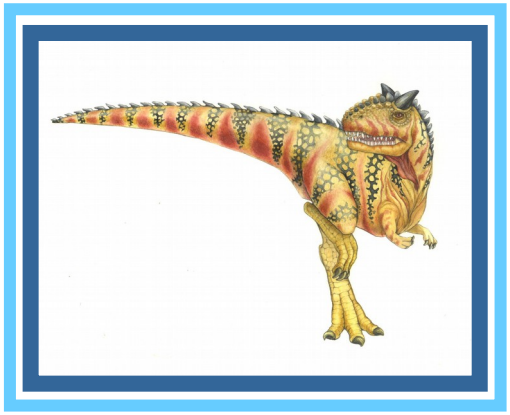
Chapter 10: Virtual Memory



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

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model
- To examine the relationship between shared memory and memory-mapped files
- To explore how kernel memory is managed

Background

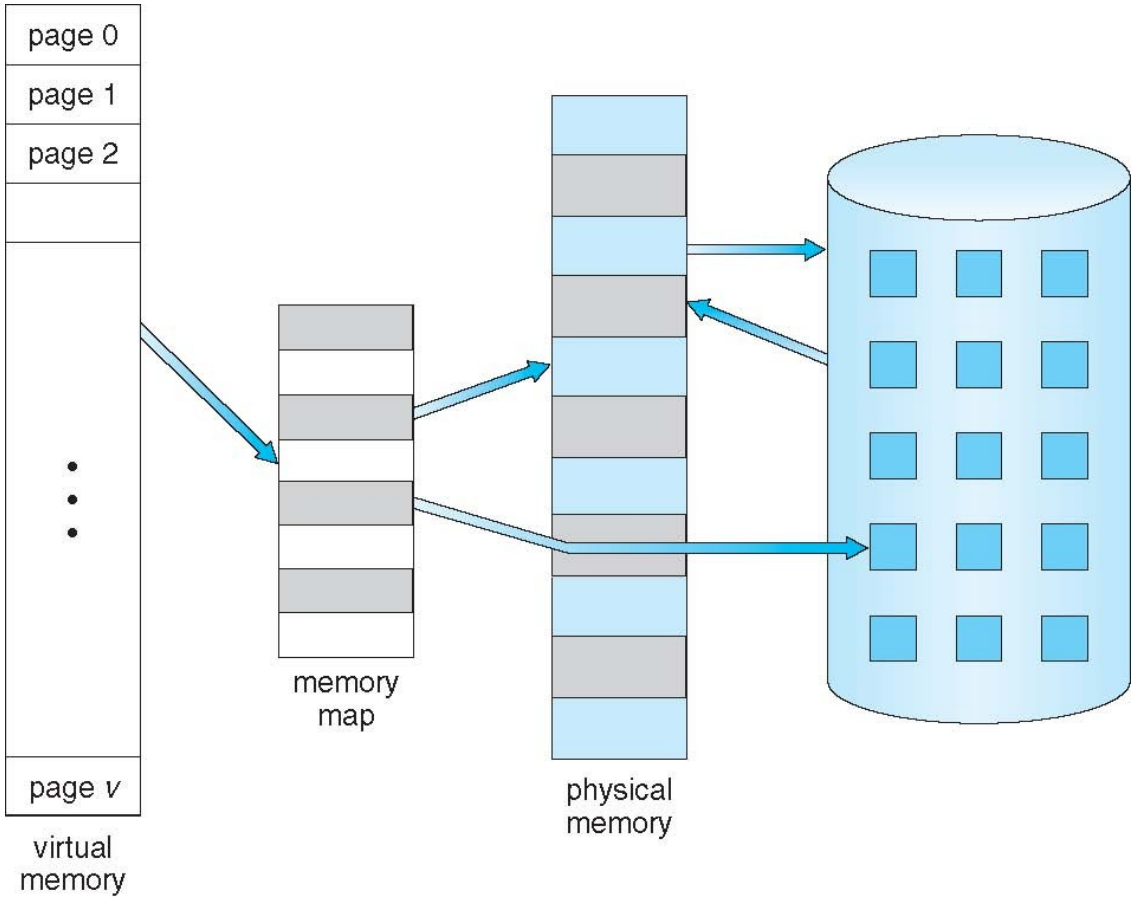
- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running; hence 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; hence, each user program runs faster

Virtual Memory

Virtual memory – separation of user logical memory from physical memory

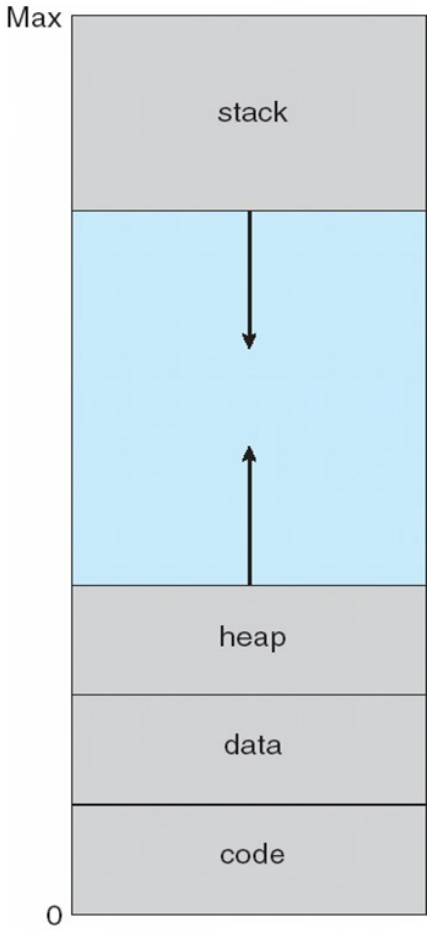
- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger than physical address space
- Allows address spaces to be shared by several processes
- Allows for more efficient process creation
- More programs running concurrently
- Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

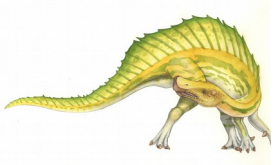
Virtual Memory That is Larger Than Physical Memory



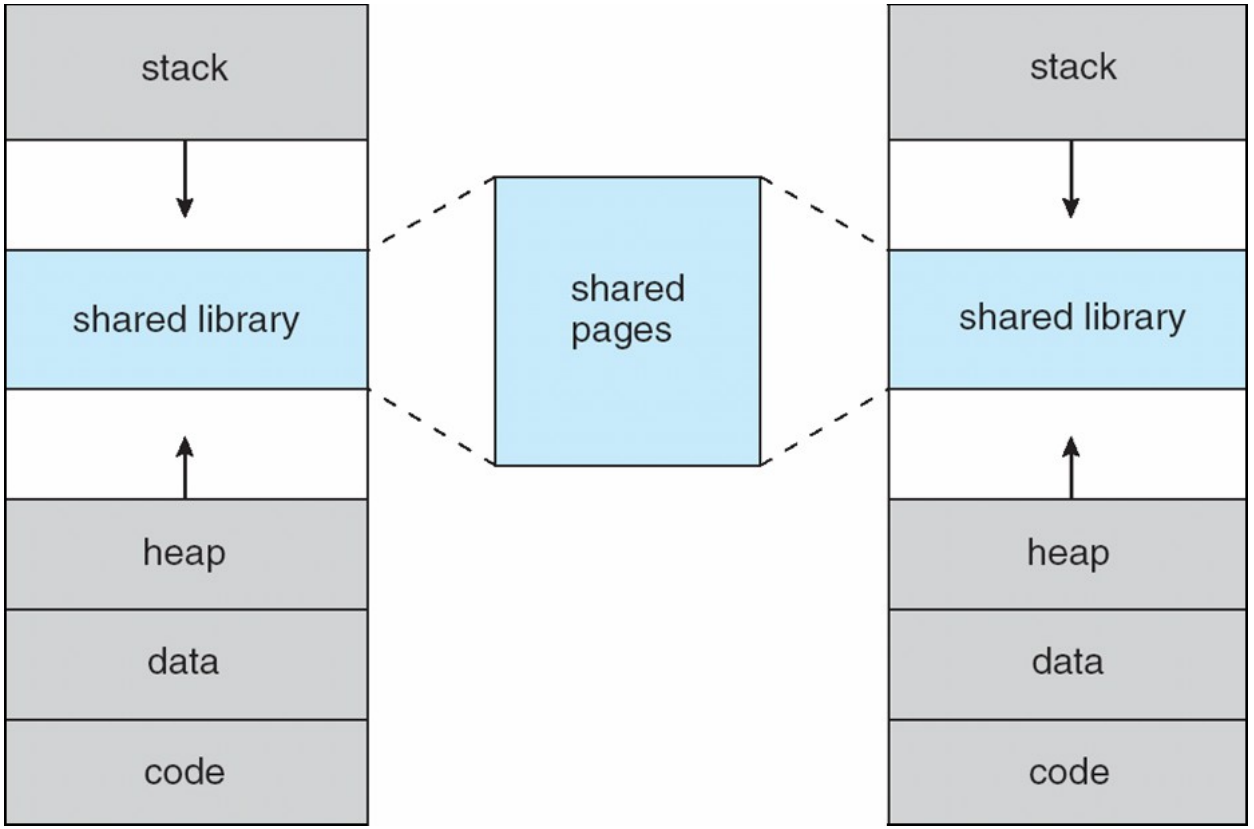
Virtual-address Space

- 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



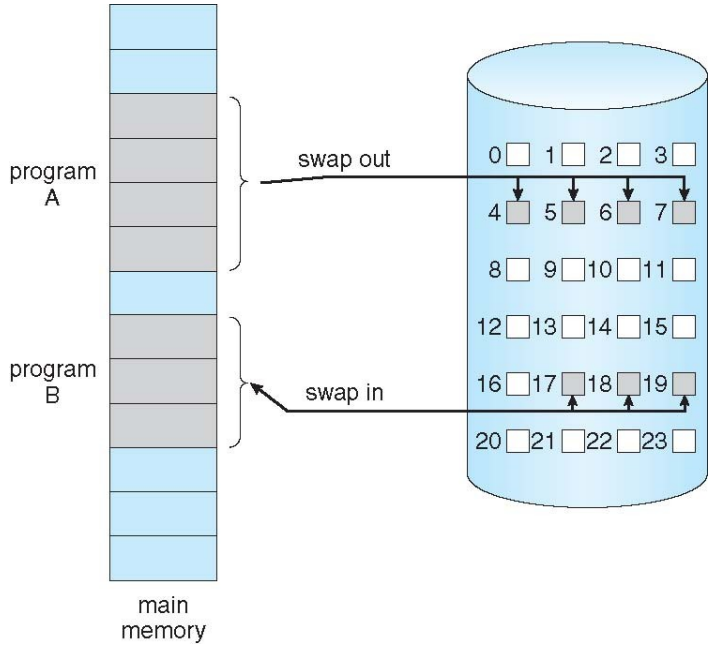


Shared Library Using Virtual Memory



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**

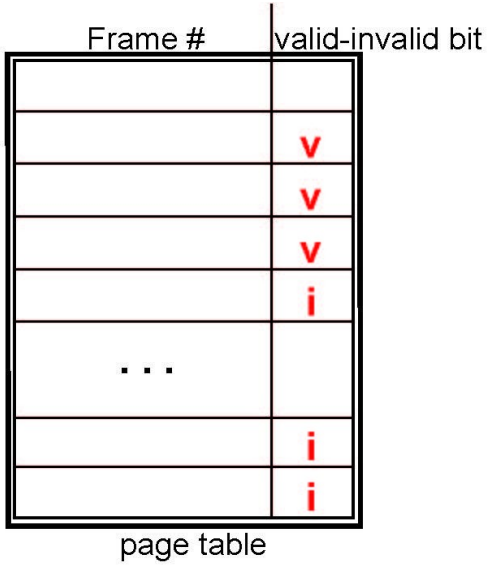


Basic Concepts

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again
- Instead of swapping in a whole process, the pager brings in only those "guessed" pages into memory
- Need new MMU functionality to implement demand paging. Need to distinguish between the pages that are in memory and the pages that are on the disk. Use a variation of the valid-invalid scheme used for protection (see next slide)
- 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

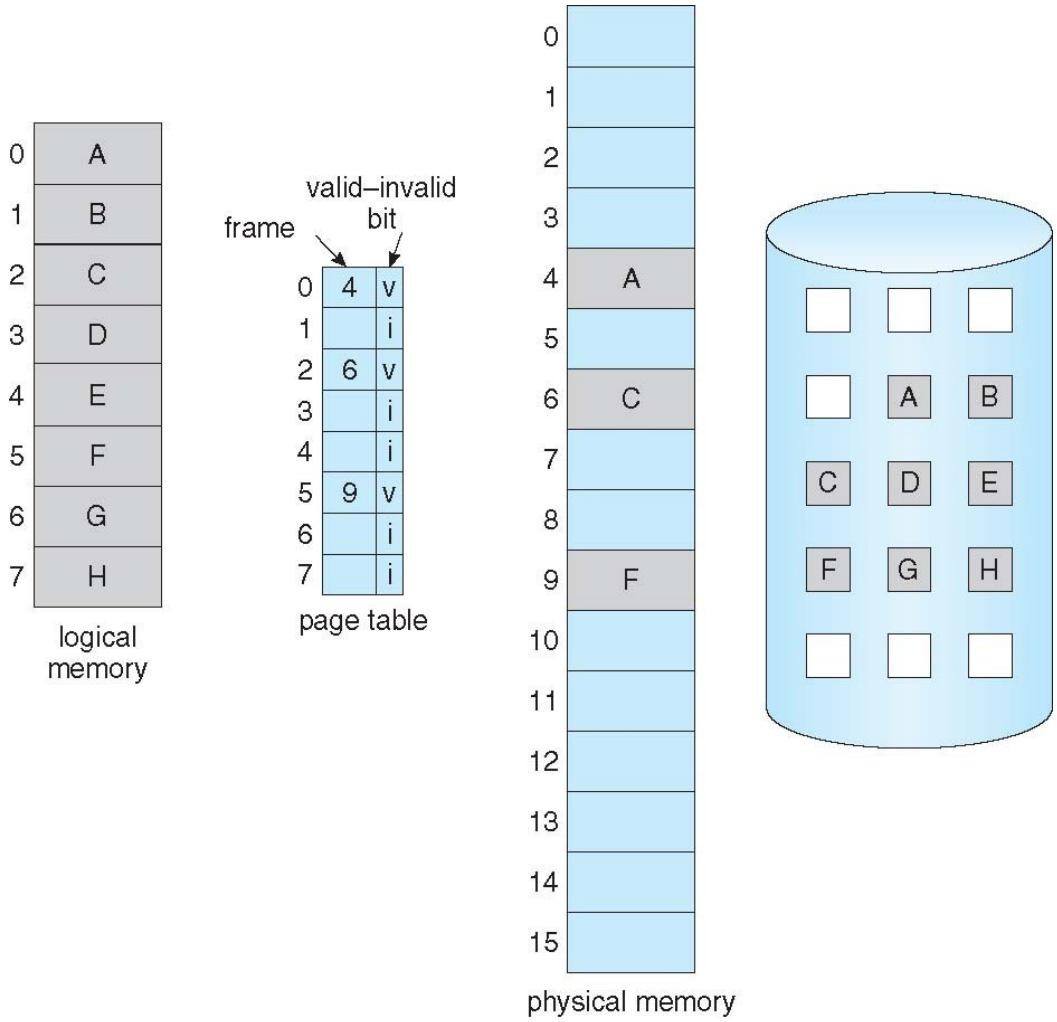
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory, **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



Page Fault

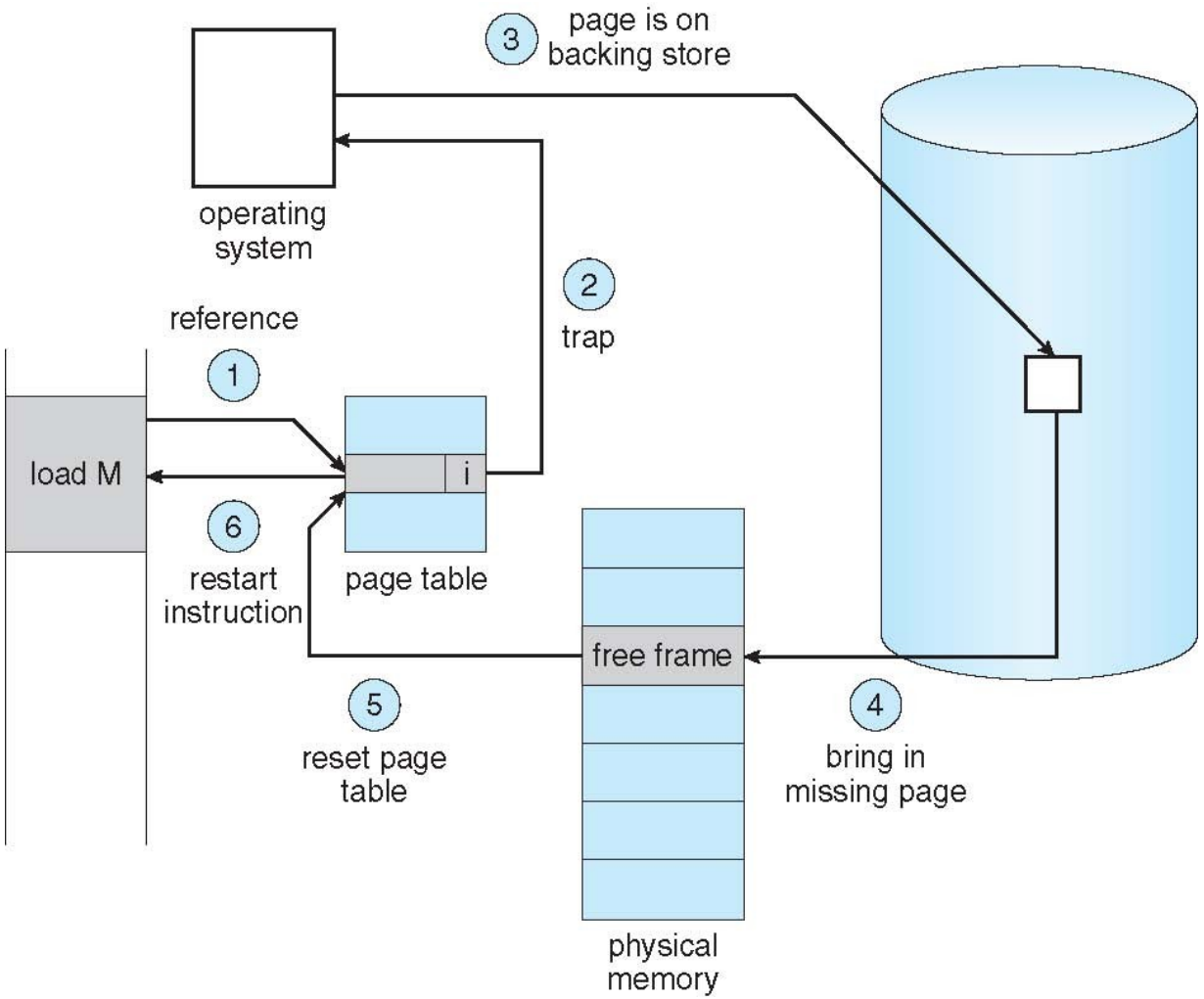
If there is a reference to a page, first reference to that page will trap to operating system:

page fault

Actions when a page fault occurs

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory. Go to step (2).
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault

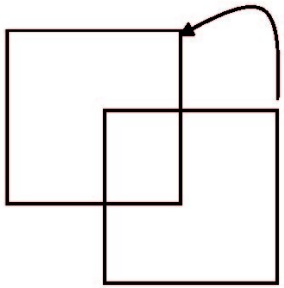


Aspects of Demand Paging

- **Pure demand paging** – start process with *no* pages in memory
  - OS sets instruction-pointer to the first instruction of the process, non-memory-resident -> page fault
  - And for every other process pages on first access
- 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
  - The two numbers may reside in two different pages
- 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
  - Page fault during the move
  - Restart the whole operation?
- What if source and destination overlap?



  - Several solution. Simplest one is to figure out all the pages that are needed to execute the instruction to completion and ensure that these pages are in memory before the instruction starts executing.

Stages in 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 I/O 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

Stages in Demand Paging (Cont.)

Stages in Demand Paging (Cont.)

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 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)

  EAT = (1 – p) x memory access

  \+ p (page fault overhead

  \+ swap page out

  \+ swap page in)

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

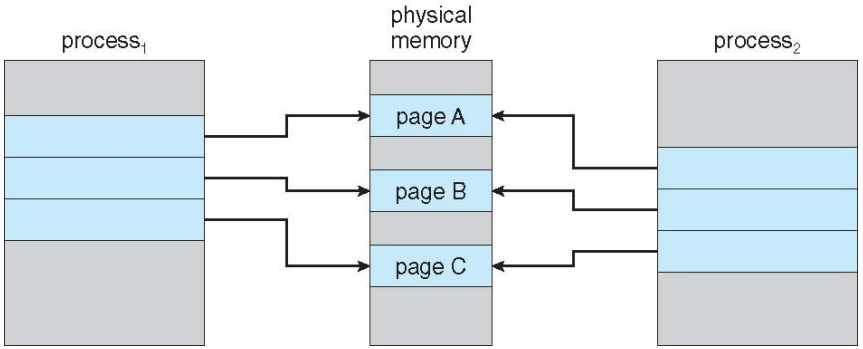
- Typically don't support swapping
- Instead, demand page from file system and reclaim read-only pages (such as code)

Copy-on-Write

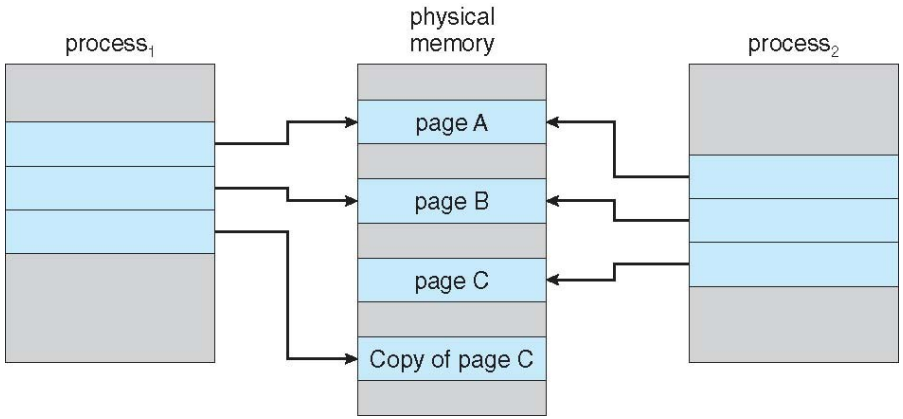
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated 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

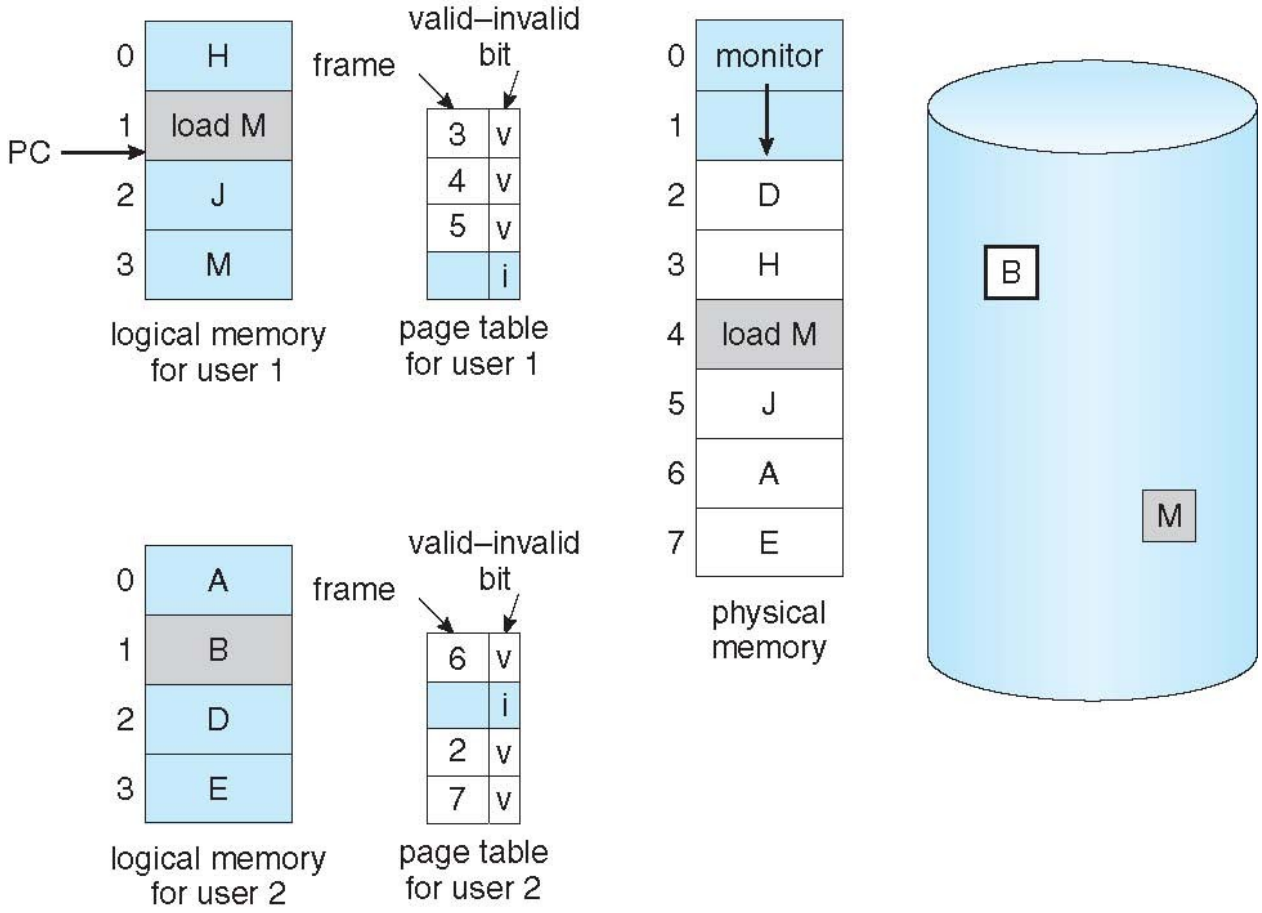


Page Replacement

- 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



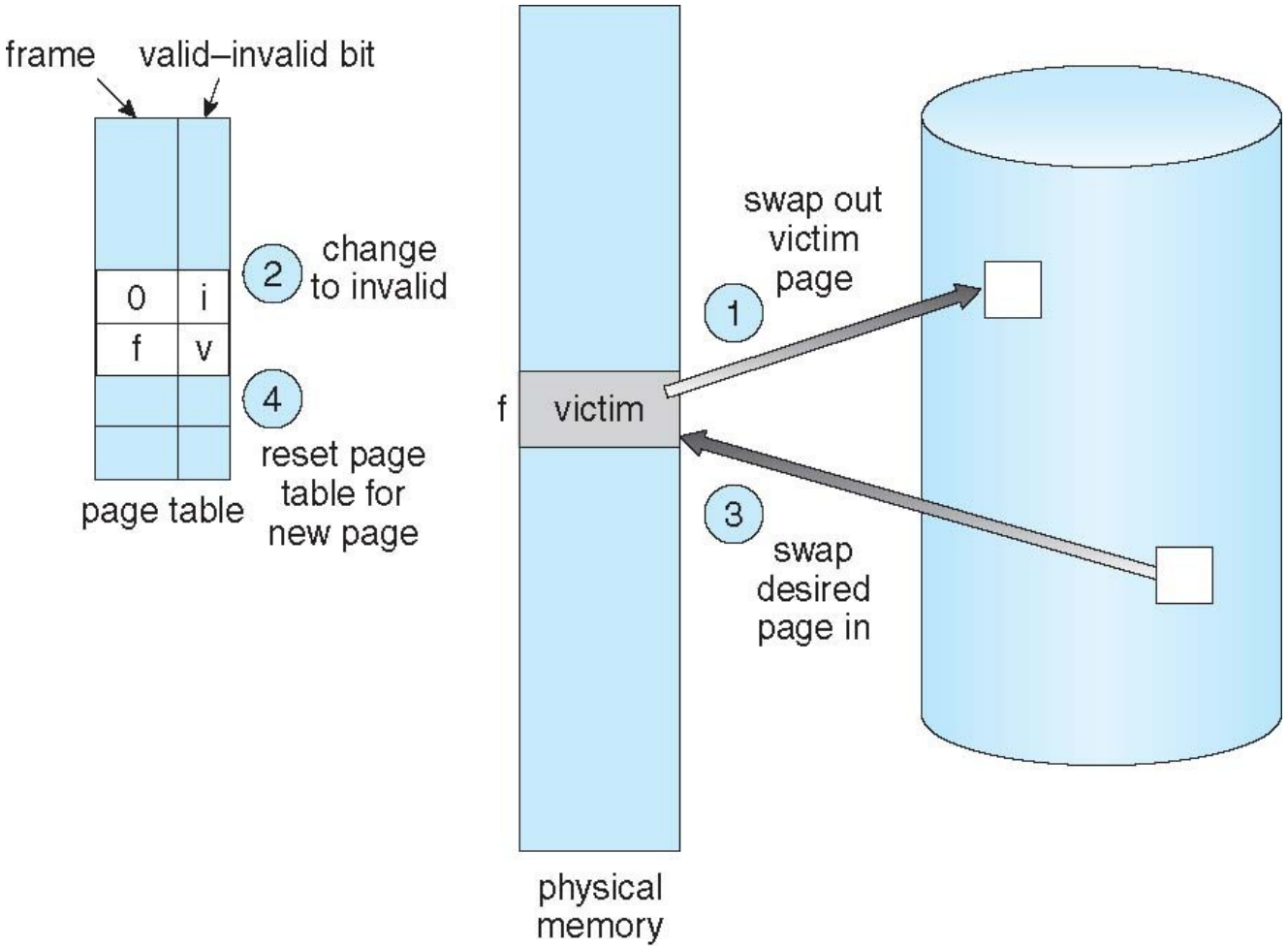
Page Replacement (Cont.)

- 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

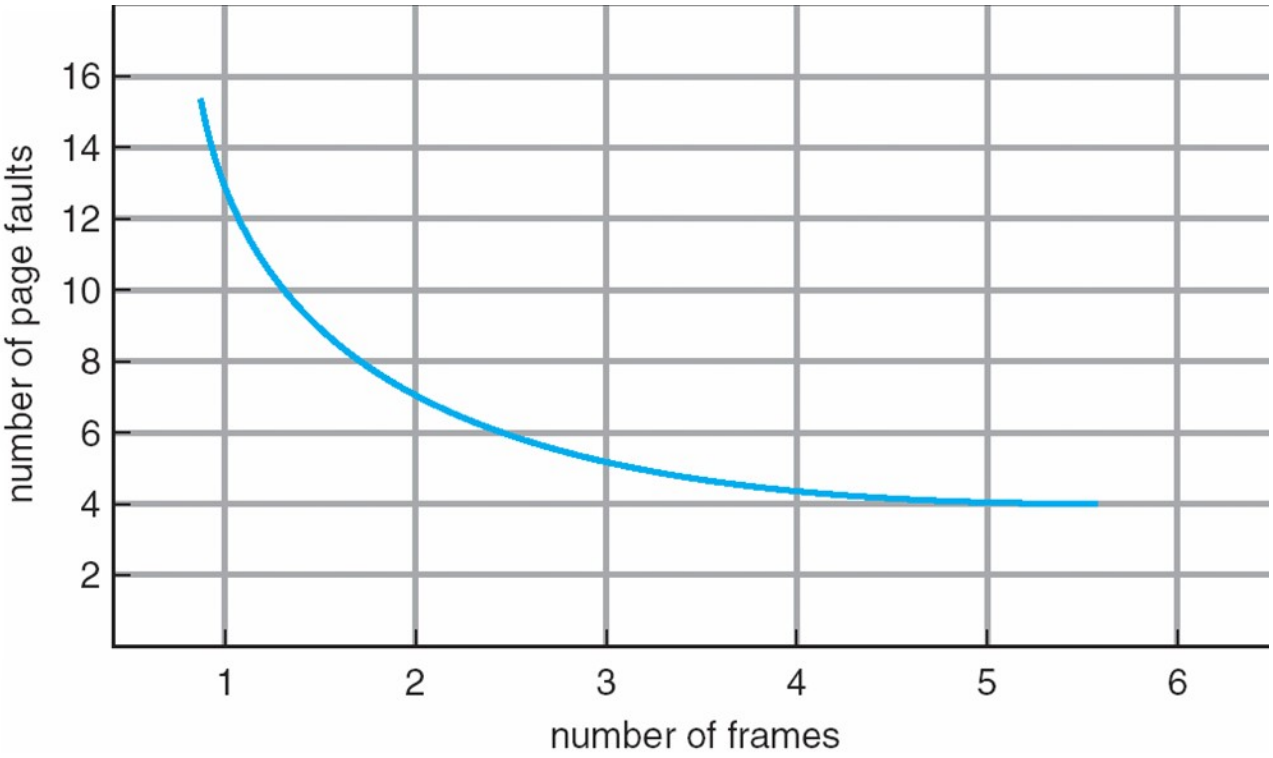


Page and Frame Replacement Algorithms

- **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

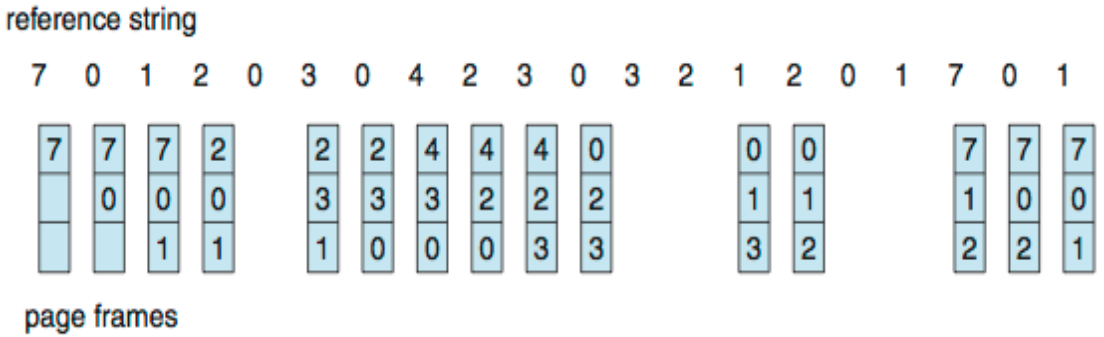
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

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)



15 page faults

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

Number of Frames vs Page Faults

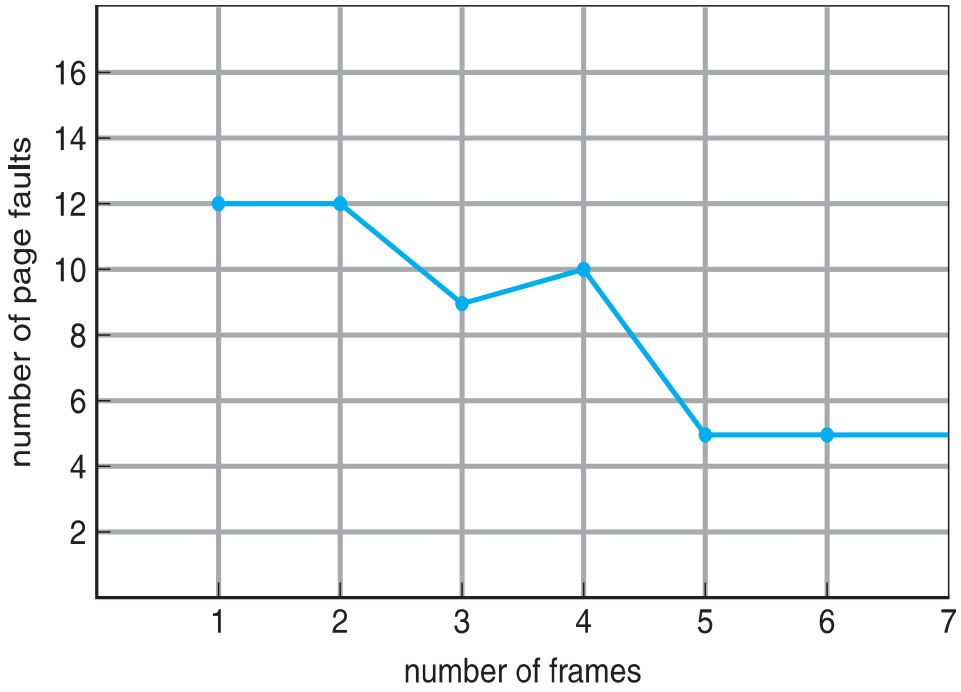
- One would expect that the more frames are allocated to a process the fewer page faults
- Consider the reference string:

  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?

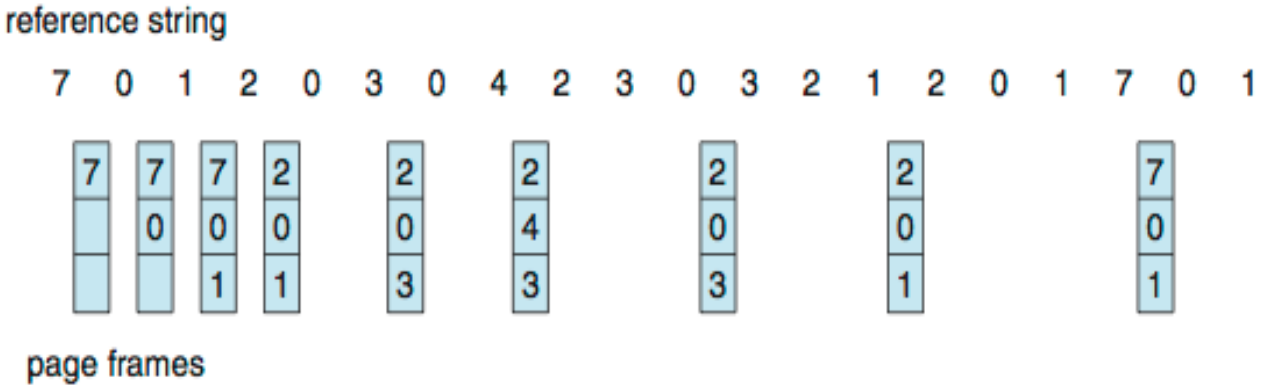
Belady's Anomaly

- 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

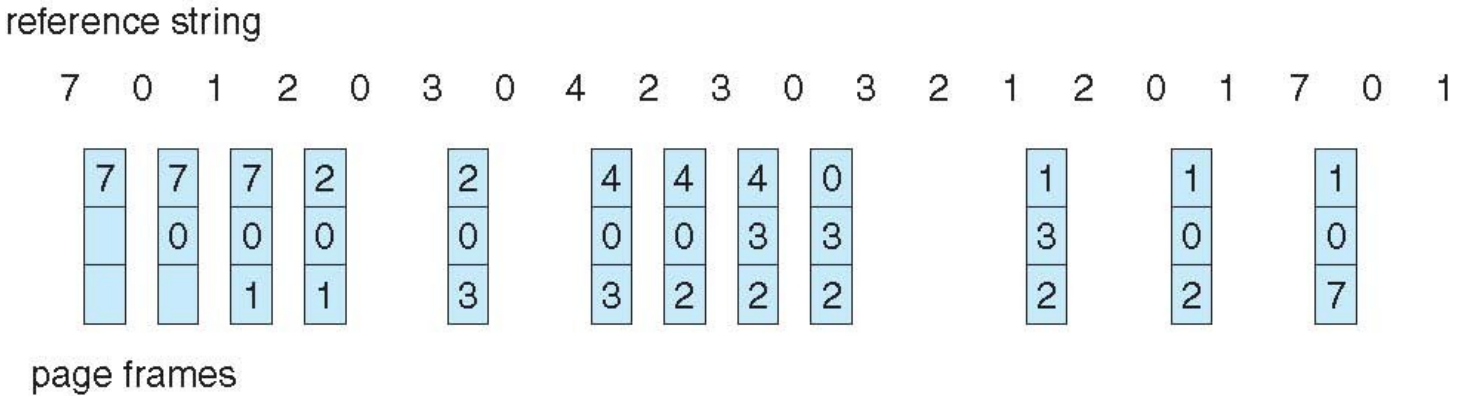
- Replace page that will not be used for longest period of time
  - With 3 frame, 9 is optimal for the example reference.



- 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



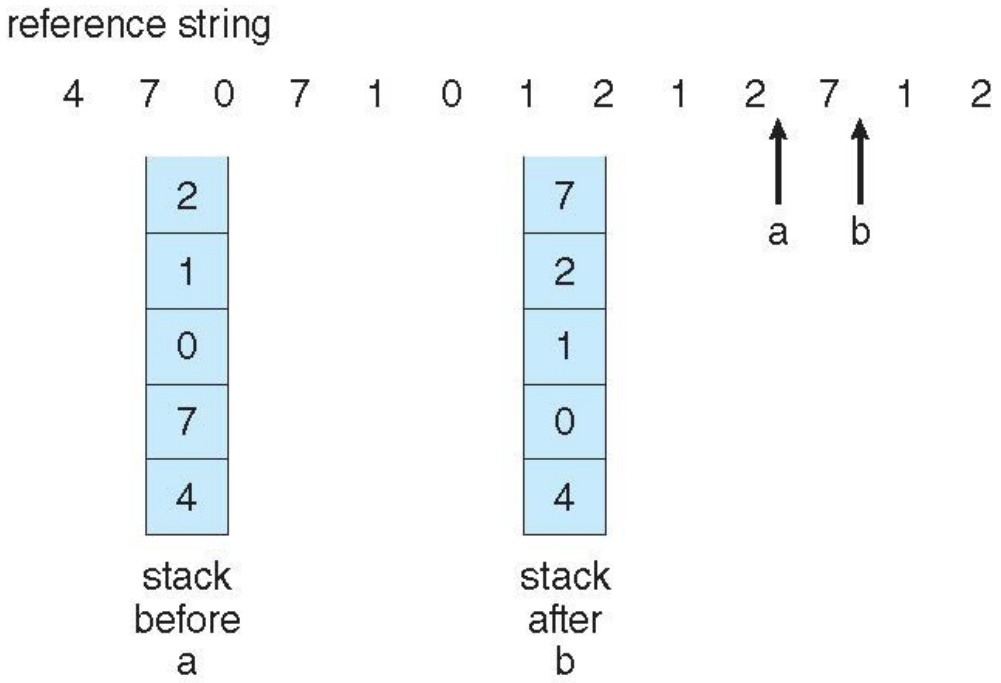
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how do we implement?

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** that don't suffer from Belady's Anomaly

Stack Algorithm Example

Use of a stack to record the most recent page references

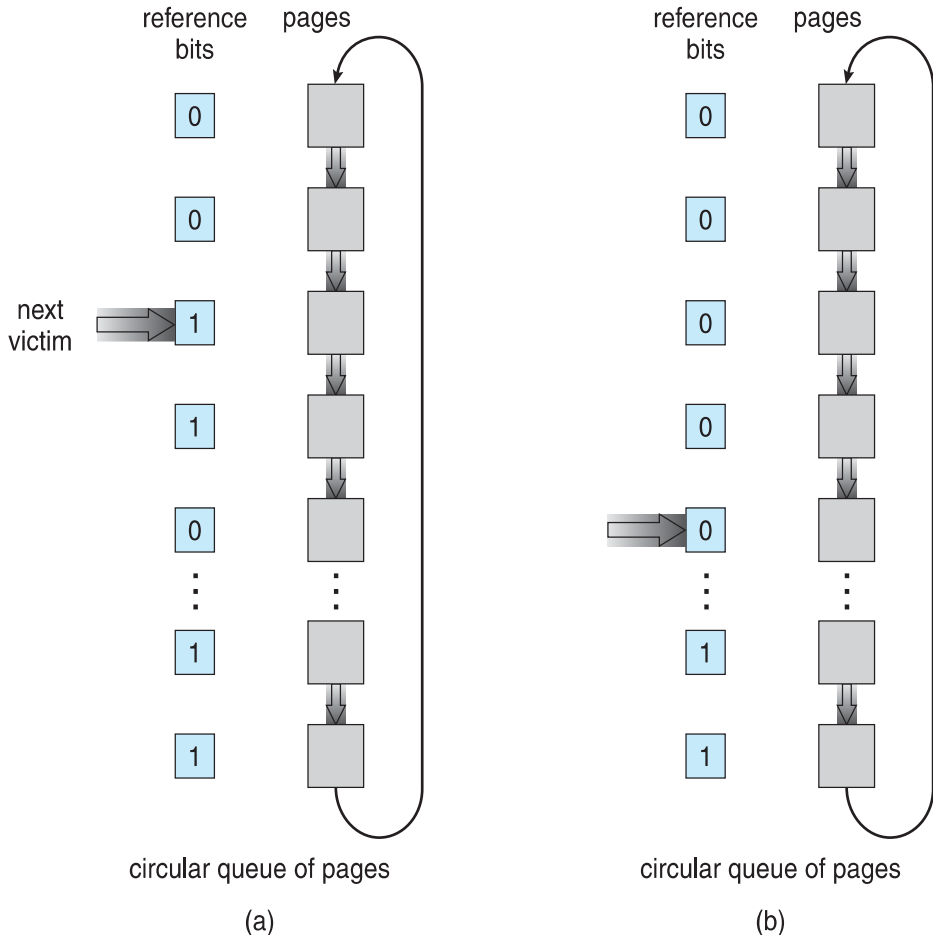


LRU Approximation Algorithms

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



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 = size of process p_i
    - $S = \sum s_i$
    - m = total number of frames
    - a_i = 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?

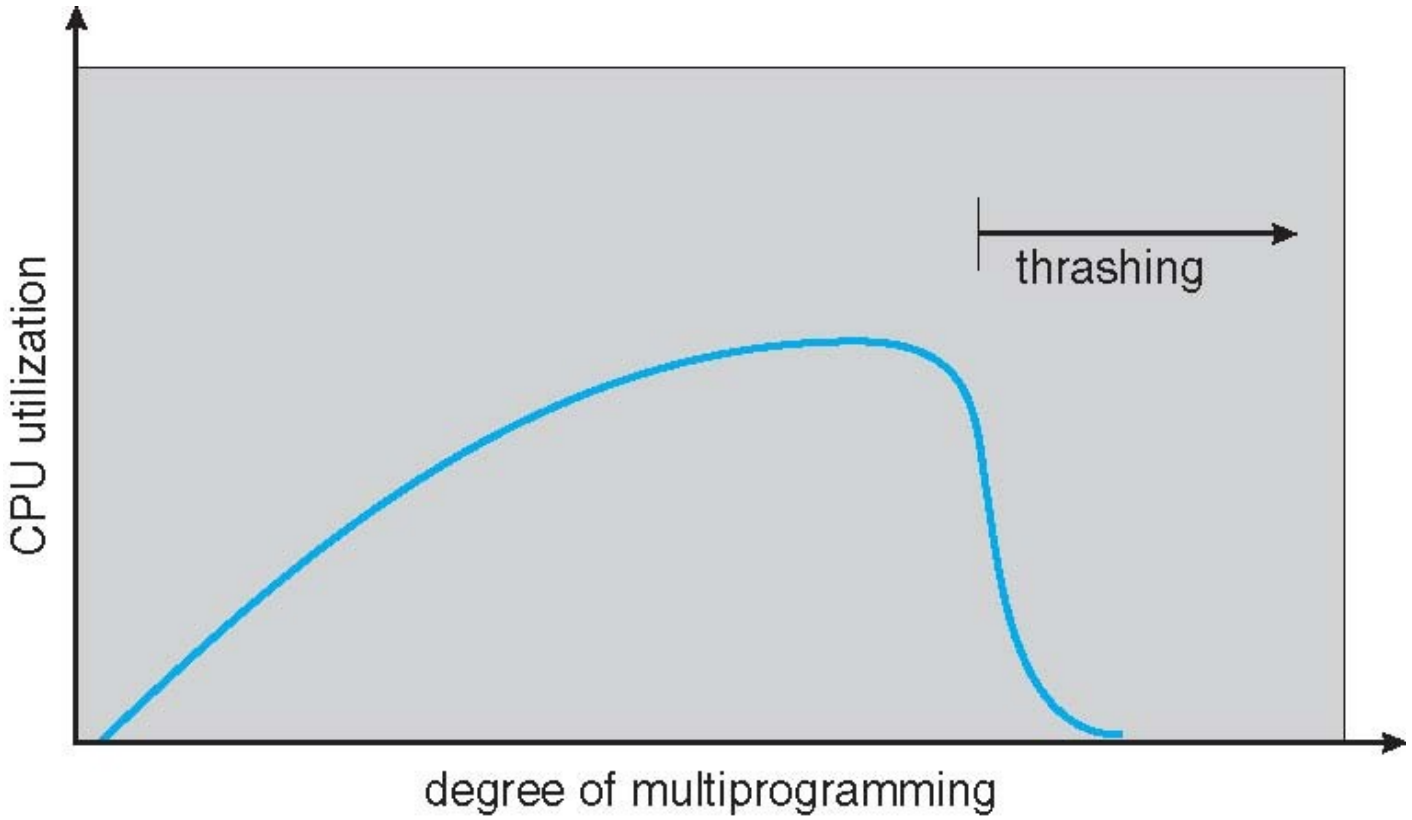
Global vs. Local Allocation

- **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.)



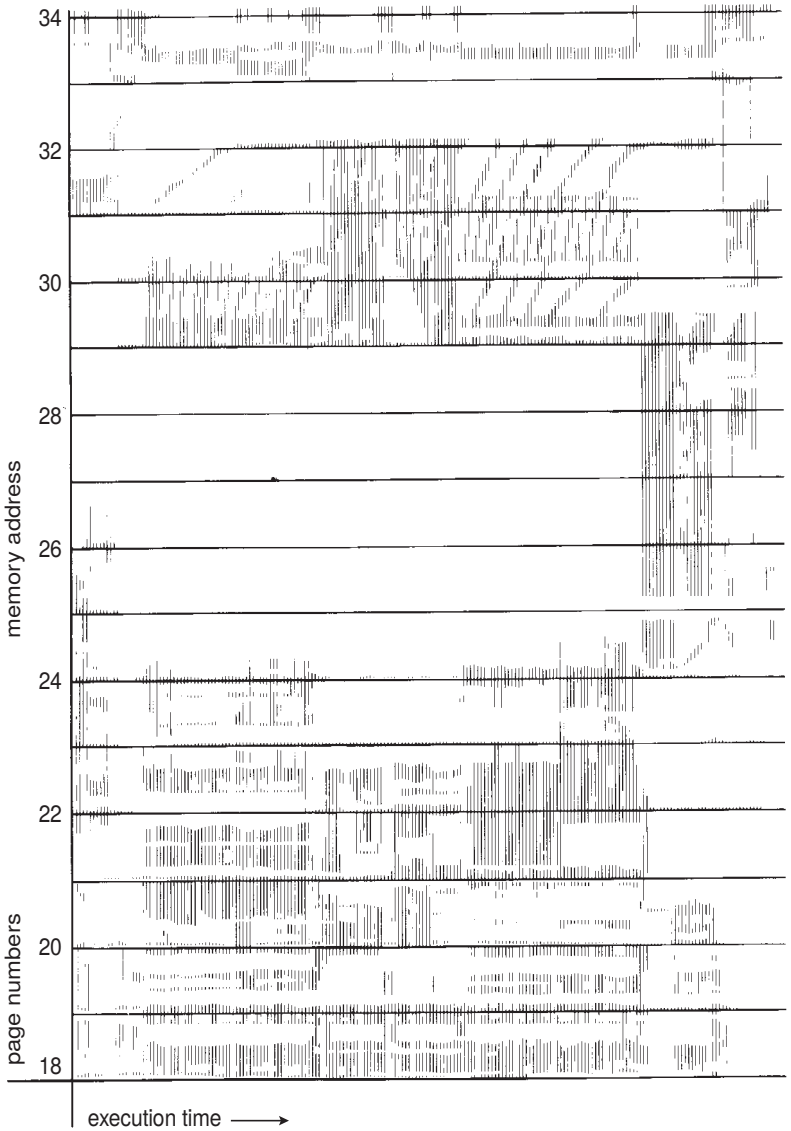
Demand Paging and Thrashing

- 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

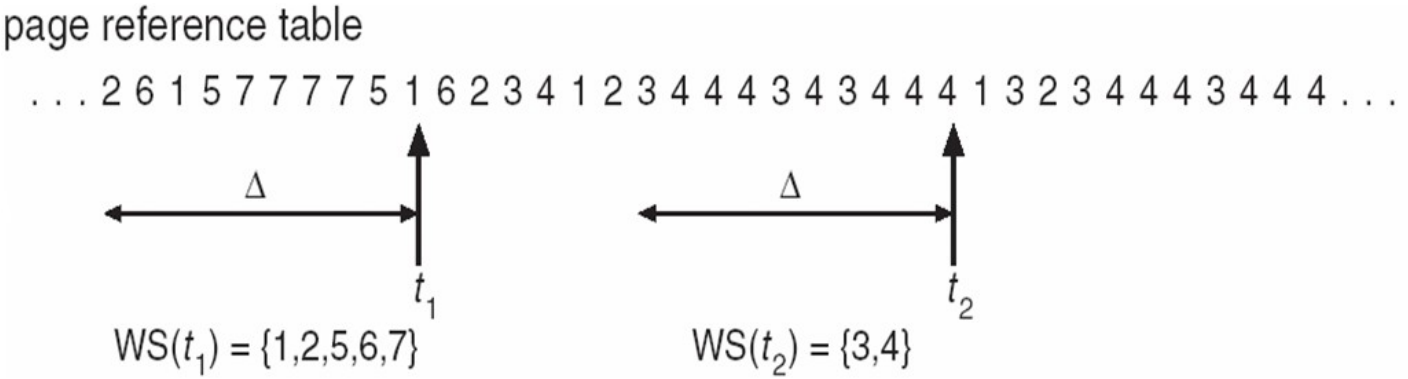
  - Can limit the effects of thrashing by using:
    - Local page replacement
    - Priority page replacement – replace a page from a process with the lowest priority.

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 $\Delta = 10$



Working-Set Model (Cont.)

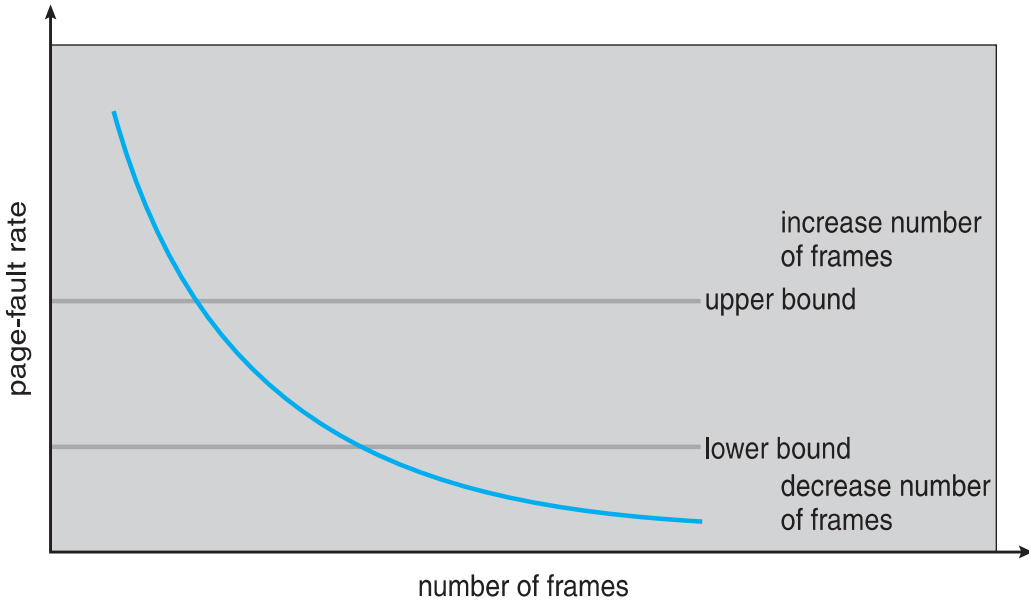
- WSS_i -- tries to approximate the size of the locality of process P_i
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ approximate the total demand 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: Δ = 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

Other Considerations -- Prepaging

- Prepaging used 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 that:
  - *s* pages are prepaged and
  - A fraction α of these *s* pages is actually used (α between 0 and 1)
  - Question -- is the cost of $s * \alpha$ saved pages faults greater or less than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages?
    - If α near zero ⇒ prepaging loses
    - If α near one ⇒ prepaging wins

Other Issues – 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

Other Issues – 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

Other Issues – Program Structure

- Program structure
  - `int[128,128] data;`
  - 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

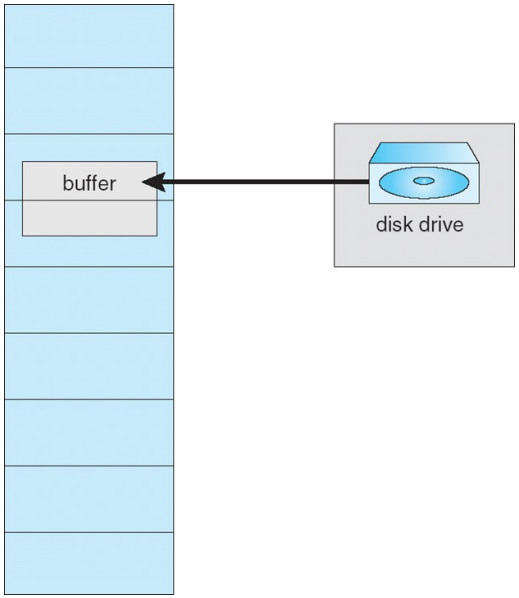
  - Program 2

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

128 page faults

Other Issues – 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



Operating System Examples

- Windows
- Solaris

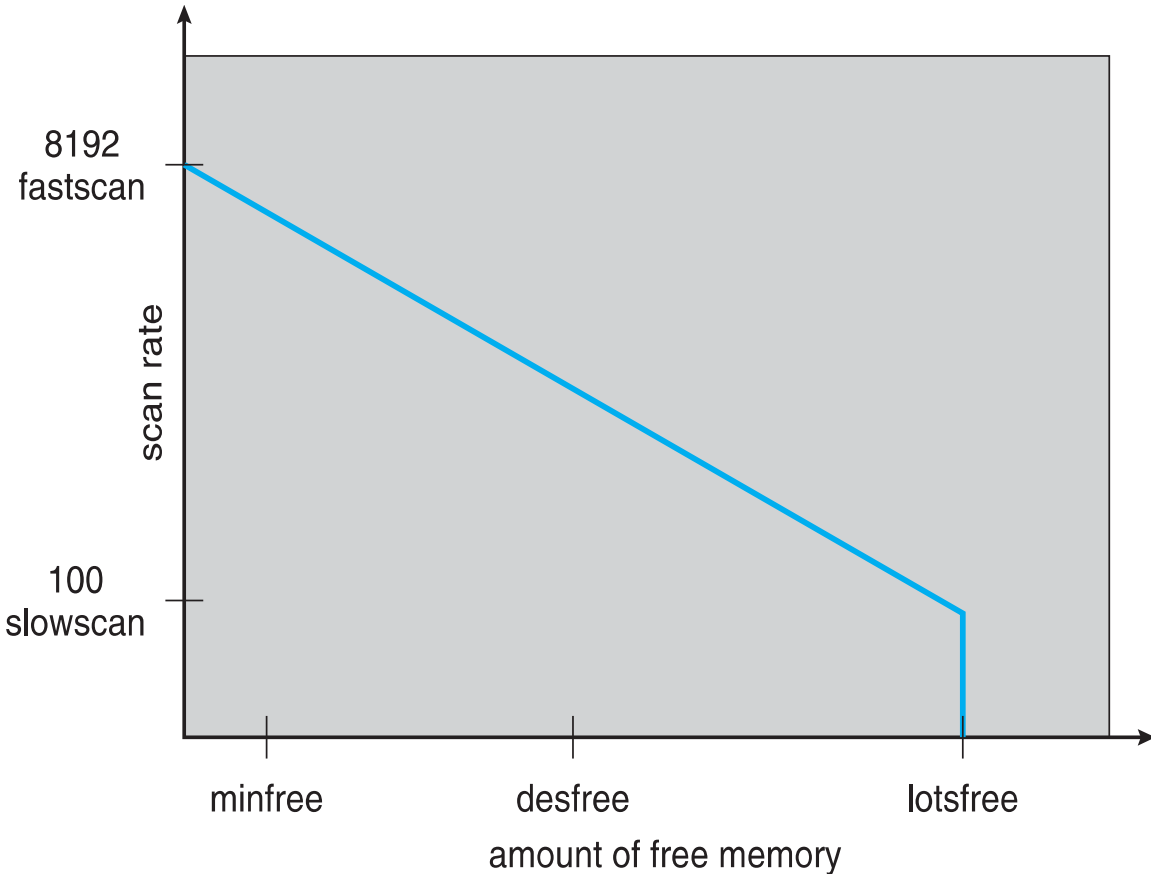
Windows

- 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