Chapter 9: Virtual Memory

Chapter 9: Virtual Memory

- Background
- Demand Paging
  - Performance of demand paging
- Page Replacement
  - Page replacement algorithms
- Allocation of Frames
- Thrashing
- Other Considerations

Background and Motivation

- Code needs to be in memory to execute, but entire program rarely used
  - Example: Error code, unusual routines, large data structures
  - Entire program code is not needed at the same time

- Consider advantages of the ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running
    - Thus, more programs can 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
    - Thus each user program runs faster

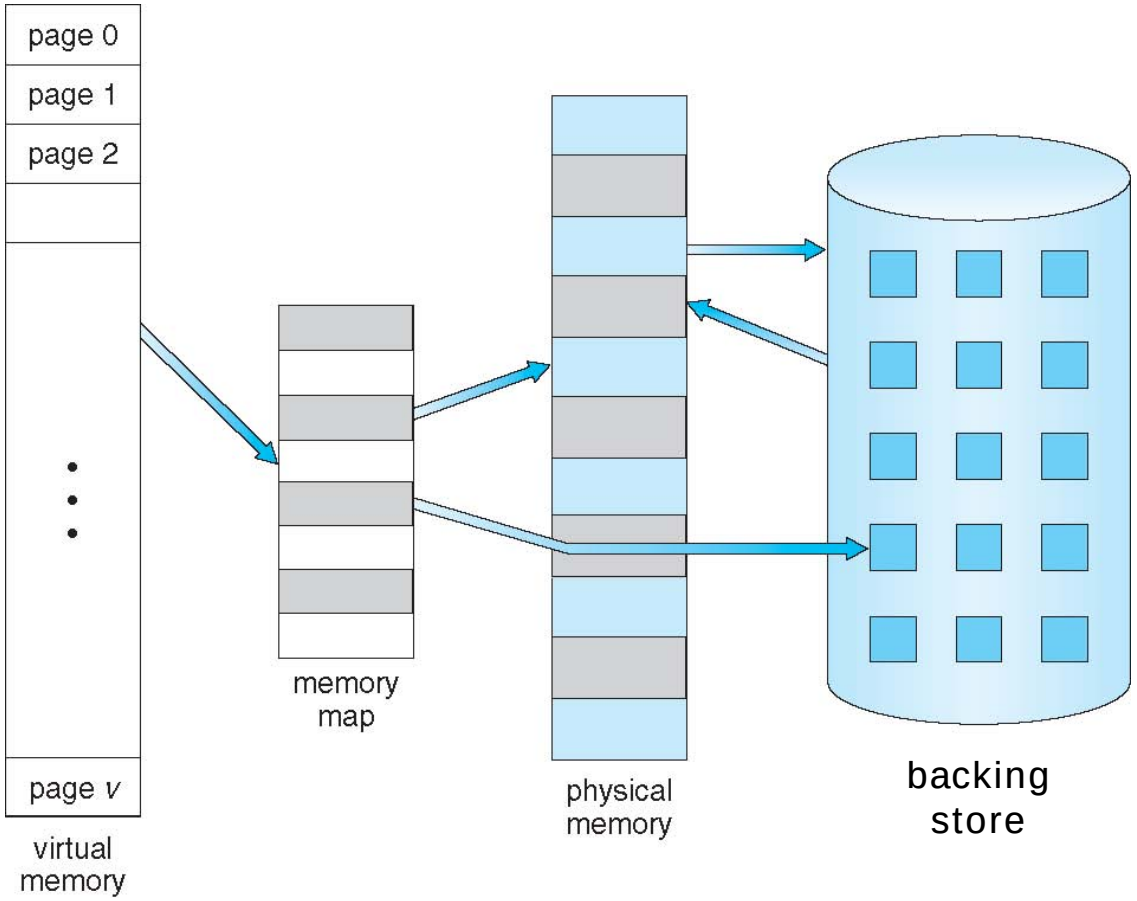
Background (Cont.)

- **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 makes the task of programming much easier
    - the programmer no longer needs to worry about the amount of physical memory available;
    - can concentrate instead on the problem to be programmed.

Background (Cont.)

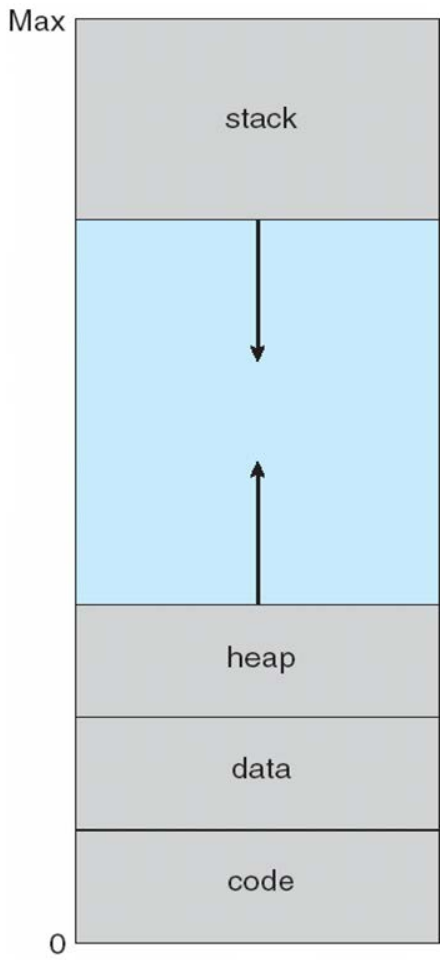
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging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

Virtual Memory That is Larger Than Physical Memory

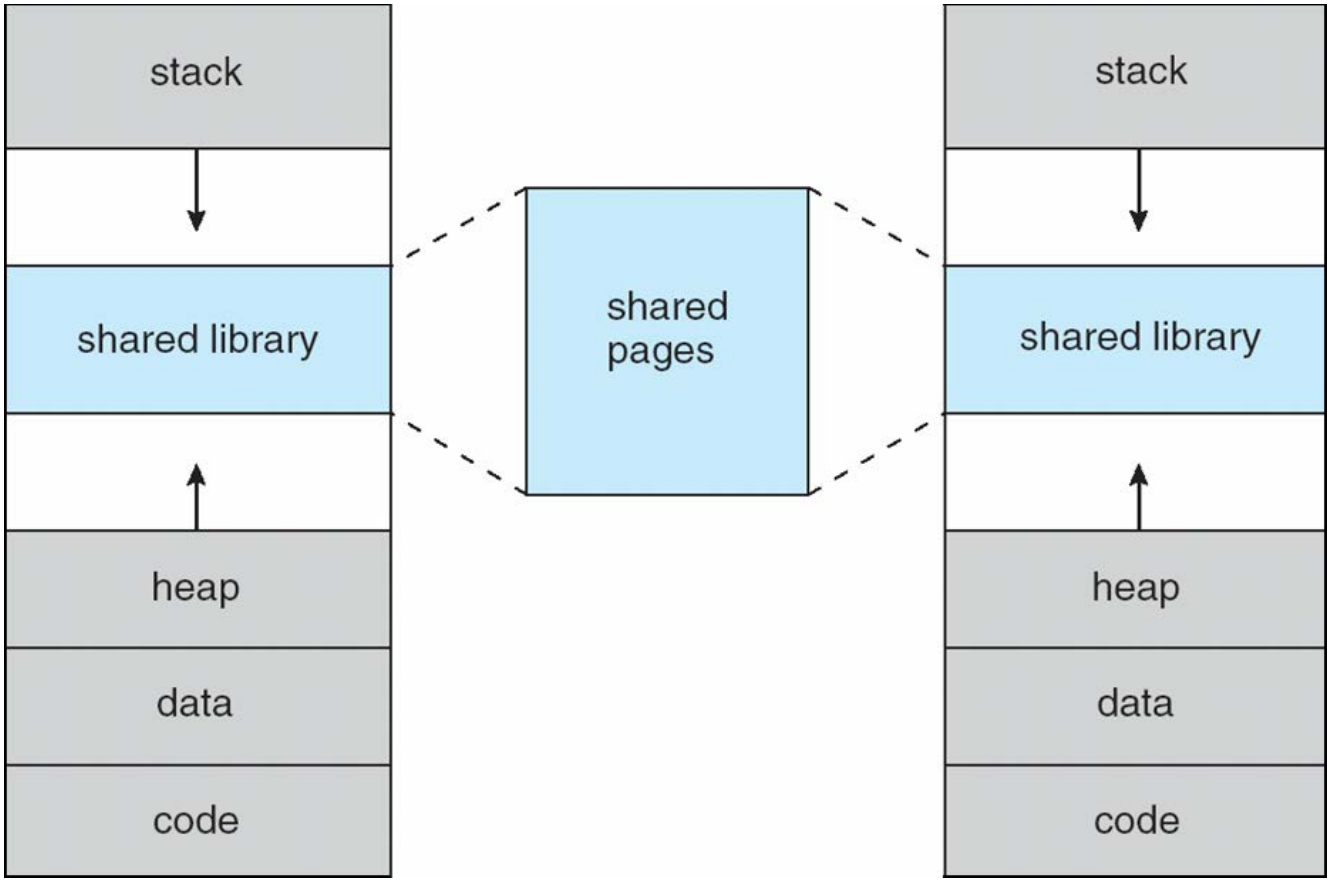


Virtual-address Space

- 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 into virtual address space



Shared Library Using Virtual Memory



Policies for Paging/ Segmentation

- Fetch Strategies
  - When should a page or segment be brought into primary memory from secondary (disk) storage?
    - Demand Fetch
    - Anticipatory Fetch
- Placement Strategies
  - When a page or segment is brought into memory, where is it to be put?
    - Paging - trivial
    - Segmentation - significant problem
- Replacement Strategies
  - Which page/segment should be replaced if there is not enough room for a required page/segment?

Demand Paging

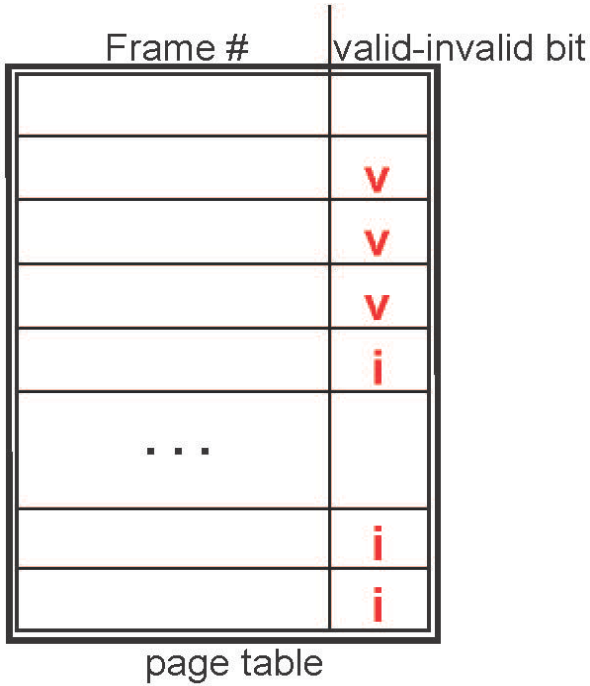
- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less Memory needed
  - Faster response
  - More users
- Demand paging is kind of a “lazy” swapping
  - But deals with pages instead of programs
    - swapping deals with entire programs, not pages
  - **Lazy swapper** – never swaps a page into memory unless page will be needed
  - **Pager --** swapper that deals with pages
- The first reference to a page will trap to OS with a page fault.
- OS looks at another table to decide
  - Invalid reference - abort
  - Just not in memory – bring it from memory

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 only those pages into memory.
- Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.
- 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

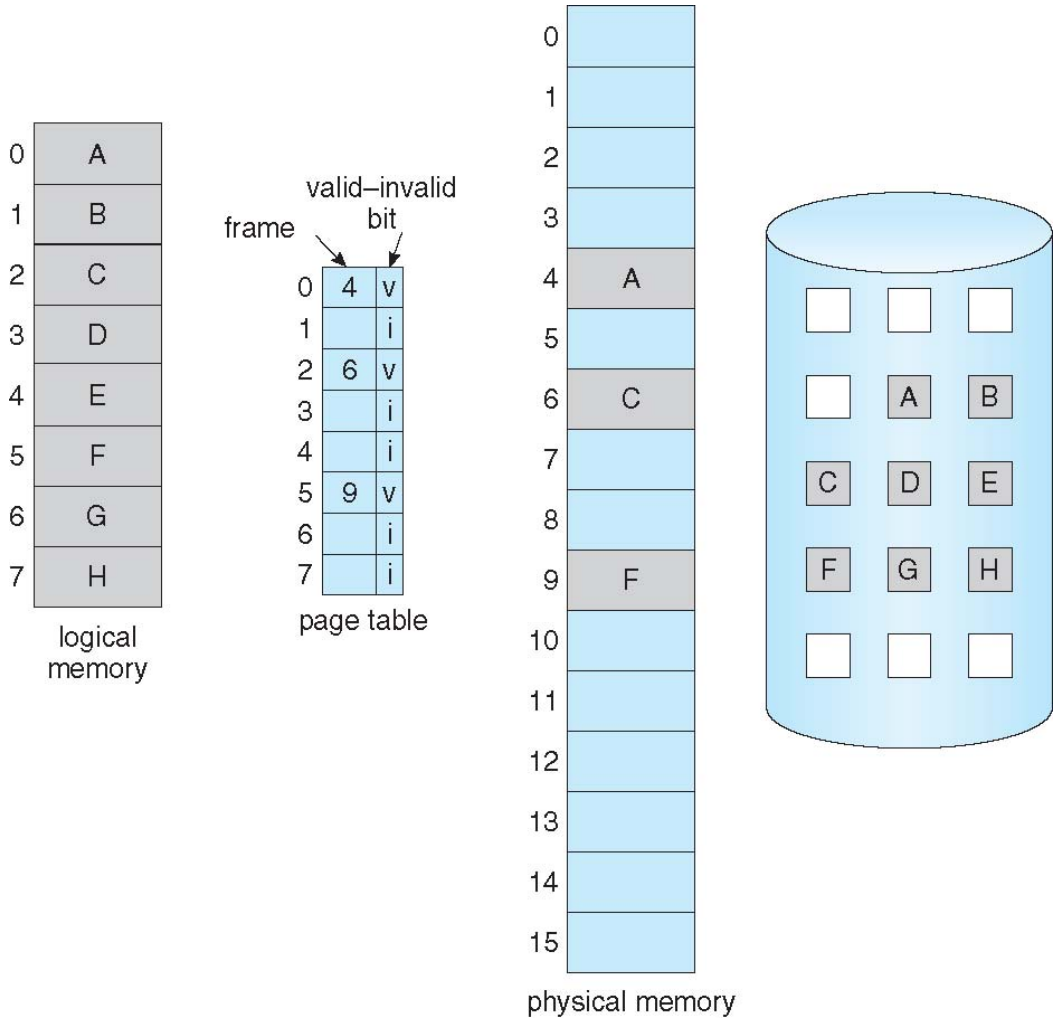
Valid-Invalid Bit

- 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 =**i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



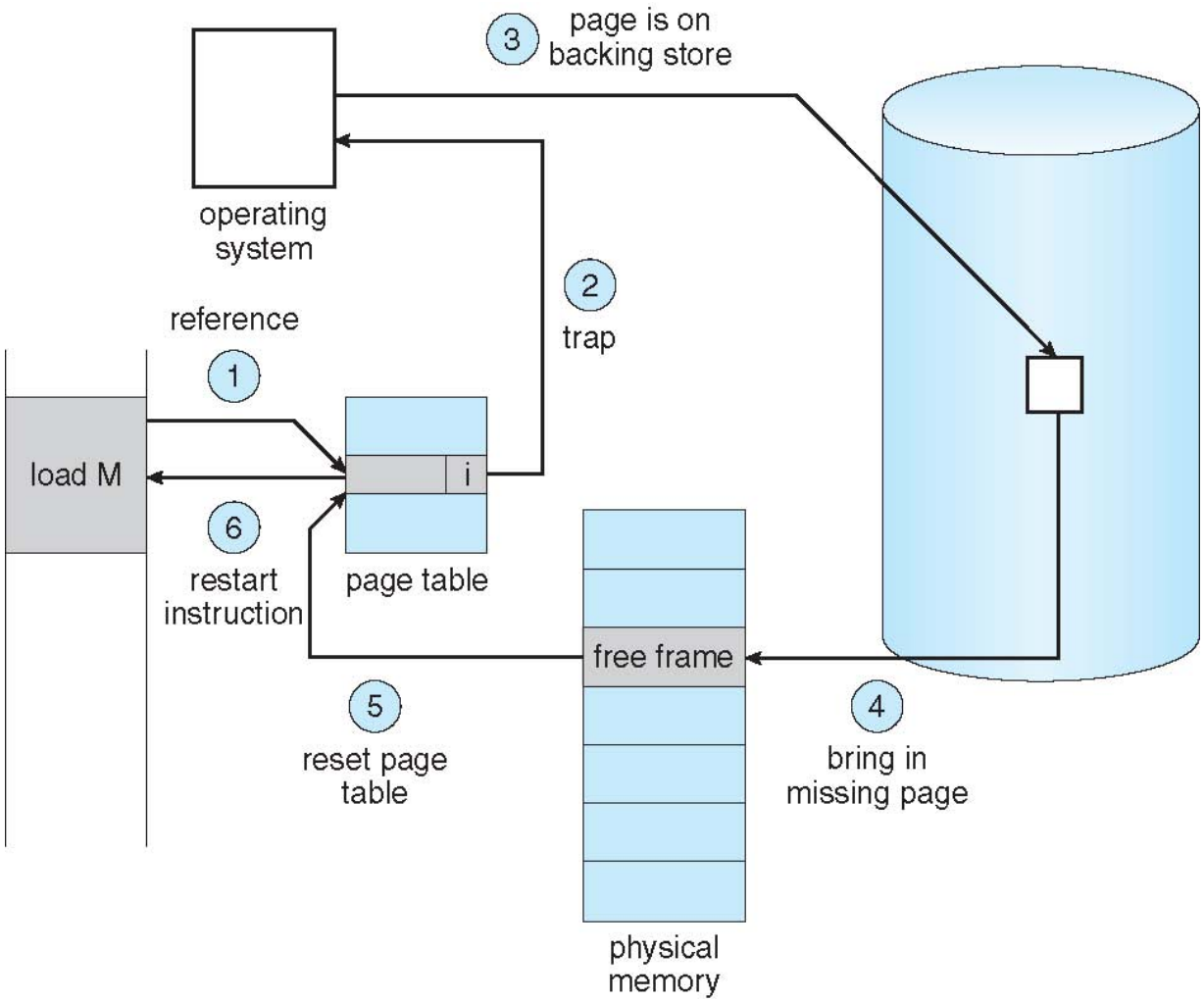
Page Fault

- If the process tries to access a page that was not brought into memory,
- Or tries to access any “invalid” page:
  - That will trap to OS, causing a **page fault**
  - Such as when the first reference to a page is made

Handling a page fault

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

Steps in Handling a Page Fault



What happens if there is no free frame?

- **Page replacement** if not free frame
  - Find some page in memory that is not really in use and swap it.
    - Need a page replacement algorithm
    - Performance Issue - need an algorithm which will result in minimum number of page faults.
  - Same page may be brought into memory many times.

Aspects of Demand Paging

- **Pure 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
    - page faults for all other pages on their first access
- A given instruction could access multiple pages
  - Causing multiple page faults
    - E.g., 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

Performance of Demand Paging

- Stages in Demand Paging (worst 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

Performance of Demand Paging (Cont.)

- Three major activities during a Page Fault
  1. **Service the page fault interrupt**
     - Fast - just several hundred instructions needed
  2. **Read in the page.**
     - Very slow - about 8 millisecs., including hardware and software time
     - 1 millisec = 10^{-3} sec, 1 microsec = 10^{-6} sec, 1 ns = 10^{-9} sec
  3. **Restart the process**
     - Fast – similar to (1)

- 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) * memory access +

    \+ p * (page fault overhead + swap page out + restart overhead)

Demand Paging Example

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

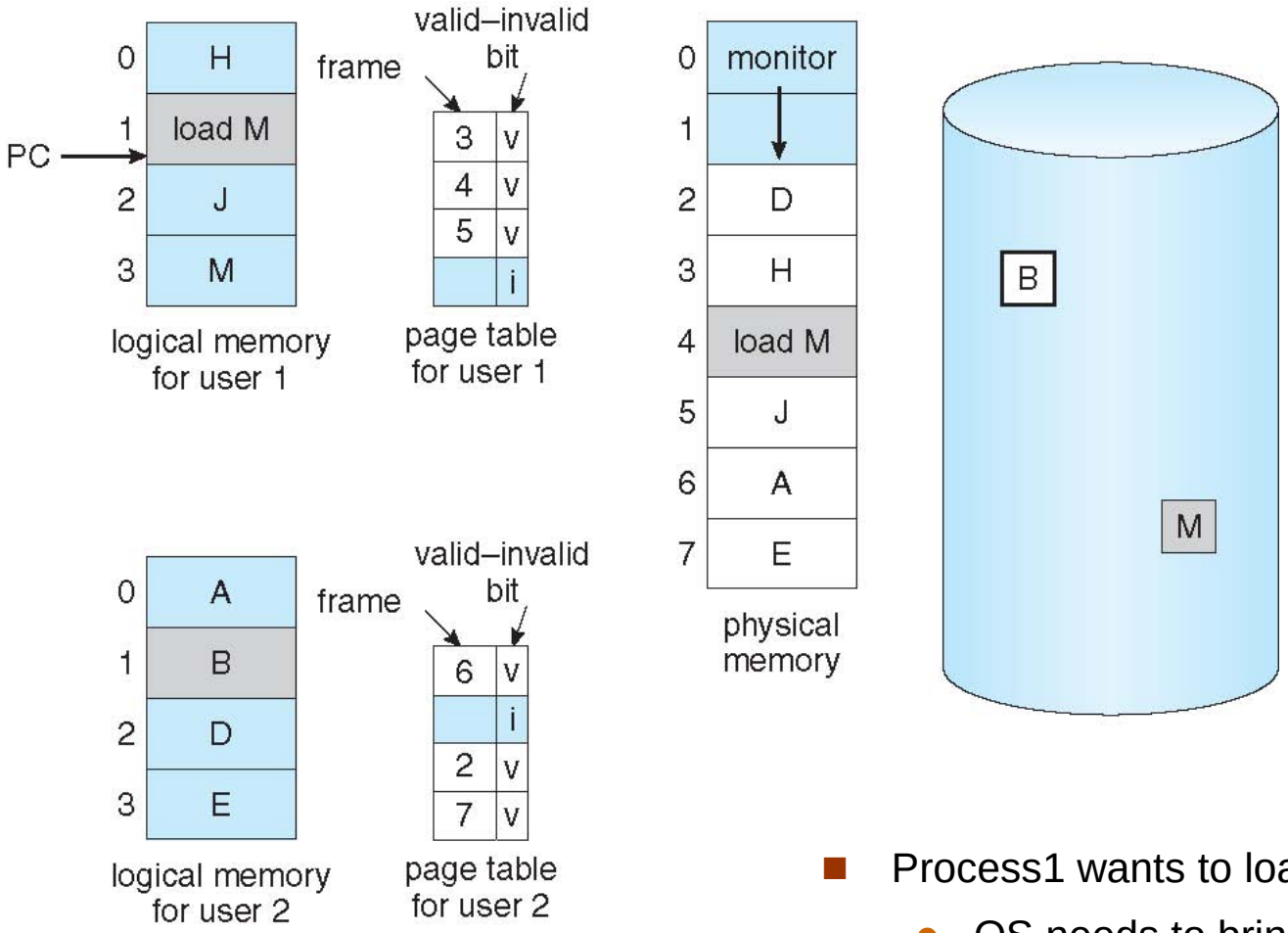
  = (1 – p) * 200 + p * 8,000,000

  = 200 + p * 7,999,800
- If one access out of 1,000 causes a page fault, then
  - EAT = 8.2 microseconds (10^{-6} of a second).
  - This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 * p
    20 > 7,999,800 x p
  - p < .0000025
  - Less than 1 page fault in every 400,000 memory accesses

Page Replacement

- **Over-allocation**
  - While a user process is executing, a page fault occurs.
  - OS determines where the desired page is residing on the disk
    - but then finds that there are no free frames on the free-frame list;
    - all memory is in use
- We need to prevent **over-allocation** of memory
  - by modifying page-fault service routine to include page replacement
- **Page replacement**
  - If no frame is free, we find one that is not currently being used and free it.
    - as explained next
  - completes separation between logical memory and physical memory
  - large virtual memory can be provided on a smaller physical memory
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers
  - The dirty bit is set for a page when it is modified
  - Only modified pages are written to disk
    - No need to save unmodified pages, they are the same

Need For Page Replacement



- Process1 wants to load memory M
  - OS needs to bring M into physical memory
- **Problem:** no free frames
  - OS needs to do page replacement

Basic Page Replacement

1. Find the location of the desired page on disk

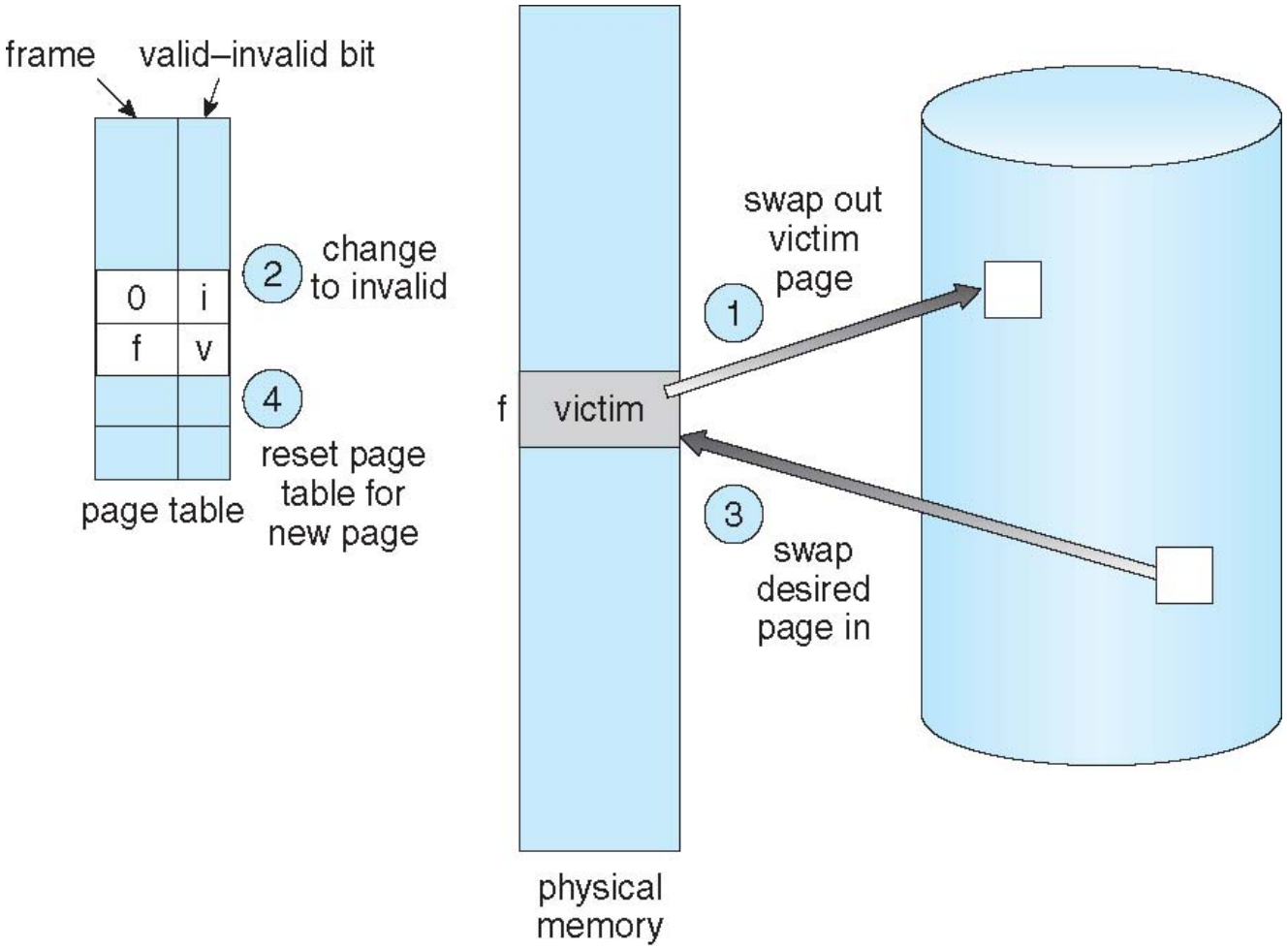
2. Find a free frame:
   - **if** (a free frame exist) **then** use it
   - **else**
     - use a page replacement algorithm to select a **victim frame**
     - write victim frame to disk, if dirty

3. Bring the desired page into the (newly) free frame;
   - update the page and frame tables accordingly

4. Continue the process by restarting the instruction that caused the trap

- Note: now potentially 2 page transfers for page fault
  - Increasing EAT

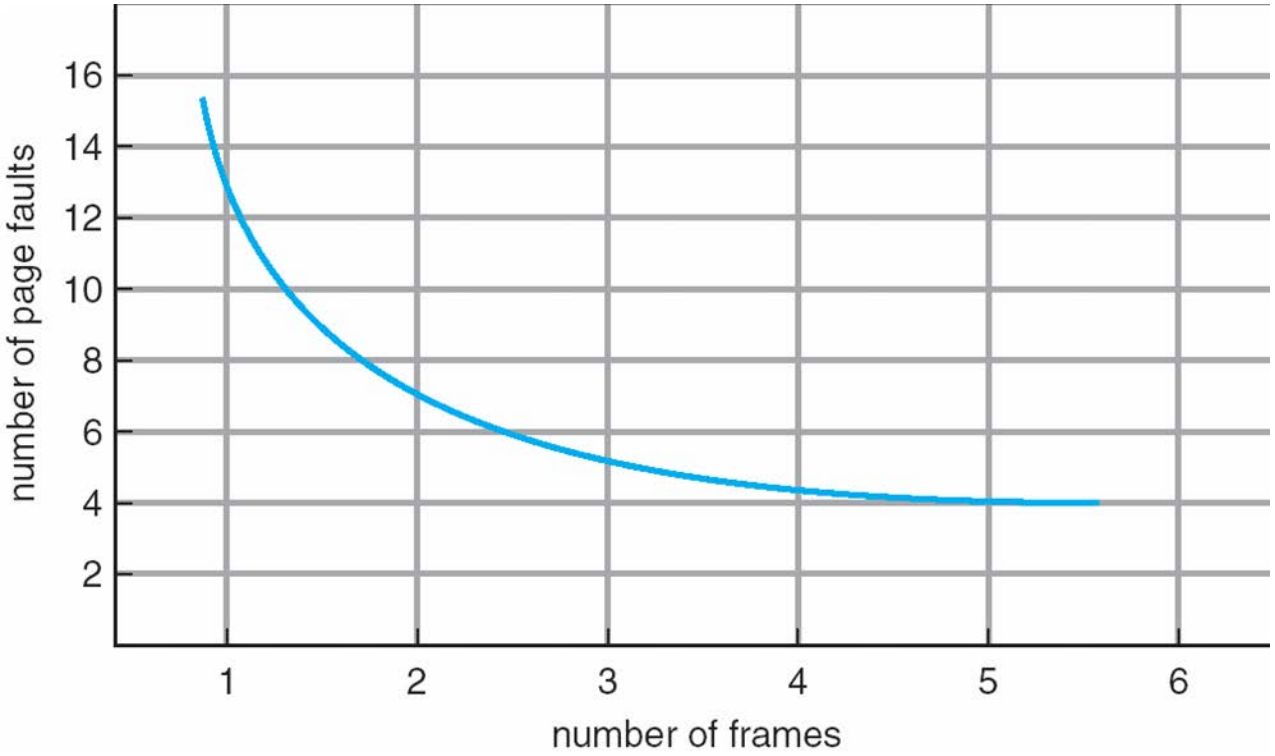
Page Replacement



Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want the lowest page-fault rate on both first access and re-access
- Evaluating different page replacement algorithms
  - By running them 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 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

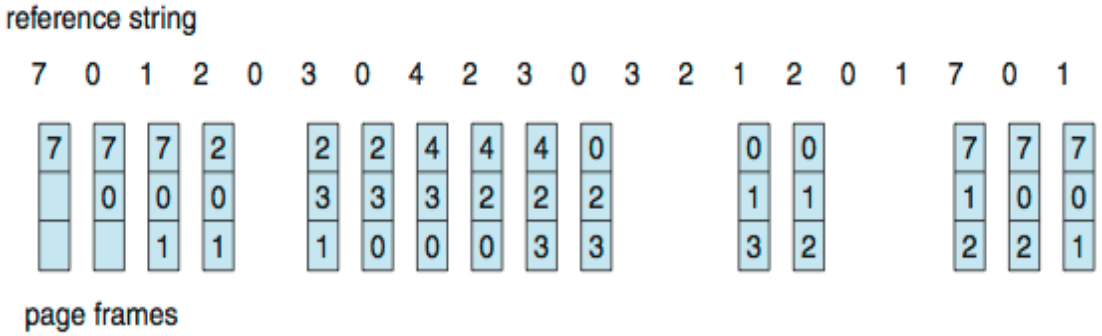


Page Replacement Strategies

- The Principle of Optimality
  - Replace the page that will not be used again the farthest time into the future.
- Random Page Replacement
  - Choose a page randomly
- FIFO - First in First Out
  - Replace the page that has been in memory the longest.
- LRU - Least Recently Used
  - Replace the page that has not been used for the longest time.
- LFU - Least Frequently Used
  - Replace the page that is used least often.
- NUR - Not Used Recently
  - An approximation to LRU
- Working Set
  - Keep in memory those pages that the process is actively using

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
- FIFO algorithm
  - Easy to implement
  - Often, not best performing

FIFO Illustrating Belady's Anomaly

- Reference String: 1,2,3,4,1,2,5,1,2,3,4,5
- Assume x frames (x pages can be in memory at a time per process)

3 frames

Frame 1	1	4	5
Frame 2	2	1	3
Frame 3	3	2	4

9 Page faults

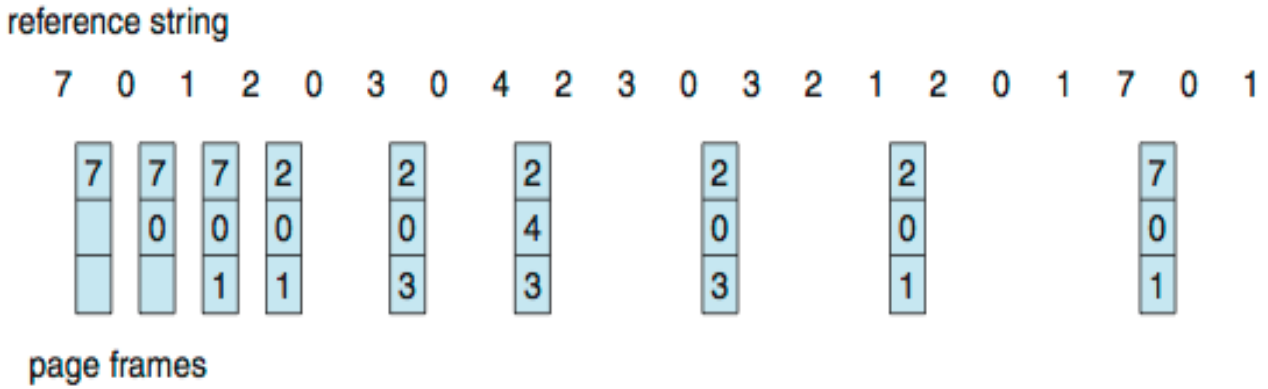
4 frames

Frame 1	1	5	4
Frame 2	2	1	5
Frame 3	3	2	
Frame 4	4	3	

10 Page faults

FIFO Replacement - *Belady's Anomaly* -- more frames does not mean less page faults!

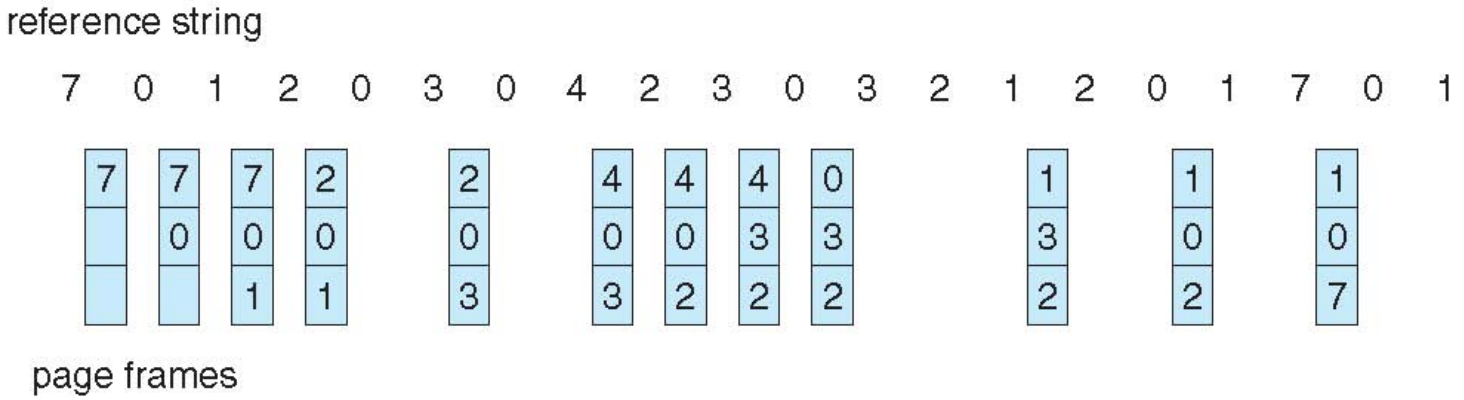
Optimal Algorithm



- Called **OPT** or **MIN**
- Replace page that will not be used for longest period of time
  - 9 page faults is optimal for the example
- How do you know this?
  - You don't, can't read the future
- Used for measuring how well other algorithms performs --
  - against the theoretical optimal solution

Least Recently Used (LRU) Algorithm

- 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



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

Implementation of LRU Algorithm

- **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 replaced,
    - LRU looks at the counters to find the smallest value
    - Search through the table is needed

Implementation of LRU Algorithm

- **Stack implementation**
  - Keep a stack of page numbers
    - Most recently used – at the top
    - Least recently used – at the bottom
    - Implemented as a double link list
      - Easier to remove entries from the middle
      - Head pointer
      - Tail pointer
  - Page referenced
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement
    - But each update is more expensive
- LRU and OPT don't have Belady's Anomaly

Implementation of LRU Algorithm

- **Notice:**
  - **Hardware assistance** is required for the above two LRU implementations
  - Let's see why
- The updating of the clock fields (or stack) must be done for **every** memory reference.
    - … and not only when a page fault happens
- How can we handle this in software?
  - If we were to use an interrupt for every reference (to allow software to update such data structures), it would slow every memory reference by a factor of at least 10
    - Hence slowing every user process by a factor of 10.
    - The overhead is too high.
- **Problem:**
  - Few computer systems provide sufficient hardware support for true LRU.
  - In fact, some systems provide **no** hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used.

LRU Approximation Algorithms

- Many systems provide hardware support only in the form of reference bit
  - Can use it to implement approximate LRU algorithms!

- **Reference bit** – simple hardware support for LRU
  - With each page associate a bit, initially = 0
  - When page is referenced, hardware sets the bit to 1
  - Idea:
    - Replace any page with reference bit = 0 (if one exists)
    - We do not know the order, however -- a crude method
  - This information is the basis for many page-replacement algorithms that approximate LRU replacement.

LRU Approximation Algorithms

Additional Reference Bits Algorithm

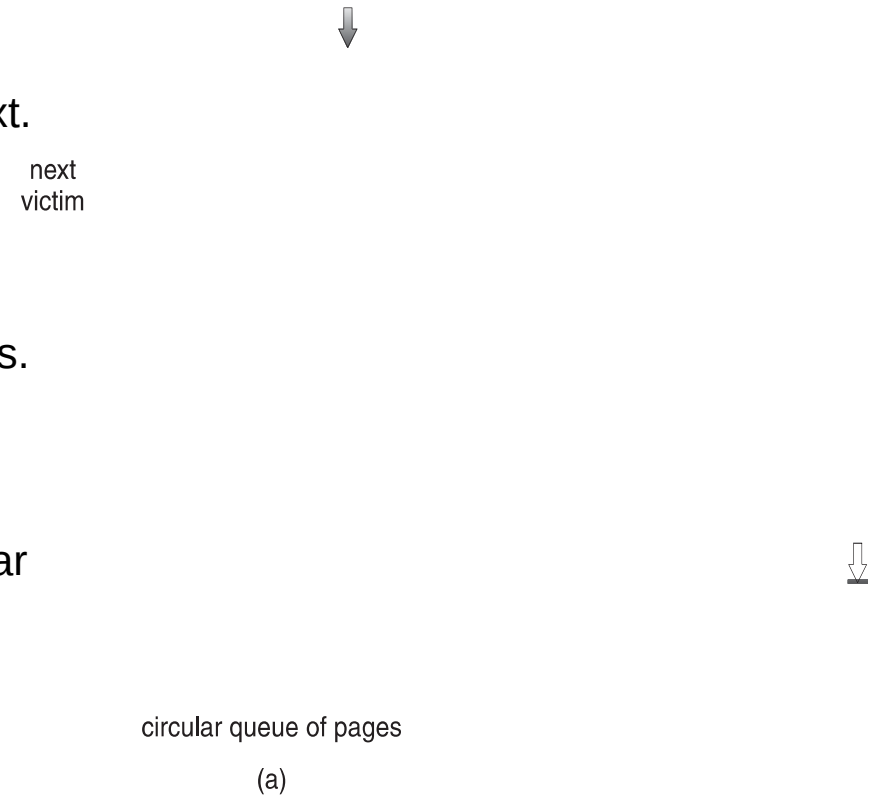
- Record reference bits at regular intervals
- Keep 8 bits (say) for each page in a table in memory
- Periodically, shift the reference bit into the high-order bit,
  - That is, shift the other bits to the right, dropping the lowest bit.
- During page replacement, interpret 8bits as unsigned integer.
- The page with the lowest number is the LRU page.

LRU Approximation Algorithms (contd.)

- **Second-chance algorithm (=Clock** algorithm**)**
  - Based of FIFO replacement algorithm
    - But also uses the reference_bit
  - If page to be replaced has
    - reference_bit = 0 => replace it
    - reference_bit = 1 =>
      - set reference_bit=0, leave page in memory
      - replace next page, subject to same rules
- **Idea:**
  - If reference_bit = 1, we give the page a 2nd chance
    - and move on to select the next FIFO page.
  - When a page gets a 2nd chance, its reference bit is cleared,
    - and its arrival time is reset to the current time.
  - Thus, a page that is given a 2nd chance will not be replaced until all other pages have been replaced (or given second chances).
  - In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

Second-Chance (clock) Page-Replacement Algorithm

- A circular queue implementation.
- Uses a pointer
  - that is, a hand on the clock
  - indicates the page to be replaced next.
- When a frame is needed, the pointer advances until it finds a page with a 0 reference bit.
- As it advances, it clears the reference bits.
- Once a victim page is found,
  - the page is replaced, and
  - the new page is inserted in the circular queue in that position.
- BSD Unix used this



Enhanced Second-Chance Algorithm

- Improve algorithm by using both
  - reference bit and
  - **modify bit** (if available)
- Consider ordered pair (reference, modify). 4 situations are possible:
  1. **(0, 0)** not 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 4 classes (different implementations are possible)
  - Replace the 1st page encountered in the lowest non-empty class
  - Might need to search circular queue several times
- This algorithm accounts for the pages that have been modified
  - in order to reduce the number of I/Os required.
- Mac OS used this

Counting Algorithms

- **Idea:** 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

- MFU, LFU
  - Not common.
  - The implementation of these algorithms is expensive
  - They do not approximate OPT replacement well

Page-Buffering Algorithms

- Other procedures are often used in addition to a specific page-replacement algo
- **Solution 1**
  - Systems commonly keep a pool of free frames.
  - When a page fault occurs, a victim frame is chosen as before.
  - However, before the victim is written out, the desired page is read into a free frame from the pool
  - This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out.
  - When the victim is later written out, its frame is added to the free-frame pool.
- **Solution 2**
  - Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- **Solution 3**
  - Remember which page was in each frame of the pool of free frames.
    - See Solution 1, last step
  - If one of those pages is references, do not page in, just get it from the pool.

Allocation of Frames

- Another issue OS needs to consider -- **frame allocation**.
  - Allocate frames for the OS
  - Keep some frames as free frame buffer pool

- How do we allocate the fixed amount of remaining free memory among the various processes?

Constraints of Allocation of Frames

- Our strategies for the allocation of frames are **constrained** in various ways
  - Cannot allocate more than the total number of available frames
    - Unless there is page sharing
  - Must also allocate at least a **minimum** number of frames per process
    - For performance
    - Instruction set constraints
- As # frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
- Recall: when a page fault occurs before an executing instruction is complete, the instruction must be restarted.
  - Thus, we must have enough frames to hold all the different pages that any single instruction can reference.
- 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*

Allocation of Frames (contd.)

- Two major allocation schemes
  1. **Fixed allocation**
  2. **Priority allocation**
- Many variations

Fixed Allocation

- **Equal Allocation**
  - Allocate free frames equally among the processes
    - 100 frames and 5 processes,
    - give each process 20 frames
- **Proportional allocation**
  - Allocate according to the size of process
  - S_j = size of process P_j
  - $S = \sum S_j$
  - m = total number of frames
  - a_j = allocation for $P_j = S_j/S * m$
  - If $m = 64$, $S_1 = 10$, $S_2 = 127$ then
    - $a_1 = 10/137 * 64 \approx 5$
    - $a_2 = 127/137 * 64 \approx 59$

Fixed Allocation (contd.)

- In both equal and proportional allocation, the allocation may vary according to the multiprogramming level.
  - If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process.
  - If the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

Priority Allocation

- May want to give a high-priority process more memory then to low-priority processes
  - to speed its execution

- **Priority Allocation** -- use a proportional allocation scheme but
  - using priorities rather than *size*, or
  - a combination of *priority* and *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

Global vs. Local Allocation

- **Global Replacement**
  - Process selects a replacement frame from the set of all frames
    - Hence, one process can take a frame from another process
  - **Cons:** a process cannot control its own page-fault rate.
    - The set of pages in memory for a process depends on the paging behavior of other processes.
    - The same process may perform quite differently due to totally external circumstances.
  - **Pros:** greater throughput than local, so more common

- **Local replacement**
  - Each process selects from only its own set of allocated frames
  - **Pros:** More consistent per-process performance
  - **Cons:** Process is slowed down even if other less used pages of memory are available

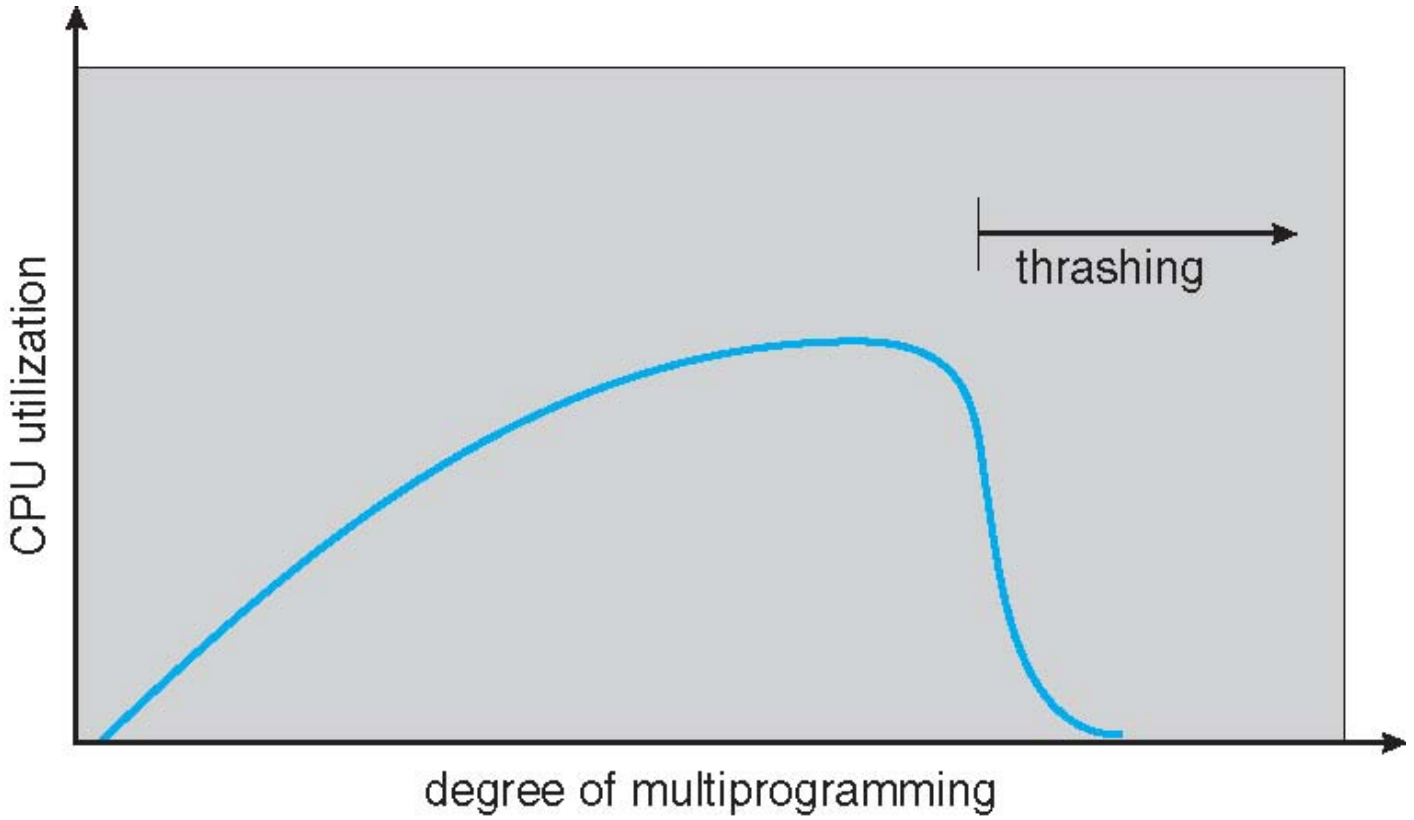
Thrashing

- 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 needs replaced frame back
  - Quickly faults again, and again,
    - replacing pages that it must bring back in immediately
- This high paging activity is called **thrashing**.
  - A process is thrashing if it is spending more time paging than executing.

Cause of Thrashing

- Thrashing results in severe performance problems.
  - Consider a scenario next
  - Actual behavior of early paging systems
- **Thrashing scenario**
  - System is busy, a process needs more pages, but not enough frames
  - Thrashing happens for this process
  - If global replacement is used
    - The process starts taking away pages from other processes
    - They also start to page fault
  - This leads to low CPU utilization
    - Hence, OS thinks it needs to increase the degree of multiprogramming
      - More processes are added to the system
      - More pages faults happen
      - System throughput plunges
        - » No work is getting done

Thrashing (Cont.)



Dealing with Thrashing

- We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm).
  - With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.
- However, the problem is not entirely solved.
  - If processes are thrashing, they will be in the queue for the paging device most of the time.
  - The average service time for a page fault will increase because of the longer average queue for the paging device.
  - Thus, the effective access time will increase even for a process that is not thrashing.

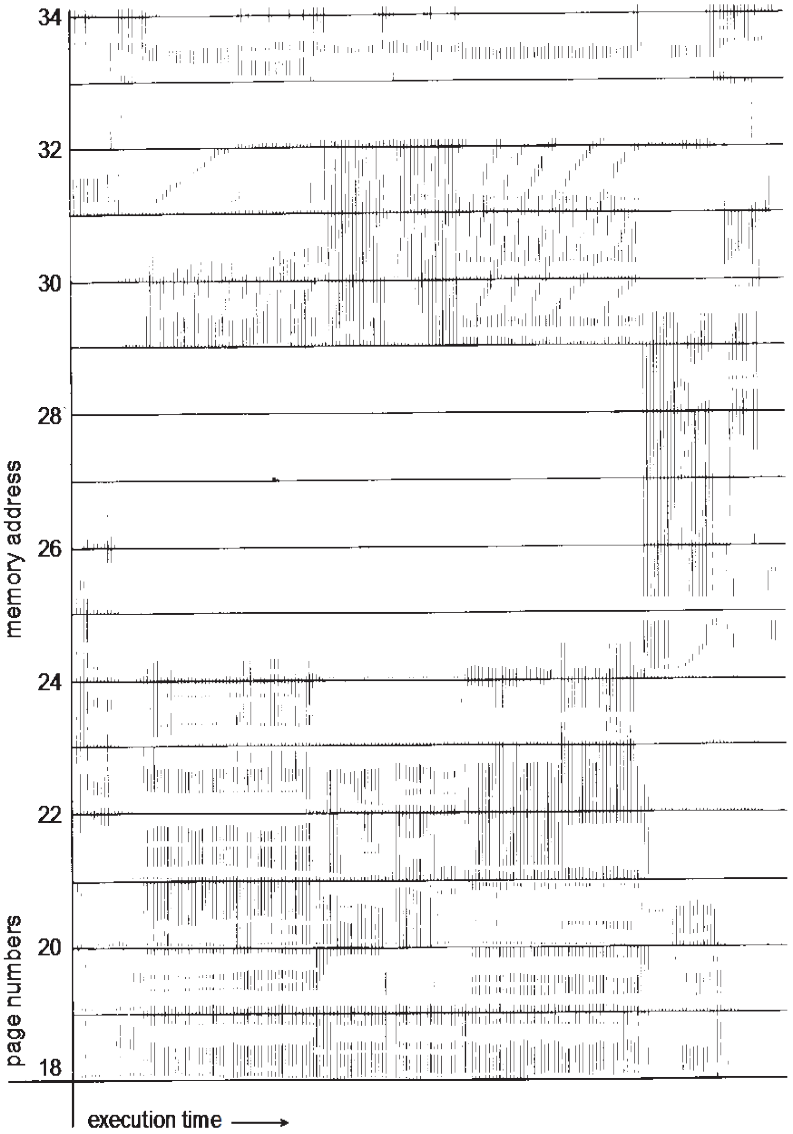
Locality Model

- To prevent thrashing, we must provide a process with as many frames as it needs.
  - But how do we know how many frames it “needs”?
  - There are several techniques.
  - The working-set strategy (discussed shortly) starts by looking at how many frames a process is actually using.
    - This approach defines **the locality model** of process execution.

Locality Model

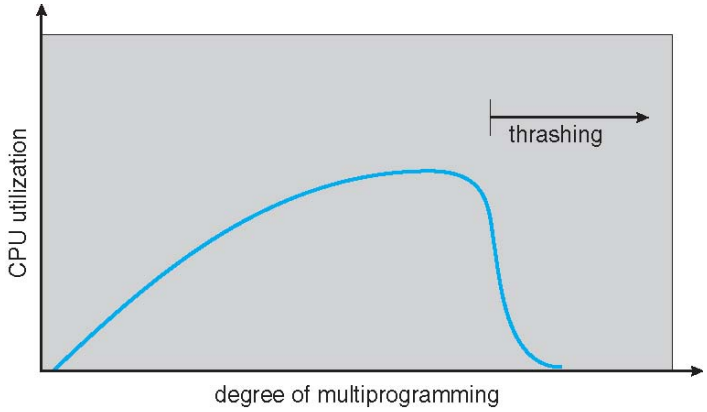
- A locality is a set of pages that are actively used together
- As a process executes, it moves from locality to locality.
- A program is generally composed of several different localities,
  - which may overlap.
- For example, when a function is called, it defines a new locality.
  - In this locality, memory references are made to
    - the instructions of the function call,
    - its local variables, and
    - a subset of the global variables.
  - When we exit the function, the process leaves this locality
  - We may return to this locality later.
- The locality model states that all programs will exhibit this basic memory reference structure.
  - Note that the locality model is the unstated principle behind the caching discussions so far in this class
  - If accesses to any types of data were random rather than patterned, caching would be useless.

Locality In A Memory-Reference Pattern



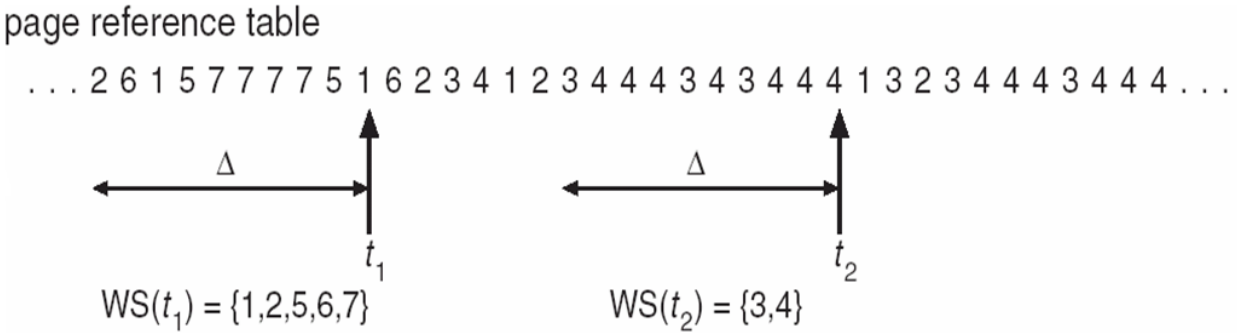
Locality Model

- Suppose we allocate enough frames to a process to accommodate its current locality.
  - It will fault for the pages in its locality until all these pages are in memory
  - Then, it will not fault again until it changes localities.
- If we do not allocate enough frames to accommodate the size of the current locality, the process **will thrash**
  - since it cannot keep in memory all the pages that it is actively using.
- Why does thrashing occur?
  - $\sum$ (size of locality) > total memory size



Working-Set Model

- The **working-set model --** a strategy for preventing thrashing
  - It is based on the assumption of locality.
- Defines $\Delta \equiv$ working-set window
  - A fixed number of page references, e.g. 10,000
- **Working set**
  - The set of pages in the most recent Δ page references
  - WS is an approximation of the program's locality.
  - If a page is in active use, it will be in the WS



Working-Set Model (contd.)

- WSS_i (working set **size** of Process P_i)
  - The 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 for frames
  - Approximation of locality
- Let m be the number of available frames
- if $D > m \Rightarrow$ Thrashing

Working-Set Model (contd.)

- **Policy:**
  - OS monitors the WS of each process
    - OS allocates enough frames to each WS
  - If enough extra frames, another process can be initiated.
  - if $D > m$, then suspend one of the processes
    - The process's pages are written out (swapped),
      - and its frames are reallocated to other processes.
      - The suspended process can be restarted later.

Working-Set Model (contd.)

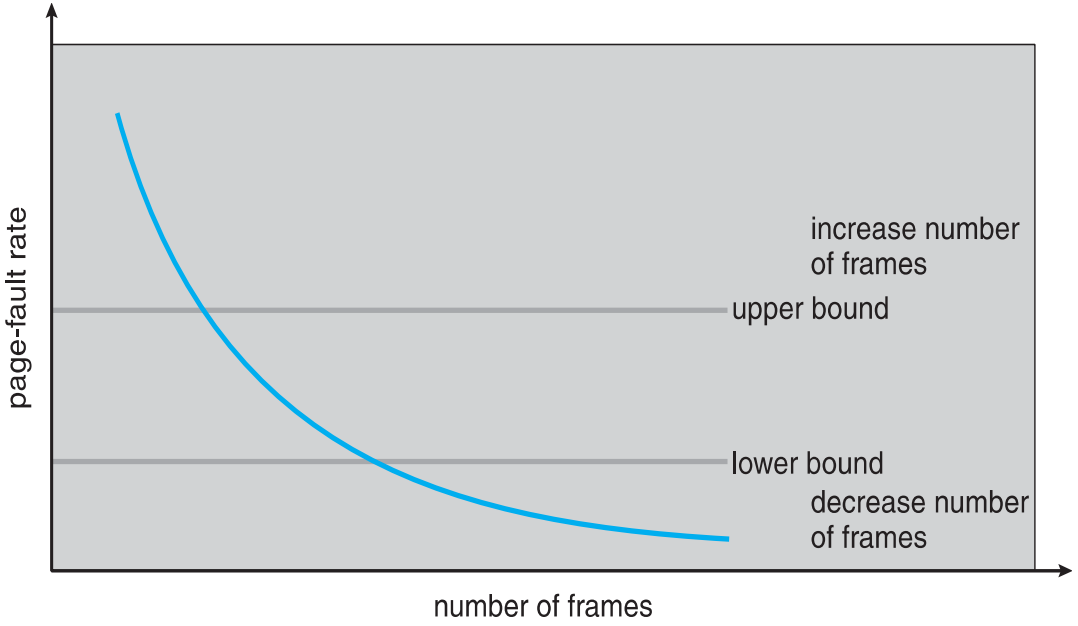
- **Advantages** of the WS model
  - Prevents thrashing, while keeping the degree of multiprogramming as high as possible.
  - Thus, it optimizes CPU utilization.
- **The difficulty** with the WS model
  - How to efficiently keep track of the working set?
  - The working-set window Δ is a moving window.
    - At each memory reference,
      - a new reference appears at one end, and
      - the oldest reference drops off the other end.
    - A page is in the WS if it is referenced anywhere in Δ
  - **Solution:**
    - Use an approximation of the WS model
    - Explained next

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 each reference bit into the corresponding part of 2 the bits
    - set the values of all reference bits to 0
  - If a page fault occurs,
    - we can examine the current reference bit and 2 in-memory bits
    - determine if a page was used within the last 10,000 to 15,000 references.
    - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
  - Cannot tell where, within an interval of 5,000, a reference occurred
- Improvement = 10 bits and interrupt every 1000 time units
  - But cost to service these more frequent interrupts will be higher

Page-Fault Frequency

- Consider another strategy to control thrashing
- Uses the page-fault frequency (PFF) per each process
- A more direct approach than the Working Set model
- **Idea:**
- Establish “acceptable” **page-fault frequency** (**PFF**) rate, and
- Use local replacement policy
- If actual rate too low => the process loses frame
- If actual rate too high => the process gains frame



Page-Fault Frequency (contd.)

- As with the working-set strategy, we may have to swap out a process.
- If the page-fault rate increases and no free frames are available:
  - Select some process and swap it out to backing store
  - The freed frames are then distributed
    - to the processes with high page-fault rates.

Other Considerations -- Prepaging

- An obvious property of pure demand paging is the large number of page faults that occur when a process is started or swapped in
  - How to deal with this?
- Prepaging – using some strategy is to bring into memory at one time all the pages that will be needed.
  - 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

Other Issues – Page Size

- The designers of an OS for an existing machine seldom have a choice concerning the page size.
- However, when new machines are being designed, a decision regarding the best page size must be made.
- There is no single best page size.
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - I/O overhead
  - Locality
  - Other factors
- Always power of 2,
  - usually in the range 2^{12} (4K) to 2^{22} (4MB)
- On average, growing over time

Other Issues – Program Structure

- Assume that pages are 128 words in size.
- Program structure
  - `int[128,128] data; //assume it is paged out initially`
  - 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;
```

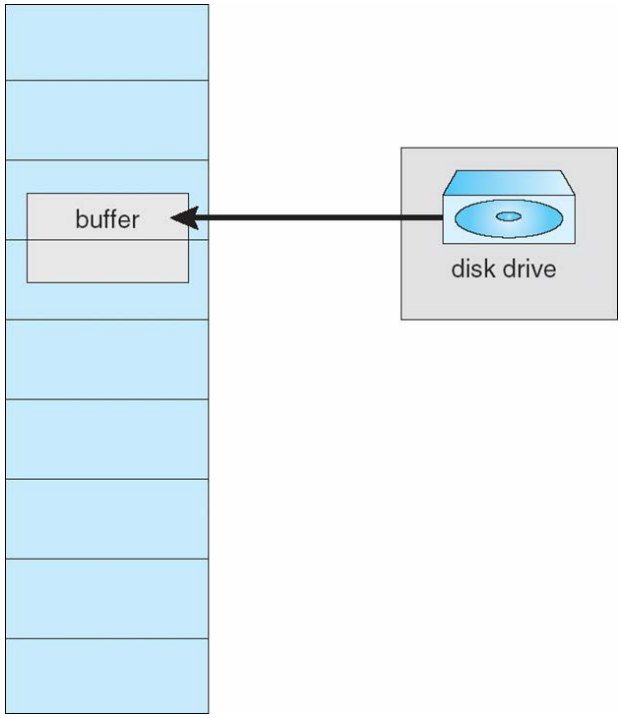
  - If the OS allocates fewer than 128 frames to the entire program
    - 128 x 128 = 16,384 page faults
  - **Program 2**

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

    128 page faults
- **A programmer should know this to write efficient code**

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



End of Chapter 9