Virtual Memory

UNIT-3

Background

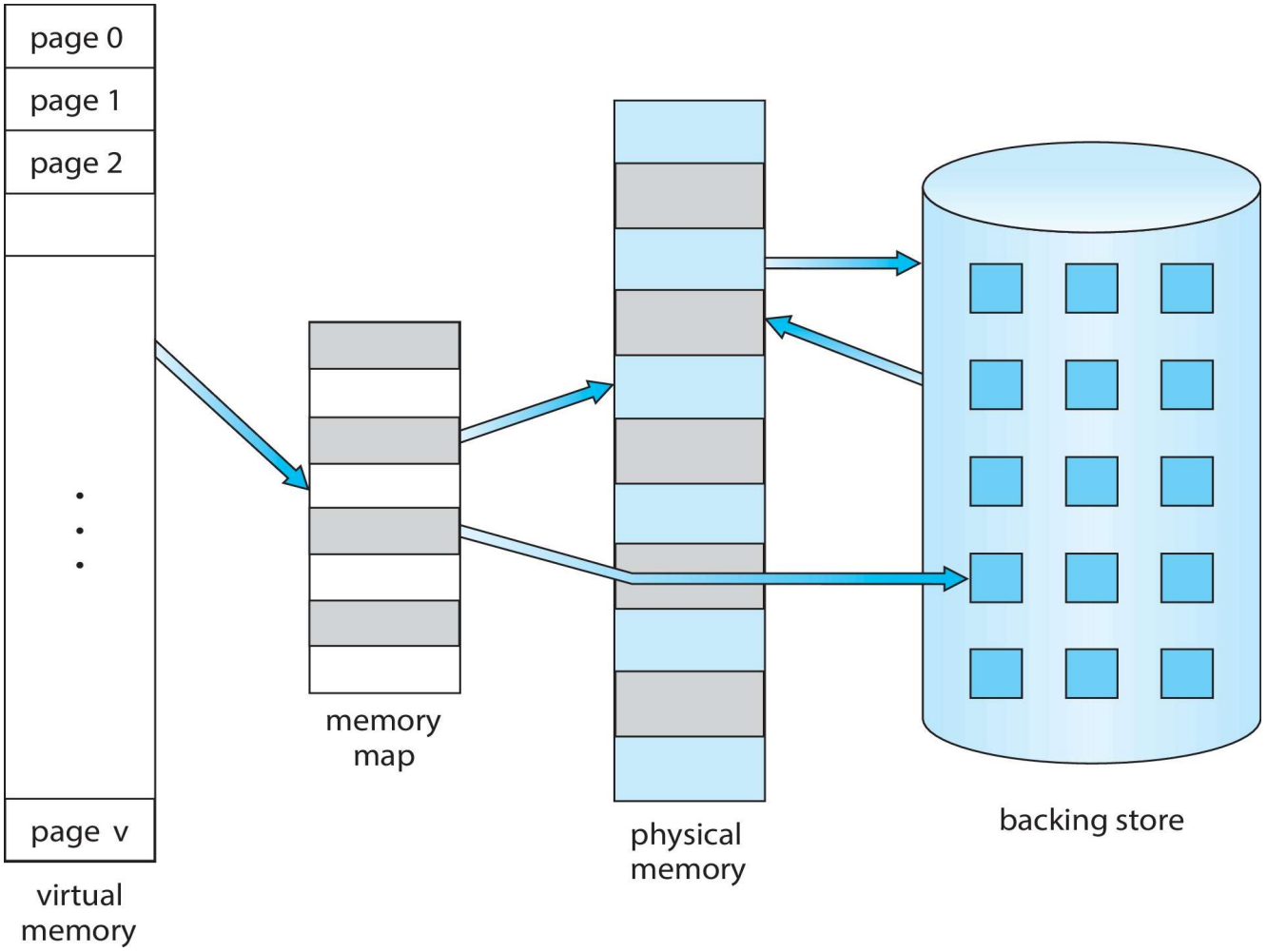
- The memory-management algorithms outlined in the previous chapter are necessary because of one basic requirement:
  - the instructions being executed must be in physical memory
- The requirement that instructions must be in physical memory limits the size of a program to the size of physical memory
- In fact, an examination of real programs shows us that ,in many cases, the entire program is not needed

- For instance, consider the following:
  - Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed
  - Arrays, lists, and tables are often allocated more memory than they actually need
  - Certain options and features of a program may be used rarely
- Even in those cases where the entire program is needed, it may not all be needed at the same time

- The ability to execute a program that is only partially in memory would confer many benefits:
- A program would no longer be constrained by the amount of physical memory that is available
- Because each program could take less physical memory, more programs could be run at the same time with a corresponding increase in
  - CPU utilization and throughput
- but with no increase in
  - response time or turnaround time

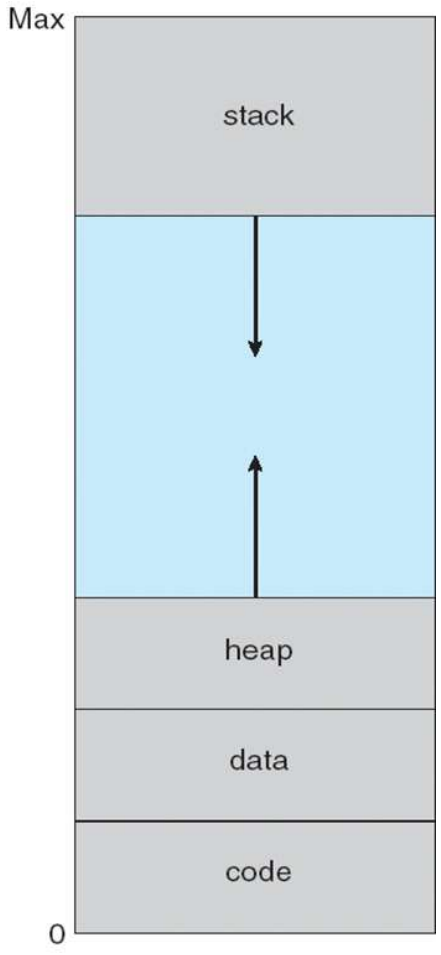
- Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster
- Virtual memory involves the separation of logical memory as perceived by developers from physical memory
- This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available
- Virtual memory makes the task of programming much easier because the programmer no longer needs to worry about the amount of physical memory available

Virtual Memory That is Larger Than Physical Memory



- The virtual address space of a process refers to the logical or virtual view of how a process is stored in memory
- Typically, this view is that a process begins at a certain logical address say address 0 and exists in contiguous memory, as shown in Figure in the next slide
- physical memory is organized in page frames and that the physical page frames assigned to a process may not be contiguous
- It is up to the memory management unit (MMU) to map logical pages to physical page frames in memory

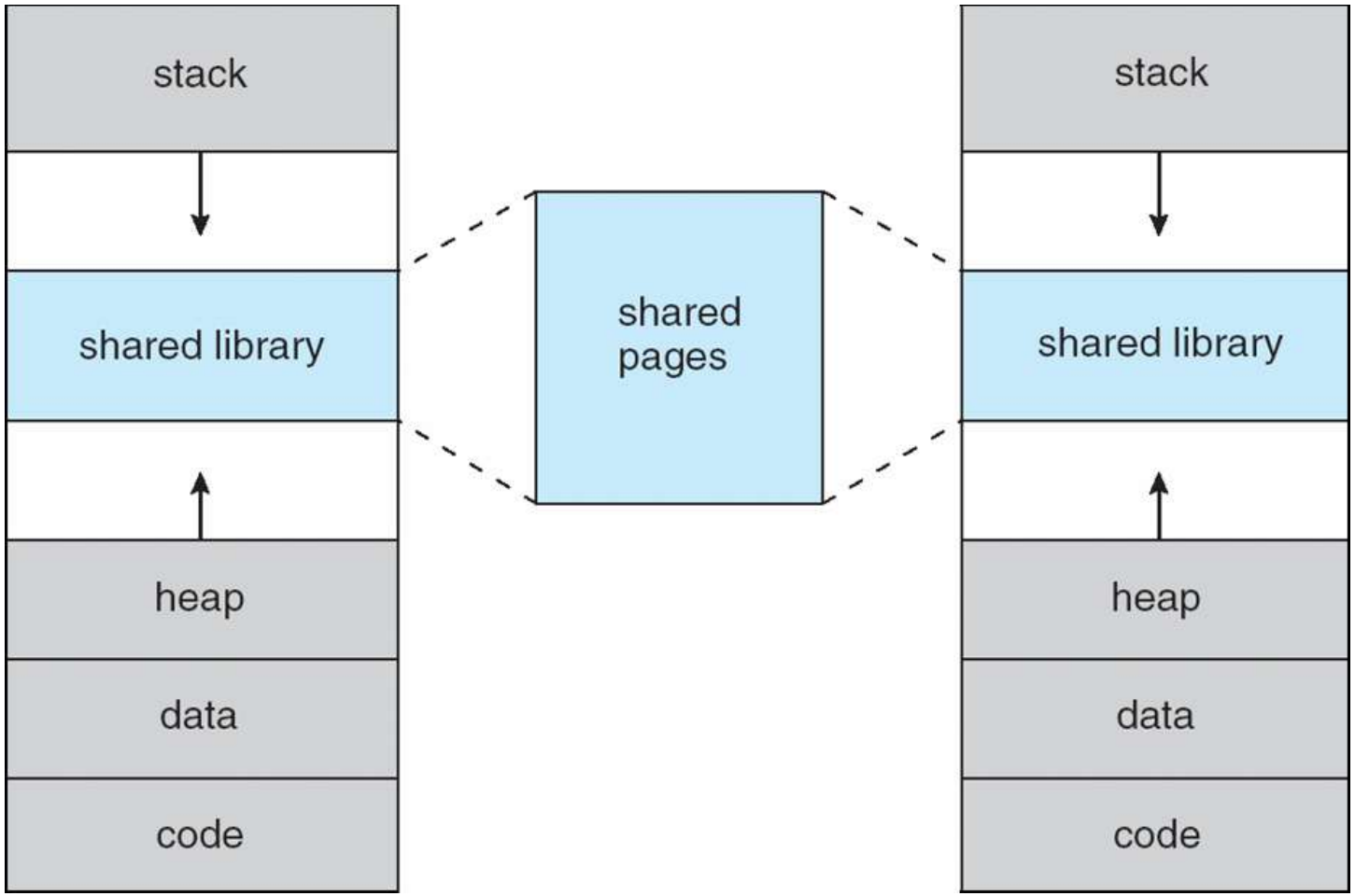
Virtual Address Space



- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
- Virtual address spaces that include holes are known as sparse address spaces.
- Using a sparse address space is beneficial because
  - The holes can be filled as the stack or heap segments grow
  - If we wish to dynamically link libraries (or possibly other shared objects) during program execution
- Virtual memory allows files and memory to be shared by two or more processes through page sharing

- This leads to the following benefits:
  - System libraries such as the standard C library can be shared by several processes through mapping of the shared object into a virtual address space (Figure in the next slide)
  - Similarly, processes can share memory
  - Pages can be shared during process creation with the fork() system call, thus speedingup process creation

Shared library using Virtual Memory



Demand Paging

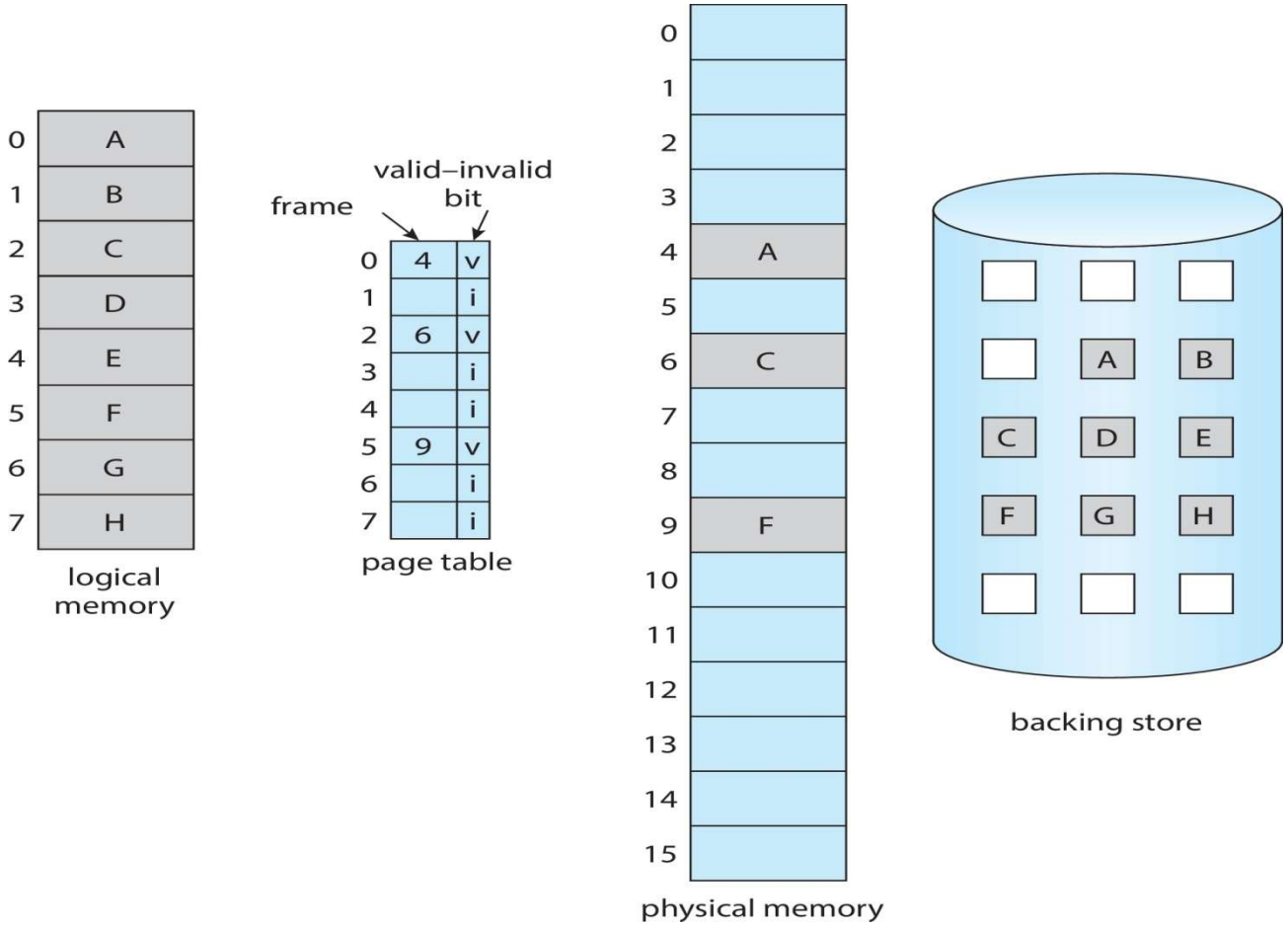
- is commonly used in virtual memory systems
- The technique of loading pages only as they are needed is known as demand paging
- Pages that are never accessed are thus never loaded into physical memory
- A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually an HDD or NVM device)

Basic Concepts

- The general concept behind demand paging, as mentioned, is to load a page in memory only when it is needed
- As a result, while a process is executing, some pages will be in memory, and some will be in secondary storage
- Thus, there is a need for some form of hardware support to distinguish between the two
- The valid– invalid bit scheme can be used for this purpose
  - when the bit is set to “valid,” the associated page is both legal and in memory

- If the bit is set to “invalid,” the page either
  - is not valid (that is, not in the logical address space of the process)
  - is valid but is currently in secondary storage
  - Figure in the next slide
- But what happens if the process tries to access a page that was not brought into memory?
  - Access to a page marked invalid causes a page fault

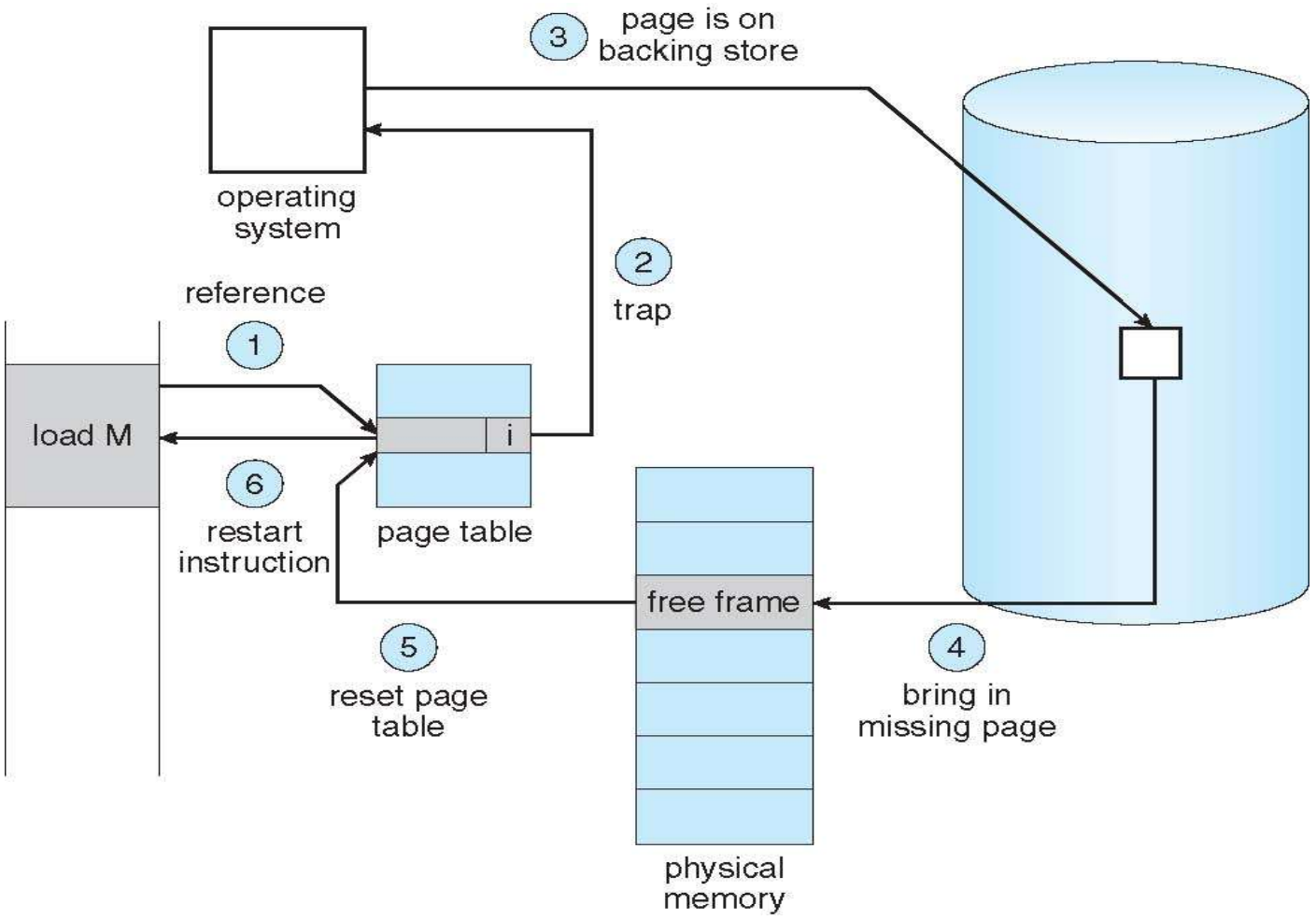
Page Table When Some Pages Are Not in Main Memory



- The procedure for handling this page fault is straightforward
  1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access
  2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in
  3. Find a free frame
  4. Schedule a secondary storage operation to read the desired page into the newly allocated frame
  5. When the storage read is complete, modify the internal table kept with the process and the page table to indicate that the page is now in memory

6. Restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory

Steps in Handling a Page Fault

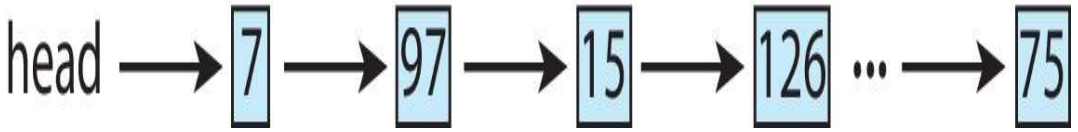


- Pure Demand Paging
  - In the extreme case, process can start executing with no pages in memory
  - When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page
  - After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory
  - At that point, it can execute with no more faults
  - This scheme is pure demand paging

- The hardware to support demand paging is the same as the hardware for paging and swapping:
  - Page table: This table has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits
  - Secondary memory: This memory holds those pages that are not present in main memory
  - The secondary memory is usually a high-speed disk or NVM device.
  - It is known as the swap device and the section of storage used for this purpose is known as swap space
- A crucial requirement for demand paging is the ability to restart any instruction after a page fault

Free Frame list

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory
- To resolve page faults, most operating systems maintain a free-frame list, a pool of free frames for satisfying such requests (Figure Below)

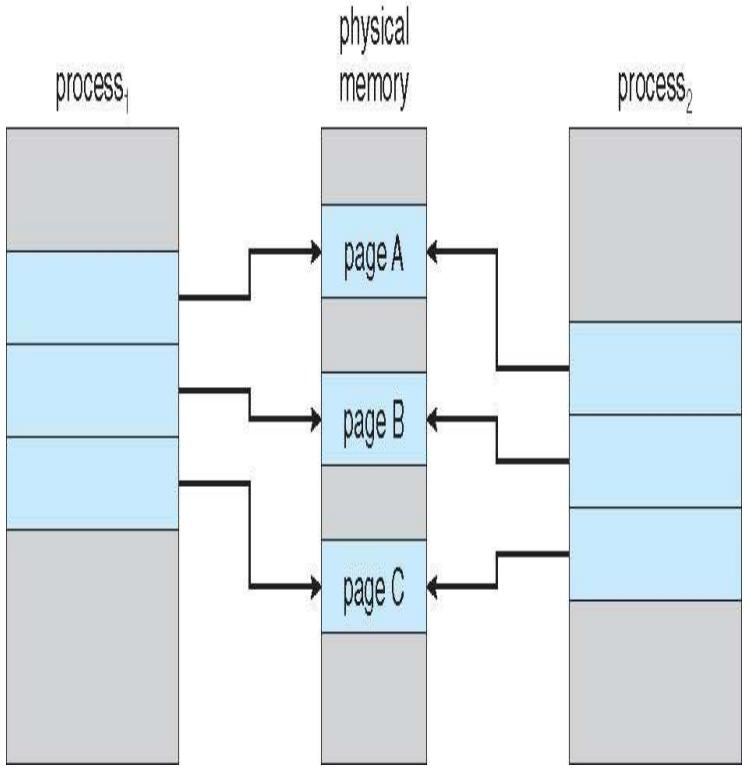


- Operating systems typically allocate free frames using a technique known as zero-fill-on-demand
- Zero-fill-on-demand frames are “zeroed-out” before being allocated, thus erasing their previous contents
- When a system starts up, all available memory is placed on the free-frame list
- As free frames are requested (for example, through demand paging), the size of the free-frame list shrinks
- At some point, the list either falls to zero or falls below a certain threshold, at which point it must be repopulated

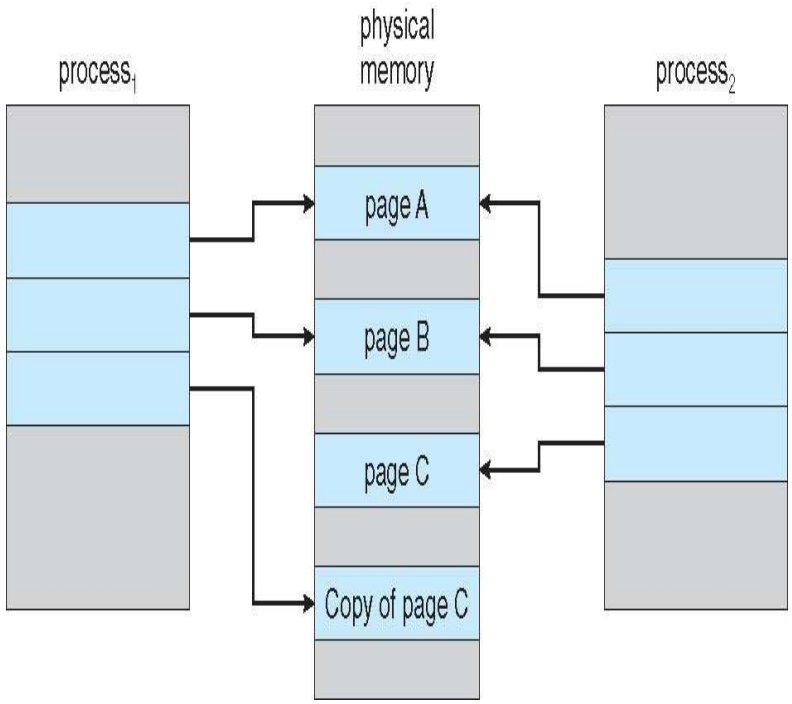
Copy-on-Write

- The fork() system call creates a child process that is a duplicate of its parent
- Traditionally, fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent
- However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary

- Instead, a technique known as copy-on-write can be used , which works by allowing the parent and child processes initially to share the same pages
- These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created
- copy-on-write is illustrated in the next two slides
- Copy-on-write is a common technique used by several operating systems, including Windows, Linux, and macOS



Before Process 1 Modifies Page C

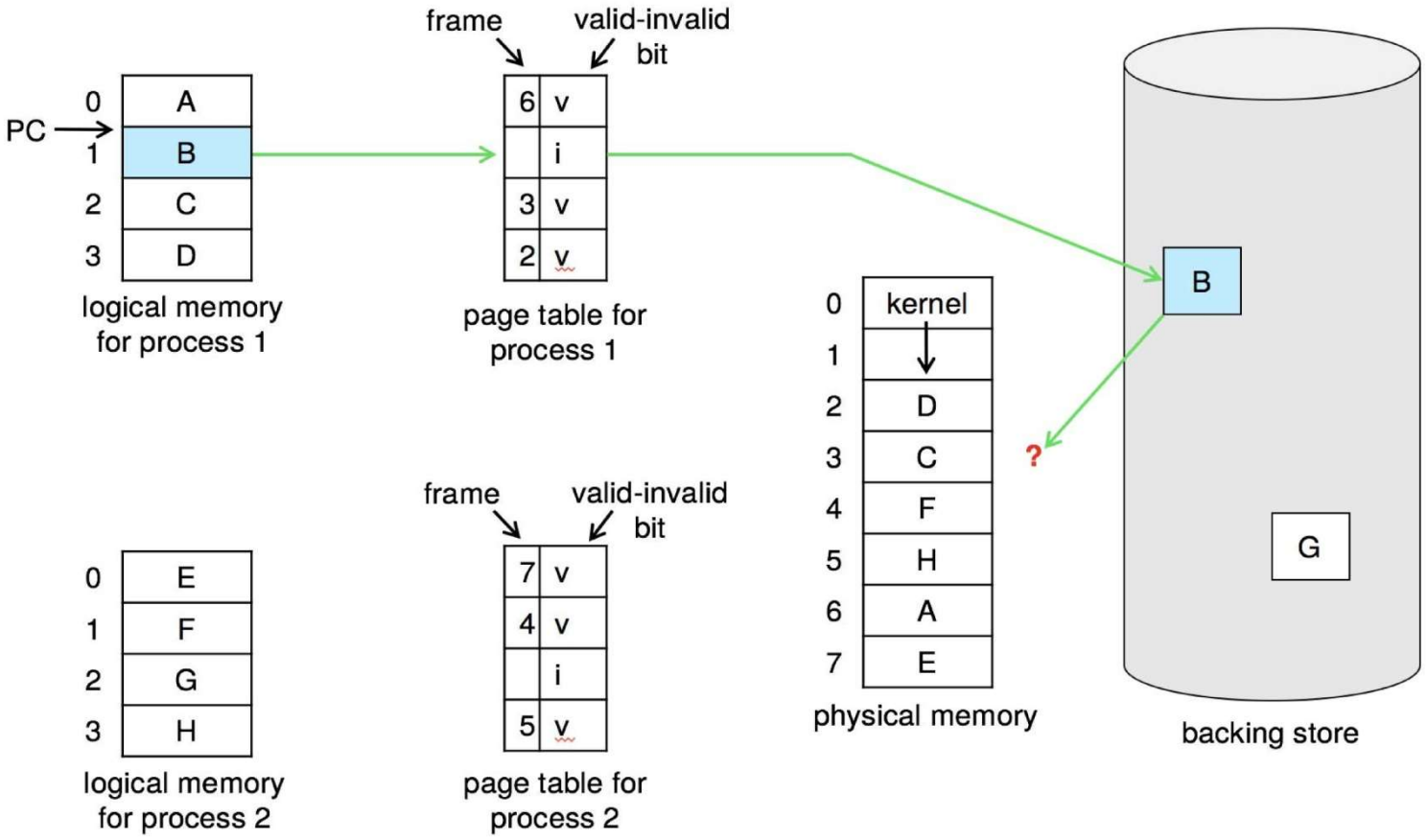


After Process 1 Modifies Page C

Page Replacement

- Increase in the degree of multiprogramming leads to over-allocating the memory
  - Ex: Suppose there are 40 frames. If six processes each of which is having 10 pages is run then 60 frames are needed.
- Over-allocation of memory manifests itself as follows
  - While a process is executing, a page fault occurs
  - The operating system determines where the desired page is residing on secondary storage but then finds that there are no free frames on the free-frame list
  - all memory is in use

Need For Page Replacement

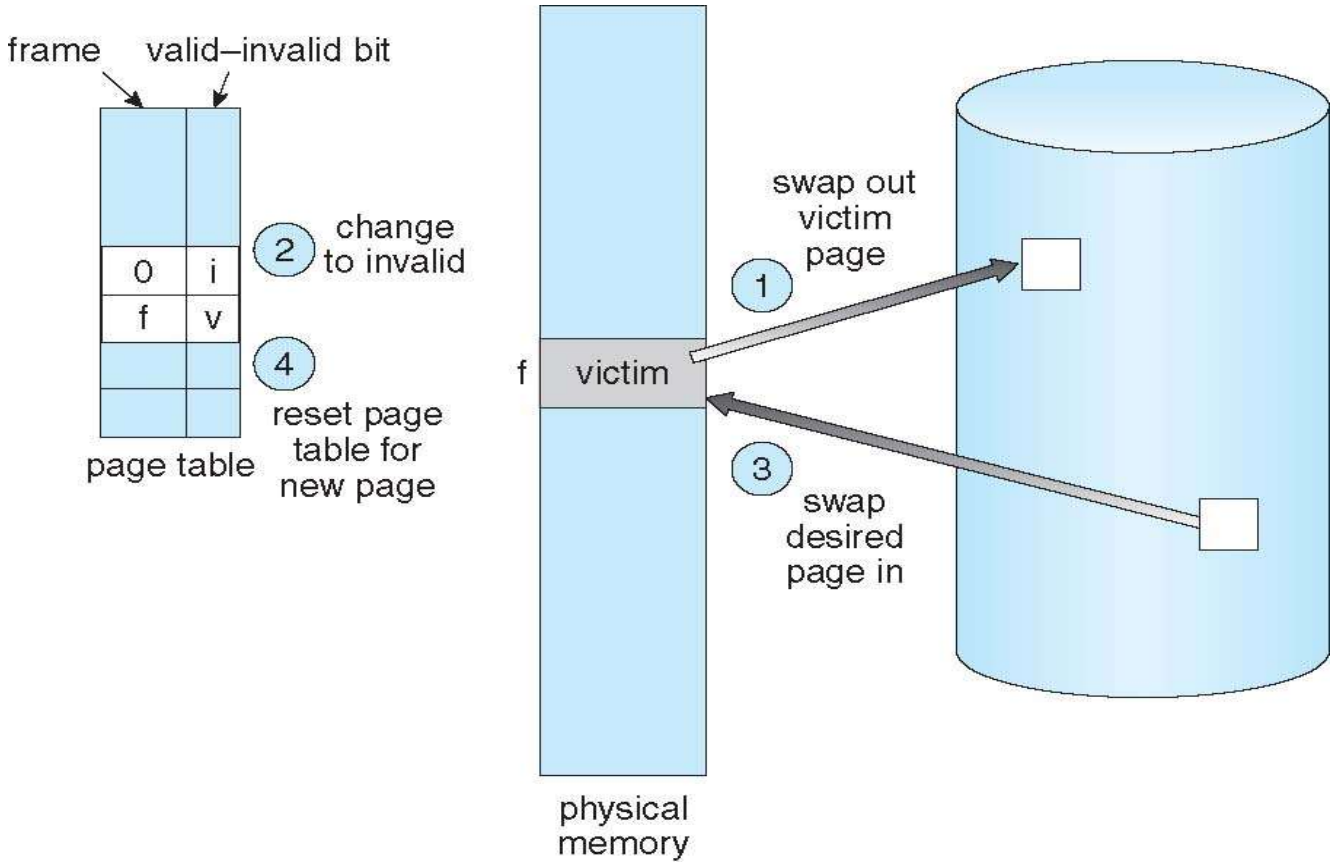


- The operating system has several options at this point
  - It could terminate the process.
    - However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput
  - The operating system could instead use standard swapping and swap out a process, freeing all its frames and reducing the level of multiprogramming
    - standard swapping is no longer used by most operating systems due to the overhead of copying entire processes between memory and swap space
- Most operating systems now combine swapping pages with page replacement

Basic Page Replacement

1. Find the location of the desired page on secondary storage
2. Find a free frame:
    a. If there is a free frame, use it
    b. If there is no free frame, use a page-replacement algorithm to select a victim frame
    c. Write the victim frame to secondary storage (if necessary); change the page and frame tables accordingly
3. Read the desired page into the newly freed frame; change the page and frame tables
4. Continue the process from where the page fault occurred

Page Replacement



- If no frames are free, two page transfers (one for the page-out and one for the page-in) are required
- This situation effectively doubles the page-fault service time and increases the effective access time accordingly
- This overhead can be reduced by using a modify bit(or dirty bit)
- Modify bit or dirty bit
  - When this scheme is used, each page or frame has a modify bit associated with it in the hardware
  - The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified

- When a page is selected for replacement, its modify bit is examined
- If the bit is set, the page has been modified since it was read in from secondary storage
- In this case, the page must be written to storage
- If the modify bit is not set, however, the page has not been modified since it was read into memory
- In this case, the memory page need not be written to the storage: it is already there
- This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half if the page has not been modified

- Page replacement is basic to demand paging
- It completes the separation between logical memory and physical memory
- With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory
- If there is a process of twenty pages, it can be executed with ten frames simply
  - by using demand paging and
  - using a replacement algorithm to find a free frame whenever necessary

- Two major problems must be solved to implement demand paging
  - By developing a frame-allocation algorithm and a page-replacement algorithm
  - If there are multiple processes in memory, it must be decided how many frames have to be allocated to each process; and
  - when page replacement is required, frames that are to be replaced must be selected
- There are many different page-replacement algorithms

- Every operating system probably has its own replacement scheme
- How to select a particular replacement algorithm?
  - In general, the replacement algorithm with the lowest page-fault rate
  - An algorithm is evaluated by running it on a particular string of memory references and computing the number of page faults
- The string of memory references is called a reference string
- Reference strings are generated artificially
  - by using a random-number generator
  - trace a given system and record the address of each memory reference

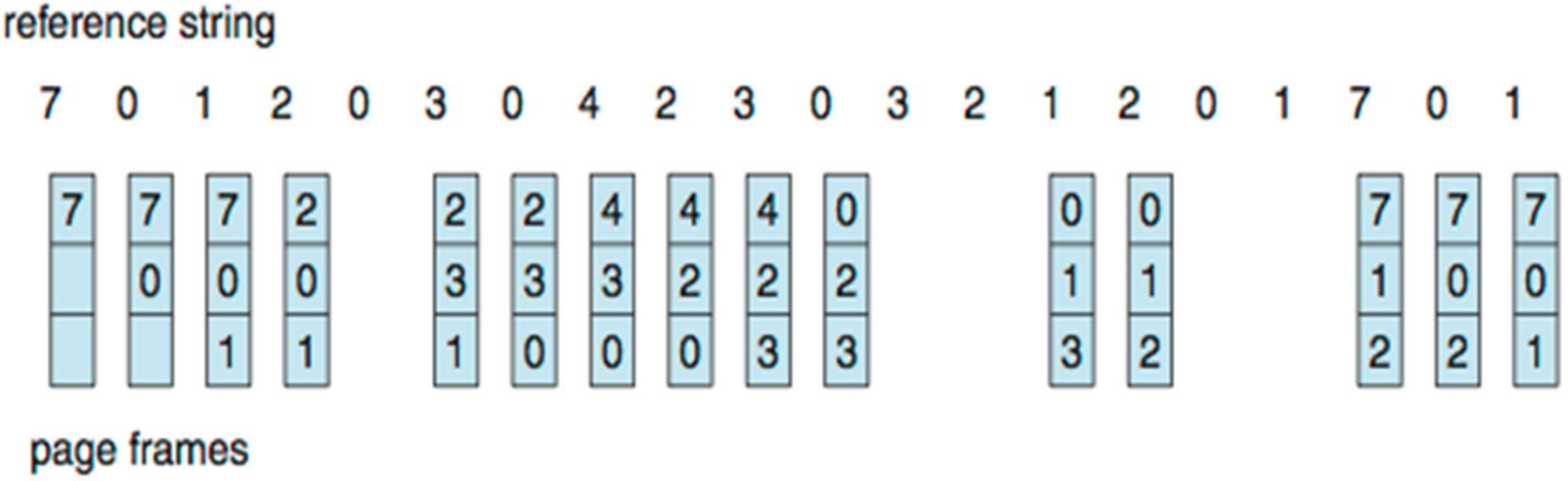
- The second choice produces a large number of data (on the order of 1 million addresses per second)
- To reduce the number of data, two facts are used:
    1. For a given page size consider only the page number, rather than the entire address
    2. If we have a reference to a page p, then any references to page p that immediately follow will never cause a page fault.

- Consider the following address sequence
  - 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
  - At 100 bytes per page, this sequence is reduced to the following reference string:
  - 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1
- To determine the number of page faults for a particular reference string and page-replacement algorithm, the number of page frames available must be known.

FIFO Page Replacement algorithm

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm
- A FIFO replacement algorithm associates with each page the time when that page was brought into memory
- When a page must be replaced, the oldest page is chosen
- It is not strictly necessary to record the time when a page is brought in
- A FIFO queue is created to hold all pages in memory

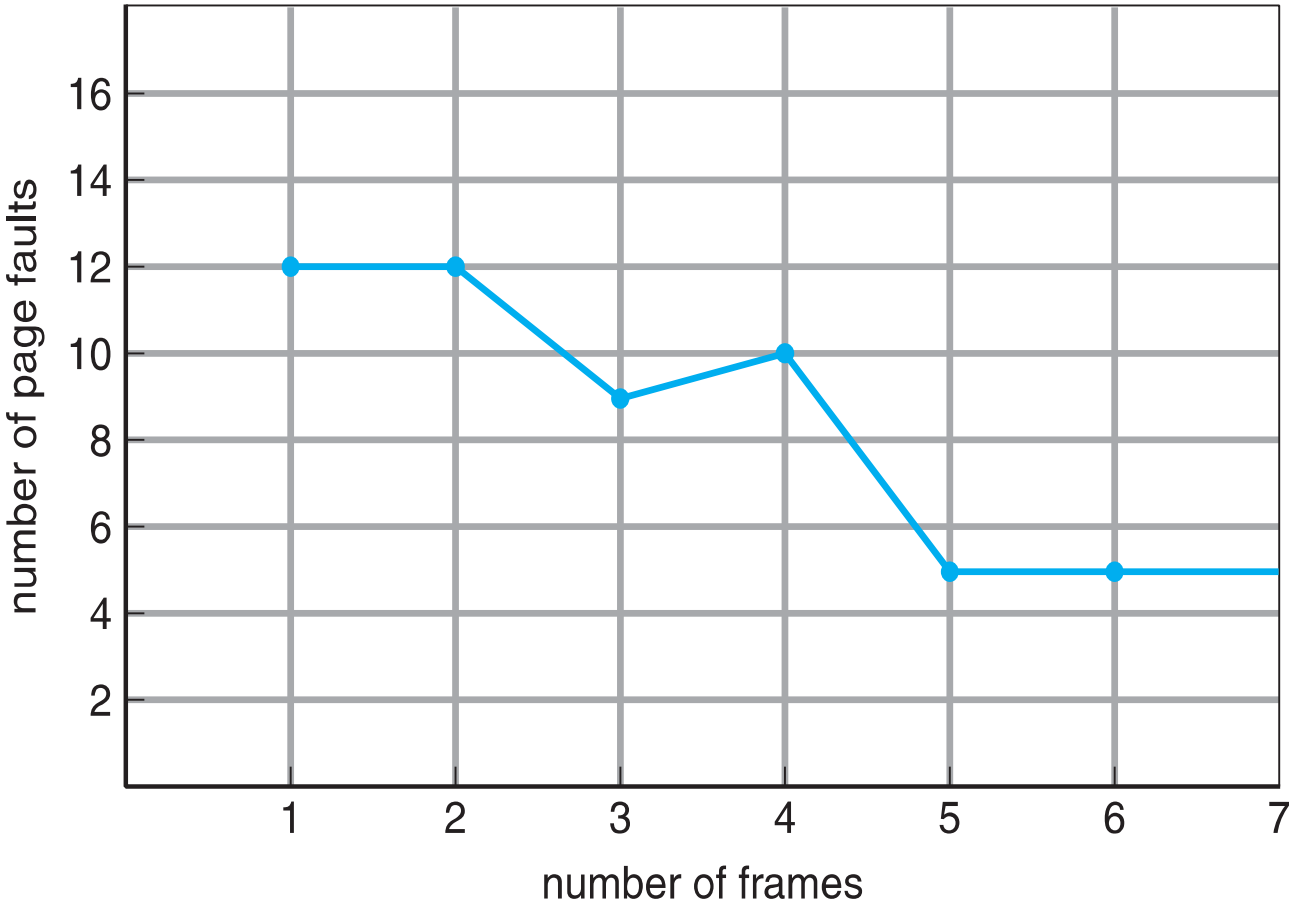
- The page at the head of the queue is replaced
- When a page is brought in to memory, it is inserted at the tail of the queue
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3, 2,1,2,0,1,7,0,1**
- 3 frames are used



- Page faults=15
- The FIFO page-replacement algorithm is easy to understand and program
- However, its performance is not always good
  - On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed
  - On the other hand, it could contain a heavily used variable that was initialized early and is in constant use

- consider the following reference string:
- 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Figure in the next slide shows the curve of page faults for this reference string versus the number of available frames
- The number of faults for four frames (ten) is greater than the number of faults for three frames (nine)
- This most unexpected result is known as Belady's anomaly:
  - for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases

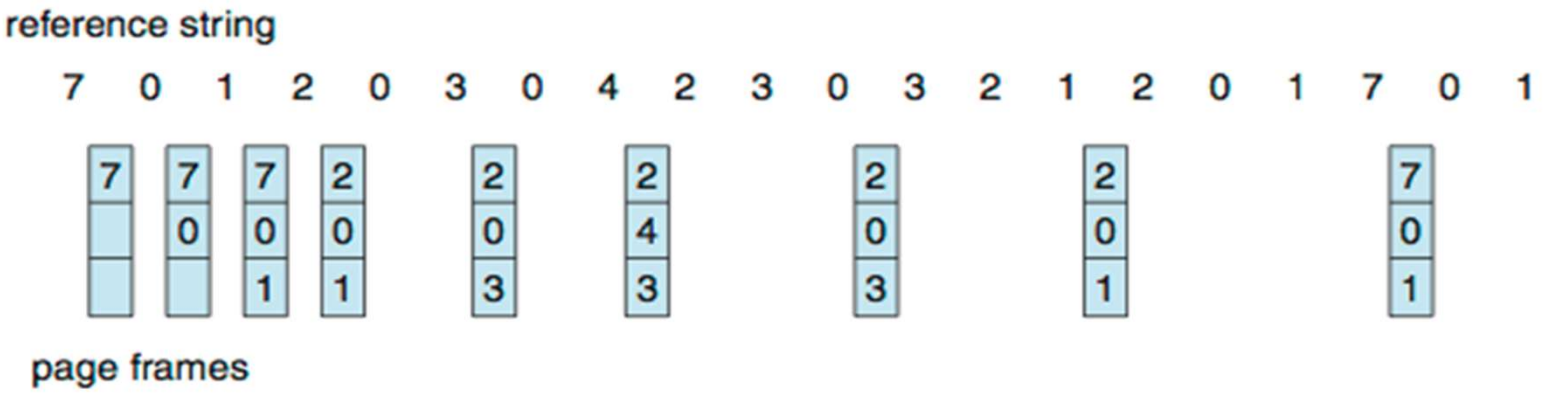
FIFO Illustrating Belady's Anomaly



Optimal Page Replacement algorithm

- The algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly
- According to this algorithm
  - Replace the page that will not be used for the longest period of time
- Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames
- Consider the reference string:
  - **7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**
- Three frames

- Number of page faults= 9

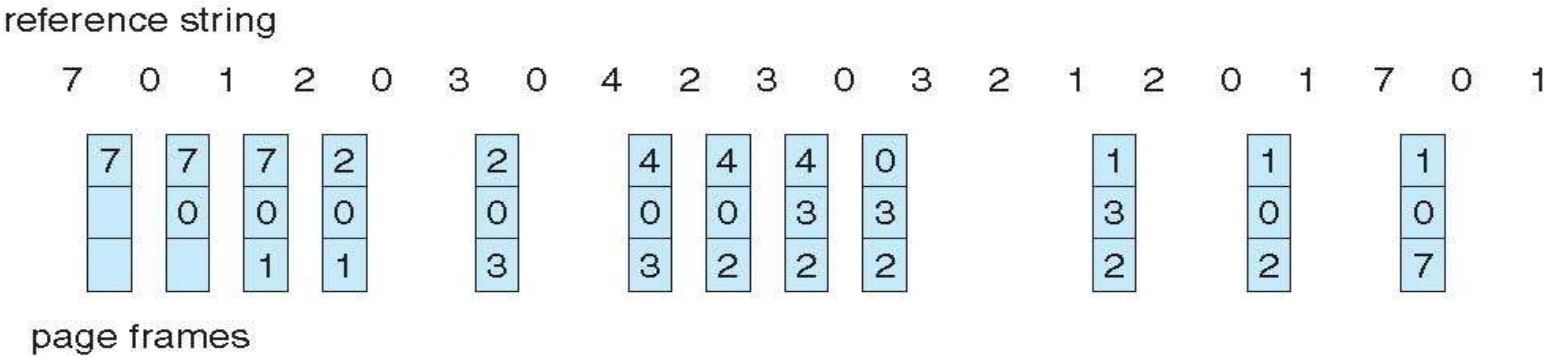


- the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string
  - The optimal algorithm is used mainly for comparison studies

LRU Page Replacement Algorithm

- Least Recently Used (LRU) replacement associates with each page the time of that page's last use
- When a page must be replaced , LRU chooses the page that has not been used for the longest period of time
- Consider the reference string:
  - **7,0,1,2,0,3,0,4,2,3,0,3, 2,1,2,0,1,7,0,1**
- Three frames

- Number of page faults=12



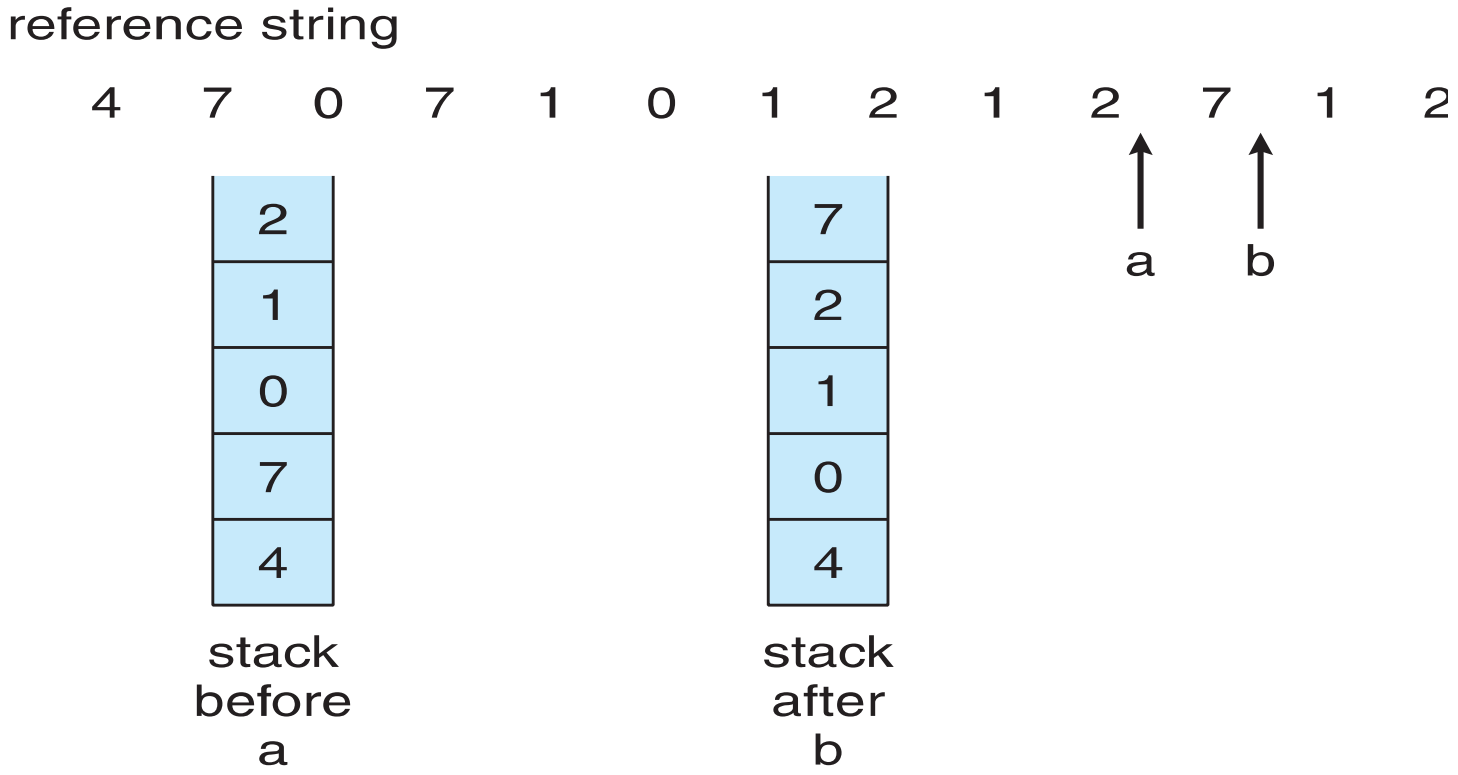
- The LRU policy is often used as a page-replacement algorithm and is considered to be good
- LRU page-replacement algorithm may require substantial hardware assistance

- Two implementations are feasible:
  - Counters
  - Stack
- Counters
  - In the simplest case, each page-table entry is associated with a time-of-use field and add to the CPU a logical clock or counter
  - The clock 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

  - In this way, the “time” of the last reference to each page is obtained
  - Replace the page with the smallest time value
  - This scheme requires a search of the page table to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access
  - Overflow of the clock must be considered
- Stack
  - This approach keeps a stack of page numbers
  - Whenever a page is referenced, it is removed from the stack and put on the top

– In this way, the most recently used page is always at the top of the stack, and the least recently used page is always at the bottom

– See the figure below



  - Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer
  - The tail pointer points to the bottom of the stack, which is the LRU page
  - This approach is particularly appropriate for software or microcode implementations of LRU replacement

- LRU and optimal page replacement algorithms do not suffer from Belady's anomaly
- Both belong to a class of page-replacement algorithms, called stack algorithms, that can never exhibit Belady's anomaly

Question

- Consider the following page reference string:
  - 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6
- How many page faults would occur for the following replacement algorithms assuming four frames?
  - LRU replacement
  - FIFO replacement
  - Optimal replacement

Solution-FIFO Page Replacement

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	1	1	1			5	5	5	5		3	3	3		3	1		1	
	2	2	2			2	6	6	6		6	7	7		7	7		3	
		3	3			3	3	2	2		2	2	6		6	6		6	
			4			4	4	4	1		1	1	1		2	2		2	

Number of page faults=14

Solution-Optimal Page Replacement

1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	1	1	1			1	1					7				1			
	2	2	2			2	2					2				2			
		3	3			3	3					3				3			
			4			5	6					6				6			

Number of page faults=8

LRU Approximation Page Replacement

- Not many computer systems provide sufficient hardware support for true LRU page replacement
- Many systems provide some help, however, in the form of a *reference bit*
- The reference bit for a page is set by the hardware whenever that page is referenced
  - either a read or a write to any byte in the page.
- Reference bits are associated with each entry in the page table

- Initially, all bits are cleared (to 0) by the operating system
- As a process executes, the bit associated with each page referenced is set (to 1) by the hardware
- After sometime, we can determine which pages have been used and which have not been used by examining the reference bits, although we do not know the order of use
- This information is the basis for many page-replacement algorithms that approximate LRU replacement

Additional-Reference-Bits Algorithm

- Additional ordering information can be gained by recording the reference bits at regular intervals
- We can keep an 8-bit byte for each page in a table in memory
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system
- The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit

- These 8-bit shift registers contain the history of page use for the last eight time periods
- If the shift register contains 00000000, for example, then the page has not been used for eight time periods
- A page that is used at least once in each period has a shift register value of 11111111
- A page with a history register value of 11000100 has been used more recently than one with a value of 01110111

- If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced

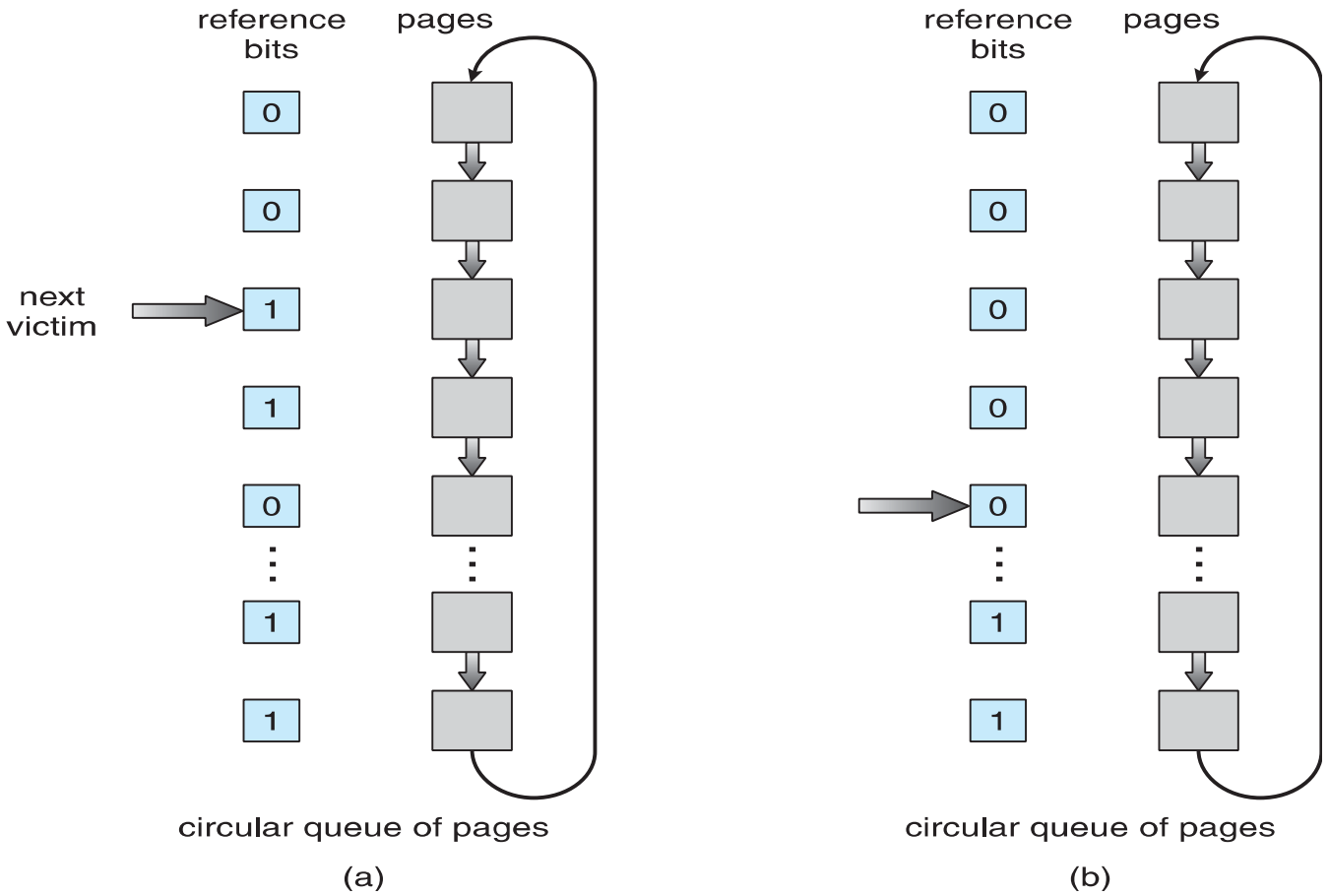
Second chance Algorithm

- The basic algorithm of second-chance replacement is a FIFO replacement algorithm
- When a page has been selected, however, we inspect its reference bit
- If the value is 0, we proceed to replace this page
  - but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page
- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time

- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced
- In addition, if a page is used often enough to keep its reference bit set, it will never be replaced
- One way to implement the second-chance algorithm is as a circular queue
- A pointer indicates which page is 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 shown in the figure in the next slide
- Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position
- In the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance
- It clears all the reference bits before selecting the next page for replacement
  - Second-chance replacement degenerates to FIFO replacement if all bits are set

Second chance algorithm



Enhanced Second-Chance Algorithm

- Second-chance algorithm can be enhanced by considering the reference bit and the modify bit as an ordered pair
- With these two bits, we have the following four possible classes
  1. (0, 0) neither recently used nor modified—best page to replace
  2. (0,1) not recently used but modified—not quite as good, because the page will need to be written 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 the page will be need to be written out to secondary storage before it can be replaced

- Each page is in one of these four classes
- When page replacement is called for
  - examine the class to which that page belongs and replace the first page encountered in the lowest non empty class

Counting-Based Page Replacement

- There are many other algorithms that can be used for page replacement
- We can keep a counter of the number of references that have been made to each page and develop the following two schemes

1. The **least frequently used (LFU) page-replacement** algorithm requires that the page with the smallest count be replaced

- The reason for this selection is that an actively used page should have a large reference count

- Problem
  - when a page is used heavily during the initial phase of a process but then is never used again
  - Since it was used heavily, it has a large count and remains in memory even though it is no longer needed
- Solution
  - Shift the counts right by 1 bit at regular intervals

2. The most frequently used (MFU) page-replacement algorithm is 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

- How do we allocate the fixed amount of free memory among the various processes?
  - Consider a simple case of a system with 128 frames
  - The operating system may take 35, leaving 93 frames for the user process
  - Under pure demand paging, all 93 frames would initially be put on the free-frame list
  - When a user process started execution, it would generate a sequence of page faults

- The first 93 page faults would all get free frames from the free-frame list
- When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94^{th}, and so on
- When the process terminated, the 93 frames would once again be placed on the free-frame list

- There are many variations on this simple strategy
- But the basic strategy is clear: the user process is allocated any free frame

Minimum Number of Frames

- Strategies for the allocation of frames are constrained in various ways
  - Cannot allocate more than the total number of available frames
  - Allocate at least a minimum number of frames
- One reason for allocating at least a minimum number of frames involves performance
- Example
  - if one-level indirect addressing is allowed then paging requires at least three frames per process

  - For example, a load instruction on frame 16 can refer to an address on frame 0, which is an indirect reference to frame 23

- The minimum number of frames per process is defined by the architecture
- The maximum number is defined by the amount of available physical memory

Allocation algorithms

- Equal allocation
  - Split m frames among n processes is to give everyone an equal share, m/n frames
  - For instance, if there are 93 frames and 5 processes, each process will get 18 frames
  - The 3 left over frames can be used as a free-frame buffer pool
- An alternative is to recognize that various processes will need differing amounts of memory

- Consider a system with a 1-KB frame size
- If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames
- Proportional allocation is used to solve this problem
- Proportional allocation
  - allocates available memory to each process according to its size

- Let the size of the virtual memory for process p_i be s_i, and define

$$S = \sum s_i$$

- Then, if the total number of available frames is m, allocate a_i frames to process p_i, where a_i is approximately

$$a_i = \frac{s_i}{S} \times m$$

- each a_i is adjusted to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding m

- Consider the example of two processes with size 10KB and 127 KB running in a system with 62 frames where the frame size is 1KB
- Substituting the values in the equation

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 62 \approx 4$$

$$a_2 = \frac{127}{137} \times 62 \approx 57$$

- In this way, both processes share the available frames according to their “needs,” rather than equally
- In both equal and proportional allocation, of course, the allocation may vary according to the multiprogramming level

Global versus Local Allocation

- Another important factor in the way frames are allocated to the various processes is page replacement
- With multiple processes competing for frames, page-replacement algorithms can be classified into two broad categories:
  1. Global replacement
  2. Local replacement

- Global replacement
  - allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process that is, one process can take a frame from another
- Local replacement
  - requires that each process elect from only its own set of allocated frames
- For example, consider an allocation scheme wherein high priority processes select frames from low-priority processes for replacement

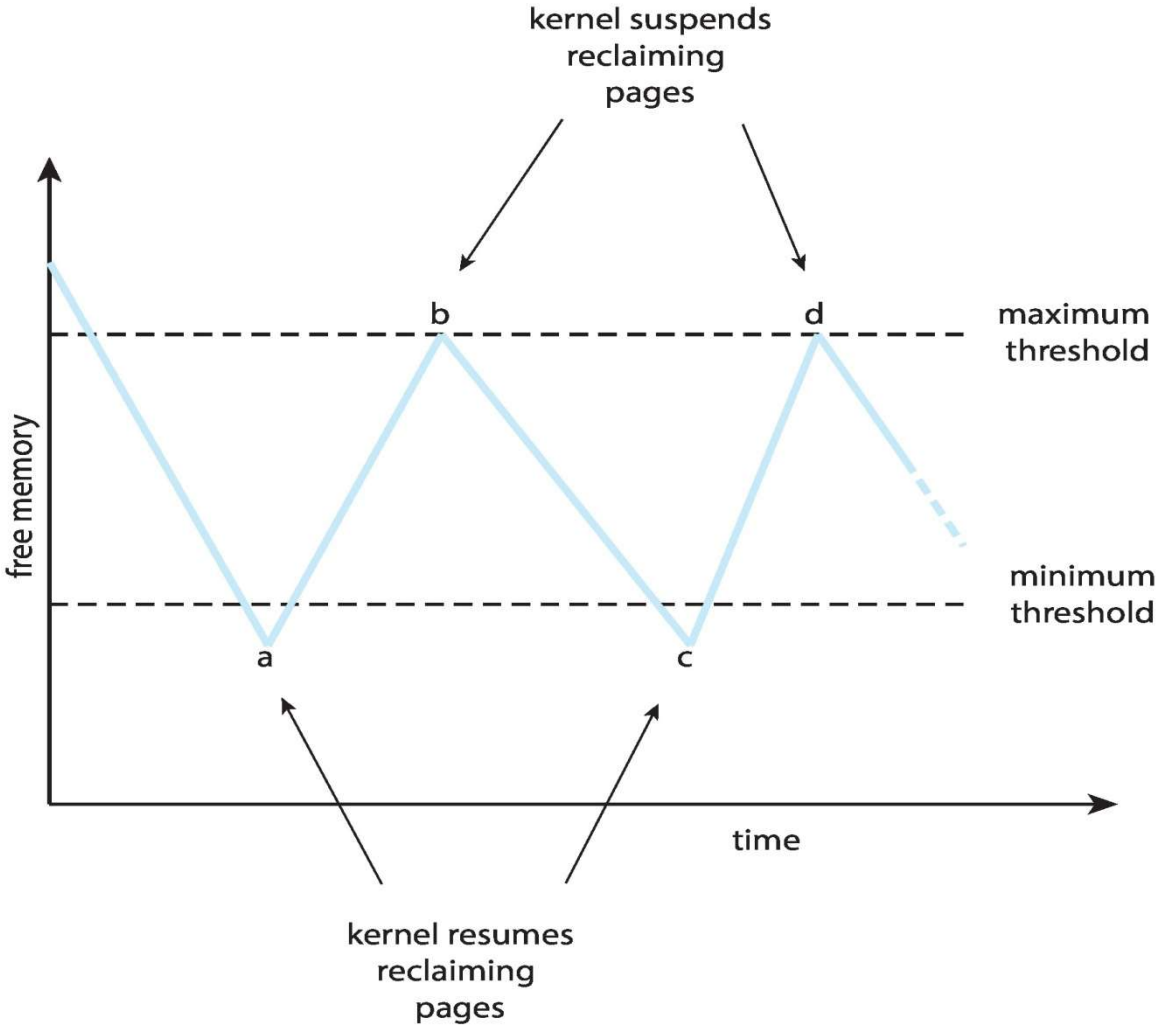
- A process can select a replacement from among its own frames or the frames of any lower-priority process
- This approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process
- Whereas with a local replacement strategy, the number of frames allocated to a process does not change

- One problem with a global replacement algorithm
  - is that the set of pages in memory for a process depends not only on the paging behavior of that process, but also on the paging behavior of other processes

- Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process
- Local replacement might hinder a process, however, by not, providing the less used pages of memory to other processes
- Global replacement generally results in greater system throughput
- It is therefore the more commonly used method

- Possible strategy to implement a global page-replacement policy
  - satisfy all memory requests from the free-frame list
  - But rather than waiting for the list to drop to zero selecting pages for replacement will begin before
  - Page replacement is triggered when the list falls below a certain threshold
  - This strategy attempts to ensure there is always sufficient free memory to satisfy new requests
  - This strategy is depicted in the next slide

Reclaiming pages



- The strategy's purpose is to keep the amount of free memory above a minimum threshold
- When it drops below this threshold, a kernel routine is triggered that begins reclaiming pages from all processes in the system
- Such kernel routines are often known as *reapers*
- When the amount of free memory reaches the maximum threshold, the reaper routine is suspended, only to resume once free memory again falls below the minimum threshold

Thrashing

- Consider what occurs if a process does not have “enough” frames; that is, it does not have the minimum number of frames it needs to support pages in the working set
- The process will quickly page-fault
- At this point, it must replace some page
- However, since all its pages are in active use, it must replace a page that will be needed again right away
- Consequently, it quickly faults again, and 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
- Thrashing results in severe performance problems

Cause of thrashing

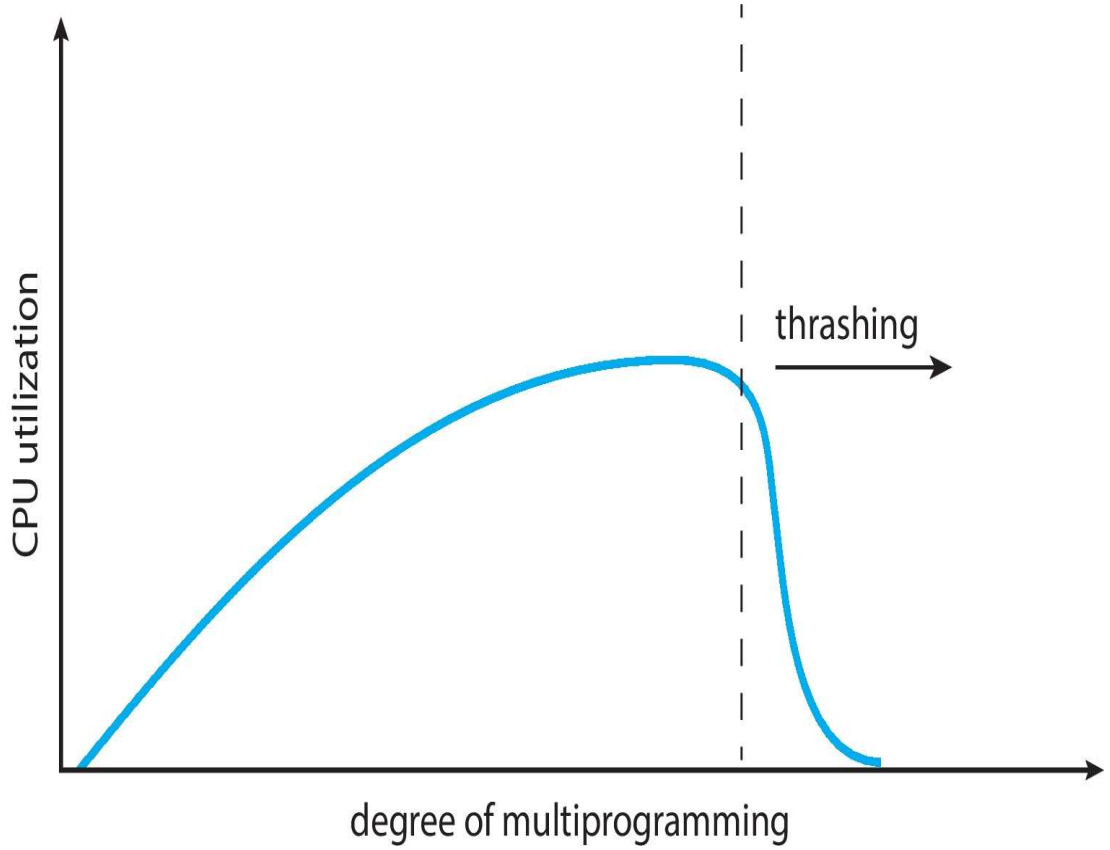
- Consider the following scenario, which is based on the actual behavior of early paging systems
- The operating system monitors CPU utilization
- If CPU utilization is too low, the degree of multiprogramming is increased by introducing a new process to the system
- A global page-replacement algorithm is used
  - it replaces pages without regard to the process to which they belong
- Now suppose that a process enters a new phase in its execution and needs more frames

- It starts faulting and taking frames away from other processes
- These processes need those pages, however, and so they also fault, taking frames from other processes
- These faulting processes must use the paging device to swap pages in and out
- As they queue up for the paging device, the ready queue empties
- As processes wait for the paging device, CPU utilization decreases

- The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result
- The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device
- As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more
- Thrashing has occurred, and system throughput plunges

- The page fault rate increases tremendously
- As a result, the effective memory-access time increases
- No work is getting done, because the processes are spending all their time paging
- This phenomenon is illustrated in Figure in the next slide
- Effects of thrashing can be limited by using a local replacement algorithm or priority replacement algorithm
  - Local replacement requires that each process select from only its own set of allocated frames

Thrashing

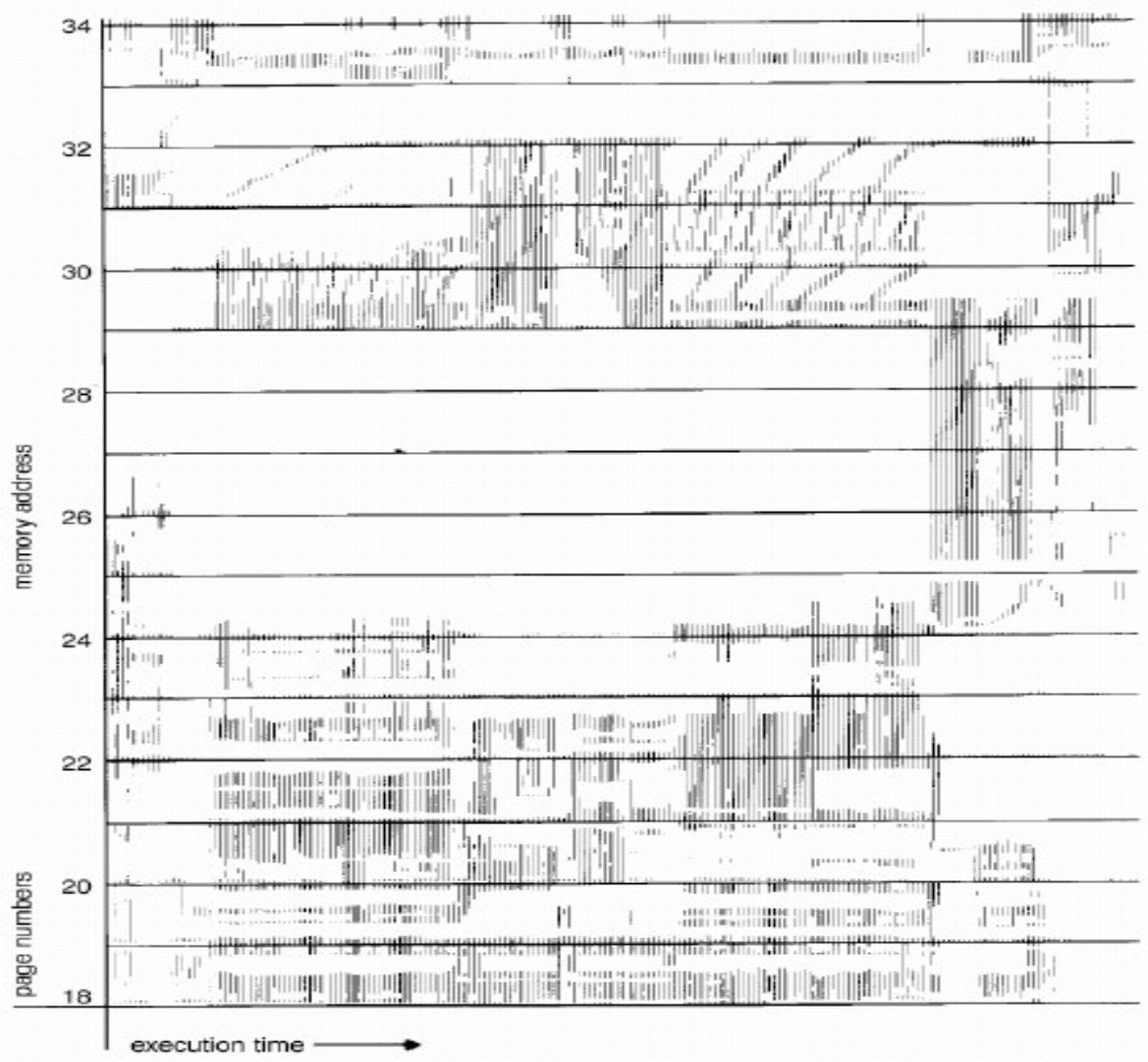


- Thus, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well
- However, the problem is not entirely solved
- If processes are thrashing, they will be in the queue for the paging device most of the time
- The average service time for a page 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

- Thrashing can be prevented using locality model approach of process execution
- The locality model states that, as a process executes, it moves from locality to locality
- A locality is a set of pages that are actively used together
- A running 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, since the local variables and instructions of the function are no longer in active use
- We may return to this locality later
- Figure illustrates the concept of locality and how a process's locality changes over time

Locality


34
32
30
28
26
24
22
20
18
memory address
page numbers
execution time

- At time (a), the locality is the set of pages {18,19,20,21,22,23,24,29,30,33}
- At time (b), the locality changes to {18,19,20,24,25,26,27,28,29,31,32,33}
- Notice the overlap, as some pages (for example, 18, 19, and 20) are part of both localities