

CS162
Operating Systems and
Systems Programming
Lecture 15

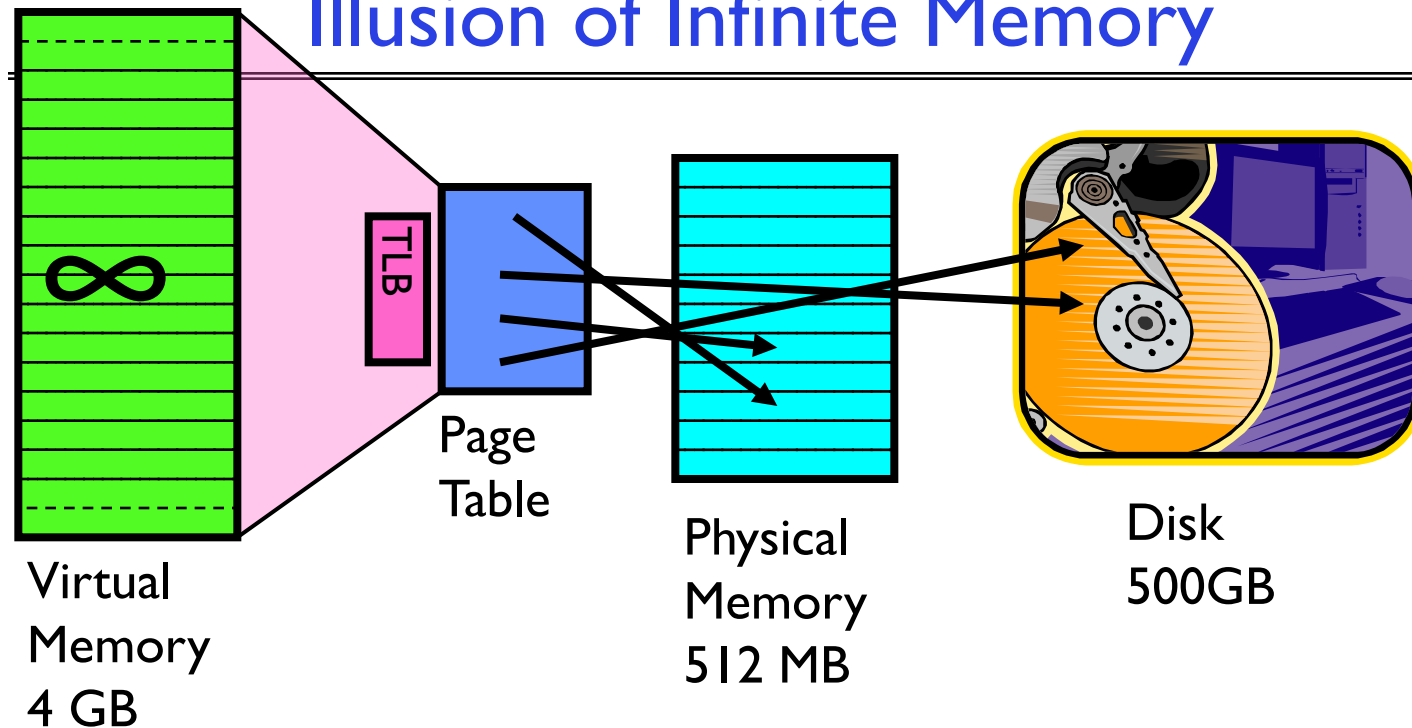
Demand Paging (Finished)

March 15th, 2017

Nathan Pemberton

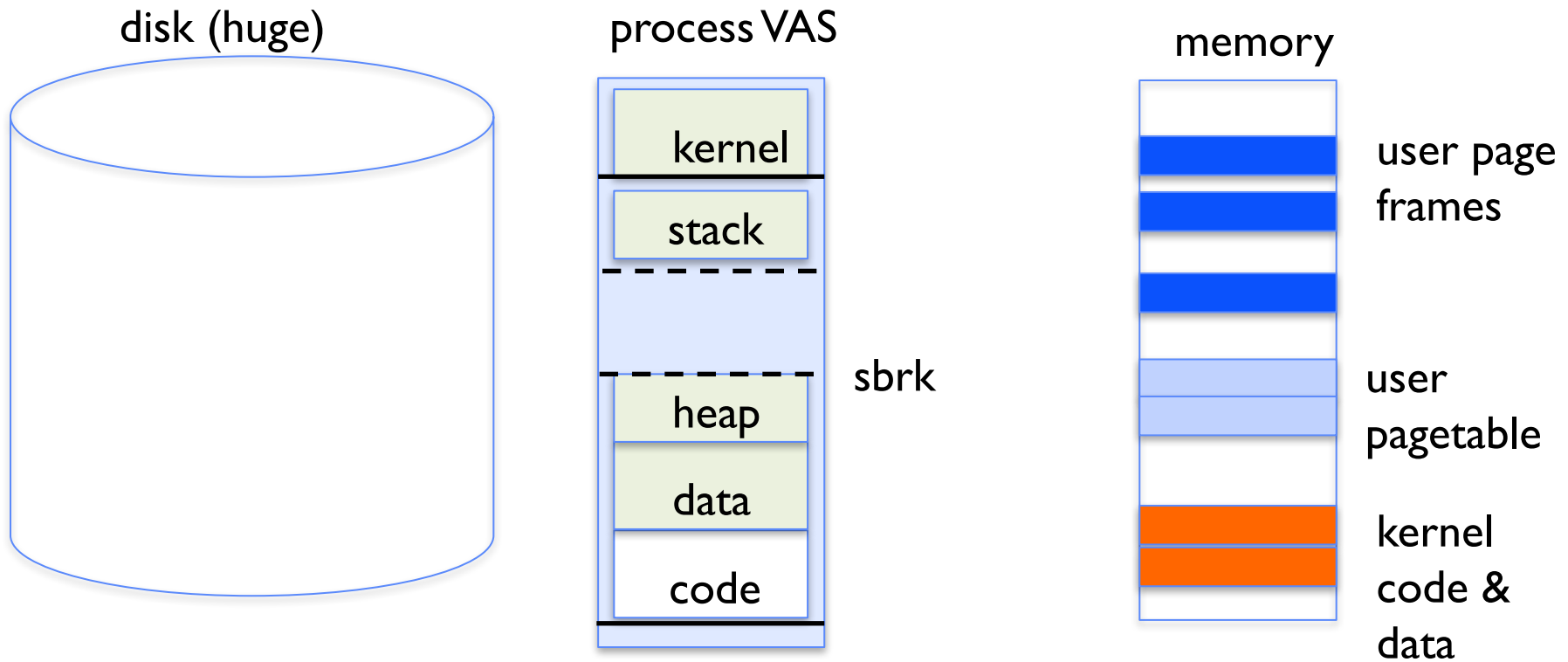
<http://cs162.eecs.Berkeley.edu>

Illusion of Infinite Memory



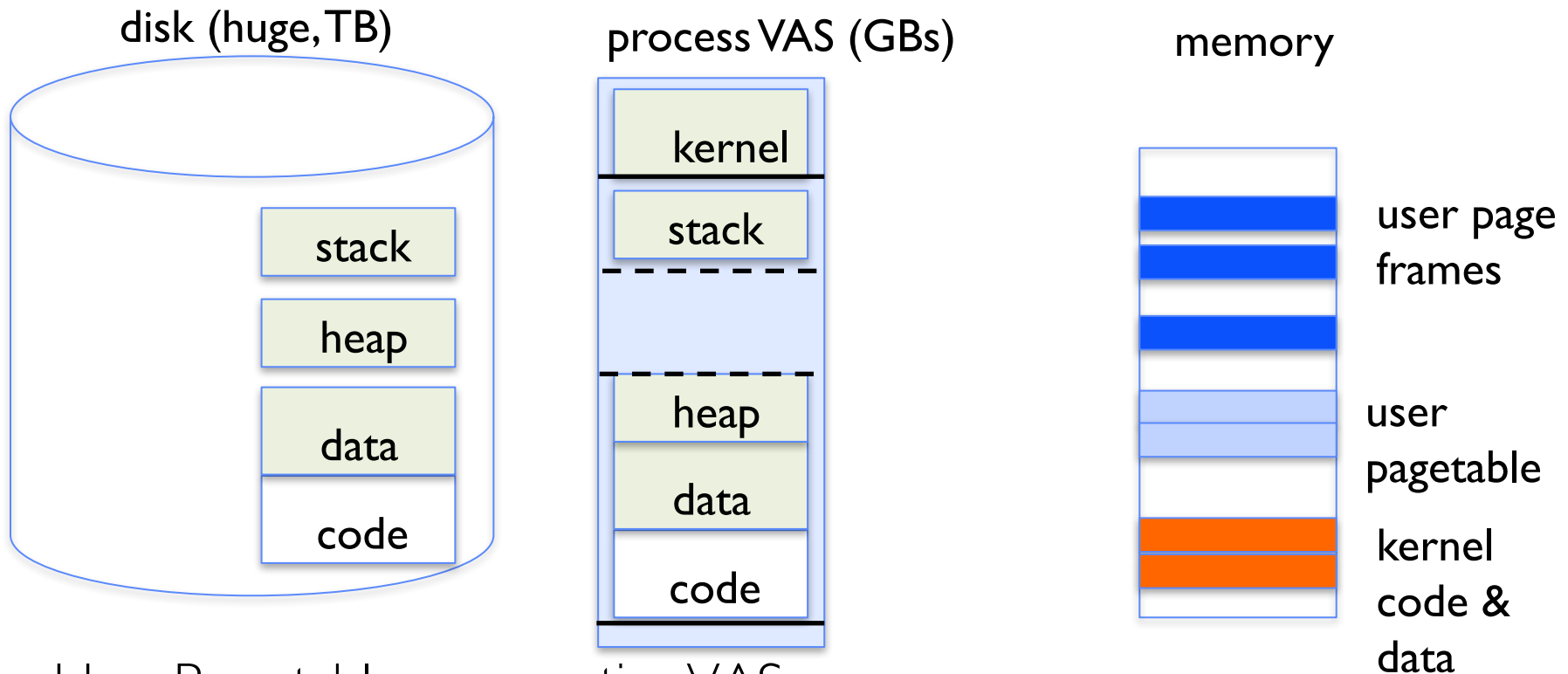
- Disk is larger than physical memory \Rightarrow
 - In-use virtual memory can be bigger than physical memory
 - Combined memory of running processes much larger than physical memory
 - » More programs fit into memory, allowing more concurrency
- Principle: **Transparent Level of Indirection** (page table)
 - Supports flexible placement of physical data
 - » Data could be on disk or somewhere across network
 - Variable location of data transparent to user program
 - » Performance issue, not correctness issue

Create Virtual Address Space of the Process



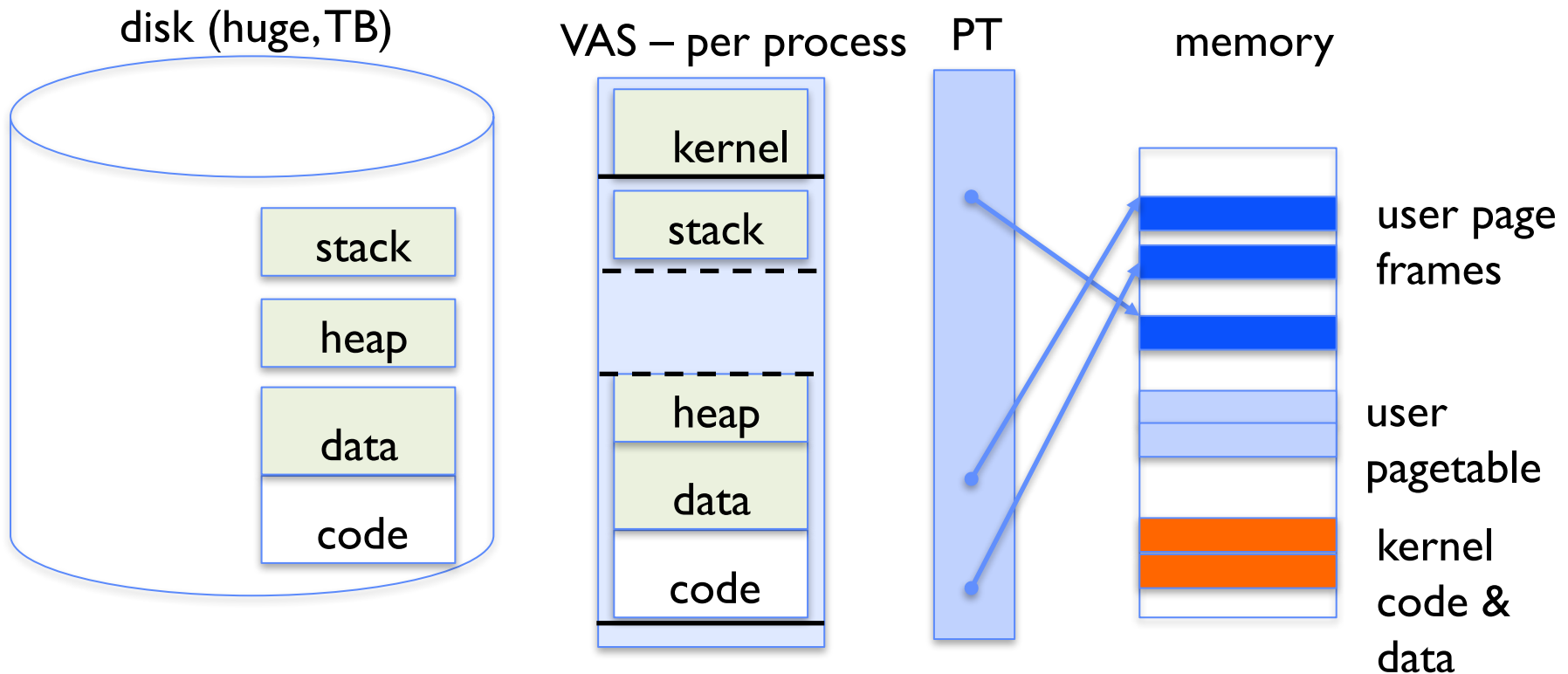
- Utilized pages in the VAS are backed by a page block on disk
 - Called the backing store or swap file
 - Typically in an optimized block store, but can think of it like a file

Create Virtual Address Space of the Process



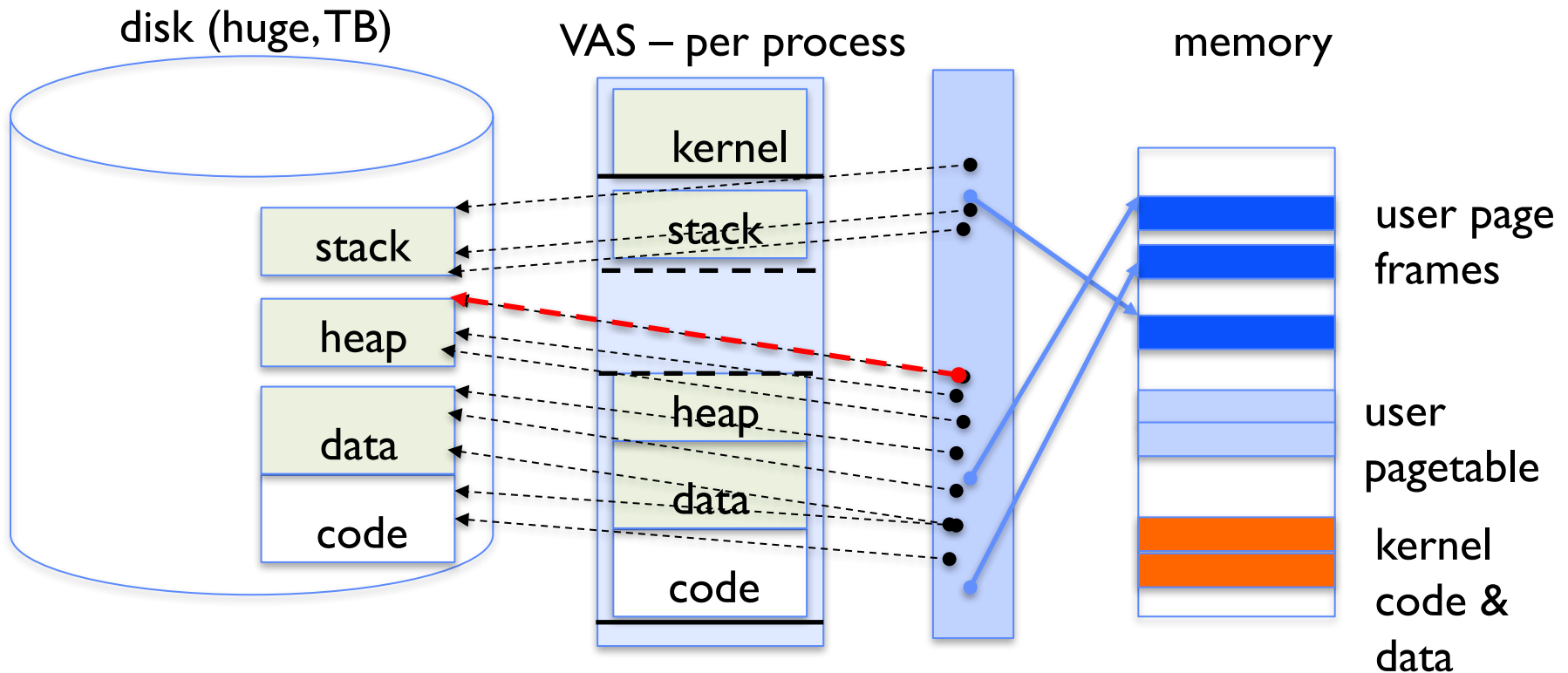
- User Page table maps entire VAS
- All the utilized regions are backed on disk
 - swapped into and out of memory as needed
- For every process

Create Virtual Address Space of the Process



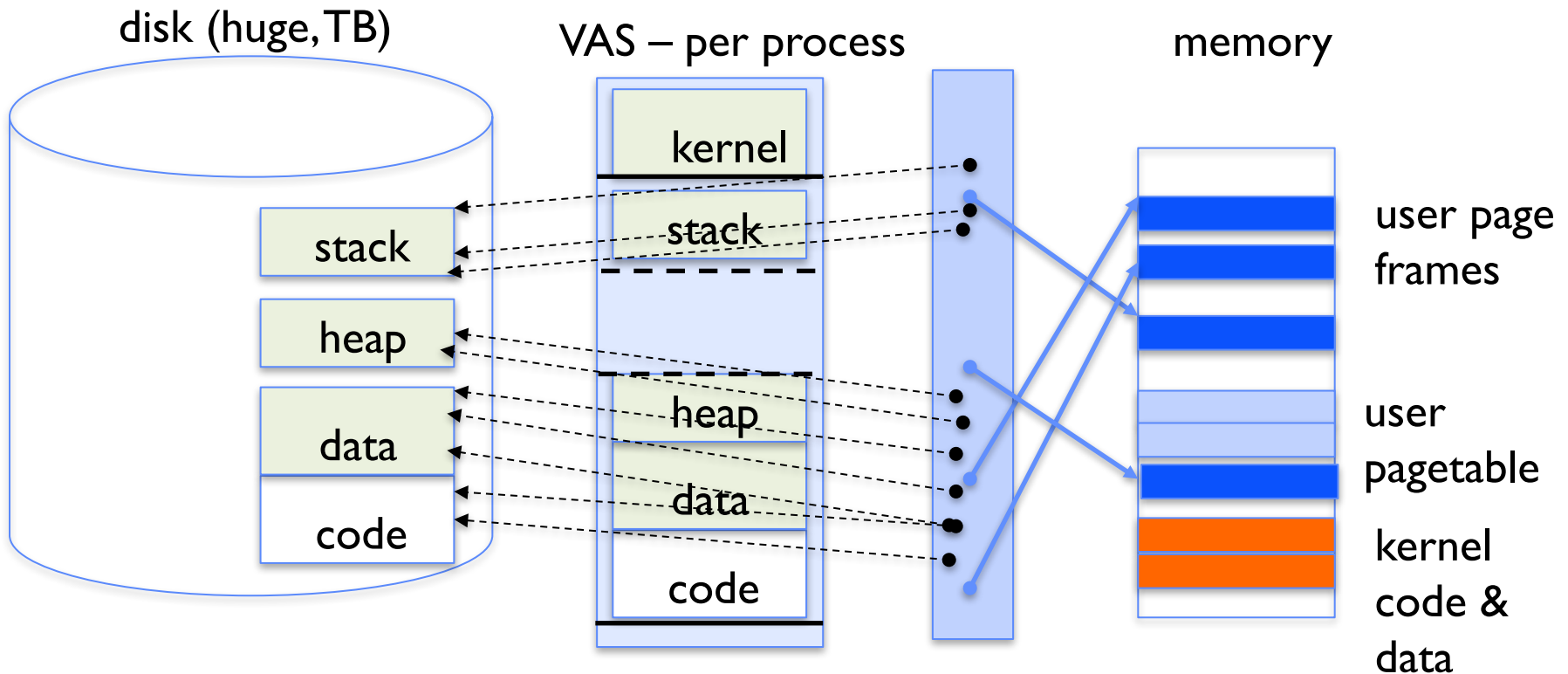
- User Page table maps entire VAS
 - Resident pages to the frame in memory they occupy
 - The portion of it that the HW needs to access must be resident in memory

Provide Backing Store for VAS



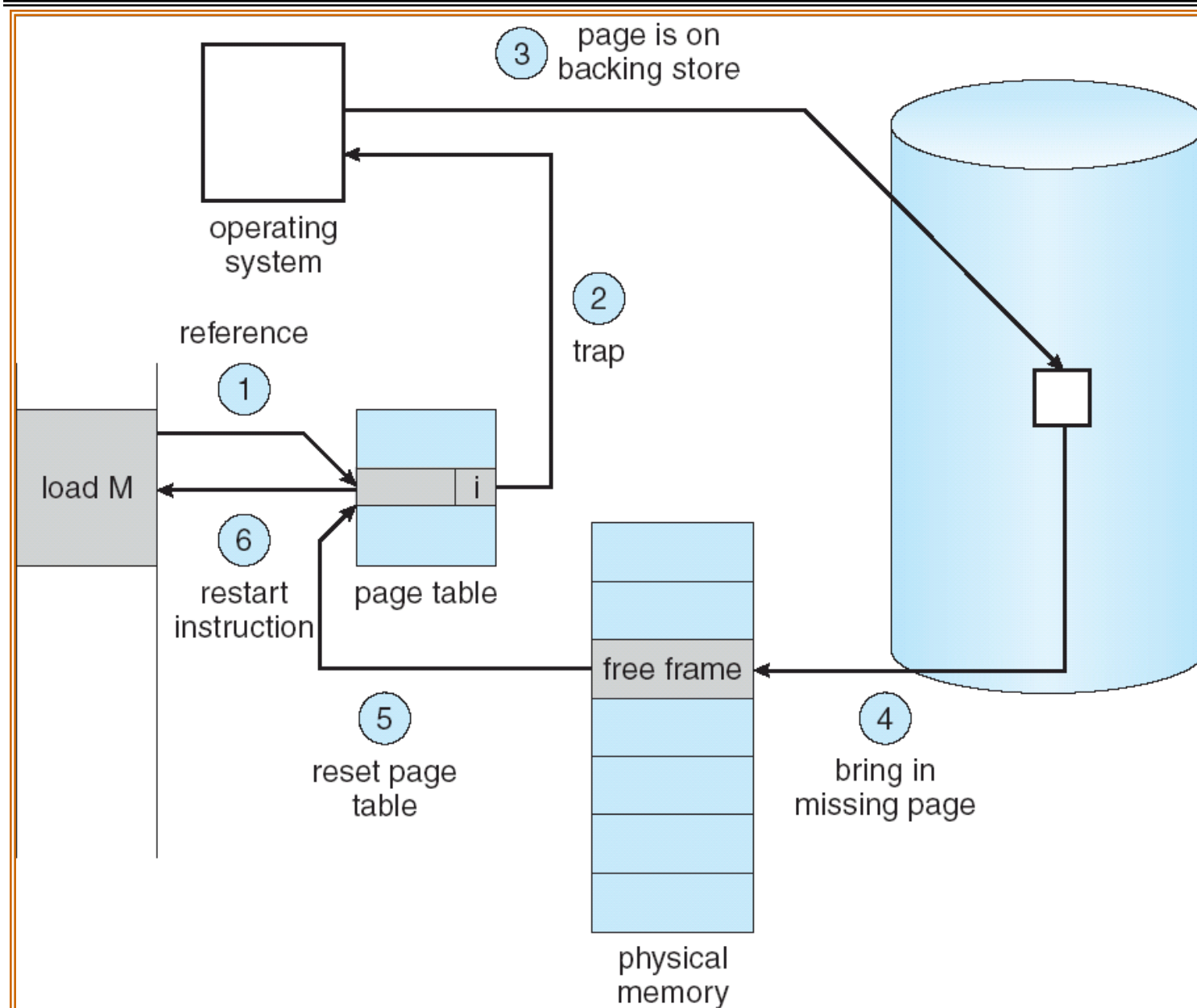
- User Page table maps entire VAS
- Resident pages mapped to memory frames
- For all other pages, OS must record where to find them on disk

Provide Backing Store for VAS



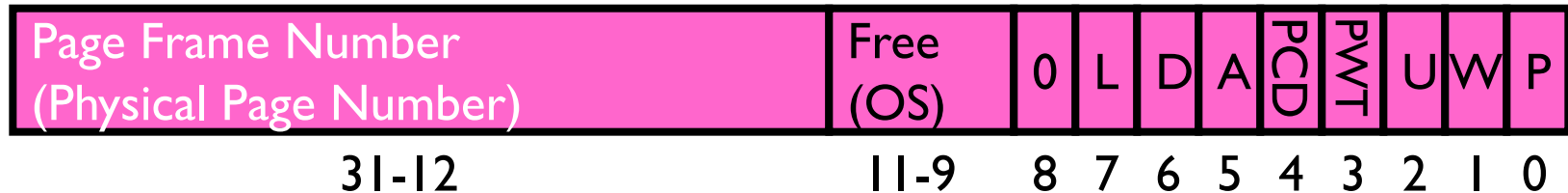
- User Page table maps entire VAS
- Resident pages mapped to memory frames
- For all other pages, OS must record where to find them on disk

Summary: Steps in Handling a Page Fault



Recall: What is in a Page Table Entry

- What is in a Page Table Entry (or PTE)?
 - Pointer to next-level page table or to actual page
 - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
 - Address same format previous slide (10, 10, 12-bit offset)
 - Intermediate page tables called “Directories”



P: Present (same as “valid” bit in other architectures)

W: Writeable

U: User accessible

PWT: Page write transparent: external cache write-through

PCD: Page cache disabled (page cannot be cached)

A: Accessed: page has been accessed recently

D: Dirty (PTE only): page has been modified recently

L: $L=1 \Rightarrow 4\text{MB}$ page (directory only).

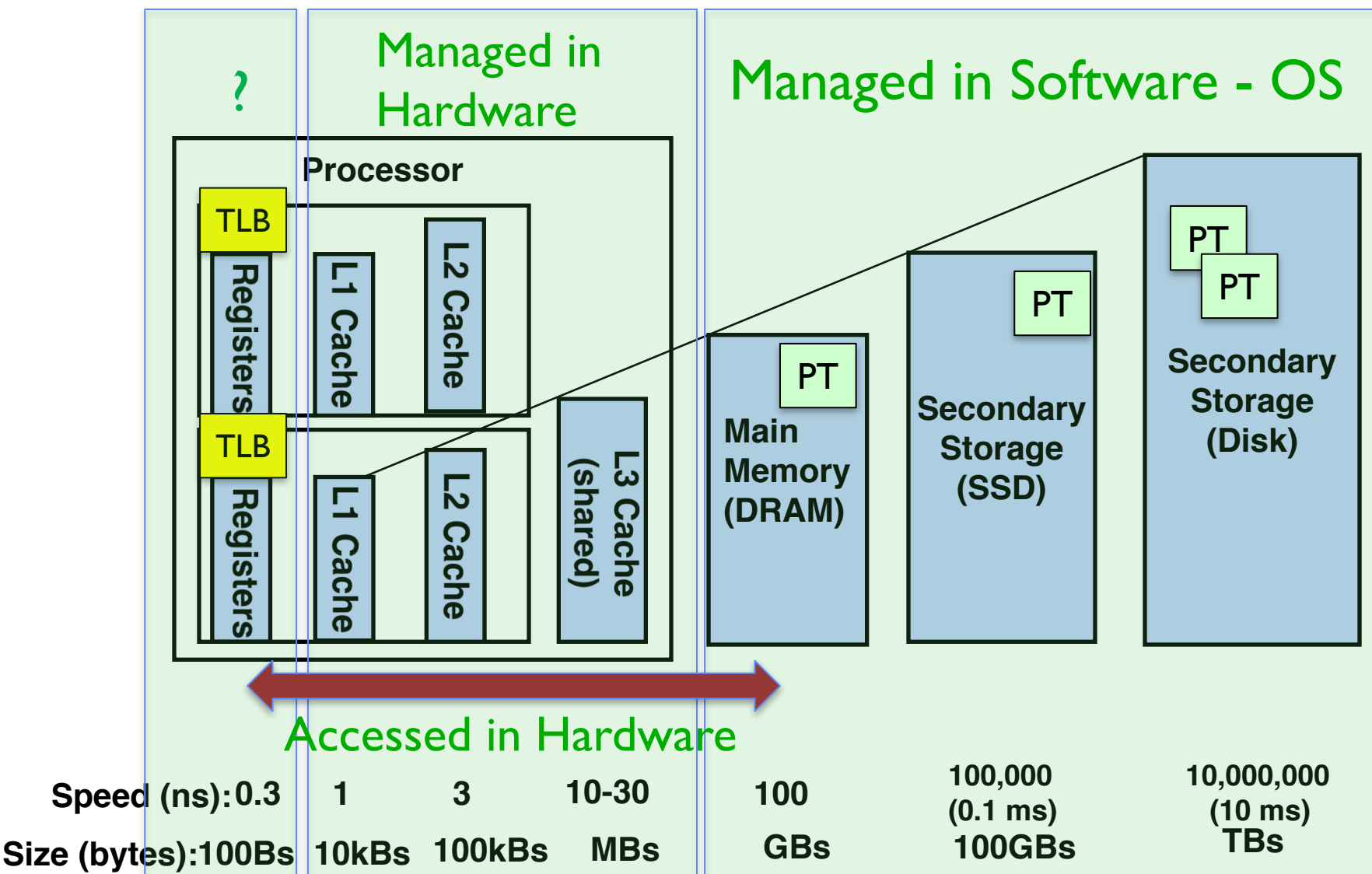
Bottom 22 bits of virtual address serve as offset

Demand Paging Mechanisms

- PTE helps us implement demand paging
 - Valid \Rightarrow Page in memory, PTE points at physical page
 - Not Valid \Rightarrow Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
 - Memory Management Unit (MMU) traps to OS
 - » Resulting trap is a “Page Fault”
 - What does OS do on a Page Fault?:
 - » Choose location to fill (might replace an old page)
 - » 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
 - Could be slow!
 - » While pulling pages off disk for one process, OS runs another process from ready queue

cache

Management & Access to the Memory Hierarchy



Recall: Some following questions

- 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
 - 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

Demand Paging Cost Model

- Since Demand Paging like caching, can compute average access time! (“Effective Access Time”)
 - $EAT = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time}$
 - $EAT = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$
- Example:
 - Memory access time = 200 nanoseconds
 - Average page-fault service time = 8 milliseconds
 - Suppose p = Probability of miss, $1-p$ = Probability of hit
 - Then, we can compute EAT as follows:
$$\begin{aligned} EAT &= 200\text{ns} + p \times 8 \text{ ms} \\ &= 200\text{ns} + p \times 8,000,000\text{ns} \end{aligned}$$
- If one access out of 1,000 causes a page fault, how much worse?
 - 10%?, 50%?, 200%?
- $200 + .001 \times 8,000,000 = 8.2 \mu\text{s} = 4000\% \text{ Slowdown!}$
- P for 10% slowdown?
 - $200\text{ns} \times 1.1 < EAT \Rightarrow p < 2.5 \times 10^{-6}$
 - This is about 1 page fault in 400000!

What Factors Lead to Misses?

- 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 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:

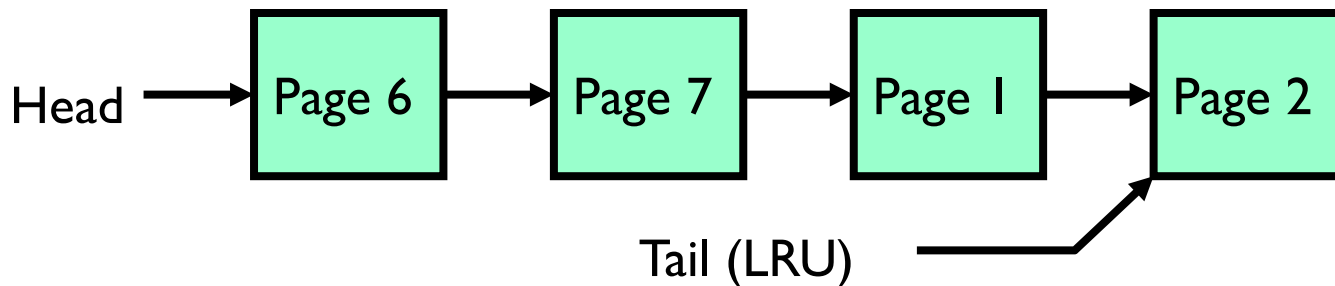
- Caused when pages were in memory, but kicked out prematurely because of the replacement policy
- How to fix? Better replacement policy

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, but can't really know future...
 - Makes good comparison case, however

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 each page used so that can change position in list...
 - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

Example: FIFO

- 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

- 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!

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):

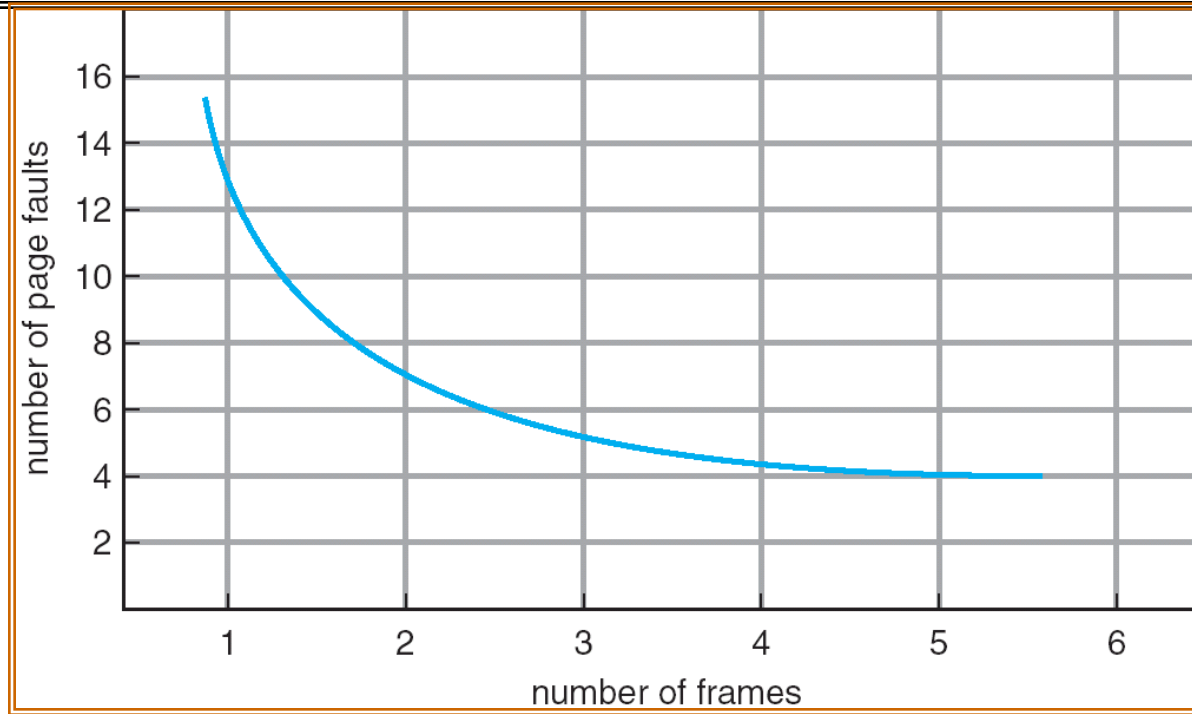
| Ref: | A | B | C | D | A | B | C | D | A | B | C | D |
|-------|---|---|---|---|---|---|---|---|---|---|---|---|
| Page: | | | | | | | | | | | | |
| 1 | A | | | D | | | C | | | B | | |
| 2 | | B | | | A | | | D | | | C | |
| 3 | | | C | | | B | | | A | | | D |

— Every reference is a page fault!

- MIN Does much better:

| Ref: | A | B | C | D | A | B | C | D | A | B | C | D |
|-------|---|---|---|---|---|---|---|---|---|---|---|---|
| Page: | | | | | | | | | | | | |
| 1 | A | | | | | | | | | B | | |
| 2 | | B | | | | | C | | | | | |
| 3 | | | C | D | | | | | | | | |

Graph of Page Faults Versus The Number of Frames



- One desirable property: When you add memory the miss rate drops
 - Does this always happen?
 - Seems like it should, right?
- No: Bélády's anomaly
 - Certain replacement algorithms (FIFO) don't have this obvious property!

Administrivia

- Project 2 design doc due today Wed 03/15
- Peer review is *NOT* optional
 - Every person must fill out the project 1 peer review
 - Due Sun 03/19
 - » Failure to submit will hurt your participation score
 - The peer review is an important part of our evaluation of partner dynamics – *please take is very seriously*

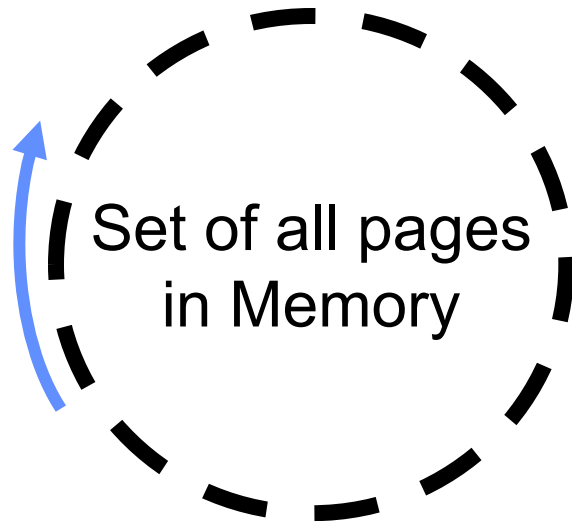
Administrivia

- Midterm 2 next week on **Tue 03/21 7-8:30PM**
 - All topics up to and including Lecture 15
 - » Focus will be on Lectures 9 – 15 and associated readings
 - » Projects 1 & 2, Homework 0 – 2
 - » Discussions
 - Closed book with 2 pages of hand-written notes both sides
 - Room assignments by **last name**:
 - » **A-H** 100 Genetics and Plant Biology Building
 - » **I-Z** 1 Pimentel
 - Review Session on Saturday
 - » 3-6pm 2050 VLSB

BREAK

Implementing LRU

- Perfect:
 - Timestamp page on each reference
 - Keep list of pages ordered by time of reference
 - Too expensive to implement in reality for many reasons
- **Clock Algorithm:** Arrange physical pages in circle with single clock hand
 - Approximate LRU (*approximation to approximation to MIN*)
 - Replace **an** old page, not **the oldest** page



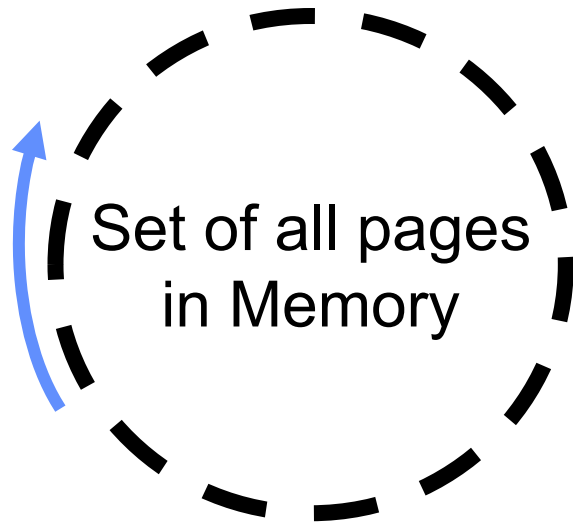
Hardware Does:

- On each reference:
- Set "use" bit (Mark pages as used recently)

OS Does:

- Advance clock hand once per page fault
- Clear "use" bit in PTE (mark page as not used recently)

Clock Algorithm: Not Recently Used



OS Does:

- Advance clock hand once per page fault
- Clear "use" bit in PTE (mark page as not used recently)

Hardware Does:

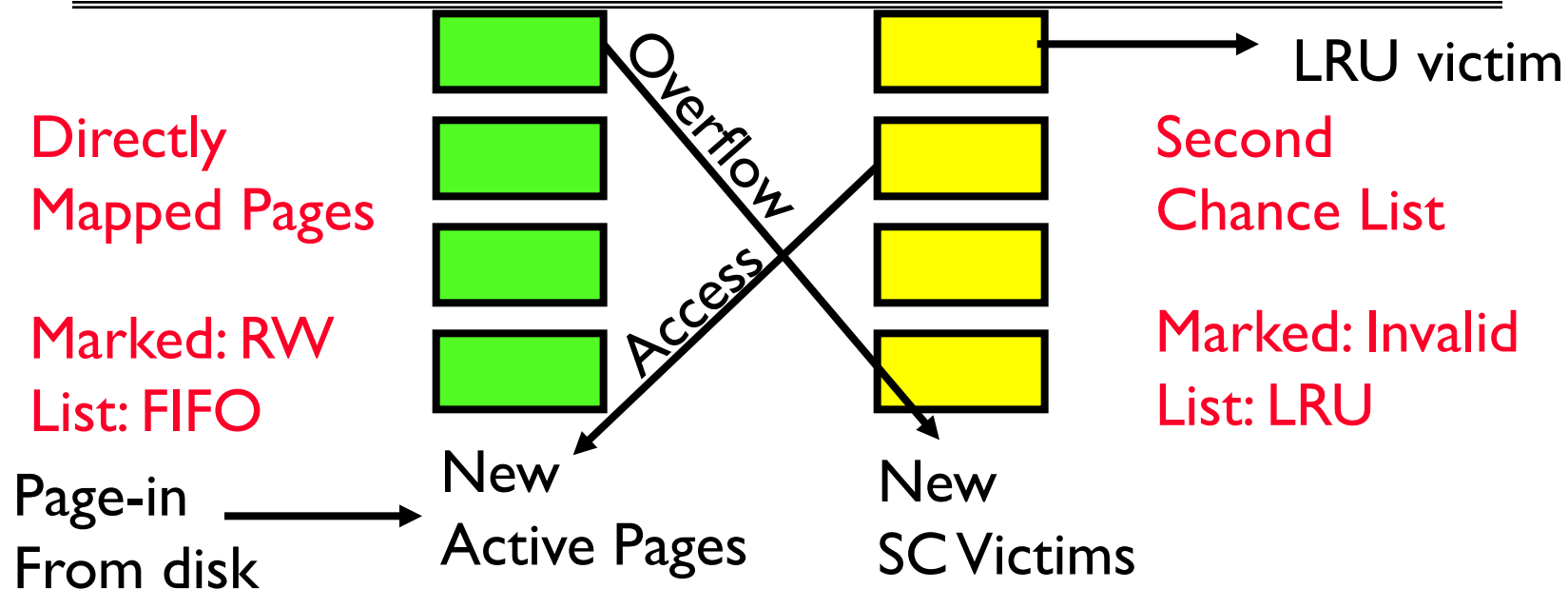
- On each reference:
- Set "use" bit (Mark pages as used recently)

- Will always find a page or loop forever?
 - Even if all use bits set, will eventually loop around \Rightarrow FIFO
- One way to view clock algorithm:
 - Crude partitioning of pages into two groups: young and old
 - Why not partition into more than 2 groups?

Nth Chance version of Clock Algorithm

- Nth chance algorithm: Give page N chances
 - OS keeps counter per page: # sweeps
 - On page fault, OS checks use bit:
 - » 1 → clear use and also clear counter (used in last sweep)
 - » 0 → increment counter; if count=N, replace page
 - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
 - Why pick large N? Better approximation to LRU
 - » If $N \sim 1K$, really good approximation
 - Why pick small N? More efficient
 - » Otherwise might have to look a long way to find free page
- What about dirty pages?
 - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
 - Common approach:
 - » Clean pages, use $N=1$
 - » Dirty pages, use $N=2$ (and write back to disk when $N=1$)

Second-Chance List Algorithm (VAX/VMS)



- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
 - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
 - Desired Page On SC List: move to front of Active list, mark RW
 - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list

Second-Chance List Algorithm (con't)

- How many pages for second chance list?
 - If 0 \Rightarrow FIFO
 - If all \Rightarrow LRU, but page fault on every page reference
- Pick intermediate value. Result is:
 - Pro: Few disk accesses (page only goes to disk if unused for a long time)
 - Con: Increased overhead trapping to OS (software / hardware tradeoff)
- One Motivation: No HW-supported "use" bit
 - Strecker (architect) asked OS people, they said they didn't need it, so didn't implement it
 - He later got blamed, but VAX did OK anyway

Allocation of Page Frames (Memory Pages)

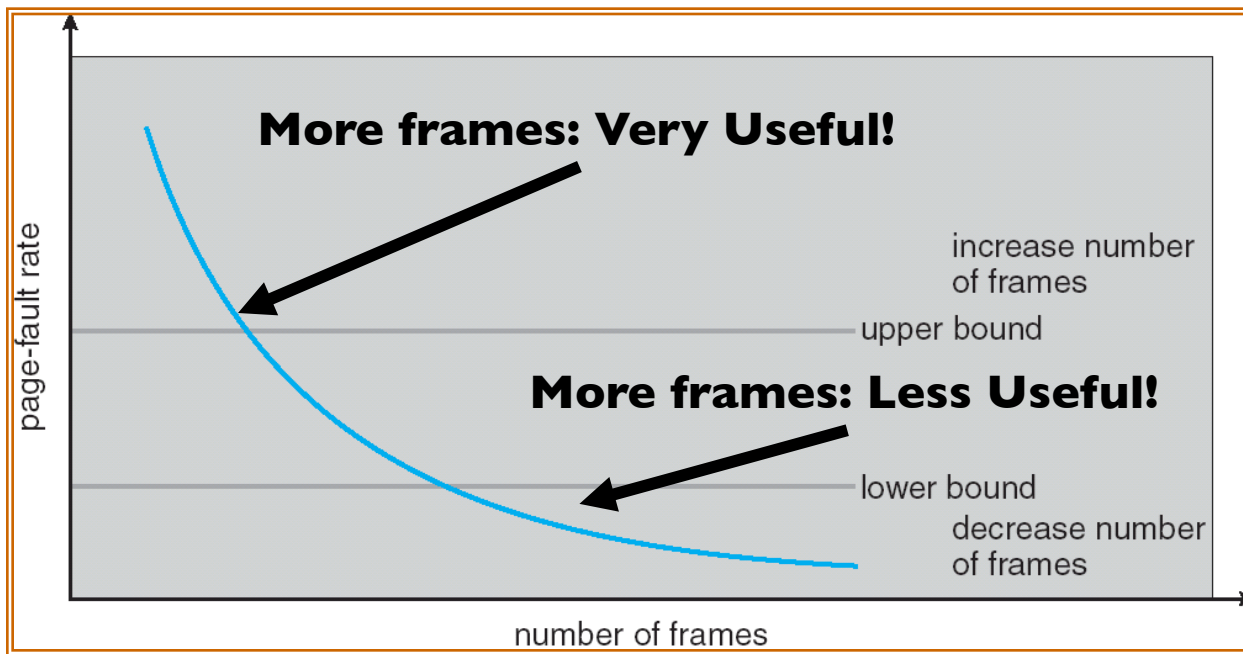
- How do we allocate memory among different processes?
 - Does every process get the same fraction of memory? Different fractions?
 - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
 - Want to make sure that all processes **that are loaded into memory** can make forward progress
 - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
 - » instruction is 6 bytes, might span 2 pages
 - » 2 pages to handle *from*
 - » 2 pages to handle *to*
- Possible Replacement Scopes:
 - **Global replacement** – process selects replacement frame from set of all frames; one process can take a frame from another
 - **Local replacement** – each process selects from only its own set of allocated frames

Fixed/Priority Allocation

- **Equal allocation** (Fixed Scheme):
 - Every process gets same amount of memory
 - Example: 100 frames, 5 processes → process gets 20 frames
- **Proportional allocation** (Fixed Scheme)
 - Allocate according to the size of process
 - Computation proceeds as follows:
 - s_i = size of process p_i and $S = \sum s_i$
 - m = total number of frames
 - $$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$
- **Priority Allocation:**
 - Proportional scheme using priorities rather than size
 - » Same type of computation as previous scheme
 - Possible behavior: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number
- Perhaps we should use an adaptive scheme instead???
 - What if some application just needs more memory?

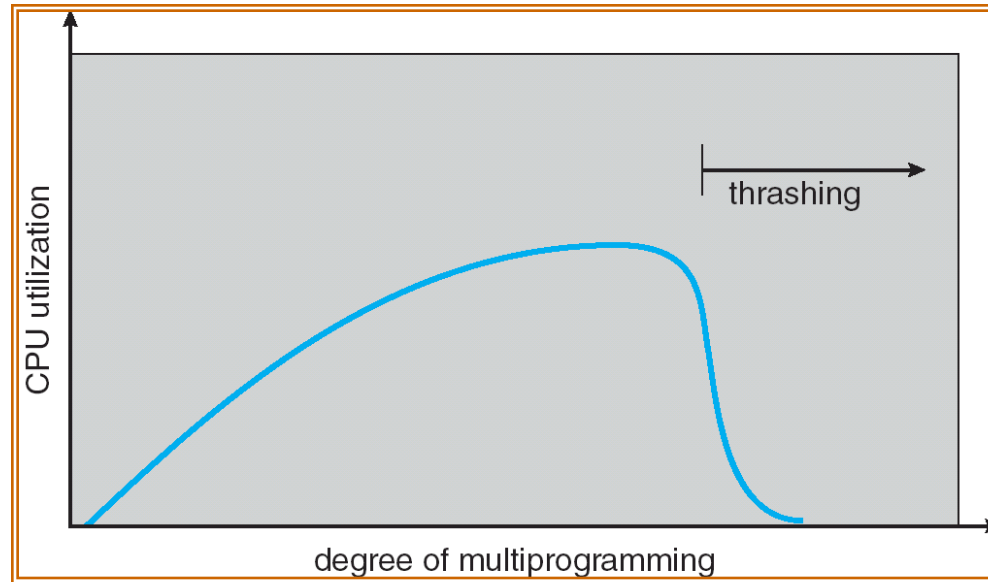
Page-Fault Frequency Allocation

- Can we reduce Capacity misses by dynamically changing the number of pages/application?



- Establish “acceptable” page-fault rate
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame
- Question: What if we just don’t have enough memory?

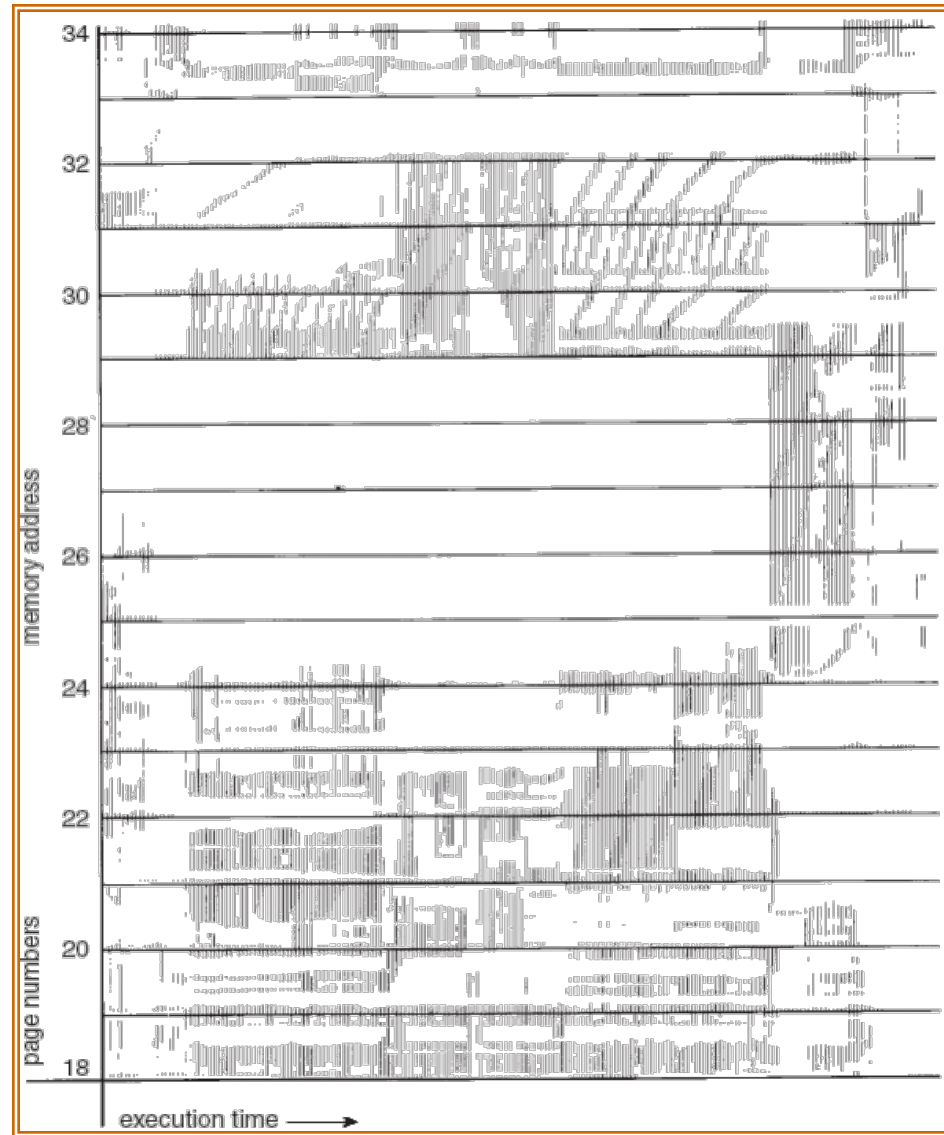
Thrashing



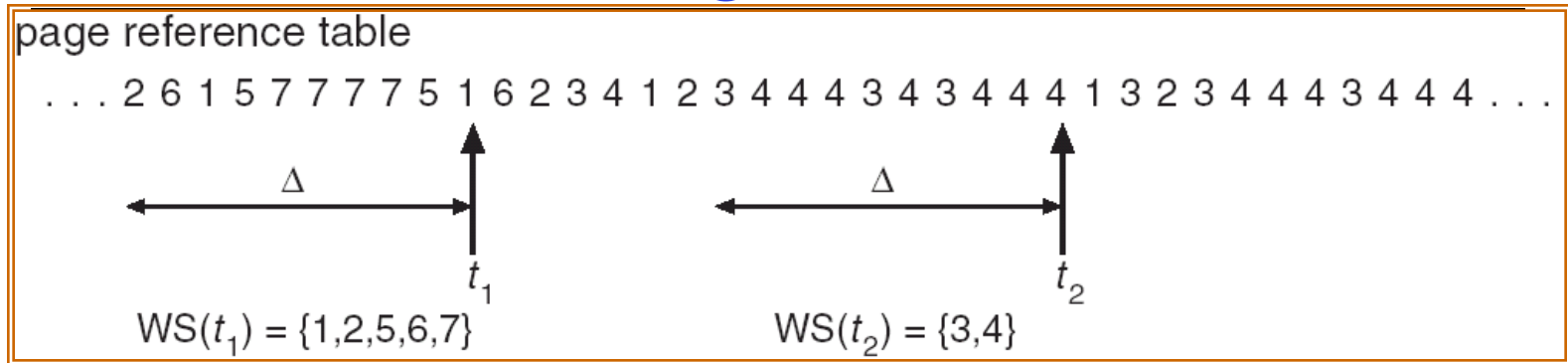
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
 - low CPU utilization
 - operating system spends most of its time swapping to disk
- **Thrashing** \equiv a process is busy swapping pages in and out
- Questions:
 - How do we detect Thrashing?
 - What is best response to Thrashing?

Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
 - Group of Pages accessed along a given time slice called the “Working Set”
 - Working Set defines minimum number of pages needed for process to behave well
- Not enough memory for Working Set \Rightarrow Thrashing
 - Better to swap out process?



Working-Set Model



- $\Delta \equiv$ working-set window \equiv fixed number of page references
 - Example: 10,000 instructions
- WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum |WS_i| \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
 - Policy: if $D > m$, then suspend/swap out processes
 - This can improve overall system behavior by a lot!

What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
 - Pages that are touched for the first time
 - Pages that are touched after process is swapped out/swapped back in
- Clustering:
 - On a page-fault, bring in multiple pages “around” the faulting page
 - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Working Set Tracking:
 - Use algorithm to try to track working set of application
 - When swapping process back in, swap in working set

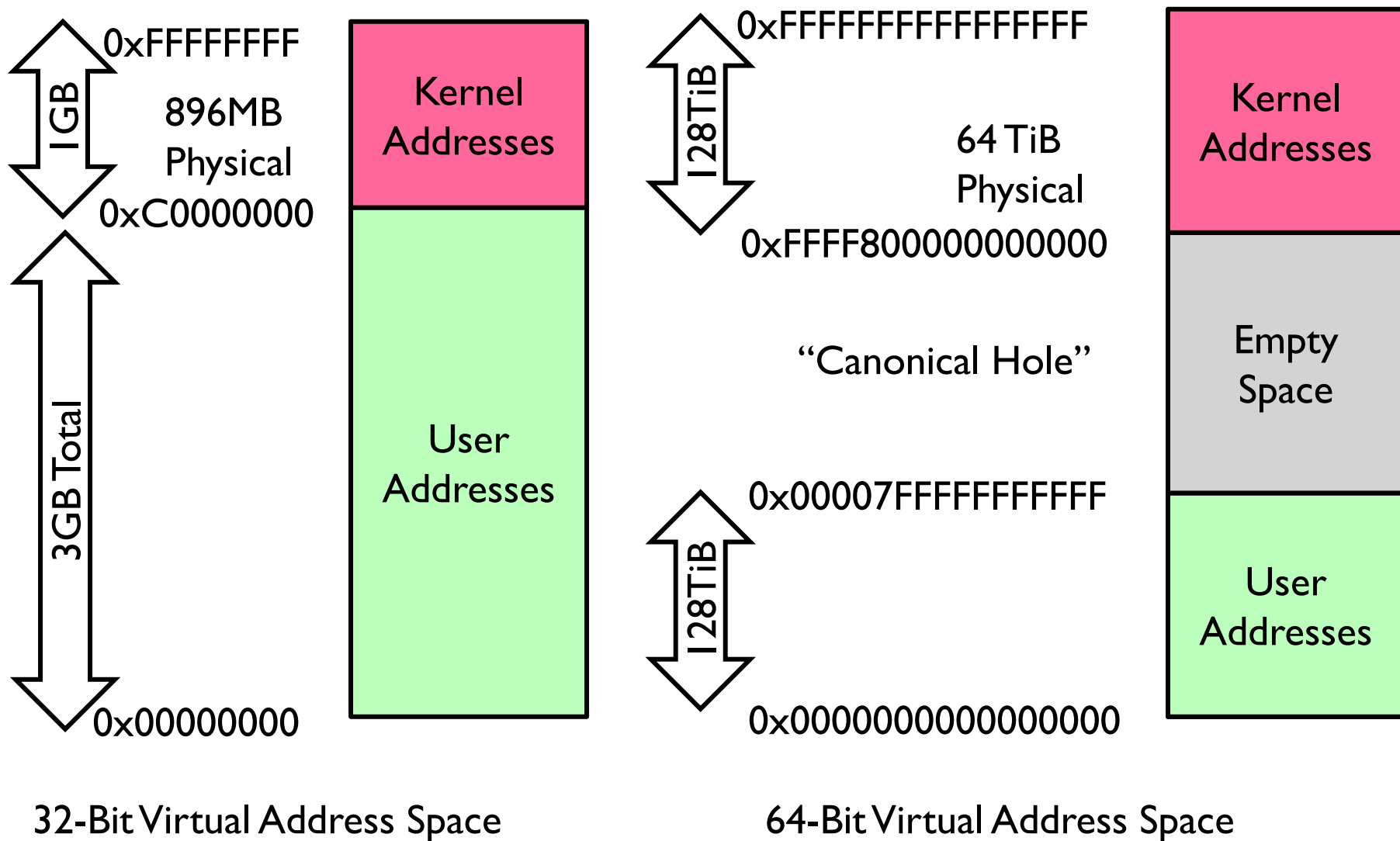
Reverse Page Mapping (Sometimes called “Coremap”)

- Physical page frames often shared by many different address spaces/page tables
 - All children forked from given process
 - Shared memory pages between processes
- Whatever reverse mapping mechanism that is in place must be very fast
 - Must hunt down all page tables pointing at given page frame when freeing a page
 - Must hunt down all PTEs when seeing if pages “active”
- Implementation options:
 - For every page descriptor, keep linked list of page table entries that point to it
 - » Management nightmare – expensive
 - Linux 2.6: Object-based reverse mapping
 - » Link together memory region descriptors instead (much coarser granularity)

Linux Memory Details?

- Memory management in Linux considerably more complex than the previous indications
- Memory Zones: physical memory categories
 - ZONE_DMA: < 16MB memory, DMAable on ISA bus
 - ZONE_NORMAL: 16MB → 896MB (mapped at 0xC0000000)
 - ZONE_HIGHMEM: Everything else (> 896MB)
- Each zone has 1 freelist, 2 LRU lists (Active/Inactive)
- Many different types of allocation
 - SLAB allocators, per-page allocators, mapped/unmapped
- Many different types of allocated memory:
 - Anonymous memory (not backed by a file, heap/stack)
 - Mapped memory (backed by a file)
- Allocation priorities
 - Is blocking allowed/etc

Recall: Linux Virtual memory map



Virtual Map (Details)

- Kernel memory not generally visible to user
 - Exception: special VDSO (virtual dynamically linked shared objects) facility that maps kernel code into user space to aid in system calls (and to provide certain actual system calls such as **gettimeofday()**)
- Every physical page described by a “page” structure
 - Collected together in lower physical memory
 - Can be accessed in kernel virtual space
 - Linked together in various “LRU” lists
- For 32-bit virtual memory architectures:
 - When physical memory < 896MB
 - » All physical memory mapped at 0xC0000000
 - When physical memory ≥ 896MB
 - » Not all physical memory mapped in kernel space all the time
 - » Can be temporarily mapped with addresses > 0xCC000000
- For 64-bit virtual memory architectures:
 - All physical memory mapped above 0xFFFF800000000000

Summary

- 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
- Clock Algorithm: Approximation to LRU
 - Arrange all pages in circular list
 - Sweep through them, marking as not “in use”
 - If page not “in use” for one pass, then can replace
- N^{th} -chance clock algorithm: Another approximate LRU
 - Give pages multiple passes of clock hand before replacing
- Second-Chance List algorithm: Yet another approximate LRU
 - Divide pages into two groups, one of which is truly LRU and managed on page faults.
- Working Set:
 - Set of pages touched by a process recently
- Thrashing: a process is busy swapping pages in and out
 - Process will thrash if working set doesn't fit in memory
 - Need to swap out a process