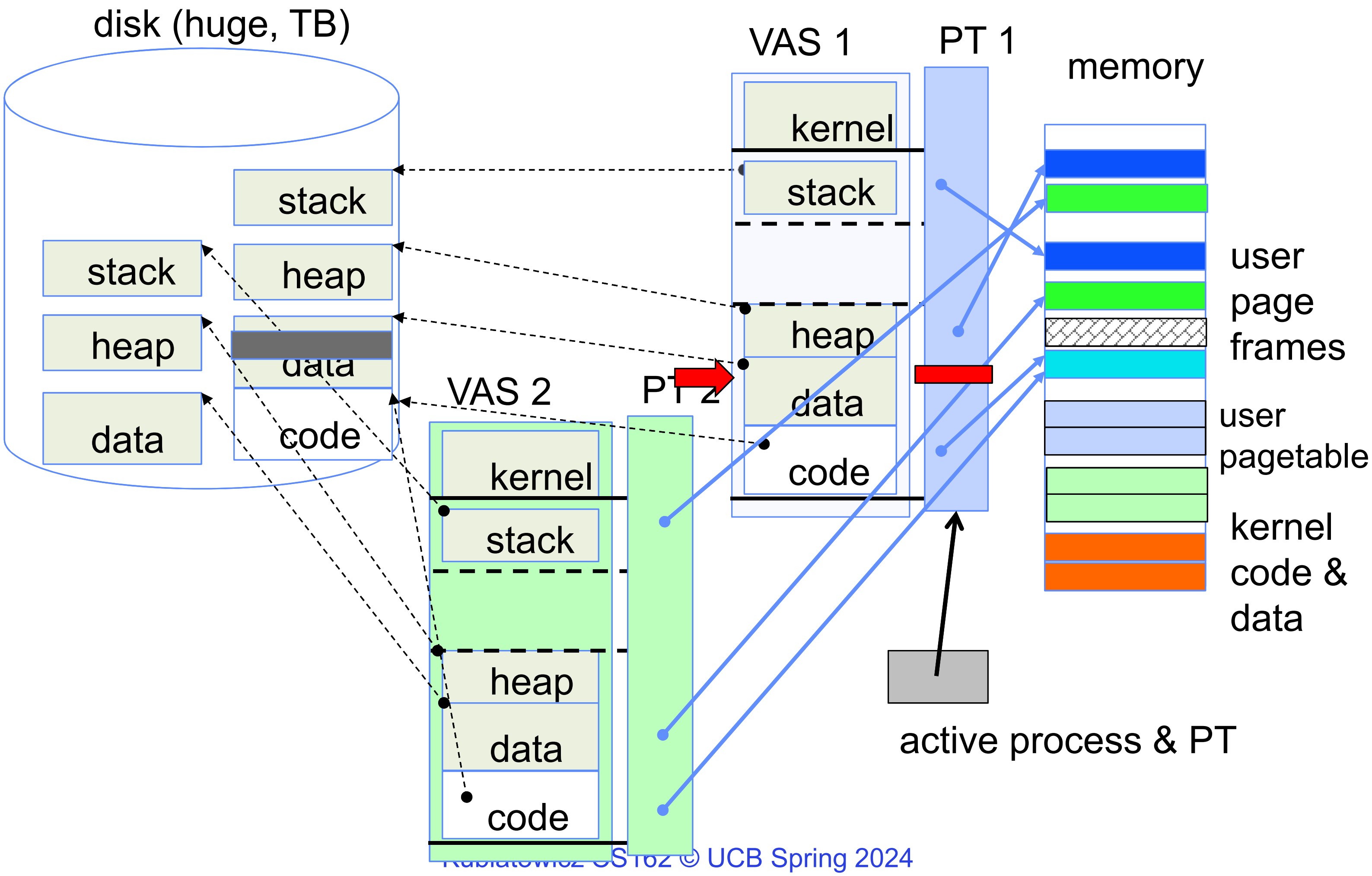
Provide Backing Store for VAS



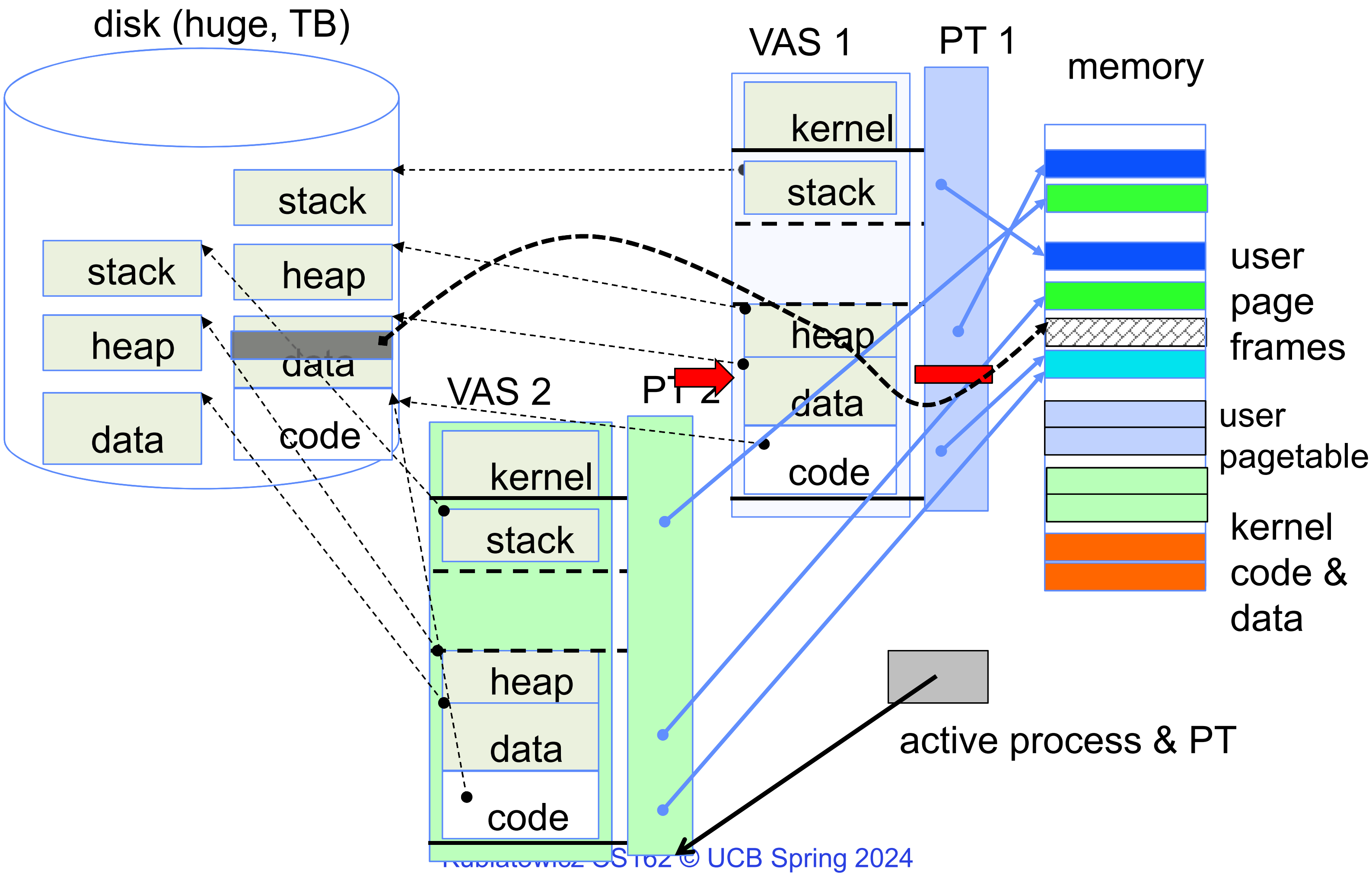
On page Fault …



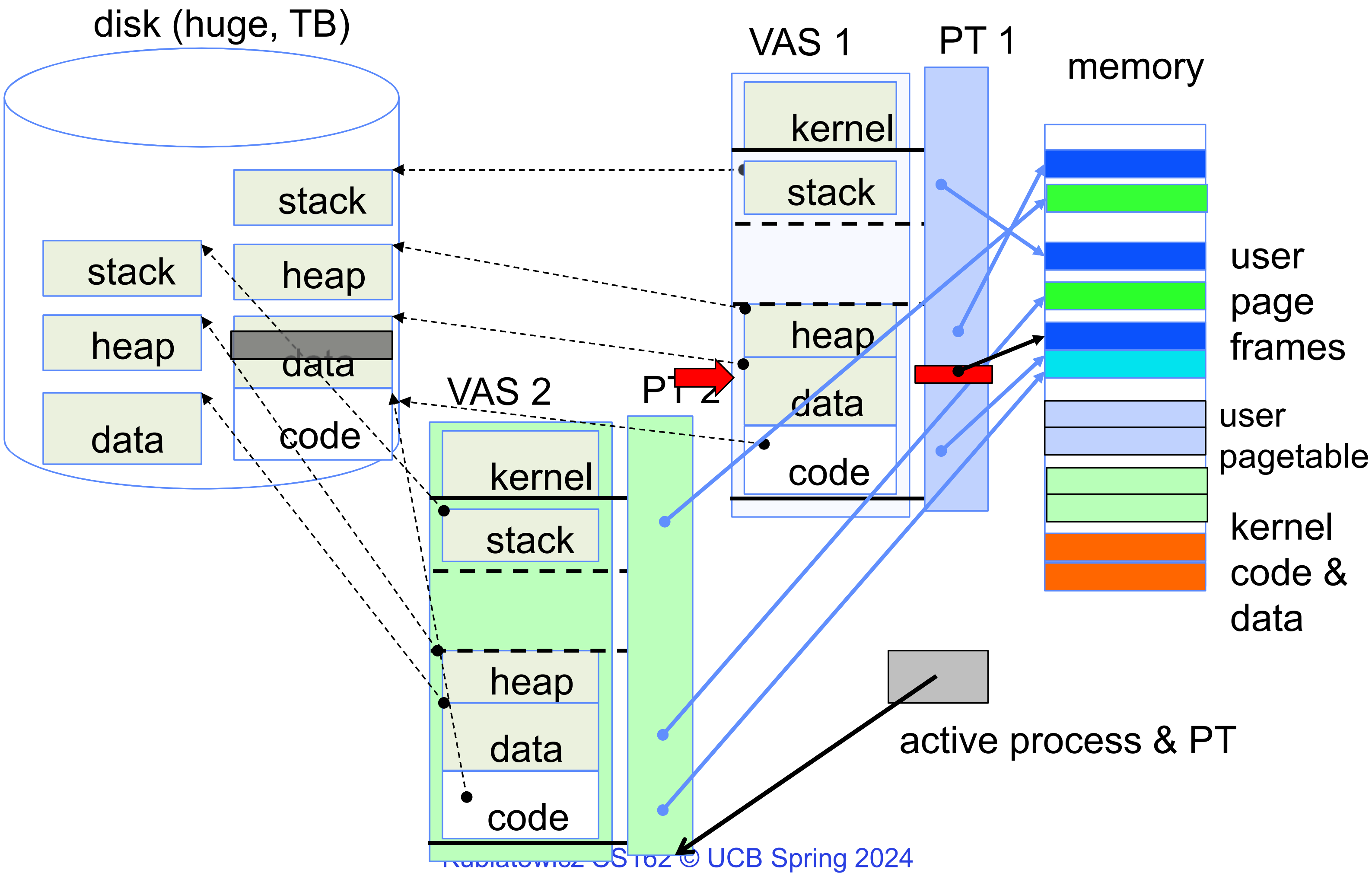
On page Fault … find & start load



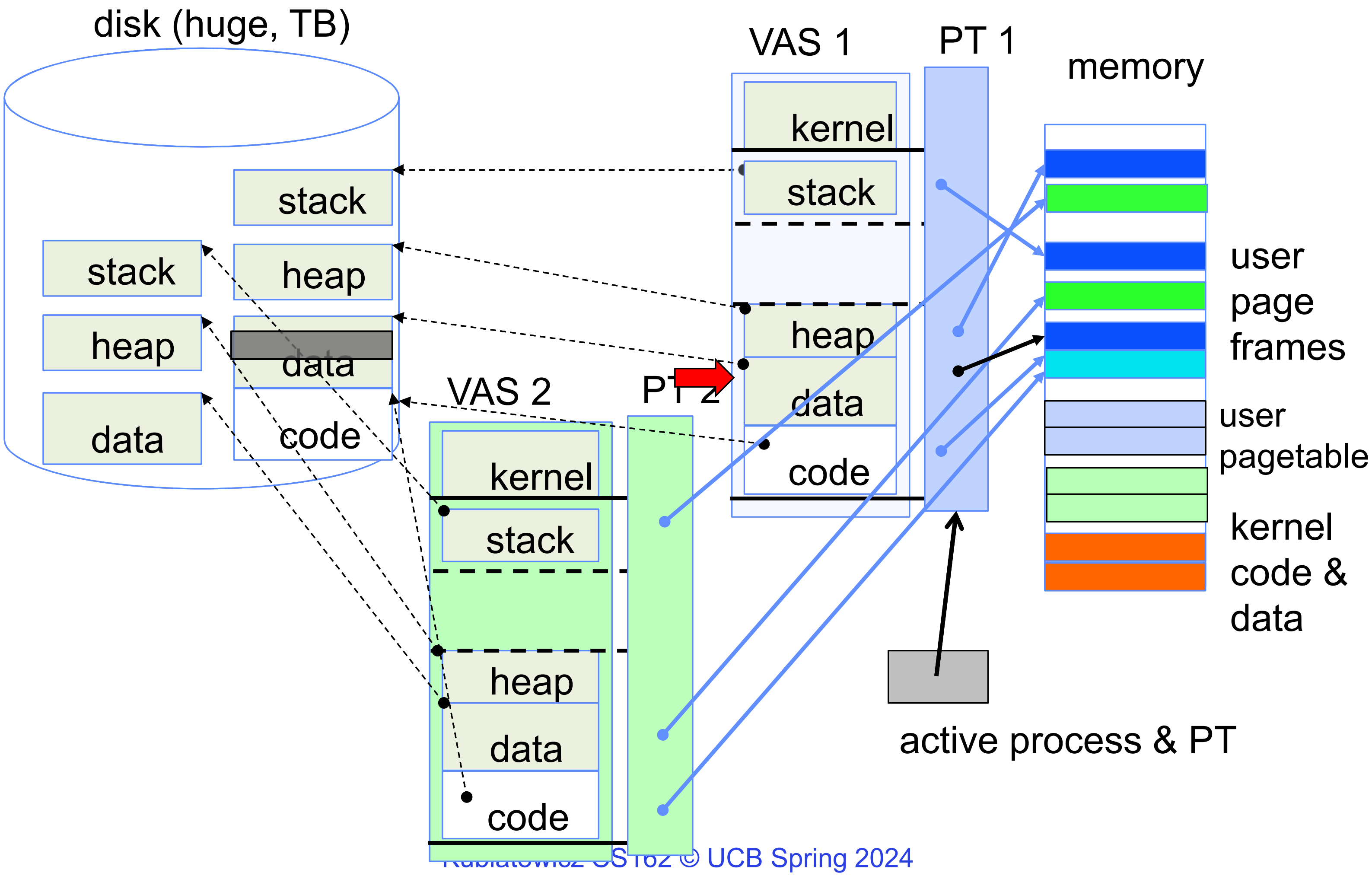
On page Fault … schedule other P or T



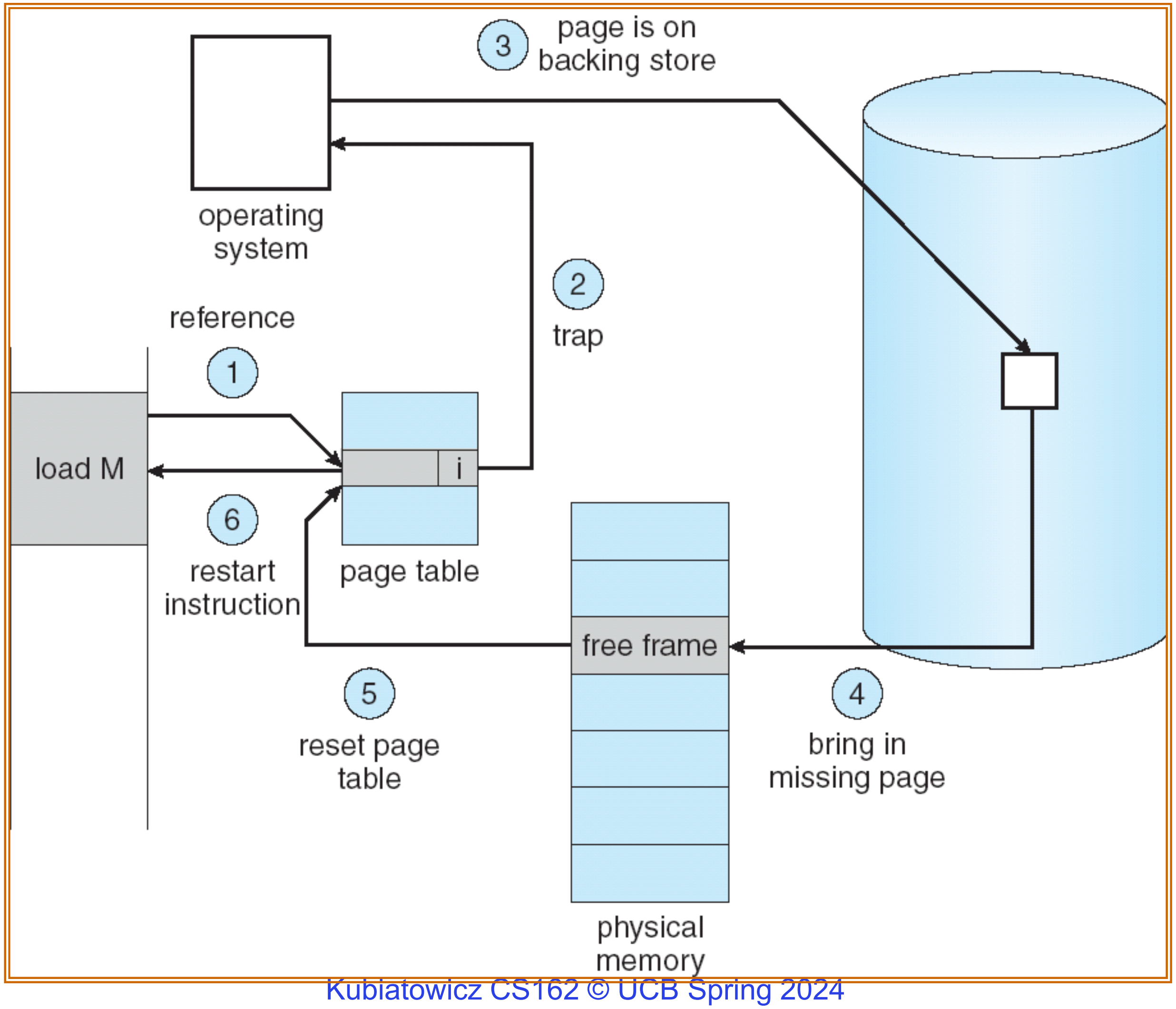
On page Fault … update PTE



Eventually reschedule faulting thread



Summary: Steps in Handling a Page Fault



Some questions we need to answer!

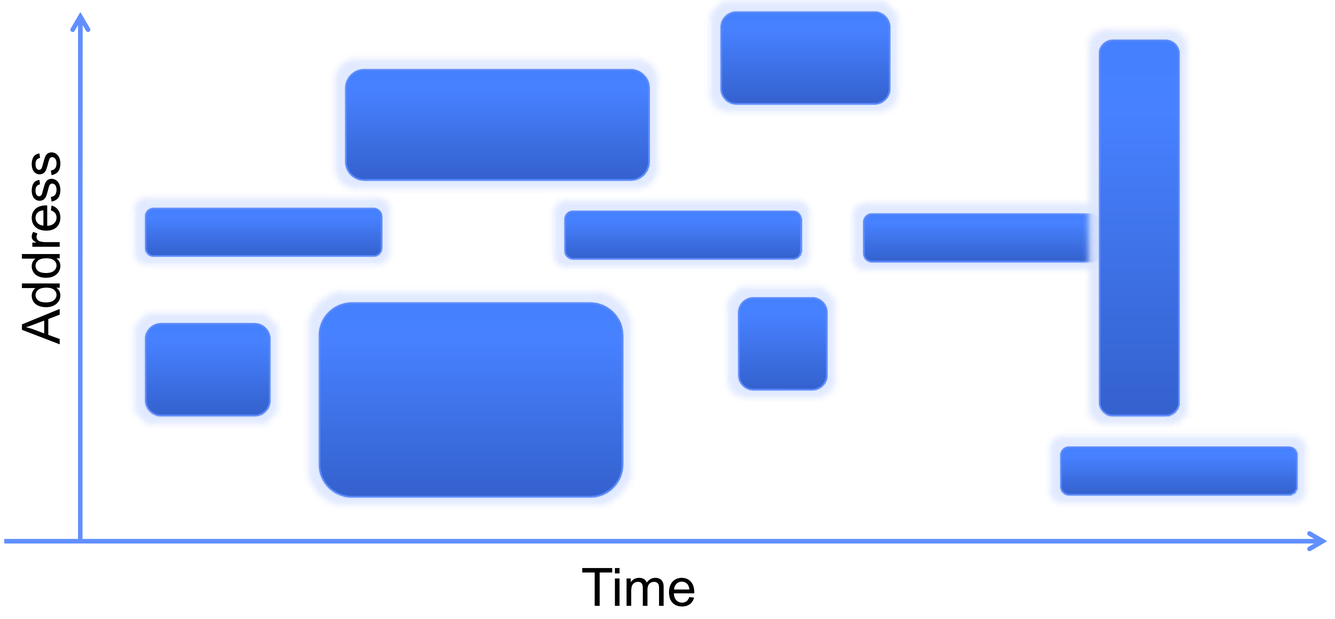
- During a page fault, where does the OS get a free frame?
  - Keeps a free list
  - Unix runs a “reaper” if memory gets too full
    - » Schedule dirty pages to be written back on disk
    - » Zero (clean) pages which haven’t been accessed in a while
  - As a last resort, evict a dirty page first

- How can we organize these mechanisms?
  - Work on the replacement policy

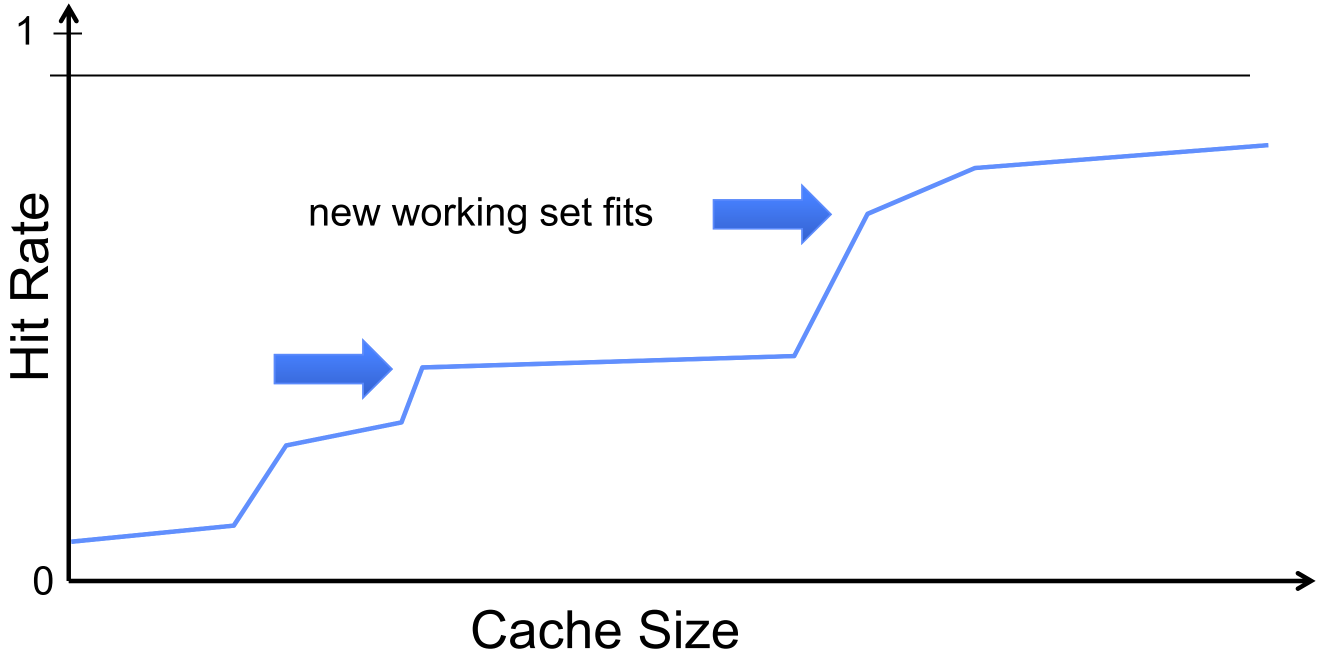
- How many page frames/process?
  - Like thread scheduling, need to “schedule” memory resources:
    - » Utilization? fairness? priority?
  - Allocation of disk paging bandwidth

Working Set Model

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

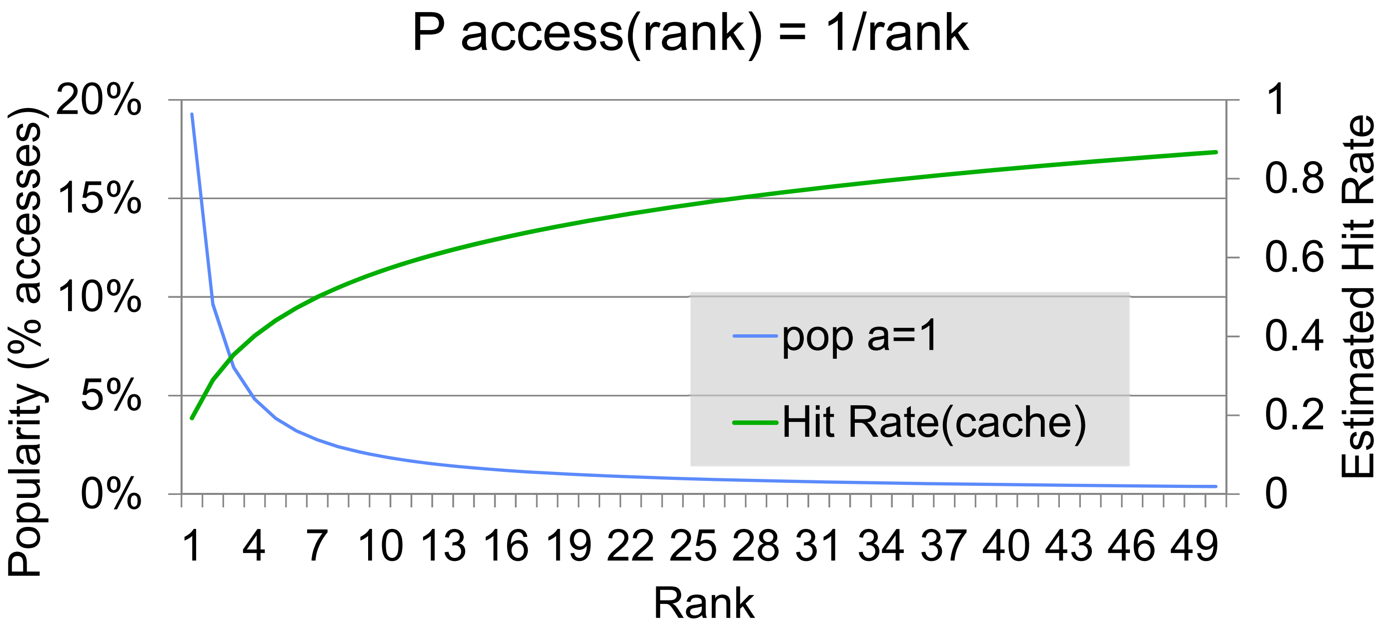


Cache Behavior under WS model



- 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:

$$\begin{aligned} \text{EAT} &= 200\text{ns} + p \times 8\text{ ms} \\ &= 200\text{ns} + p \times 8{,}000{,}000\text{ns} \end{aligned}$$

- 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%?
  - $\text{EAT} < 200\text{ns} \times 1.1 \Rightarrow p < 2.5 \times 10^{-6}$
  - This is about 1 page fault in 400,000!

What Factors Lead to Misses in Page Cache?

- 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 available memory 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 – 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

- “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