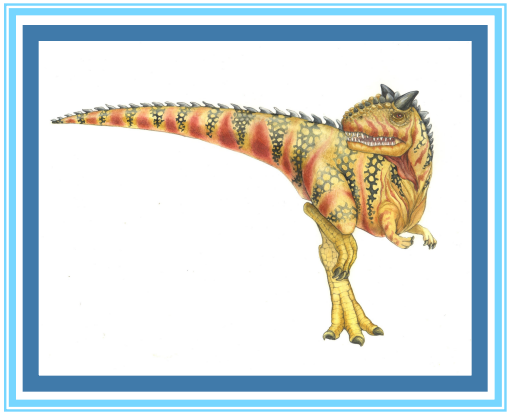
Chapter 10: Virtual Memory



Chapter 10: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Objectives

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process, and explain how it is related to program locality.
- Describe how Linux, Windows 10, and Solaris manage virtual memory.
- Design a virtual memory manager simulation in the C programming language.

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

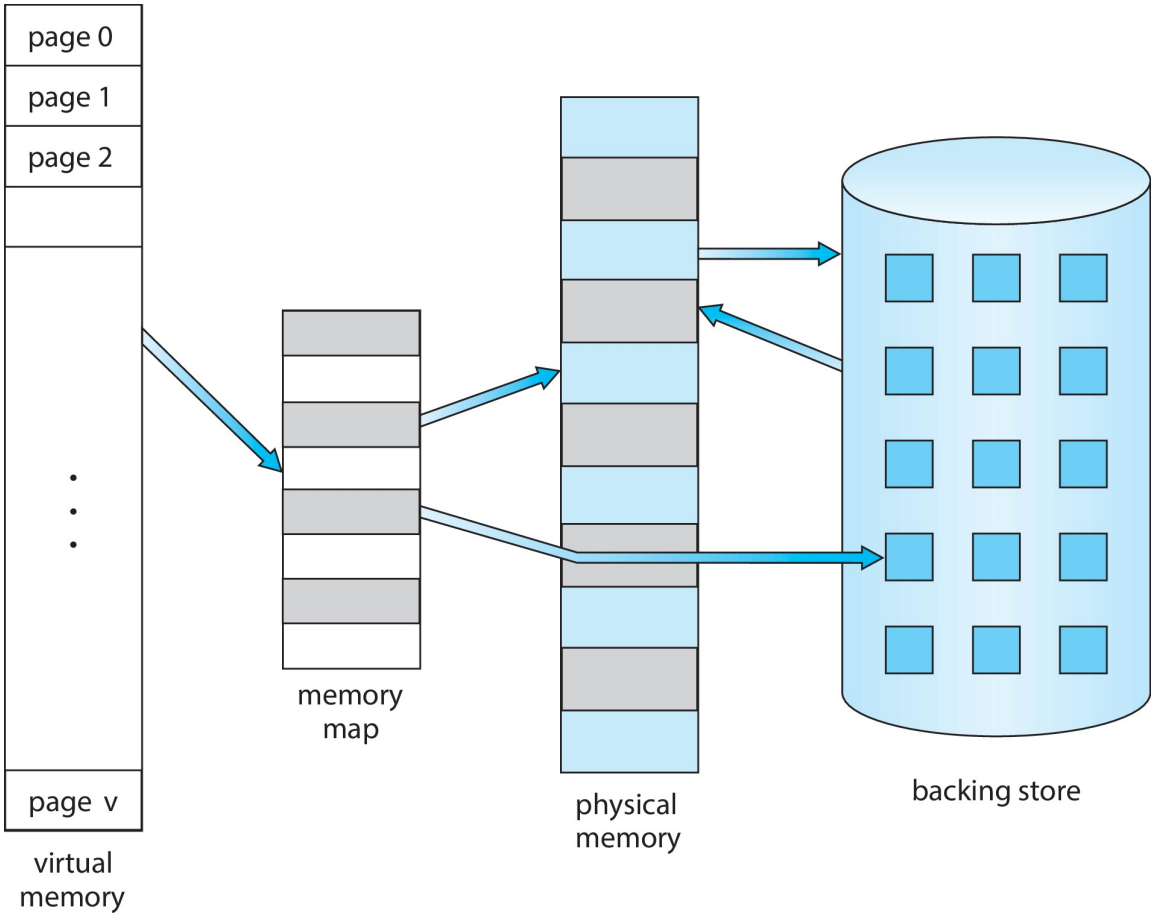
Virtual memory

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

Virtual memory (Cont.)

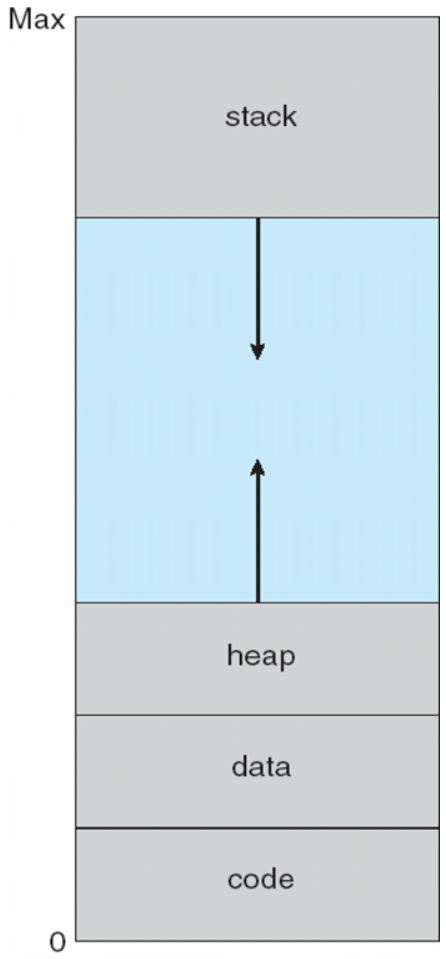
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

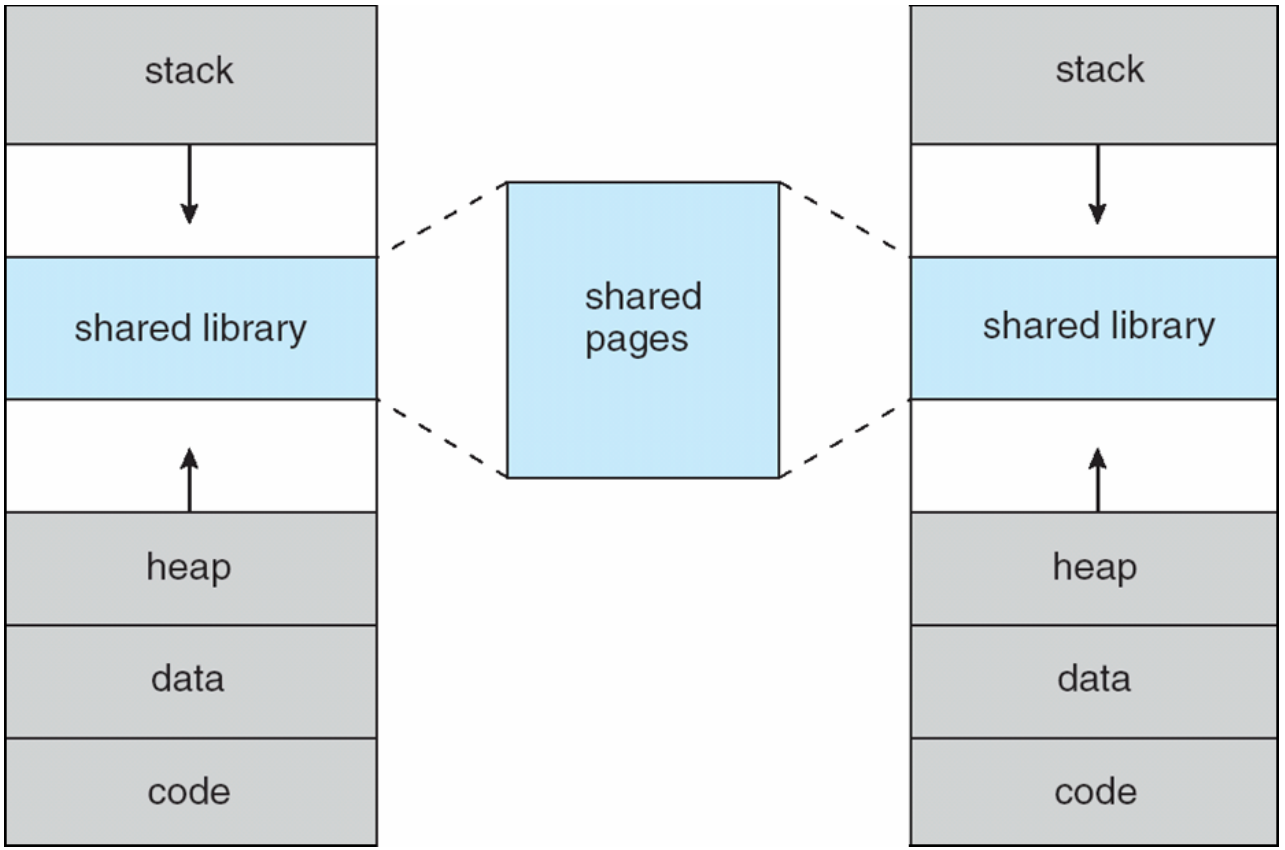


Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



Shared Library Using Virtual Memory

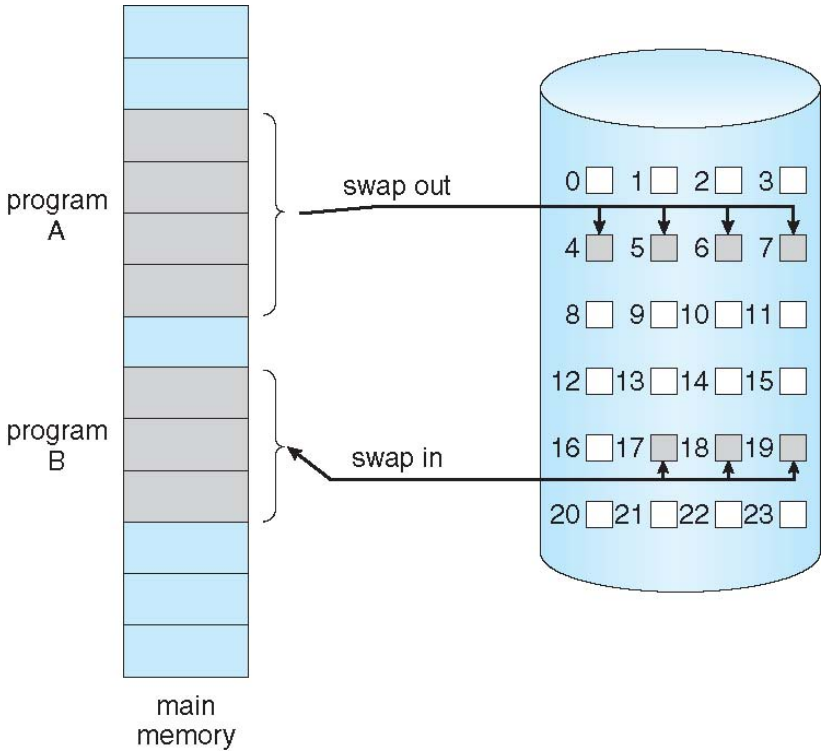


Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)

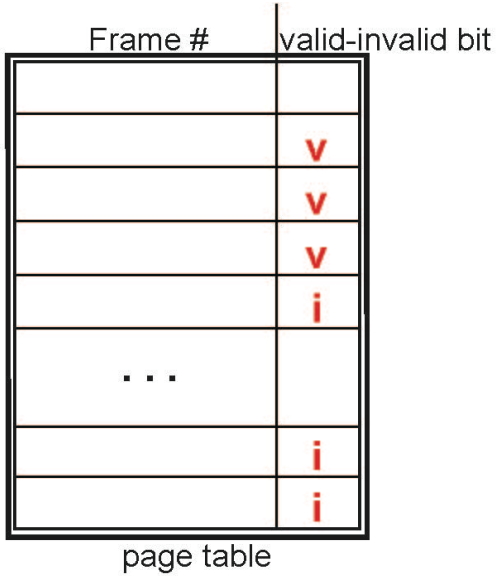


Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

Valid-Invalid Bit

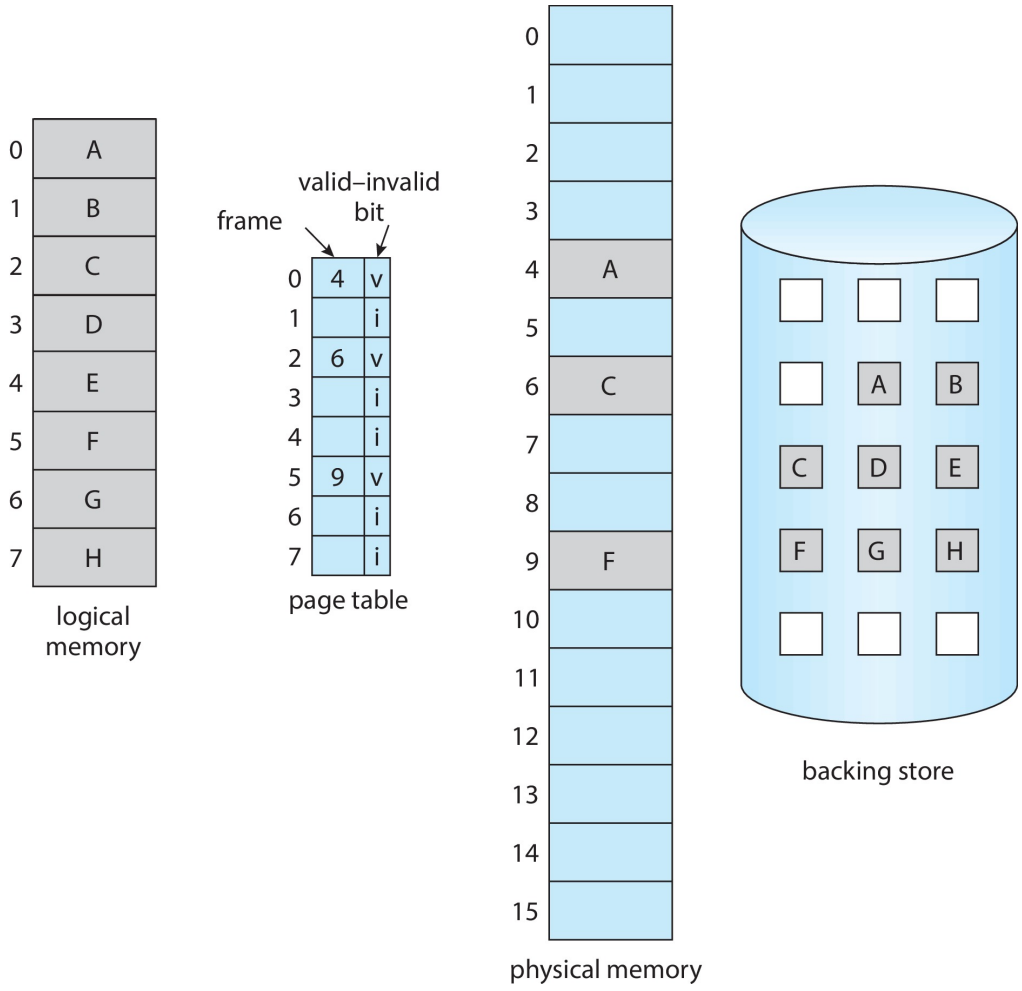
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory

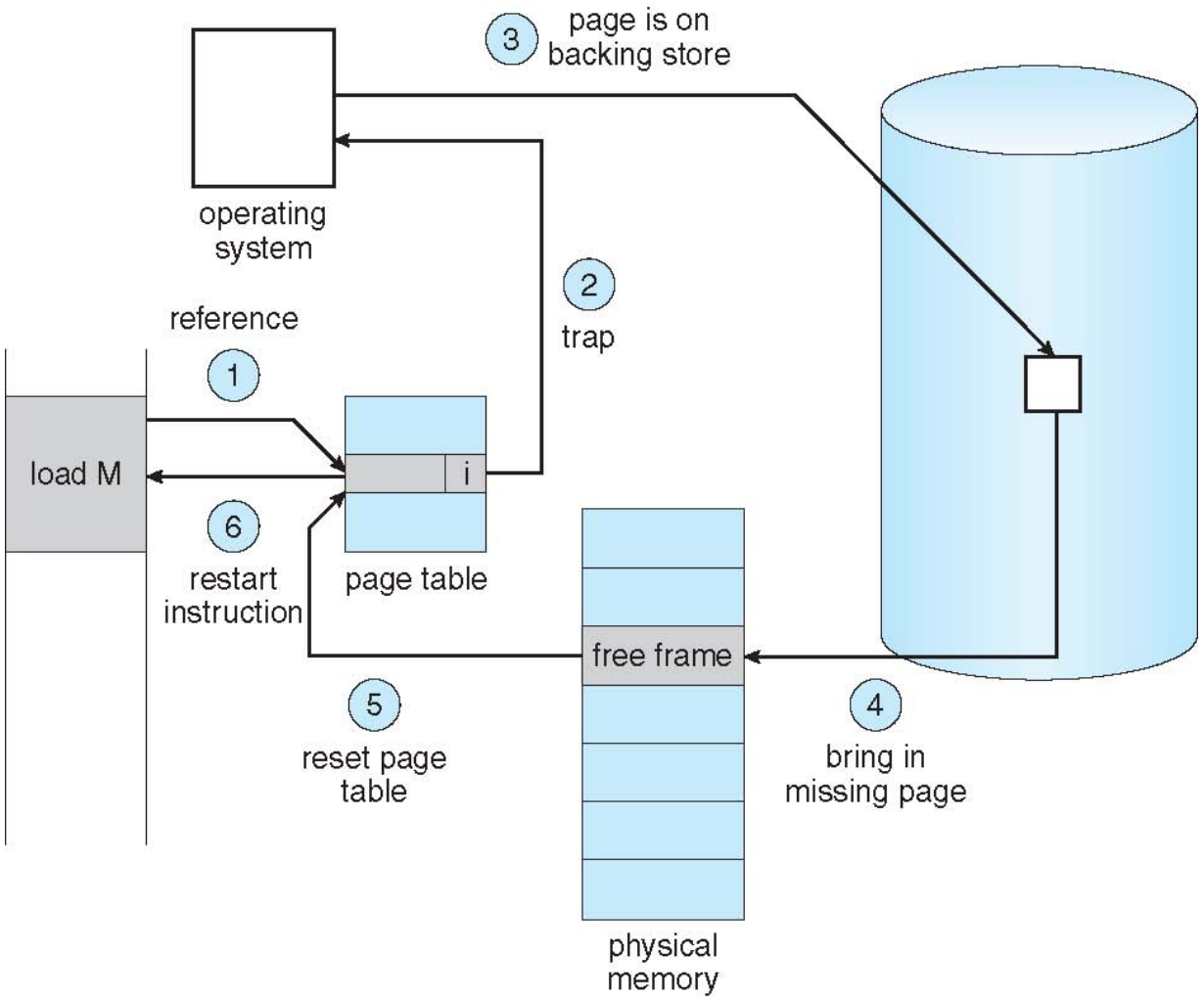
When the process modifies A the copy on disk is not the same as the copy on memory.



Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault (Cont.)

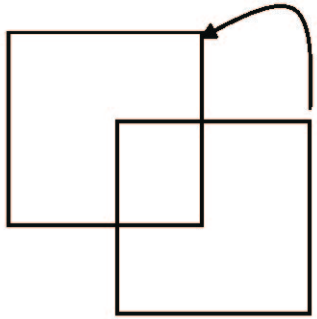


Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Instruction Restart

- Consider an instruction that could access several different locations
  - Block move



  - Auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?

Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.

- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

Stages in Demand Paging – Worse Case

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame

Stages in Demand Paging (Cont.)

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – p) x memory access
  + p (page fault overhead
  + swap page out
  + swap page in)

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

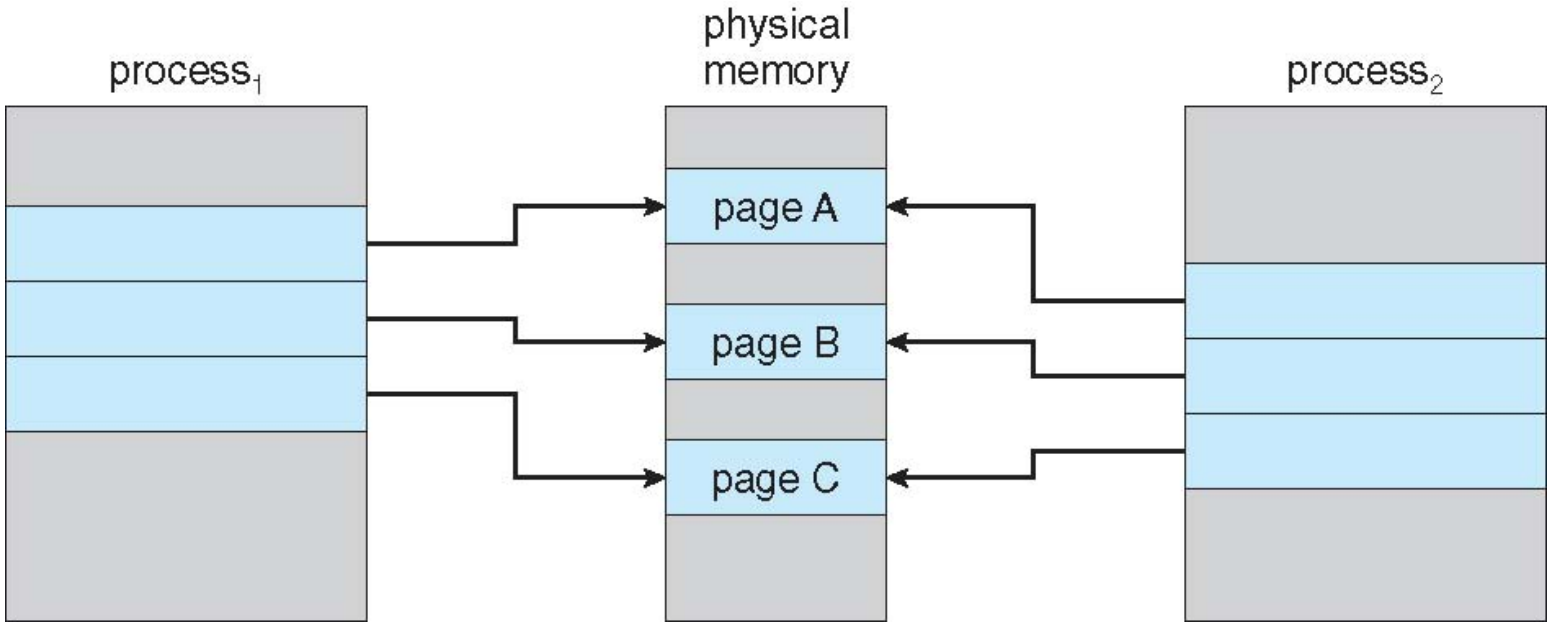
Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

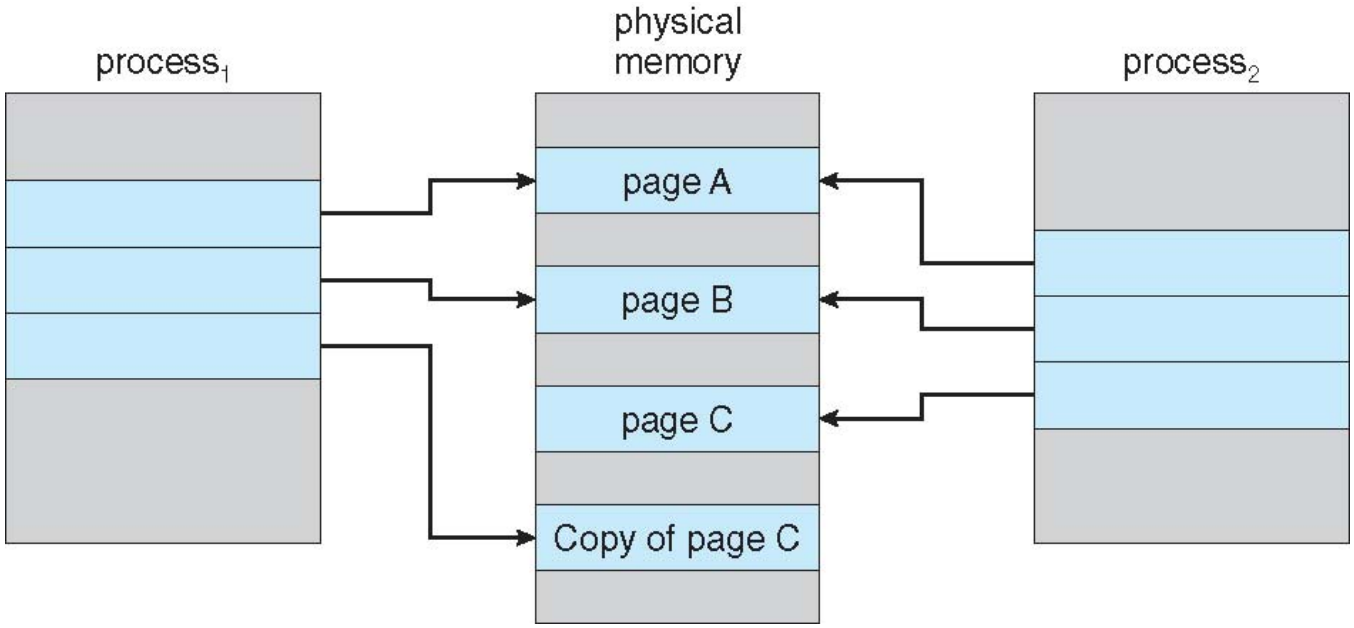
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Here we coming to the point of saying if there is no free frames what does it mean? free fram overall or ina process?
Lets just consider a given process has a certain number of valuable frames and all of them are used. How many do we allocate to each process?

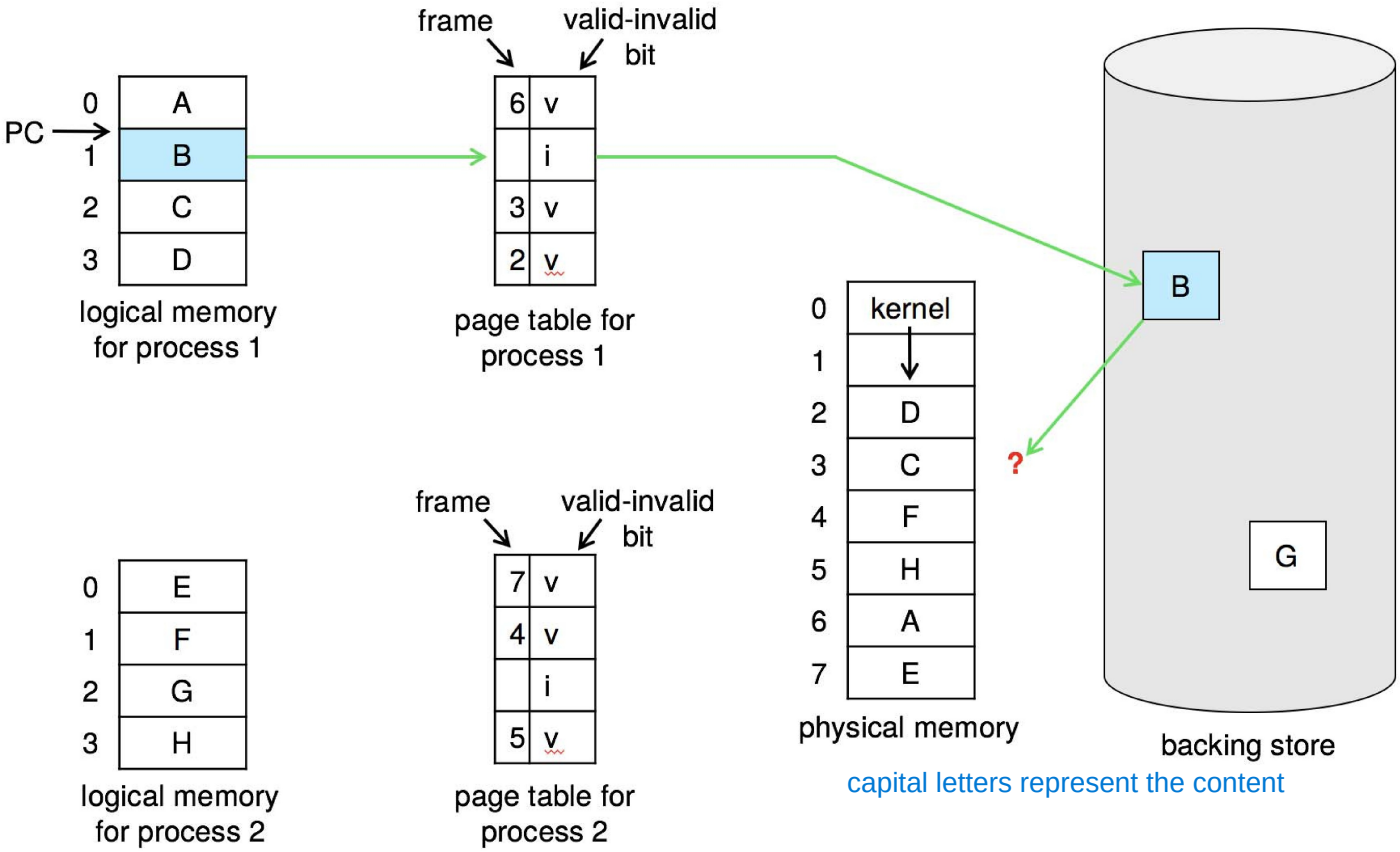
Page Replacement

an idea we need to consider to provide a more flexible strategy.

The last step to provide virtual memory. Fully dynamic management of frames.

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement



capital letters represent the content

Basic Page Replacement

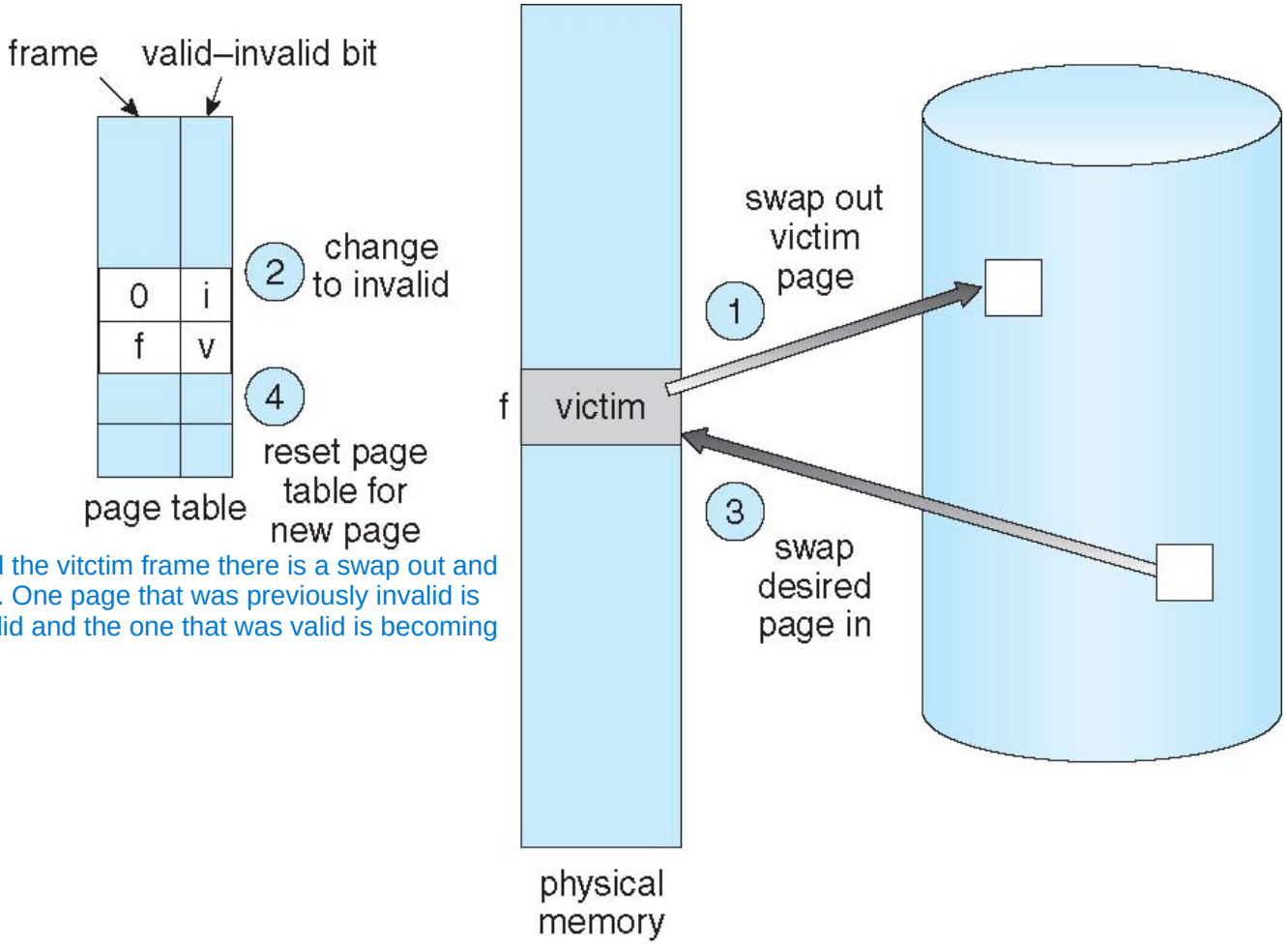
1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

once we have loaced the page on disk lets say B

We need to find a page or cadidate for replacement if we dont find it.

Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement



once you find the vitctim frame there is a swap out and swap in here. One page that was previously invalid is becoming valid and the one that was valid is becoming invalid.

Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

Replace one page that is not necessary for you in furture. It is a predicative algo. How do we know the algo is good? Is future = past? depends? Evaluation should be experimental. training test and test set should be similar.

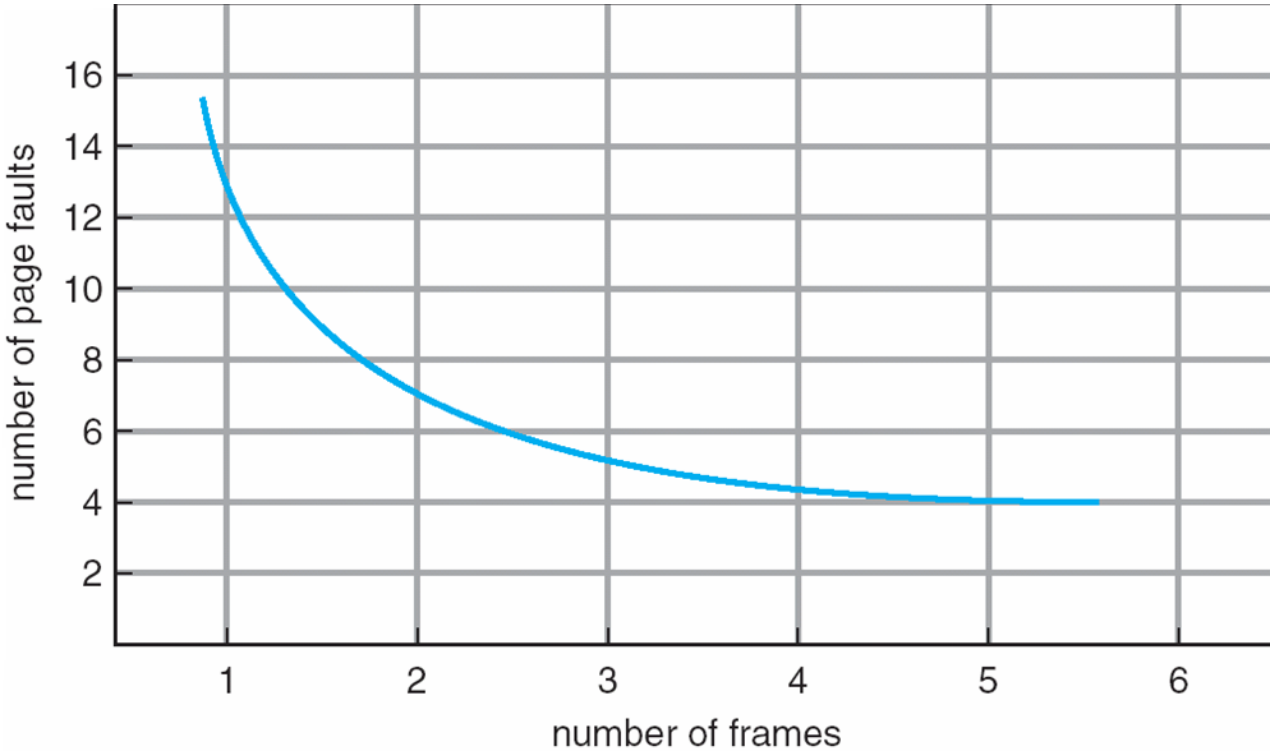
The only meaningful data that we need for algo are the ref string. A program is going to make several read and writes to mem, reads can be instructions like fetch. Could be a list of mem locations that have been accessed for reading and writing.That could be a ref string. Reference string that has already filtered out page numbers

what we can see certain pages are accessed several times like page 7 or 1 and few are used very less.

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

page 7, 0, 1.....

Graph of Page Faults Versus The Number of Frames



how many frames is that program going to use?

The more frames we have the less page faults we will trigger.
If you have only one frame it means that every time the program is changing the page there will be a page fault.

Page replacement strategies

Page Fault Frequency (empirical probability)

$$f(A,m) = \sum_{\forall w} p(w) \frac{F(A,m,w)}{len(w)}$$

- *A* page replacement algorithm under evaluation
- *w* a given reference string
- *p*(*w*) probability of reference string *w*
- *len*(*w*) length of reference string *w*
- *m* number of available page frames
- *F*(*A*,*m*,*w*) number of page faults generated with the given reference string (*w*) using algorithm *A* on a system with *m* page frames.

benchmarking of a car for eg would mean longer travel, speed, then take an avg of everything. lets say we have multiple reference string W. Ref string like the one we saw in the above slides. If you count the num of page faults and call it F and devide by length of ref string you get the frequency of page fault. Num of page faults over total num of accesses.

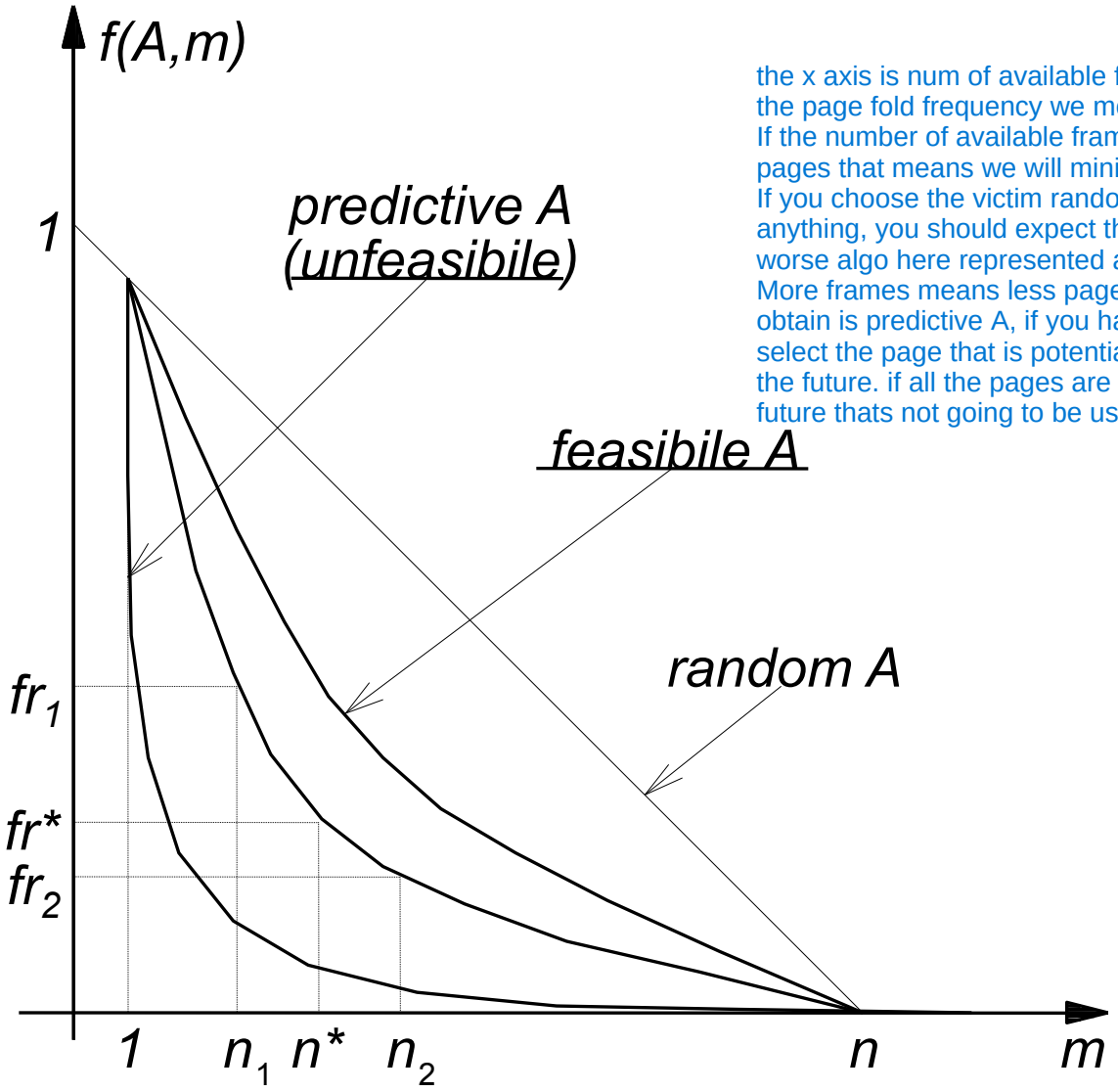
what are the variables that are influencing f?
If you have pages changing very rapidly you will prolly have more page faults. if you choose the right victim not to be used in future you will end up with less page faullts.

Once you have this fraction for ref string 1, ref string 2, ref string 3..... How to consider an average? weighted average? You are assuming all ref strings are meaningful.

Lets say 1st ref string 50% of y behavior, 2nd 25% and 3rd 10%. It is basically a weight of probabily. You compare it and choose the best one.

formula used in exam so remember.

page fault frequency as a function of m



the x axis is num of available frames, and on y the page fold frequency we measured.
If the number of available frame is equal to overall num of pages that means we will minimize.
If you choose the victim randomly without considering anything, you should expect that random should be the worse algo here represented as linear (random A).
More frames means less page folds. The best you can obtain is predictive A, if you have to select the victim you select the page that is potentially never going to be used in the future. if all the pages are being used we use the one in future thats not going to be used. it end up with best curve.

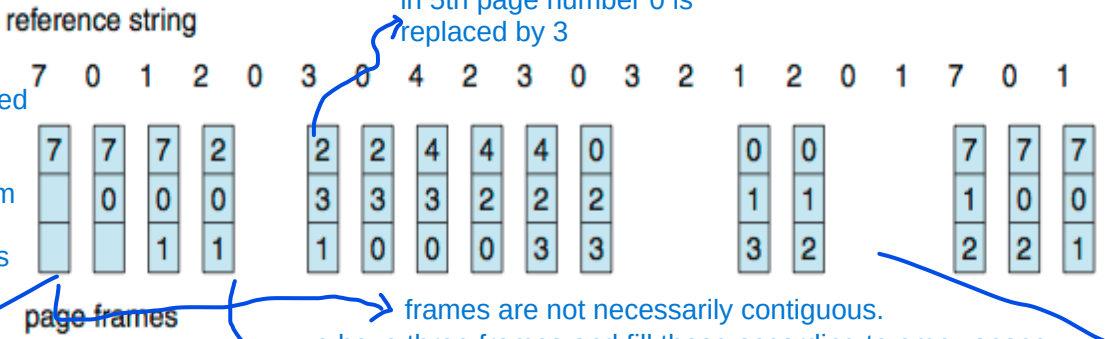
Select victim using modify bit

Reference bit	Modify bit
0	0
0	1
1	0
1	1

what kinda data are the algo allowed to observe to make predictions.
Reference bit 0 says no access to page since bit was cleared. modify bit 0 means most of the writes modify it is very rare that a write is rewriting the data. The modify bit means no modification .
1 1 means access and modification means okay. 0,1 means how can the page be modified and not accessed? Ref bit and modify bit are not necessarily cleared or sent at the same moment. it can mean modify bit was cleared long time ago and ref bit was cleared recently. The reference bit will be used by algorithms and reset frequently. simply because ref is cleared frequently we can have 0,1.

First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



in 5th page number 0 is replaced by 3

Typical queue. If you have a ref string, when i need a replacement the victim will be the page that among all pages in memory. the page that was entered in the frame earliest in the past.

the three frames represented close to each other but can be anywhere in the memory. we represent them close to each other only for representation purposes

frames are not necessarily contiguous.
we have three frames and fill these according to empy space.

in 4th(3 frames) one page num 7 is replaced by 2 as fifo.

15 page faults\

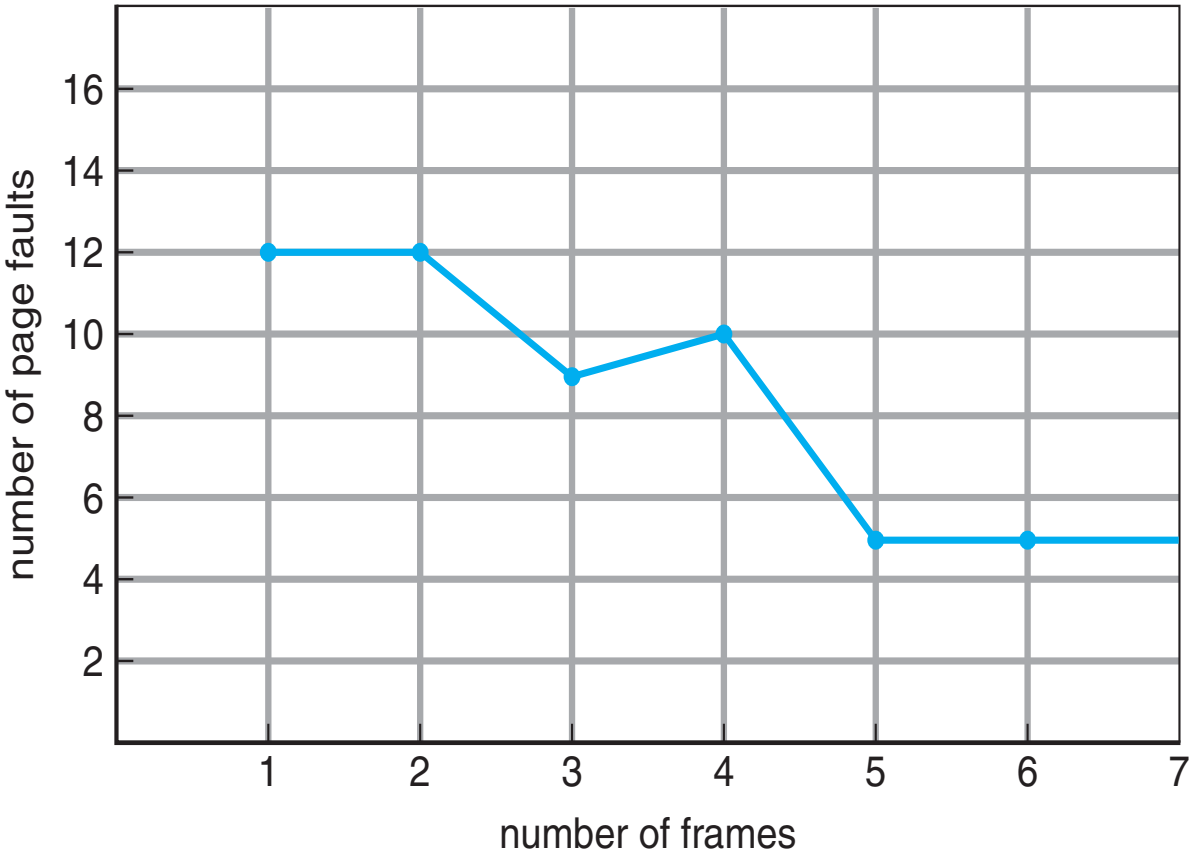
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

this means i have extracted 7 and inserted 2. so the head and tail hast moved so we simply need a pointer to know where the head and tail is.

we start with 0 faults. any time you see a rectangle with three frames means there is one page fault. no rectanges mean no page fault

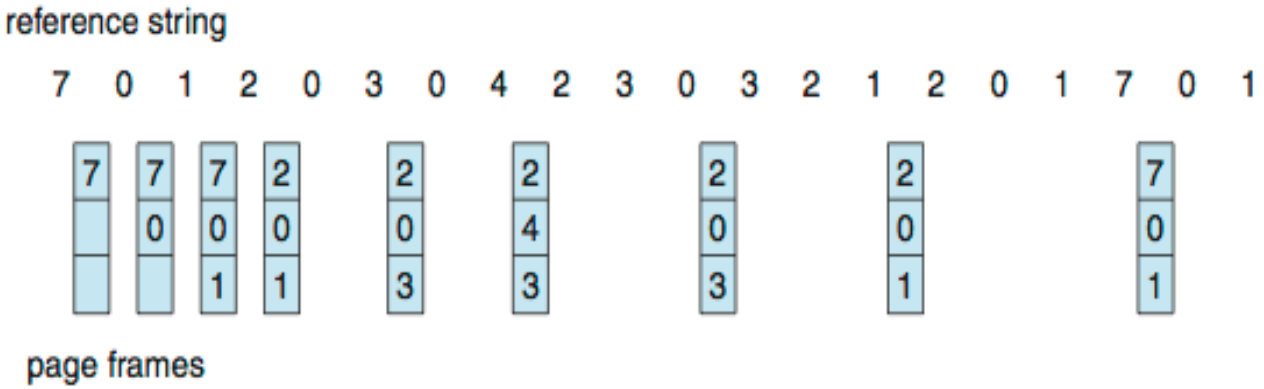
no page fault here

FIFO Illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs

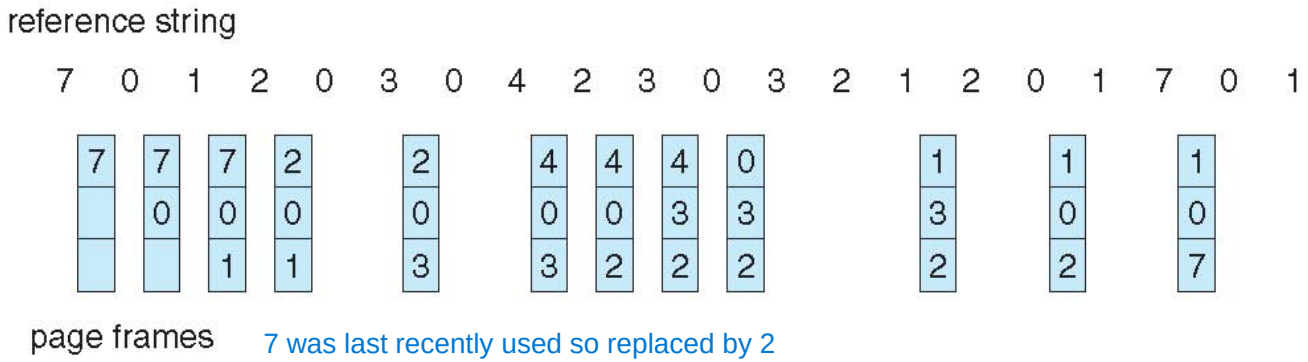


If you already had the string of ref from now to the end. If I knew the future what could I do? I know when the page will be used in the future. after the third one we need a replacement cuz all frames are full so for 2 we need replacement. We look in the future which is the farthest???

Least Recently Used (LRU) Algorithm

I need to look at the past and not consider when a page entered in the system but the last time it was used cuz it is indicative of how much the page has been used in the near past. The closer this page has been used the higher probability that it is still used.

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



7 was last recently used so replaced by 2

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

Least Recently Used Algorithm

On page fault with no free frame, victim is page that has not been used in the most amount of time

Time		1	2	3	4	5	6	7	8	9	10	11	12
String		4	3	2	1	4	3	5	4	3	2	1	5
Fault		*	*	*	*	*	*	*			*	*	*
Frame	0	4	4	4	1	1	1	5	5	5	2	2	2
Frame	1	-	3	3	3	4	4	4	4	4	4	1	1
Frame	2	-	-	2	2	2	3	3	3	3	3	3	5

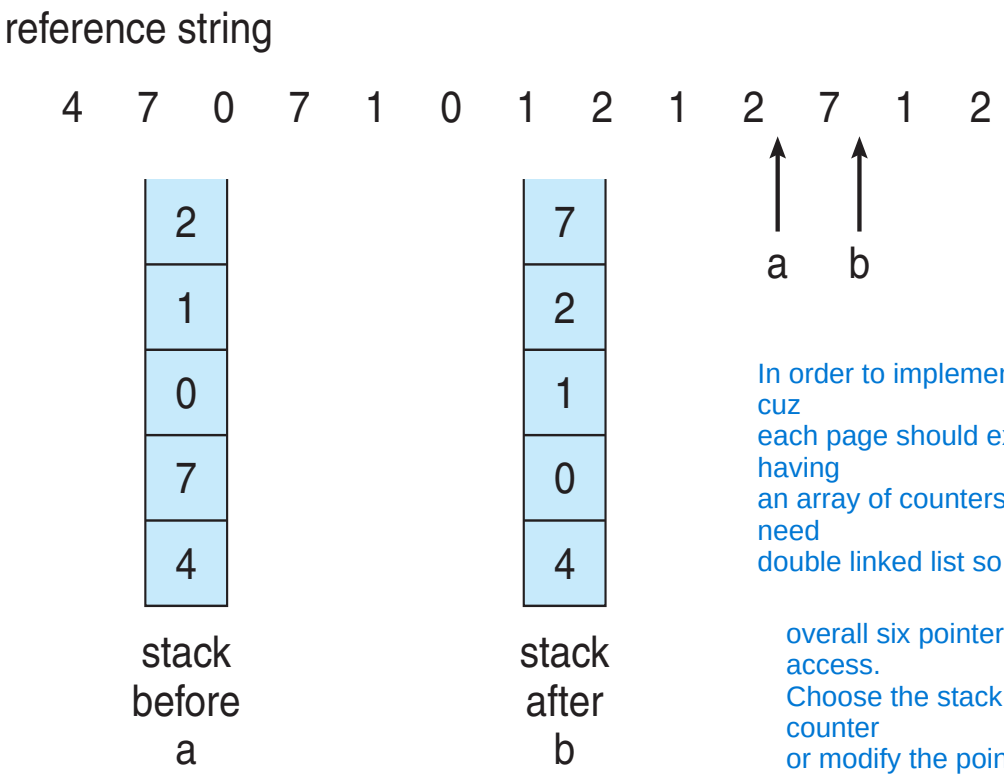
victim

$f = 10 / 12 = 0.83 \Rightarrow 83\ \%$

LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

Use Of A Stack to Record Most Recent Page References



In order to implement this you need a double linked list cuz
each page should exactly have 2 points. Instead of having
an array of counters we have an array of pointers. we need
double linked list so that removal is possible.

overall six pointers to be modified for every memory access.
Choose the stack implementation, either restore the counter
or modify the pointers.

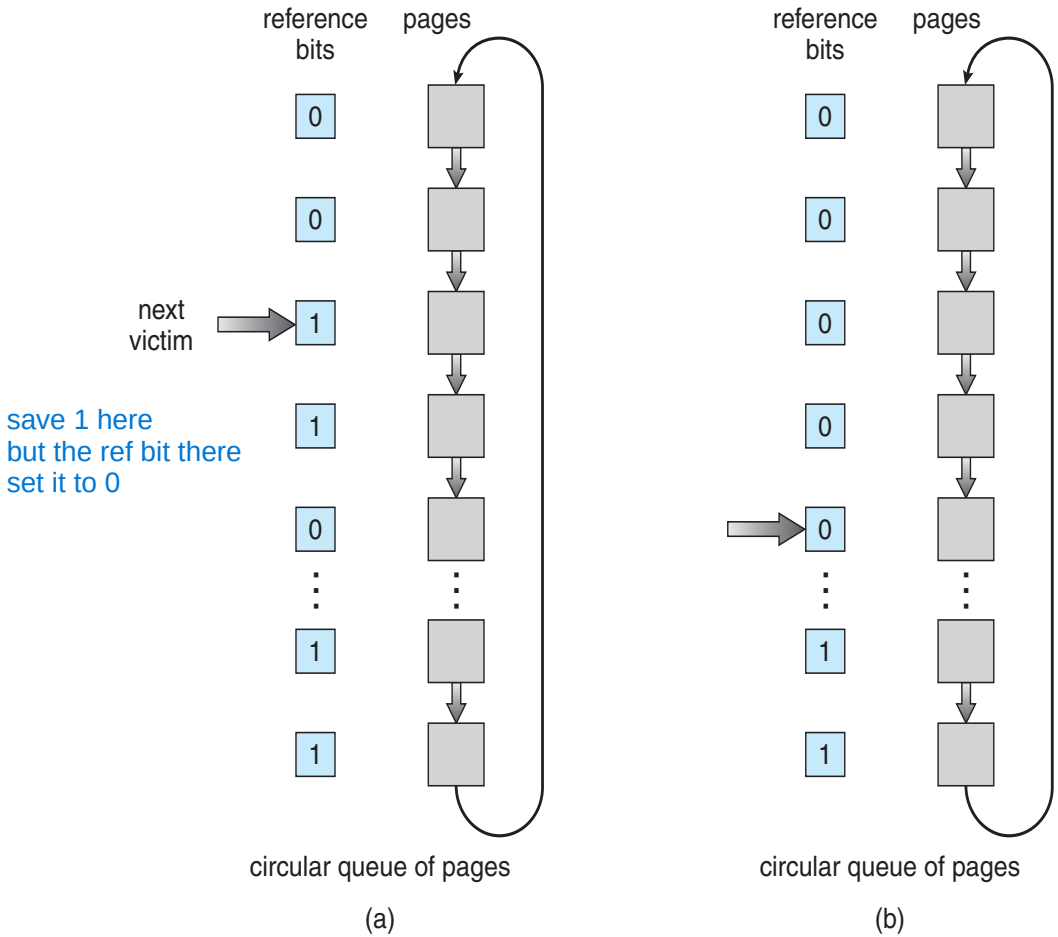
LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

this one is more evolved as it is less vague! When do we clear the ref bit?
it is fifo plus management of ref bit! Clock replacement means circular buffer

The victim in priciple is the oldest that entered the memory.

Second-Chance (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used not modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Least Frequently Used** (**LFU**) **Algorithm**: replaces page with smallest count
- **Most Frequently Used** (**MFU**) **Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Counting how many times a page has been used. The other counter was a counter timer. So many references that the page had and instead of using the least. If the page was used less frequenntly than other pages in the past than it will be used less frequently even in the future. We can have least frequently and most frequently. It is an idea what could be considered as another potential.

Page-Buffering Algorithms

If you don't find a free page, take the victim replace it and free the frame. Why don't we have a pool of free frames and choose from it when required. There should be some moments at time when there are not enough free frames.

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim

Free frame is available when needed but its not created at full time. In other moments we are taking care of creating free frames. The problem with freeing a frame when the page is modified is saving that page back in storage. So the idea is the following: I have a page fault and i need a free frame for that page fault, I use the free frame here and label a victim. So the idea is i take the free frame and find a candidate for another free frame.

- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Another advtg of this kinda buffering is: Suppose the op system has not yet saved the page, you can access the page number in that frame that's marked ready to be freed.

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc

when you have many copies better to use intermediate buffer as you don't req full synconization

A page is considered imp or not imp based on times its accessed.

Allocation of Frames

- Each process needs ***minimum*** number of frames
- 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*
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

The location of frames? We have seen process with three frames, extreme case 1 frame or enough frame for all pages. Which is the minimum num of frames req by a process? Suppose an instruction triggers a page fault! Is it sufficient to get one frame for it?

are you going to trigger 6 page faults yes, but you need 6 frames. So this is simply telling you don't think olnly 1 frame is req.

Its not true! One fault doesn't require one frame. There are instuction that need upto six pages. An instuction typically needs the page where it is, an instruction of 6 bytes could overlap of ending of one and beginning of another page. so for instuction fetch you need two pages. so for 1 fetch you could need 2. most of the instructions when you execute them typically they move data.

when allocating frames to processes.

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames

  equal allocation is fair or unfair based on criteria you are choosing.

  - Keep some as free frame buffer pool

    if a larger process will take more frames the smaller process will take less frames.

- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change
    - s_i = size of process p_i
    - $S = \sum s_i$

      proportional to the relative size of the process
    - m = total number of frames
    - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

2 processes here one with 10 pages and one with 127 pages.

$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$

instead of having 127 we have 64 frames less than half of what is needed!!! its a bug prolly 62 or 64.

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

The process could select frame from all frames.

this opens more flexibility and optimization but also means that a given process can highly be impacted by other processes as well.

Local rep is very simple,

G

it is clear that global replacement has more flexibility and room for improvement but its just impacted by other processes.

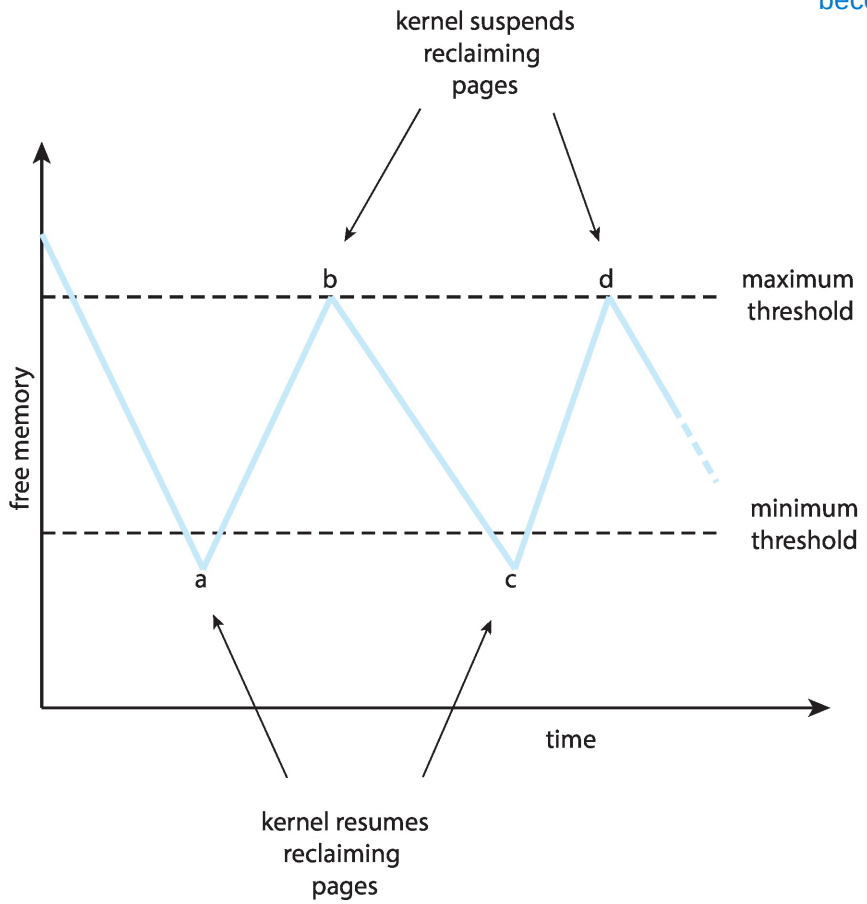
Reclaiming Pages

- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement,
- Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

One possibility of global replacement is the following:

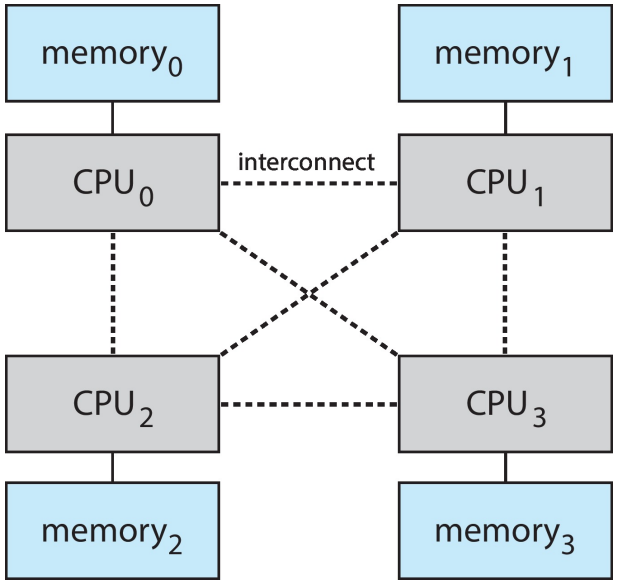
Reclaiming Pages Example

at certain moments the free frame list becomes empty.



Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- NUMA multiprocessing architecture



One should also be aware of the fact that in multicore platforms we have one shared memory which can rather be seen as a network in which we have a distributed memory. Part of the memory can be considered near and fast to access

cpu0 and cpu1 have local memory thats also accessed by other cpu but with more time.

Non-Uniform Memory Access (Cont.)

- Optimal performance comes from allocating memory "close to" the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating **lgroups**
    - Structure to track CPU / Memory low latency groups
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup

Optimal performance should be provided if you allocate frames to processes running on 0.??? not sure bout this one.

when we schedule a process should we place it closer to memory or should be allocate the frames closer in the cpu.

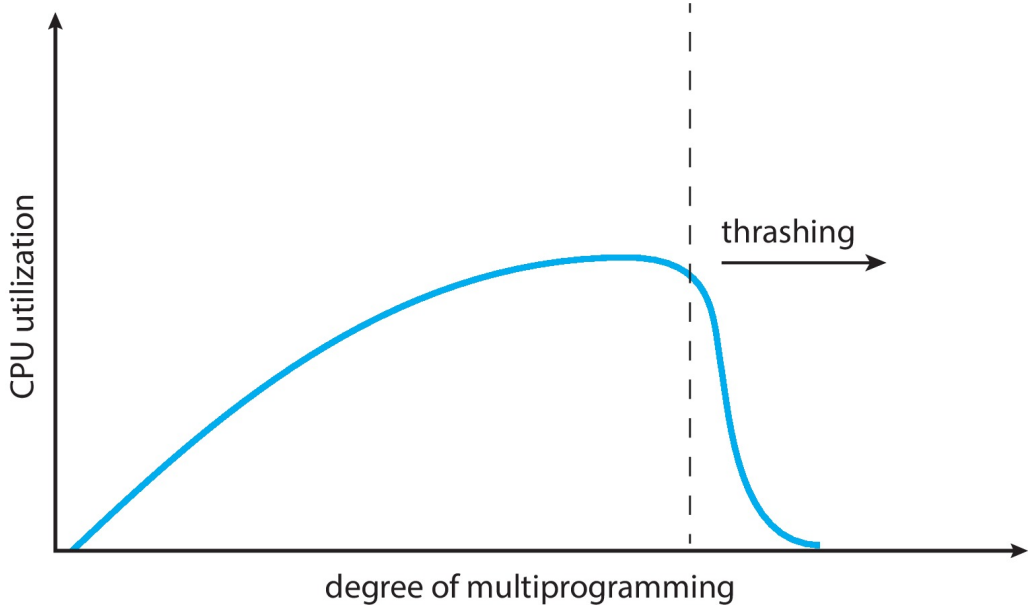
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

Thrashing is a phenomenon in computer operating systems where the system is spending a significant amount of time swapping data between memory (RAM) and virtual memory (disk), due to excessive paging. When a process doesn't have enough pages in memory to execute efficiently, it leads to a high rate of page faults. Page faults occur when the operating system needs to fetch a page from disk into memory because it's not currently present.

Thrashing (Cont.)

- **Thrashing**. A process is busy swapping pages in and out



Demand Paging and Thrashing

- Why does demand paging work?

  Locality model

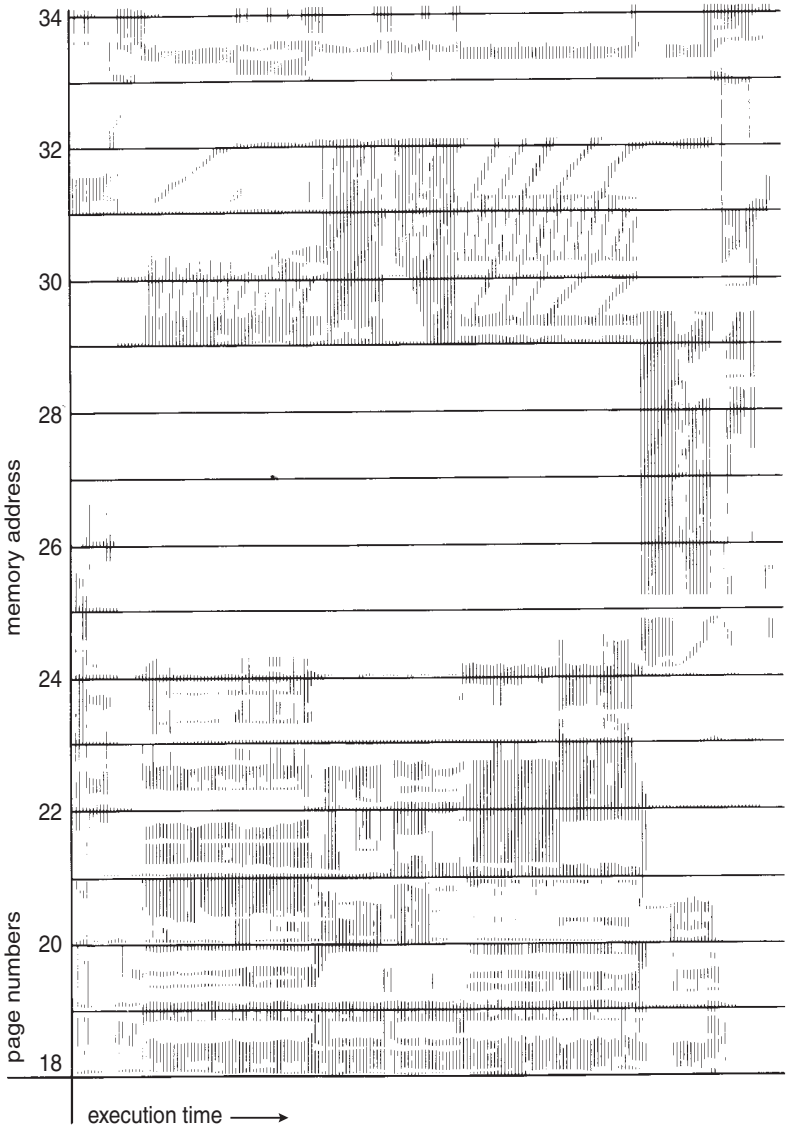
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?

  Σ size of locality > total memory size

- Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern



Working-Set Model

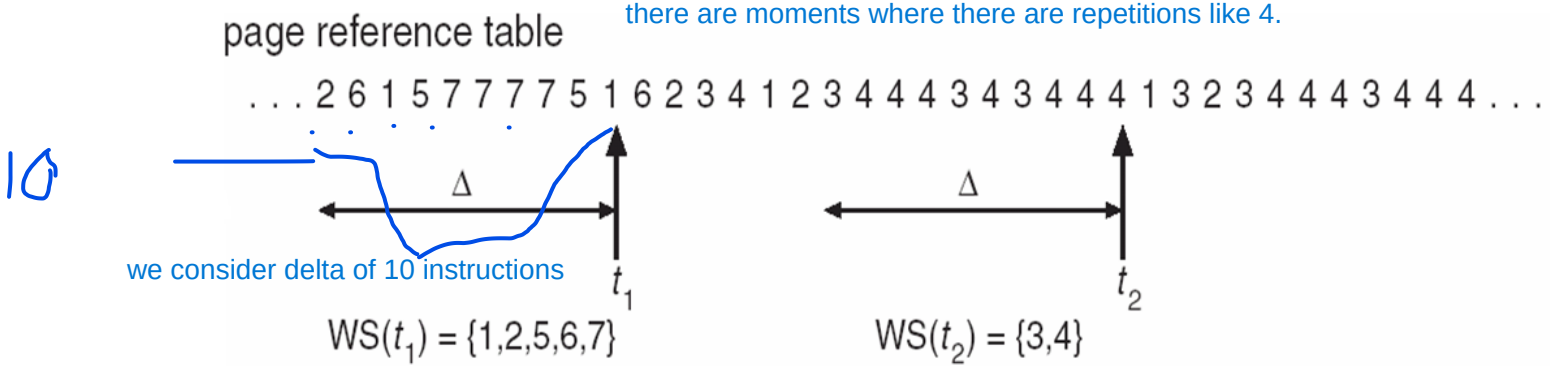
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality

Working set is the set of pages that represent the locality of program. The set of pages where the program is working. How to define this? In some interval time we can say the program was working in that place. A window in time (working set window), over 10000 instructions my working set is been working over those time. working set size is the size of the working set how many pages. 10000 instr how many pages used here? it could be any number upto 10000 (working set size).

Working-Set Model (Cont.)

- if $D > m \Rightarrow$ Thrashing
- Policy if $D >$ m, then suspend or swap out one of the processes

there are moments where there are repetitions like 4.



the working set referred to t1 is the amount of pages that were used in the last delta instructions.

The ws size is 5.

the working set model says lets try to keep in memory the pages i am working on. So far we consider the number of frames is fixed. Now consider if we are able to assign a given process 5 frames here maybe its enough to assign 2 frames in the next one.

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

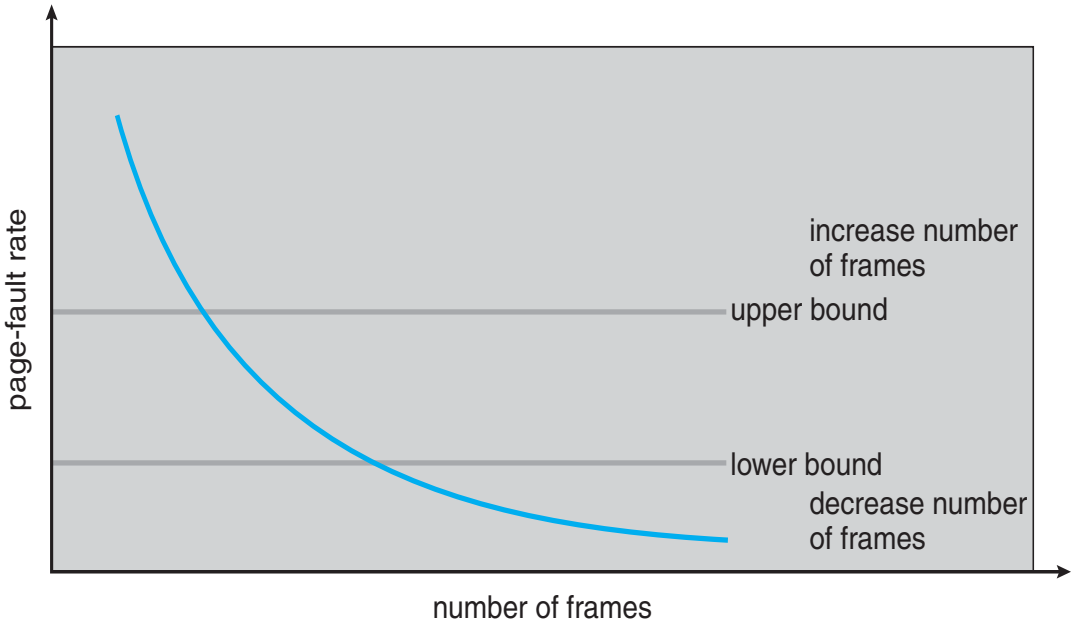
every 5000-time units you try to remove from memory pages not in the working set. By keeping 2 bits in each page,

Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

instead of trying to predict observe the result.

actual rate is below threshhold it means we have too many frames so we can remove frames and if its too high we need to gain frames



Page Fault Frequency Algorithm

- Activated at every page fault, not at page reference.
- Based on time interval τ from previous page fault.
- if $\tau < c$, i.e. page fault frequency greater than desired, add a new frame to the Resident Set of the page faulting process.
- if $\tau \geq c$, i.e. page fault frequancy OK, remove from Resident Set of page faulting process all pages with reference bit at 0;

then clear (set 0) reference bit of all other pages in Resident Set.

This algorithm will also be used in exam. Lets measure the distance between two faults the last and the present page fault. Tou (T) is less than a constant c it means two page faults are closer than a threshold whihc means frequency high. if page frequency hihger distance is more.

if T>=c it means im happy because page fault freq is smaller or equal to constant so i can release frames. if freq is okay i will remove from the resident sent all frames that i can consider out of the working set. How? if they have ref bit 0 it means they have not been used recently so not in working set. All pages that i keep whihc i have reference bit 0 i clear them.

Page Fault Frequency

access with no page fault

Tempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Stringa	**6**	**4**	**4**	**3**	**2**	**4**	**4**	**4**	**1**	**3**	**3**	**2**	**4**	**4**	**5**	**1**	**2**	**2**	**2**	**6**	**1**	**2**	**5**	**4**
Frame 0	6	6	6	6	6	6	6	6	6	6	6	6	6	6	5	5	5	5	5	6	6	6	6	6
Frame 1	-	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	2	2	2	2	2
Frame 2	-	-	-	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	1	1	1	1	1
Frame 3	-	-	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	-	-	-	5	5
Frame 4	-	-	-	-	-	-	-	-	1	1	1	1	1	1	-	1	1	1	1	-	-	-	-	4
Fault	*	*		*	*				*						*	*				*			*	*
Ref Bit									0						0					0				

close together so high freq

simulation can be seen here.

C=3

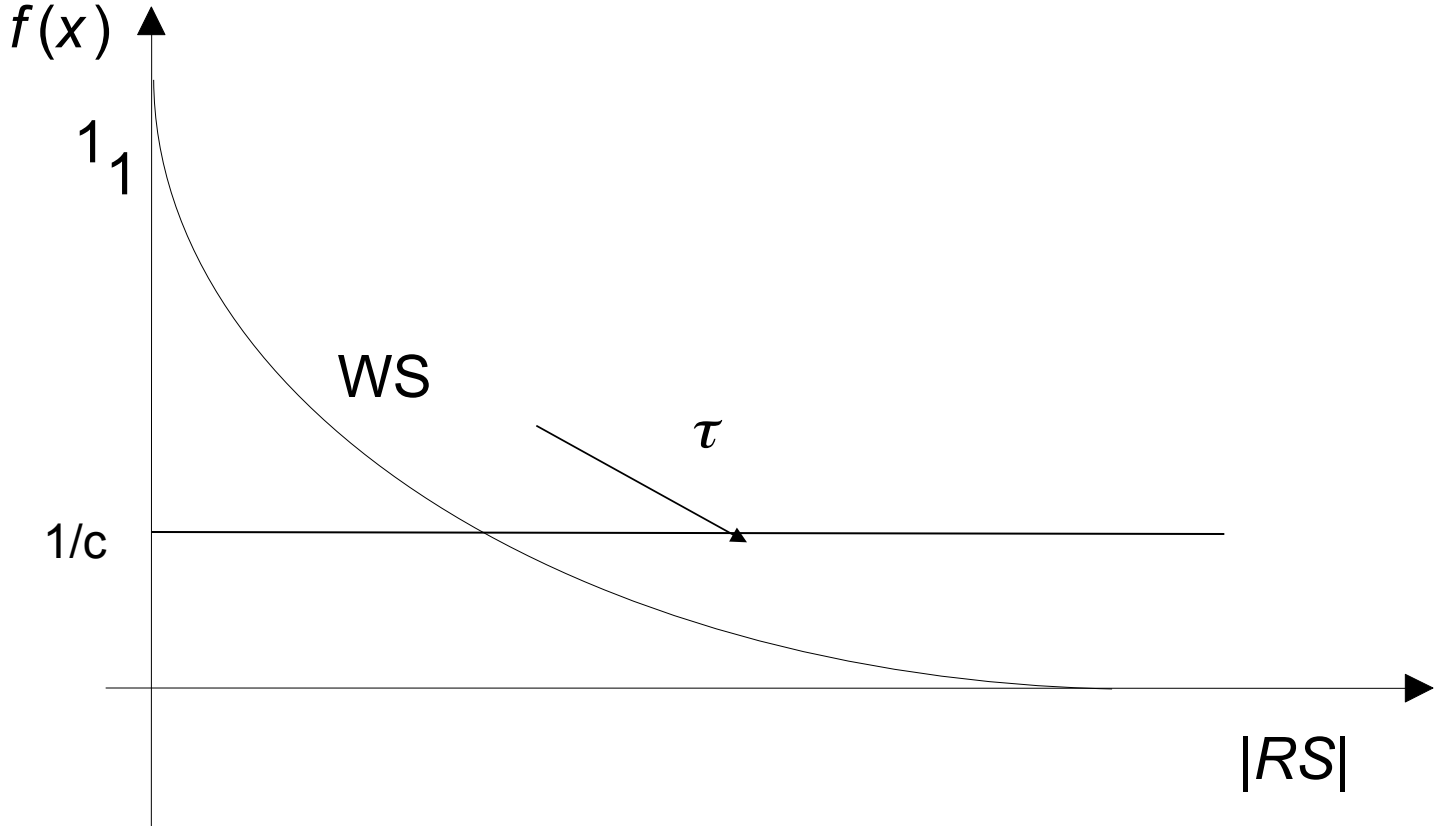
Page referenced

victim

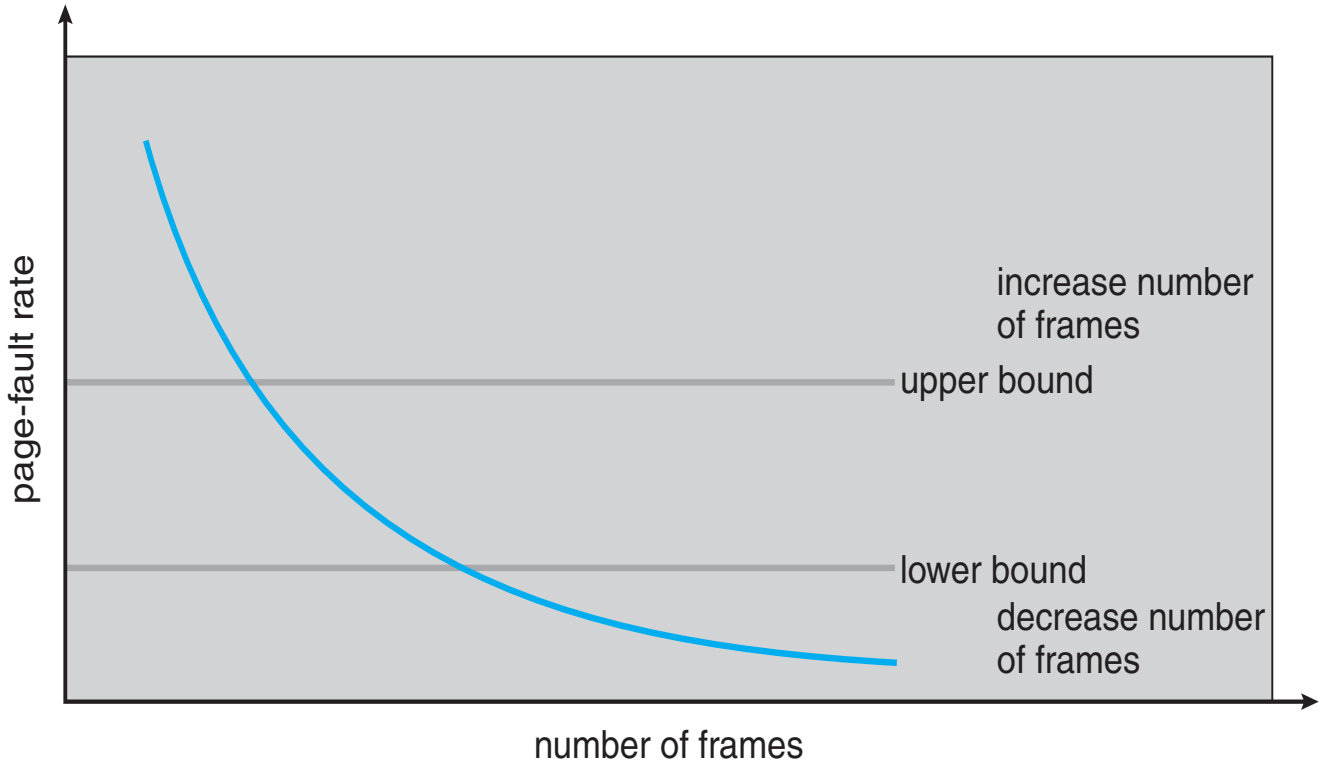
far enough from previous page fault so we activate the algorithm.

the victim are removed from resident set

Page fault frequency function



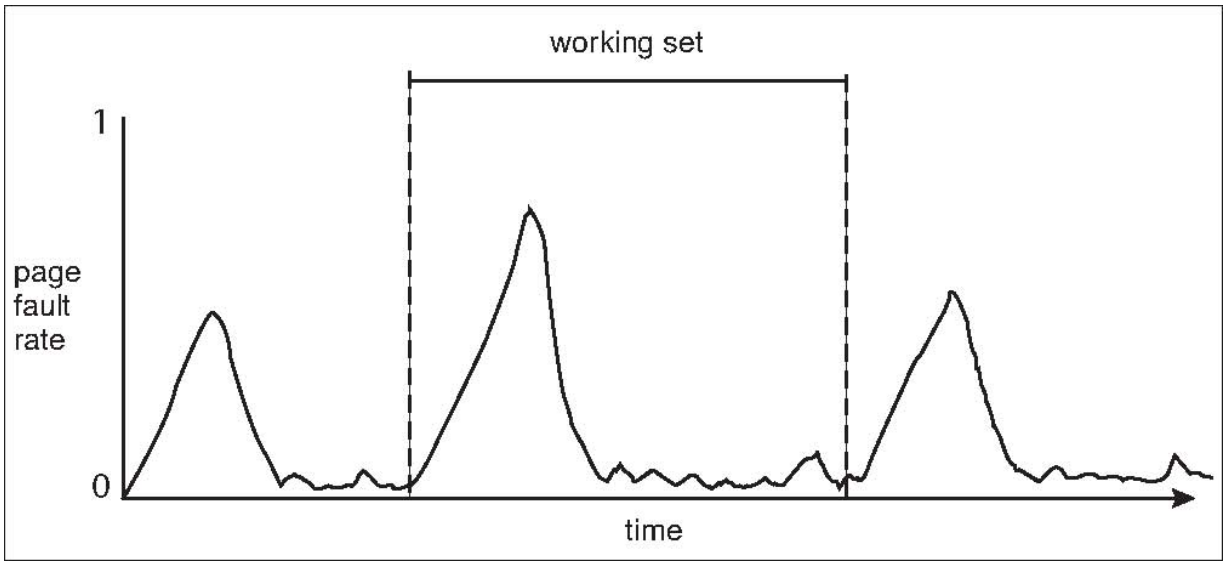
Page Fault Frequency with two thresholds



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time

You have increasing time that means new pages have been introduced. Page faults are increasing or decreasing. Delta should be close to this interval. If you take a delta smaller than this interval you will prolly trigger more page faults.



Allocating Kernel Memory

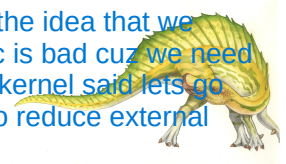
- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O

at the end of the chapter we are saying the kernel is palying another game, the kernel is not necessarrily using the memory simply becasue the kernel has some requirement on contiguous allocation of memory! We will see in os161 the microprocessor has diff partition of memory, for the kernel we will see part of memory is contiguous allocation.
Kernel has other requirements. In most os the kernel will have a set of available frames or memory partition distinct from the user partition. Kernel is needing contiguous allocation. Suppose DMA controllers and other simple devices.

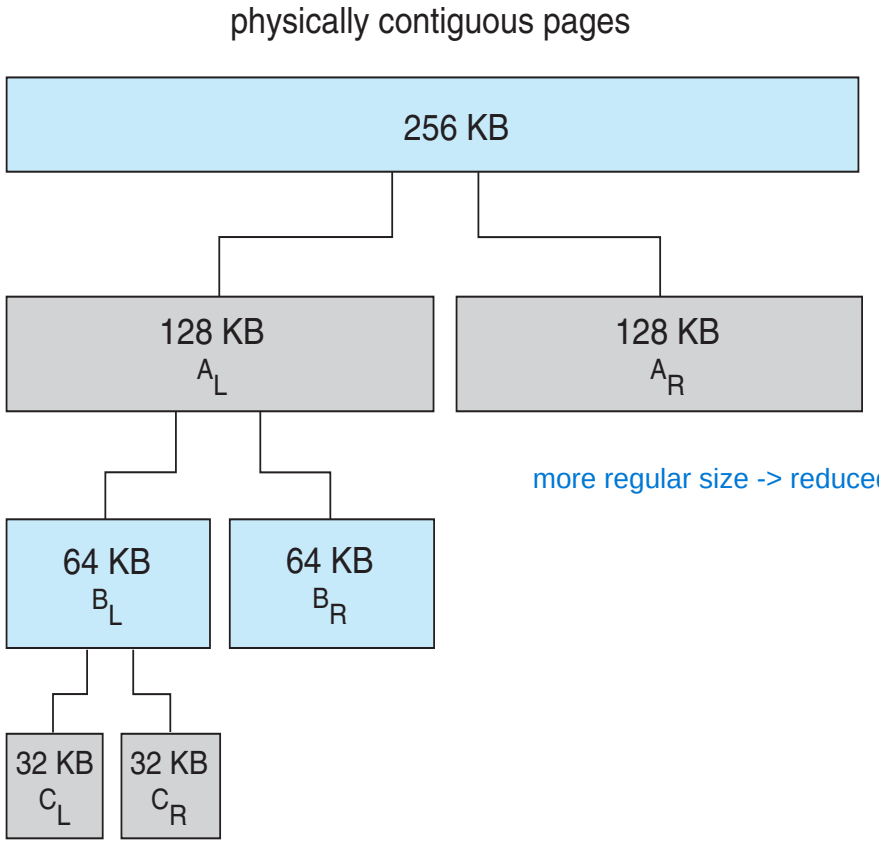
Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation

We see two strategies, the buddy system is based on the idea that we would like contiguous alloc for kernel. Contiguous alloc is bad cuz we need compaction which is expensive and we don't like. The kernel said lets go contiguous but pay something internel fragmentation to reduce external fragmention.



Buddy System Allocator

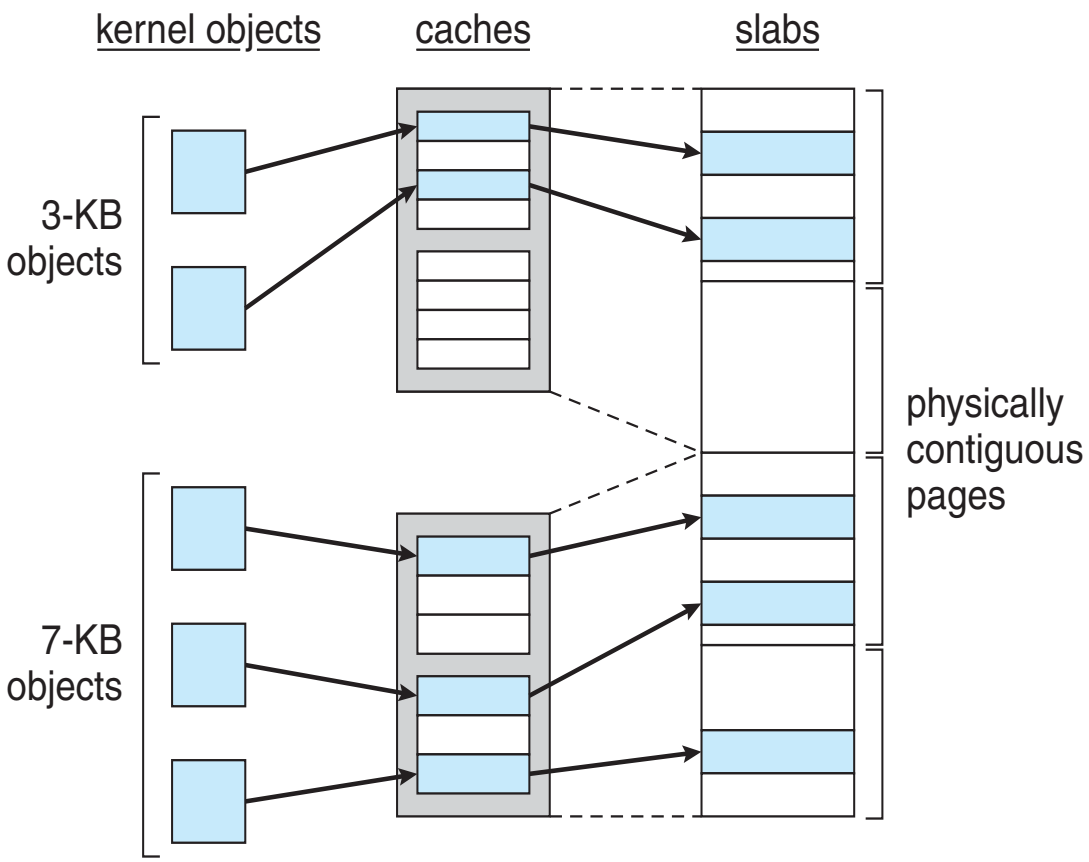


more regular size -> reduced external fragmentation.

Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **`free`**
- When structures stored, objects marked as **`used`**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Slab is one or more physicall contiguous pages, but multiple pages can be not enough or flexible in terms of potential fragmention so you group slabs in caches and you play alloc internally in caches. You can see how k malloc is implemented in os161. Each cache for a particular kernel obj.

Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking

We have considered how use memory is alloc, paging and demand paging.
What can we do to gurantee good performance? We consider few things:
A small variant to demang paging with replacement scheme PREPAGING

Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume *s* pages are prepaged and *α* of the pages is used
  - Is cost of ***s * α*** save pages faults > or < than the cost of prepaging
    s * (1- α) unnecessary pages*?*
  - ***α*** near zero ⇒ prepaging loses

In order to decrease the num of page faults, try to find a good victim when you need a victim. If you know which is the policy, choosing a good victim is to reducing the num of page faults. Another possiblily to reduce the num of page faults when u have time to do it is bringing something in memory which is not in the memory but will be used in the future. Bing able to guess a page not in the memory but will be used in future bring in advance can avoid the next page fault. If prepaging done when I am already playing with a page fault and while bringing page in a memory i bring for eg page b! I speculate when bringing this page in i say okay the next one might be needed as well so this the cost in data transfer is reduced. Instead of starting with zero paging let's guess the initial pages that's needed to start. Is this worse than the cost of what i have been doing. Suppose s pages are prepaged. 100 pages have been prepaged for eg. what percentage of those pages was actually beneficial? Lets say a the ratio of pages effectively used. It means all pages were effectively needed. 1-a is the num or ratio of pages bought in.

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

Can we adjust page size for best performance? Fragmentation means that if we want to minizmize internal fragmentation prolly small pages are better. The only fragmented pages of a given page is the last case. Average case 50% . Small page means. More entry in page table -> larger pages.

I/O overhead will generally prefer larger pages cuz once you pay a certain overhead in order to startup and activate its better to have more data to transfer.

Num o fpage faults, in principle you can say suppose you have large pages, one process stays in a given large page. In principle you can say if pages are very large i am reducing the um of page faults the side effect is onhe big page is prolly going to bring in memory. In a given large page you will prolly have both instruction and data that are being used and ones not being used. Eg one only page in a given process you are only using 30 percent and 70 percent not used. Smaller pages tend to only contain used things

Locality similar reasoning as page faults. If page size larger than locality it means you are including things in mem not needed. Smaller in locality means better cuz locality will be covered by some pages.

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Tlb the num of memory addressees covered by the tlb is given by the num of entries in tlb multiplied by how many pages is a given page covering. Let's say a tlb is not adjustable! Let's say tlb is a part of the microprocessor.

It is clear that if you have 100 entries in tlb, smaller page tables means tlb is covering smaller num of addresses. Most microprocessors especially if the designer or vendor is tryna cover diff platforms.

Program Structure

- Program structure
  - `int[128,128] data;` matrix 128x128
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

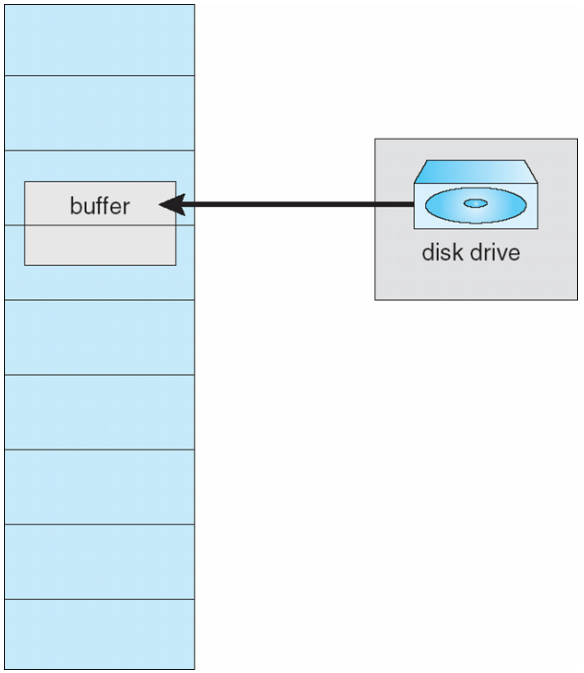
```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

Each row is contained in one page, or each row is spanning over three or four pages. If you want to traverse all the matrix better to go by rows. Suppose if one row is one page. It is clear you trigger more page faults if you traverse by columns. So hence should be by rows.

I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory



The problem of I/O we have talked about in two models. If you have seen two solutions do not move that process. Use kernel buffers.
Consider when talking about swapping in the framework of both contiguous allo and paging. In order to make room for other process to start. swapping could be critical if process you trying to move is in IO. But swapping involed entire process but here we say even with paging when you find a victim there is possible that victime can be locked in io so we can't move back in store.

Operating System Examples

- Windows
- Solaris

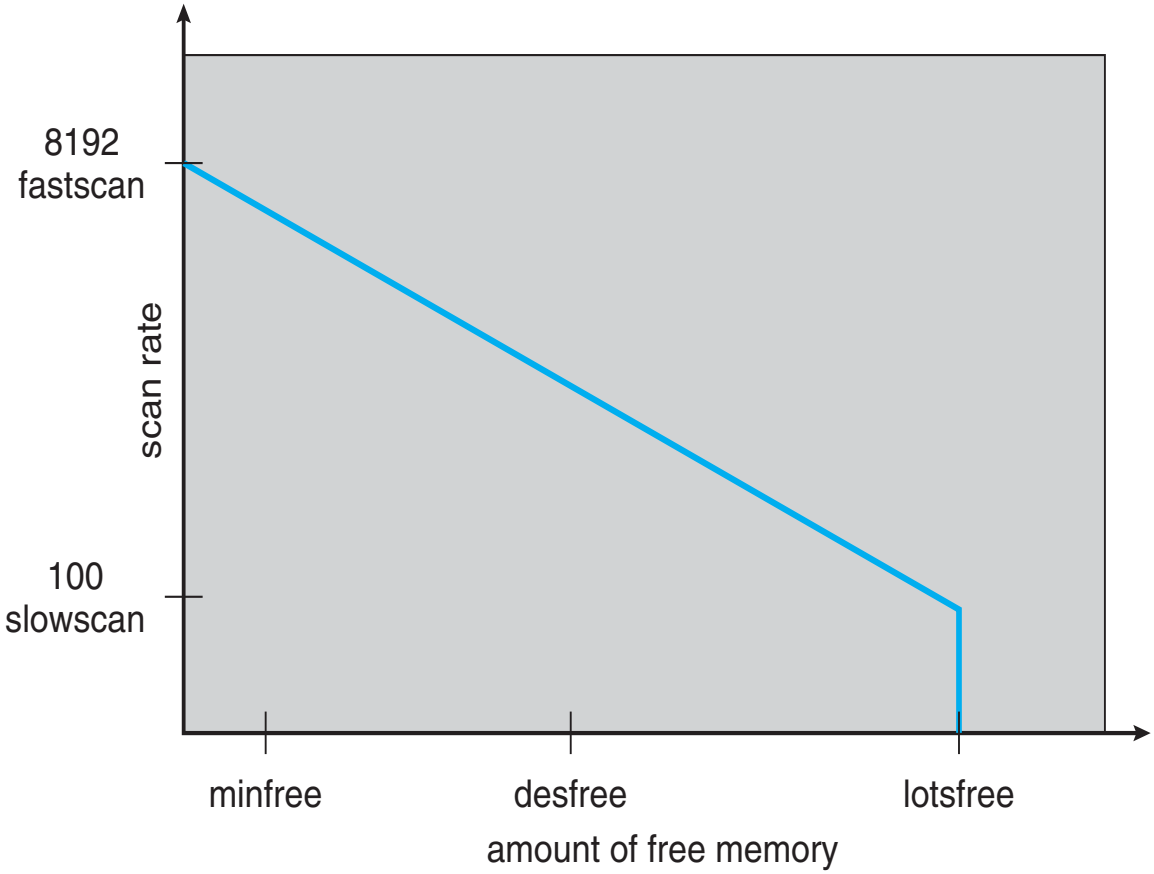
Windows

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

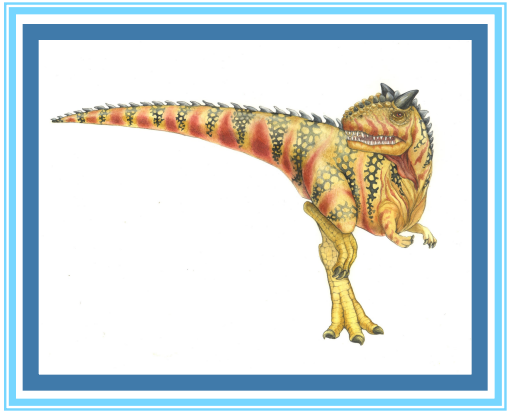
Solaris

- Maintains a list of free pages to assign faulting processes
- **`Lotsfree`** – threshold parameter (amount of free memory) to begin paging
- **`Desfree`** – threshold parameter to increasing paging
- **`Minfree`** – threshold parameter to being swapping
- Paging is performed by **`pageout`** process
- **`Pageout`** scans pages using modified clock algorithm
- **`Scanrate`** is the rate at which pages are scanned. This ranges from **`slowscan`** to **`fastscan`**
- **`Pageout`** is called more frequently depending upon the amount of free memory available
- **Priority paging** gives priority to process code pages

Solaris 2 Page Scanner



End of Chapter 10

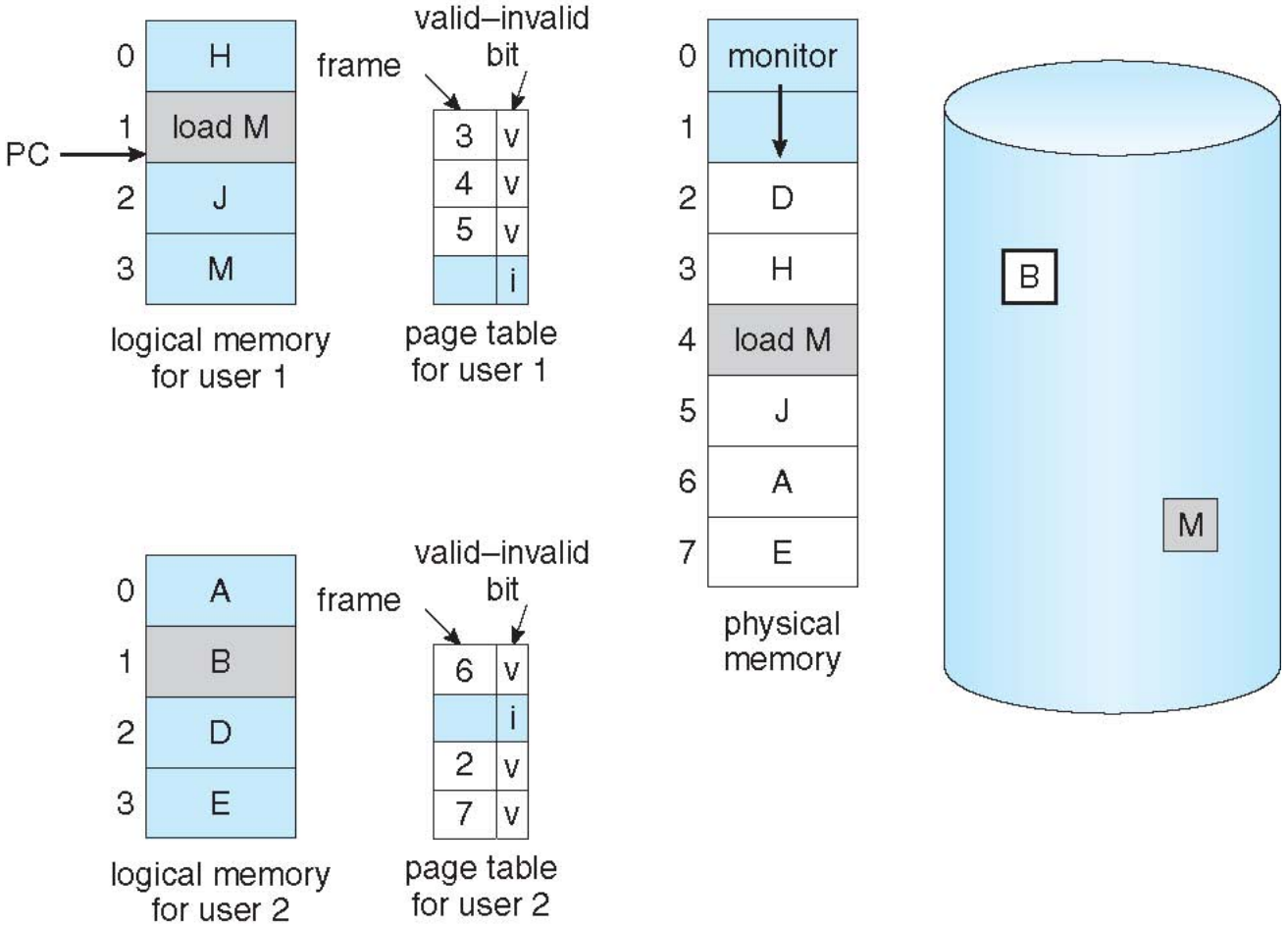


Performance of Demand Paging

- Stages in Demand Paging (worse case)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Need For Page Replacement

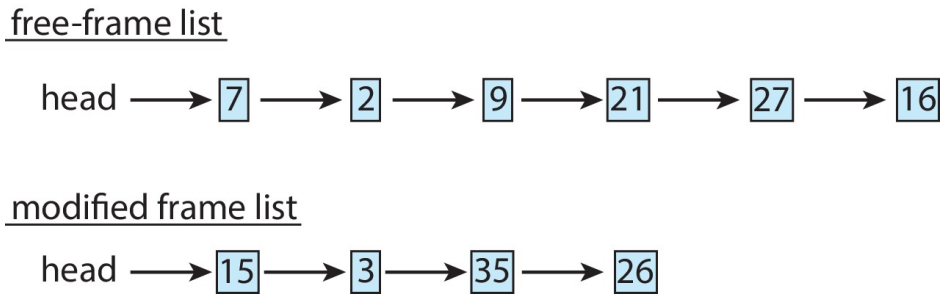


Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Memory Compression

- **Memory compression** -- rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.
- Consider the following free-frame-list consisting of 6 frames



- Assume that this number of free frames falls below a certain threshold that triggers page replacement. The replacement algorithm (say, an LRU approximation algorithm) selects four frames -- 15, 3, 35, and 26 to place on the free-frame list. It first places these frames on a modified-frame list. Typically, the modified-frame list would next be written to swap space, making the frames available to the free-frame list. An alternative strategy is to compress a number of frames{\mdash}say, three{\mdash}and store their compressed versions n a single page frame.

Memory Compression (Cont.)

- An alternative to paging is **memory compression**.
- Rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.