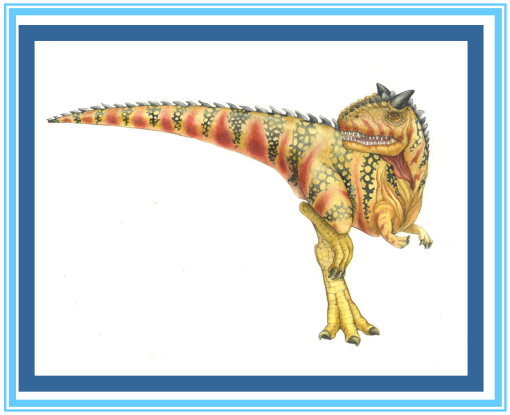
Chapter 9: Virtual Memory



Chapter 9: 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

Background

- Physical memory space is always not large enough for us
  - Our programs are getting larger and larger
  - We want to run as many as possible programs at the same time
- How to meet these requirements?

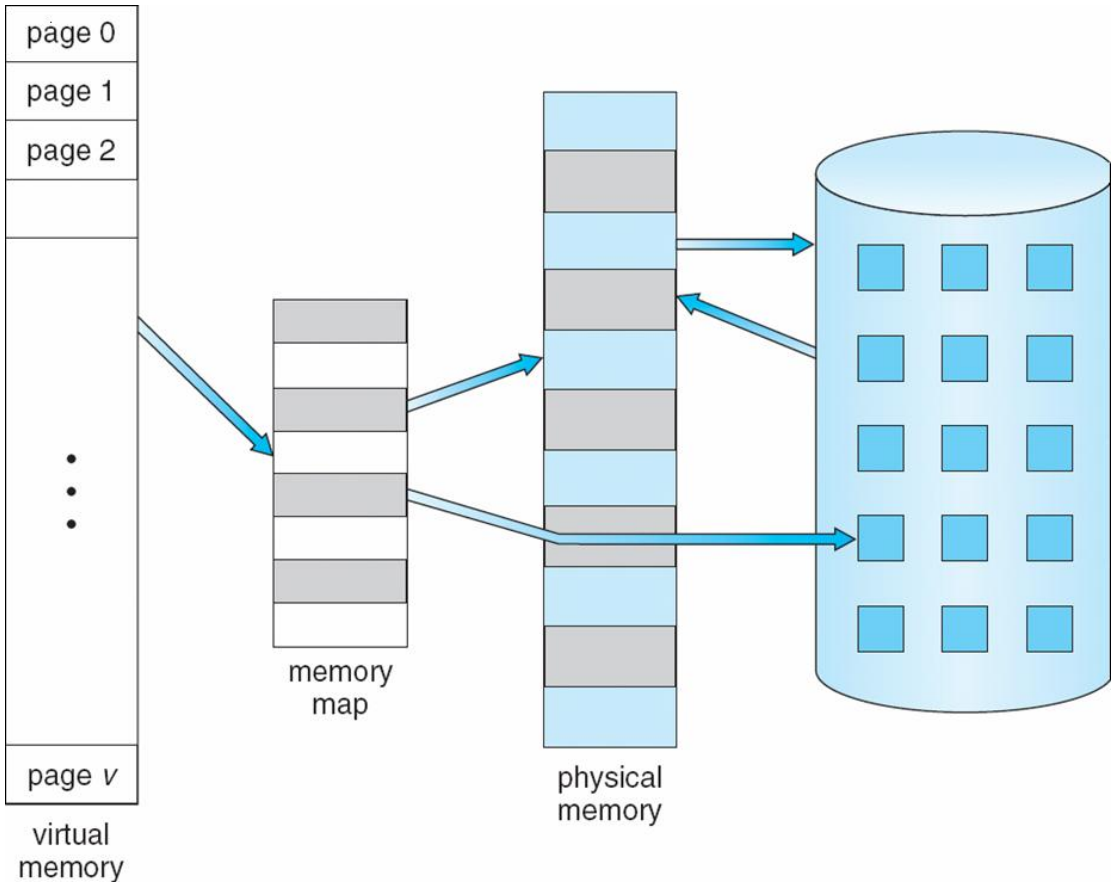
To provide an extremely large logical memory

- Abstract disk and memory as one large logical memory
- Separate logical memory from the physical memory
- Allow the execution of processes that are not completely in memory

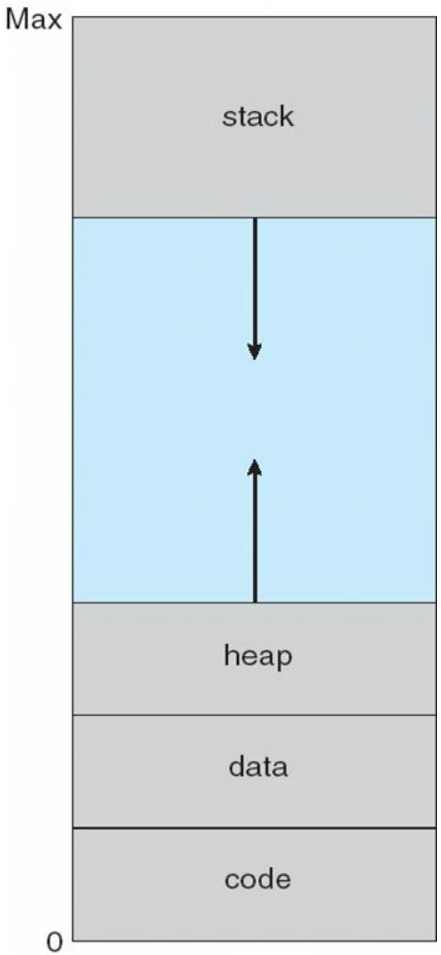
Background

- **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

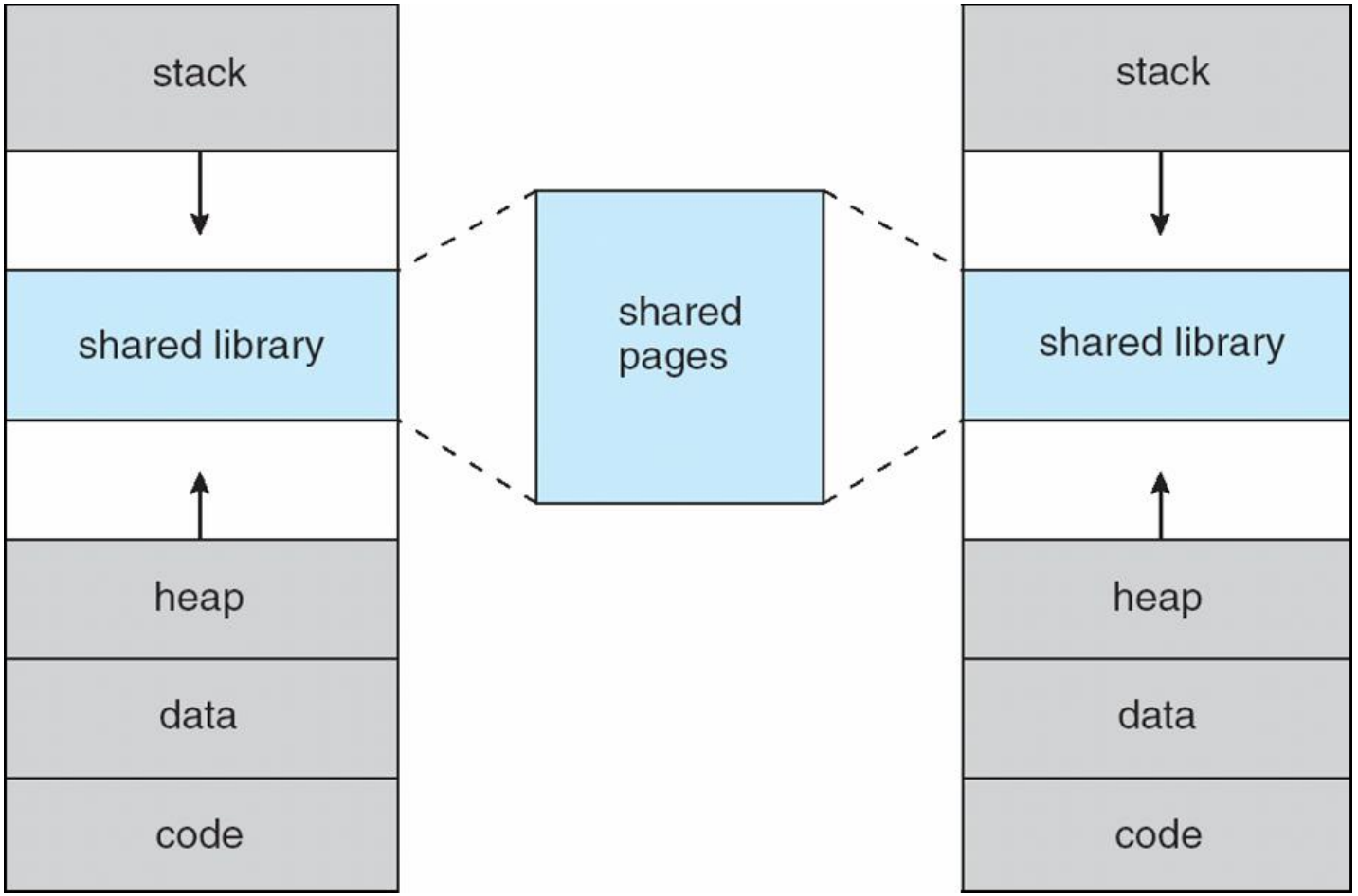
Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



Shared Library Using Virtual Memory



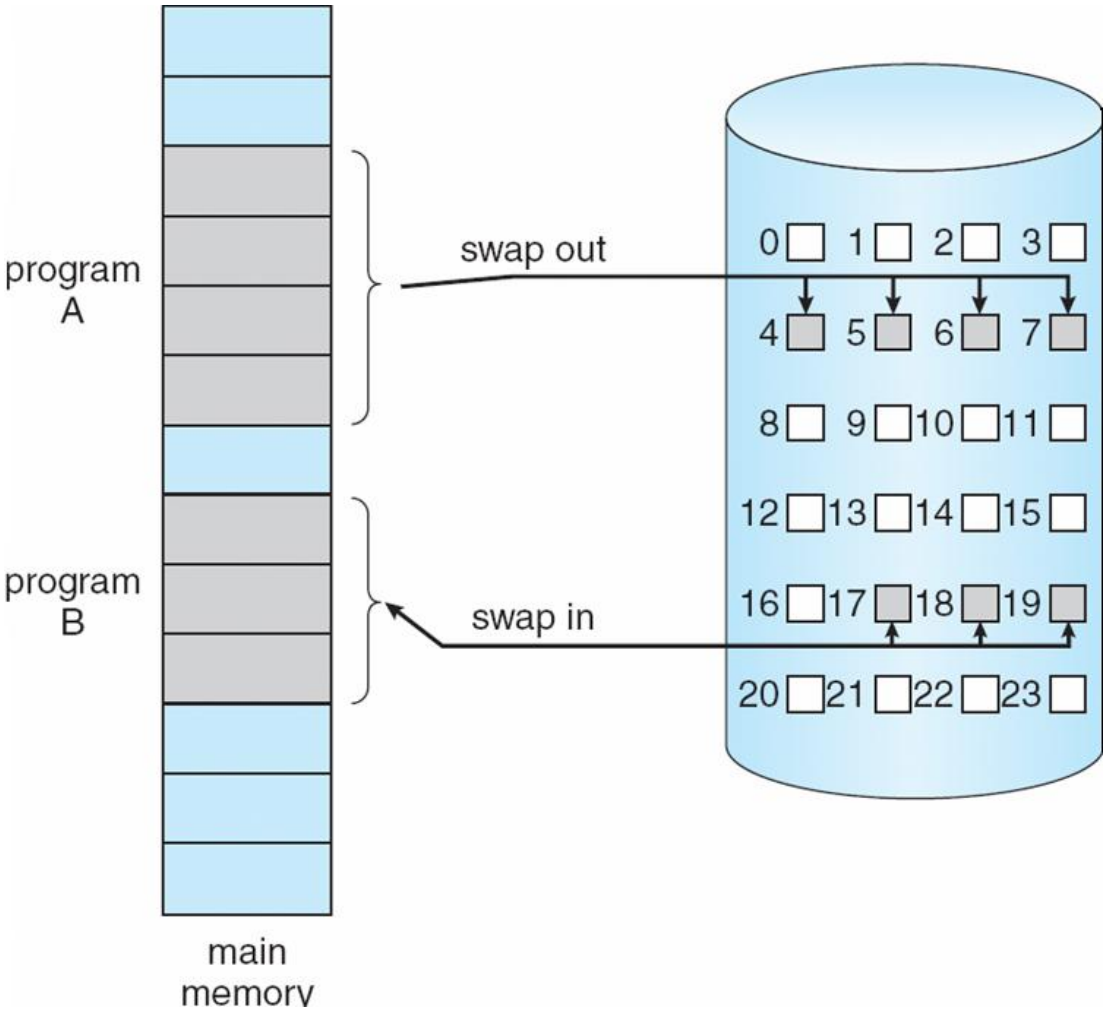
Demand Paging

- Pages are only loaded when they are demanded during program execution
- Demand paging= paging + page fault + page replacement
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

Demand Paging

- 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**

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

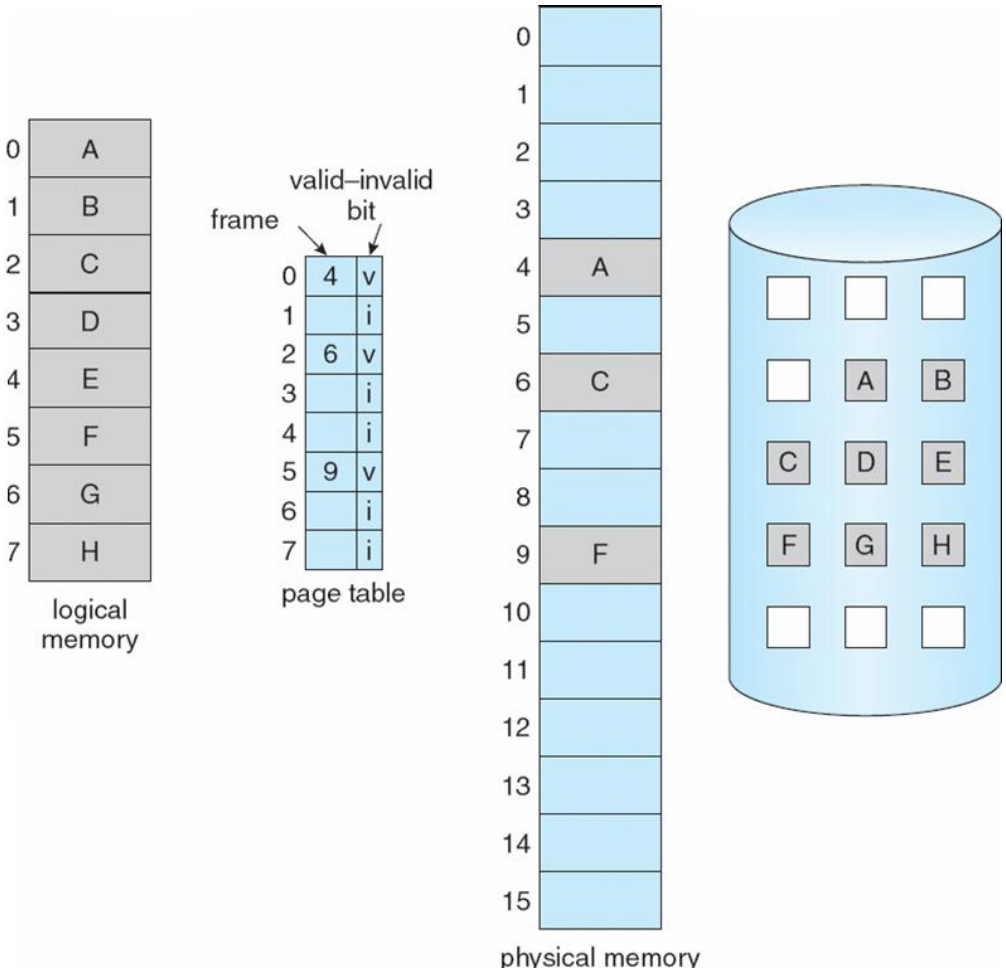
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

page table

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

Page Table When Some Pages Are Not in Main Memory



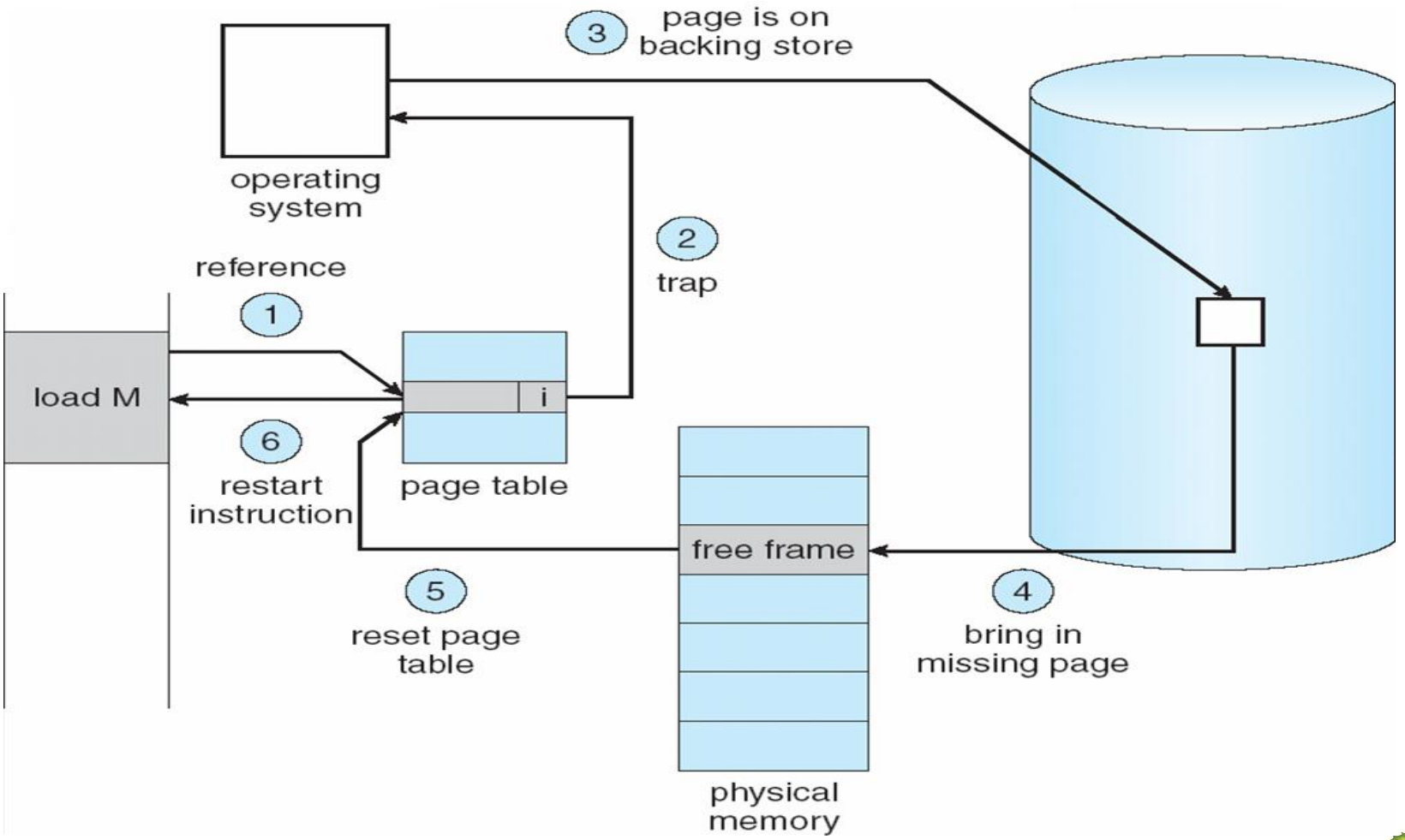
Page Fault

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

  page fault

1. Operating system needs to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



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)

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

Demand Paging Example

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

  = (1 – p) x 200 + p x 8,000,000

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

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!

Process Creation

- Virtual memory allows other benefits during process creation:

  - Copy-on-Write

  - Memory-Mapped Files (later)

Copy-on-Write

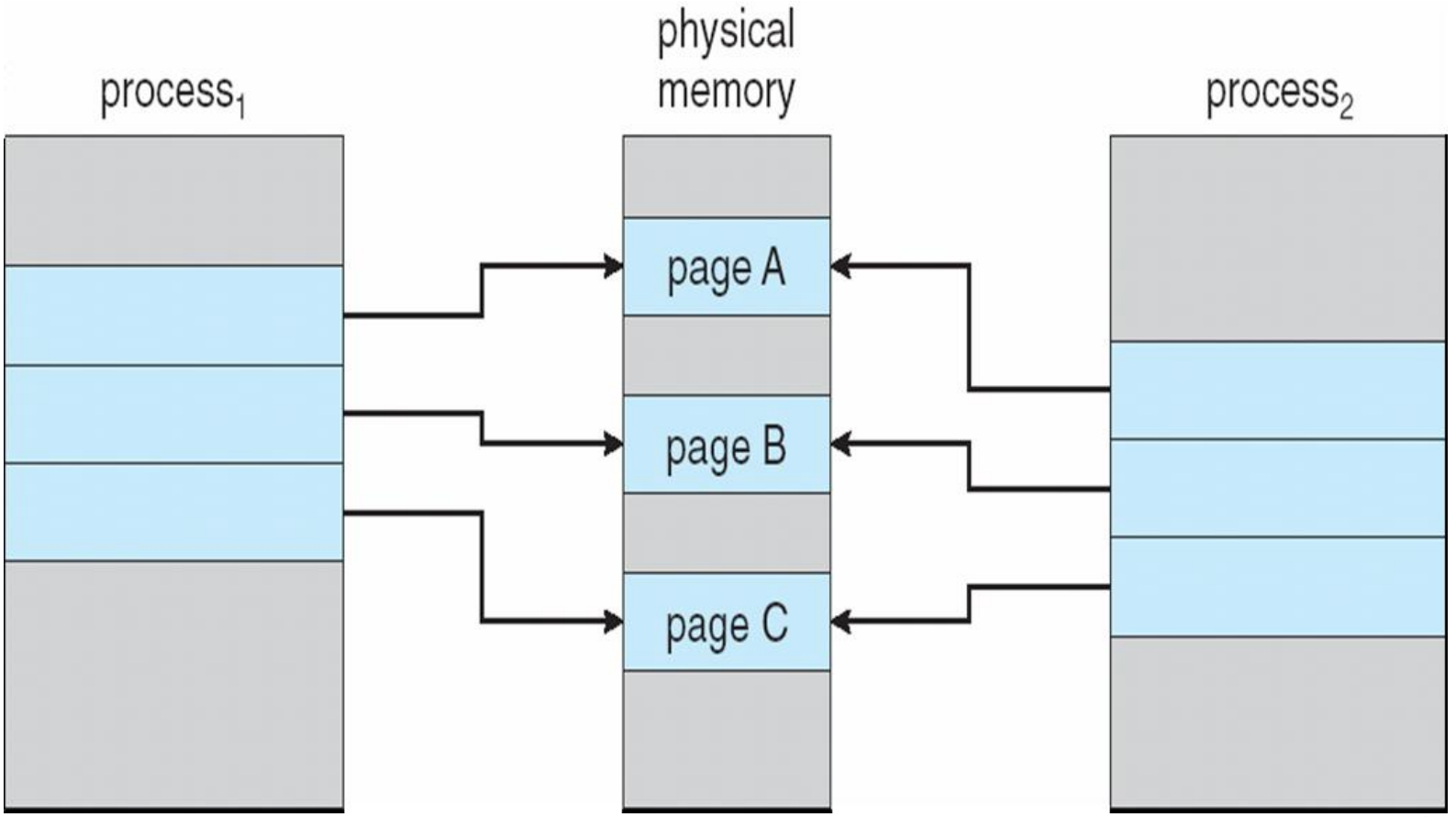
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

  If either process modifies a shared page, only then is the page copied

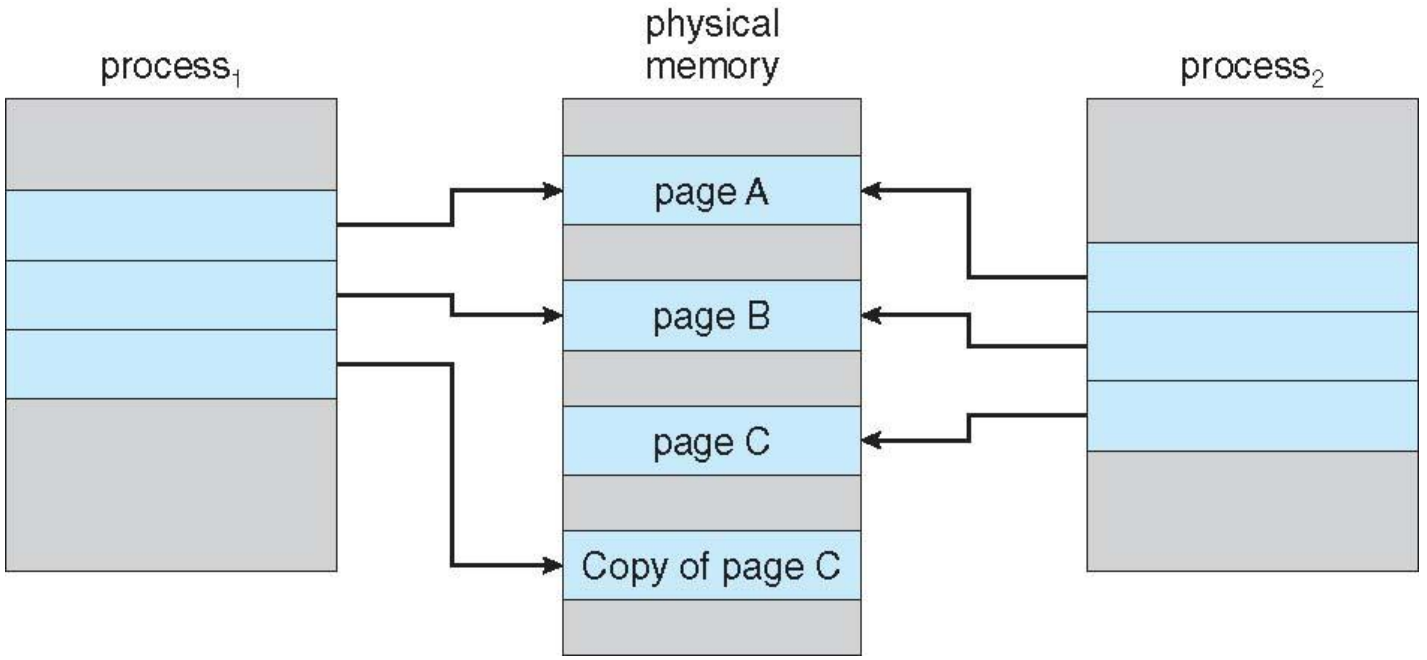
- COW allows more efficient process creation as only modified pages are copied

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

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



Page Replacement

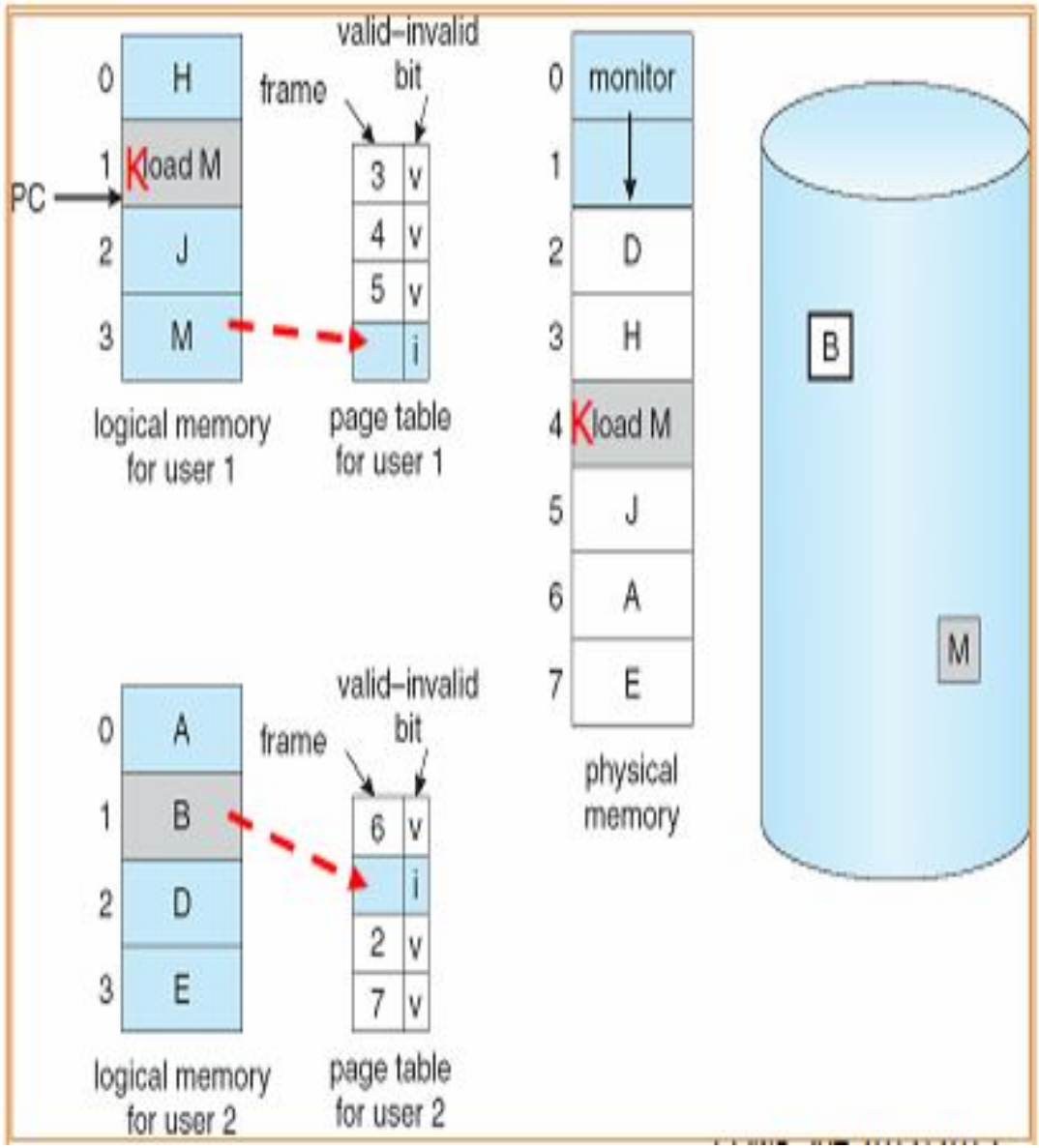
- Recall that to handle a page fault, a free frame is needed to hold the page to be swapped from disk into memory.
- Sooner or later, the system runs out of free frames.
- Unused frame in memory should be freed.
- If all frames are still in use, try to find some frame that is probably not used and remove it.
- This action of finding a frame (page) in memory to be removed in order to admit a newly needed page is called page replacement.
- Issues:
  - Which page is unlikely to be used and can be thrown away?
  - Could a process throw away pages of another process?
  - Should the removed page be saved back to the disk (some data have been modified)?

Page Replacement

- Example:

executing instruction in page 1 (i.e. K) of user1.

  - Instruction needs to access page 3 (i.e. M), which is not in memory.
  - All 6 pages, 3 for each user, are used up already.
  - We need to remove a page to free a frame to hold page M.



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

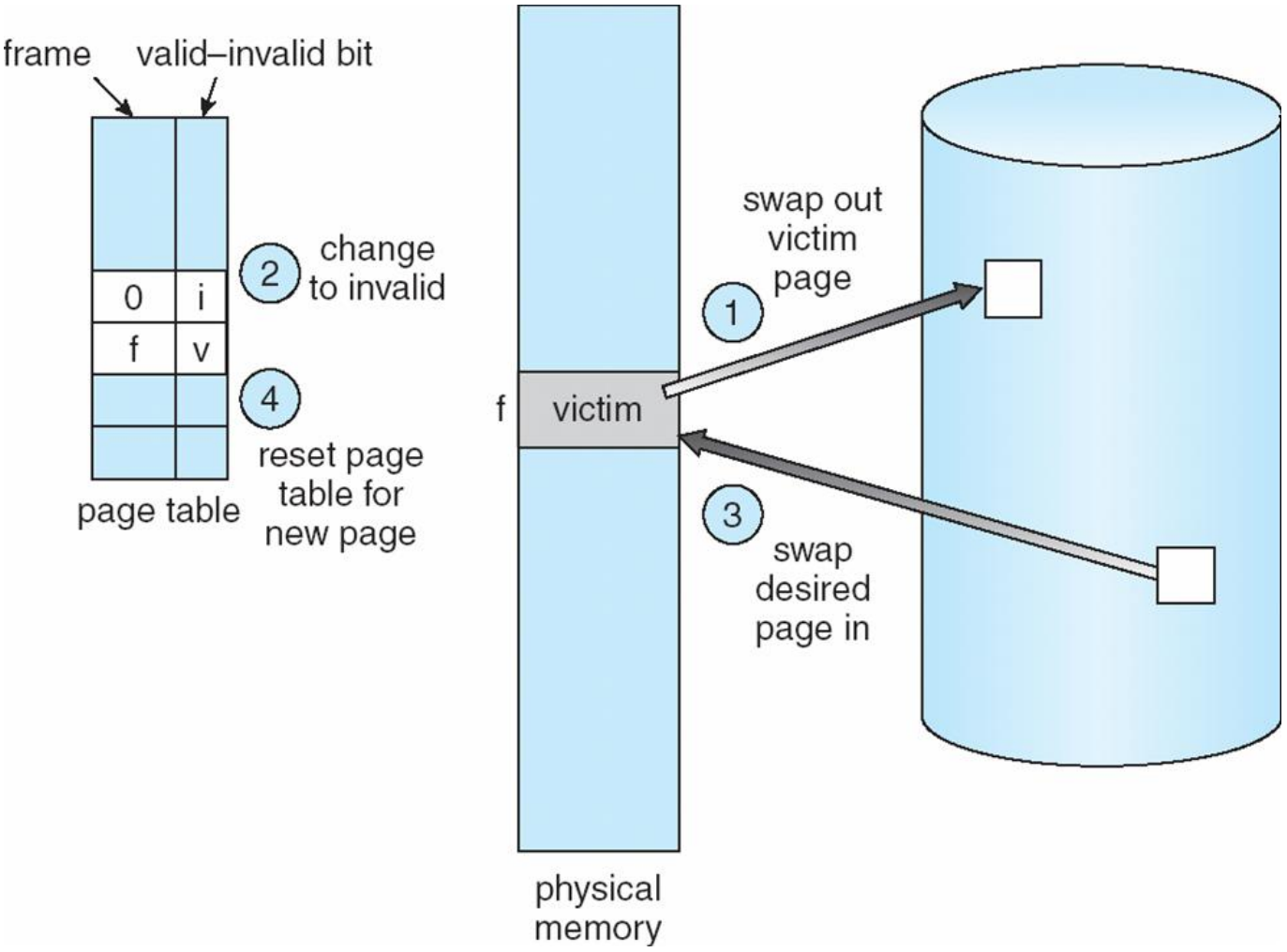
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
  - It is more effective to remove a page that has not been changed or modified, since that page does not need to be written back on the disk.
    - Program text represents read-only pages that require no saving.
    - Sometimes some data pages may not be modified at the moment of page replacement.
  - Use a modify or dirty bit to indicate whether a page has been modified.
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Basic Page Replacement

- Find the location of the desired page on disk
- 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
- Bring the desired page into the (newly) free frame
- Update the page table for both the victim and the new page.
- Return from page fault handler and resume the user process generating the page fault.

Page Replacement



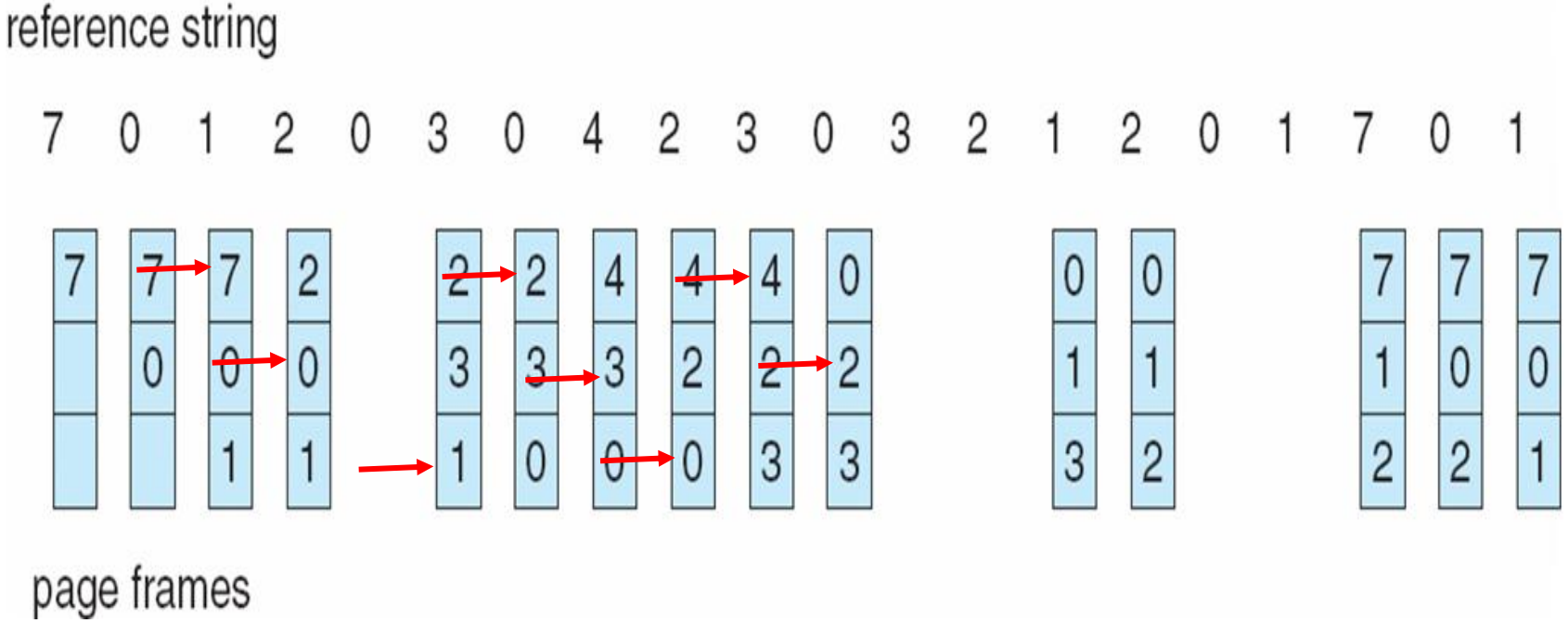
Page Replacement Algorithms

- 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

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

FIFO Page Replacement

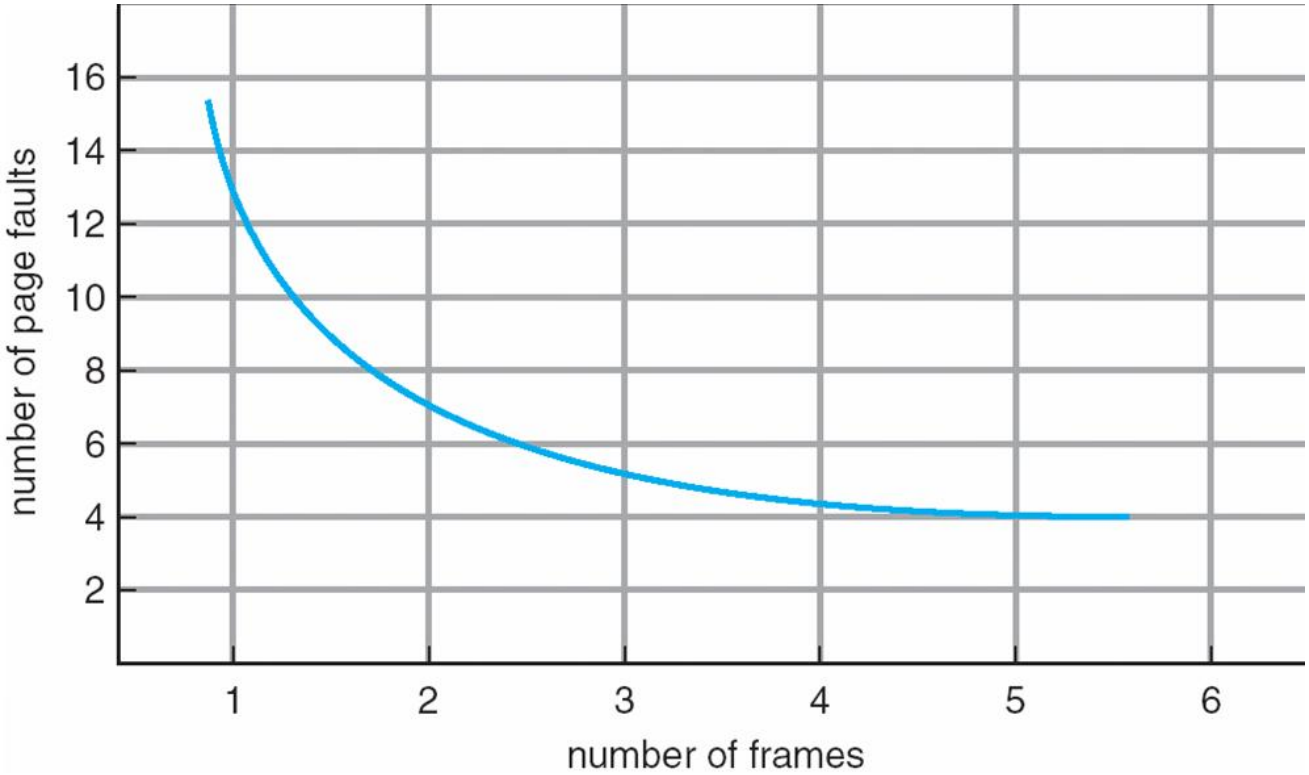
If a page comes in earlier than another page, the earlier page will be replaced before the later page.



Page fault: 15

Page replacement: 12

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

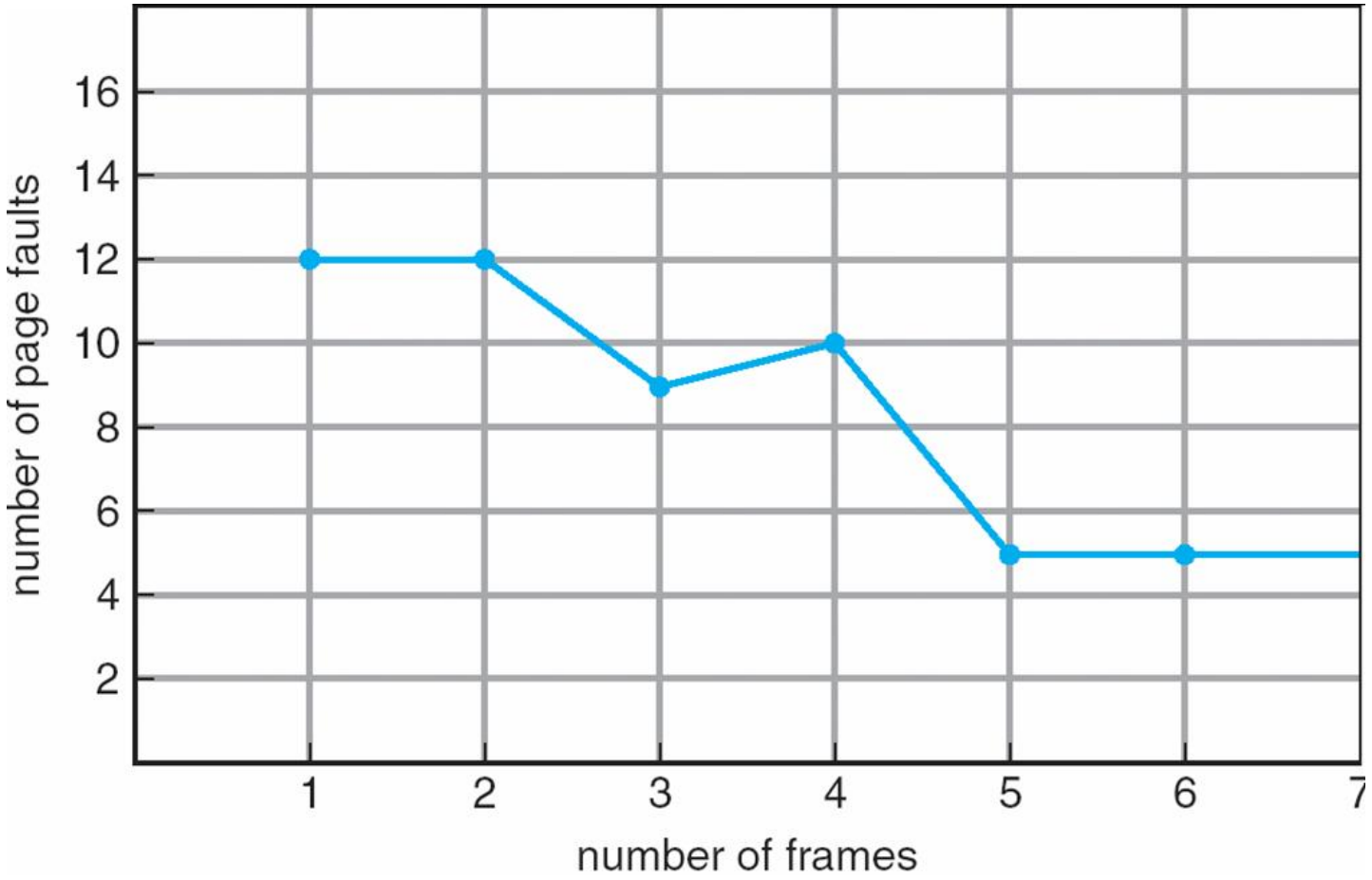
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

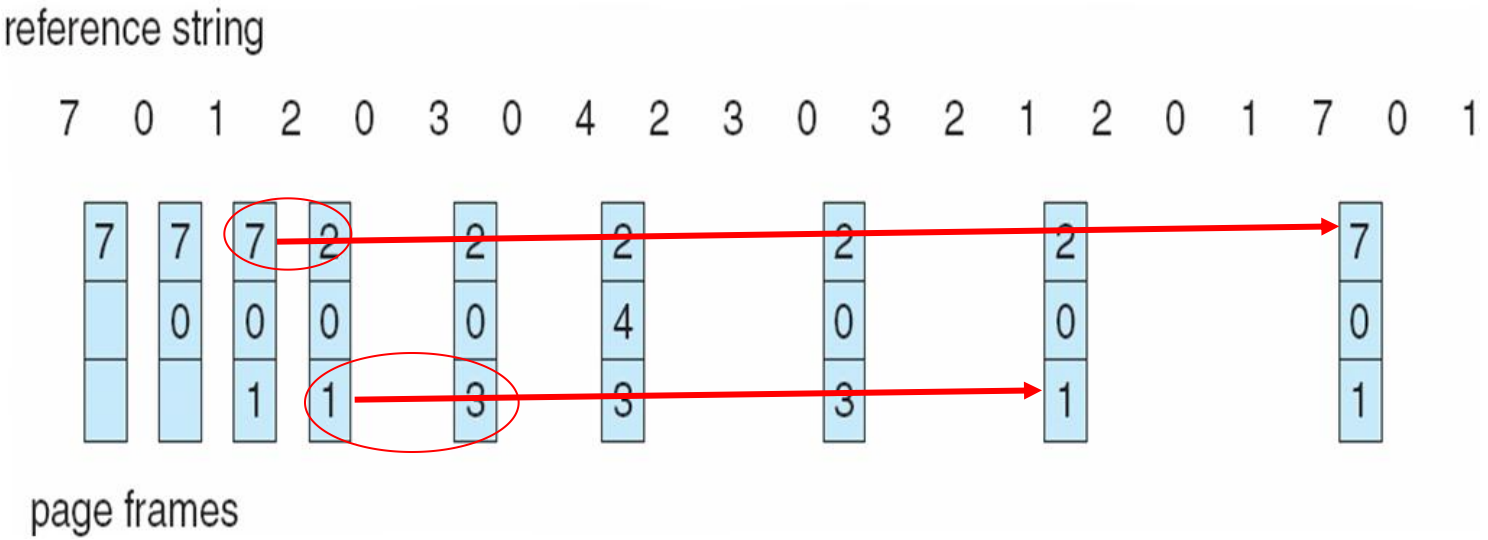
1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

FIFO Illustrating Belady's Anomaly



Optimal Page Replacement

Look into the future, and replace the page that will not be needed in the furthest future.

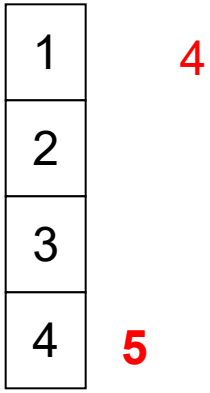


Page fault: 9
Page replacement: 6

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

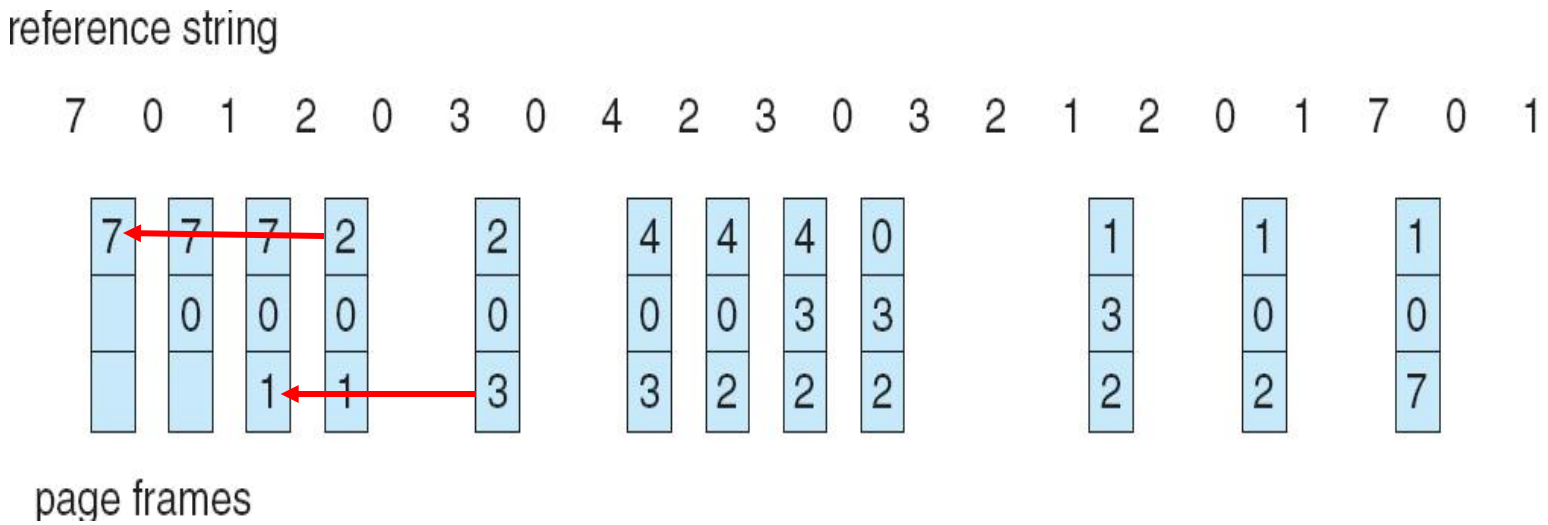


6 page faults

- How do you know this?
- Used for measuring how well your algorithm performs

Least Recently Used (LRU) Algorithm

Look into the past, and replace the page that has not been used for the longest time.

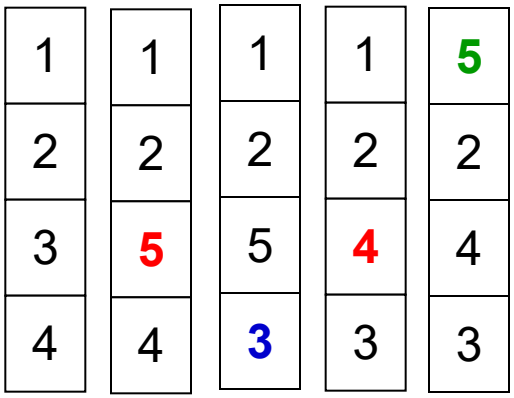


Page fault: 12

Page replacement: 9

Least Recently Used (LRU) Algorithm

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



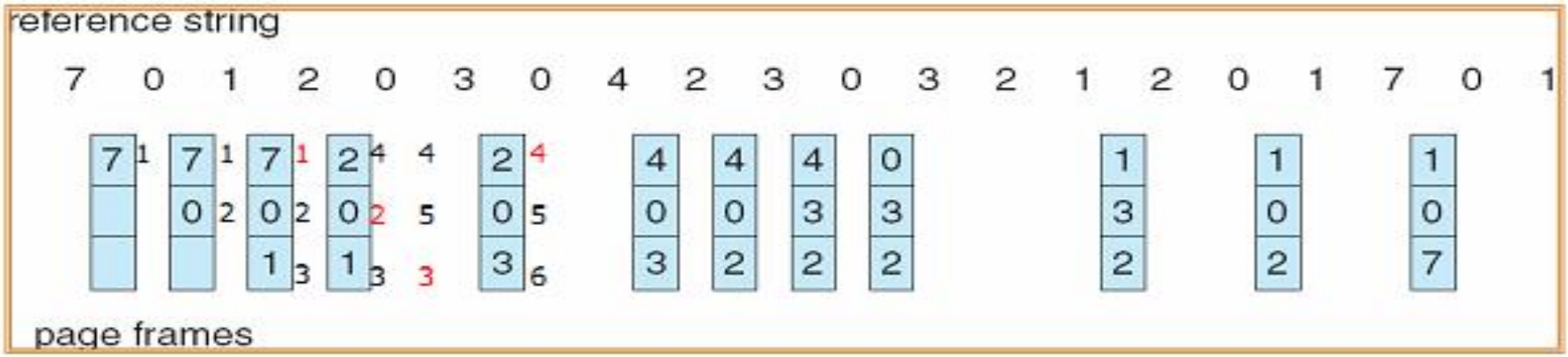
- 8 page faults

LRU Algorithm (Cont.)

- The problem of LRU implementation is to determine an order for the frames defined by the time of last use.
- How to implement LRU?
  - Counter implementation
  - Stack implementation
  - Approximation implementation

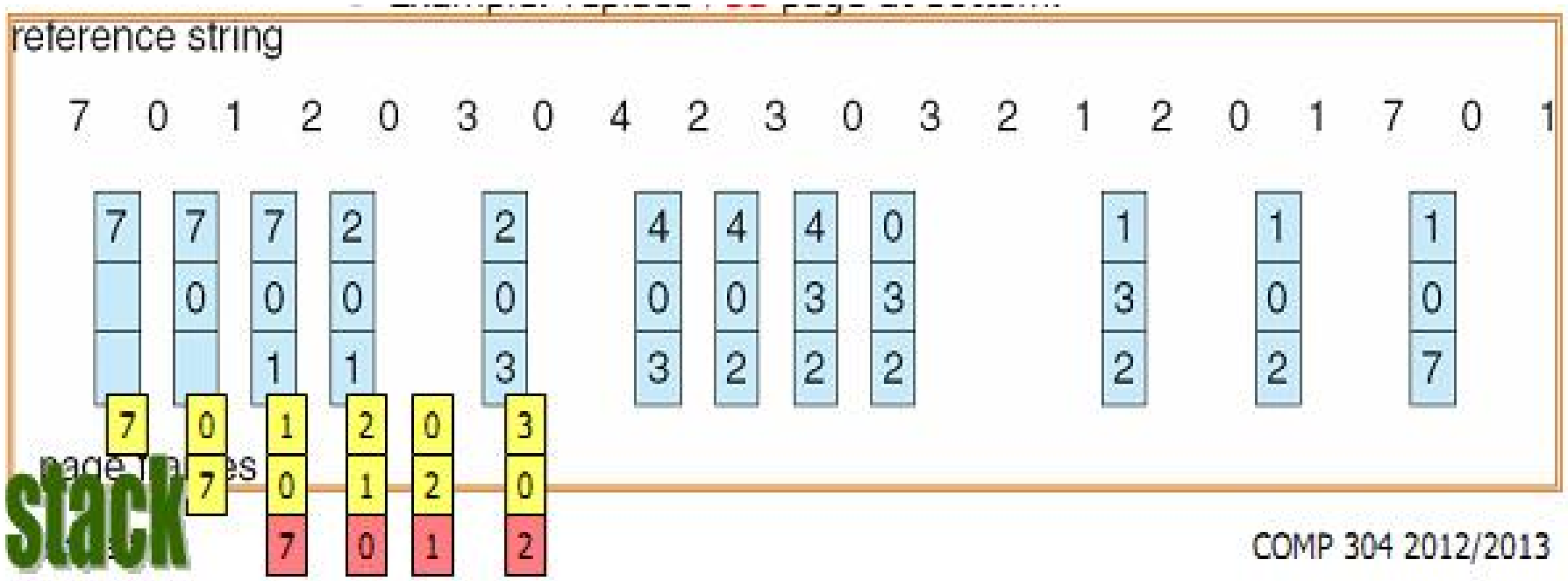
LRU Algorithm (Cont.)

- Counter implementation
  - Associate with each page-table entry a time-of-use field
  - Add to the CPU a logical clock or counter which is incremented for every memory reference
  - Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page table entry for that page
  - We replace the page with the smallest time value.

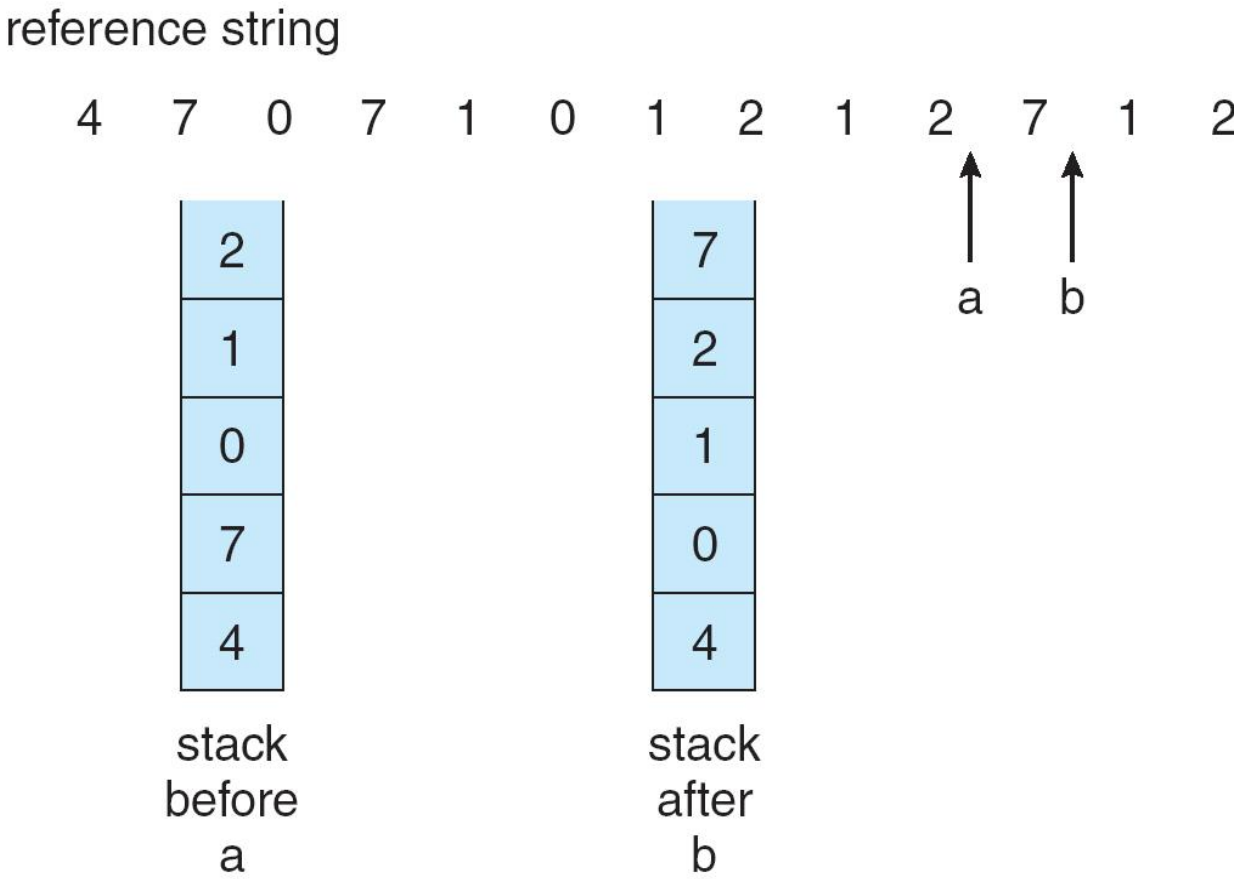


LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced: move it to the top
  - So, the most recently used page is always at the top of the stack
  - No search for replacement



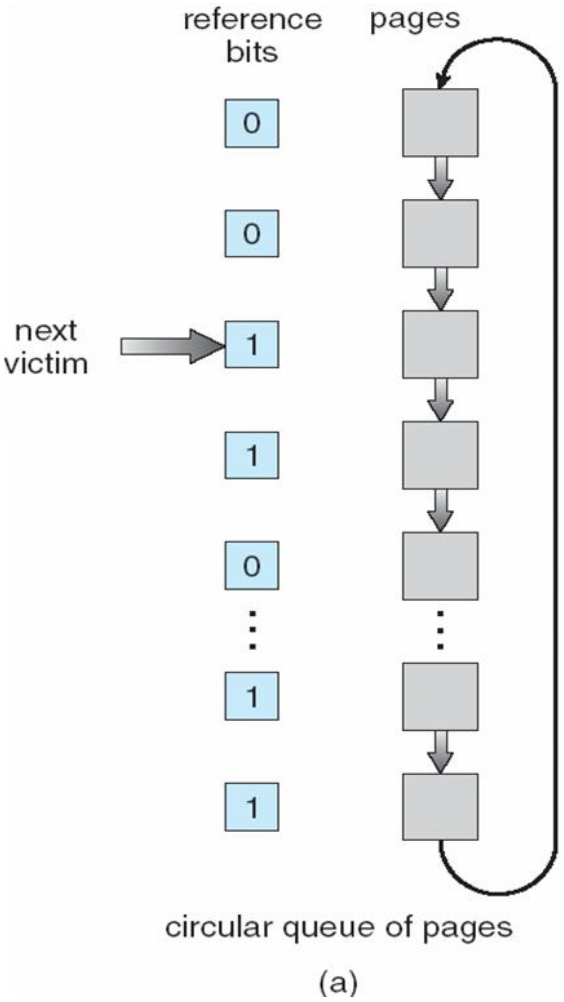
Use Of A Stack to Record The Most Recent Page References

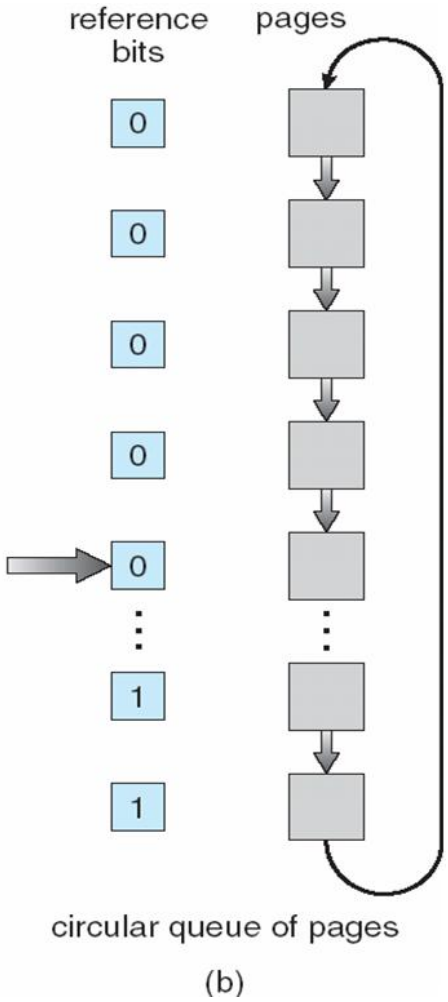


LRU Approximation Algorithms

- 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

Second-Chance (clock) Page-Replacement Algorithm





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

Allocation of Frames

- Each process needs *minimum* number of pages
- *The minimum* number of frames is defined by the computer architecture
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

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

$$m = 64$$
$$s_i = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

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

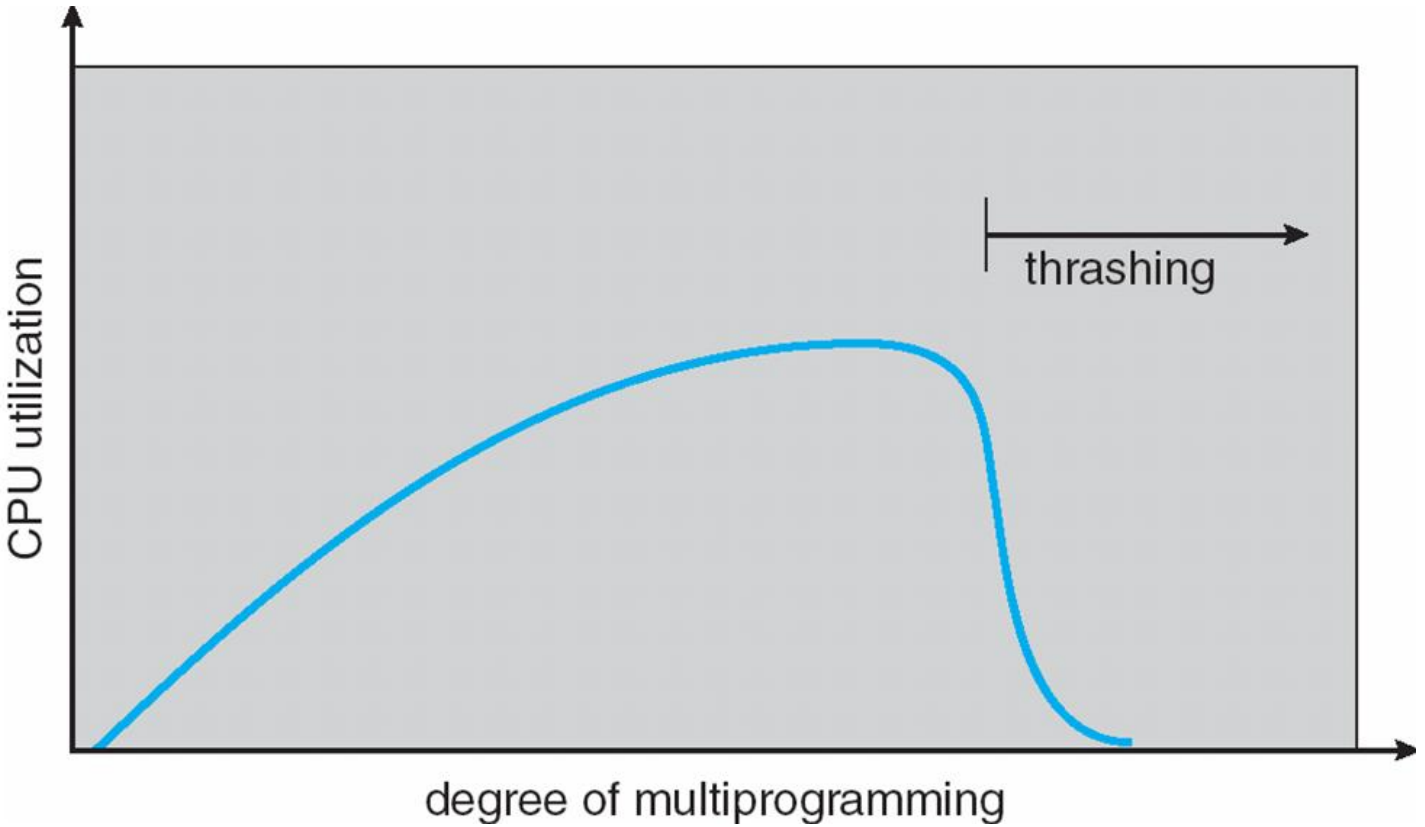
Global vs. Local Allocation

- **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 (Cont.)

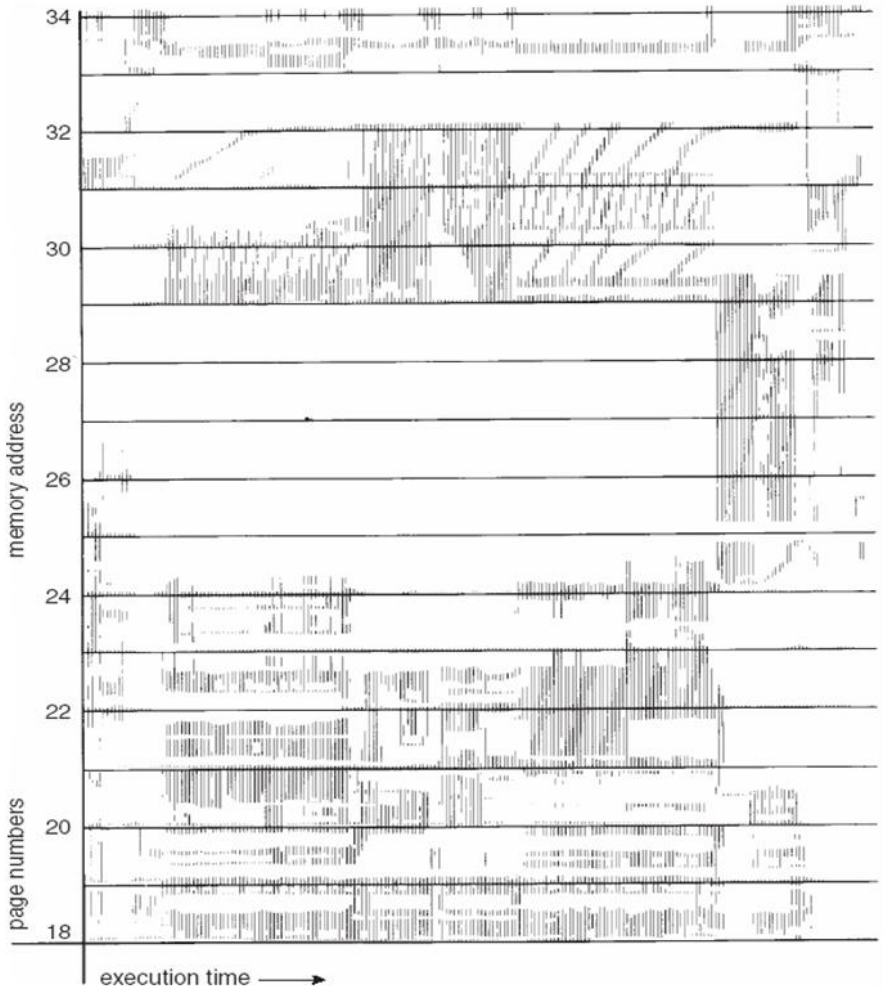


Demand Paging and Thrashing

- 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

Locality In A Memory-Reference Pattern

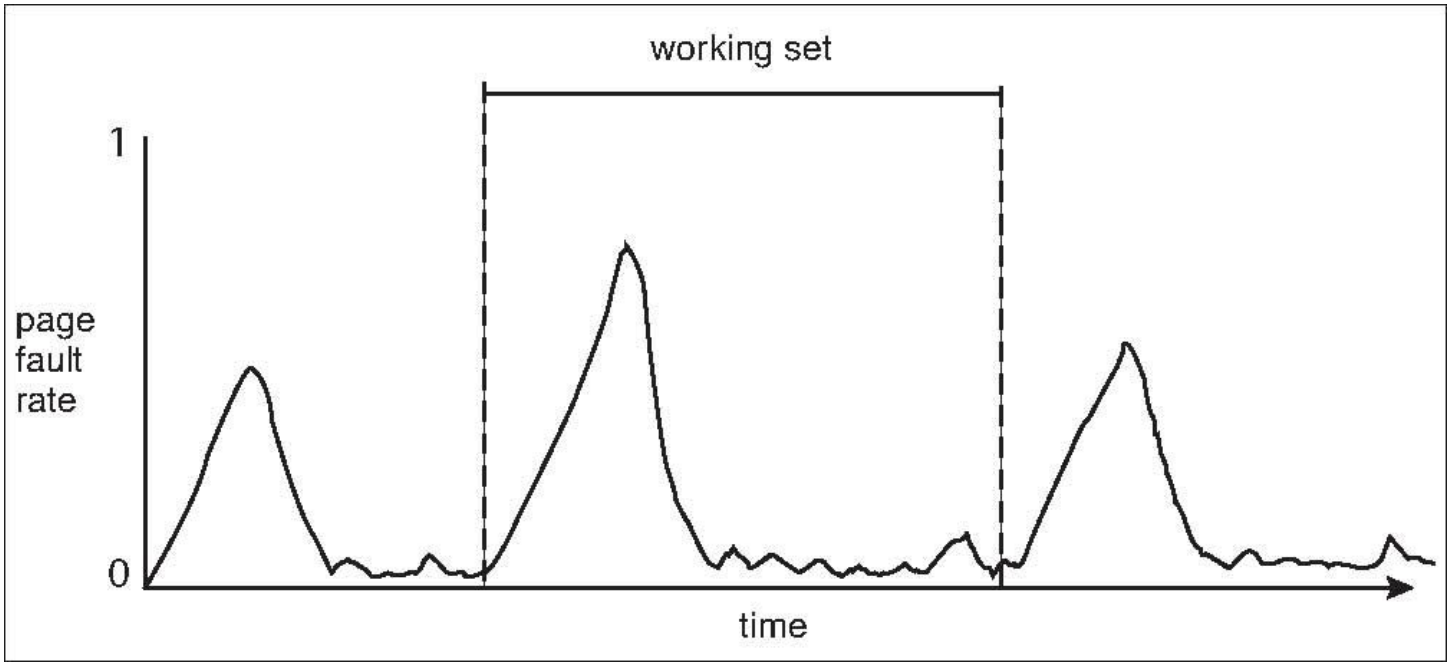


Working-Set Model

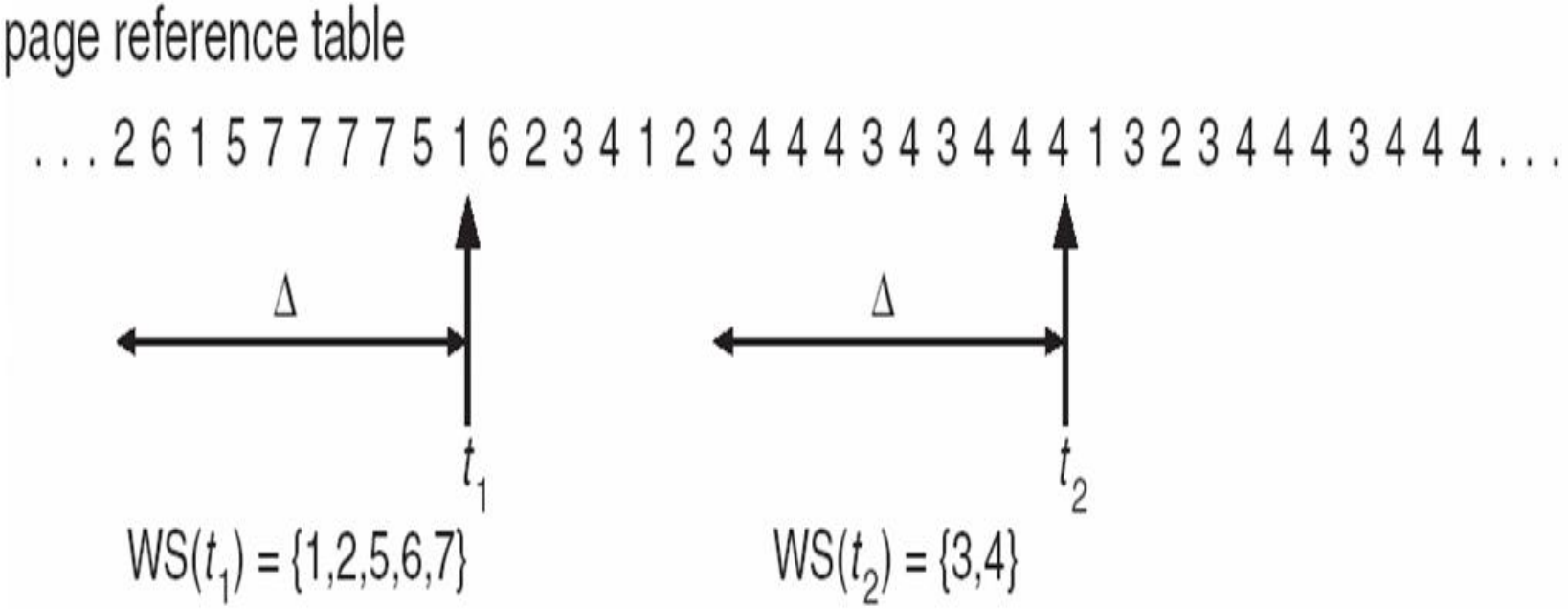
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- WSS_i (working set of Process P_i) =
  total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass 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 Sets and Page Fault Rates

There is a direct relationship between the working set of a process and its page-fault rate.



Working-set model

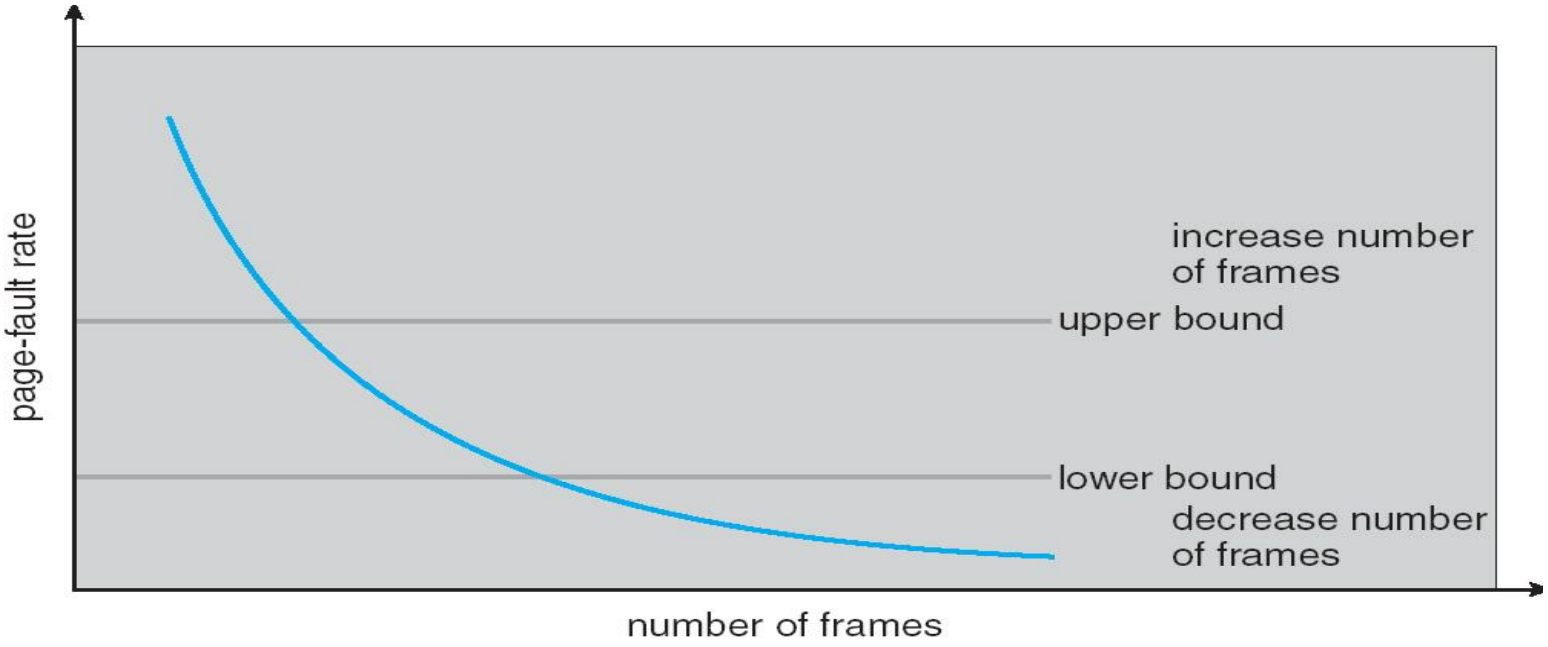


Keeping Track of the Working Set

- Approximate with a fixed-interval timer + a reference bit
- Example: Δ = 10,000 references
  - Timer interrupts after every 5000 references
  - 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



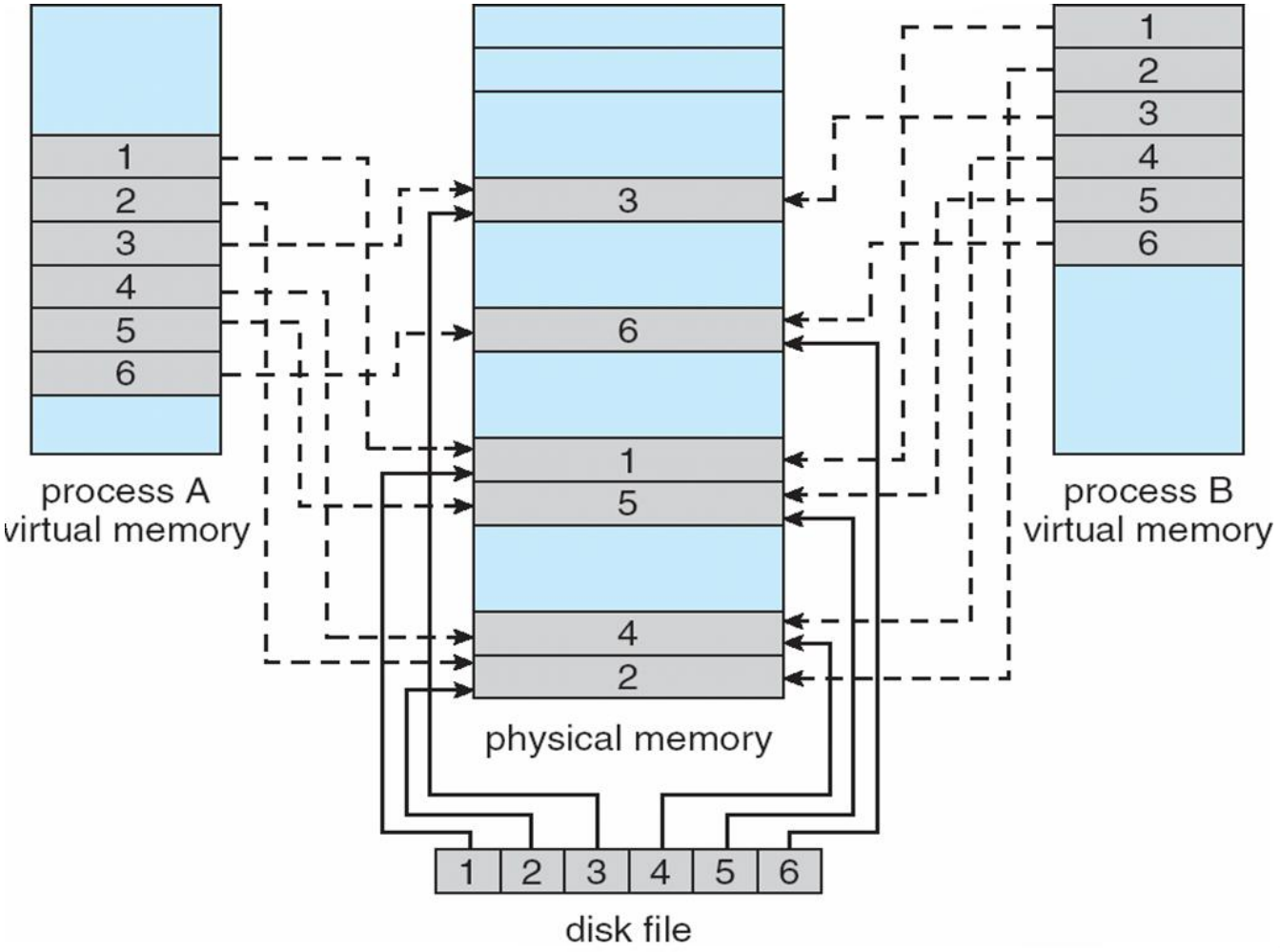
Memory-Mapped Files

- 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
- Less overhead than using read() and write() system calls

Memory-Mapped Files

- Also allows several processes to map the same file allowing the pages in memory to be shared
- Writes by any of the processes modify the data in virtual memory and can be seen by all others that map the same section of the file.

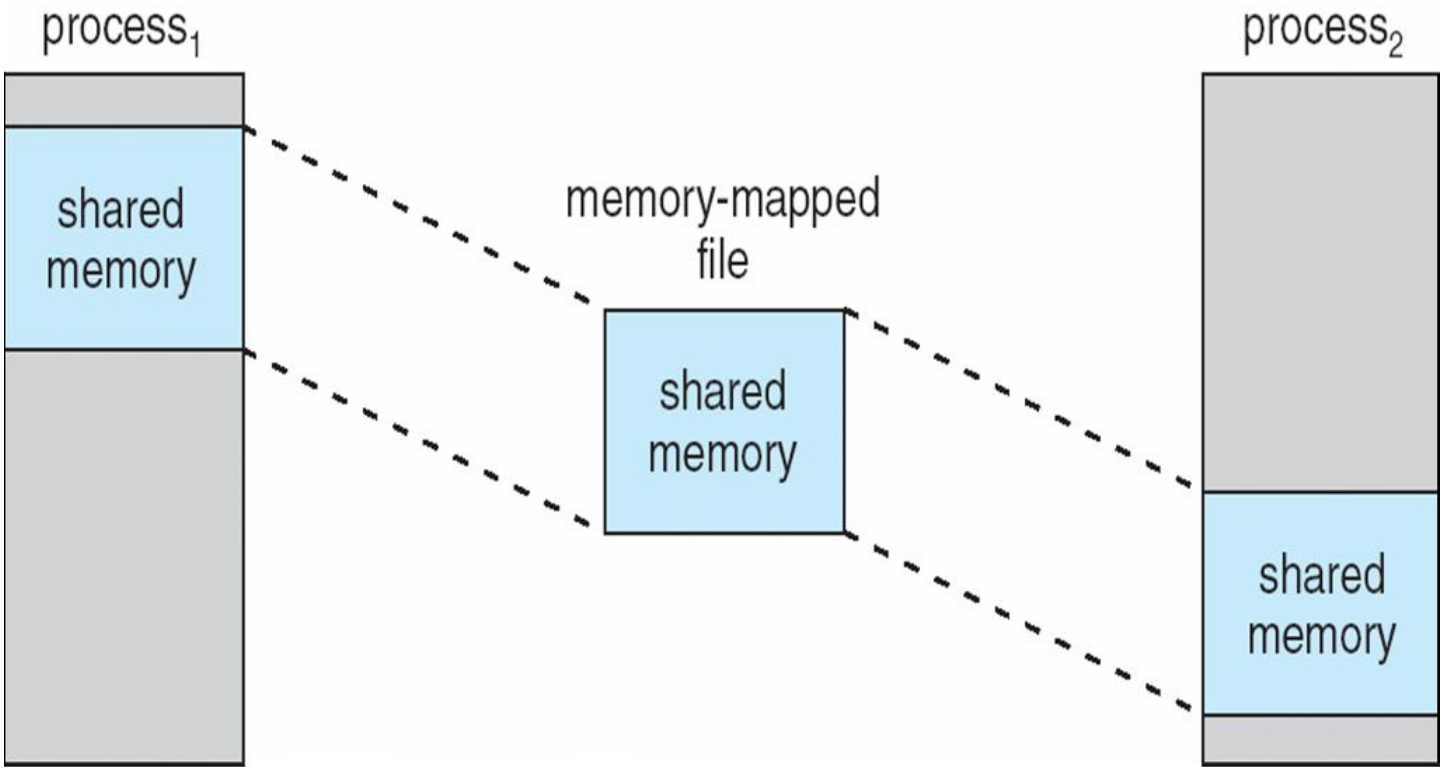
Memory Mapped Files



Memory-Mapped Files

- The sharing of memory-mapped files is similar to shared memory
- But not all systems use the same mechanism for both
- On UNIX and Linux, use mmap() for memory mapping, use shmget() and shmat() for shared memory
- On Windows, shared memory is accomplished by memory mapping files.

Memory-Mapped Shared Memory in Windows



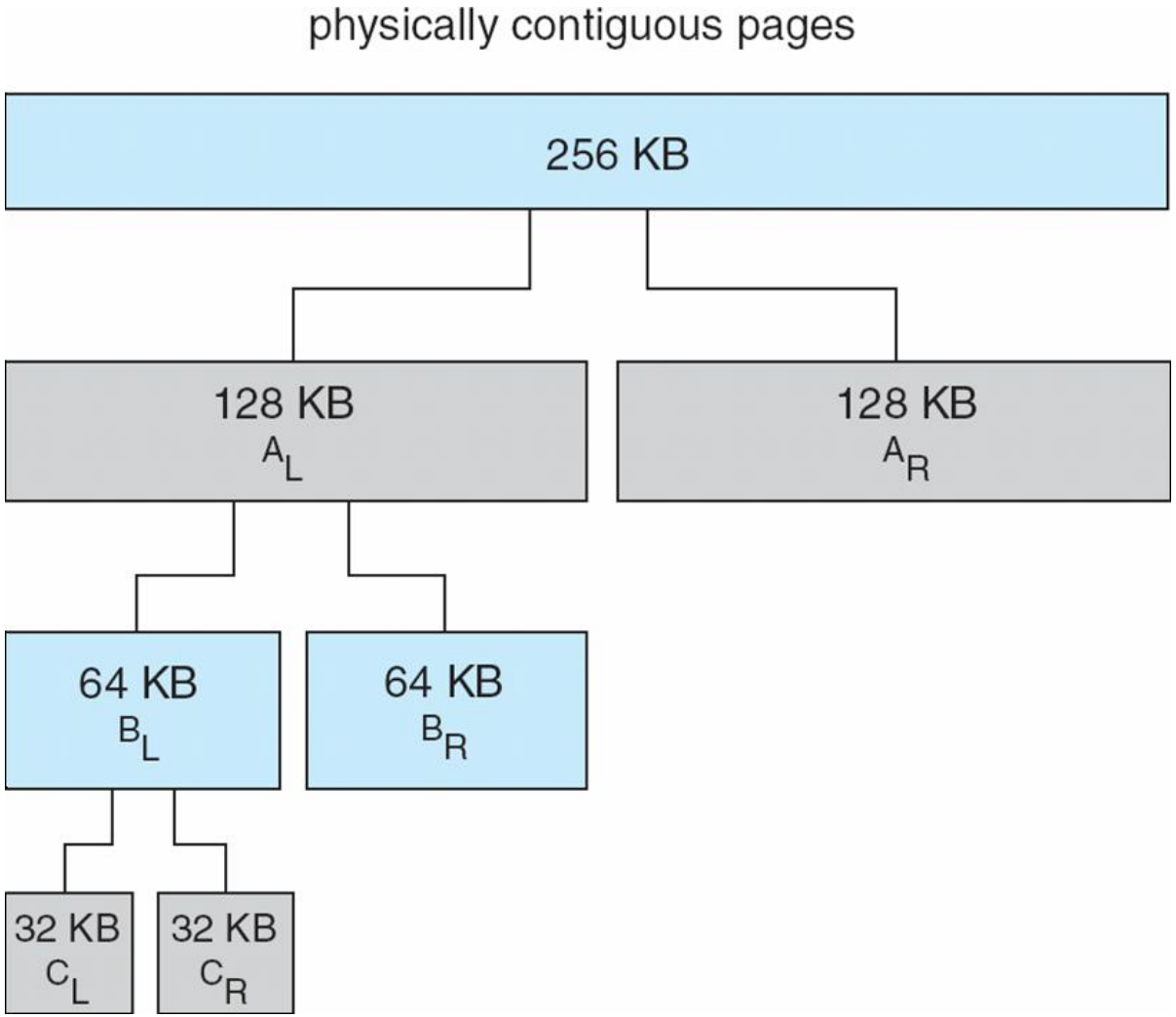
Allocating Kernel Memory

- Kernel memory is treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
- Two strategies for managing free memory that is assigned to kernel processes
  - Buddy system
  - Slab allocation

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available

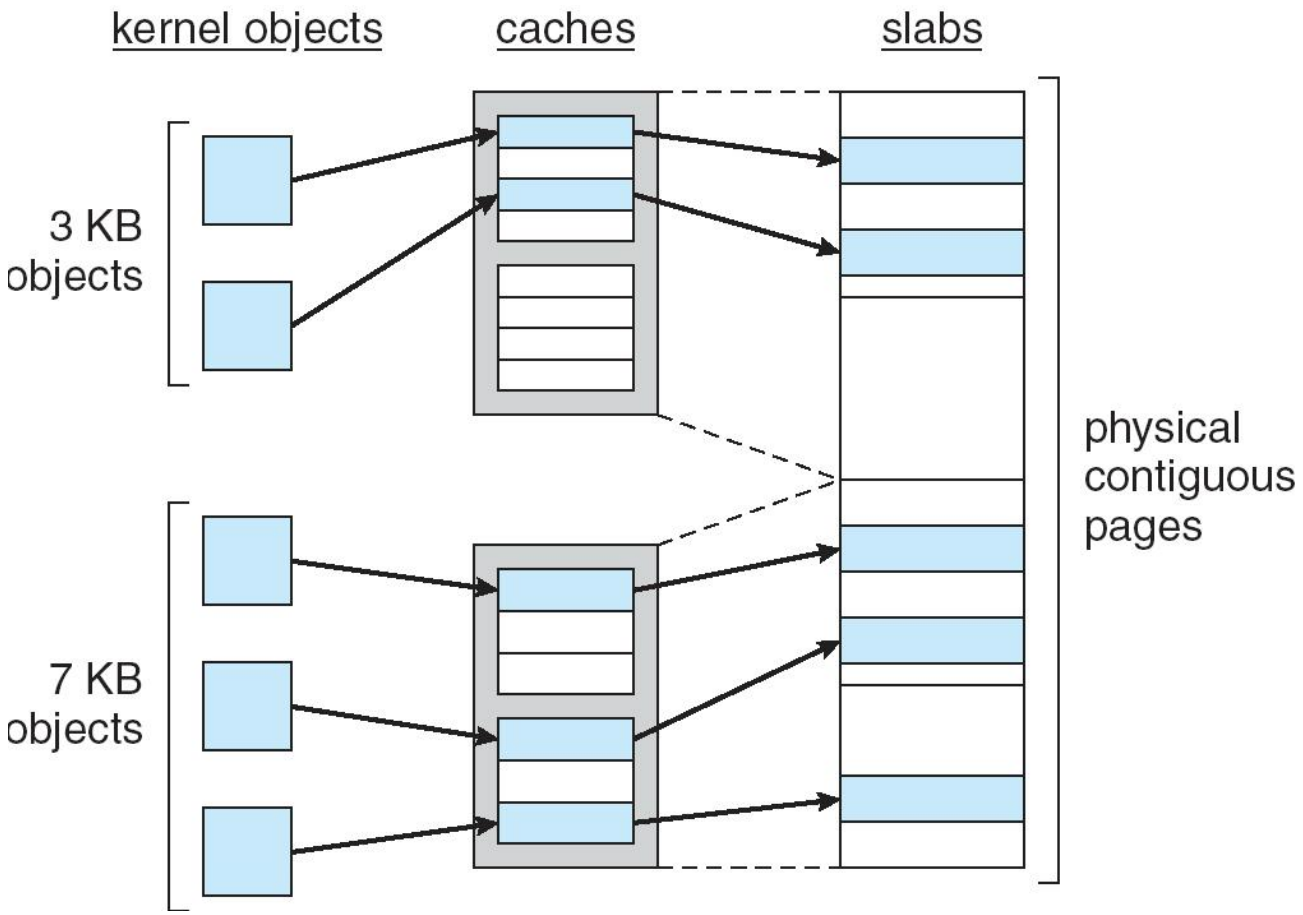
Buddy System Allocator



Slab Allocator

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

Slab Allocation



Other Issues -- Prepaging

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

Other Issues – Page Size

- Page size selection must take into consideration:
  - Fragmentation
    - Need a small page size to minimize internal fragmentation
  - Page table size
    - Need a large page size to decrease the size of page table
  - I/O overhead
    - Need a large page size to minimize I/O time
  - Locality
    - A smaller page size allows each page to match program locality more accurately
    - Thus a smaller page size should result in less I/O and less total allocated memory.
  - Page fault
    - Need a large page size to minimize the number of page faults

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

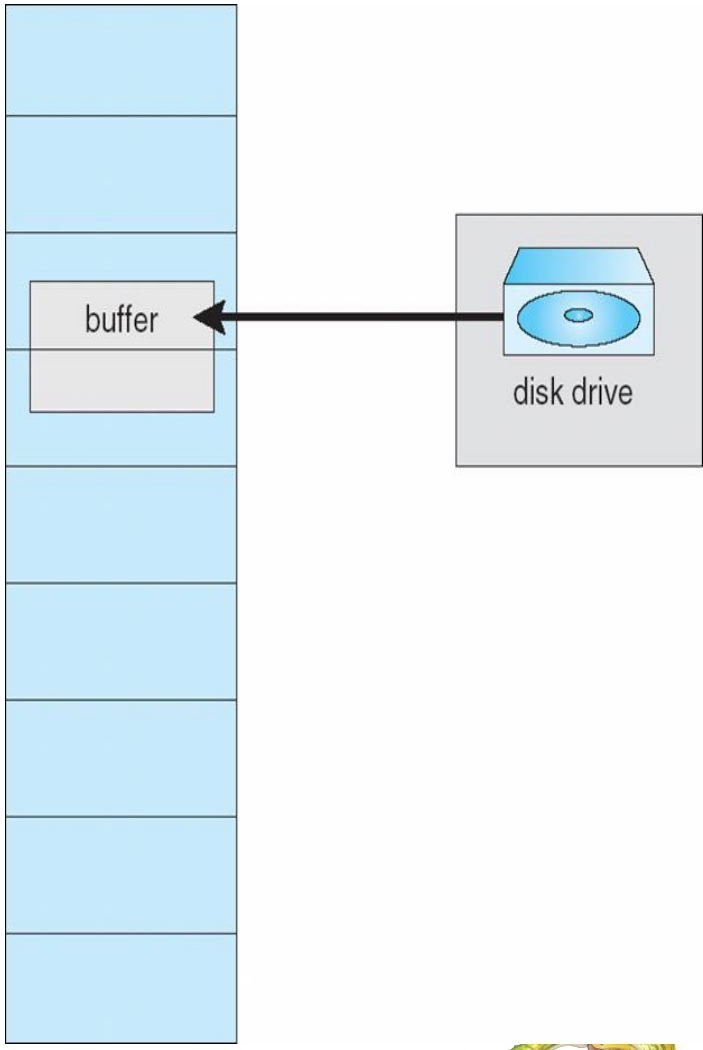
  - Program 2

```
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
- How to lock pages into memory?
  - A lock bit is associated with every frame.
  - Set lock bit on if a page need to be locked



Operating System Examples

- Windows XP
- Solaris

Windows XP

- 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
  - For most applications, the value of working set minimum and maximum is 50 and 345 pages respectively.

Windows XP

- The virtual memory manager maintains a list of free page frames.
  - If a page fault occurs for a process that is below its working-set maximum, the virtual memory manager allocates a page from the list of free pages
  - If a page fault occurs for a process that is at its working-set maximum, it must select a page for replacement using a local page-replacement policy.

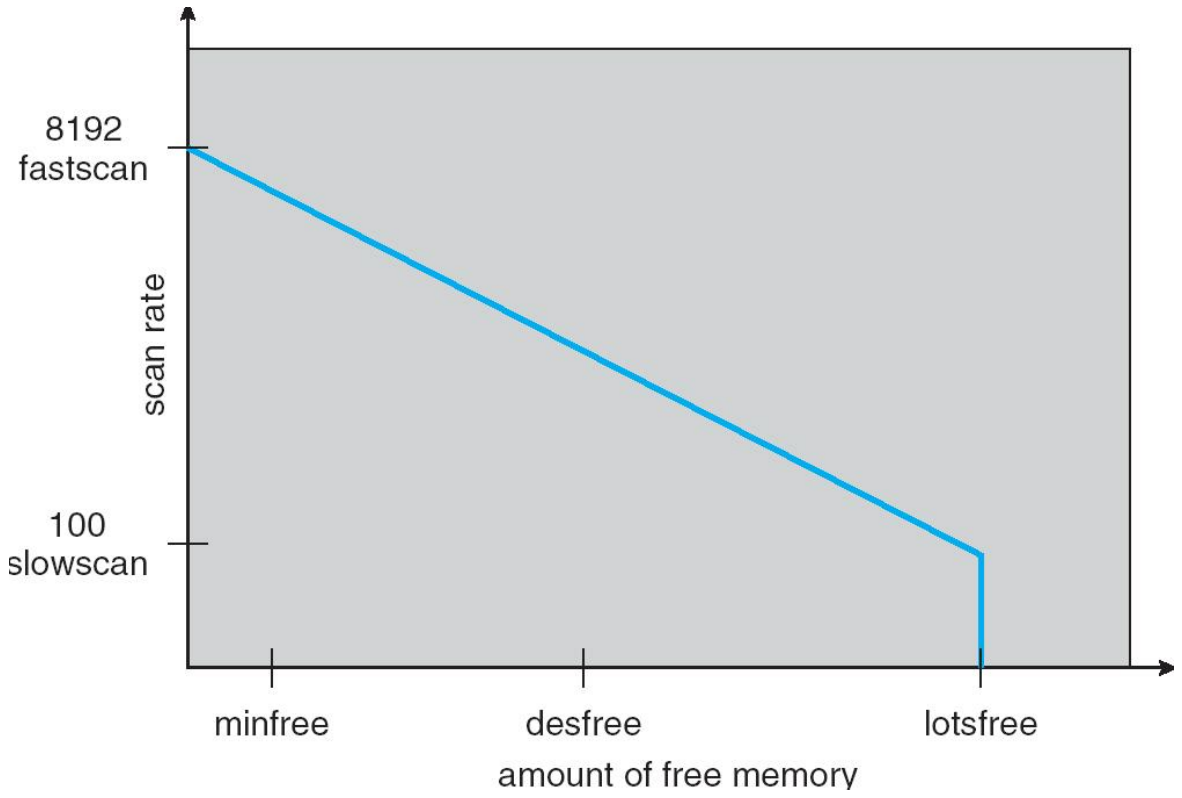
Windows XP

- 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
  - Once sufficient free memory is available, a process that is at its working-set minimum may be allocated pages from the free page frame list.

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

Solaris 2 Page Scanner



Solaris

- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
  - *Lotsfree* is typically set to 1/64 the size of the physical memory
  - 4 times per second, the kernel checks whether the amount of free memory is less than lotsfree
    - If below *lotsfree*, a process pageout starts up, which is similar to the second-chance algorithm
- If free memory falls below *desfree*, pageout will run 100 times per second with the intention of keeping at least *desfree* free memory available

Solaris

- If the pageout is unable to keep free memory at *desfree* for a 30-second average, the kernel begins swapping processes.
  - Free all pages allocated to swappped processes.
  - The kernel looks for processes that have been idle for a long time.
- If the system unable to maintain free memory at *minfree*, the pageout process is called for every request for a new page.

Assignments

- P 366-368
  - 9.4
  - 9.10
  - 9.13

End of Chapter 9