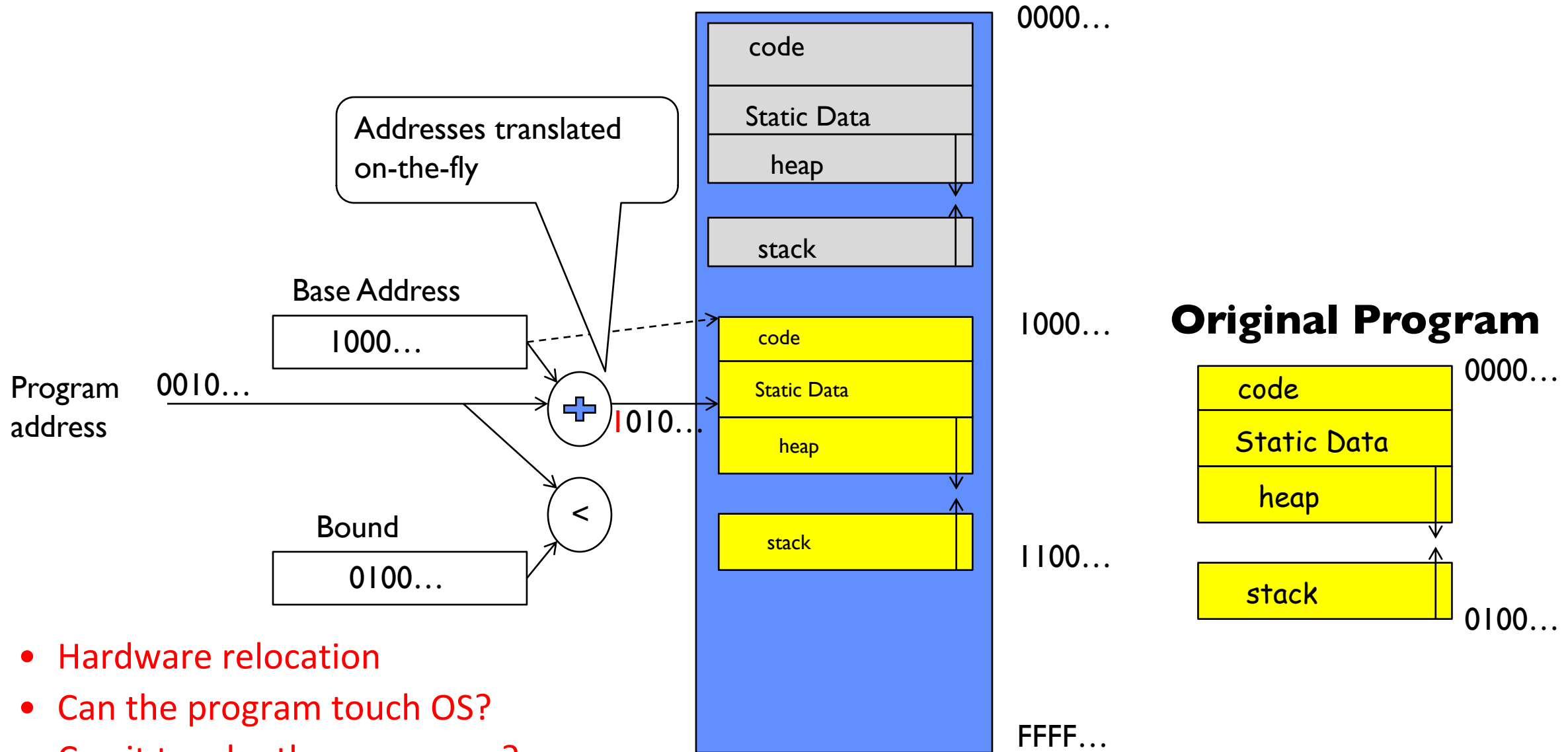# Operating Systems
(Honor Track)

Memory 3: Demand Paging

Xin Jin

Spring 2023

Recap: Base and Bound (with Translation)
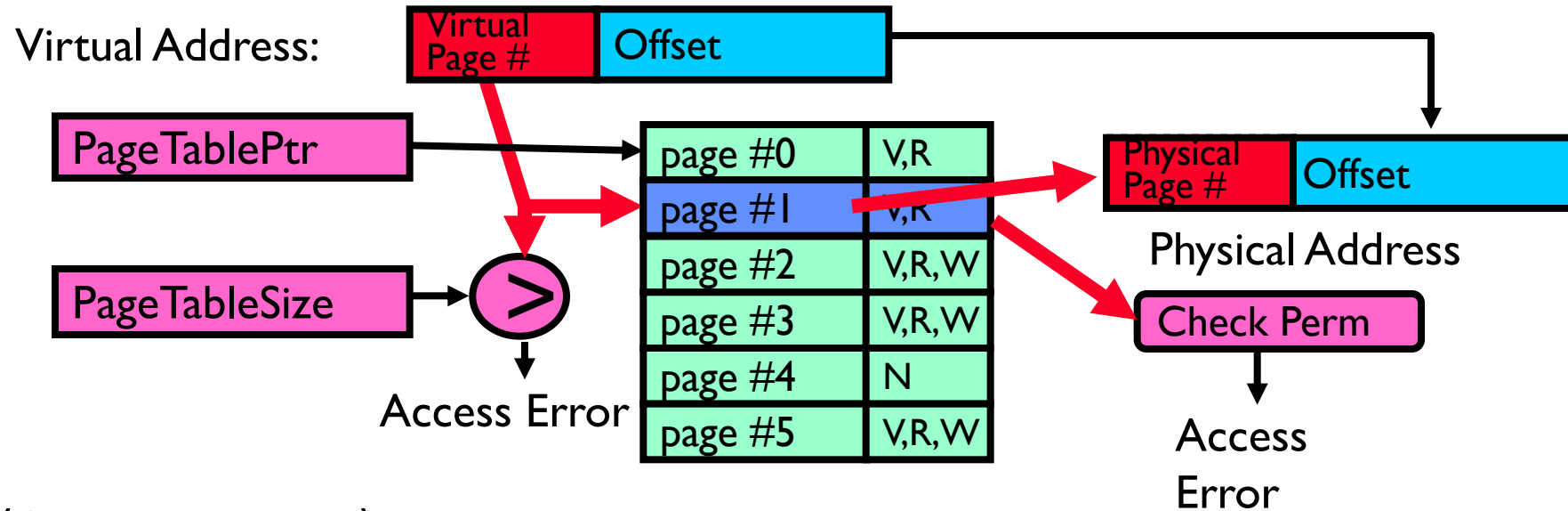
Addresses translated on-the-fly

code

Static Data

heap

stack

0000…

Base Address

1000…

Program address

0010…

+

1010…

<

Bound

0100…

code

Static Data

heap

stack

1000…

1100…

FFFF…

Original Program

code

Static Data

heap

stack

0000…

0100…

- Hardware relocation
- Can the program touch OS?
- Can it touch other programs?

2

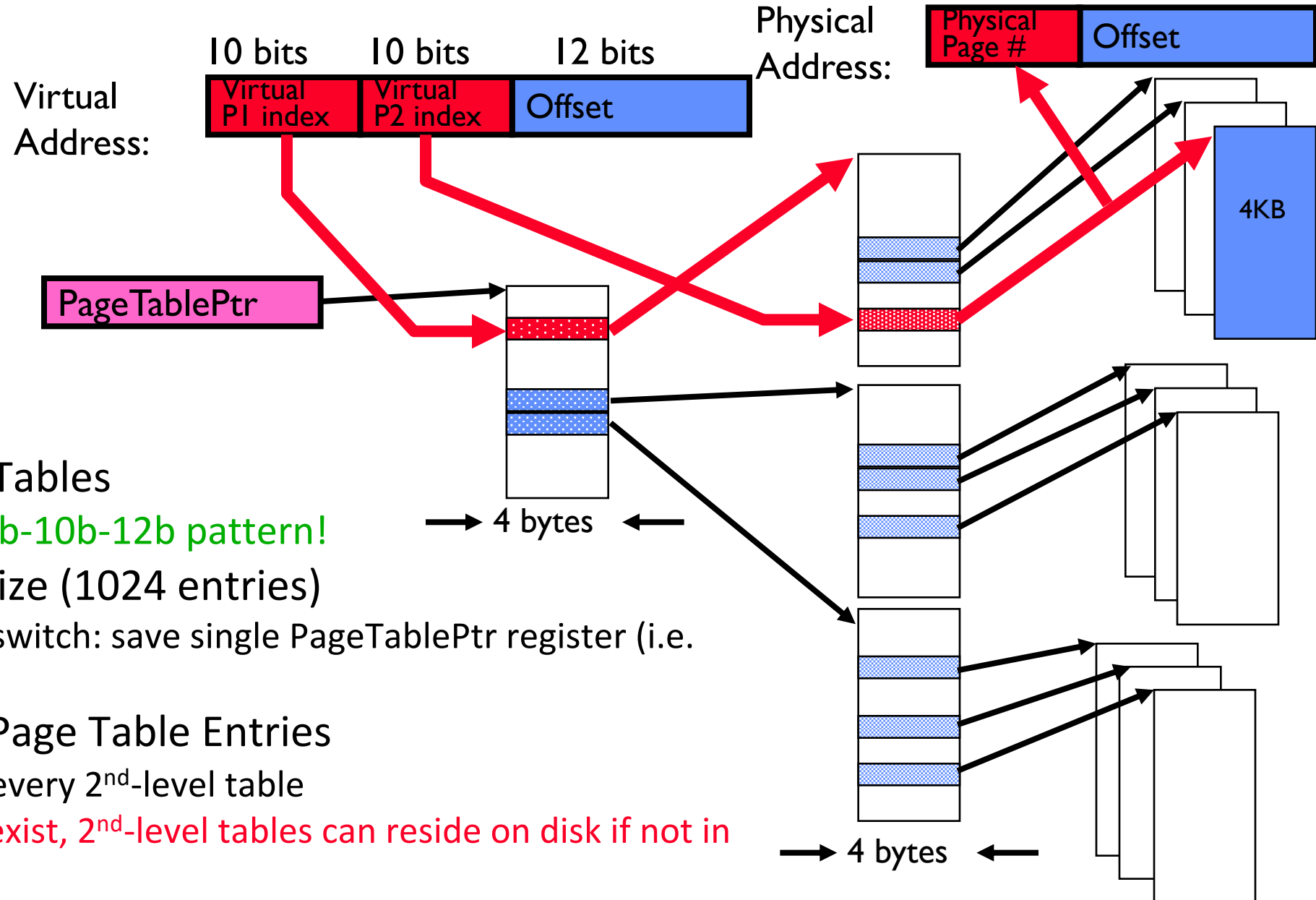# Recap: Implementation of Multi-Segment Model
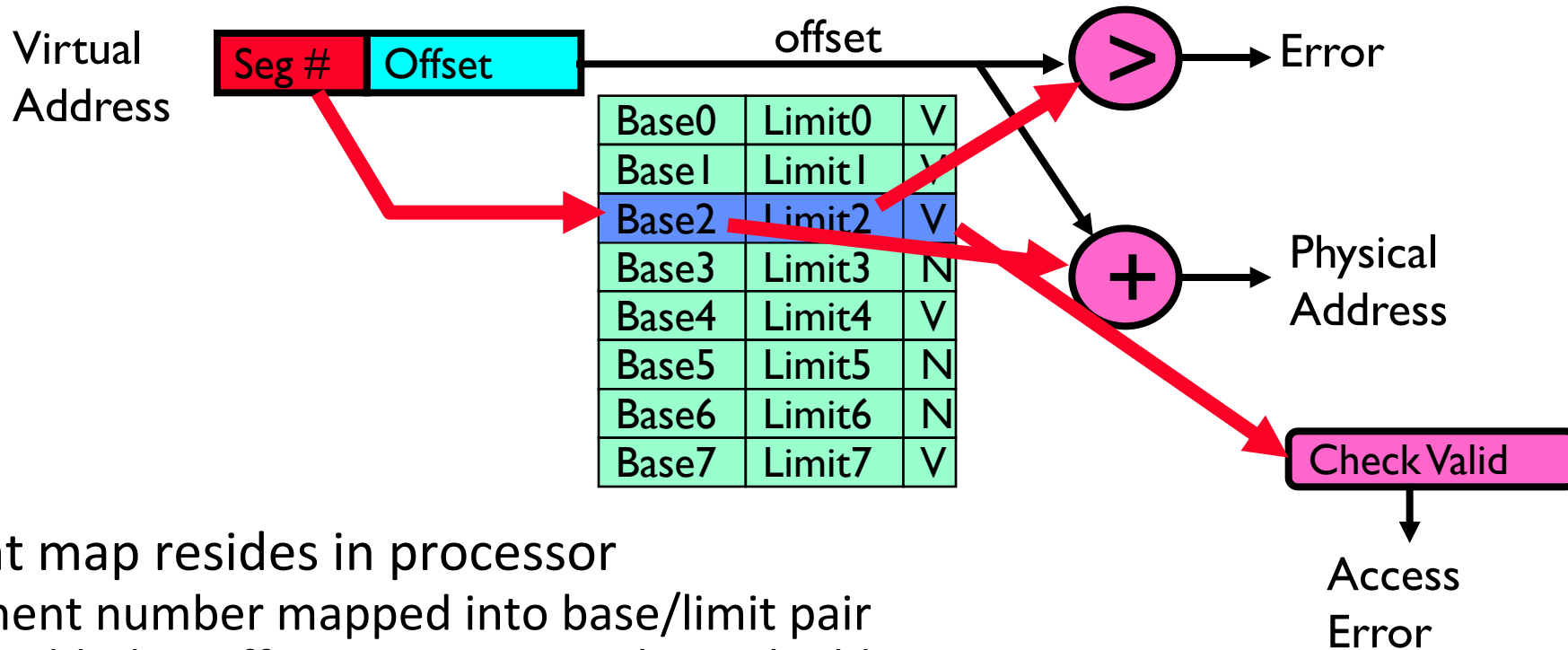
- **Segment map resides in processor**
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- **As many chunks of physical memory as entries**
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    » x86 Example: mov [es:bx],ax.
- **What is "V/N" (valid / not valid)?**
  - Can mark segments as invalid; requires check as well

Recap: How to Implement Simple Paging?

Virtual Address:

Virtual Page #	Offset

PageTablePtr

PageTableSize

>

Access Error

page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

Physical Page #	Offset

Physical Address

Check Perm

Access Error

- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc.)
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    » Example: 10 bit offset $\Rightarrow$ 1024-byte pages
  - Virtual page # is all remaining bits
    » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    » Physical page # copied from table into physical address
 - Check Page Table bounds and permissions

Recap: The two-level page table

Virtual Address:

10 bits — Virtual P1 index
10 bits — Virtual P2 index
12 bits — Offset

Physical Address: Physical Page # | Offset

PageTablePtr

4 bytes

4 bytes

4KB

- 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

5

# Recap: Multi-level Translation: Segments + Pages

- What about a tree of tables?
  - Lowest level page table $\Rightarrow$ memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):
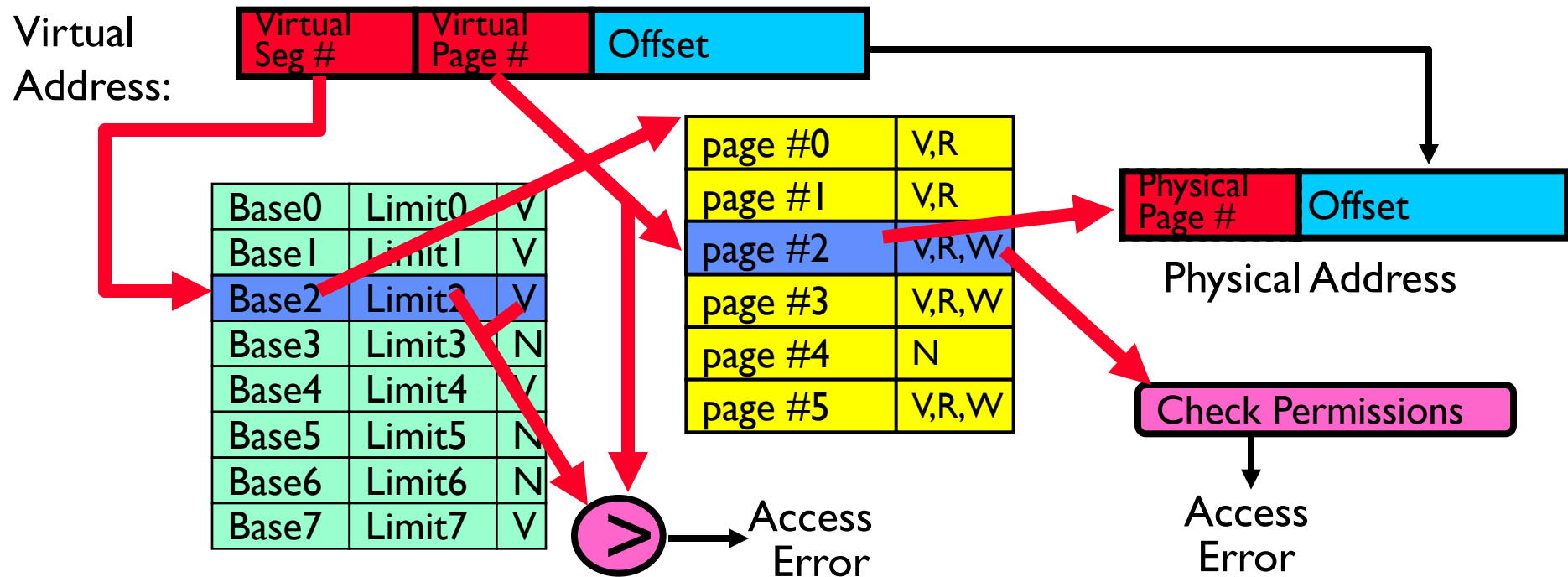
- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

Recap: Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - » Much of process space may be out on disk or not in use
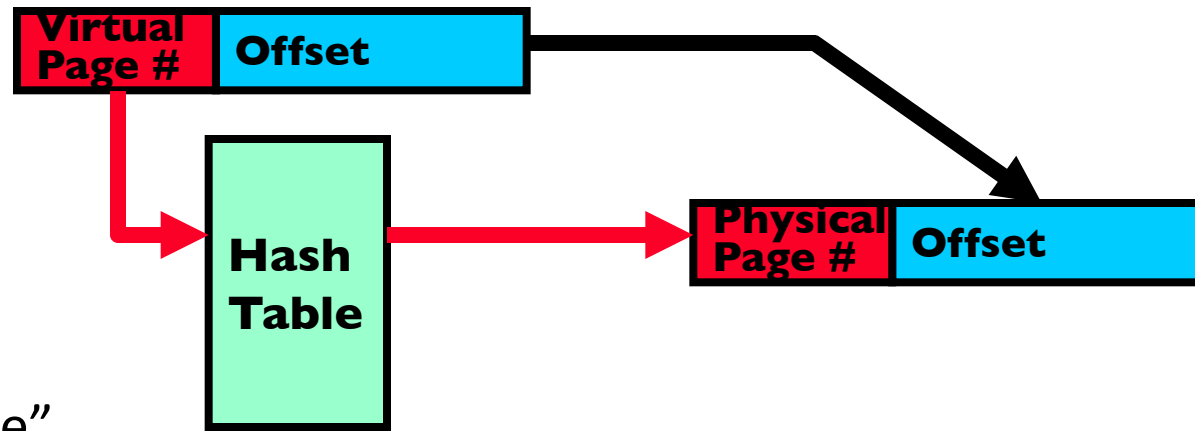
- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address s
  - Directly related to amount of physical
  - Very attractive option for 64-bit addre
    - » PowerPC, UltraSPARC, IA64

Total size of page table ≈ number of pages used by program in physical memory. Hash more complex
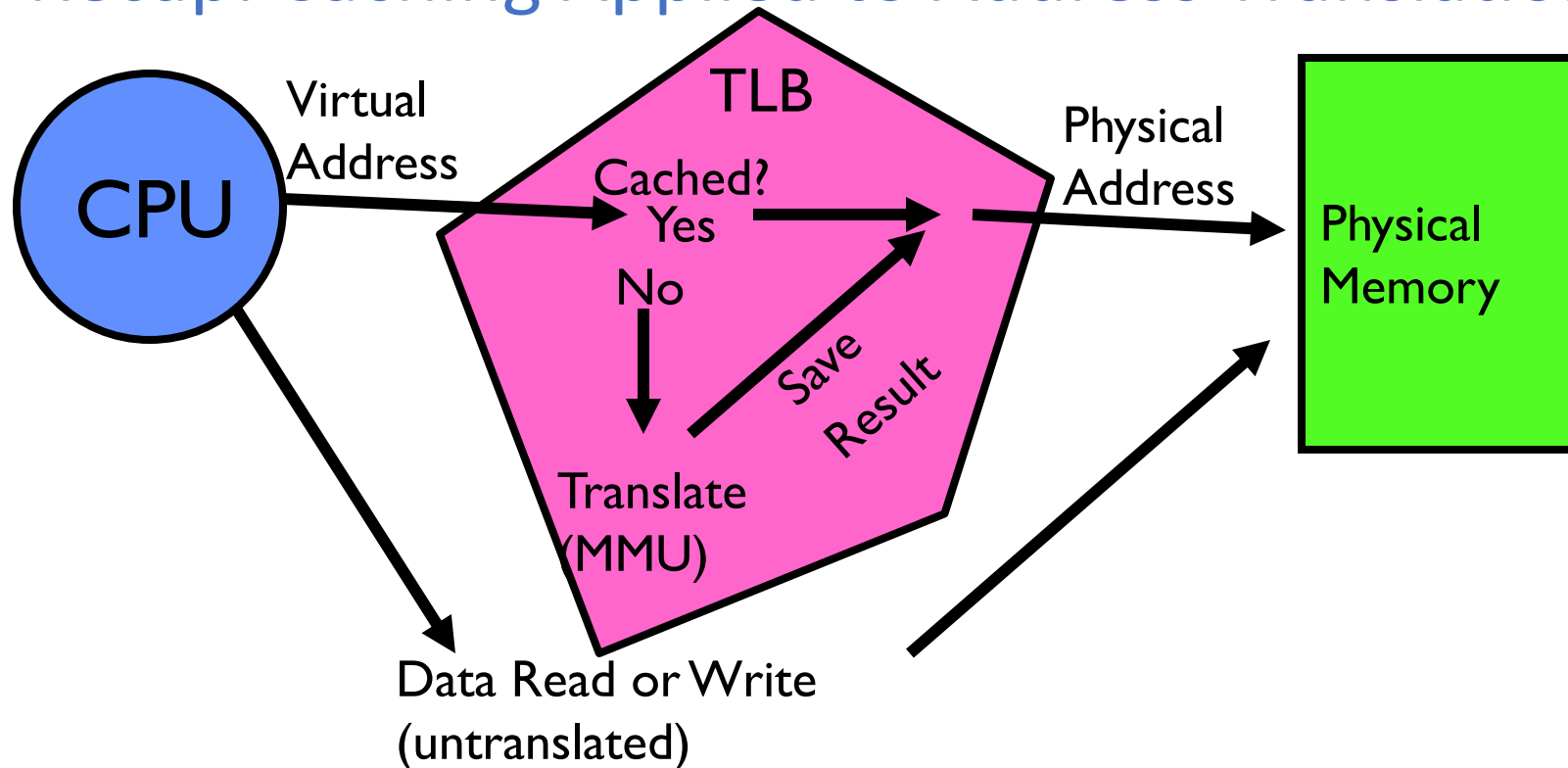
- Cons:
  - Complexity of managing hash chains: Often in hardware!
  - Poor cache locality of page table

Recap: Address Translation Comparison

	Advantages	Disadvantages
Simple Segmentation	Fast context switching (segment map maintained by CPU)	Internal/External fragmentation
Paging (Single-Level)	No external fragmentation Fast and easy allocation	Large table size (~ virtual memory) Internal fragmentation
Paged Segmentation	Table size ~ # of pages in virtual memory Fast and easy allocation	Multiple memory references per page access
Multi-Level Paging		
Inverted Page Table	Table size ~ # of pages in physical memory	Hash function more complex No cache locality of page table

Recap: Caching Applied to Address Translation

CPU

Virtual Address

TLB

Cached? Yes

No

Translate (MMU)

Save Result

Physical Address

Physical Memory

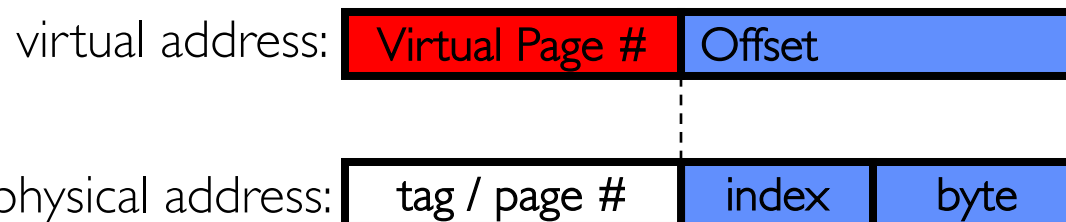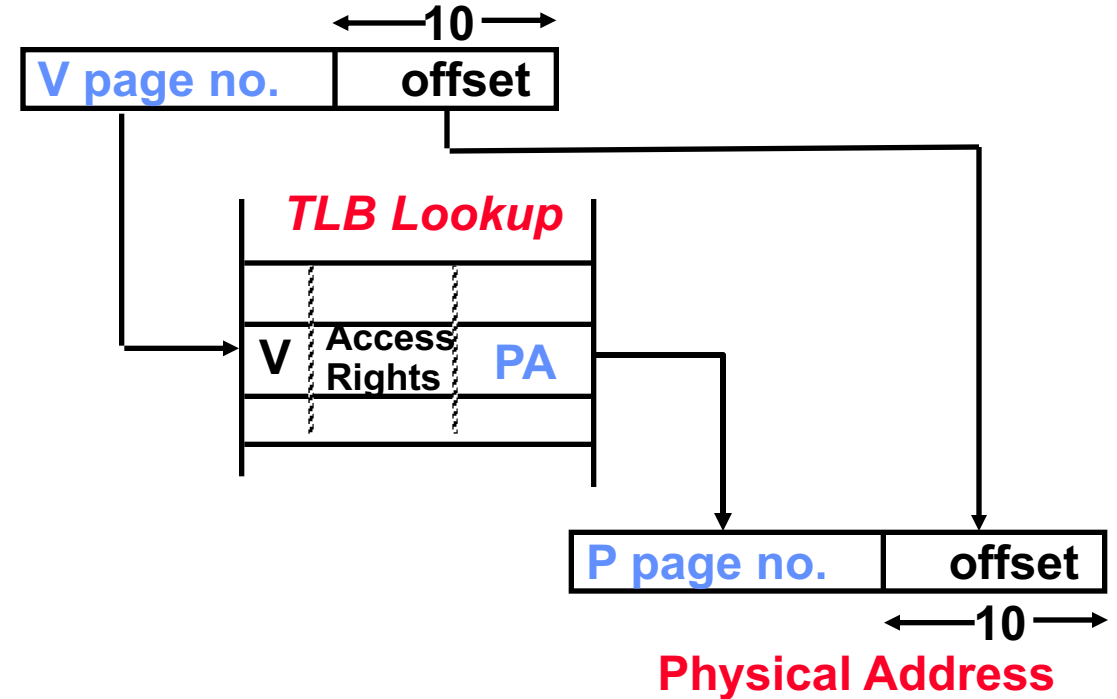Data Read or Write (untranslated)

- 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

Recap: 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

Virtual Address

←—10—→

| V page no. | offset |

TLB Lookup

| V | Access Rights | PA |

| P page no. | offset |

←—10—→

Physical Address

virtual address: | Virtual Page # | Offset |

physical address: | tag / page # | index | byte |
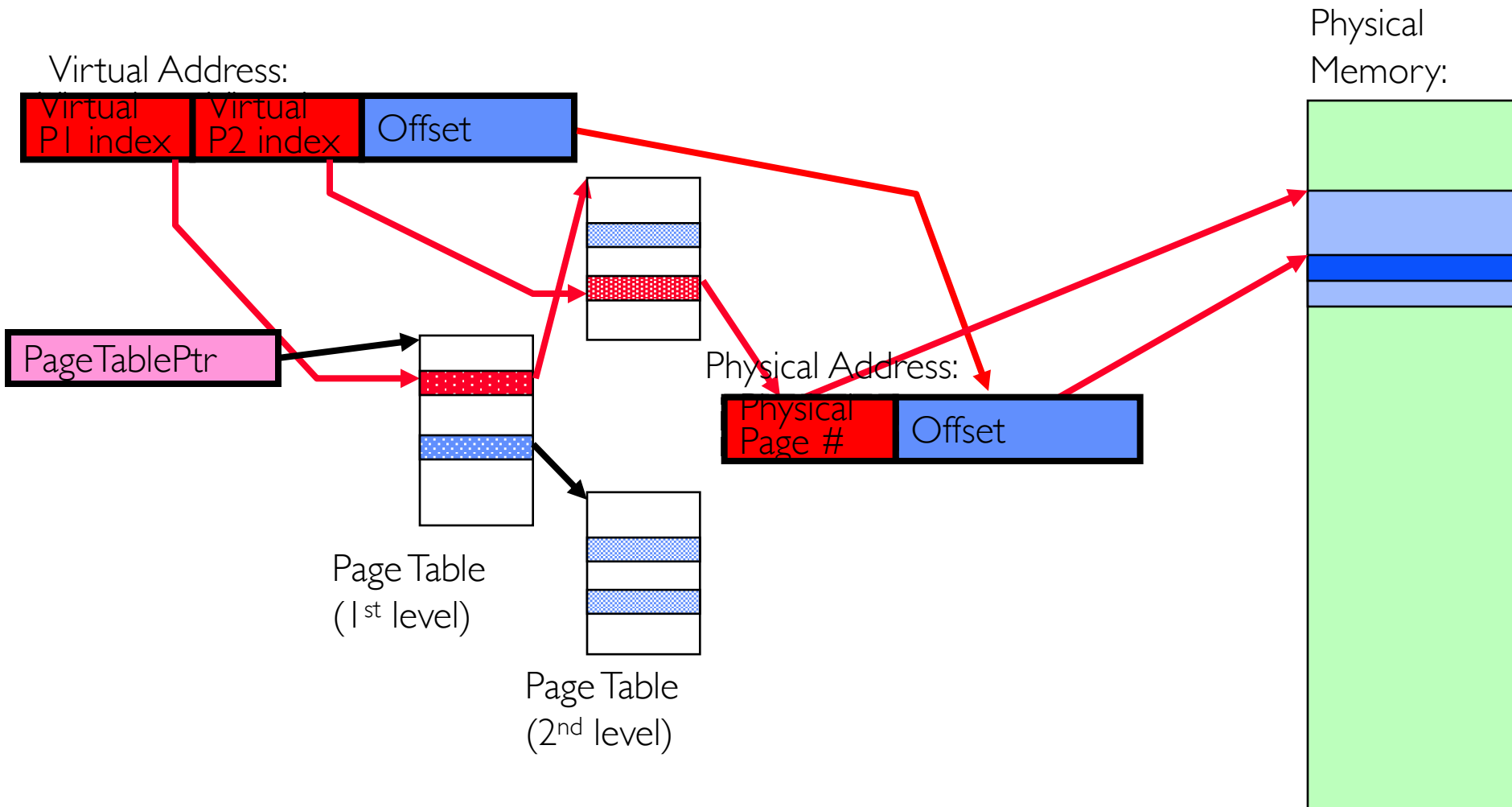
Recap: Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:
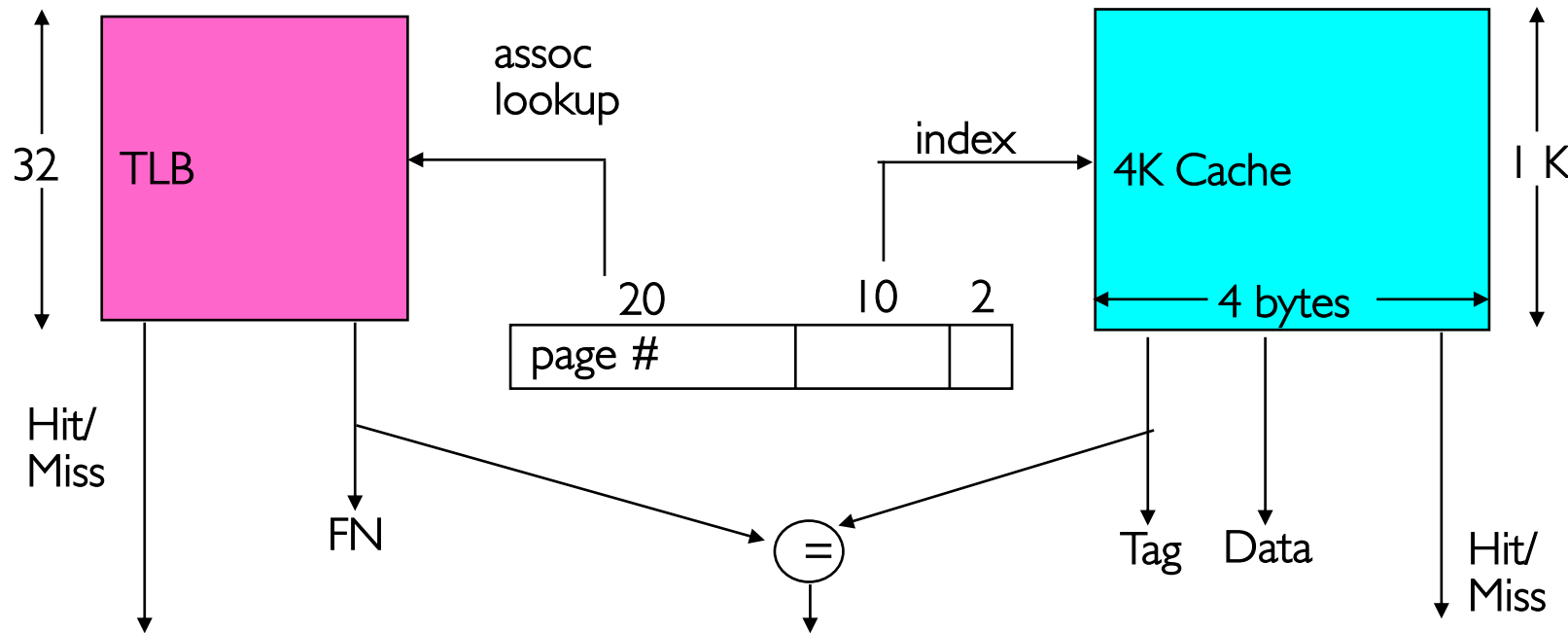
- **What if cache size is increased to 8KB?**
  - Overlap not complete
  - Need to do something else
- **Another option: Virtual Caches would make this faster**
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

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:
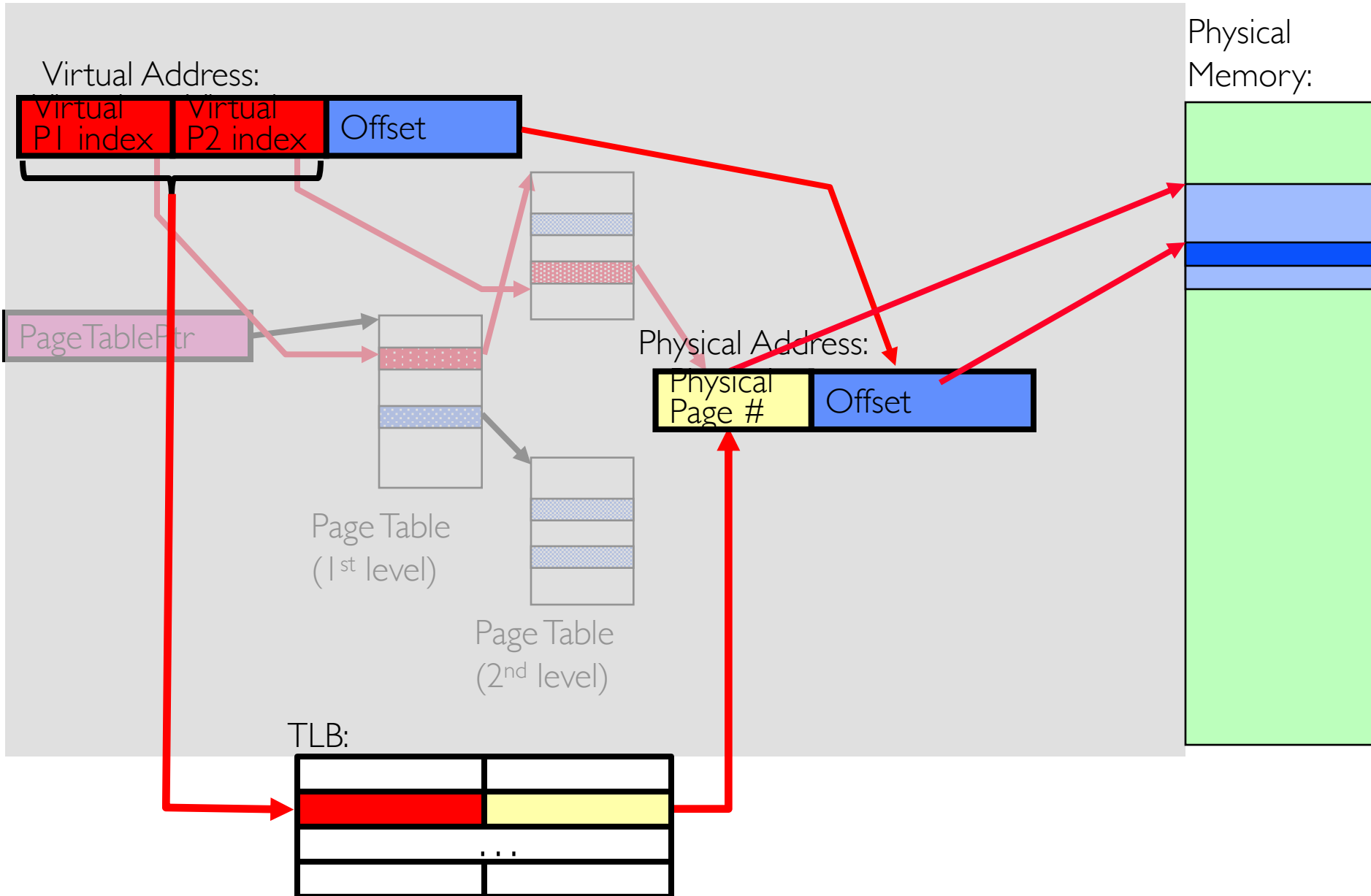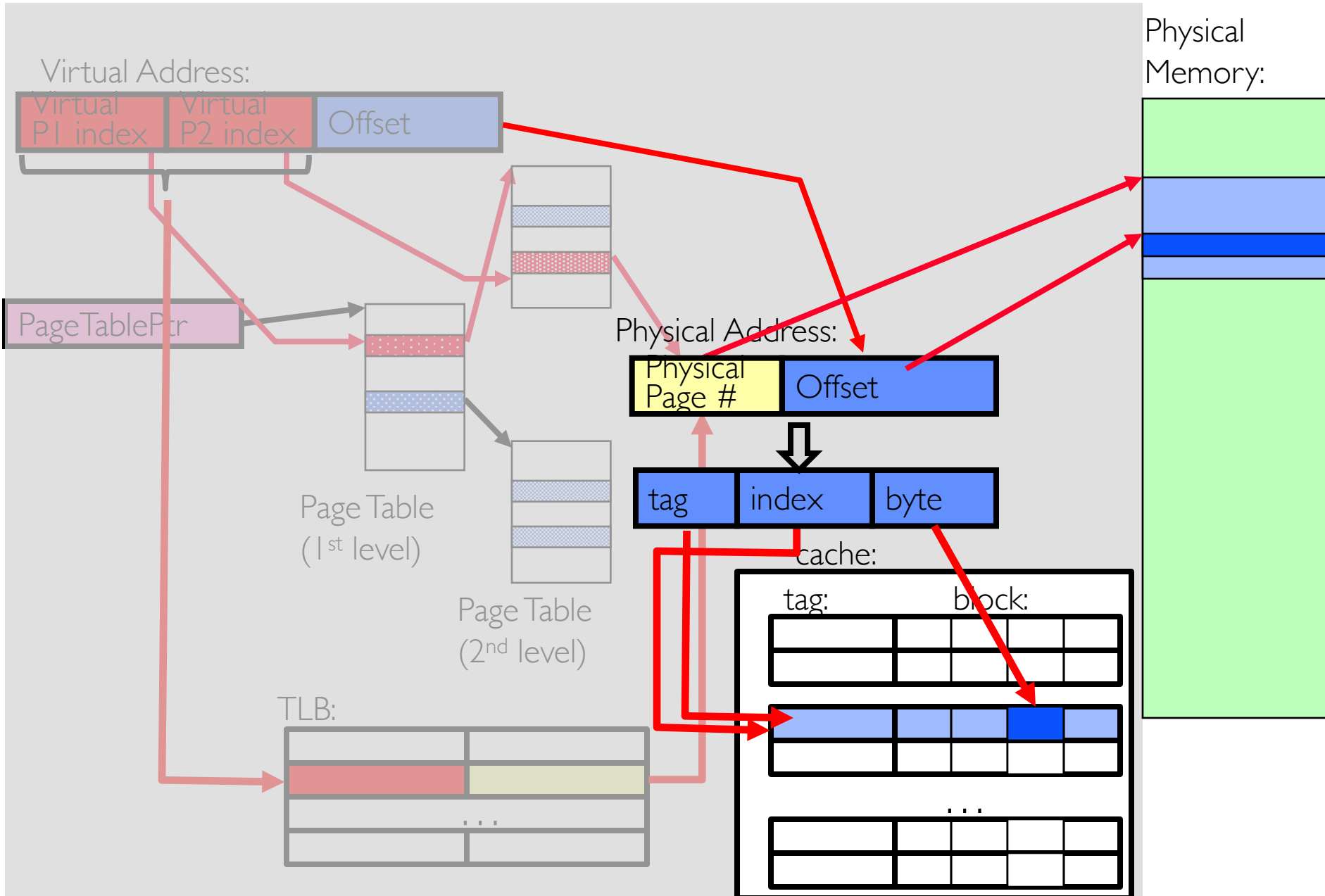
Page Fault

- The Virtual-to-Physical Translation fails
  - PTE marked invalid, Privilege Level Violation, Access violation, or does not exist
  - Causes a Fault / Trap
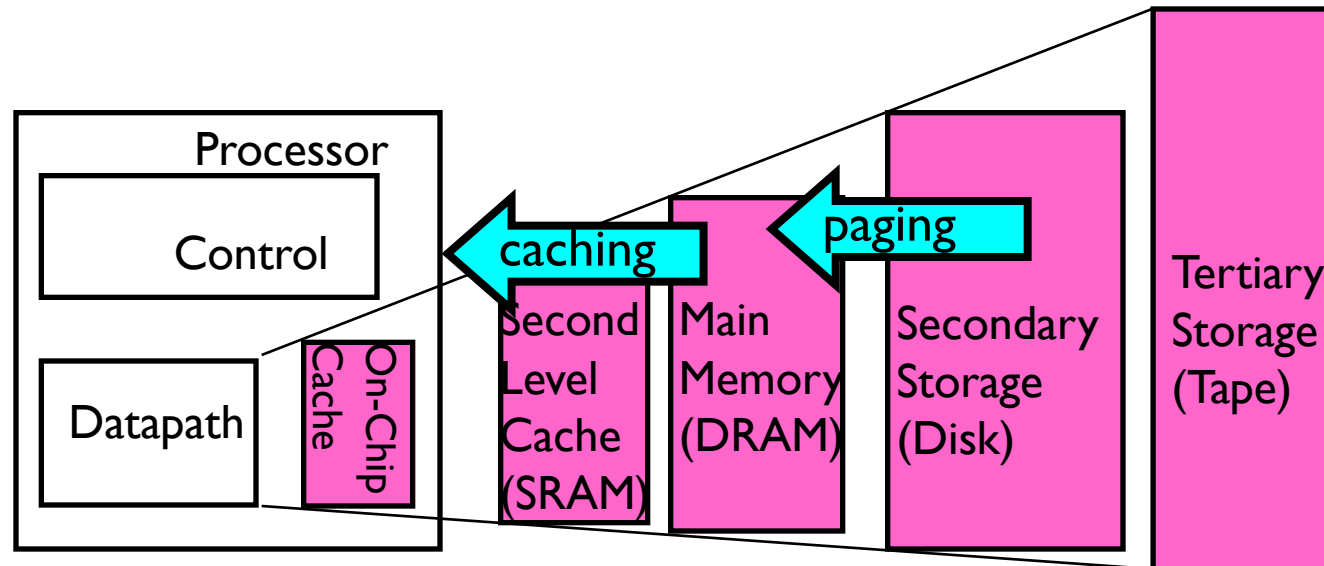   - » Not an interrupt because synchronous to instruction execution
  - May occur on instruction fetch or data access
- Protection violations typically terminate the instruction
- Other Page Faults engage operating system to fix the situation and retry the instruction
  - Allocate an additional stack page, or
  - Make the page accessible - Copy on Write,
  - Bring page in from secondary storage to memory – demand paging
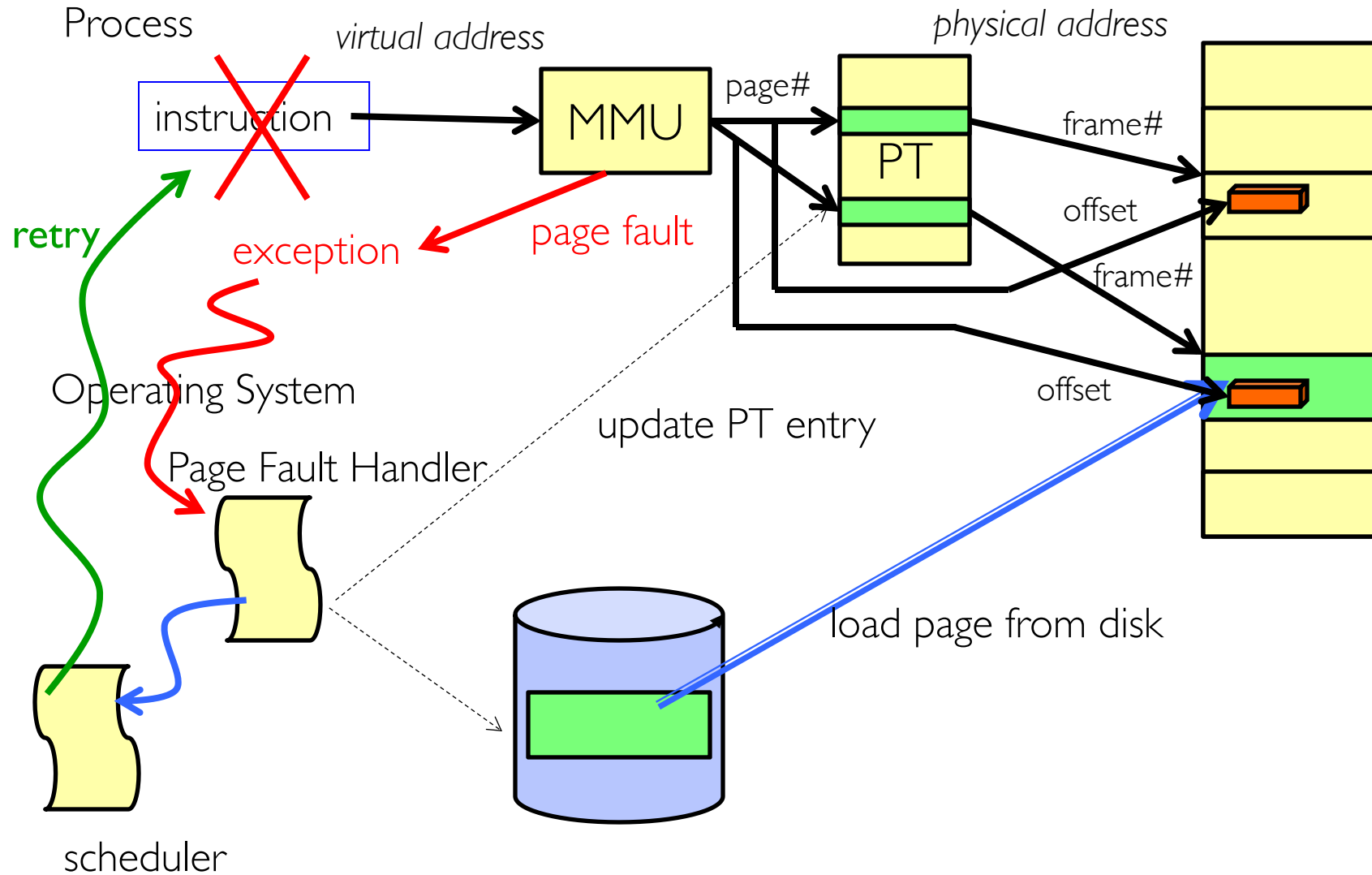- Fundamental inversion of the hardware / software boundary

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

Process   *virtual address*   *physical address*

instruction   →   MMU   page#   PT   frame#

retry   exception   page fault   offset   frame#

Operating System   offset

update PT entry

Page Fault Handler

load page from disk

scheduler

Group Discussion: Demand Paging as Caching, ...

- What "block size"?
- What "organization" i.e., direct-mapped, set-associative, fully-associative?
- How do we locate a page?
- What is page replacement policy? (i.e., LRU, Random...)
- What happens on a miss?
- What happens on a write? (write-through, write back)

Demand Paging as Caching, …

- What "block size"? - 1 page (e.g., 4 KB)
- What "organization" i.e., direct-mapped, set-associative, fully-associative?
  - Fully associative since arbitrary 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… (more later)
- 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

∞

Virtual
Memory
4 GB

TLB

Page
Table

Physical
Memory
512 MB
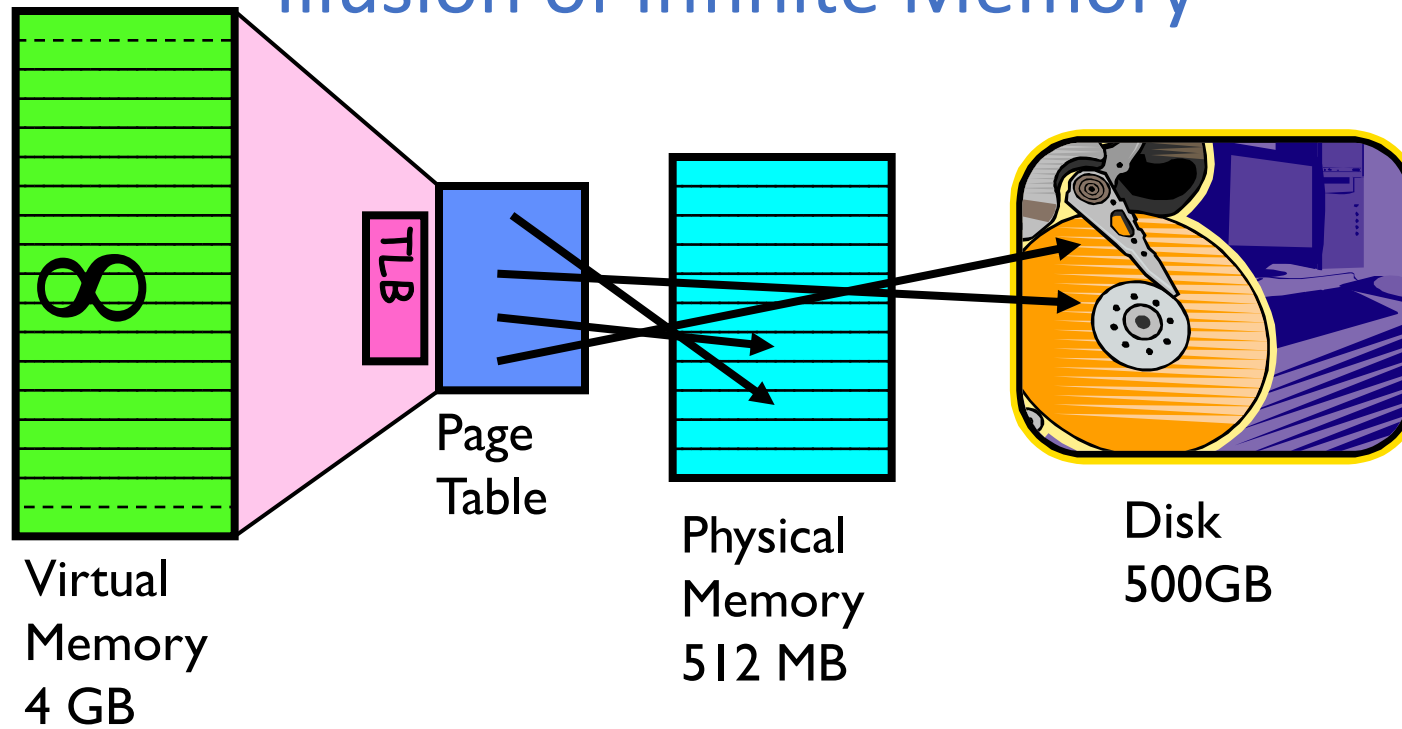
Disk
500GB

- 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 (NSDI'17 InfiniSwap, OSDI'20 AIFM)
  - Variable location of data transparent to user program
    » Performance issue, not correctness issue

Demand Paging Mechanisms

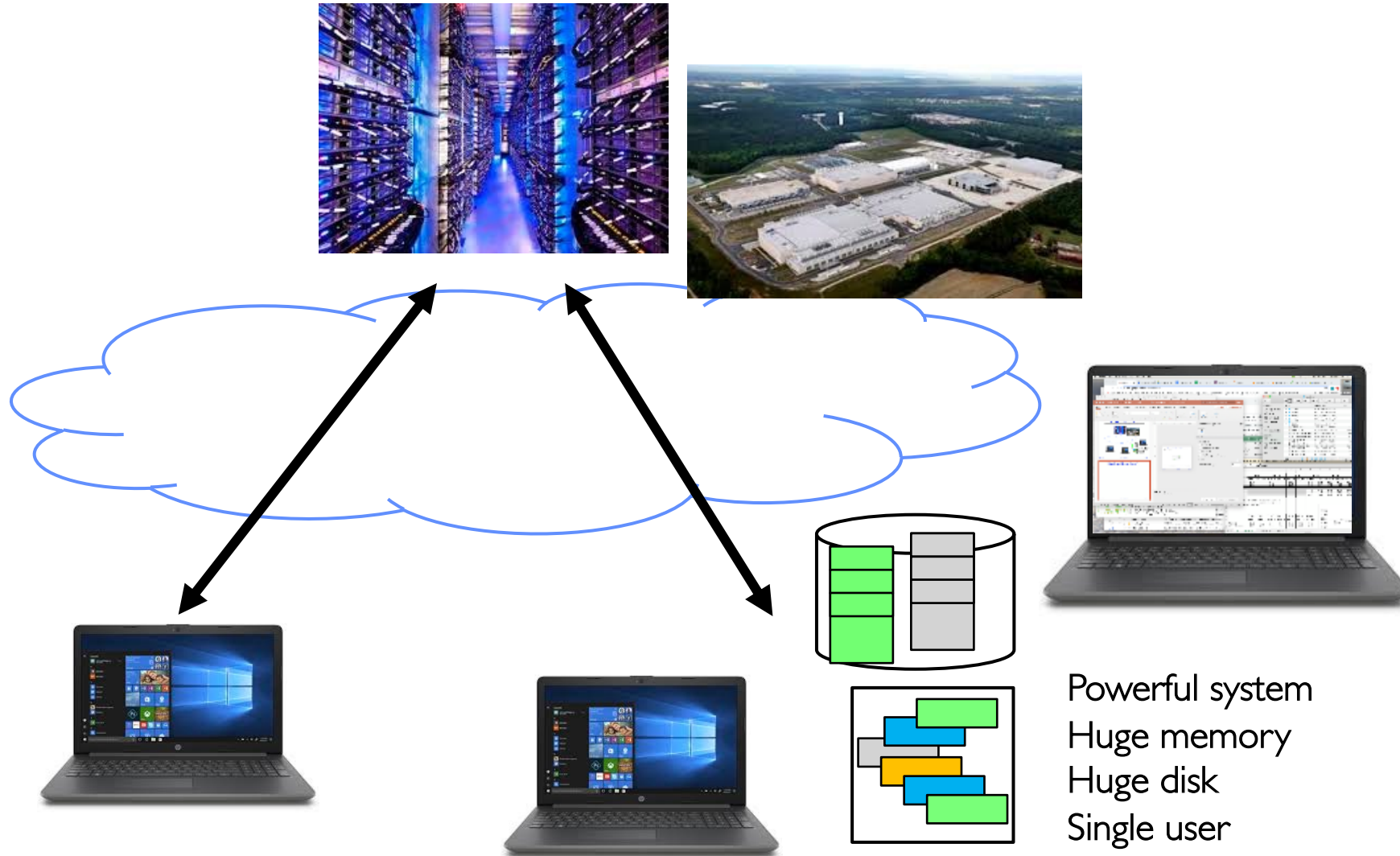- PTE makes demand paging implementable
  - Valid $\Rightarrow$ Page in memory, PTE points at physical page
  - Not Valid $\Rightarrow$ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:
    - » Choose an old page to replace
    - » If old page modified ("Dirty=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
    - » 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

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

…
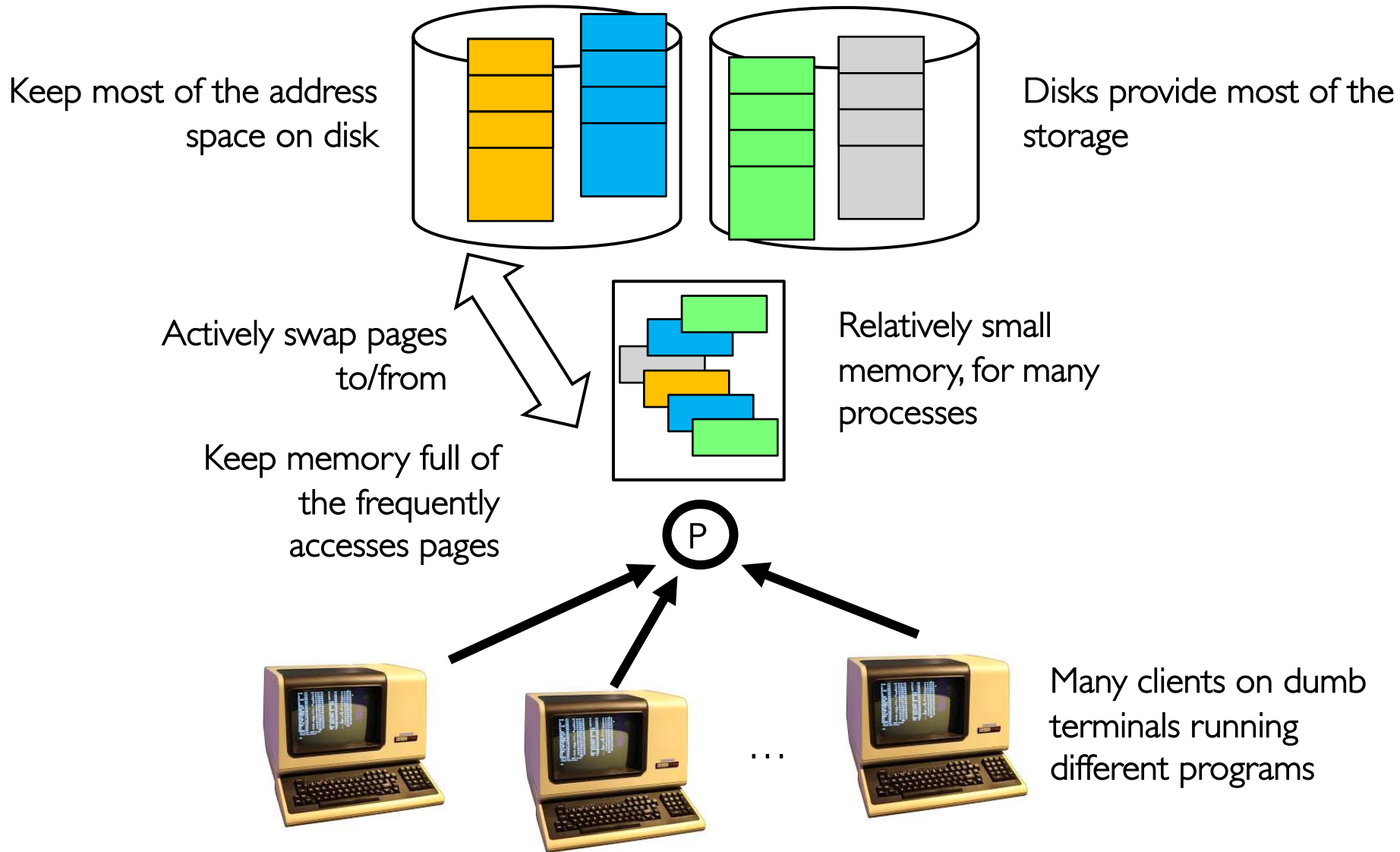
Very Different Situation Today

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       0    21      1080K  0B    564K   90498  90497  sleeping
90497  login        0.0   00:00.10       1    31      1236K  0B    1220K  90497  90496  sleeping
90496  Terminal     0.5   01:43.28       1    378-    103M-  16M   13M    90496  1      sleeping
89197  siriknowledg 0.0   00:00.83       2    45      2664K  0B    1528K  89197  1      sleeping
89193  com.apple.DF 1.0   00:17.34       1    68      2688K  0B    1700K  89193  1      sleeping
82655  LookupViewSe 0.0   00:10.75       1    169     13M    0B    8064K  82655  1      sleeping
82453  PAH_Extensio 0.   00:25.89        1    235     15M    0B    7996K  82453  1      sleeping
75819  tzlinkd      0.0   00:00.01       2    1       452K   0B    444K   75819  1      sleeping
75787  MTLCompilerS 0.0   00:00.10       2    74      9032K  0B    9020K  75787  1      sleeping
75776  secd         0.0   00:00.78       2    36      3208K  0B    2328K  75776  1      sleeping
75098  DiskUnmountW 0.0   00:00.48       2    34      1420K  0B    728K   75098  1      sleeping
75093  MTLCompilerS 0.0   00:00.06       2    21      5924K  0B    5912K  75093  1      sleeping
74938  ssh-agent    0.0   00:00.00       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 80% used
- A lot of it is shared 1.9 GB

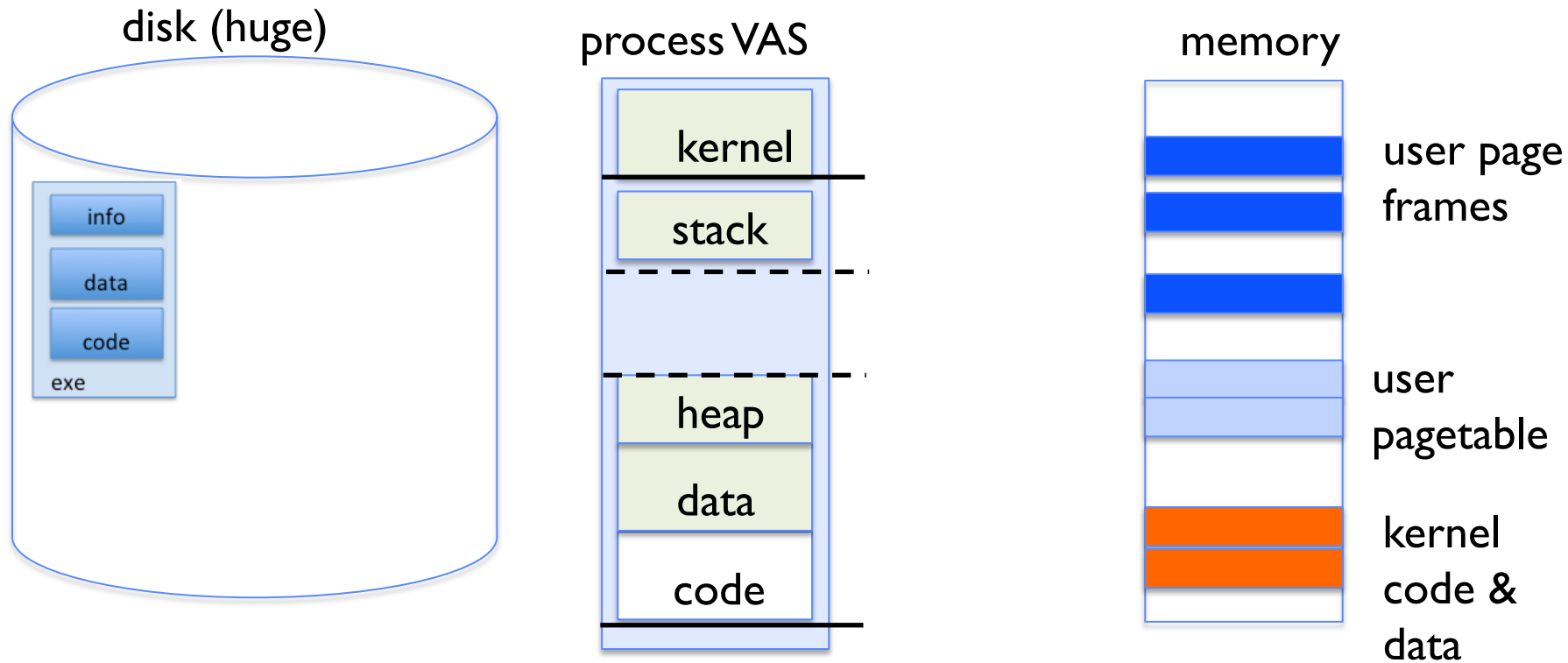# Many Uses of Virtual Memory and "Demand Paging" …

- Extend the stack
  - Allocate a page and zero it
- Extend the heap
- 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

disk (huge)

memory

info

data

code

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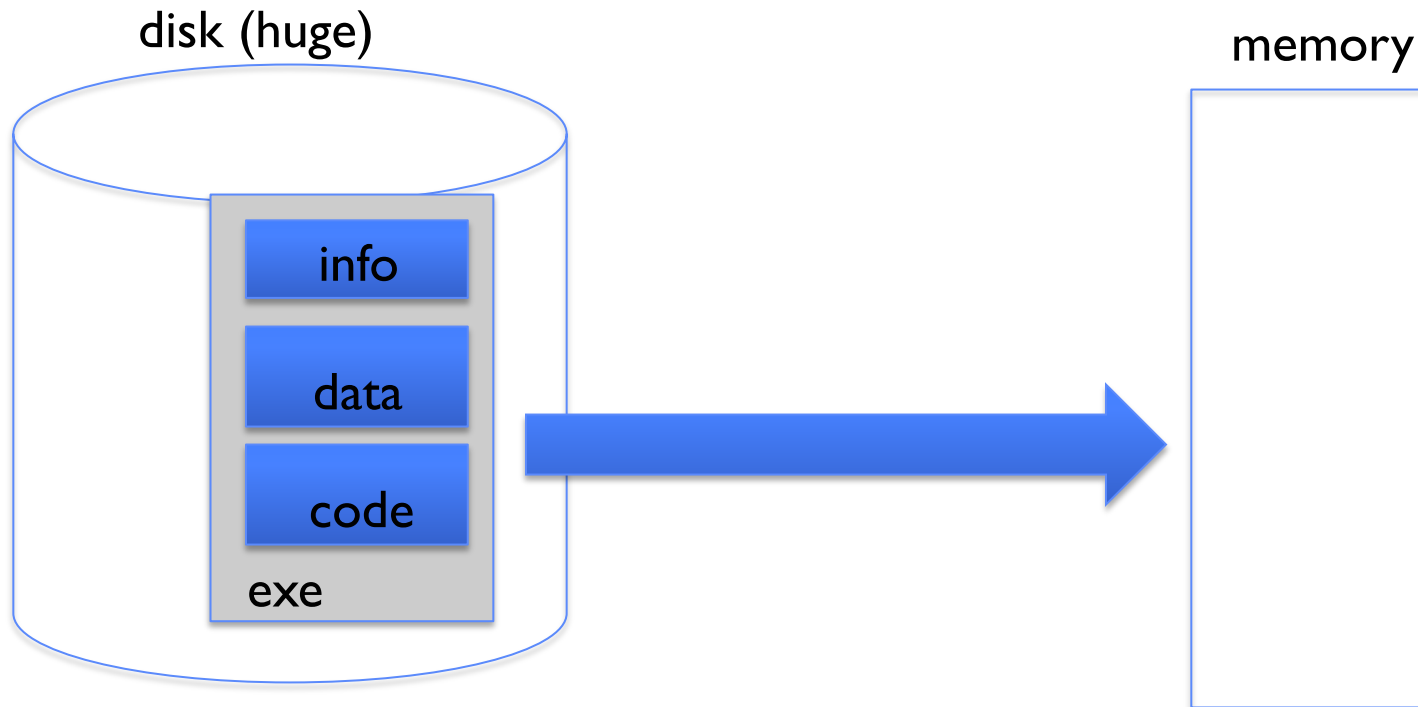
- .exe
  - lives on disk in the file system
  - contains contents of code & data segments, relocation entries and symbols
  - OS loads it into memory, initializes registers (and initial stack pointer)
  - program sets up stack and heap upon initialization

Create Virtual Address Space of the Process

disk (huge)

info

data

code

exe

process VAS

kernel

stack

heap

data

code

memory

user page frames
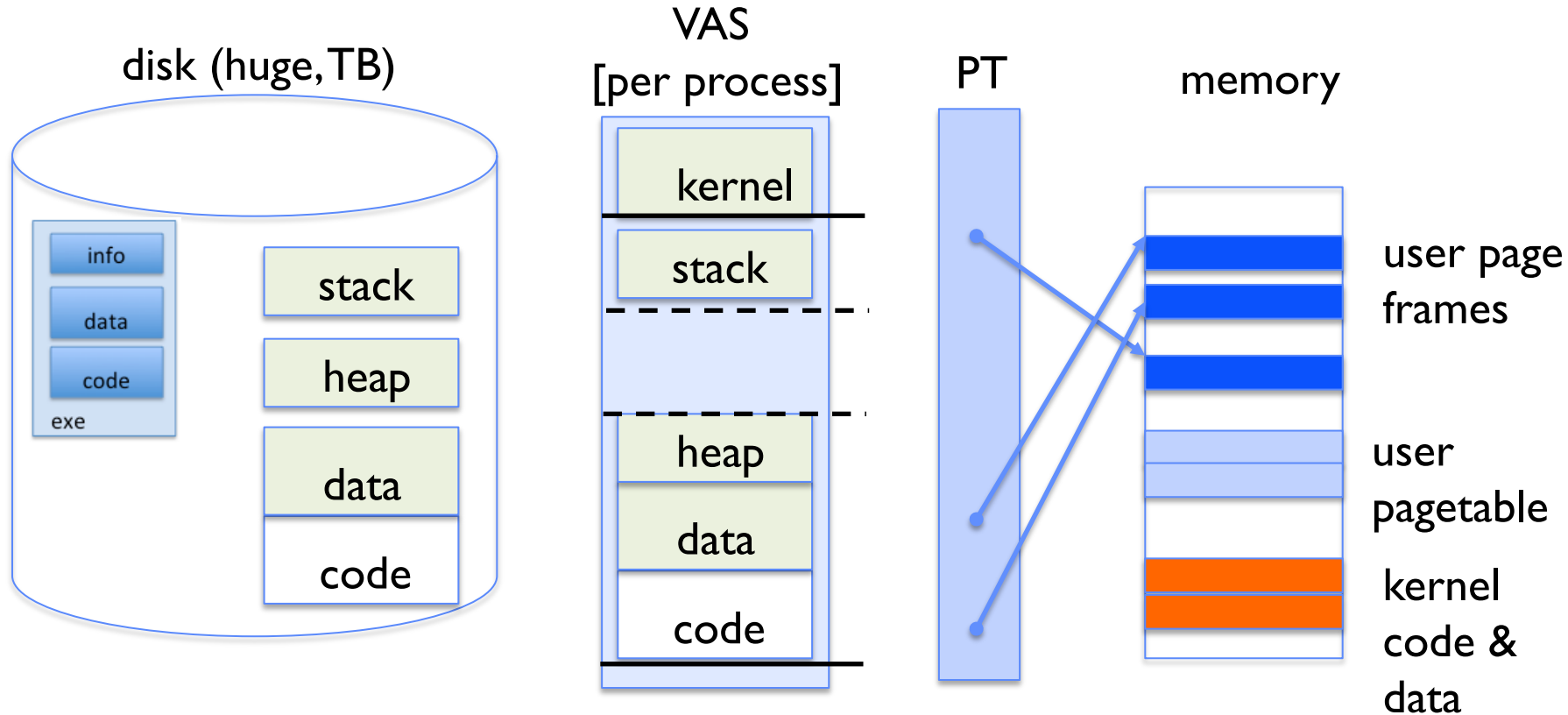
user pagetable

kernel code & data

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

Create Virtual Address Space of the Process

disk (huge,TB)

process VAS (GBs)
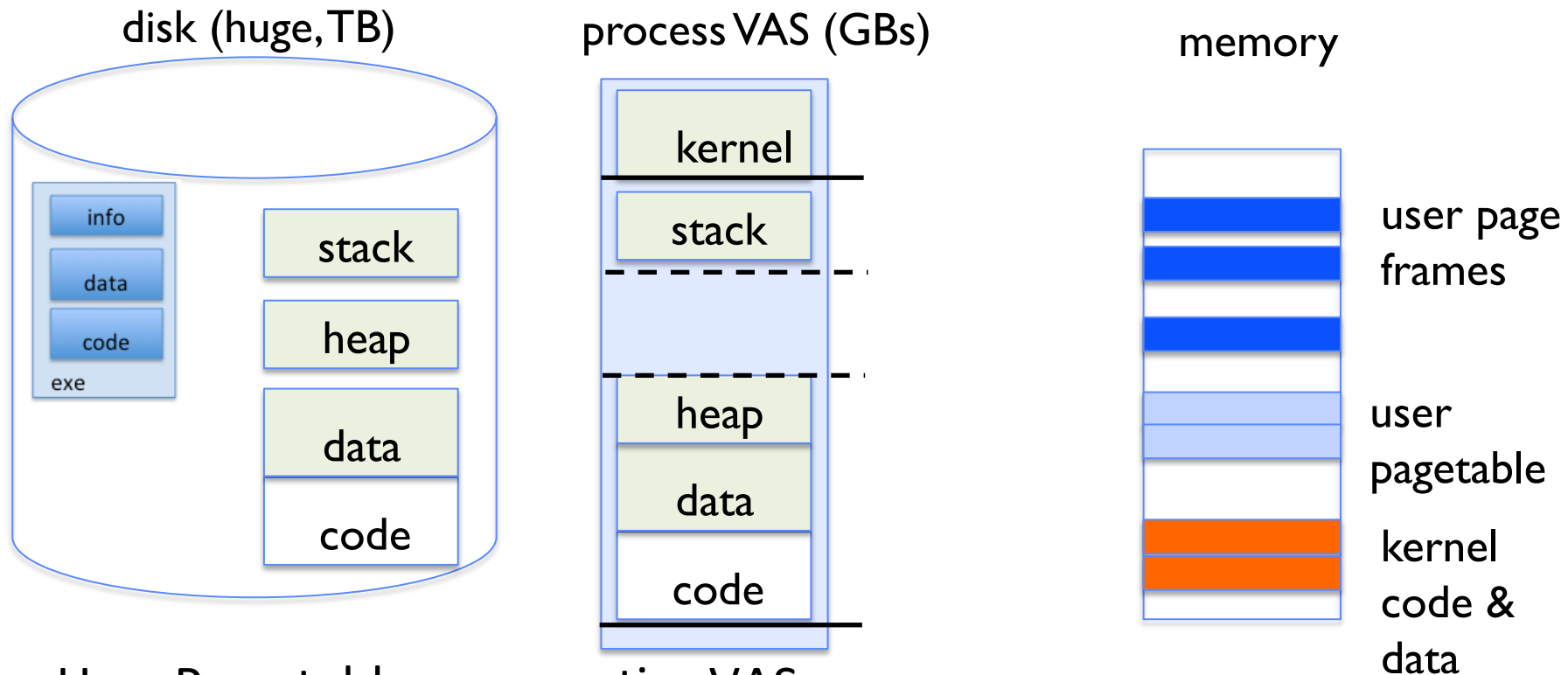
memory

- 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 mapped to the frame in memory they occupy
  - The portion of page table that the HW needs to access must be resident in memory
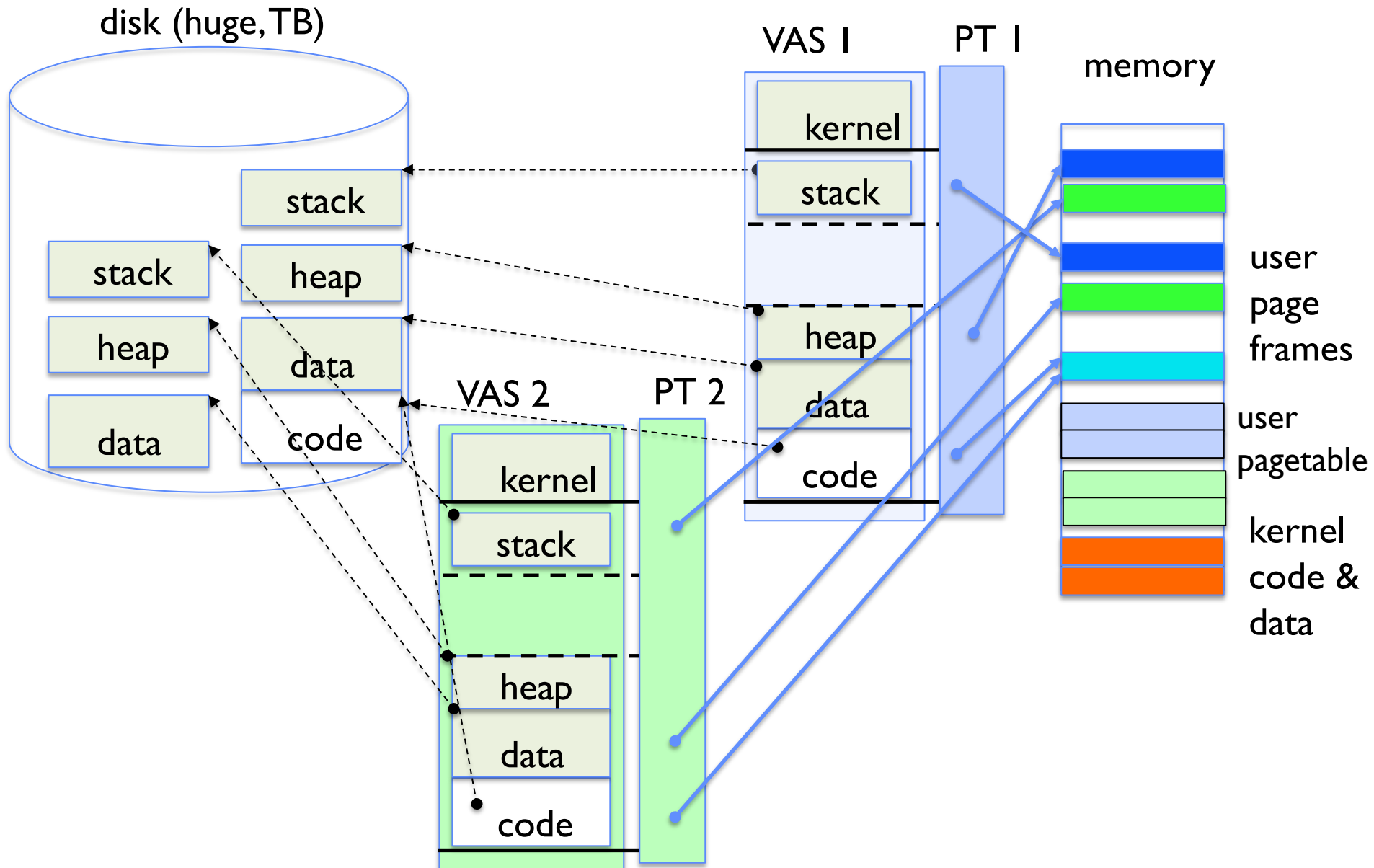
Provide Backing Store for VAS
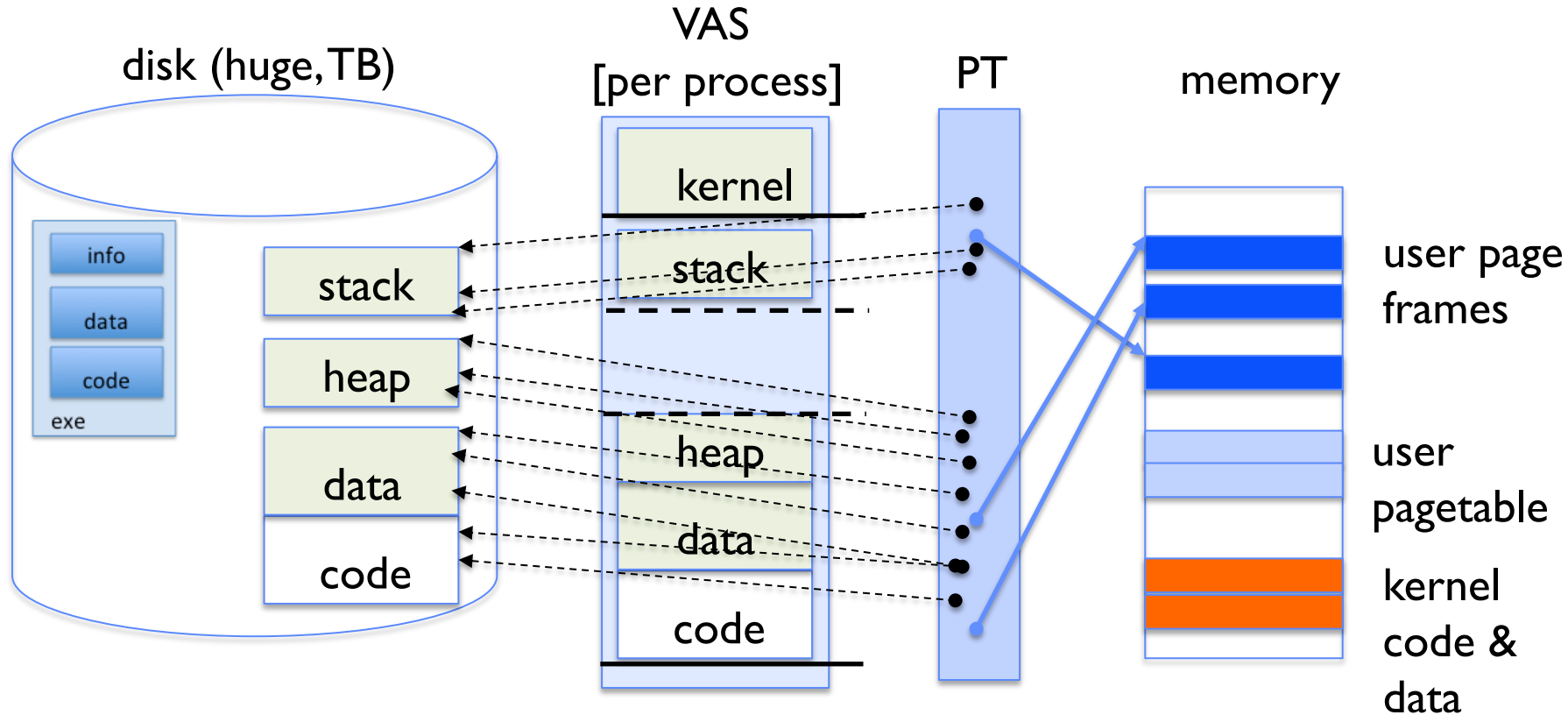
- User Page table maps entire VAS
- Resident pages mapped to the frame in memory they occupy
- For all other pages, OS must record where to find them on disk

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

disk (huge,TB)

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

On Page Fault …

disk (huge,TB)

stack
heap
data

stack
heap
data
code

VAS 1

kernel
stack

heap
data
code

PT 1

VAS 2

kernel
stack

heap
data
code

PT 2

memory

user
page
frames

user
pagetable

kernel
code
& data

active process & PT

33

disk (huge, TB)

stack

stack

heap

heap

data

data

data

code

VAS 1

kernel

stack

heap

data

code

PT 1

VAS 2

kernel

stack

heap

data

code

PT 2

memory

user
page
frames

user
pagetable

kernel
code
& data

active process & PT

On page Fault ... schedule other Process or Thread

disk (huge,TB)

stack

stack

heap

heap

data

data

code

VAS 1

kernel

stack

heap

data

code

VAS 2

kernel

stack

heap

data

code

PT 1

PT 2

memory

user
page
frames

user
pagetable

kernel
code &
data

active process & PT

35

disk (huge,TB)

stack

stack

heap

heap

data

data

code

VAS 1

kernel

stack

heap

data

code

VAS 2

kernel

stack

heap

data

code

PT 1

PT 2

memory

user page frames

user pagetable

kernel code & data

active process & PT
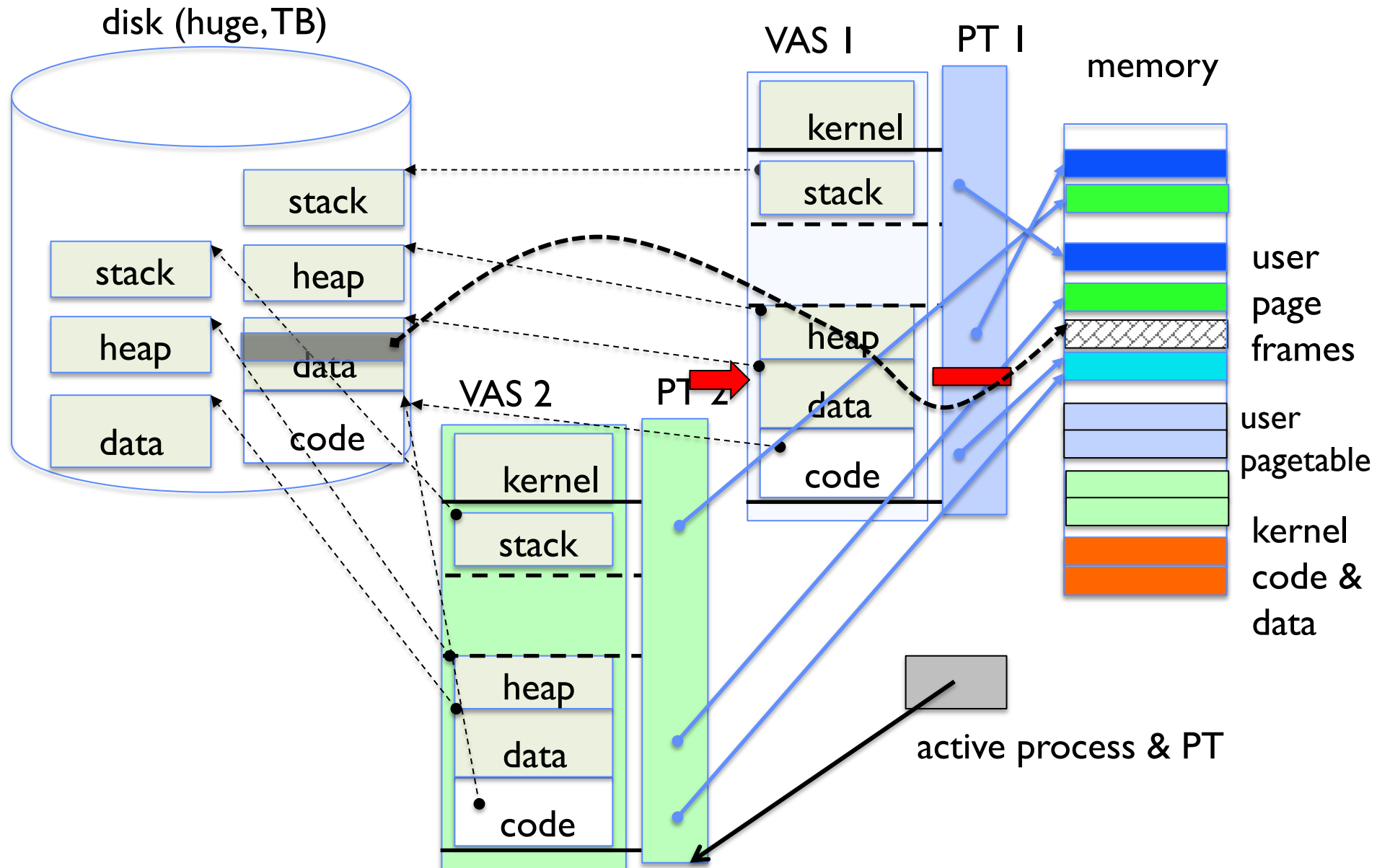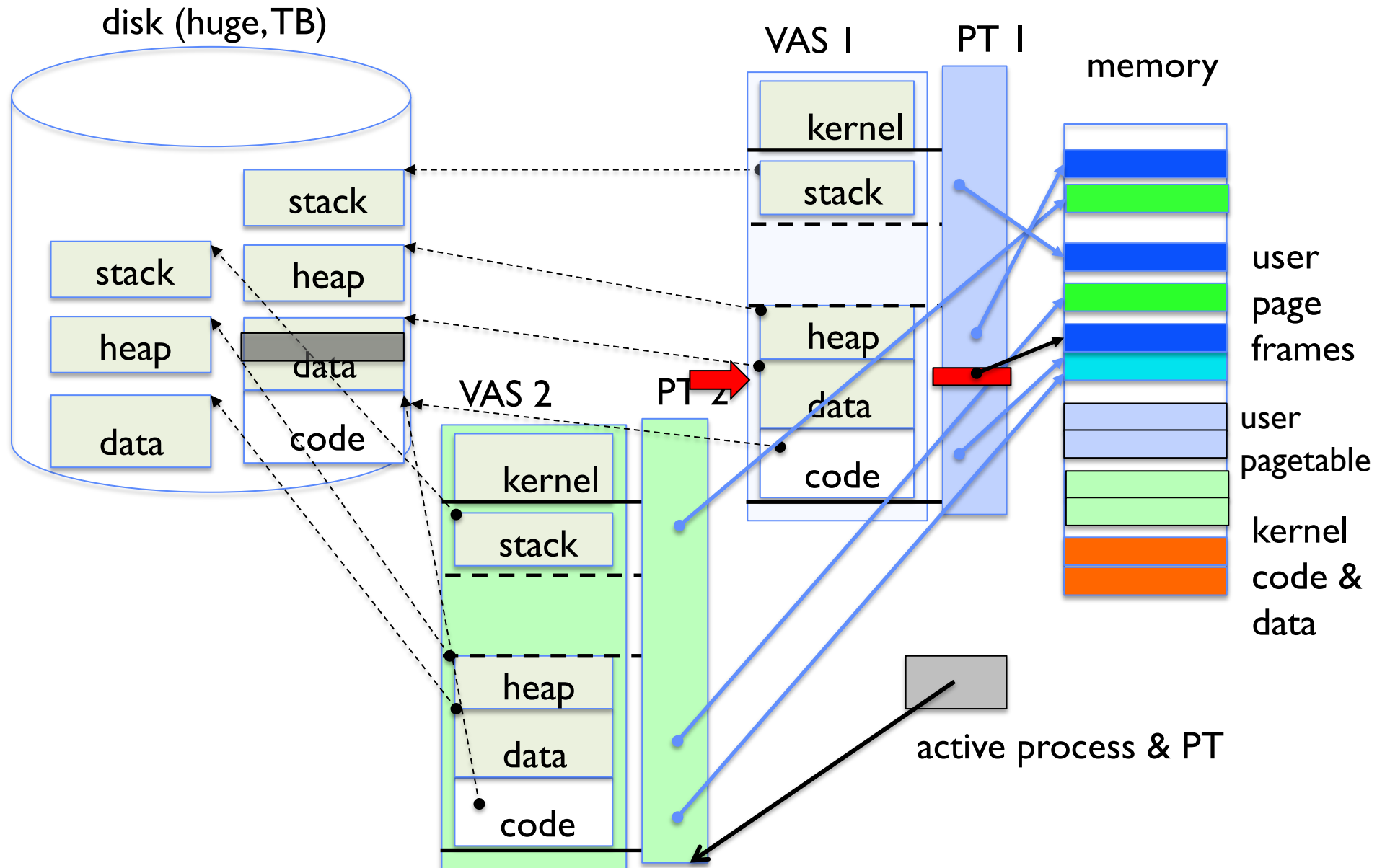
Eventually reschedule faulting thread

disk (huge,TB)

stack

stack

heap

heap

data

data

code

VAS 1

kernel

stack

heap

data

code

PT 1

VAS 2

kernel

stack

heap

data

code

PT 2

memory

user
page
frames

user
pagetable

kernel
code &
data

active process & PT
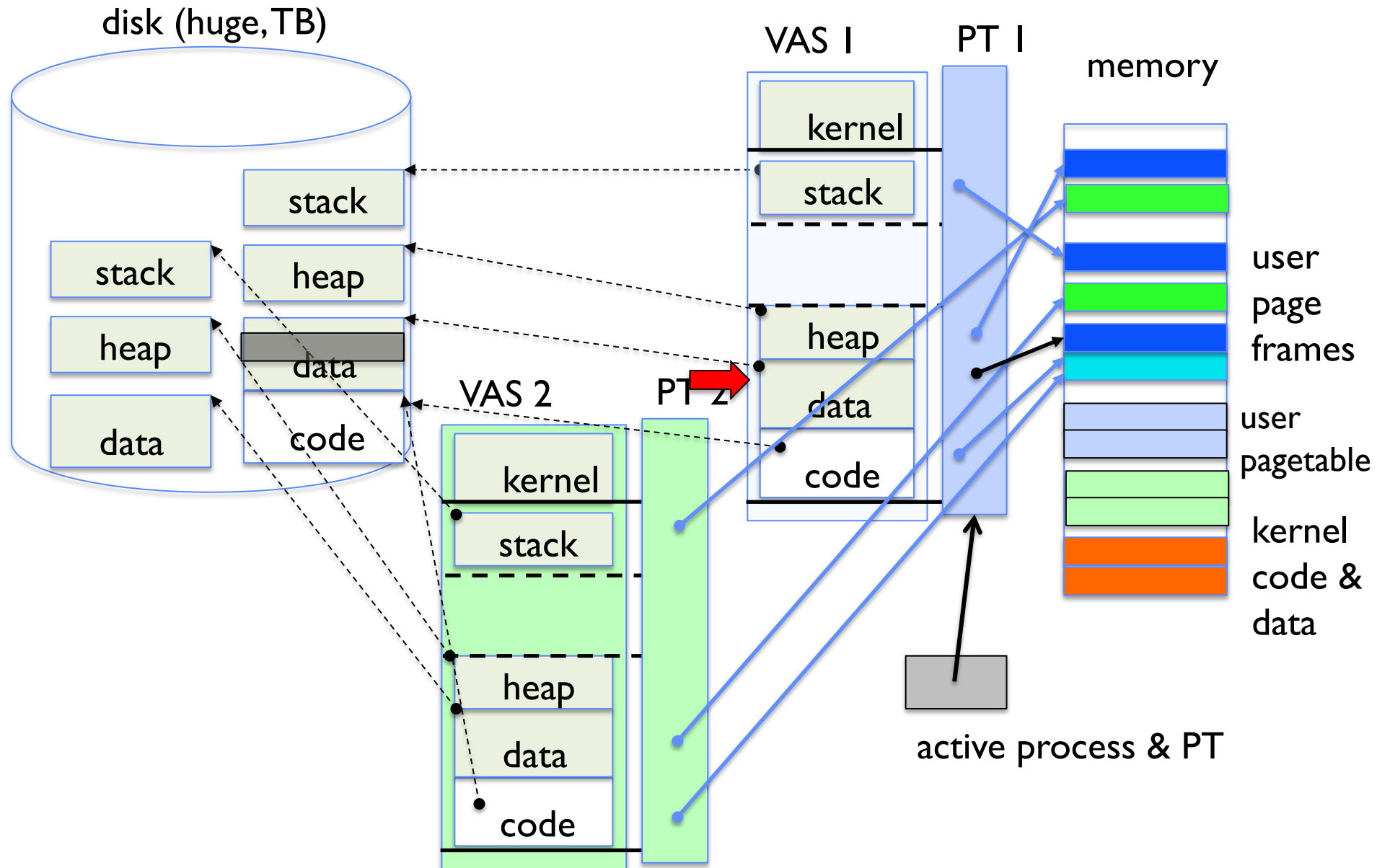
Summary: Steps in Handling a Page Fault

3 page is on backing store

operating system

2 trap

reference

1

load M

i

page table

6 restart instruction

5 reset page table

free frame

4 bring in missing page

physical memory
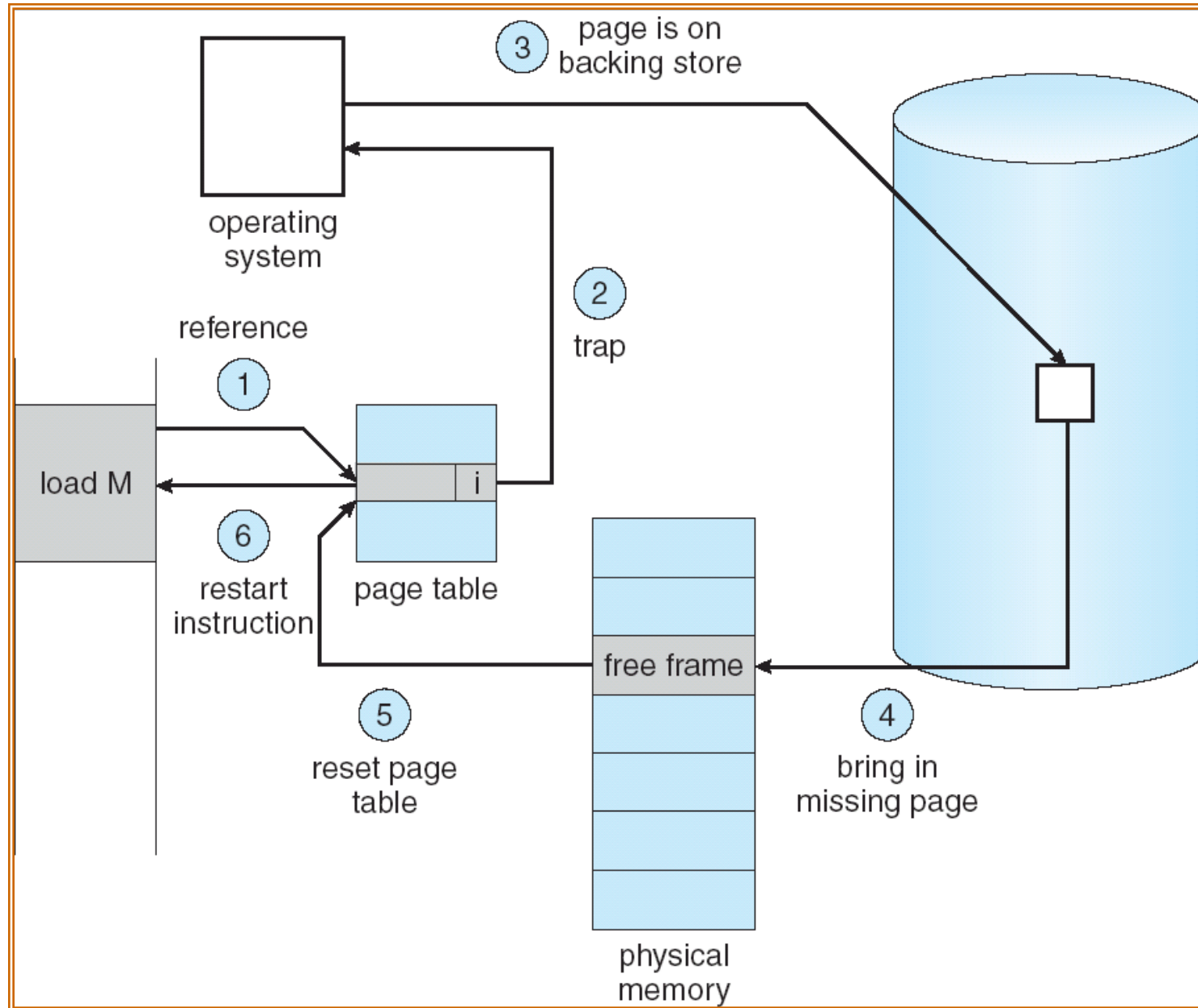
# Some questions we need to answer!

- 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 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?
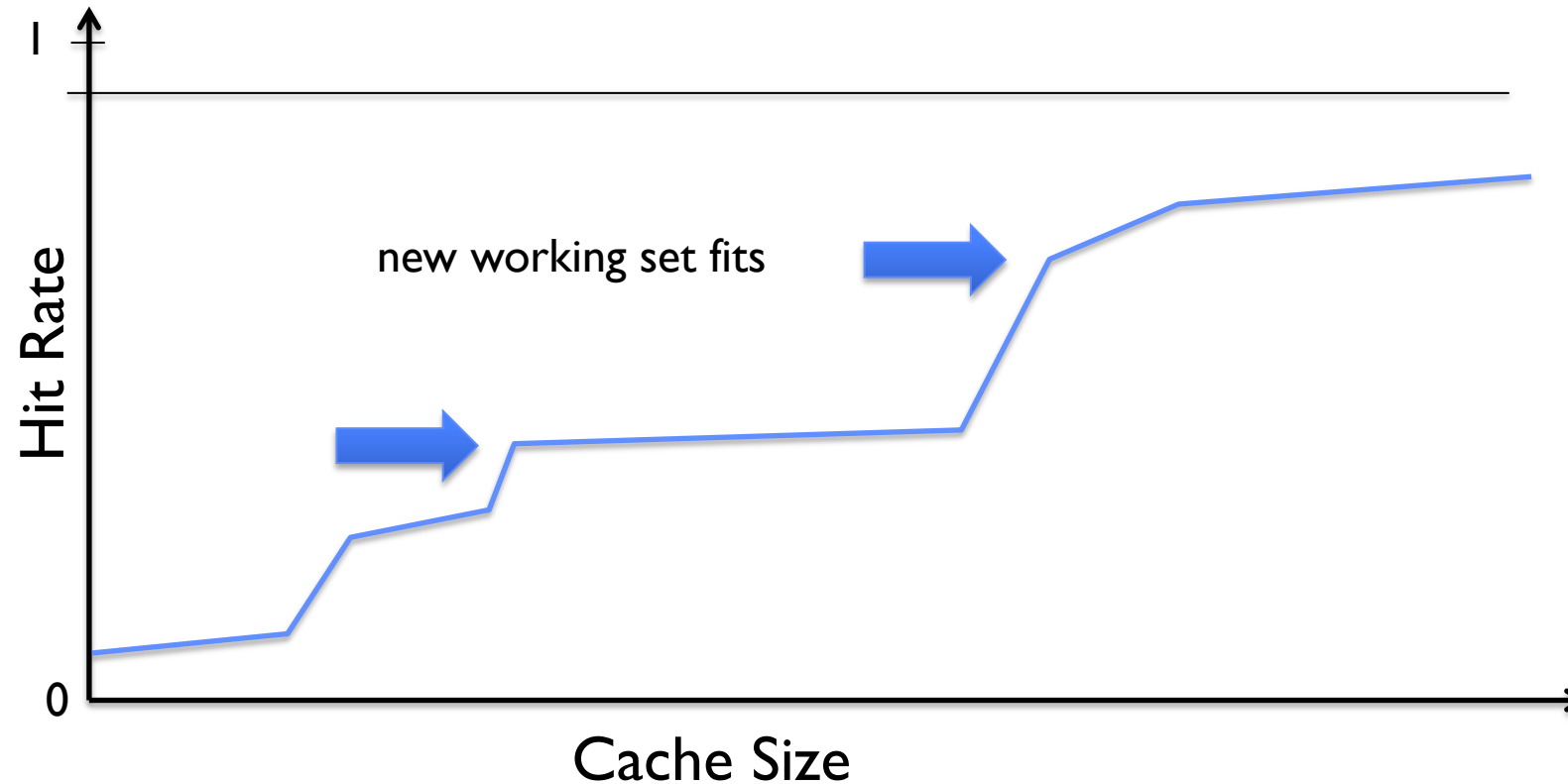 - Allocation of disk paging bandwidth

Working Set Model

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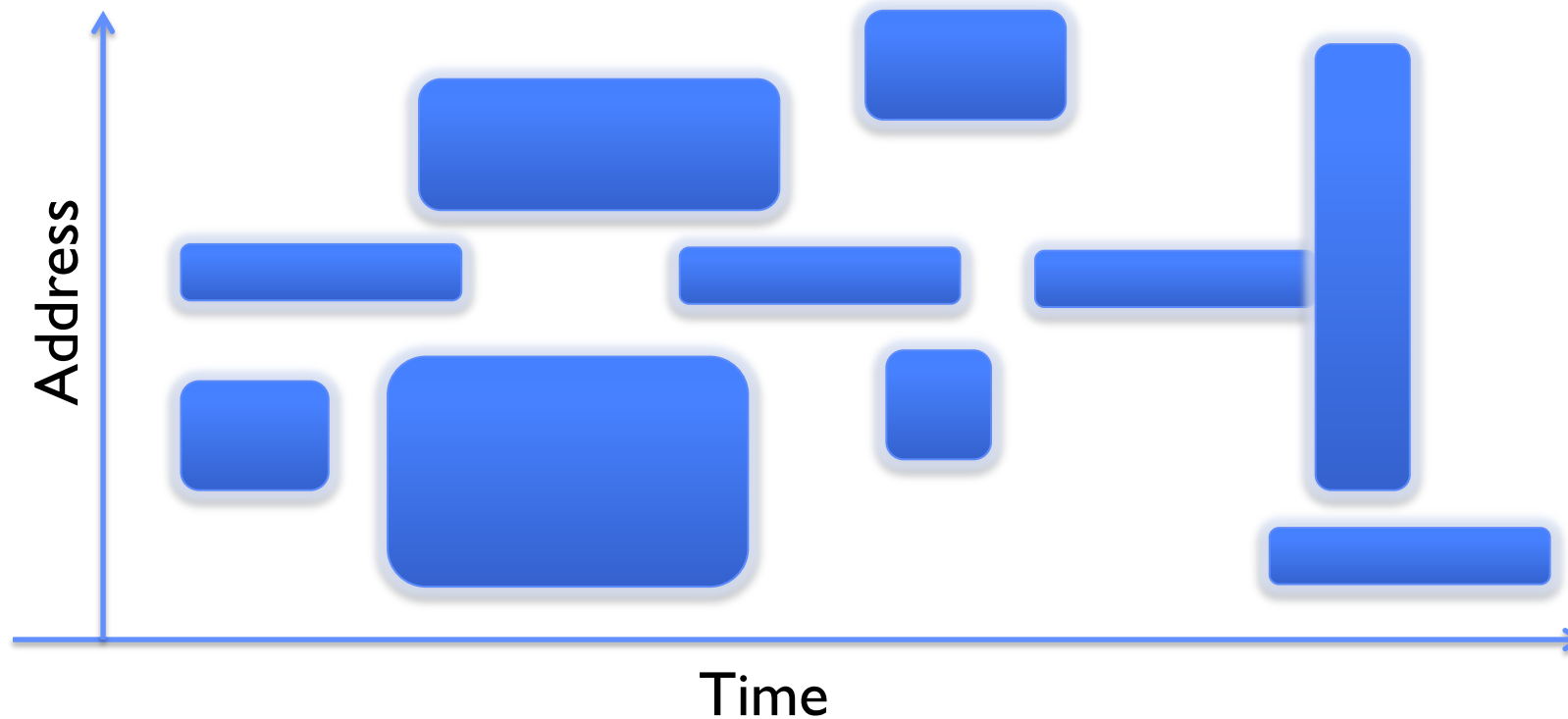
Cache Behavior under WS model

- 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

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 (Hit Rate + Miss Rate = 1)
  - EAT = Hit Time + Miss Rate x Miss Penalty (Miss Penalty = Miss Time – Hit Time)
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time (Miss Penalty) = 8 milliseconds
  - Suppose p = Probability of miss, 1-p = Probably of hit
  - Then, we can compute EAT as follows:

    EAT $\quad$ = 200ns + p x 8 ms

    $\quad$ = 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 40x !
- 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!

Group Discussion

- **Compulsory Misses:**
  - What are they in the context of demand paging?
  - How might we remove these misses?

- **Capacity Misses:**
  - What are they in the context of demand paging?
  - How might we remove these misses?

- **Conflict Misses:**
  - What are they in the context of demand paging?
  - How might we remove these misses?

- **Policy Misses:**
  - What are they in the context of demand paging?
  - How might we remove these misses?

# 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!
- **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 …

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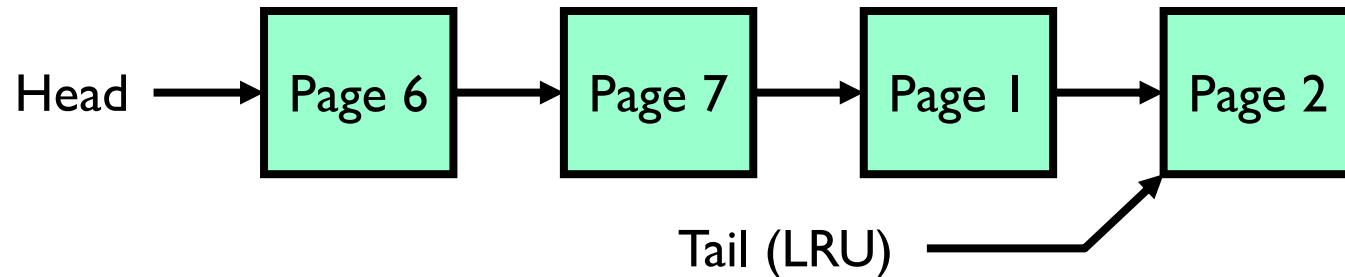
  - On each use, remove page from list and place at head
  - LRU page is at tail
- Problems with this scheme for paging?
  - Need to know immediately when page used so that can change position in list…
  - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

Group Discussion

- Topic: replacement policies
  - Can you compare FIFO, RANDOM, MIN and LRU?
  - What are the pros and cons of each approach?

- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

Example: FIFO (strawman)

- 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:	A	B	C	A	B	D	A	D	B	C	B
Page:												
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 / LRU

- 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!

Is LRU guaranteed to perform well?

- 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!
- Fairly contrived example of working set of N+1 on N frames

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

- 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