Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read in 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)

$$\begin{aligned} \text{EAT} = (1 - p) \times \text{memory access} \\ + p\,(\text{page fault overhead} \\ + \text{swap page out} \\ + \text{swap page in}\,) \end{aligned}$$

EAT 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

Demand Paging in Mobile Systems

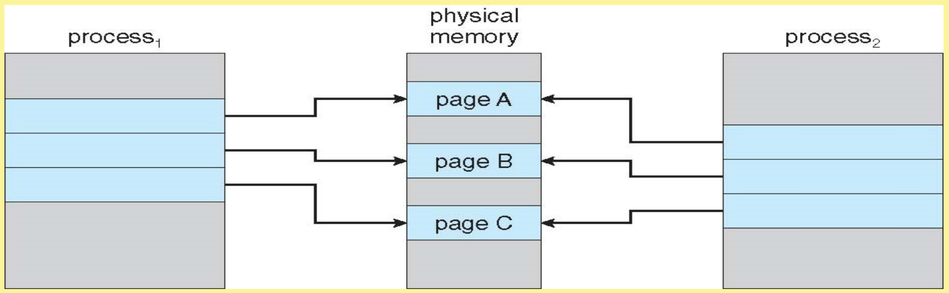
- Typically don’t support swapping
- Instead, demand page from file system and reclaim read-only pages (such as code)

Copy-on-Write

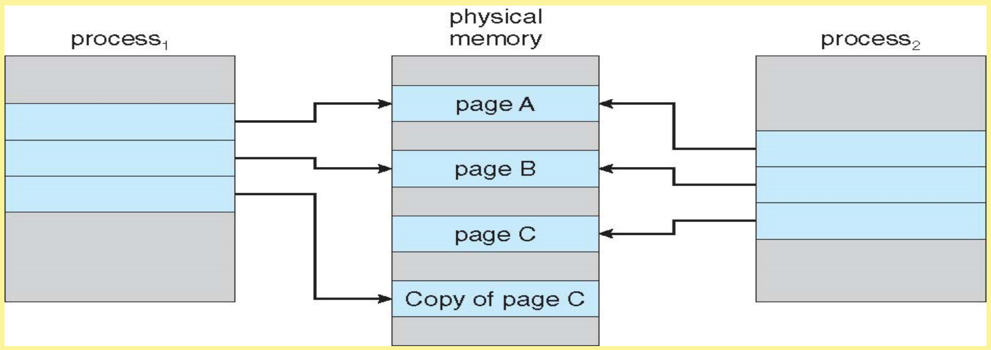
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages (pages whose content has been zeroed out before allocation)
  - 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
- `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 and after Process 1 Modifies Page C

Before



After

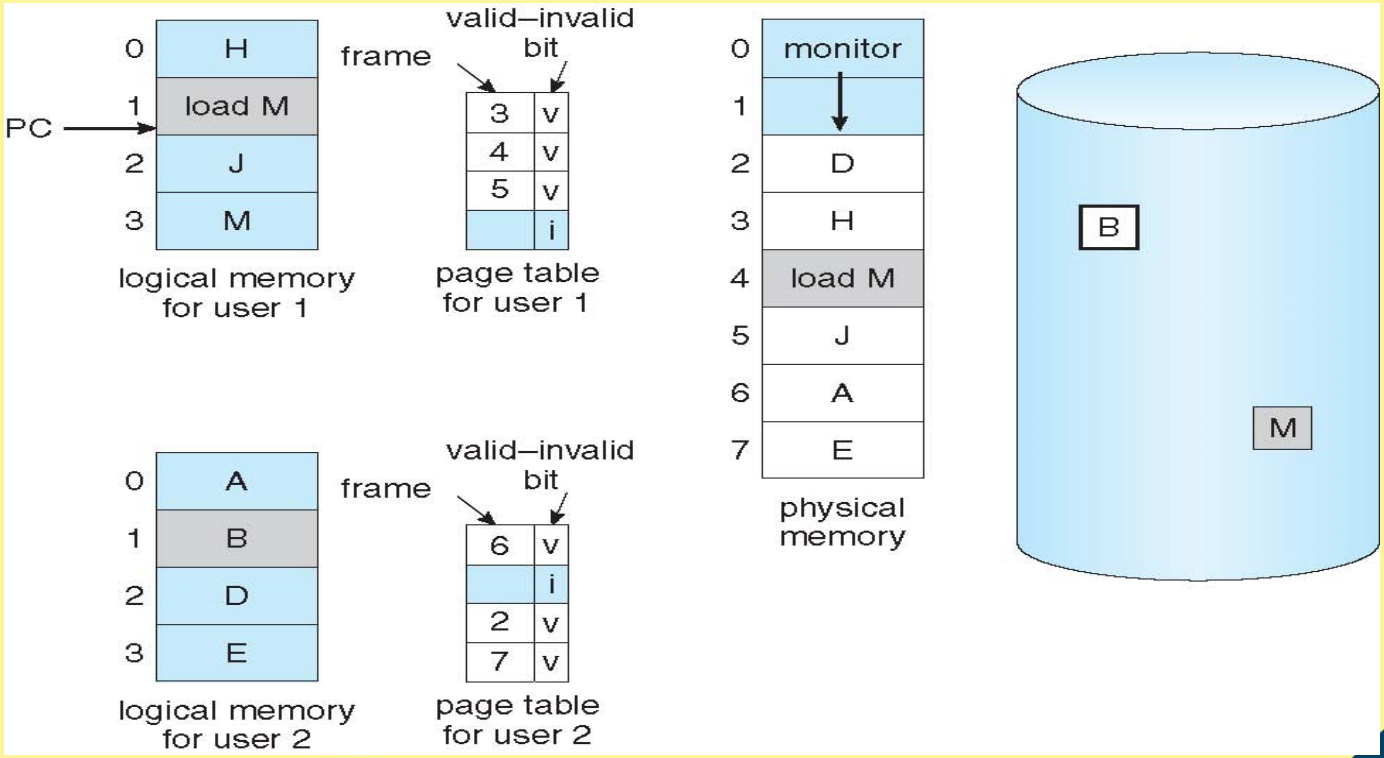


Page Replacement

- Page replacement occurs when:
  - A page fault occurs and we need to bring the desired page into memory
  - There are NO free frames.
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – decide which frame to free
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Need For Page Replacement

All frames are used. No free fames. User 2 needs B



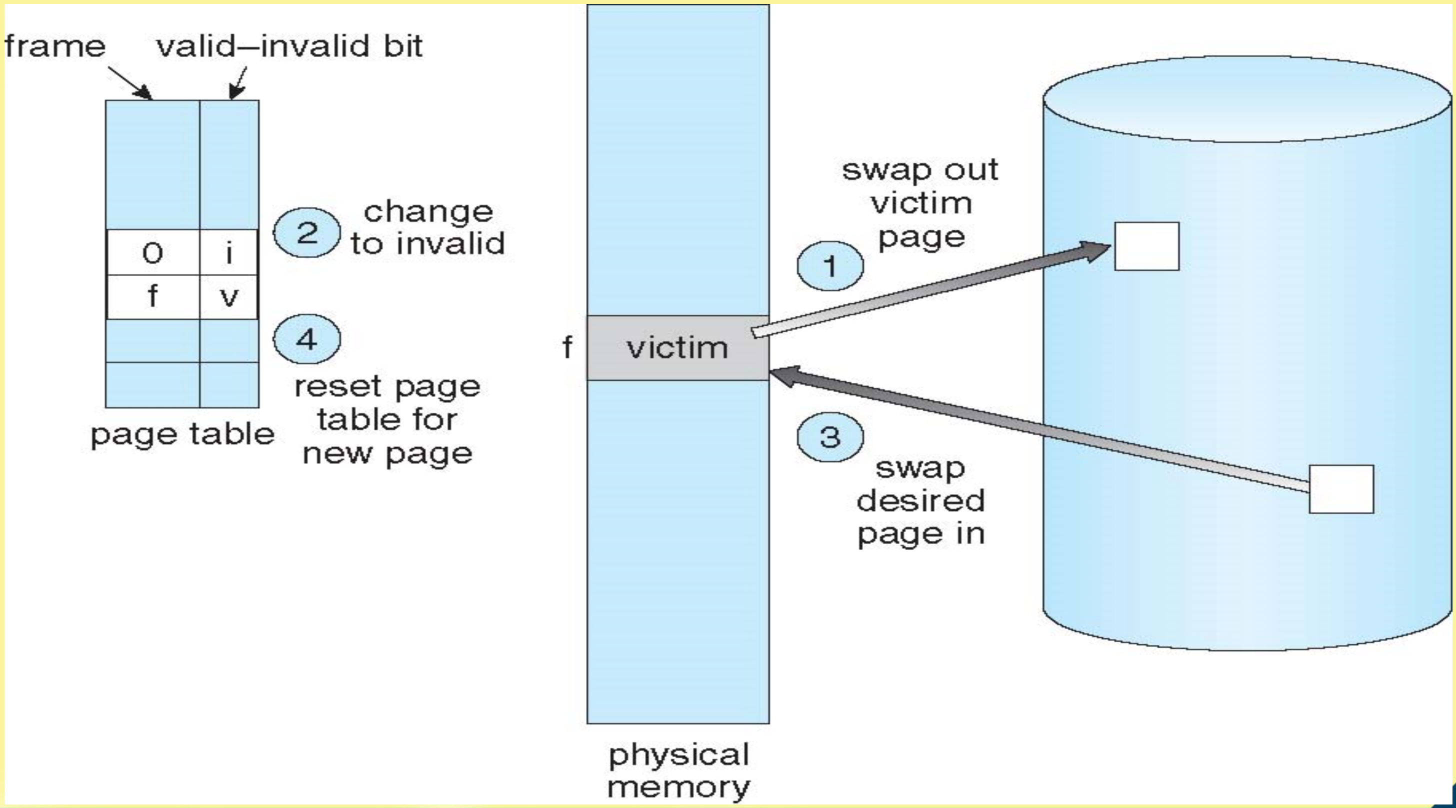
Page Replacement (Cont.)

- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to back to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

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:
     - 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

Page Replacement

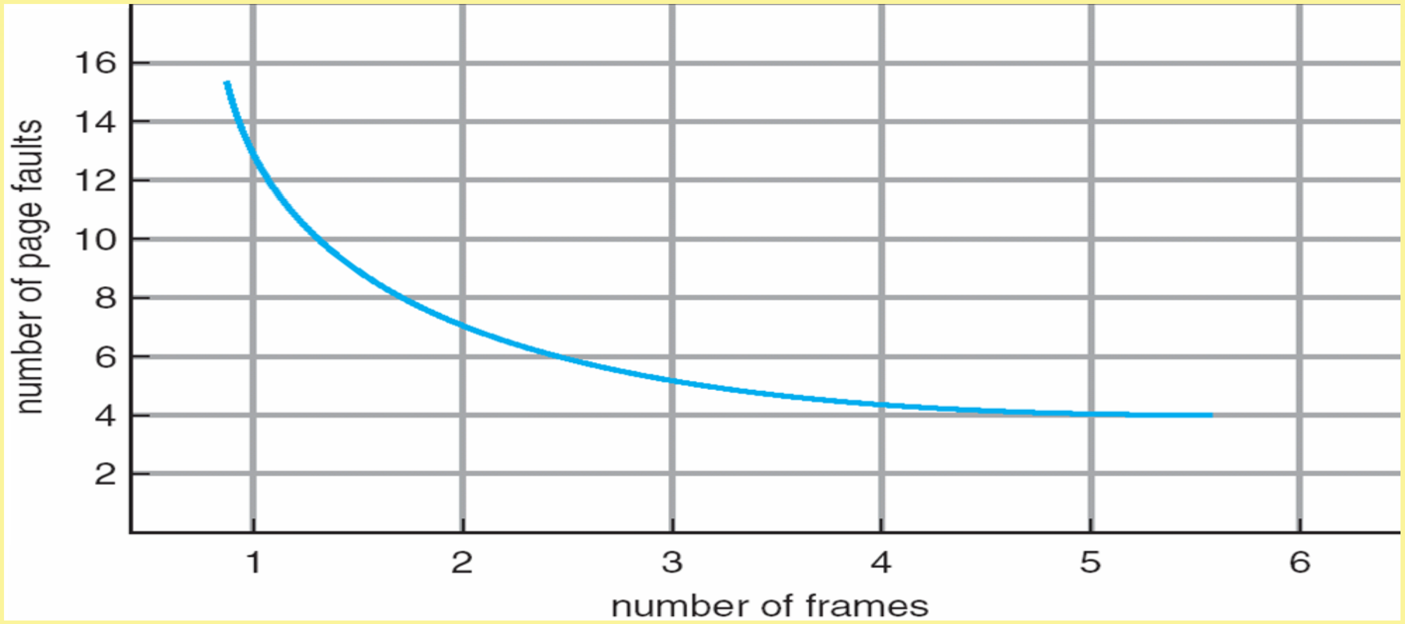


Page and Frame Replacement Algorithms

- **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

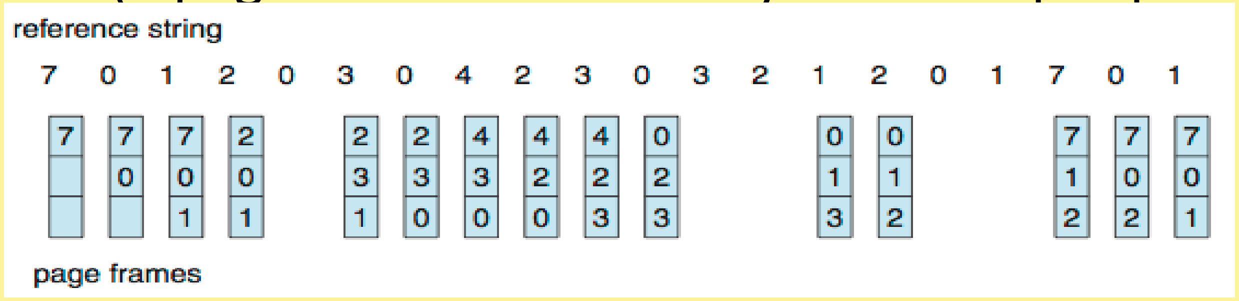
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus The Number of Frames



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)



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

15 page faults

- How to track ages of pages?
  - Just use a FIFO queue

Number of Frames vs Page Faults

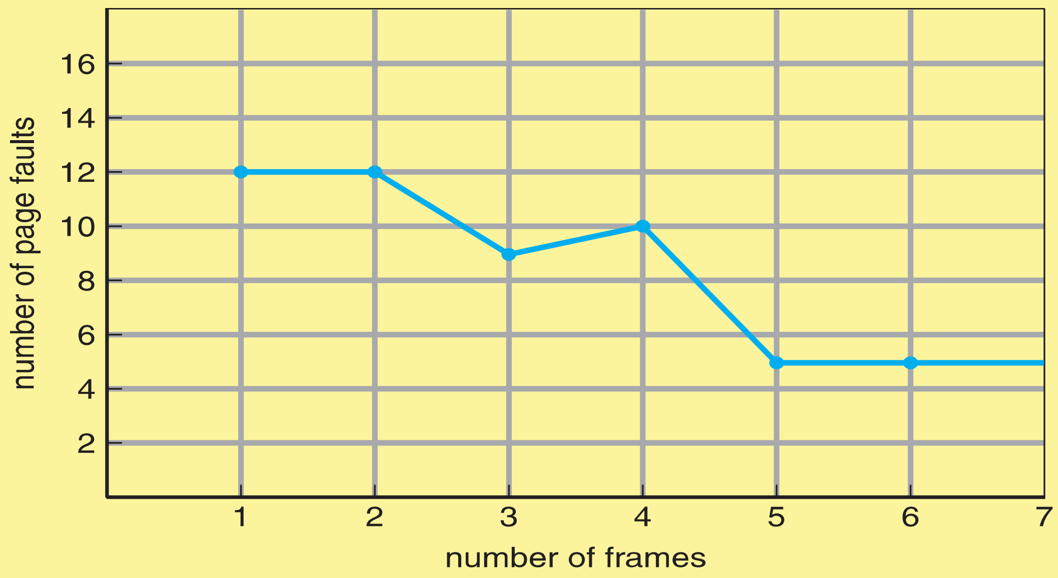
- One would expect that the more frames are allocated to a process the fewer page faults
- Consider the reference string:

  1,2,3,4,1,2,5,1,2,3,4,5

- How many page faults if we have 3 frames?
- How many page faults if we have 4 frames?

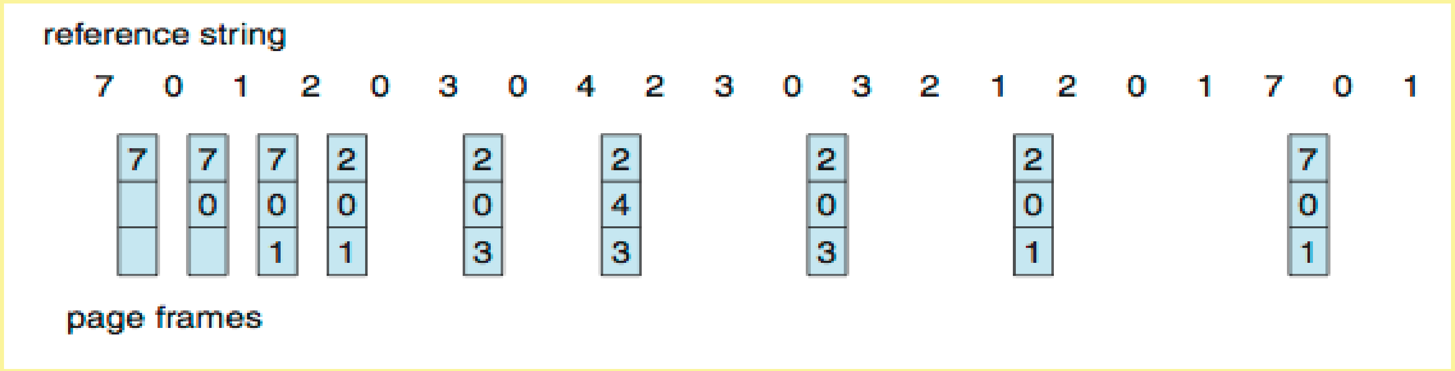
Belady’s Anomaly

- If we use FIFO for page replacement we can encounter the anomaly that more frames may lead to more page faults
- FIFO Illustrating Belady’s Anomaly



Optimal Algorithm

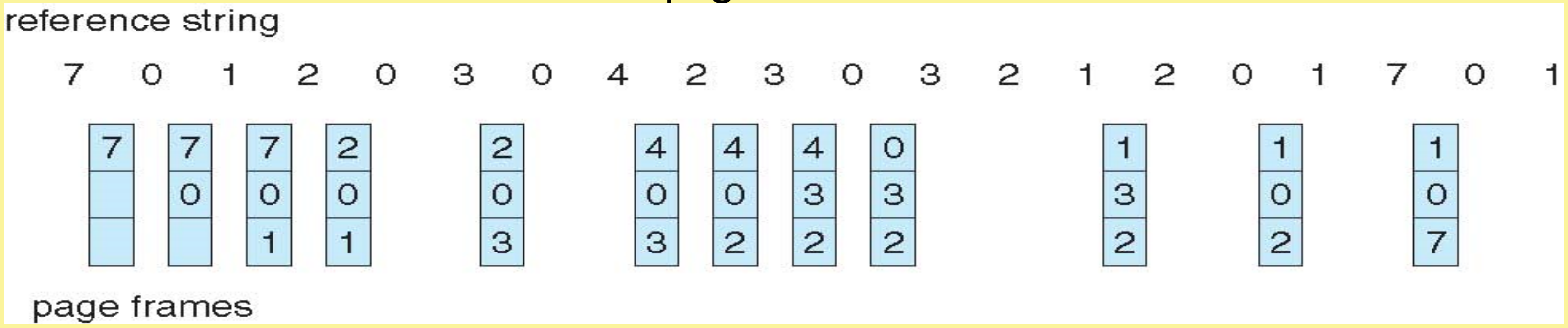
- Replace page that will not be used for longest period of time
  - With 3 frame, 9 is optimal for the example reference.



- How do you know which page will not be used for longest period of time
  - Can’t read the future
- Used mainly for measuring how well a given algorithm performs

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used for the longest period of time.
- Associate time of last use with each page



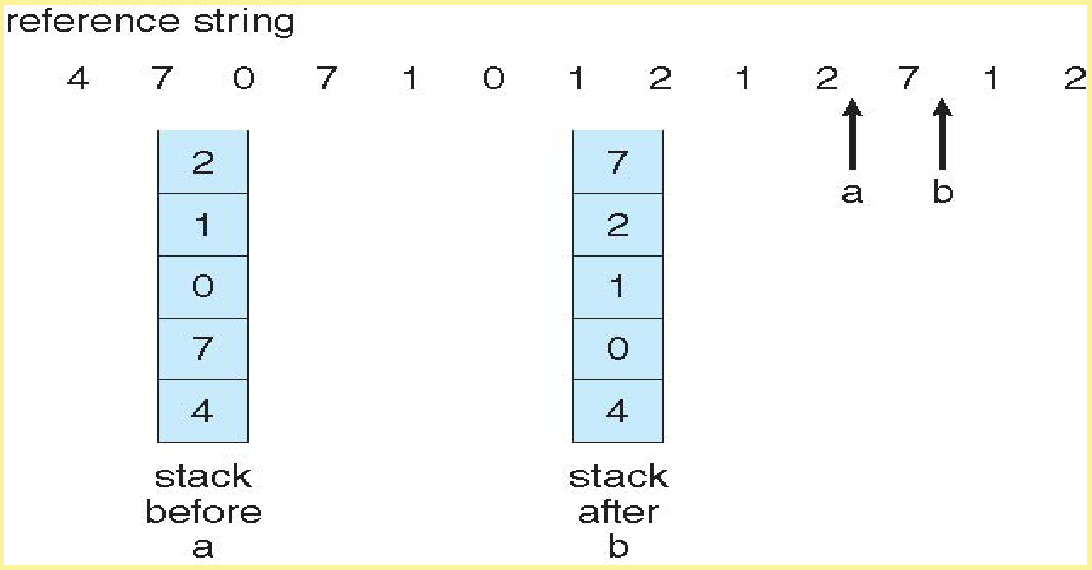
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how do we implement?

LRU Algorithm (Cont.)

- Counter (time-of-use) implementation
  - Every page-table entry has a counter associated with it; every time a page is referenced through this entry, the content of the clock is copied into the counter
  - We replace the page with the smallest time 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
  - No search for replacement
- LRU needs special hardware and is still slow
- LRU and OPT are cases of **stack algorithms** not suffering from Belady's Anomaly

Stack Algorithm Example

Use of a stack to record the most recent page references

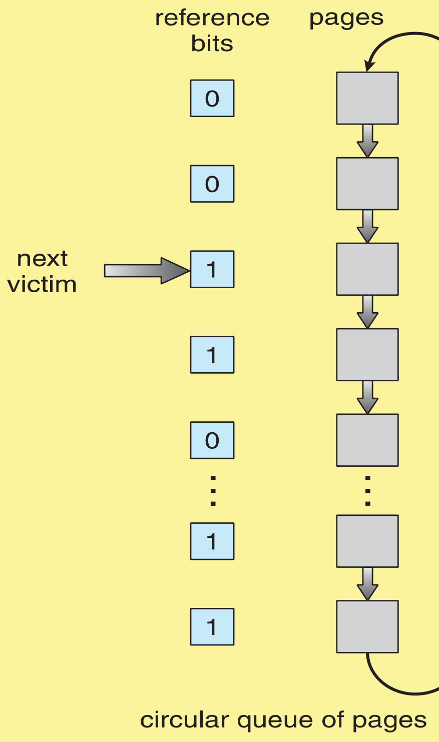


LRU Approximation Algorithms

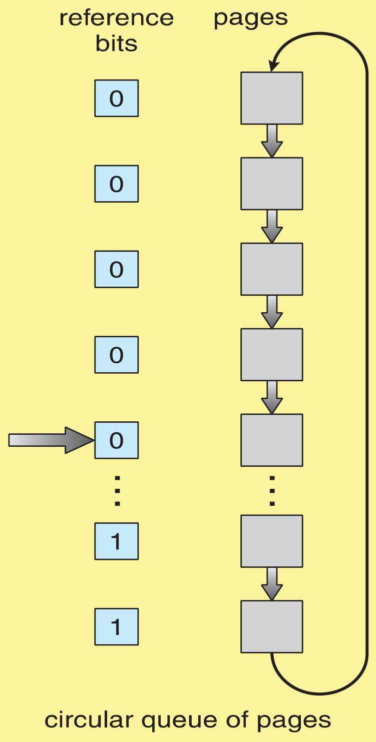
To get efficient implementation we use an approximation of LRU

- **Reference bit**
  - With each page associate a hardware-provided bit; initially = 0
  - When a page is referenced the associated bit is set to 1
  - Replace any page with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance algorithm**
  - FIFO scheme, plus hardware-provided reference bit
  - 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

Second-Chance (clock) Page-Replacement Algorithm



(a)



(b)

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)
  1. (0, 0) neither recently used not modified – best page to replace
  2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
  3. (1, 0) recently used but clean – probably will be used again soon
  4. (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
- **Lease 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
- Both LFU and MFU are expensive to use and are not commonly used.

Page-Buffering Algorithms

Other procedures are often used in addition to a specific page-replacement algorithm. Several schemes:

- Keep a pool of free frames
  - When a page fault occurs, a victim page is chosen as before.
  - The desired page that needs to brought into memory is read into one of the free frame from the pool before the victim page is written out. Start the process immediately.
  - When the victim page is finally written out, the frame is added to pool of free framed
- Maintain a list of modified pages
  - Whenever a backing store is idle, a modified page is selected and written to the disk and its modified bit is reset.
- Possibly, keep a pool of free frame and remember which page was in each frame
  - If page is referenced again before the frame is reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

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

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

Frame Allocation

- Equal allocation – split *m* frames among *n* processes equally -- *m/n* For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

$$s_i = \text{size of process } p_i$$
$$S = \sum s_i$$
$$m = \text{total number of frames}$$
$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 62$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

  - Does a "large" process need more frames? Locality?

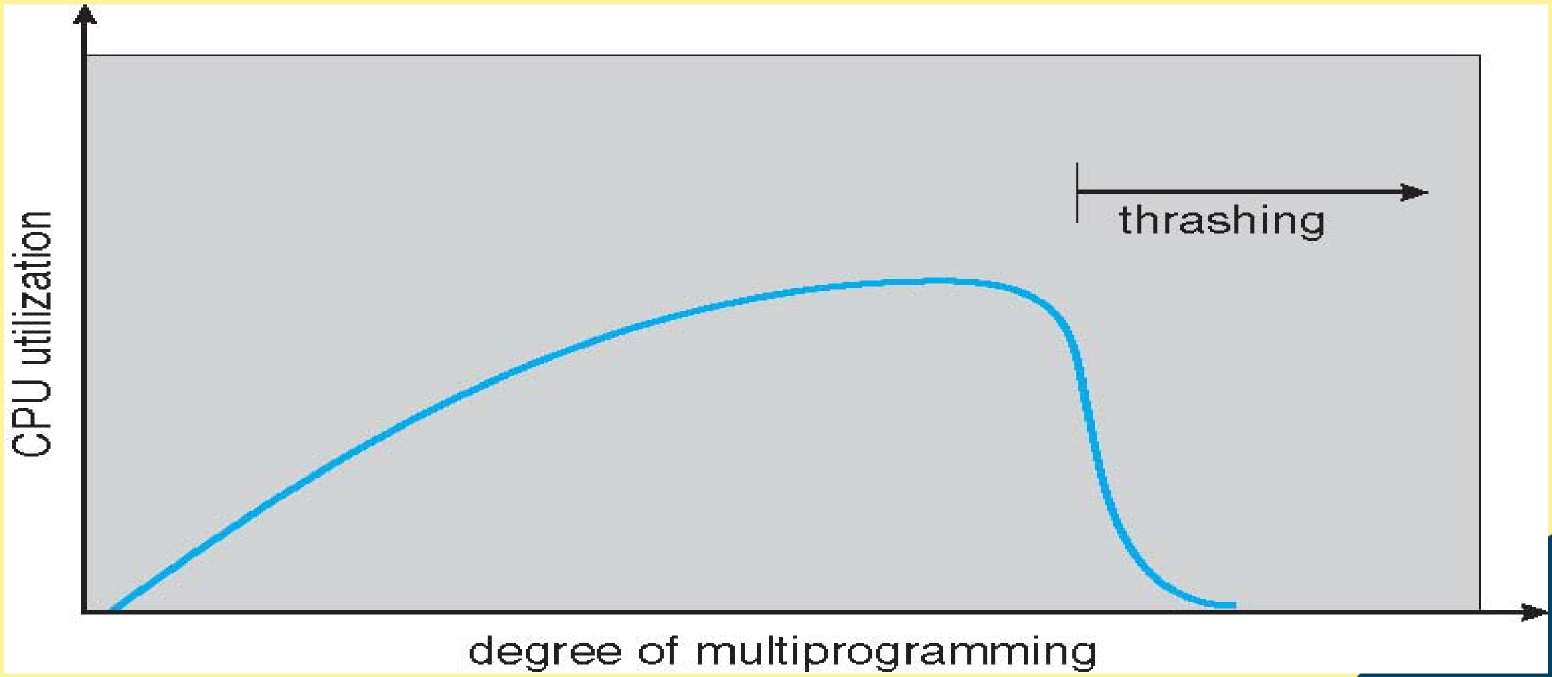
Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - Process execution time can vary greatly
  - A process cannot control its own page-fault rate
  - Greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - If a process does not have sufficient number of frames allocated to it, the process will suffer many page faults (thrashing).
  - Possibly underutilized memory

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** ≡ a process is busy swapping pages in and out

Thrashing (Cont.)



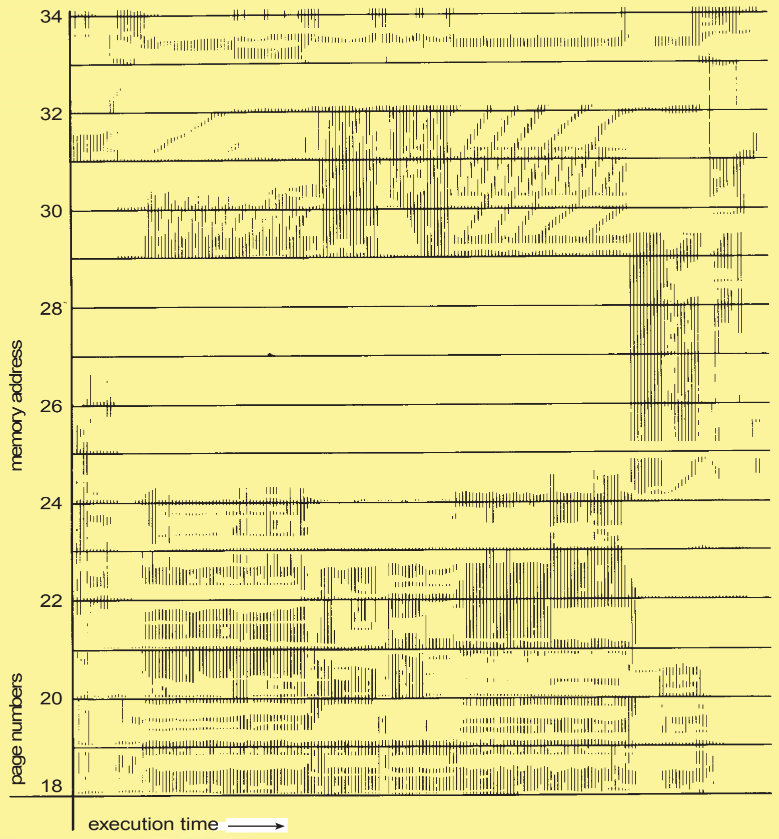
Demand Paging and Thrashing

- 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

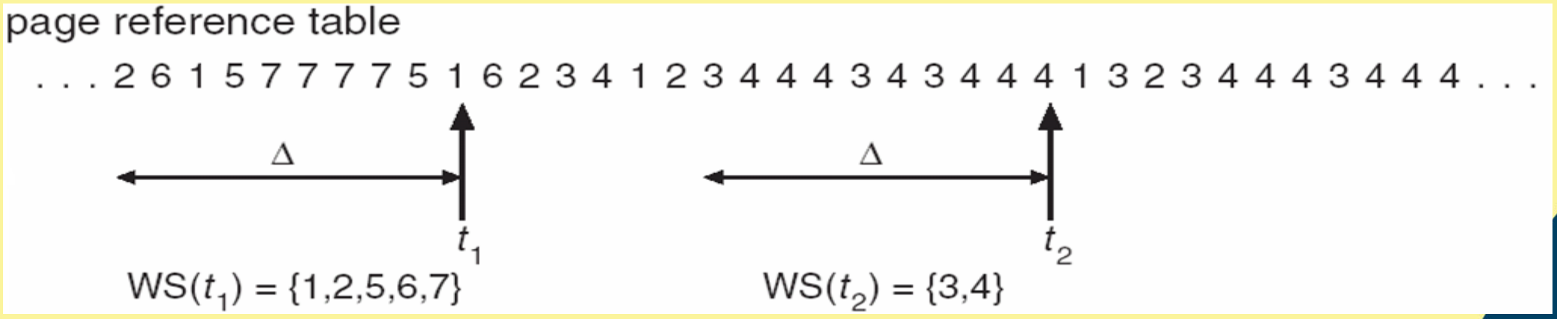
  - Can limit the effects of thrashing by using:
    - Local page replacement
    - Priority page replacement – replace a page from a process with the lowest priority.

Locality In A Memory-Reference Pattern



Working-Set Model

- Define Δ to be a working-set window. Δ is a fixed number of page references For example: 10,000 instructions
- *WSS_i* (working set of Process *P_i*) is defined to be the total number of pages referenced in the most recent Δ (varies in time)
- Example with Δ = 10



Working-Set Model (Cont.)

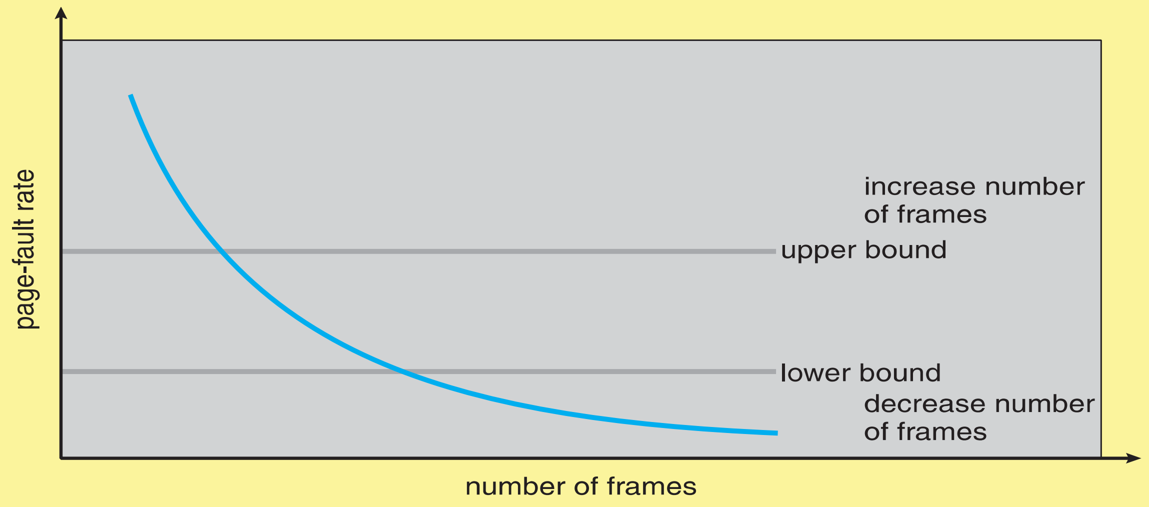
- WSS_i -- tries to approximate the size of the locality of process P_i
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ approximate the total demand frames
  - Approximation of ALL localities
- m = total number of frames.
- if $D > m \Rightarrow$ Thrashing
- Policy: if D > m, then suspend or swap out one of the processes

Keeping Track of the Working Set

- Keeping the exact information about each working set is impractical since the working set window is a moving window.
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page.
  - View the reference bit and 2-in memory bits as a register (3-bits), with the reference bit being the most significant bit
  - Whenever a timer interrupts shift to the right by one place the 3-bits and set the values of all reference bits to 0.
  - If a page fault occurs, we can examine the 3-bits to determine whether a page was used within the last 10,000 to 15,000 references (at least one bit is on). If so the page is in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

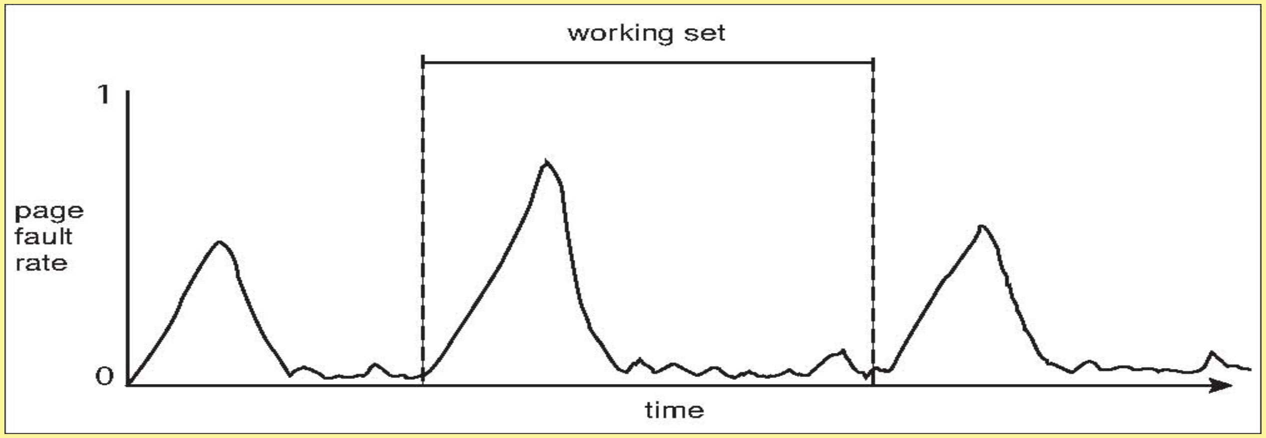
Page-Fault Frequency

- 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



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



Other Considerations

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