10. Virtual Memory

Sungyoung Lee

College of Engineering
KyungHee University

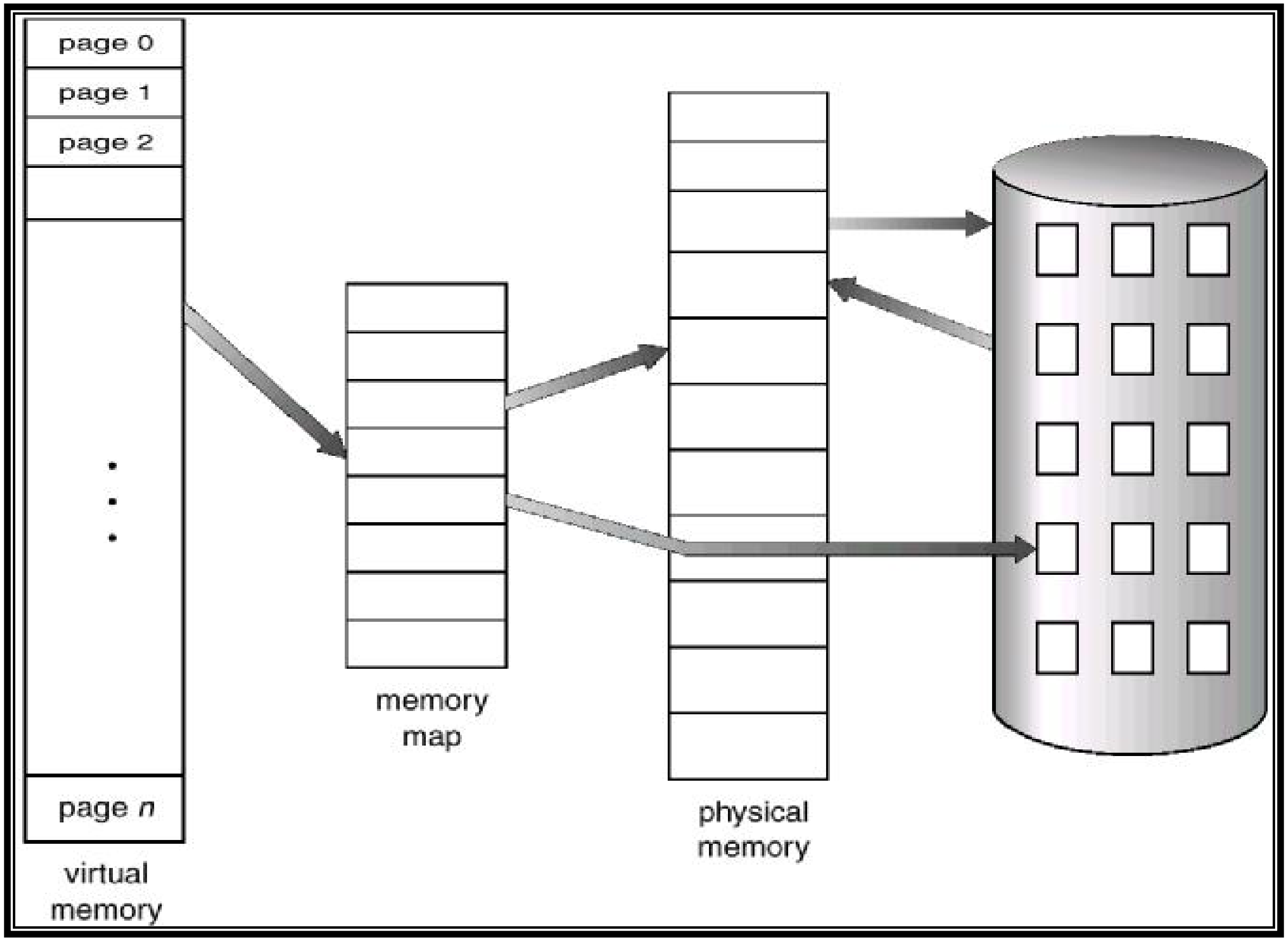
Contents

Background

- **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

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

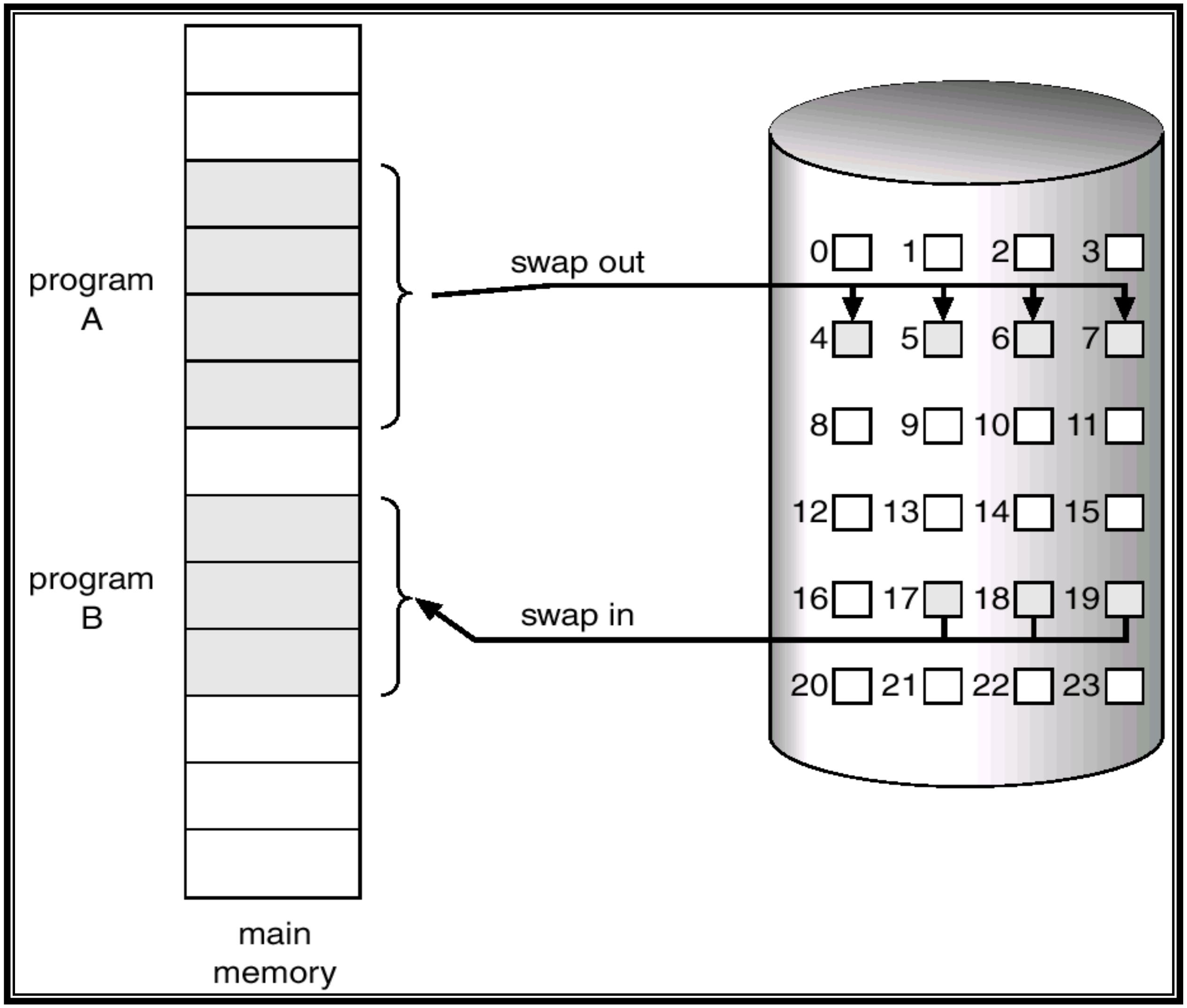


Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

Transfer of a Paged Memory to Contiguous Disk Space



Demand Paging

- A paging system with (page-level) swapping
- Bring a page into memory only when it is needed
  - Cf) swapping: entire process is moved
- OS uses main memory as a (page) cache of all of the data allocated by processes in the system
  - Initially, pages are allocated from physical memory frames
  - When physical memory fills up, allocating a page requires some other page to be evicted from its physical memory frame
- Evicted pages go to disk (only need to write if they are dirty)
  - To a swap file
  - Movement of pages between memory/disks is done by the OS
  - Transparent to the application

Demand Paging (Cont’d)

- Why does this work? à Locality
  - Temporal locality: locations referenced recently tend to be referenced again soon
  - Spatial locality: locations near recently referenced locations are likely to be referenced soon

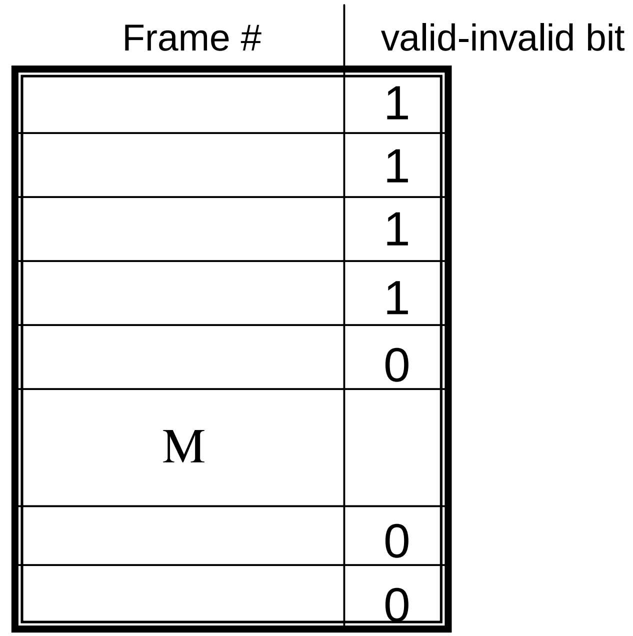
- Locality means paging can be infrequent
  - Once you’ve paged something in, it will be used many times
  - On average, you use things that are paged in
  - But this depends on many things:
    - Degree of locality in application
    - Page replacement policy
    - Amount of physical memory
    - Application’s reference pattern and memory footprint

Demand Paging (Cont'd)

- Why is this "demand" paging?
  - When a process first starts up, it has a brand new page table, with all PTE valid bits "false"
    - No pages are yet mapped to physical memory
  - When the process starts executing:
    - Instructions immediately fault on both code and data pages
    - Faults stop when all necessary code/data pages are in memory
    - Only the code/data that is needed (demanded!!) by process needs to be loaded
    - What is needed changes over time, of course…

Valid-Invalid Bit

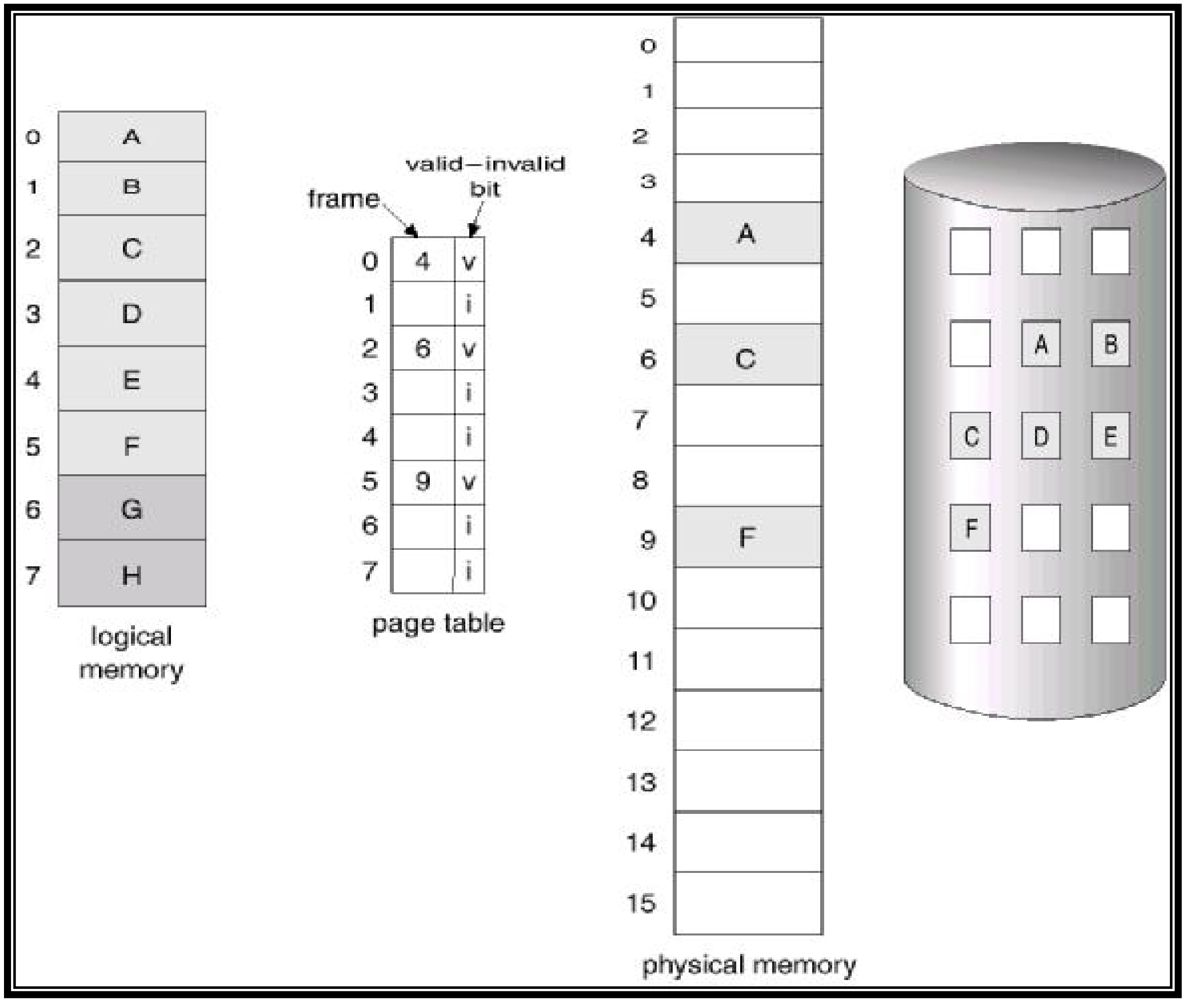
- With each page table entry a valid–invalid bit is associated
  (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- Example of a page table snapshot



page table

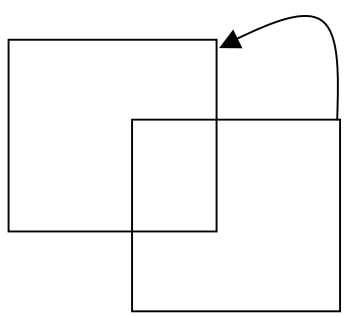
- During address translation, if valid–invalid bit in page table entry is 0
  ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



Page Fault

- If there is ever a reference to a page, first reference will trap to OS
  ⇒ page fault
- OS looks at another table to decide:
  - Invalid reference ⇒ abort
  - Just not in memory
- Get empty frame
- Swap page into frame
- Reset tables, validation bit = 1
- Restart instruction (if cannot be restarted?)
  - block move

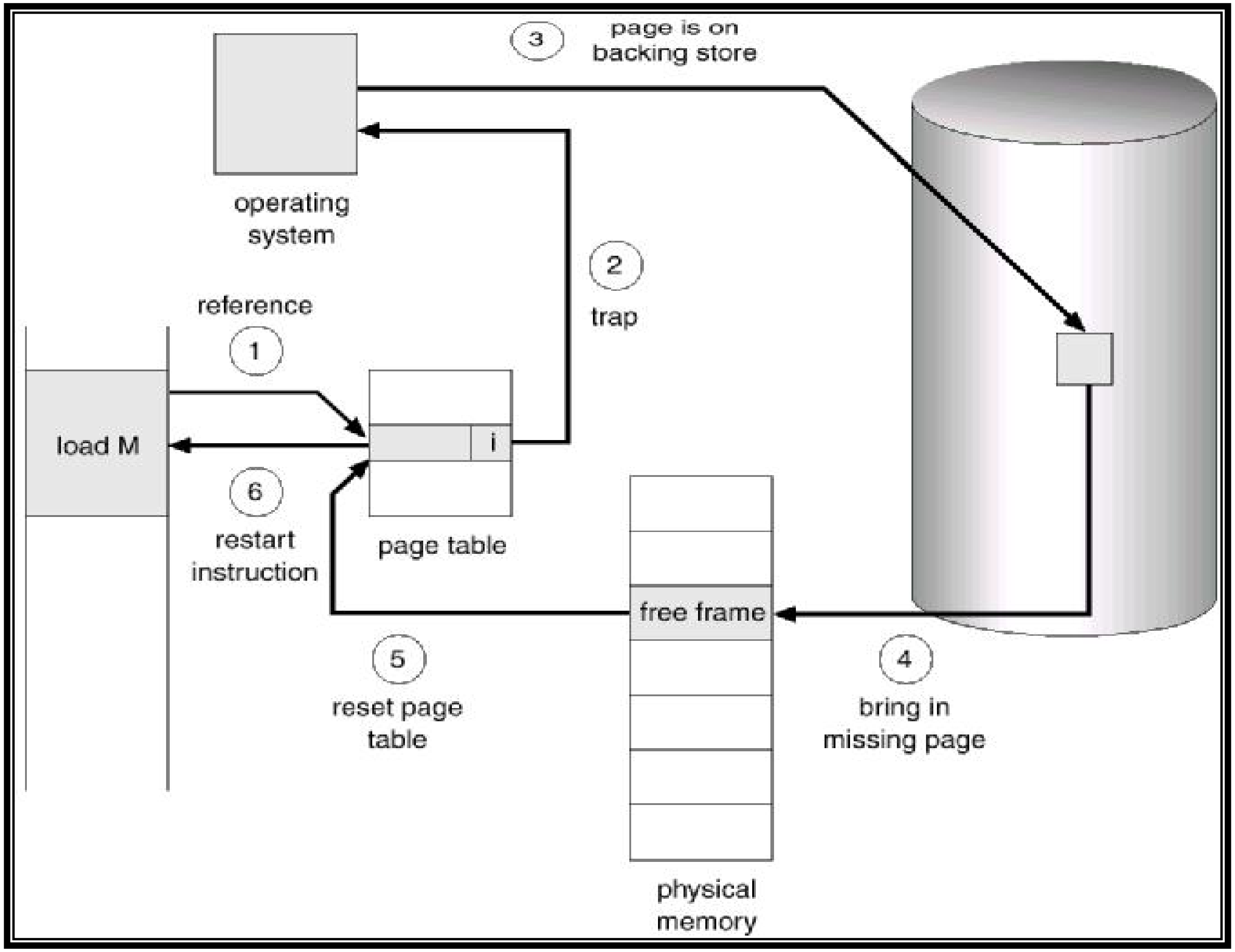


  - auto increment/decrement location

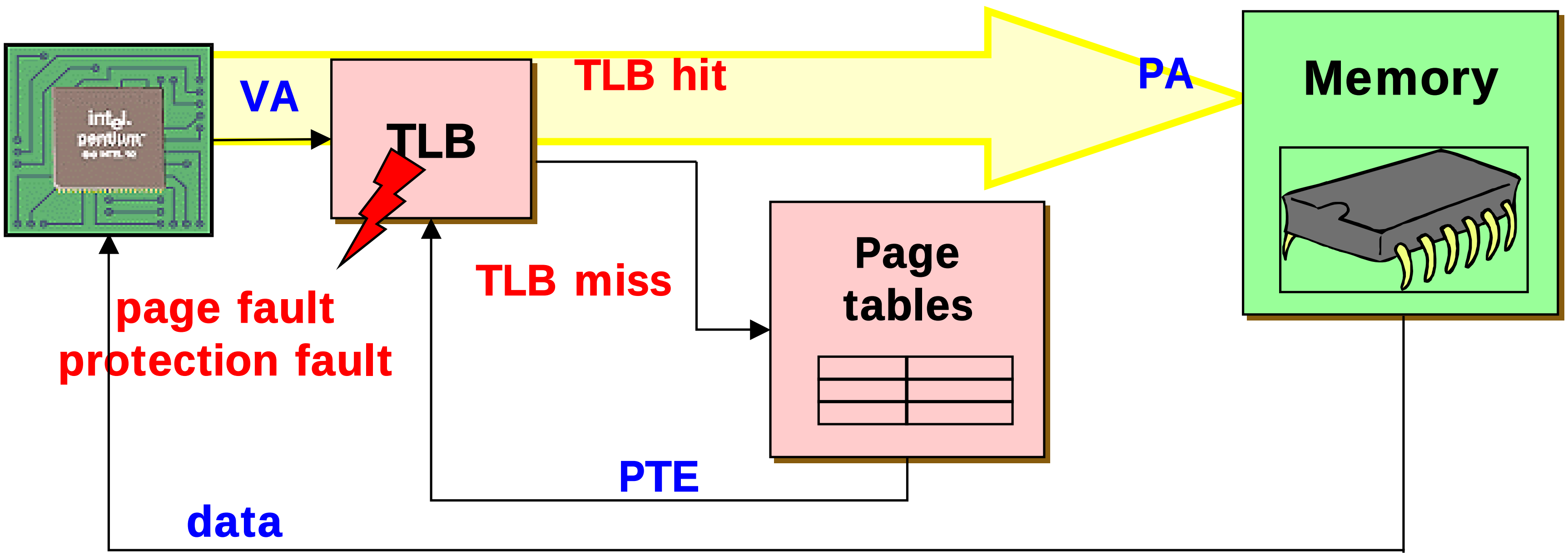
Page Fault

- What happens to a process that references a virtual address in a page that has been evicted?
  - When the page was evicted, the OS sets the PTE as invalid and stores (in PTE) the location of the page in the swap file
  - When a process accesses the page, the invalid PTE will cause an exception to be thrown
- The OS will run the page fault handler in response
  - Handler uses invalid PTE to locate page in swap file
  - Handler reads page into a physical frame, updates PTE to point to it and to be valid
  - Handler restarts the faulted process
- Where does the page that's read in go?
  - Have to evict something else (page replacement algorithm)
  - OS typically tries to keep a pool of free pages around so that allocations don't inevitably cause evictions

Steps in Handling a Page Fault



- Situation
  - Process is executing on the CPU, and it issues a read to a (virtual) address



Memory Reference (Cont'd)

- The common case
  - The read goes to the TLB in the MMU
  - TLB does a lookup using the page number of the address
  - The page number matches, returning a PTE
  - TLB validates that the PTE protection allows reads
  - PTE specifies which physical frame holds the page
  - MMU combines the physical frame and offset into a physical address
  - MMU then reads from that physical address, returns value to CPU

Memory Reference (Cont'd)

- TLB misses: two possibilities
  - (1) MMU loads PTE from page table in memory
    - Hardware managed TLB, OS not involved in this step
    - OS has already set up the page tables so that the hardware can access it directly
  - (2) Trap to the OS
    - Software managed TLB, OS intervenes at this point
    - OS does lookup in page tables, loads PTE into TLB
    - OS returns from exception, TLB continues

  - At this point, there is a valid PTE for the address in the TLB

Memory Reference (Cont'd)

- TLB misses
  - Page table lookup (by HW or OS) can cause a recursive fault if page table is paged out
    - Assuming page tables are in OS virtual address space
    - Not a problem if tables are in physical memory
  - When TLB has PTE, it restarts translation
    - Common case is that the PTE refers to a valid page in memory
    - Uncommon case is that TLB faults again on PTE because of PTE protection bits (e.g., page is invalid)

Memory Reference (Cont’d)

- Page faults
  - PTE can indicate a protection fault
    - Read/Write/Execute – operation not permitted on page
    - Invalid – virtual page not allocated, or page not in physical memory
  - TLB traps to the OS (software takes over)
    - Read/Write/Execute – OS usually will send fault back to the process, or might be playing tricks (e.g., copy on write, mapped files)
    - Invalid (Not allocated) – OS sends fault to the process (e.g., segmentation fault)
    - Invalid (Not in physical memory) – OS allocates a frame, reads from disk, and maps PTE to physical frame

What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - Algorithm
  - Performance
    - want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - 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 x (page fault overhead
  + [swap page out]
  + swap page in
  + restart overhead)

Demand Paging Example

- Memory access time = 1 microsecond

- 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out

- Swap Page Time = 10 msec = 10,000 usec

$$EAT = (1 – p) \times 1 + p \times (15000)$$
$$= 1 + 14999P \qquad \text{(in usec)}$$

Process Creation

- Virtual memory allows other benefits during process creation:
  - Copy-on-Write
  - Memory-Mapped Files

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

- Free pages are allocated from a *pool* of zeroed-out pages

Copy-On-Write

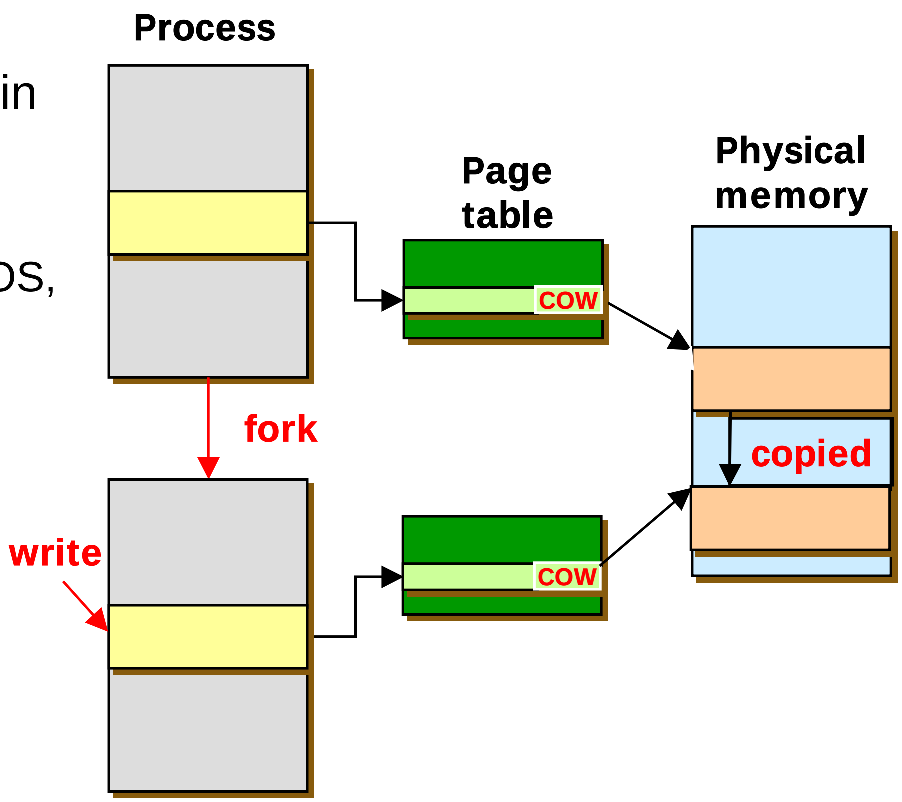
- Process creation
  - requires copying the entire address space of the parent process to the child process
  - Very slow and inefficient!

- Solution 1: Use threads
  - Sharing address space is free

- Solution 2: Use vfork() system call
  - vfork() creates a process that shares the memory address space of its parent
  - To prevent the parent from overwriting data needed by the child, the parent's execution is blocked until the child exits or executes a new program
  - Any change by the child is visible to the parent once it resumes
  - Useful when the child immediately executes exec()

Copy-On-Write (Cont'd)

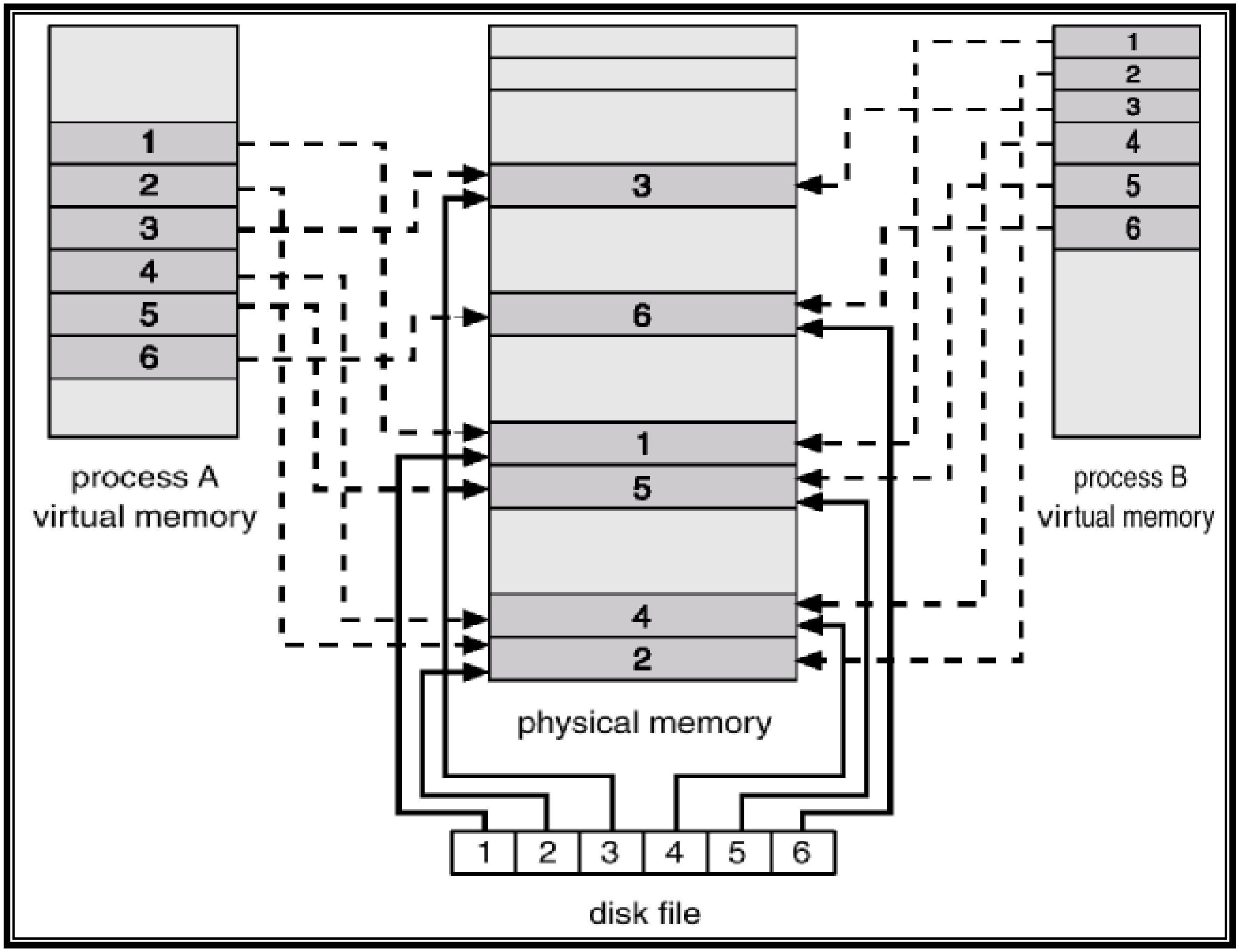
- Solution 3: Copy On Write (COW)
  - Instead of copying all pages, create shared mappings of parent pages in child address space.
  - Shared pages are protected as read-only in child.
    - Reads happen as usual
    - Writes generate a protection fault, trap to OS, and OS copies the page, changes page mapping in client page table, restarts write instruction



Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by *mapping* a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies file access by treating file I/O through memory rather than **read() write()** system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory Mapped Files



Memory-Mapped Files

- Memory-mapped files
  - Mapped files enable processes to do file I/O using memory references
    - Instead of open(), read(), write(), close()
  - mmap(): bind a file to a virtual memory region
    - PTEs map virtual addresses to physical frames holding file data
    - <Virtual address base + N> refers to offset N in file
  - Initially, all pages in mapped region marked as invalid
    - OS reads a page from file whenever invalid page is accessed
    - OS writes a page to file when evicted from physical memory
    - If page is not dirty, no write needed

Memory-Mapped Files (Cont'd)

- Note:
  - File is essentially backing store for that region of the virtual address space (instead of using the swap file)
  - Virtual address space not backed by “real” files also called “anonymous VM”

- Advantages
  - Uniform access for files and memory (just use pointers)
  - Less copying

- Drawbacks
  - Process has less control over data movement
    - OS handles faults transparently
  - Does not generalize to streamed I/O (pipes, sockets, etc.)

Page Replacement

- 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

Page Replacement

- When a page fault occurs, the OS loads the faulted page from disk into a page frame of memory

- At some point, the process has used all of the page frames it is allowed to use

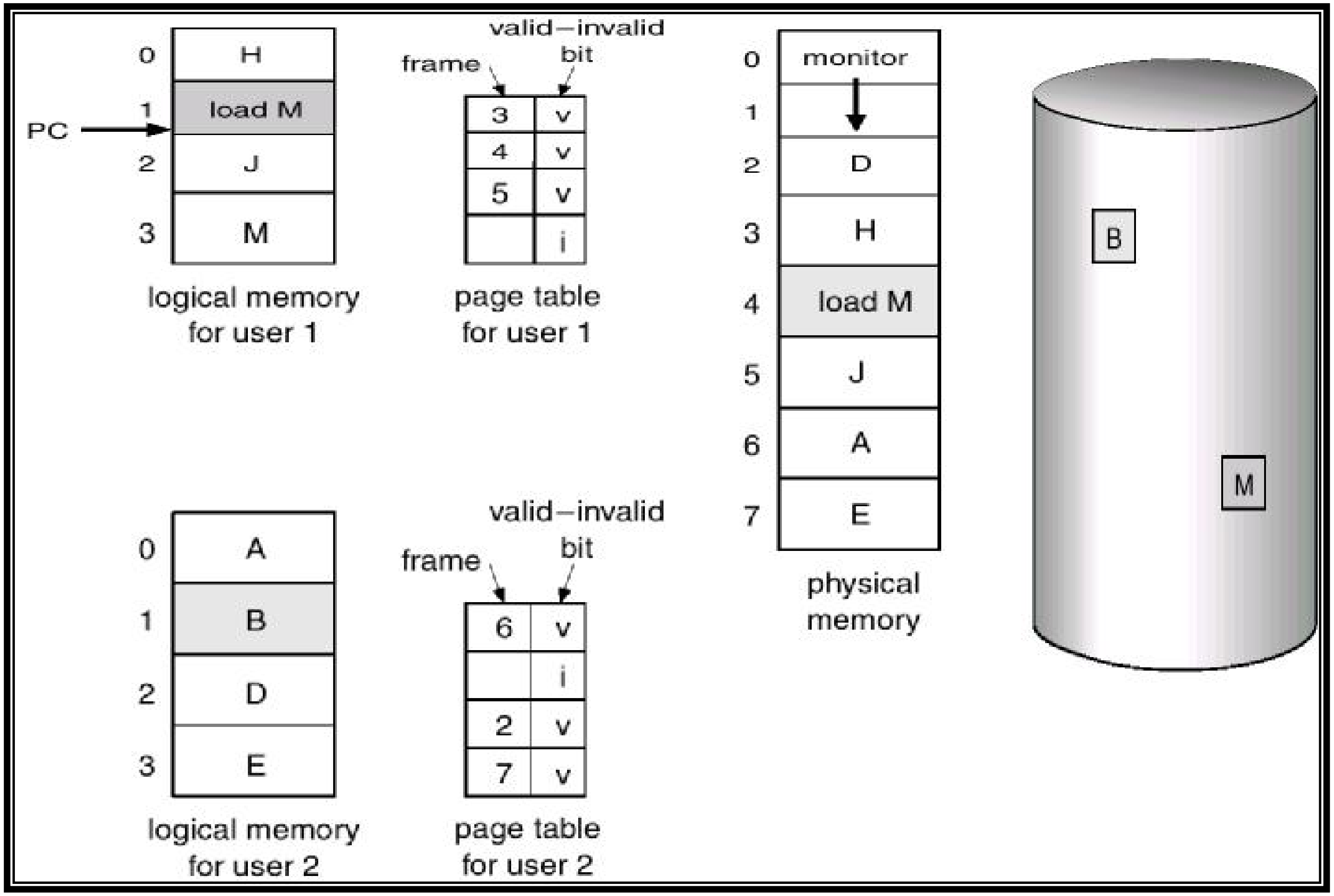
- When this happens, the OS must replace a page for each page faulted in
  - It must evict a page to free up a page frame

- The page replacement algorithm determines how this is done

Page Replacement (Cont'd)

- Evicting the best page
  - The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove
  - The best page to evict is the one never touched again
    - as process will never again fault on it
  - "Never" is a long time, so picking the page closest to "never" is the next best thing
    - Belady's proof: Evicting the page that won't be used for the longest period of time minimizes the number of page faults

Need For Page Replacement



Basic Page Replacement

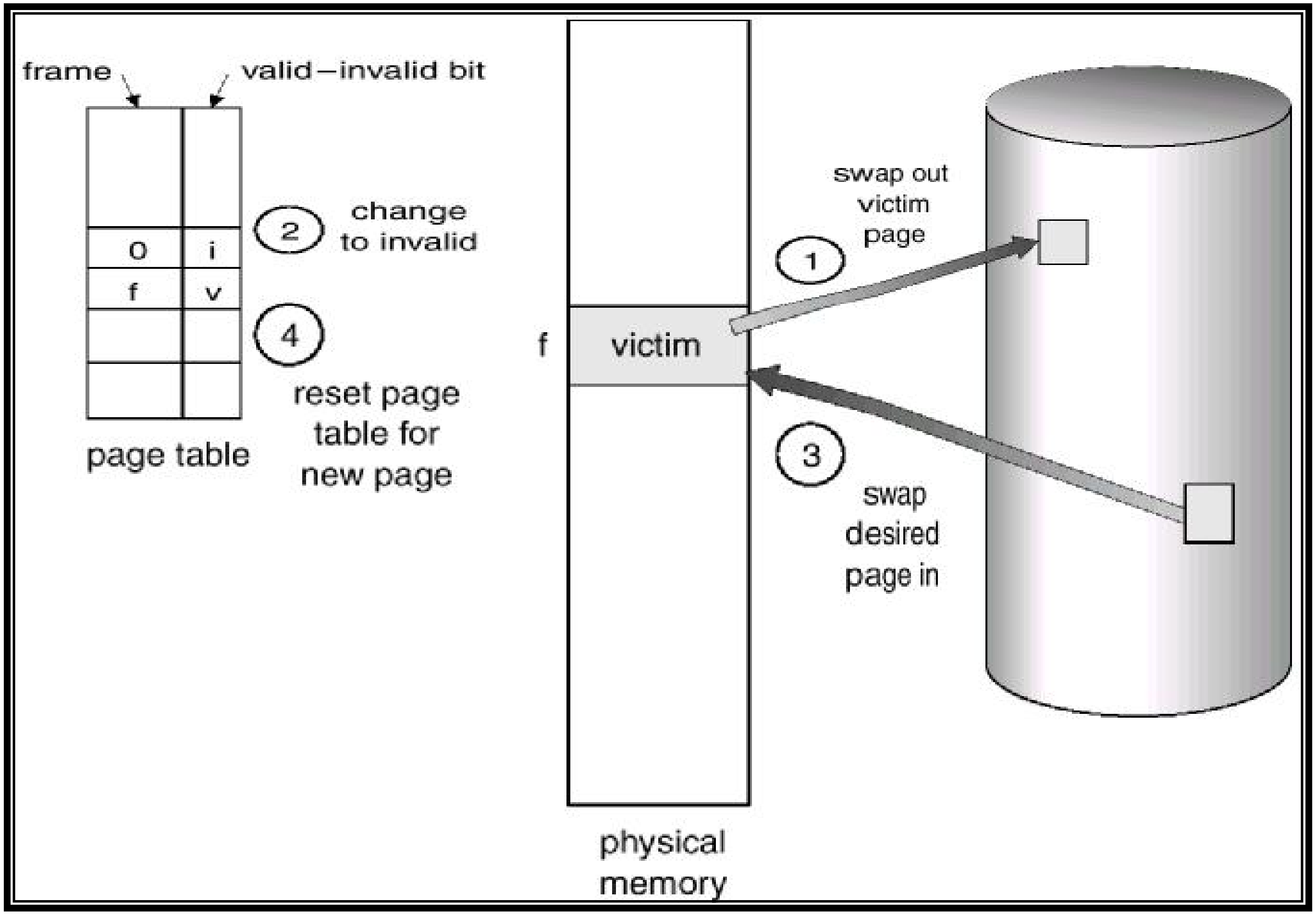
1. Find the location of the desired page on disk

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

3. Read the desired page into the (newly) free frame
   - Update the page and frame tables

4. Restart the process

Page Replacement

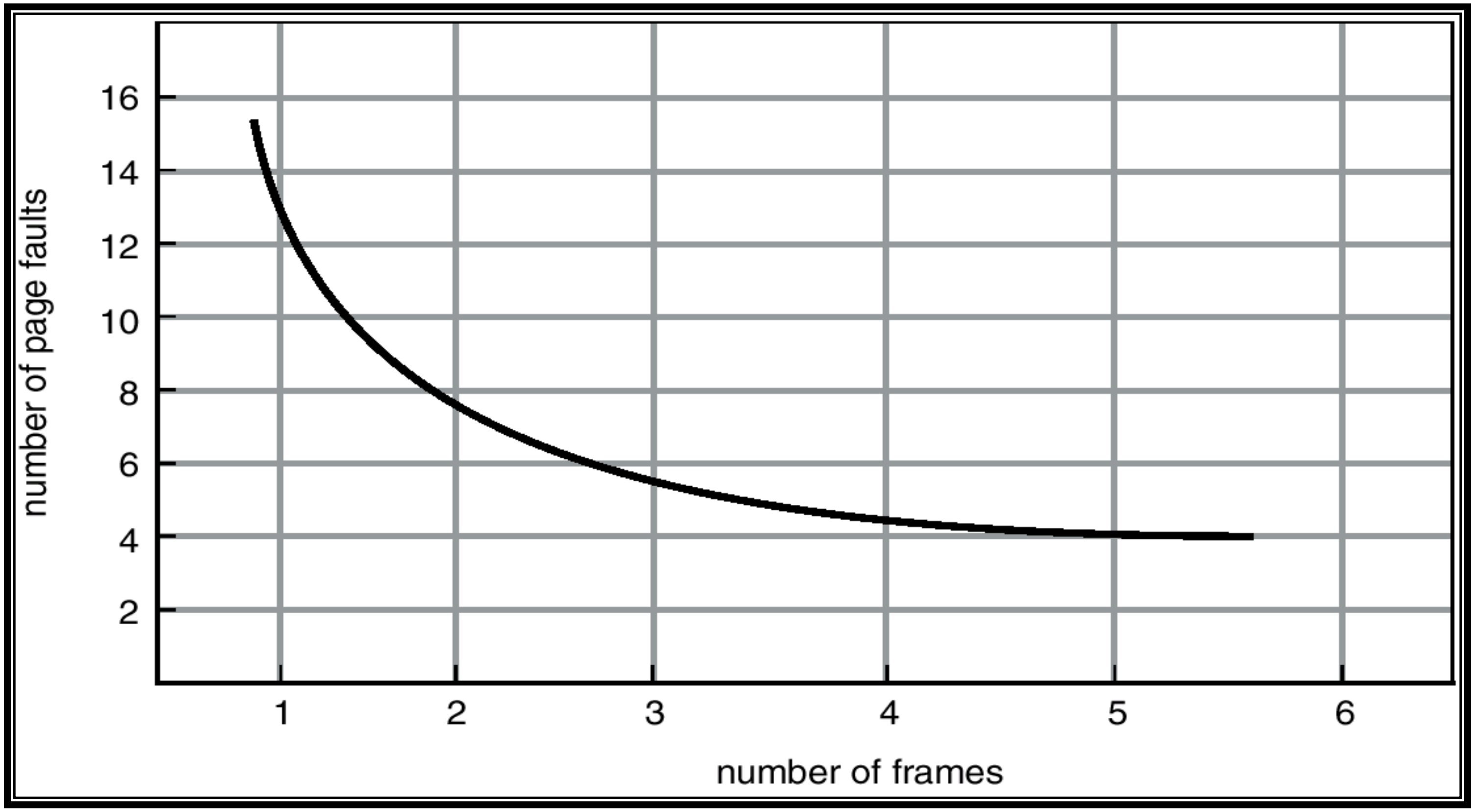


Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

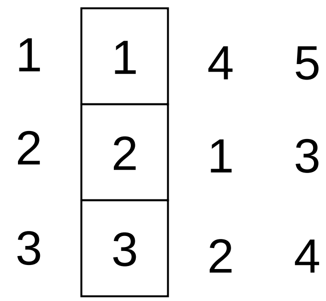
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Graph of Page Faults Versus The Number of Frames



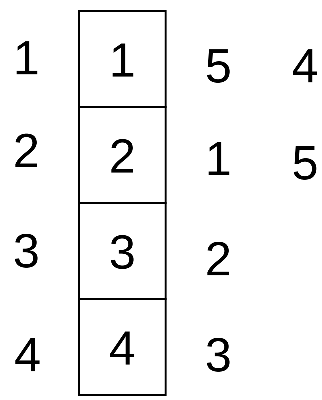
First-In-First-Out (FIFO) Algorithm

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



9 page faults

- 4 frames



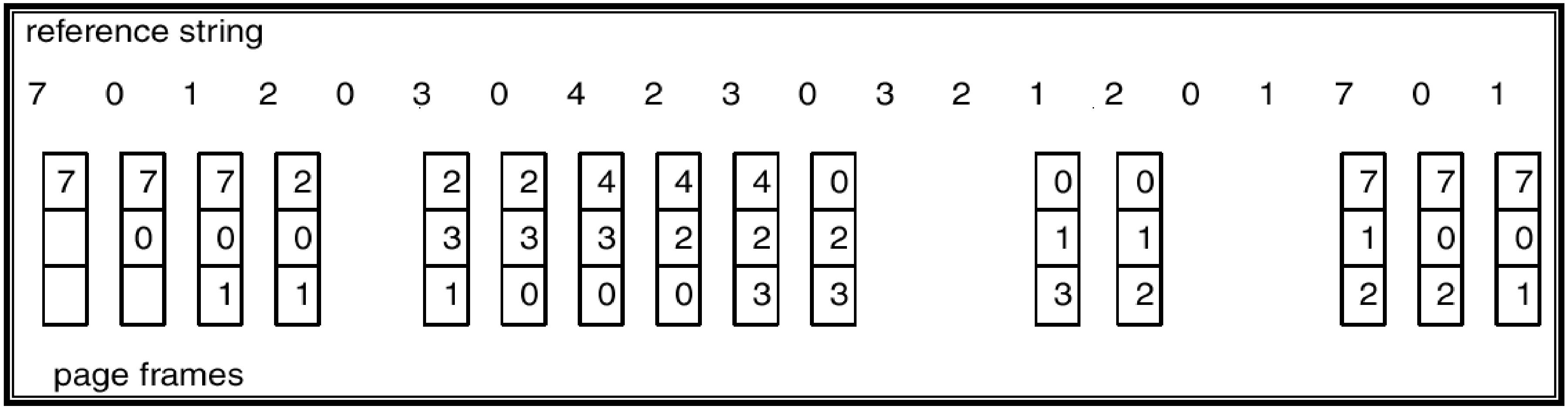
10 page faults

- FIFO Replacement – Belady's Anomaly
  - more frames ⇒ less page faults

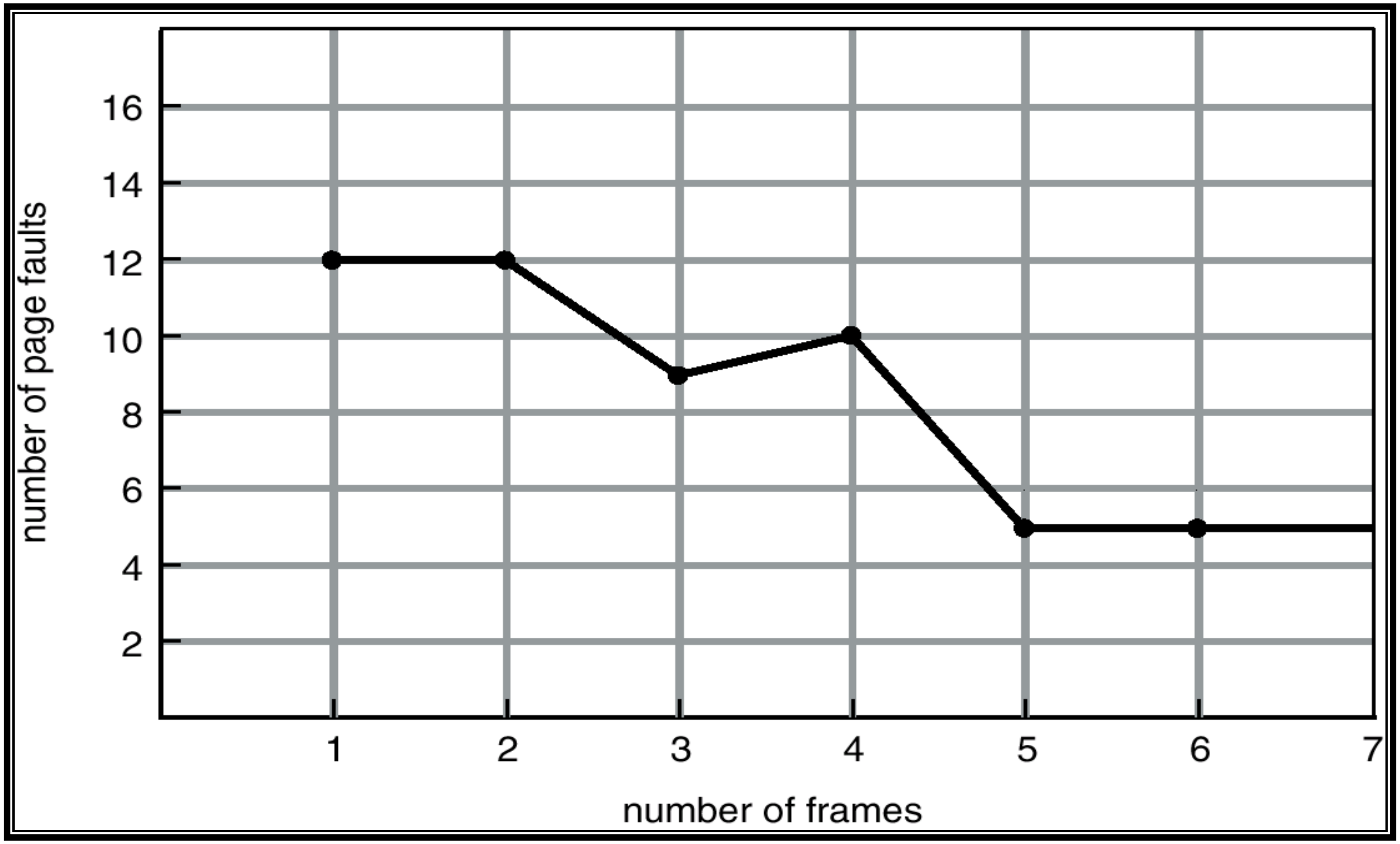
FIFO

- Obvious and simple to implement
  - Maintain a list of pages in order they were paged in
  - On replacement, evict the one brought in longest time ago
- Why might this be good?
  - Maybe the one brought in the longest ago is not being used
- Why might this be bad?
  - Maybe, it’s not the case
  - We don’t have any information either way
- FIFO suffers from “Belady’s Anomaly”
  - The fault rate might increase when the algorithm is given more memory

FIFO Page Replacement

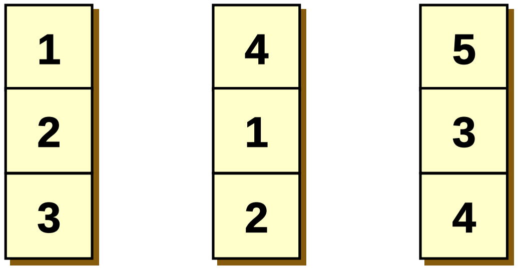


FIFO Illustrating Belady's Anomaly

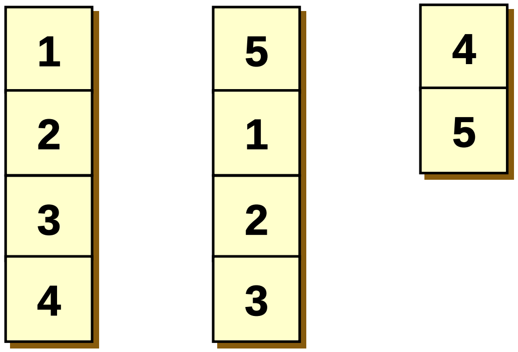


FIFO (Cont'd)

- Example: Belady's anomaly
  - Reference string: 1,2,3,4,1,2,5,1,2,3,4,5
  - 3 frames: 9 faults



  - 4 frames: 10 faults

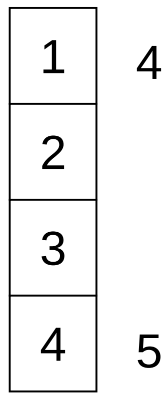


Optimal Algorithm

- Replace page that will not be used for longest period of time

- 4 frames example

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

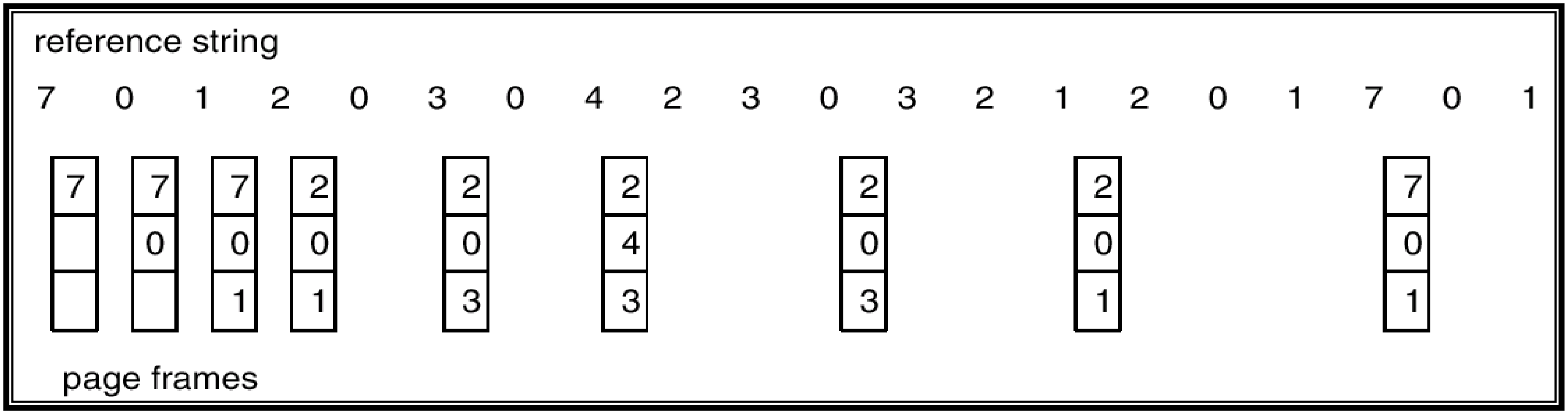


6 page faults

- How do you know this?

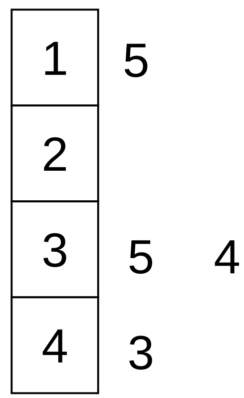
- Used for measuring how well your algorithm performs

Optimal Page Replacement



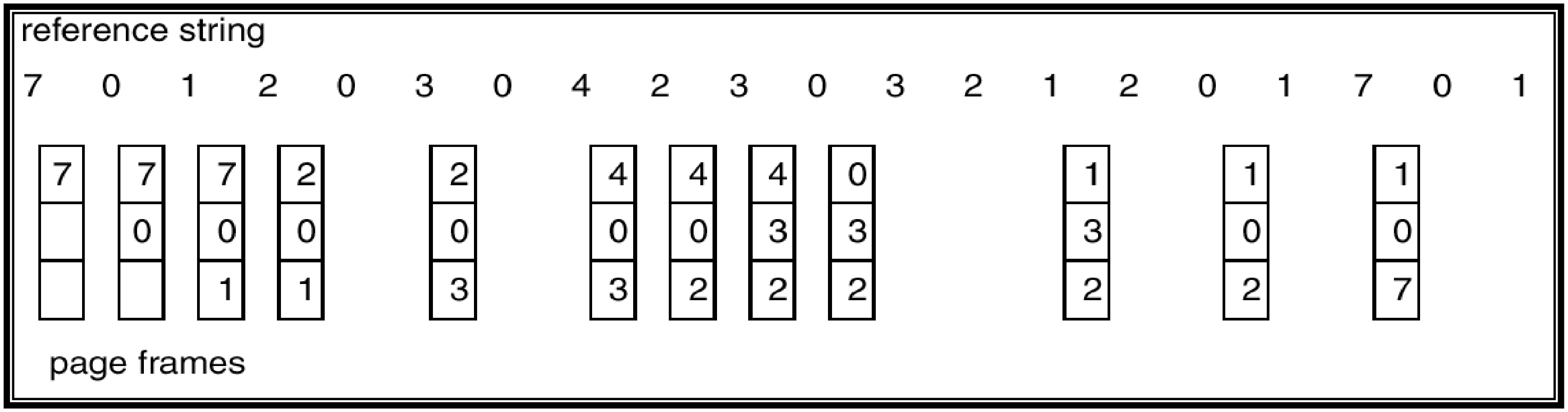
Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- 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 determine which are to change

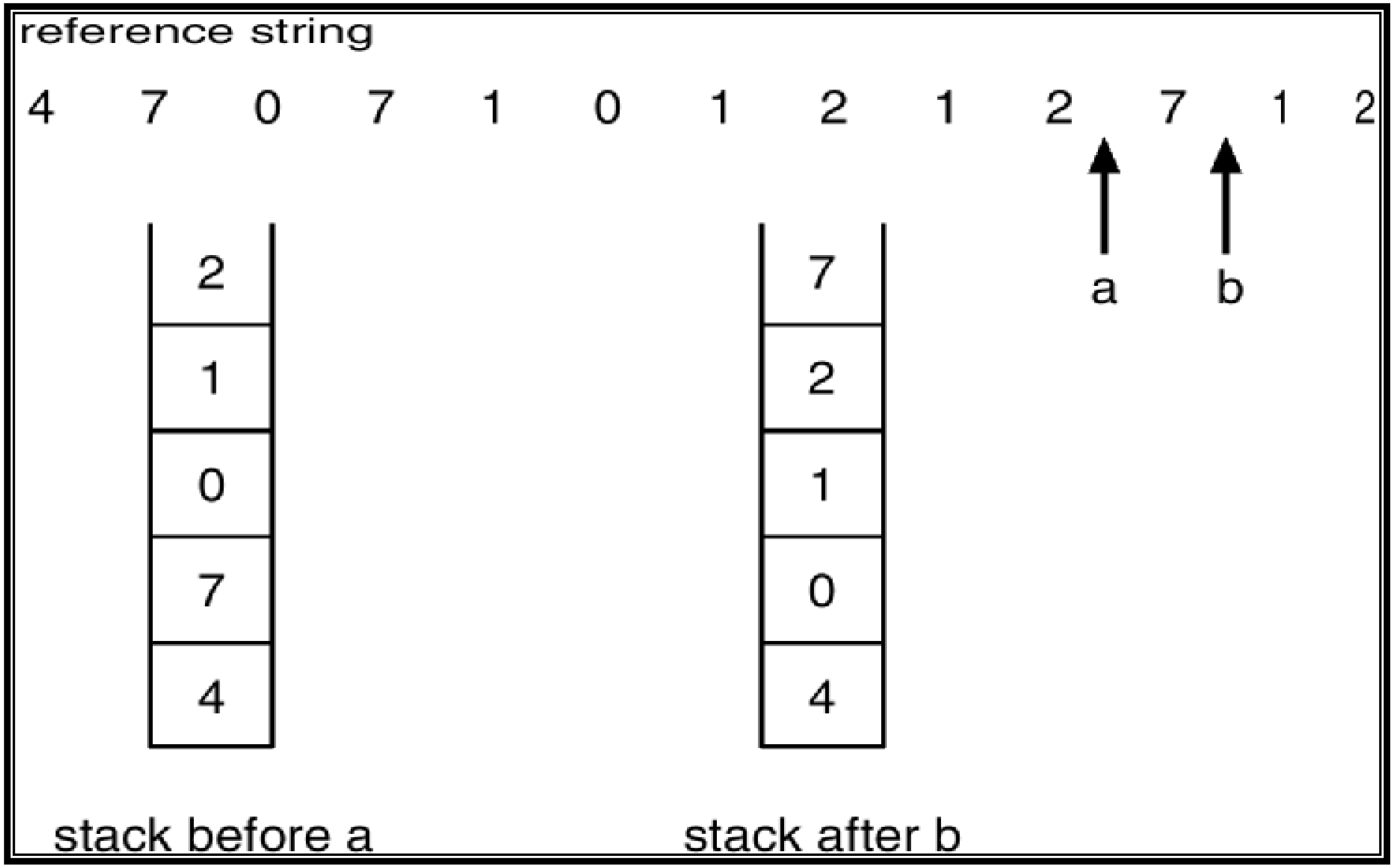
LRU Page Replacement



LRU Algorithm (Cont'd)

- 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

Use Of A Stack to Record The Most Recent Page References



LRU

- LRU uses reference information to make a more informed replacement decision
  - Idea: past experience gives us a guess of future behavior
  - On replacement, evict the page that has not been used for the longest time in the past
  - LRU looks at the past, Belady’s wants to look at future

- Implementation
  - To be perfect, need to timestamp every reference and put it in the PTE (or maintain a stack) – too expensive
  - So, we need an approximation

LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists). We do not know the order, however

- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1, then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

Approximating LRU

- Many LRU approximations use the PTE reference (R) bit
  - R bit is set whenever the page is referenced (read or written)

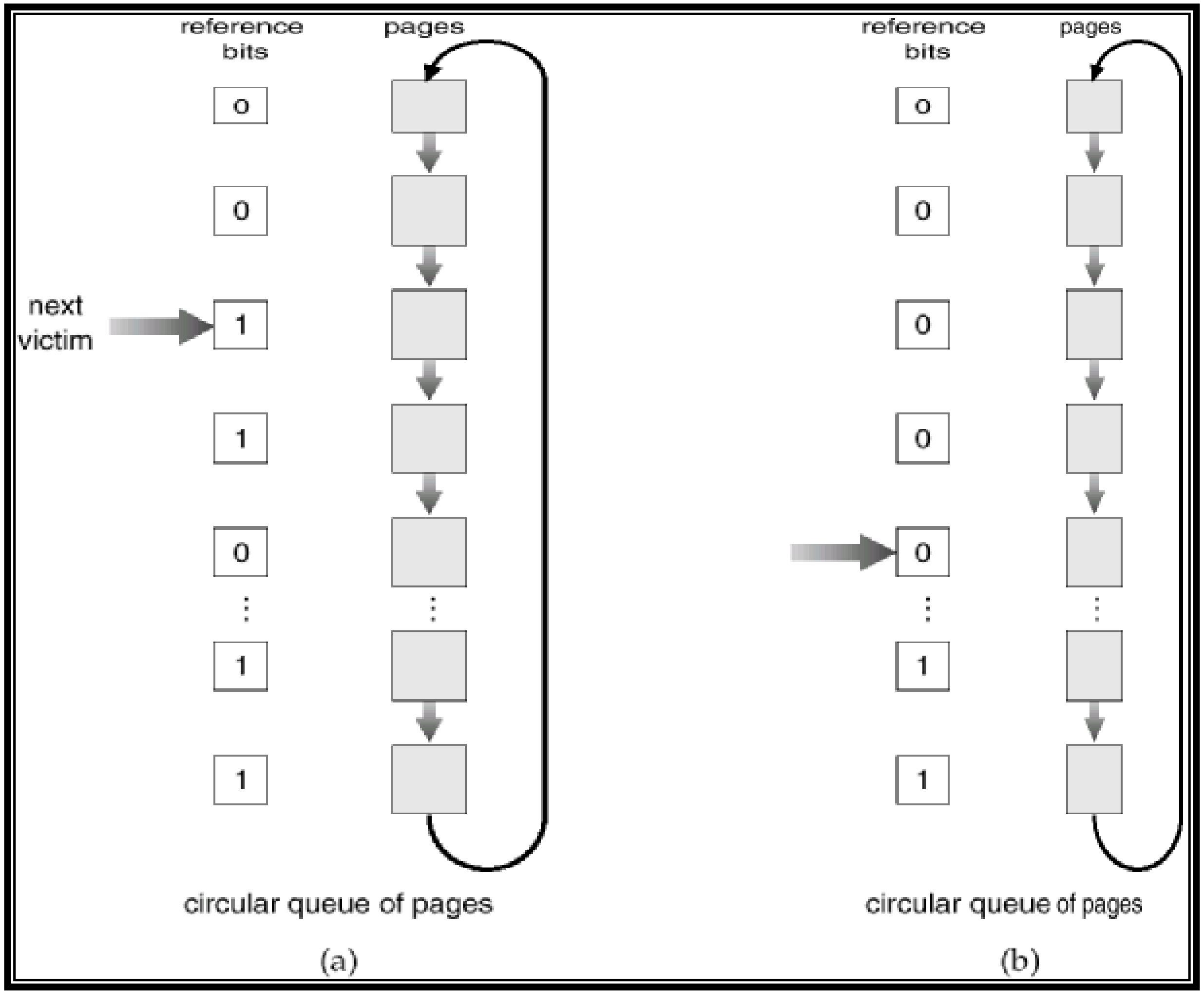
- Counter-based approach
  - Keep a counter for each page
  - At regular intervals, for every page, do:
    - If R = 0, increment the counter (hasn’t been used)
    - If R = 1, zero the counter (has been used)
    - Zero the R bit
  - The counter will contain the number of intervals since the last reference to the page
  - The page with largest counter is the least recently used

- Some architectures don’t have a reference bit
  - Can simulate reference bit using the valid bit to induce faults

Second Chance or LRU Clock

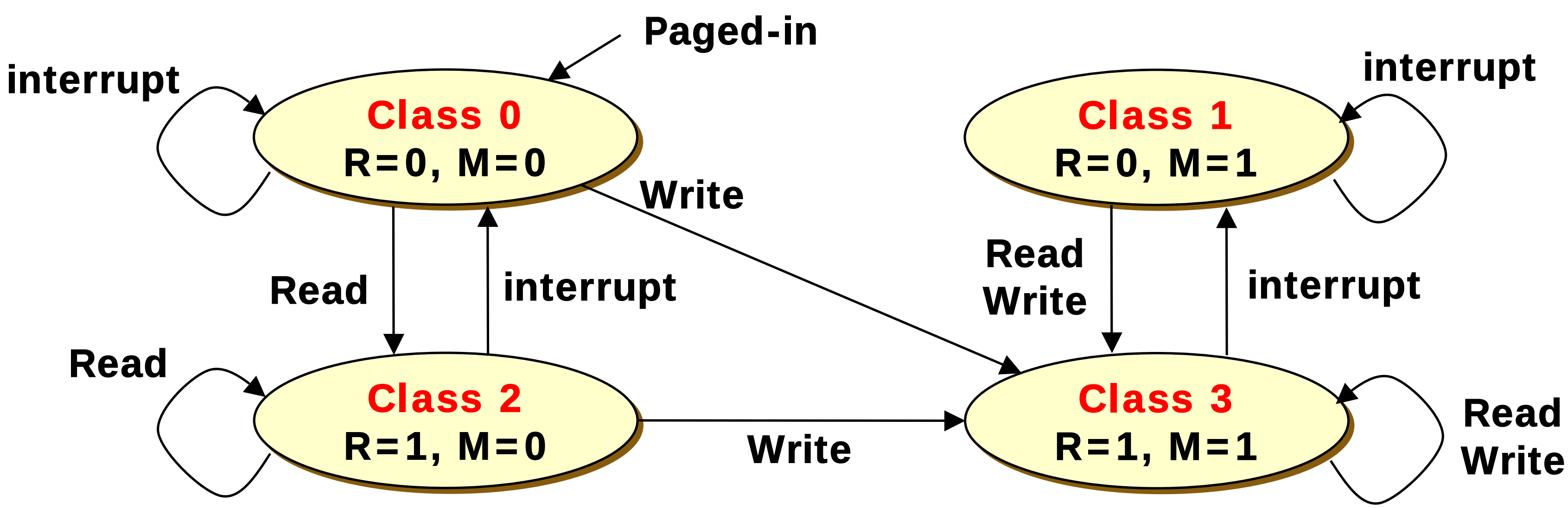
- FIFO with giving a second chance to a recently referenced page
- Arrange all of physical page frames in a big circle (clock)
- A clock hand is used to select a good LRU candidate
  - Sweep through the pages in circular order like a clock
  - If the R bit is off, it hasn't been used recently and we have a victim
  - If the R bit is on, turn it off and go to next page
- Arm moves quickly when pages are needed
  - Low overhead if we have plenty of memory
  - If memory is large, "accuracy" of information degrades

Second-Chance (clock) Page-Replacement Algorithm



Not Recently Used

- NRU or enhanced second chance
  - Use R (reference) and M (modify) bits
    - Periodically, (e.g., on each clock interrupt), R is cleared, to distinguish pages that have not been referenced recently from those that have been



Not Recently Used (Cont'd)

- Algorithm
  - Removes a page at random from the lowest numbered nonempty class
  - It is better to remove a modified page that has not been referenced in at least one clock tick than a clean page that is in heavy use

- Advantages
  - Easy to understand
  - Moderately efficient to implement
  - Gives a performance that, while certainly not optimal, may be adequate

Counting Algorithms

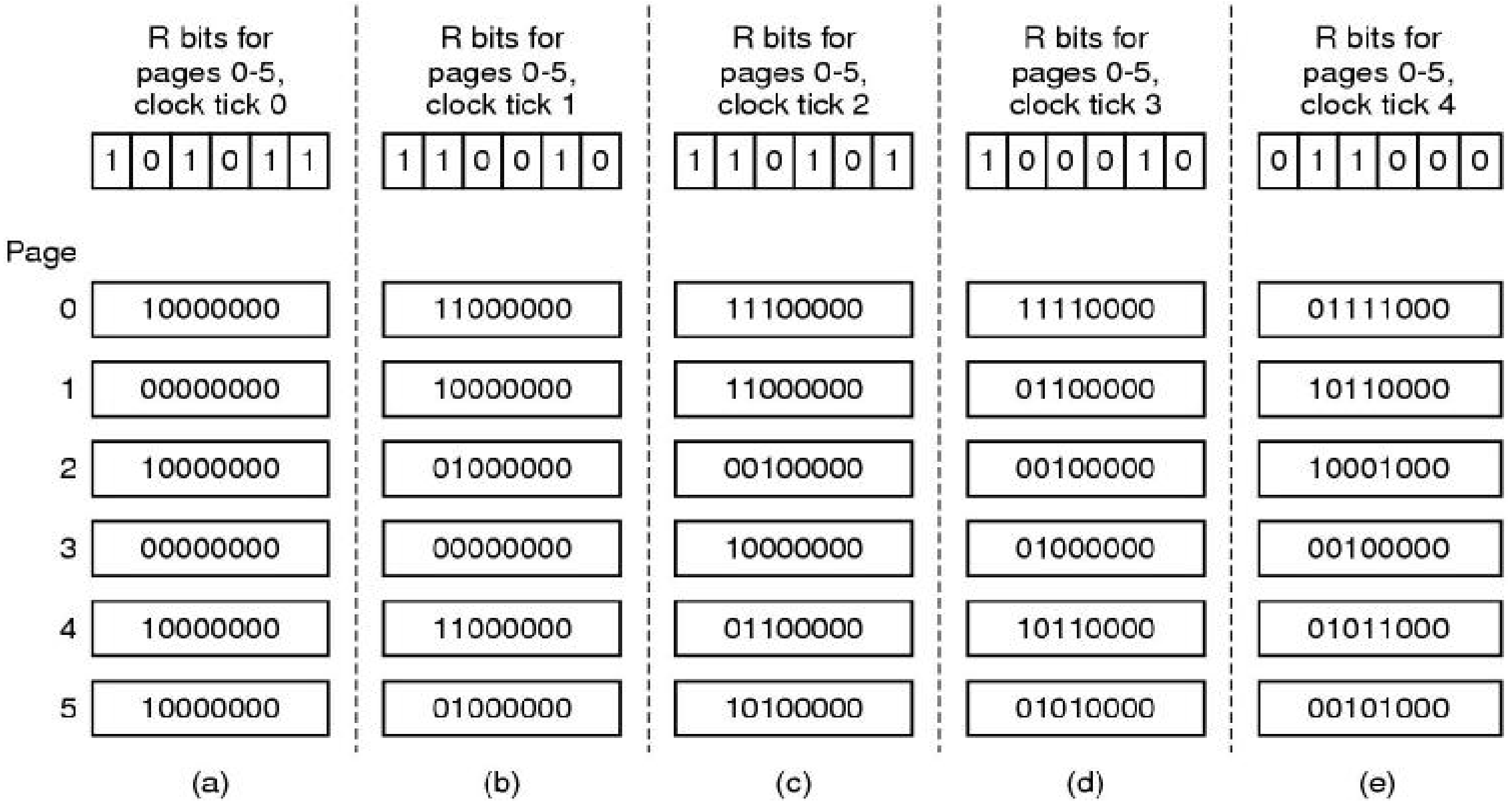
- Keep a counter of the number of references that have been made to each page
- LFU Algorithm: replaces page with smallest count
- MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

LFU

- Counting-based page replacement
  - A software counter is associated with each page
  - At each clock interrupt, for each page, the R bit is added to the counter
    - The counters denote how often each page has been referenced

- Least Frequently Used (LFU)
  - The page with the smallest count will be replaced
  - Cf) Most frequently used (MFU) page replacement
    - The page with the largest count will be replaced
    - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used
  - It never forgets anything
    - A page may be heavily used during the initial phase of a process, but then is never used again

- Aging
  - The counters are shifted right by 1 bit before the R bit is added to the leftmost



Allocation of Frames

- Each process needs **minimum** number of pages

- 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**

- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages
- Proportional allocation – Allocate according to the size of process
  - $s_i = \text{size of process } p_i$
  - $S = \sum s_i$
  - $m = \text{total number of frames}$
  - $a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

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

Global vs. Local Allocation

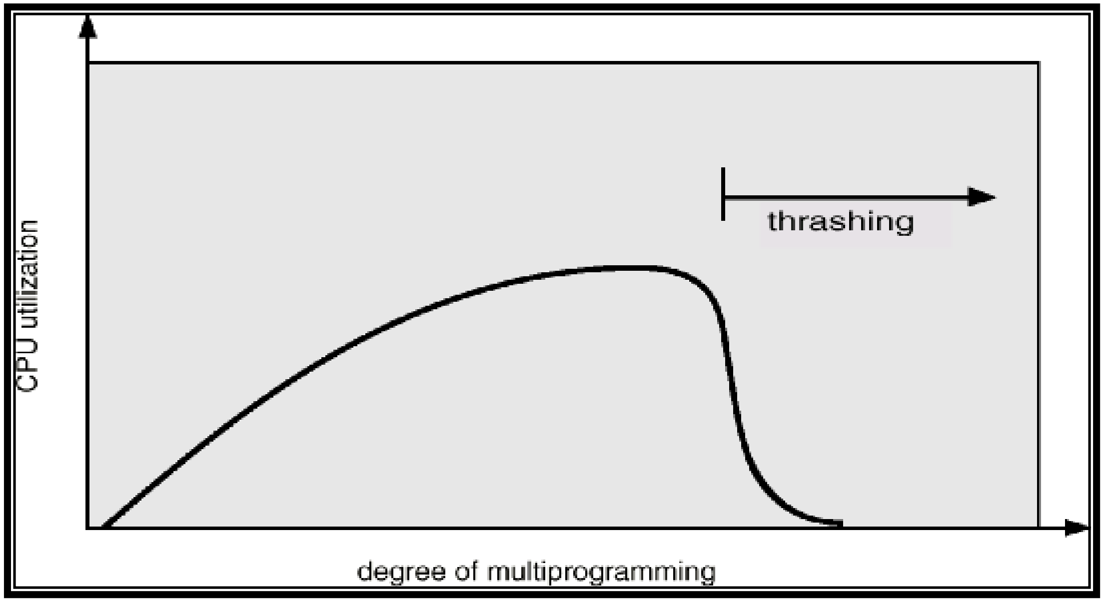
- **Global** replacement
  - Process selects a replacement frame from the set of all frames
  - One process can take a frame from another

- **Local** replacement
  - Each process selects from only its own set of allocated frames

Thrashing

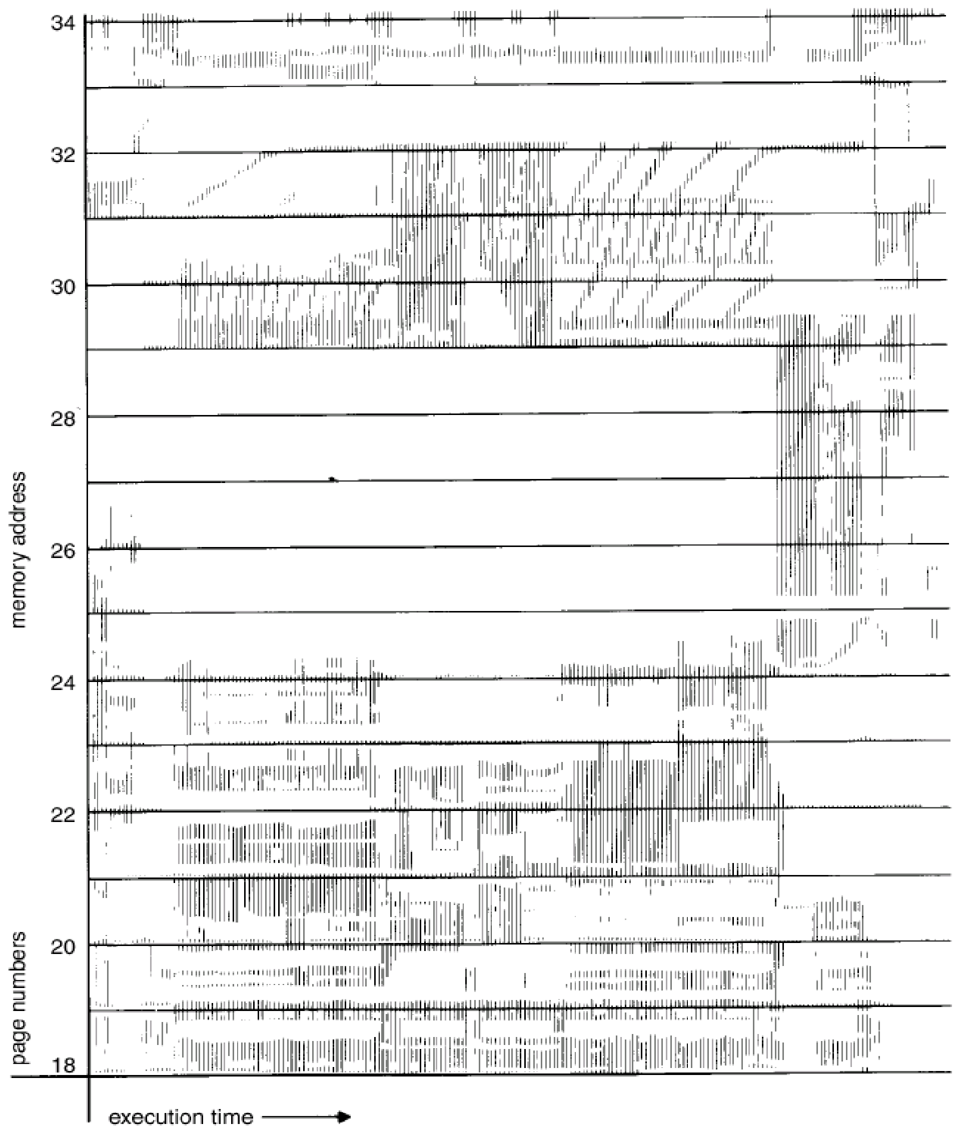
- If a process does not have “enough” pages, the page-fault rate is very high
- This leads to:
  - Low CPU utilization
  - Operating system thinks that it needs to increase the degree of multiprogramming
  - Another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing



- Why does paging work? à Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  - Σ size of locality > total memory size

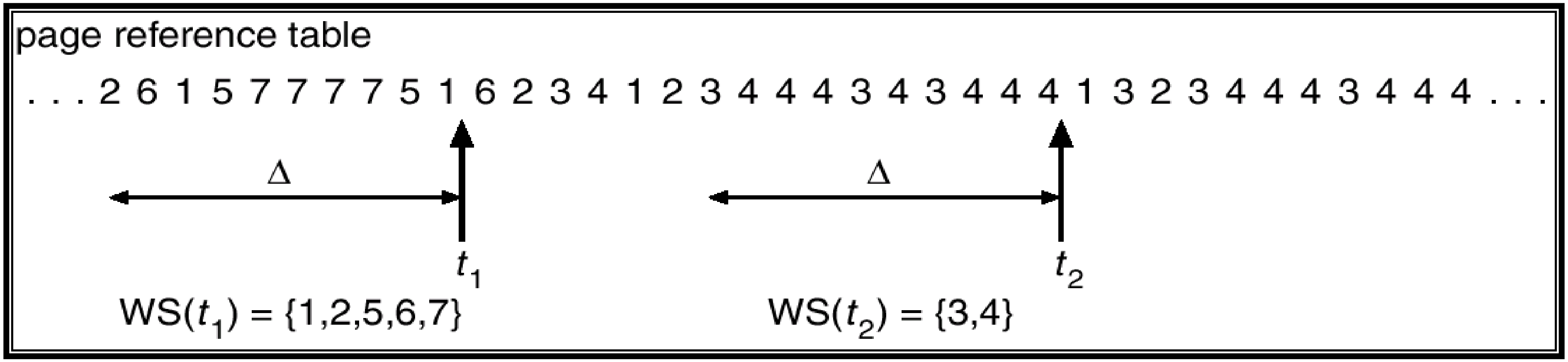
Locality In A Memory-Reference Pattern



Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  - Example: 10,000 instruction
- WSS_i (Working Set Size 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
- if $D > m \Rightarrow$ Thrashing
- Policy if D > m, then suspend one of the processes

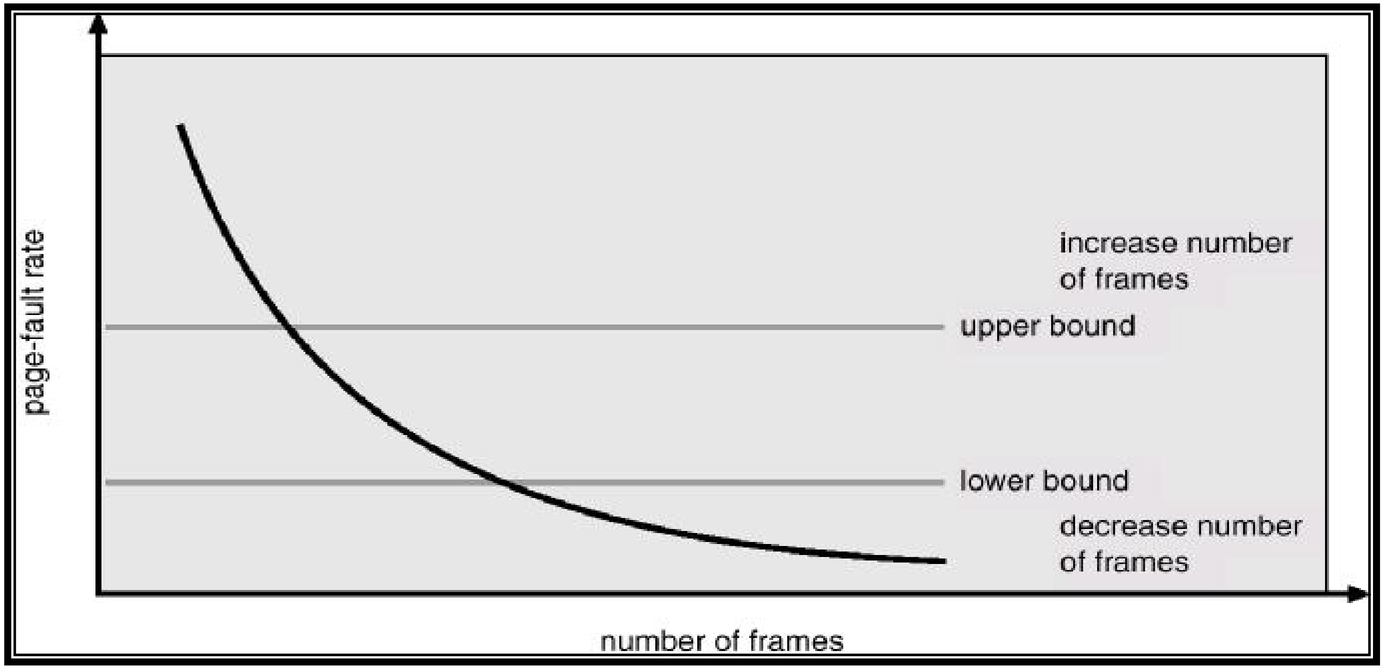
Working-Set Model (Cont'd)



Keeping Track of the Working Set

- 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
  - 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

Page-Fault Frequency Scheme



- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

Other Considerations

- Prepaging

- Page size selection
  - Fragmentation
  - Table size
  - I/O overhead
  - Locality

Other Considerations (Cont'd)

- 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

Increasing the Size of the TLB

- 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 Considerations (Cont'd)

- Program structure
  - **int A[][] = new int[1024][1024];**
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
        A[i,j] = 0;
```

    1024 x 1024 page faults

  - Program 2

```
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
        A[i,j] = 0;
```

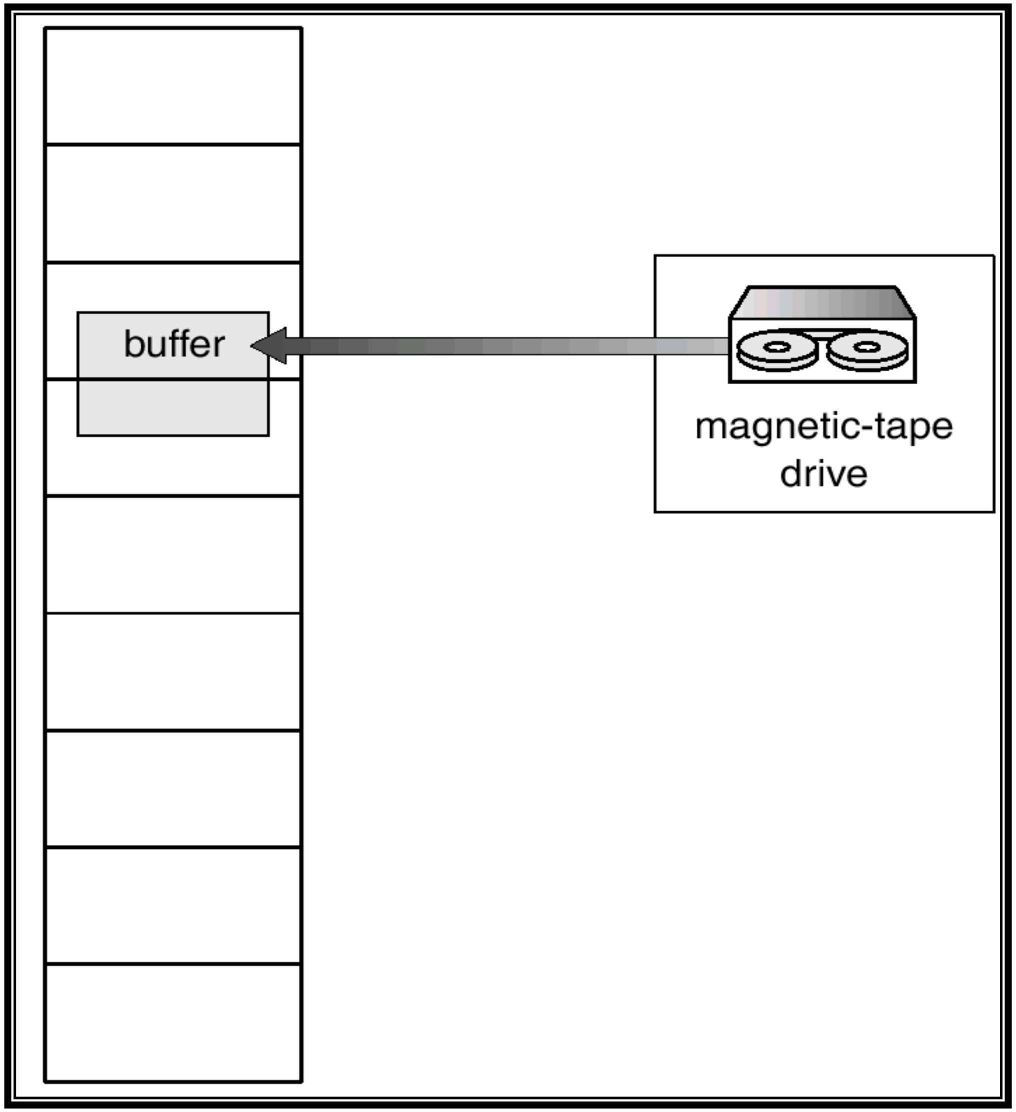
    1024 page faults

Other Considerations (Cont'd)

- 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

Reason Why Frames Used For I/O Must Be In Memory



Operating System Examples

- Windows NT
- Solaris 2

Windows NT

- 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

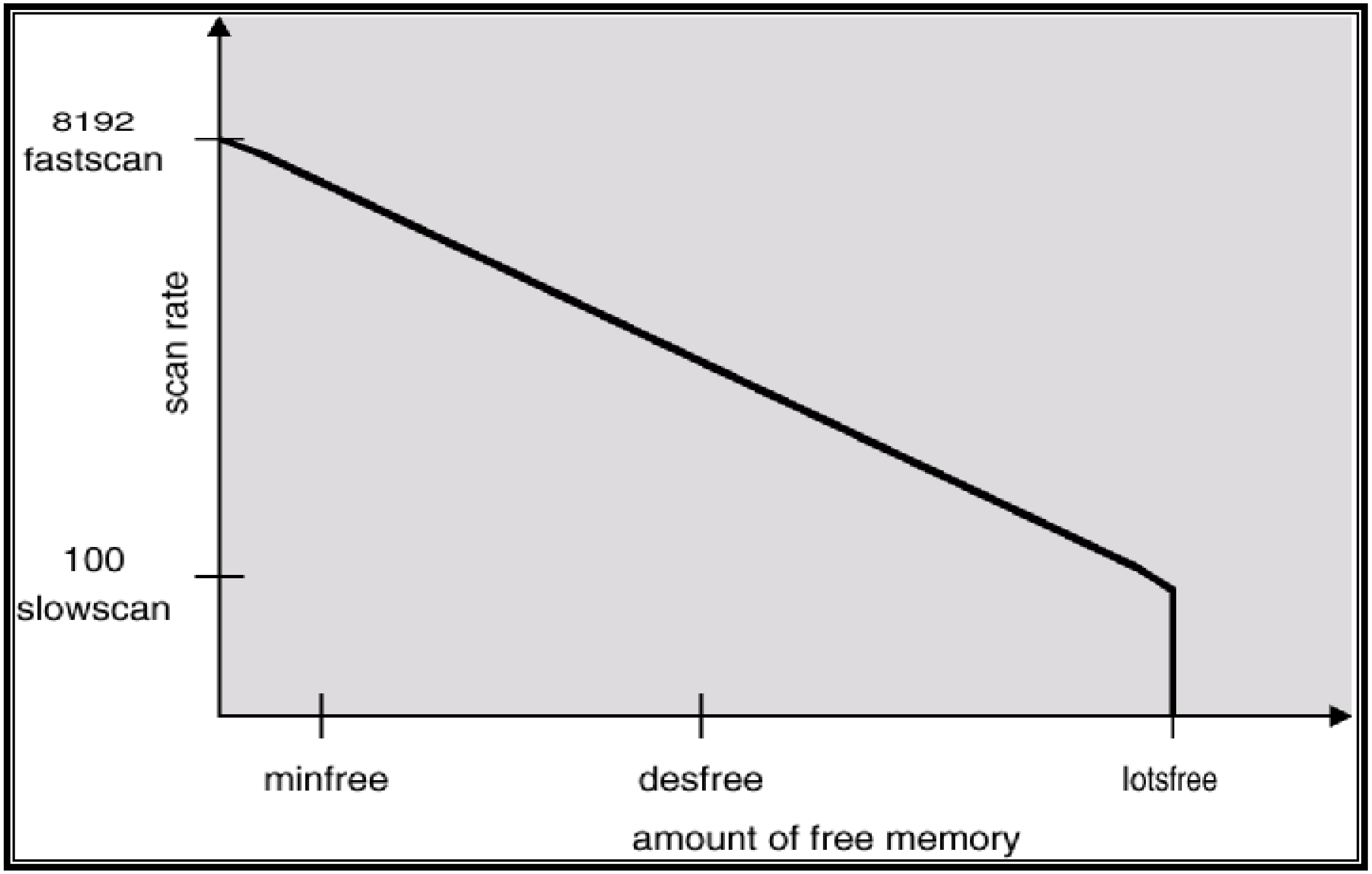
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory

- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Solaris 2

- Maintains a list of free pages to assign faulting processes
- **Lotsfree** – threshold parameter to begin paging
- Paging is peformed by *pageout* process
- Pageout scans pages using modified clock algorithm
  - Two-handed-clock algorithm (similar to the second-chance algorithm)
  - *handspread*
- **Scanrate** is the rate at which pages are scanned
  - This ranged from **slowscan** to **fastscan**
- Pageout is called more frequently depending upon the amount of free memory available

Solar Page Scanner



Virtual Memory

- Advantages
  - Separates user's logical memory from physical memory
    - Abstracts main memory into an extremely large, uniform array of storage
    - Frees programmers from the concerns of memory-storage limitations
  - Allows the execution of processes that may not be completely in memory
    - Programs can be larger than physical memory
    - More programs could be run at the same time
    - Less I/O would be needed to load or swap each user program into memory
  - Allows processes to easily share files and address spaces
  - Provides an efficient mechanism for protection and process creation

- Disadvantages
  - Performance!!!
    - In terms of time and space

Virtual Memory (Cont'd)

- Optimizations
  - Managing page tables (space)
  - Efficient Translation (TLBs) (time)
  - Demand paging (space)

- Advanced functionality
  - Sharing memory
  - Copy on write
  - Mapped files