slides kindly provided by:

Tam Vu, Ph.D
Professor of Computer Science
Director, Mobile & Networked Systems Lab
Department of Computer Science
University of Colorado Boulder

CSCI 3753 Operating Systems
CSCI 3753 – Spring 2019

Christopher Godley

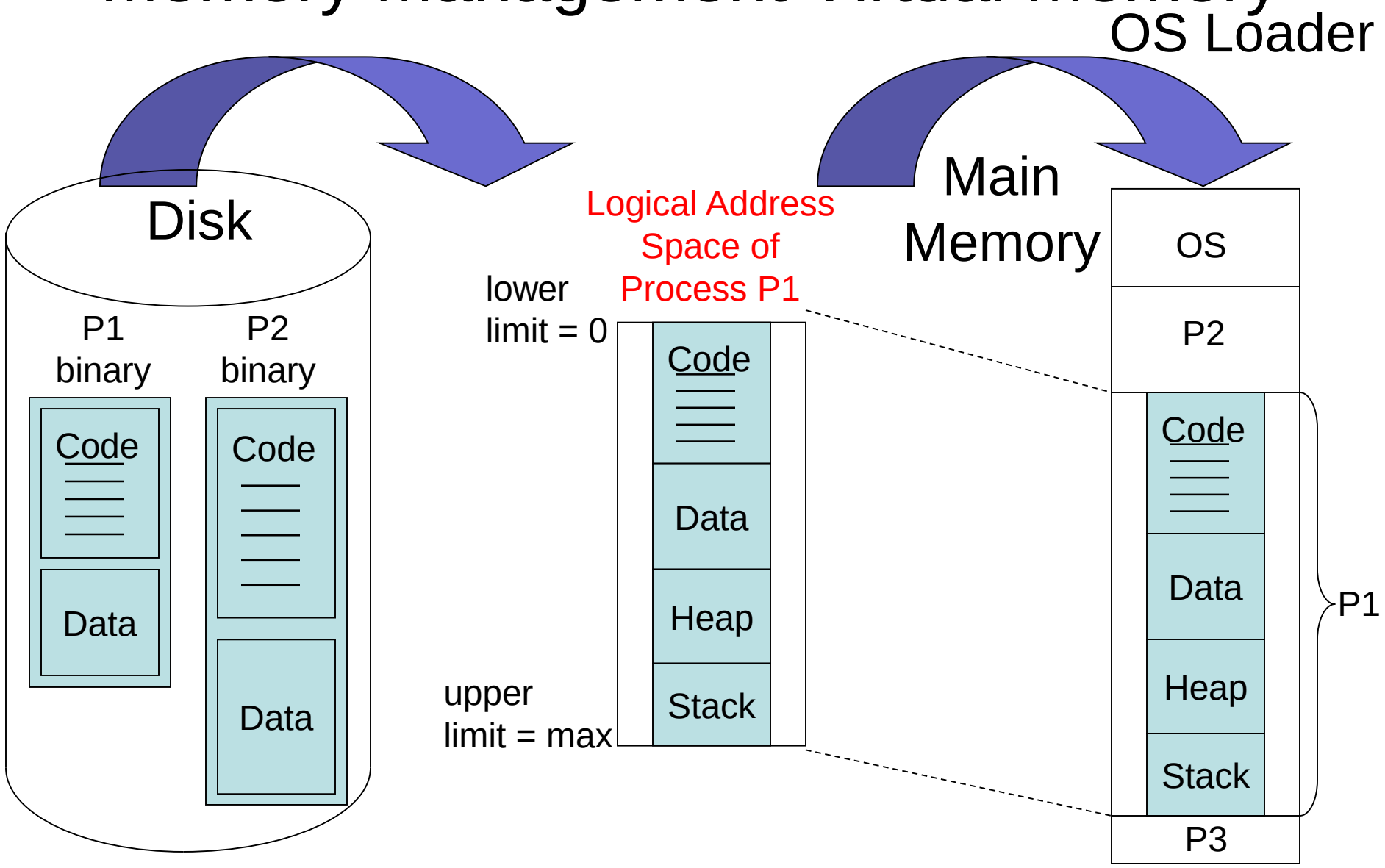
PhD Student

Department of Computer Science

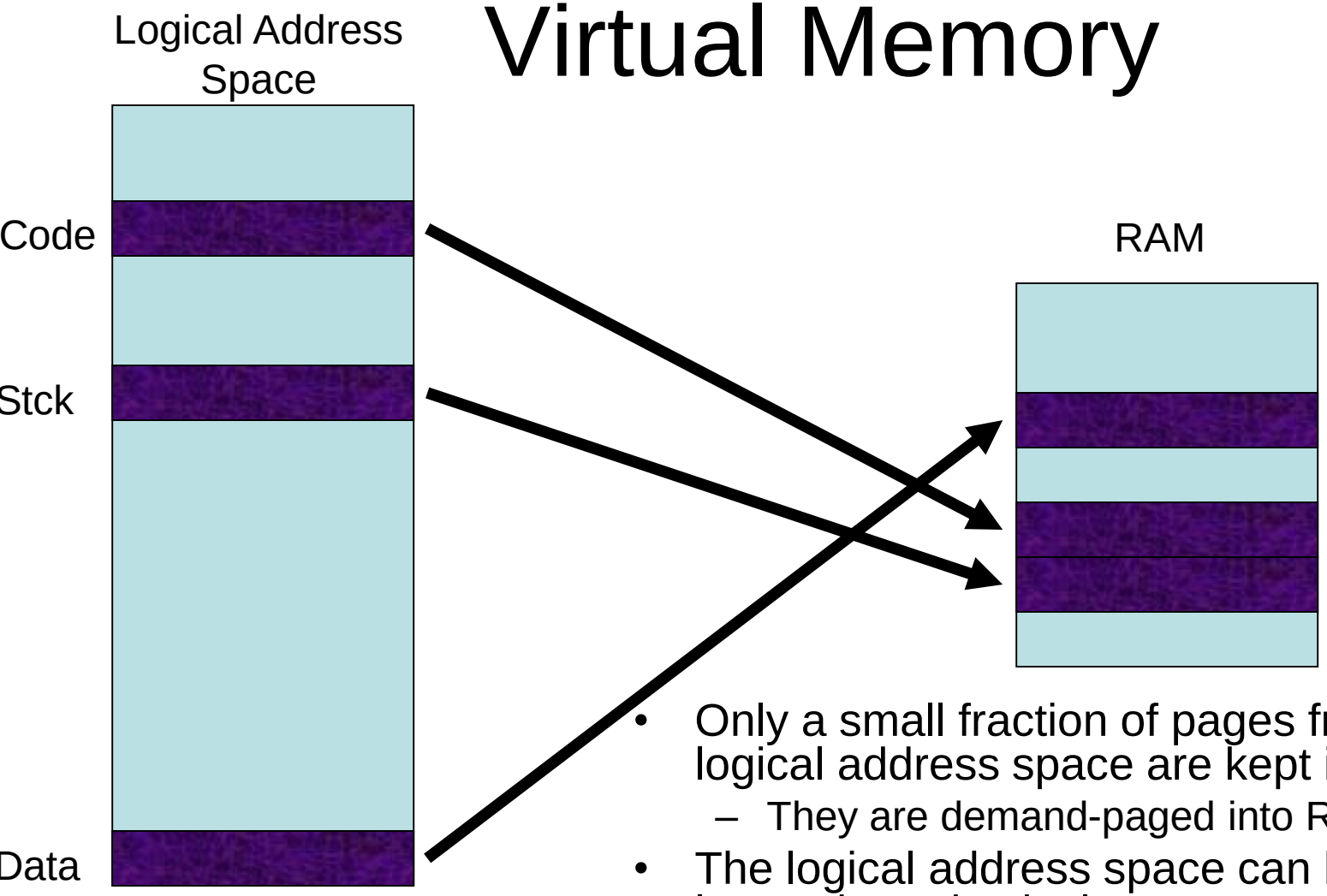
University of Colorado Boulder

Virtual Memory

Memory Management Virtual Memory



Virtual Memory



- Only a small fraction of pages from the logical address space are kept in RAM
  - They are demand-paged into RAM
- The logical address space can be much larger than physical RAM
  - Swap in the pages needed, when needed

On-Demand Paging

- Page tables may be large, consuming much RAM
- Key observation: not all pages in a logical address space need to be kept in memory
  - in the simplest case, just keep the current page where the PC is executing
  - all other pages could be on disk
  - when another page is needed, retrieve the page from disk and place in memory before executing
    - this would be costly and slow, because it would happen every time that a page different from the current one is needed

On-Demand Paging

- Instead of just keeping one page, keep a *subset* of a process's pages in memory
  - Load just what you need, not the entire address space
  - use memory as a cache of frequently or most recently used pages
  - rely on a program's behavior to exhibit locality of reference
    - if an instruction or data reference in a page was previously invoked, then that page is likely to be referenced again in the near future

On-Demand Paging

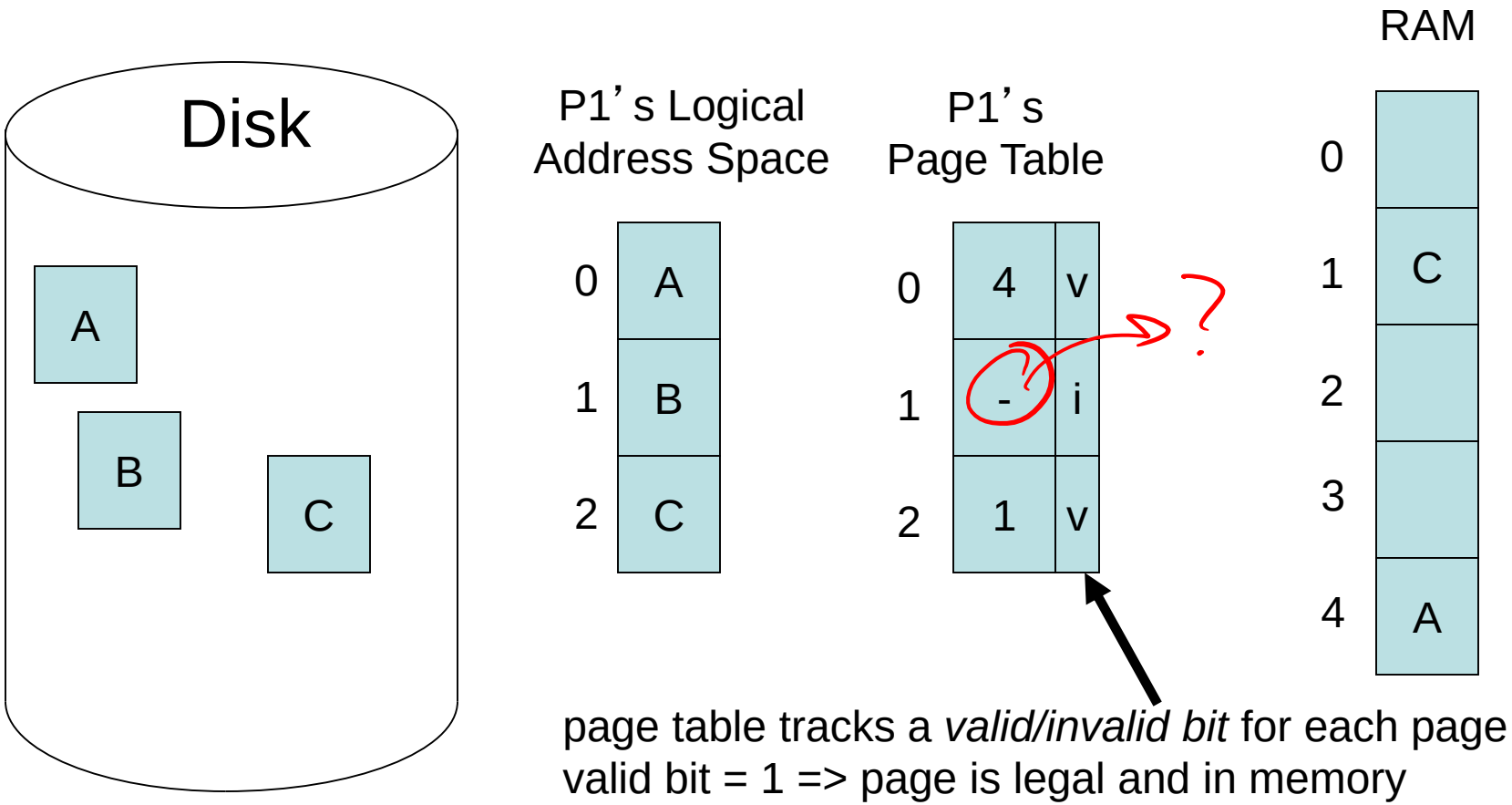
- Most programs exhibit some form of **locality**
  - looping locally through the same set of instructions
  - branching through the same code
  - executing linearly, the next instruction is typically the next one immediately after the previous instruction, rather than some random jump
- Thus process execution revisits pages already stored in memory
  - so you don’t have to go to disk each time the program counter (PC) jumps to a different page

On-Demand Paging

- On-demand paging is used to page in new pages from disk to RAM
  - only when a page is needed is it loaded into memory from disk
- Can page in an **entire process on demand**:
  - starting with “zero” pages
  - the reference to the first instruction causes the first page of the process to be loaded on demand into RAM.
  - Subsequent pages are loaded on demand into RAM as the process executes.

On-Demand Paging

- On-demand paging loads a page from disk into RAM only when needed
  - in the example below, pages A and C are in memory, but page B is not

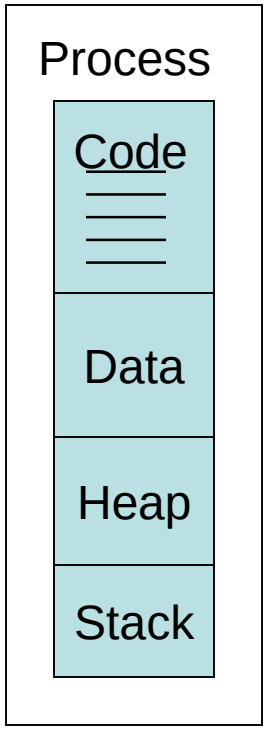


Virtual Memory Advantages

1. Virtual address space can now exceed physical RAM!
   - Only a subset (most demanded) of pages are kept in RAM
   - We have decoupled virtual memory from physical memory.
2. Fit many more processes in memory

Virtual Memory Advantages

3. Decreases swap time
   – there is less to swap
4. Can have large sparse address spaces
   – most of the address space is unused
   – Does not take up physical RAM
   – a large heap and stack that is mostly unused/empty won't take up any actual RAM until needed



On-Demand Paging

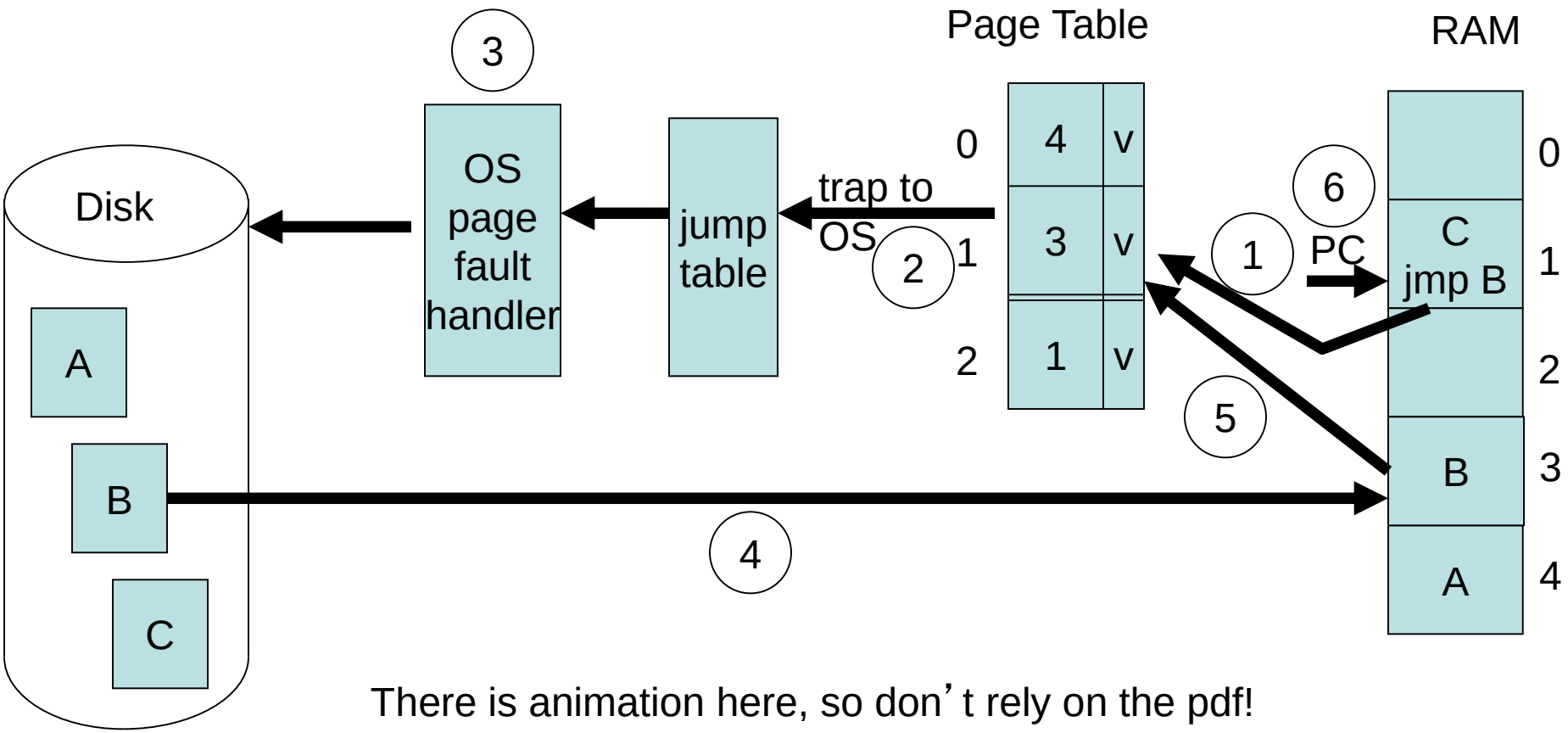
- Open questions to be answered:
  - how is a needed page loaded into memory from disk?
  - how many pages in memory should be allocated to a process?
  - if the # of pages allocated to a process is exceeded, i.e. the cache is full, then how do you choose which page to replace?

Virtual Memory

- Page-fault steps to load a page into RAM:
  1. MMU detects a page is not in memory (invalid bit set) which causes a *page-fault trap* to OS
  2. OS saves registers and process state. Determines that the fault was due to demand paging and jumps to page fault handler
  3. *Page fault handler*
     a) If reference to page not in logical A.S. then seg fault.
     b) Else if reference to page in logical A.S., but not in RAM, then load page
     c) OS finds a free frame
     d) OS schedules a disk read. Other processes may run in meantime.
  4. Disk returns with interrupt when done reading desired page. OS writes desired page into free frame
  5. OS updates page table, sets valid bit of page and its physical location
  6. Restart interrupted instruction that caused the page fault

Virtual Memory

On-demand paging causes a page fault, which goes through the following steps



There is animation here, so don't rely on the pdf!

Virtual Memory

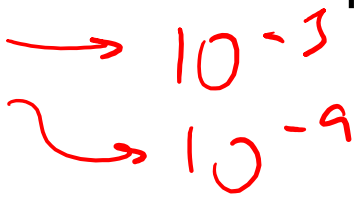
- OS can retrieve the desired page either from the disk's
  - file system, or
  - swap space/backing store
    - faster, avoids overhead of file system lookup
- pages can be in swap space because:
  - the entire executable file was copied into swap space when the program was first started.
    - Avoids file system, but also allows the copied executable to be laid out contiguously on disk's swap space, for faster access to pages (no seek time)

Virtual Memory

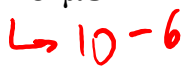
- pages can be in swap space because:
  - as pages have to be replaced in RAM, they are written to swap space instead of the file system's portion of disk.
    - The next time they're needed, they're retrieved quickly from swap space, avoiding a file system lookup.

Performance of On-Demand Paging

- Want to limit the number/frequency of page faults, which cause a read from disk, which slows performance
  - disk read is about 10 ms
  - memory read is about 10 ns



- What is the average memory access time?
  - average access time = p*10 ms + (1-p)*10 ns, where p = probability of a page fault.
  - if p=.001, then average access time = 10 μs >> 10 ns (1000 X greater!)

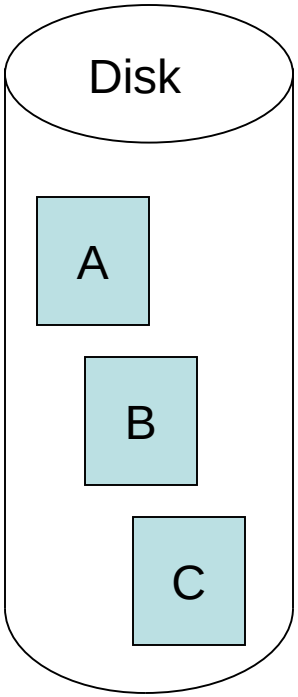


  - to keep average access time within 10% of 10 ns, would need a page fault rate lower than $p<10^{-7}$
  - Reducing page fault frequency improves performance in a big way

Page Replacement Policies

Page Replacement Policies

- Demand paging loads a page from disk into RAM only when needed
  - in the example below, pages A and C in memory, but page B is not
  - How can we access B if all other frames are already used



P1's Address Space

0	A
1	B
2	C

B is not in memory and there are no frames available

P1's Page Table

0	4	v
1	-	i
2	1	v

RAM

Y	0
C	1
X	2
Z	3
A	4

Must decided which page is to be evicted from memory

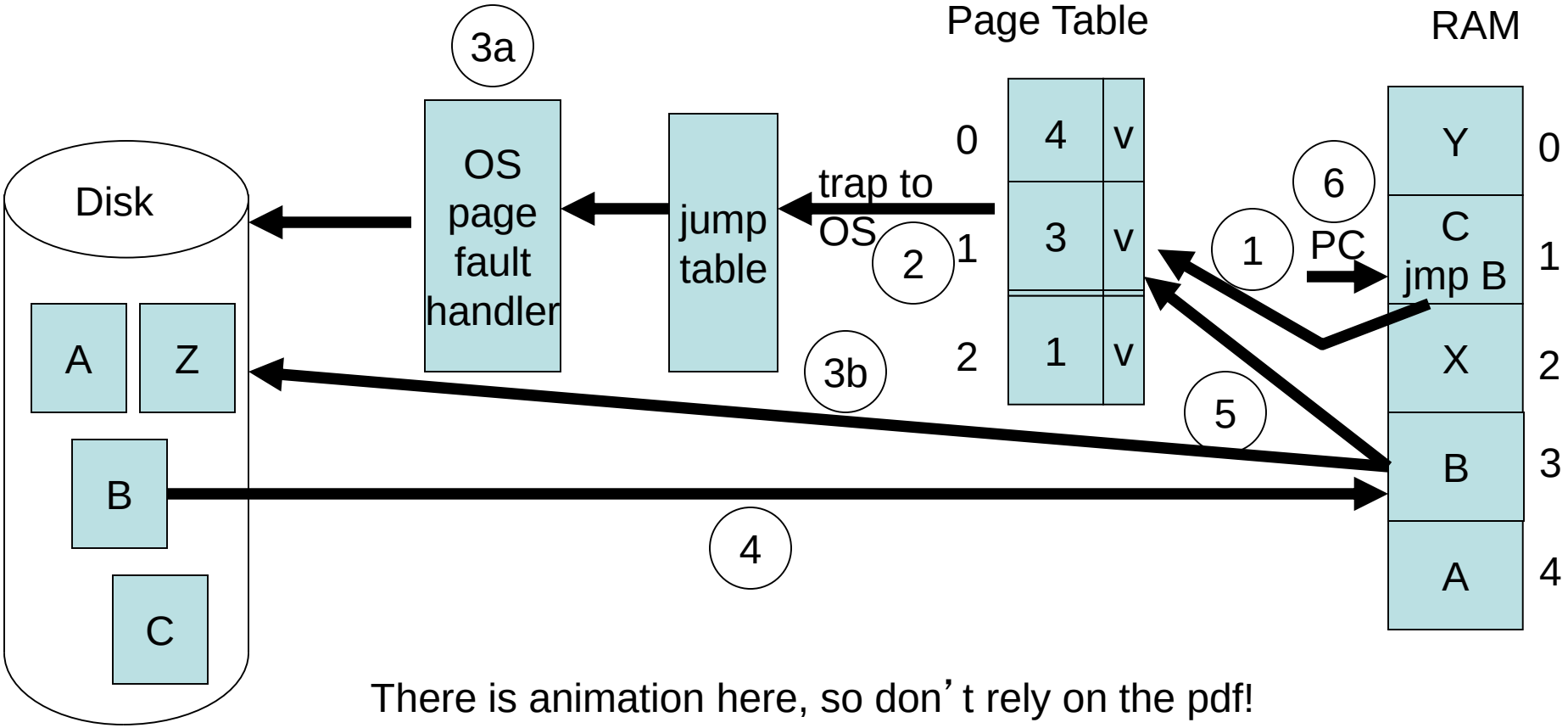
Page Replacement Policies

- As processes execute and bring in more pages on demand into memory, eventually the system runs out of free frames
  - need a *page replacement policy*
  - *Steps:*
    1. select a victim frame that is not currently being used
    2. save or write the victim frame to disk, update the page table (page now invalid)
    3. load in the new desired page from disk
  - If out of free frames, each page fault causes 2 disk operations, one to write the victim, and one to read the desired page

    - this is a big performance penalty

Page Replacement Policies

In Step 3b, we modify traditional on-demand paging to include a write of the victim frame to disk



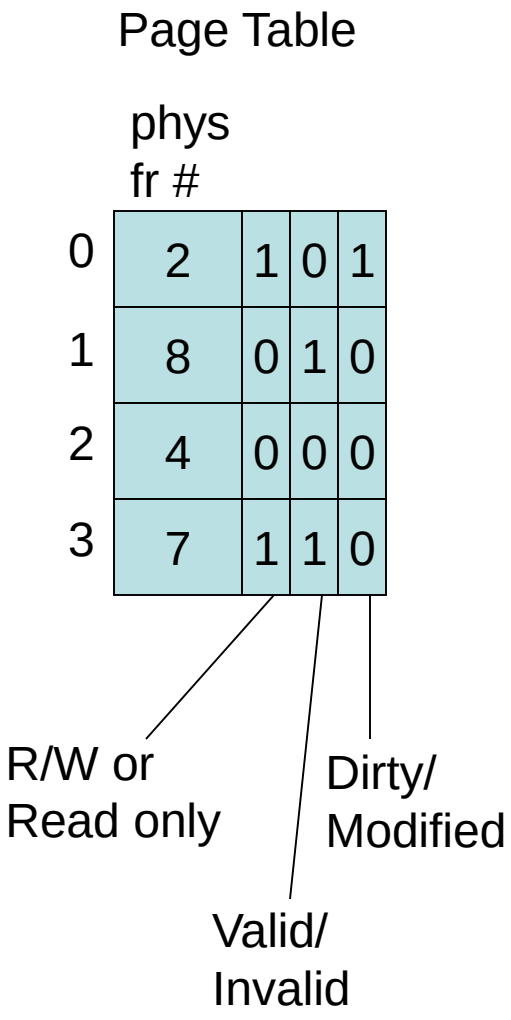
There is animation here, so don't rely on the pdf!

Page Replacement Policies

- To reduce the performance penalty of 2 disk operations, systems can employ a *dirty/modify bit*
  - modify bit = 0 initially
  - when a page in memory is written to, set the bit = 1
  - when a victim page is needed, select a page that has not been modified (dirty bit = 0)
    - such an unmodified page need not be written to disk, because its mirror image is already on disk!
    - this saves on disk I/O - reduces to only 1 disk operation (read of desired page)

Page Table Status Bits

- Each entry in the page table can conceptually store several extra bits of metadata information along with the physical frame # f
  - *Valid/invalid bits* - for memory protection, accessing an invalid page causes a page fault
    - Is the logical page in the logical address space?
    - If there is virtual memory (we'll see this later), is the page in memory or not?

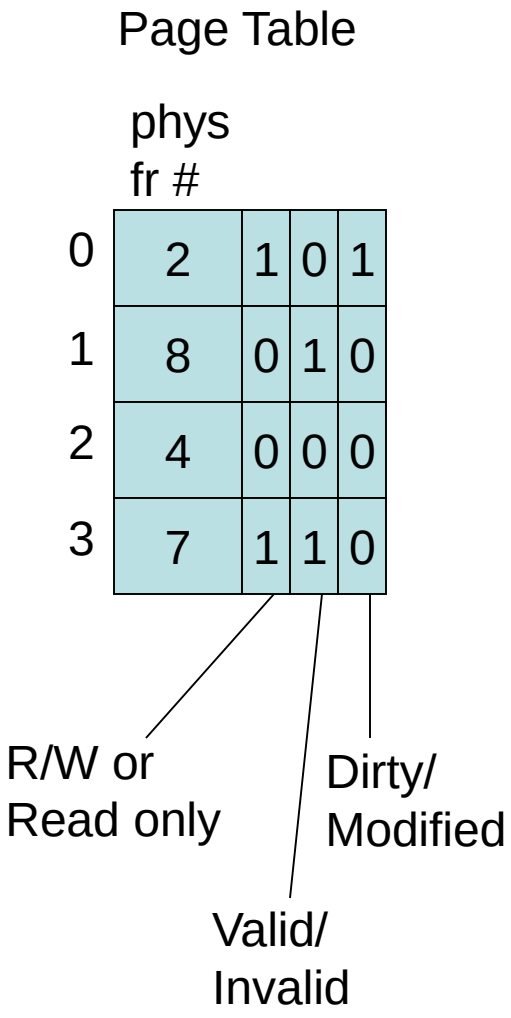


Page Table Status Bits

– *dirty bits* - has the page been modified for page replacement?

– *R/W or Read-only bits* - for memory protection, writing to a read-only page causes a fault and a trap to the OS

– *Reference bit* – useful for Clock page replacement algorithm



Recap …

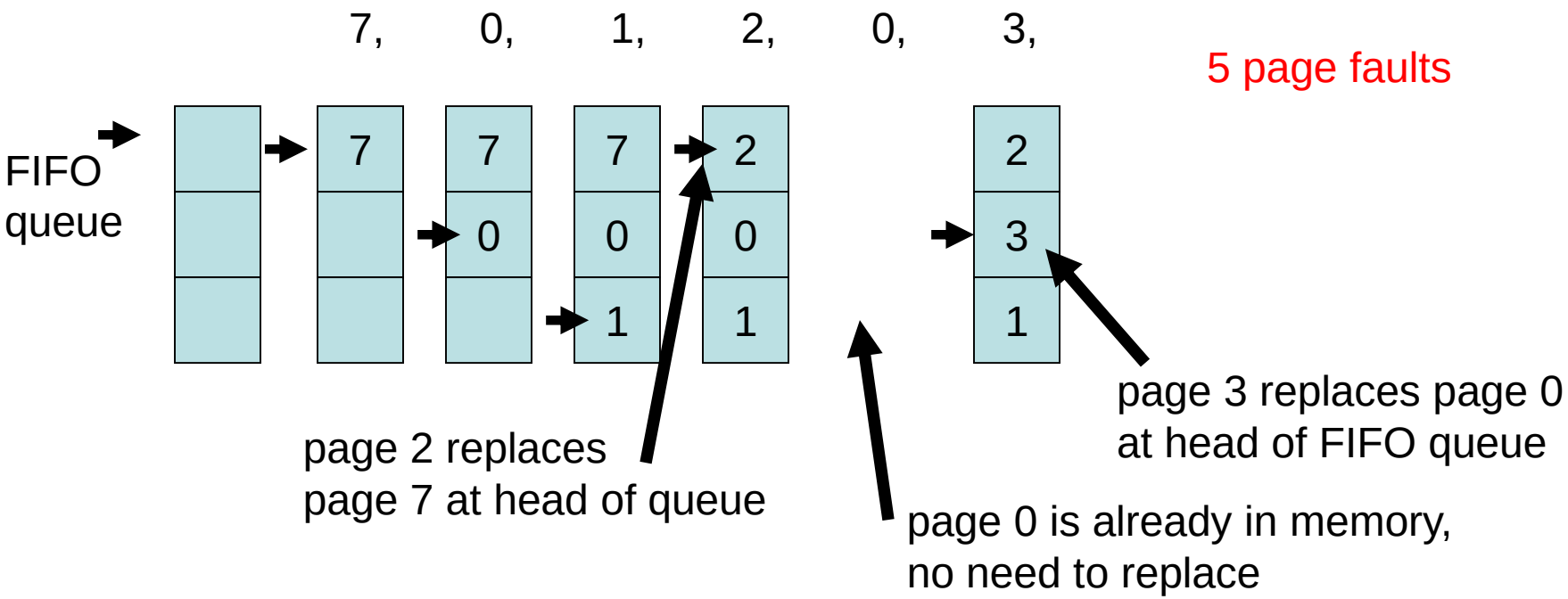
- Virtual memory
  - Keep only a few pages in memory, rest on disk
  - On-demand paging: retrieve a page when needed
  - Page fault
    - A referenced page is not loaded in memory
    - OS blocks the process and retrieves the referenced page
    - Significant performance overhead – need to keep page fault frequency low, e.g. less than 1 in 10^7 for overhead <10%
  - Page replacement algorithm
    - Principle of locality of reference
    - Dirty bit: choose a clean page before a dirty page

Page Replacement Policies

- Evaluation
  - Variables: algorithm, page reference sequence, # of memory frames
  - algorithm with lowest # of page faults is most desirable

FIFO Page Replacement

- FIFO - create a FIFO queue of all pages in memory
  - example reference string: 7, 0, 1, 2, 0, 3, ...
  - assume also that there are 3 frames of memory total

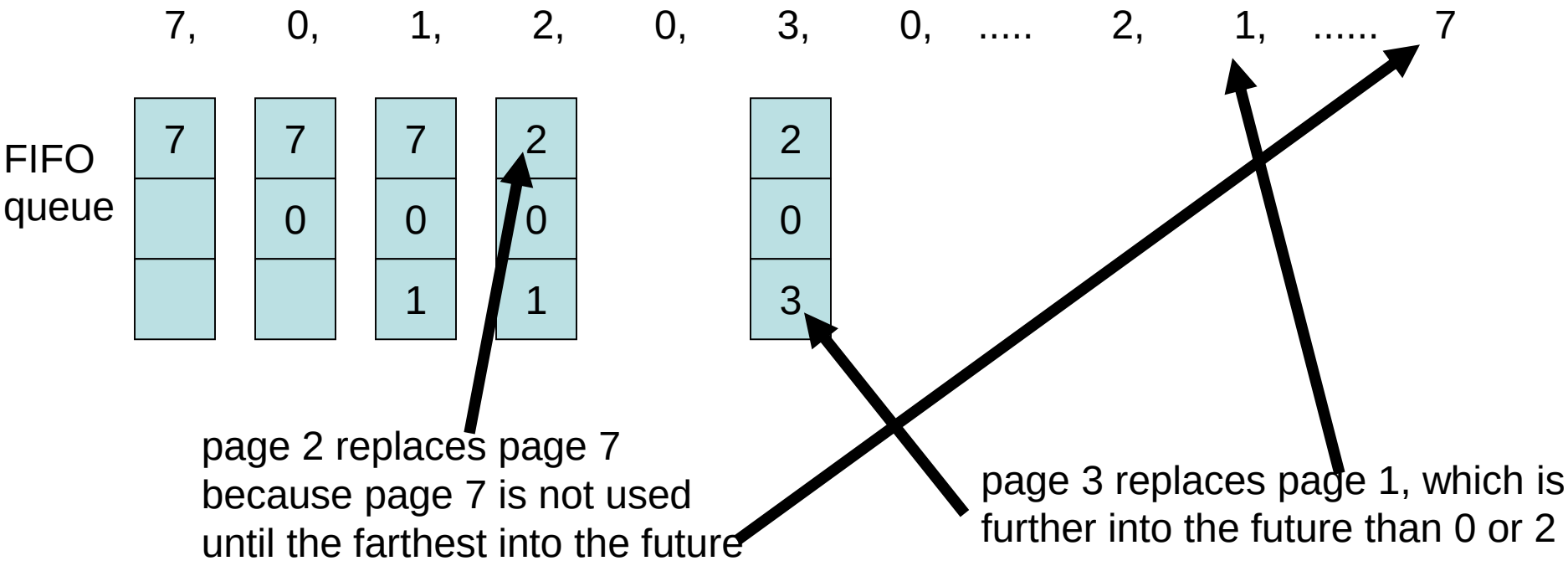


FIFO Page Replacement

- FIFO is easy to understand and implement
- Performance can be poor
  - Suppose page 7 that was replaced was a very active page that was frequently referenced, then page 7 will be referenced again very soon, causing a page fault because it's not in memory any more
  - In the worst case, each page that is paged out could be the one that is referenced next, leading to a high page fault rate
  - Ideally, keep around the pages that are about to be used next – this is the basis of the OPT algorithm in the next slide

OPT Page Replacement

- OPT = Optimal
  - Replace the page that will not be referenced for the longest time
  - Guarantees the lowest page-fault rate
  - Problem: requires future knowledge

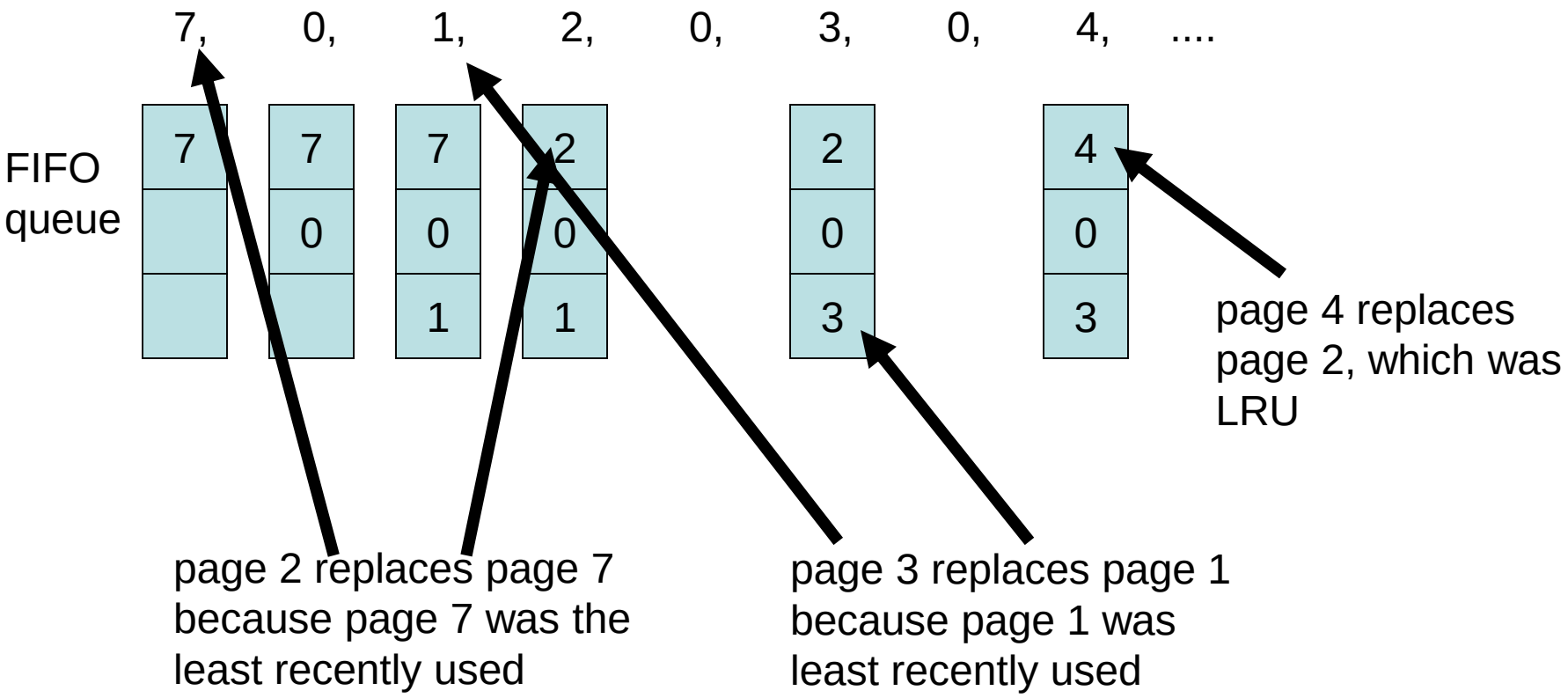


LRU Page Replacement

- LRU = Least Recently Used
  - Use the past to predict the future
    - if a page wasn't used recently, then it is unlikely to be used again in the near future
    - if a page was used recently, then it is likely to be used again in the near future
    - so select a victim that was least recently used
  - Approximation of OPT
    - page fault rate LRU > OPT, but LRU < FIFO
  - Variations of LRU are popular

LRU Page Replacement

- LRU example



LRU Implementation Options

- Keep a history of past page accesses
  - the entire history (lots of memory)
  - a sliding window

  - Complicated and slow

- Variations of LRU are popular

LRU Implementation Options

- Timers
  - keep an actual time stamp for each page as to when it was last used
  - Problem: expensive in delay (consult system clock on each page reference), storage (at least 64 bits per absolute time stamp), and search (find the page with the oldest time stamp)

LRU Implementation Options

- Counters
  - Approximate time stamp in the form of a counter that is incremented with any page reference, i.e. each page's counter must be incremented on each page reference
  - Counter is stored with that entry in the page table. Counter is reset to 0 on a reference to a page.
  - Problem: expensive
    - Must update each page's counter (on each reference)
    - Must search list

LRU Implementation Options

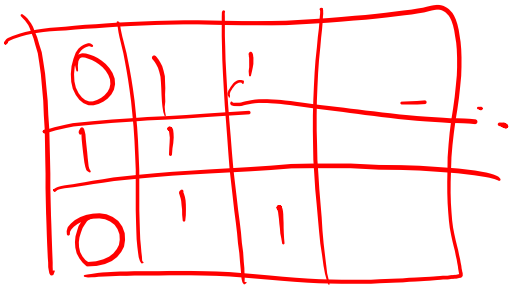
- Linked List
  - whenever a page is referenced, put it on the end of the linked list, removing if it from within the linked list if already present in list
  - Front of linked list is LRU
  - Problem: managing a (doubly) linked list and rearranging pointers becomes expensive
- Similar problems with a Stack data structure

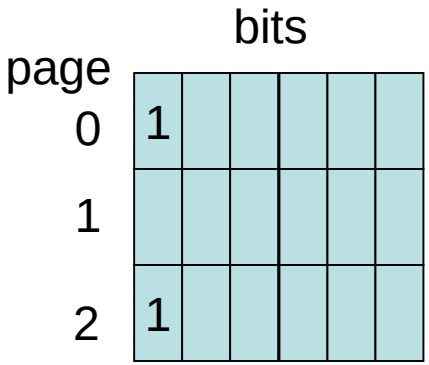
LRU approximation algorithms

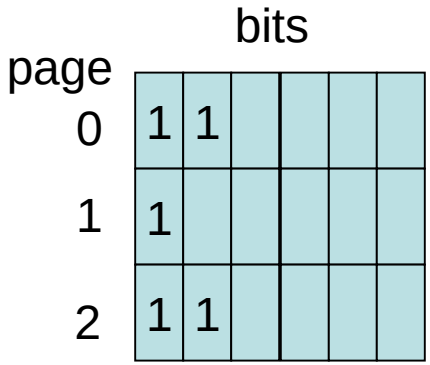
- Add an extra HW bit called a *reference bit*
  - This is set any time a page is referenced (read or write)
  - Allows OS to see what pages have been used, though not fine-grained detail on the order of use
  - Reference-bit based algorithms only *approximate* LRU, i.e. they do not seek to exactly implement LRU
  - 3 types of reference-bit LRU approximation algorithms:
    - Additional Reference-Bits Algorithm
    - Second-Chance (Clock) Algorithm
    - Enhanced Clock Algorithm with Dirty/Modify Bit

LRU approximation algorithms

- Additional Reference-Bits Algorithm
  - Record the last 8 reference bits for each page
  - Periodically a timer interrupt shifts right the bits in the record and puts the reference bit into the MSB
  - Example to compare times:
    - 11000100 > 01110111
  - So LRU = lowest valued record







Periodically shift all the bit to the right

Reference-bit based LRU approximation algorithms

- Second-Chance Algorithm
  - In-memory pages + reference bits conceptually form a *circular* queue; a hand points to a page.
  - If page pointed to has R = 0, it is replaced and the hand moves forward.
  - Otherwise, set R = 0; hand moves forward and checks the next page.

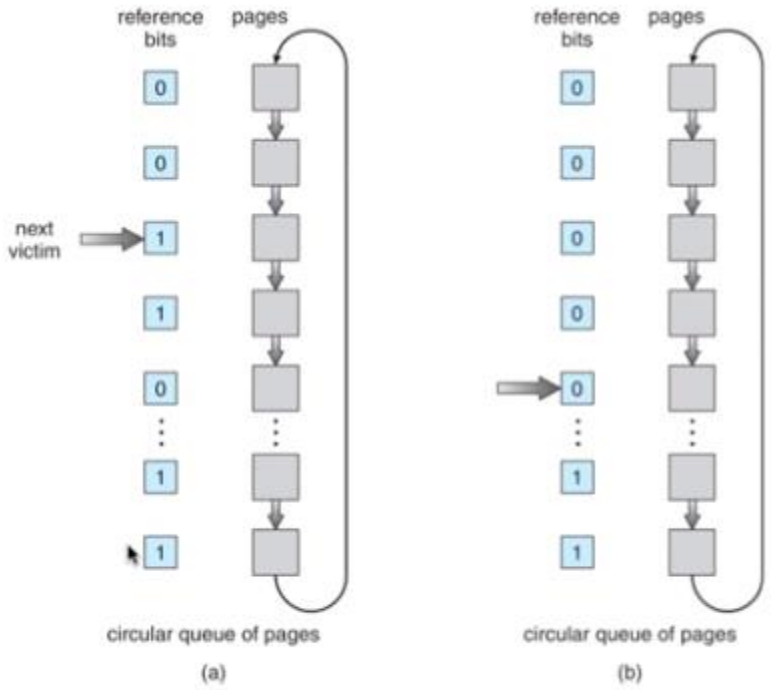


Figure 9.17 Second-chance (clock) page-replacement algorithm.

Second-Chance (Clock) Algorithm

- Advantages:
  - simple to implement (one pointer for the clock hand + 1 ref bit/page.
    - Note the circular buffer is actually just the page table with entries where the valid bit is set, so no new circular queue has to be constructed),
  - fast to check reference bit
  - usually fast to find first page with a 0 reference bit
  - approximates LRU
- Disadvantages:
  - in the worst case, have to rotate through the entire circular buffer once before finding the first victim frame

Counting-Based Page Replacement

- Keep a counter of number of page accesses for each page since its introduction, which is an activity or popularity index
- **Most Frequently Used**
  - Replace page with highest count
  - Assumes the smallest counts are most recently loaded
- **Least Frequently Used**
  - Replace page with lowest count
  - What if a page was heavily used in the beginning, but not recently?
    - Age the count by shifting its value right by 1 bit periodically - this is exponential decay of the count.
- **These kinds of policies are not commonly used**

Approximation of OPT Algorithm

- We use the past to approximate the future

- We can be smarter about looking at the past
  - Not just a sequence, but a pattern of use
    - Use this to create a better approximation of page use
  - Take advantage of the locality
  - Create a *working set* of pages

Belady's Anomaly

FIFO Page Replacement: Another example

Let page reference stream, $\mathcal{R}$ = `012301401234`

Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	0	0	0	3	3	3	4	4	4	4	4	4
1		1	1	1	0	0	0	0	0	2	2	2
2			2	2	2	1	1	1	1	1	3	3

- FIFO with m = 3 has 9 faults
- Goal: To reduce the number of page fault
  - Increase the size of memory

FIFO Page Replacement: Another example

Let page reference stream, $\mathcal{R}$ = `012301401234`

Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	0	0	0	0	0	0	4	4	4	4	3	3
1		1	1	1	1	1	1	0	0	0	0	4
2			2	2	2	2	2	2	1	1	1	1
3				3	3	3	3	3	3	2	2	2

- FIFO with m = 4 has 10 faults

Belady's anomaly: Increasing the size of memory may result in increasing the number of page faults for some programs.

Stack Algorithms

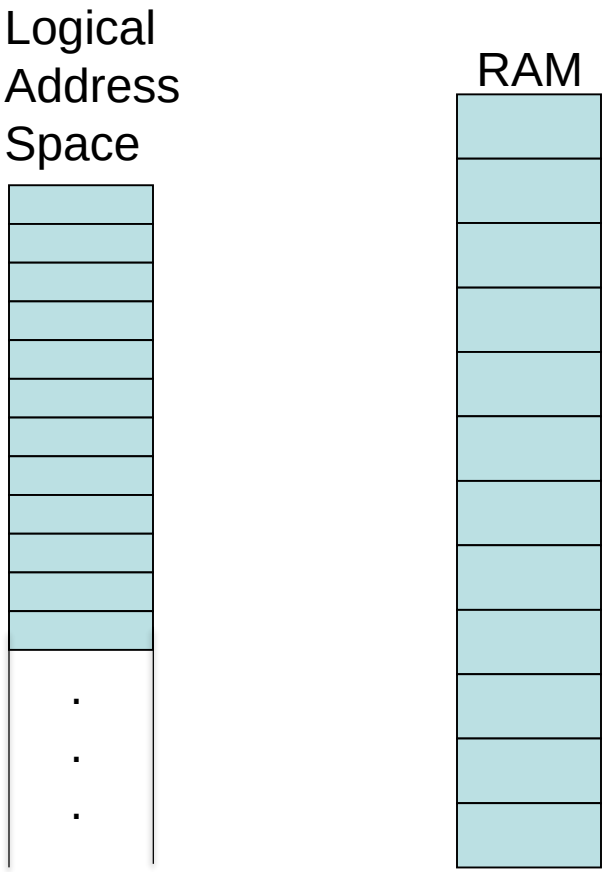
- Stack algorithms are a class of page replacement algorithms that do not suffer from Belady's anomaly

- **Key property**:
  - Set of pages in memory for *n* frames is always a subset of the set of pages that would be in memory with *n+1* frames, irrespective of the page reference string

- **OPT and LRU are stack algorithms, while FIFO is not a stack algorithm**
  - The OPT and LRU algorithms will always keep a subset of the pages used in a larger number of frames
  - FIFO may end up with a different set of pages depending on the number of frames

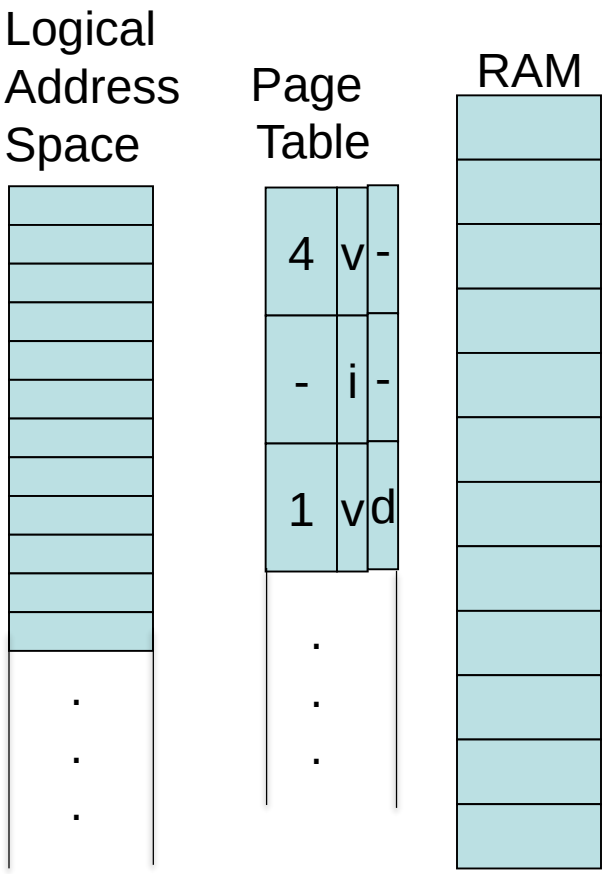
Frame Allocation

Techniques to Improve Page Replacement Performance



Techniques to Improve Page Replacement Performance

1. Use a **dirty/modify bit** to reduce disk writes
2. Choose a **smart page replacement algorithm**
   - keeps the most important pages in memory and evicts the least important
3. Make the **search for the least important page be fast**

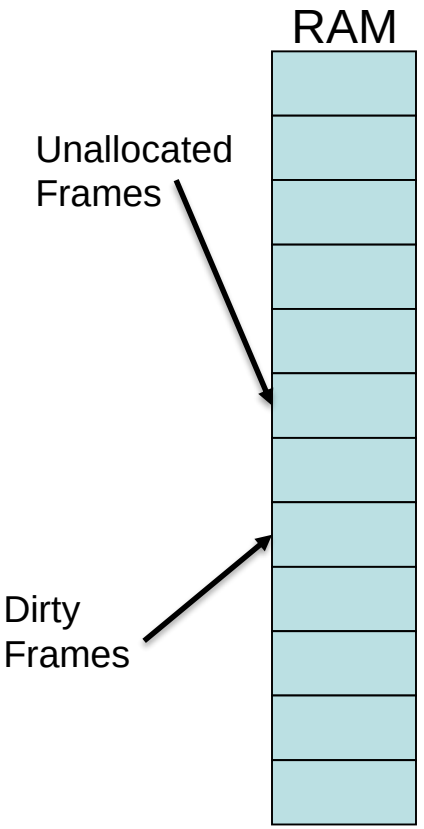


Techniques to Improve Page Replacement Performance

4. **Page-buffering**
   - read in a frame first and start executing, then select a victim and write it out at a later time - faster perceived performance
   - periodically write out all modified frames to disk when no other activity. Thus most frames are clean so few disk writes on a page fault.

5. **Keep a pool of free frames** and remember their content.
   - Reuse this free frame if the same contents are needed again. Reduces disk reads.

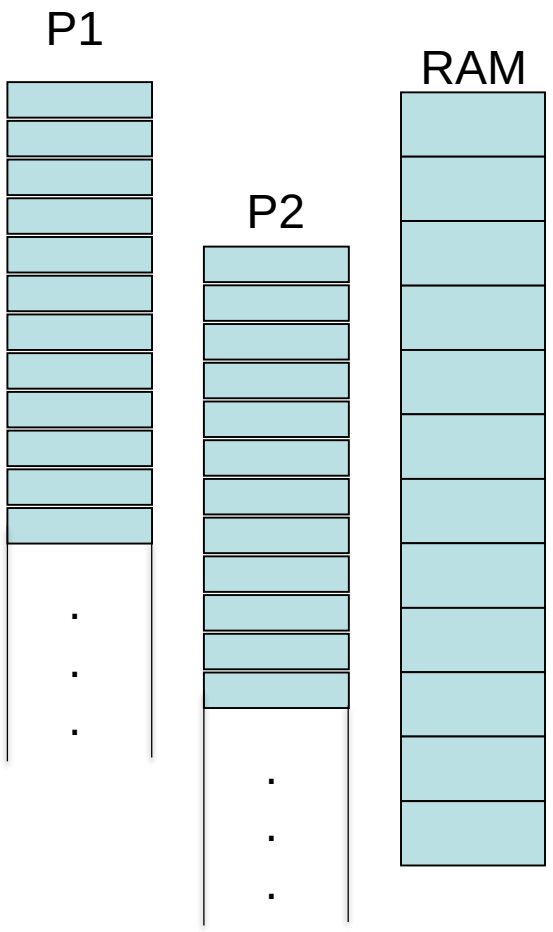
6. Allocate the **appropriate # of frames** so a process avoids thrashing – we'll see this next



Allocation of Memory Frames

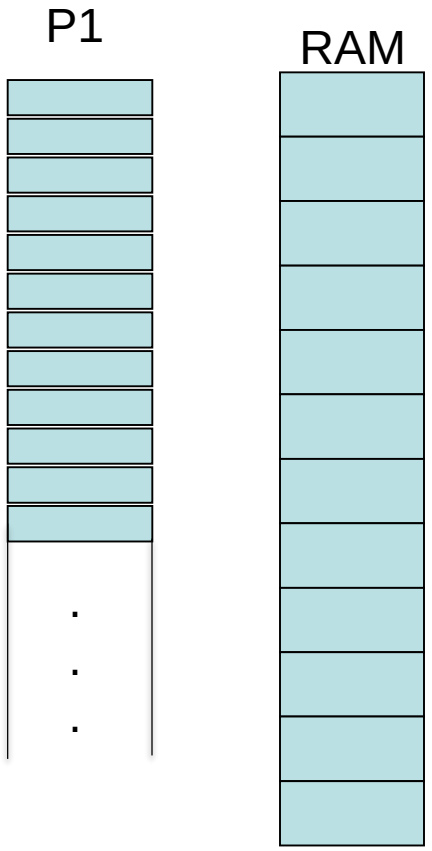
- Given that the OS employs paging for memory management, then physical memory is divided into fixed-sized frames or pages.

- How many frames does each process get allocated? How many frames does the OS get allocated versus user processes?
  - Variety of policies:
    - based on number of frames
    - based on whether frames are allocated locally or globally
  - Example: given the OS, and processes P1 and P2, and 15 frames of physical memory, how do we allocate the frames among these three entities?



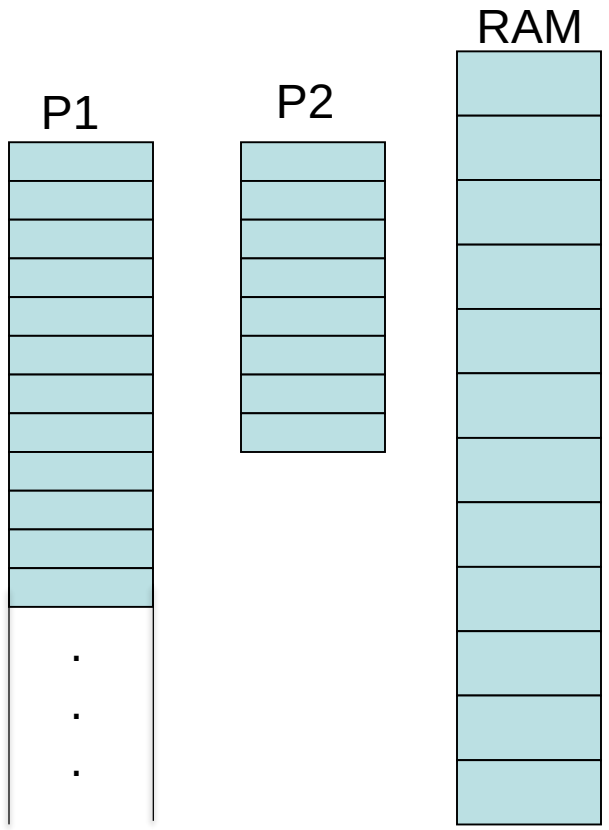
Memory Allocation Policies

1. **Minimum # frames:**
   - **determine the minimum # of frames that allow a process to allocate**. Ideally, this is just one page, i.e. the page in which the program counter is currently executing.
     Or one for code, one for data
   - In practice, some CPU's support complex instructions.
     - multi-address instructions. Each address could belong to a different page.
   - Also, there can be multiple levels of indirection in the addressing, i.e. pointers.
     - Each such level of indirection could result in a different page being accessed in order to execute the current instruction. Up to N levels of indirection may be supported, which means may need up to N pages as the minimum.



Memory Allocation Policies

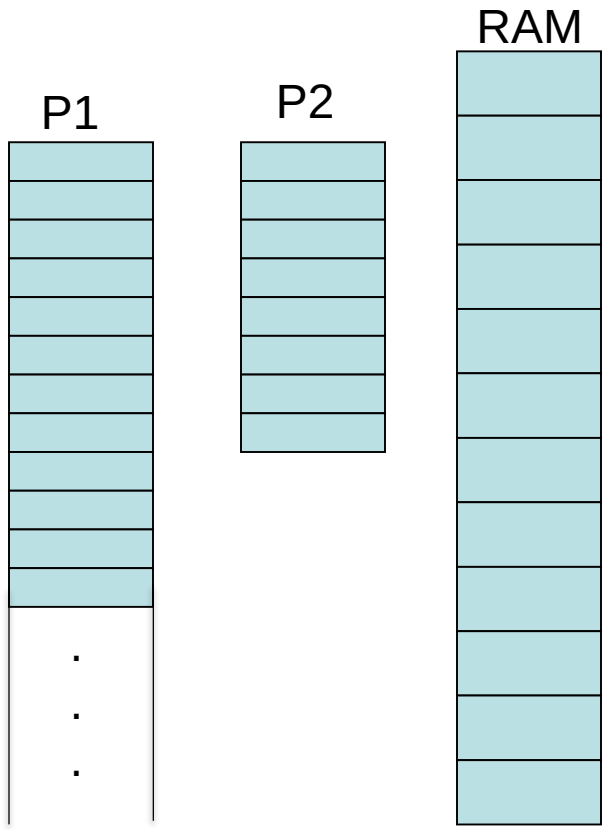
2. **Equal allocation:**
   - split m frames equally among n process
   - m/n frames per process
   - problem: doesn't account for size of processes, e.g. a large database process versus a small client process whose size is << m/n
   - needs to be adaptive as new process enter and the value of n fluctuates



Memory Allocation Policies

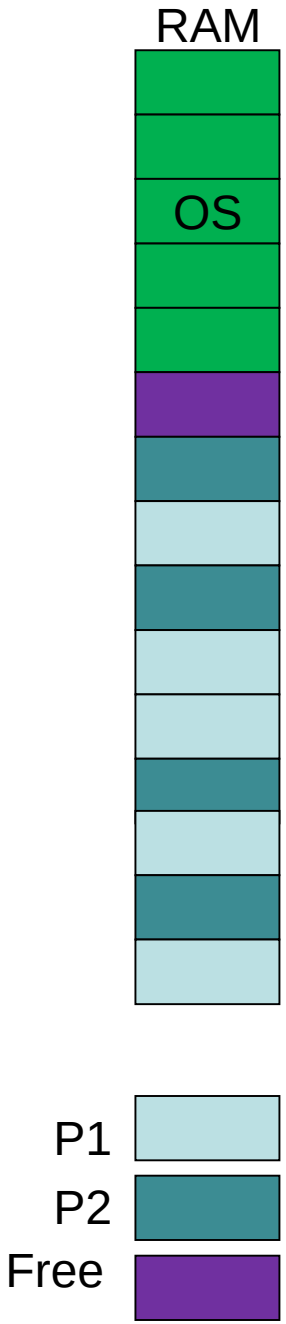
3. Proportional allocation

- allocate the number of frames relative to the size of each process
- Let S_i = size of process P_i
- $S = \Sigma S_i$
- Allocate $a_i = (S_i / S) * m$ frames to process P_i
- proportion a_j can vary as new processes start and existing processes finish
- Also, if size is based on the code size, or address space size, then that is not necessarily the number of pages that will be used by a process



Local vs Global Allocation/Replacement

- In local allocation/page replacement, a process is assigned a fixed set of N memory pages for the lifetime of the process
  - When a page needs to be replaced, it is chosen only from this set of N pages
  - Easy to manage
  - Processes isolated from each other (not fully true)
  - Windows NT follows this model

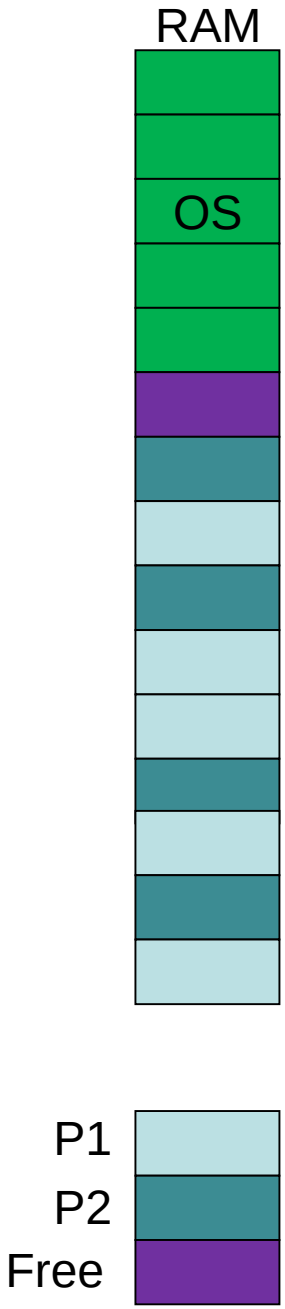


Local vs Global Allocation/Replacement

- Problems with local allocation:
  1. the behavior of processes may change as they execute
     - Sometimes they'll need more memory, sometimes less
     - local replacement doesn't allow a process to take advantage of unused pages in another process
     - Want a more adaptive allocation strategy that would allow a page fault to trigger the page replacement algorithm to increase its page allocation

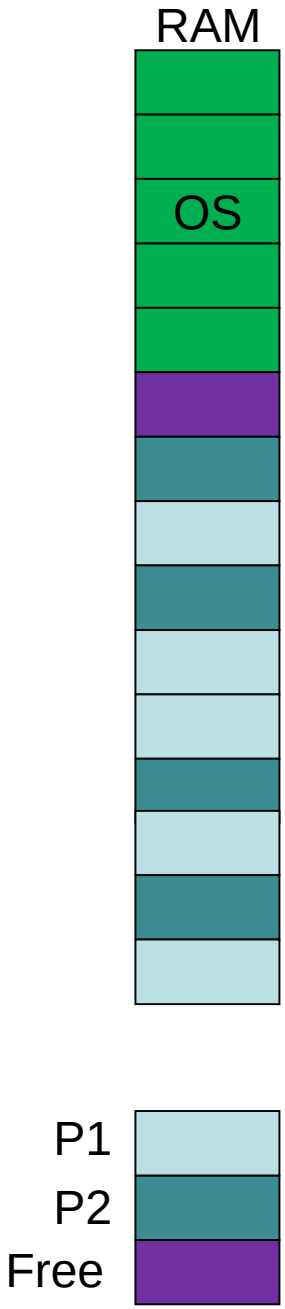
Local vs Global Allocation/Replacement

- Problems with local allocation:
  2. Local replacement would seem to isolate a process's paging behavior from other processes
     - Amount of memory allocated to each process is fixed
     - However, isolation is not perfect:
     - If a process P1 page faults frequently, then P1 will queue up many requests to read/write from/to disk.
     - If another process P2 needs to access the disk, e.g. to page fault, then even if P2 page faults less frequently, P2 will still be slowed down by P1's many queued up reads and writes to disk



Local vs Global Allocation/Replacement

- Global allocation and page replacement
  - Pool all pages from all processes together
  - When a page needs to be replaced/evicted, choose it from the global pool of pages
  - Linux follows this model
  - Global allocation and page replacement allows the # pages allocated to a process to fluctuate dramatically
    - Good: system adapts to let a process utilize “unused” pages of another process, leading to better memory utilization overall and better system throughput
    - Bad: a process cannot control its own page fault rate under global replacement, because other processes will take its allocated frames, potentially increasing its own page fault rate




MOBILE & NETWORKED SYSTEMS LAB
MNS
UNIVERSITY OF COLORADO - BOULDER

Thrashing

Memory Management Thrashing

FIFO Page Replacement Example

Let page reference stream, $\mathcal{R}$ = 012301401234

Frame	0	1	2	3	0	1	2	3	0	1	2	3
0	0	0	0	3	3	3	2	2	2	1	1	1
1		1	1	1	0	0	0	3	3	3	2	2
2			2	2	2	1	1	1	0	0	0	3

- FIFO with m = 3 has 12 faults
- Every request is causing a page fault
- This is an extreme example of page thrashing

Thrashing

- **Describes situation of repeated page faulting**
  - this significantly slows down performance of the entire system, and is to be avoided
- **occurs when a process' allocated # of frames < size of its recently accessed set of frames**
  - each page access causes a page fault
  - must replace a page, load a new page from disk
  - but then the next page access also is not in memory, causing another page fault, resulting in a domino effect of page faults - this is called *thrashing*
  - a process spends more time page faulting to disk than executing – is I/O bound, causing a severe performance penalty

Thrashing

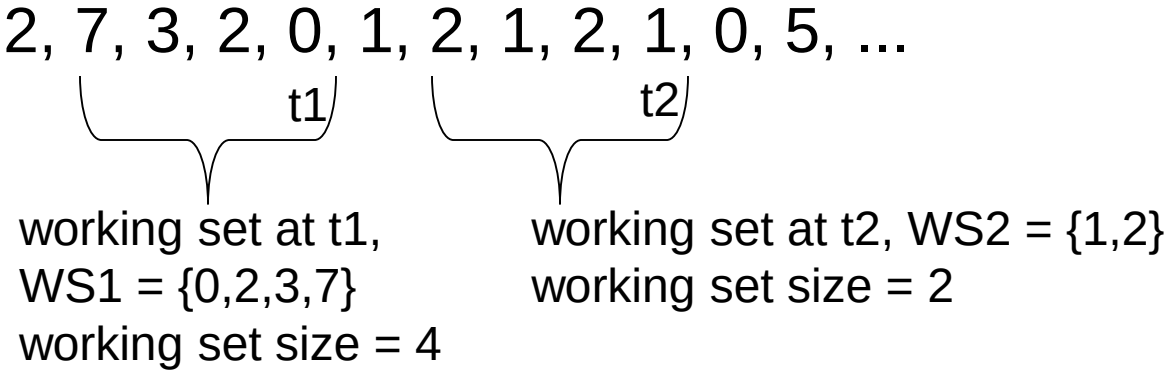
- Thrashing under global replacement
  - a process needs more frames, page faults and takes frames away from other processes, which then take frames from others, ... - domino effect
  - as processes queue up for the disk, CPU utilization drops
  - The process isolation is not perfect. A process's disk reads and writes will slow down other processes that also have disk I/Os
  - OS can add fuel to the fire:
    - suppose OS has the policy that if it notices CPU utilization dropping, then it thinks that the CPU is free, so it restarts some processes that have been frozen in swap space
    - these new processes page fault more, causing CPU utilization to drop even further

Thrashing

- **Solution 1: Build a *working set* model**.
  - Programs tend to exhibit *locality* of behavior, i.e. they tend to access/reuse same code/data pages
  - Find a set of recently accessed pages that captures this locality
  - To measure locality, define a window size Δ of the Δ most recent page references
    - Within the set of pages in Δ, the working set is the set of *unique* pages
    - pages recently referenced in working set will likely be referenced again
  - Then allocate to a process the size of the working set

Thrashing

- working set solution (continued):
  - Example: Assume $\Delta = 4$ and we have a page reference sequence of



    - at time t1, process only needs to be allocated 4 frames; at time t2, process only needs 2 frames
    - Is $\Delta = 4$ the right size window to capture the locality?

Thrashing

- working set solution (cont.)
  - Choose window Δ carefully
    - if Δ is too small, Δ won't capture the locality of a process
    - if Δ is too large, then Δ will capture too many frames that aren't really relevant to the local behavior of the process
      - which will result in too many frames being allocated to the process

Thrashing

- working set solution (cont.)
  - Once Δ is selected, here is the working set solution:
    1. Periodically compute a working set and working set size WSS_i for each process P_i
    2. Allocate at least WSS_i frames for each process
    3. Let demand $D = \Sigma\ WSS_i$. If D > m total # of free frames, then there will be thrashing. So swap out a process, and reallocate its frames to other processes.
  - This working set strategy limits thrashing

Thrashing

- **Approximate working set with a timer & a reference bit**
  - set a timer to expire periodically
  - any time a page is accessed, set its reference bit
  - at each expiration of the timer, shift the reference bit into the most significant bit of a record kept with each page,
    - a byte records references in the last 8 timer epochs
  - If any reference bit is set during the last N timer intervals, then the page is considered part of the working set, i.e. if record > 0
  - re-allocate frames based on the newly calculated working sets. Thus, it is not necessary to recalculate the working set on every page reference.

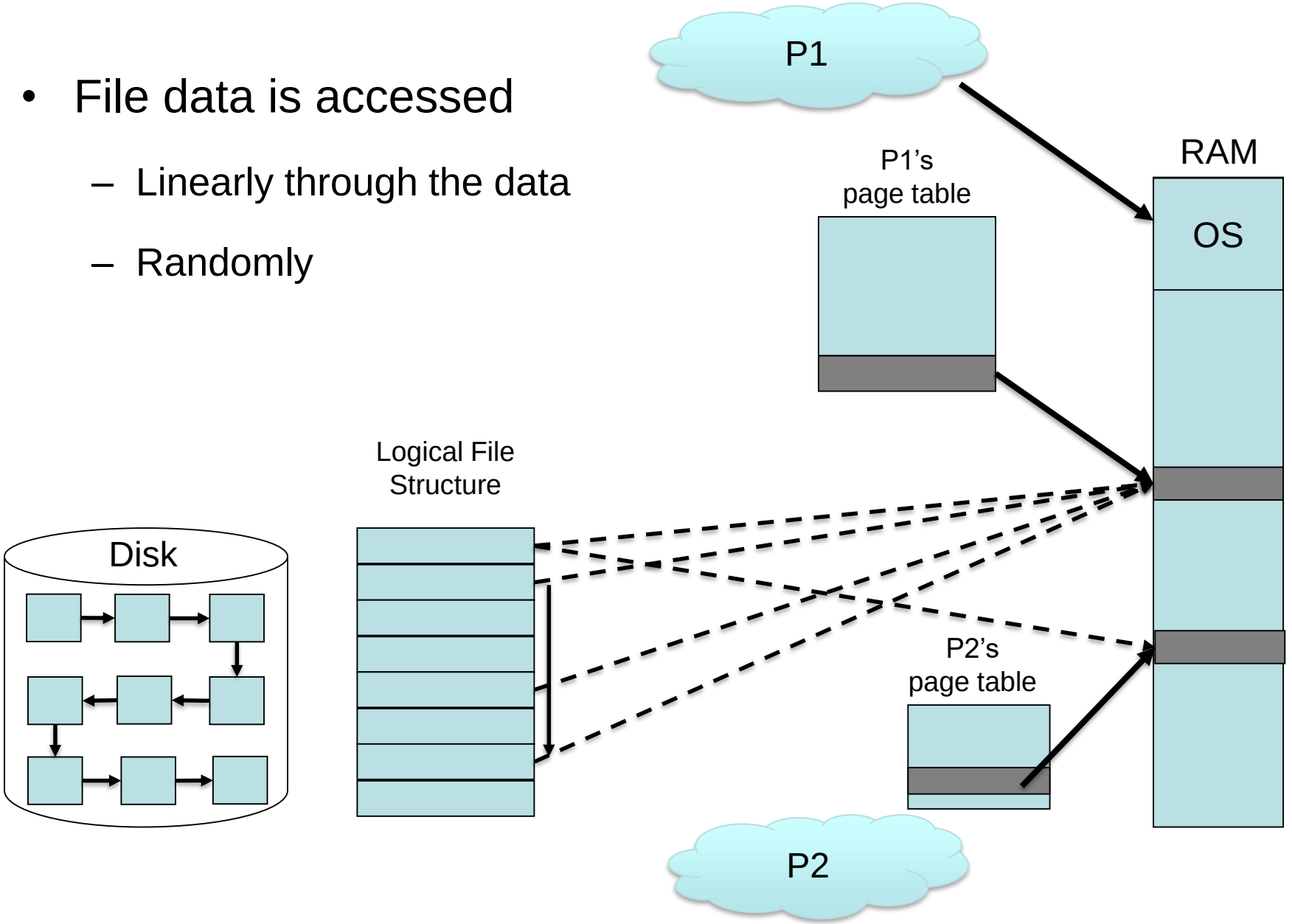
Thrashing

- **Solution 2: Instead of using a working set model, just directly measure the page fault frequency (PFF)**
  - When PFF > upper threshold, then increase the # frames allocated to this process
  - When PFF < lower threshold, then decrease the # frames allocated to this process (it doesn't need so many frames)
  - Windows NT used a version of this approach

Memory Mapped Files

Memory-Mapped Files

- File data is accessed
  - Linearly through the data
  - Randomly



Memory-Mapped Files

- Map some parts of a file on disk to pages of virtual memory
  - normally, each read/write from/to a file requires a system call plus file manager involvement plus reading/writing from/to disk
    - programmer can improve performance by copying part of or entire file into a local buffer, manipulating it, then writing it back to disk
      - this requires manual action on the part of the programmer
  - instead, it would be faster and simpler if the file could be loaded into memory (almost) transparently so that reads/writes take place from/to RAM
  - Use the virtual memory mechanism to map (parts of) files on disk to pages in the logical address space

Steps for memory-mapping a file

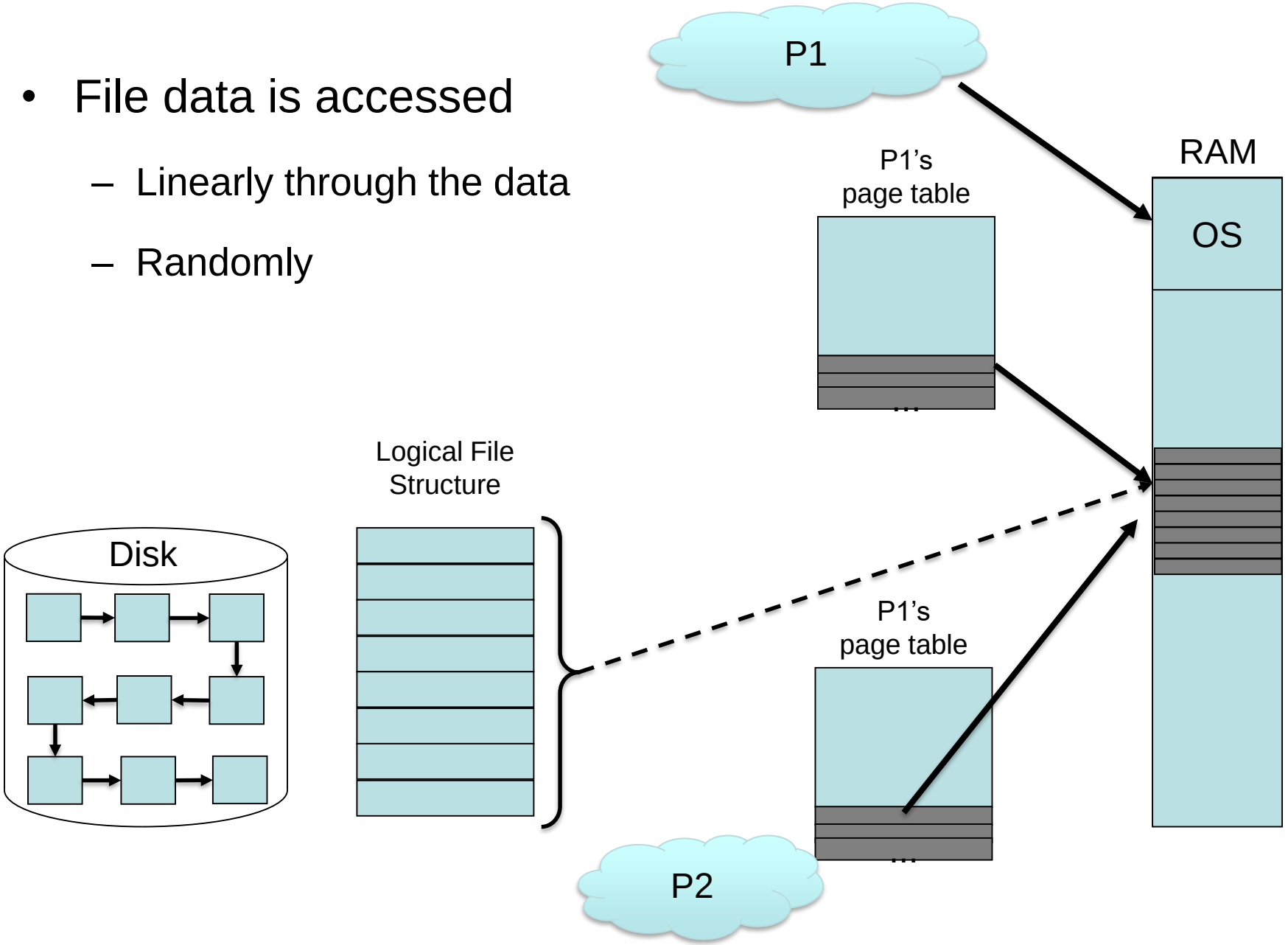
Steps:

1. Obtain a handle to a file by creating or opening it
2. Reserve virtual address space for the file in your logical address space
3. Declare a (portion of a) file to be memory mapped by establishing a mapping between the file and your virtual address space

- Use an OS function like *mmap*()
- *void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)*
  - map length bytes beginning at offset into file fd, preferably at address start (hint only), prot = R/W/X/no access, flags = maps_fixed, map_shared, map_private
  - returns pointer to mmap'ed area

Memory-Mapped Files

- File data is accessed
  - Linearly through the data
  - Randomly



Memory-Mapped Files

Steps for memory-mapping a file: (cont.)

4. When file is first accessed, it’s demand paged into physical memory
5. Subsequent read/write accesses to (that portion of) the file are served by physical memory

Memory-Mapped Files

- Advantages of memory-mapping files:
  1. after the first accesses, all subsequent reads/writes from/to a file (in memory) are fast
     - No longer use file system to read/write. Instead use MMU
  2. multiple processes can map the same file concurrently and
     - share efficiently, as shown in the previous figure
     - In Windows, this mapping mechanism is also used to create shared memory between processes and is the preferred memory for sharing information among address spaces
     - on Linux, have separate mmap() and shared memory calls, e.g. shmget() and shmat()

Memory-Mapped Files

3. Remember that the entire file **need not** be mapped into memory, just a portion or “view” of that file
   - in previous figure, F could represent just a portion of some larger file
   - F can also be subdivided into pages, each of which is mapped to a separate page in memory by the page table
4. File system has to be consulted less often
   - Step 4 to determine where on disk a view of a file is located
   - File Open is an expensive operation

Memory-Mapped Files

1) Problem 1 - Writes to a file in memory can result in momentary inconsistency between the file cached in memory and the file on disk
   - could write changes immediately (synchronously) to disk
   - Or delay and cache writes (asynchronous policy)
     - wait until OS periodically checks if dirty/modify bit has been set
     - wait until activity is less, then write altogether so that individual writes don't delay execution of process
     - this also allows the OS to group writes to the same part of disk to optimize performance. (we'll see this later)

MM I/O vs. MMFiles)

- Similar behavior to memory-mapped files
- Recall from pre-midterm lecture slides that memory-mapped I/O maps device registers (instead of file pages) to memory locations
  - reads/writes from/to these memory addresses are easy and are automatically caught by the MMU (just as for mem-mapped files), causing the data to be automatically sent from/to the I/O devices
  - e.g. writing to a display's memory-mapped frame buffer, or reading from a memory-mapped serial port
  - Devices use an API to a virtual file

Summary

- Virtual memory
  - Making logical space independent of physical memory space
  - Advantages vs. disadvantages
- Page replacement policies
- Belady’s Anomality
- Frame allocation
- Thrashing
- Memory Mapped Files