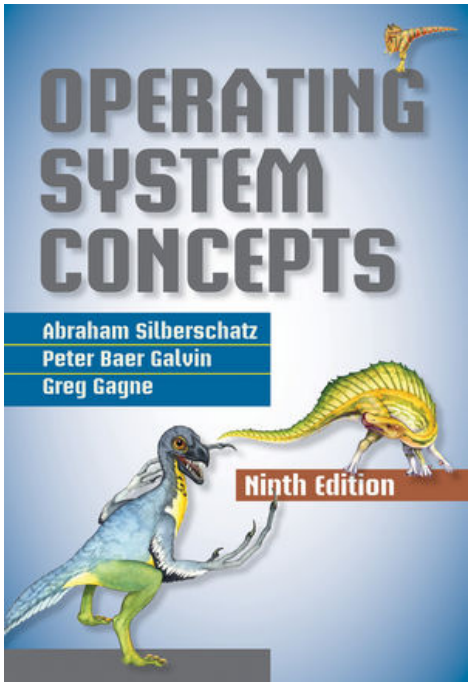
7 - Virtual Memory

EECE 315 (101)
ECE – UBC
2013 W2



Acknowledgement: This set of slides is partly based on the PPTs provided by the Wiley's companion website (including textbook images, when not explicitly mentioned/referenced).

Chapter 9: Virtual Memory

- **Background**

- Demand Paging
- Copy-on-Write
- Page Replacement

- Allocation of Frames
- Thrashing
- Allocating Kernel Memory

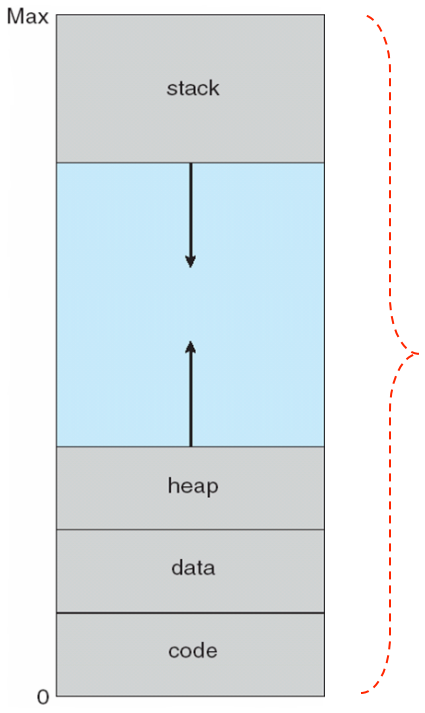
- Memory-mapped Files
- Other Considerations

Background

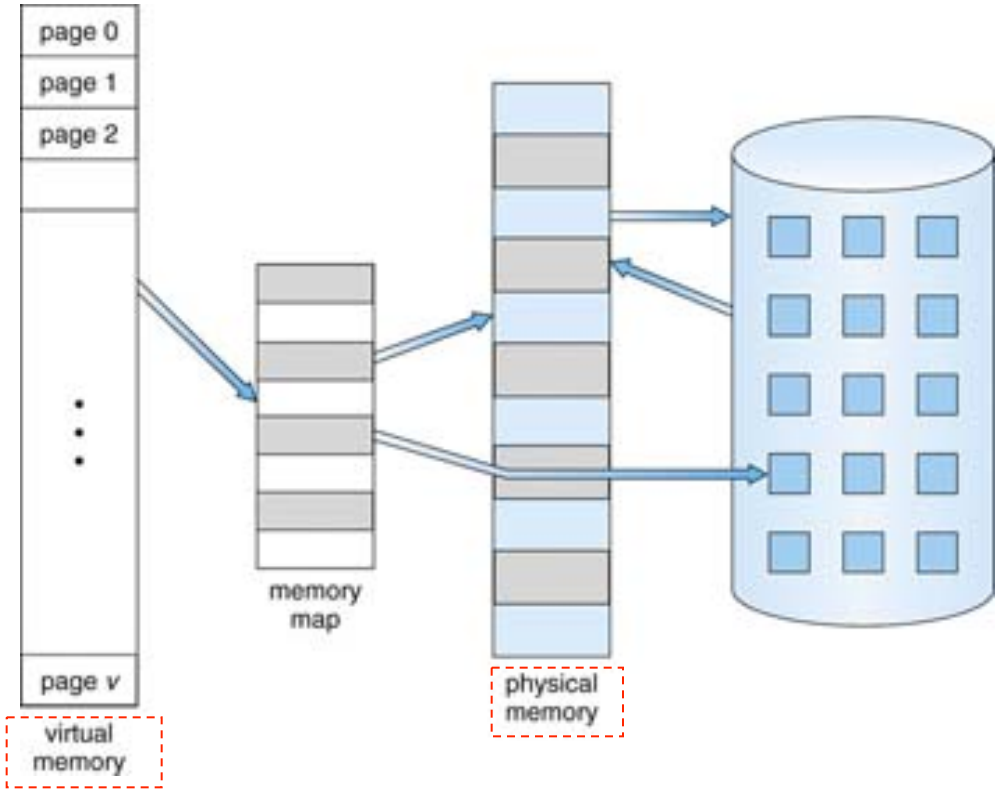
- All memory-management strategies we have discussed so far
  - are necessary: the instructions being executed must be in the physical memory,
  - have the same goal: to keep many processes in memory simultaneously
- Though they tend to require that an entire process be in memory
  - dynamic loading can help, but requires special precautions and extra work by the programmer
- Having 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
  - user programs take less physical memory
    - certain options, allocated memory and codes that are rarely used are loaded only when needed
  - less I/O would be necessary to load or swap user programs

Virtual Memory

- **Virtual memory** involves separation of user logical memory from physical memory
  - only part of the program needs to be in memory for execution
  - the logical address space can therefore be much larger than the physical address space



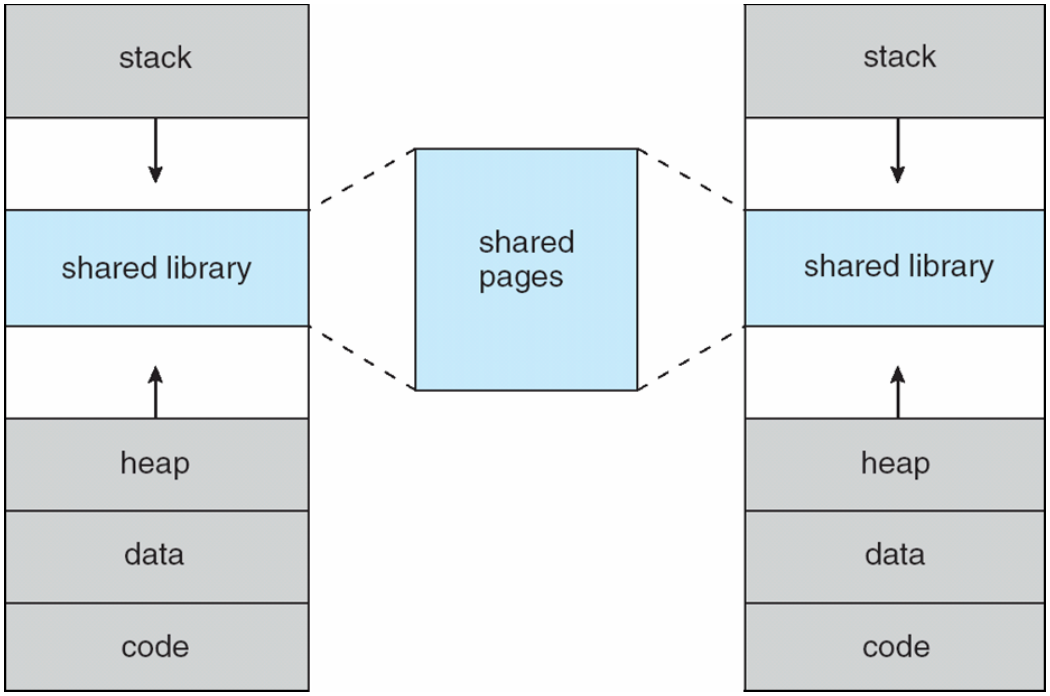
virtual address space



Virtual Memory (cont)

- **Virtual memory** also allows files and memory to be shared by two or more processes through page sharing

- System libraries can be shared by several processes through mapping of the shared object into a virtual address space
- Similarly, virtual memory enables processes to share memory



- Virtual memory allows pages to be shared during process creation with the `fork()` system call, thus it can help to speed up process creation

Chapter 9: Virtual Memory

- Background

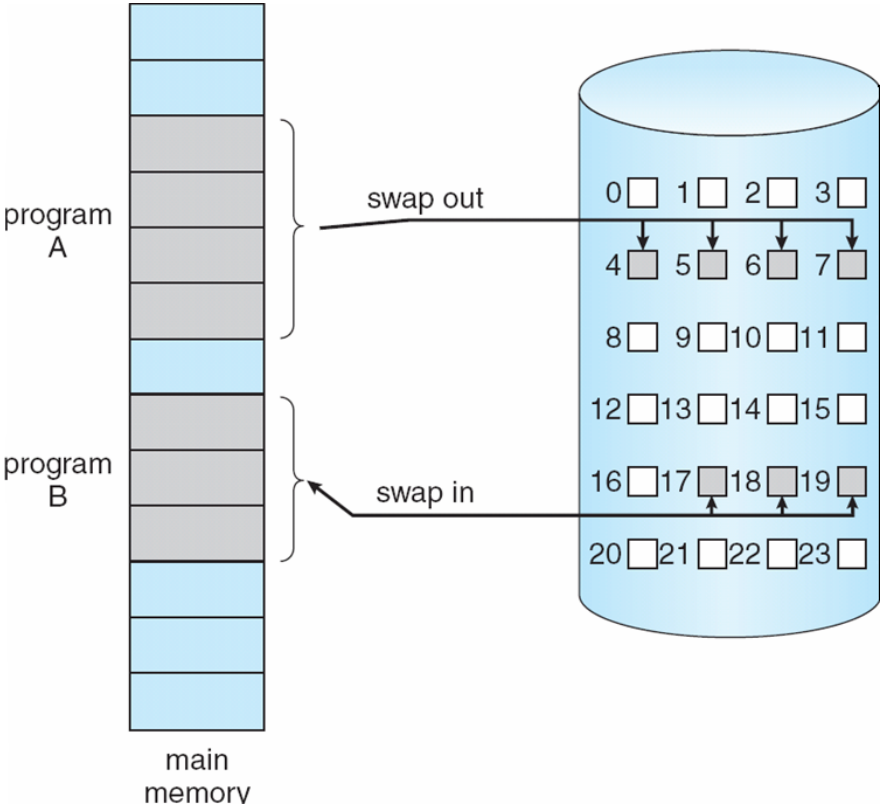
- **Demand Paging**
- Copy-on-Write
- Page Replacement

- Allocation of Frames
- Thrashing
- Allocating Kernel Memory

- Memory-mapped Files
- Other Considerations

Demand Paging

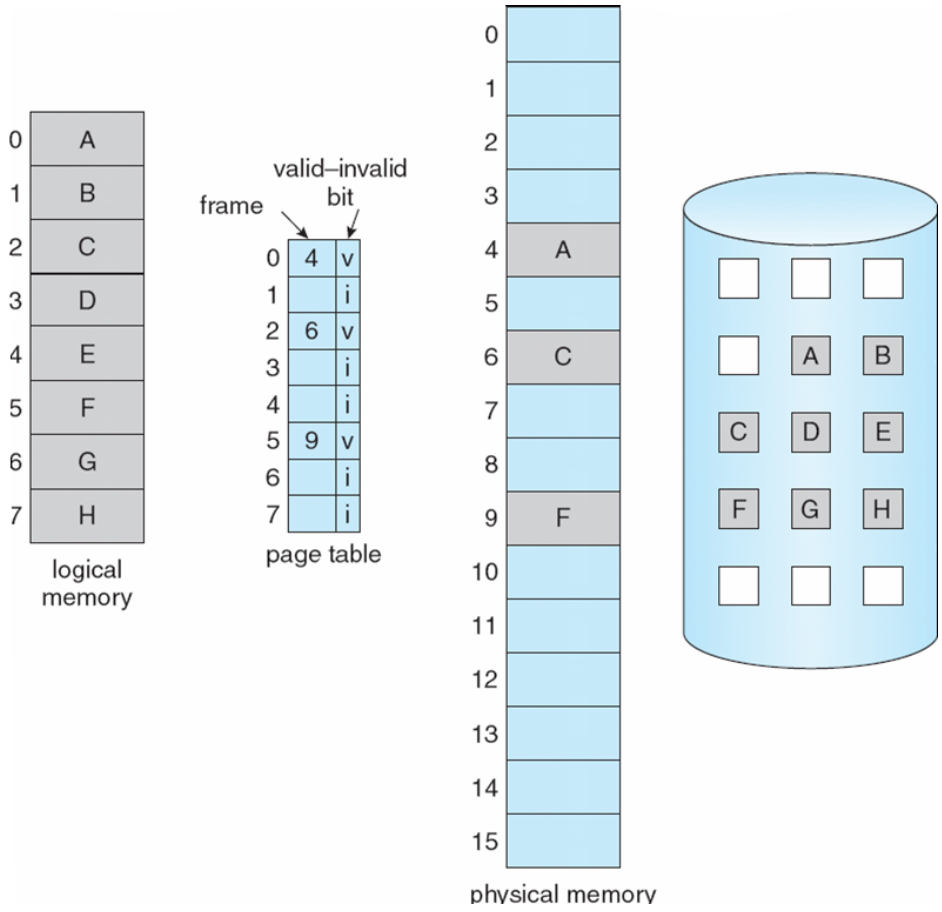
- Virtual memory can be implemented through **demand paging**:
  - load a page into memory only when it is needed
    - less memory needed
    - faster response
    - less I/O needed
    - more users
- Demand paging is similar to a paging system with swapping
  - to execute a process, we swap it into memory
  - however, we use a **lazy swapper** i.e. it never swaps a page into memory unless the page will be needed
    - a swapper that deals with pages is a **pager**



Basic Concepts

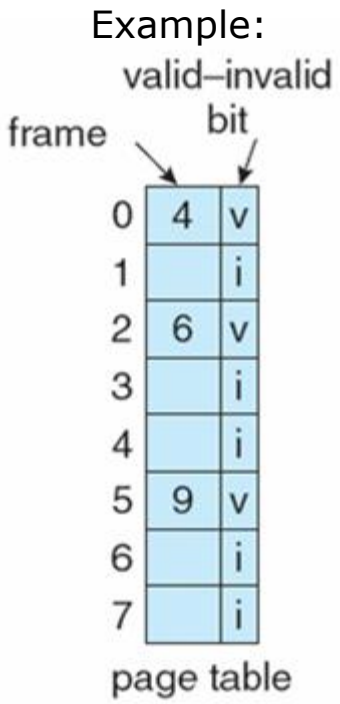
- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again

- With this scheme, we need some form of hardware support to distinguish between the pages that are in the memory and those on the disk
  - The valid-invalid bit scheme we discussed previously can be used for this purpose



Valid-Invalid Bit

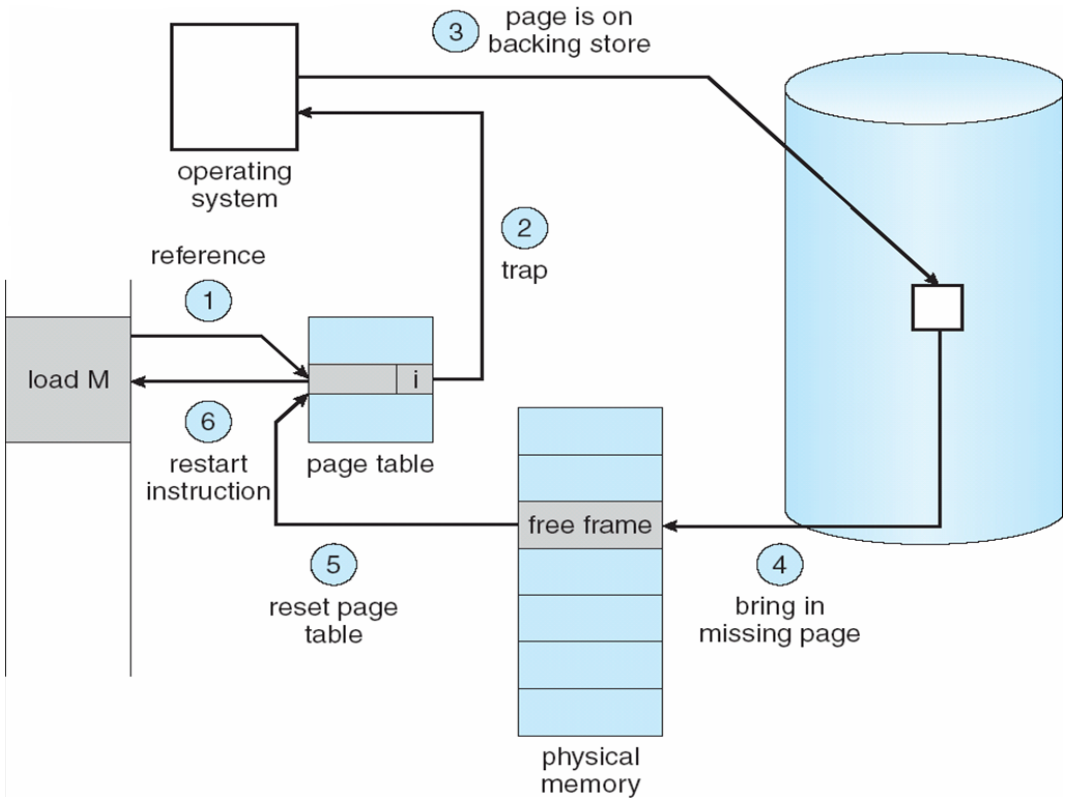
- With each page table entry a valid–invalid bit is associated
  - if set **valid (v)** ⇒ in-memory and legal
  - if set **invalid (i)** ⇒ not-in-memory or invalid



- a) The page-table entry for a page that is brought into memory is set as usual,
- b) but the page-table entry for a page not currently in memory is set to invalid
  - marking a page invalid will have no effect if the process never attempts to access that page
  - while the process executes and accesses pages that are memory resident, execution proceeds normally
  - though access to a page marked invalid causes a page fault

Page Fault

- The paging hardware, in translating the address through the page table, will notice if the invalid bit is set and causes a trap to the OS.
  - this trap is the result of the OS's failure to bring the desired page into memory
- The procedure for handling a **page fault** is straightforward



Page Fault (cont)

- The procedure for handling a **page fault** is straightforward (see the figure on the previous slide):
  1. operating system looks at another internal table to determine whether the reference was a valid or an invalid memory access
  2. if the reference is not valid, we terminate the program. Otherwise, we just need to page it in
  3. we find a free frame (if needed with a swap page out)
  4. we schedule a disk operation to read the page into newly allocated frame
  5. we now modify the internal table and page table to indicate the page is now in memory
  6. we restart the instruction that was interrupted by the trap

- So the time spent for page fault would be: page fault overhead + possible swap page out+ swap page in + restart overhead

Demand Paging (cnot)

- Unacceptable system performance would result, if some program accesses several new pages of memory (e.g. one for instruction and many for data)
  - although theoretically possible, this behavior is exceedingly unlikely
  - since programs tend to have locality of reference

- **Pure demand paging** is the extreme case: never bring a page into memory until it is required

- The hardware to support demand paging is the same as the hardware for paging and swapping
  - page table
  - swap space (a secondary memory)
  - a crucial requirement for demand paging is the ability to restart any instruction after a page fault

Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer
- Let *p* be the probability of a page fault (i.e. 0 ≤ *p* ≤ 1.0, page fault rate)
  - if *p* = 0: no page faults
  - if *p* = 1: every reference is a fault
  - we would expect *p* to be close to zero
  - The memory access time, *ma*, ranges from 10 to 200 ns
  - The **Effective Access Time** is then:

$$\text{effective access time} = (1 - p) \times ma + p \times \text{page fault time}$$

- To compute the effective access time, we must know how much time is needed to service a page fault
  - see page 403 in text (slide 8-10) for detailed sequence (for the Essential edition see page 325)
  - in any case, we are faced with three major components: service the page-fault interrupt, read the page, restart the process

Demand Paging Example

- Let's consider the following example:
- Assume that
  - memory access time = 200 nanoseconds
  - average page-fault service time = 8 milliseconds

$$\begin{aligned} \textit{Effective Access Time} &= (1 - p) \times 200 + p \times (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8{,}000{,}000 \\ &= 200 + p \times 7{,}999{,}800 \end{aligned}$$

- we see that the effective access time is directly proportional to *p*
  - If one access out of 1,000 causes a page fault, then:

    Effective Access Time = 8.2 microseconds

    which is a slowdown by a factor of 40.
  - For a performance degradation of less than 10% (i.e. effective access time of 220 here), we need $p < 0.0000025$.

Chapter 9: Virtual Memory

- Background

- Demand Paging
- **Copy-on-Write**
- Page Replacement

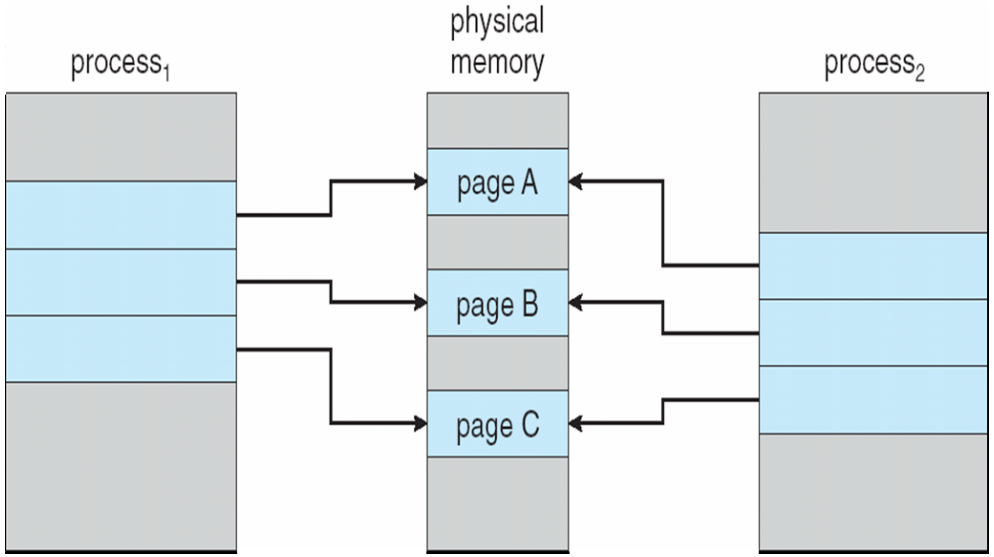
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory

- Memory-mapped Files
- Other Considerations

Copy-on-Write

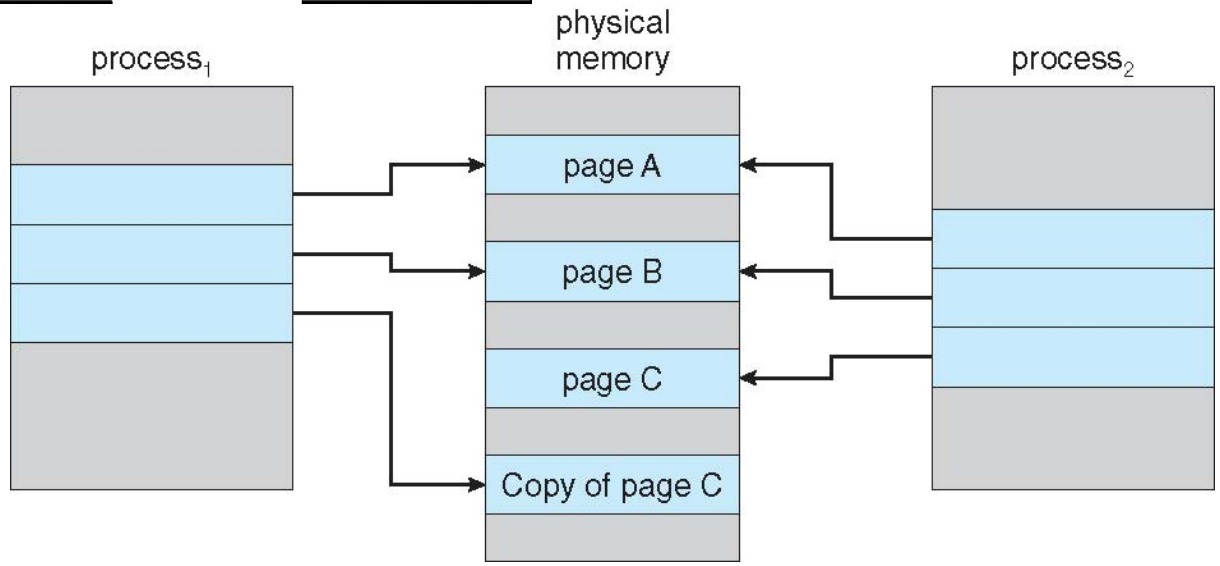
- Virtual memory allows other benefits during process creation
- Process creation using the `fork()` system call may initially bypass the need for demand paging by using a technique similar to page sharing
  - **Copy-on-Write** allows both parent and child processes to initially *share* the same pages in memory
    - If either process modifies a shared page, only then is the page copied
  - This technique provides for rapid process creation and minimizes the number of new pages that must be allocated to the newly created process
    - a more efficient process creation as only modified pages are copied
- Many OSs provide a pool of free pages from which a page is allocated
  - *Zero-fill-on-demand* pages have been zeroed-out before being allocated, thus erasing the previous contents

Example



Before process 1 modifies page C

After process 1 modifies page C:



Chapter 9: Virtual Memory

- Background

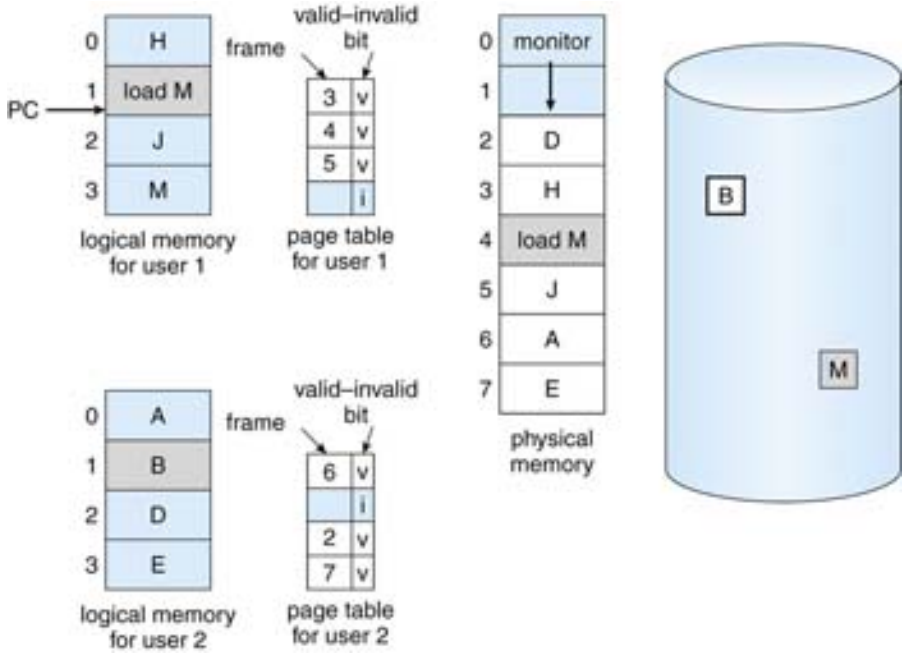
- Demand Paging
- Copy-on-Write
- **Page Replacement**

- Allocation of Frames
- Thrashing
- Allocating Kernel Memory

- Memory-mapped Files
- Other Considerations

Page Replacement

- If we increase our degree of multiprogramming, we may be over-allocating memory. For example:
  - assume while a user process is executing, a page fault occurs,
  - the OS system determines where the desired age is residing on the disk,
  - but then finds out that there are no free frame on the free-frame list



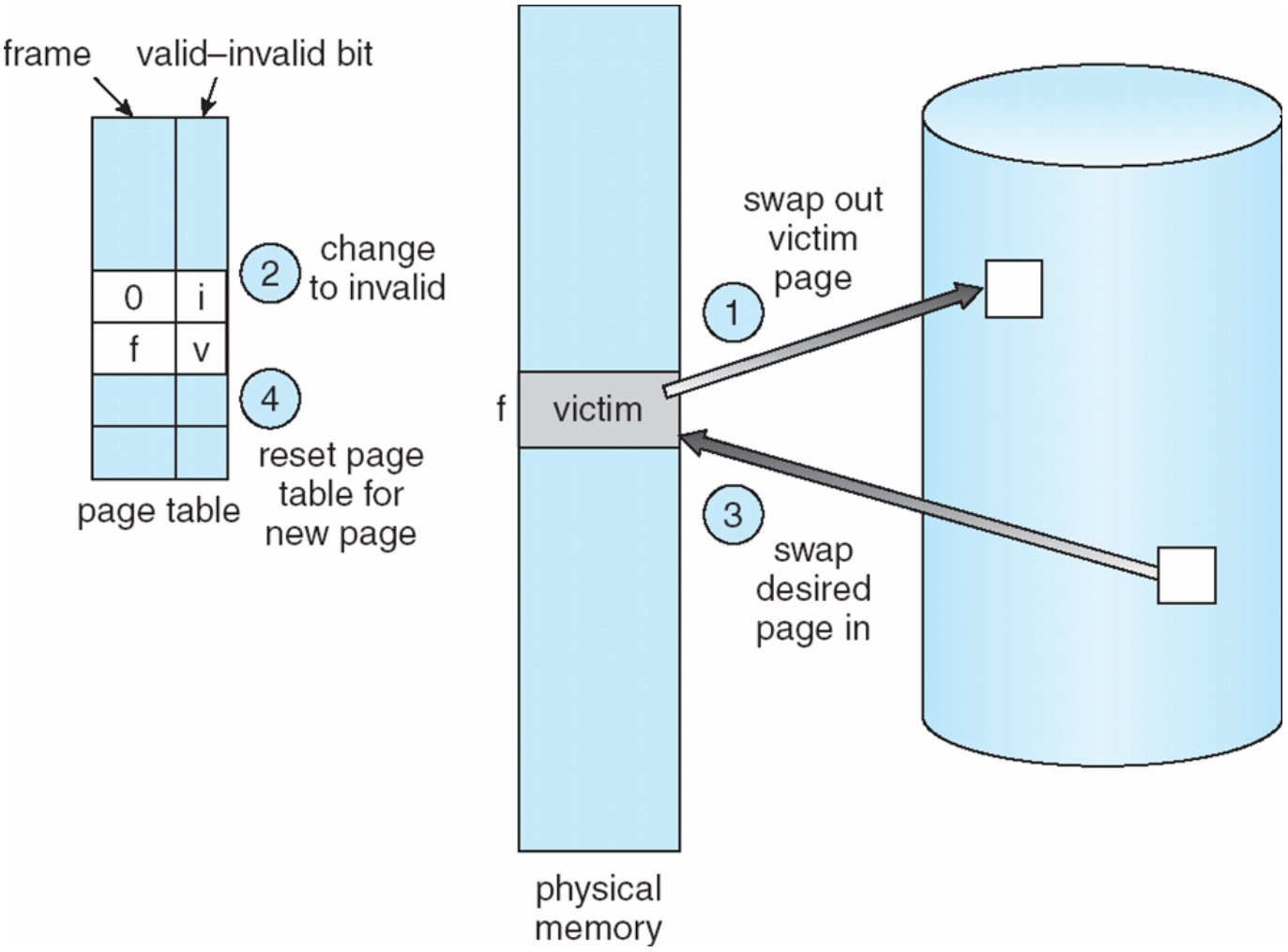
- Further note that system memory is not used only for holding program pages
  - buffers for I/O also consume a considerable amount of memory
  - deciding on how much to allocate to I/O and how much to program pages is a significant challenge

Basic Page Replacement

- The most common solution for the problem described in the previous slide is **page replacement**
  - if no frame is free, we find one that is not currently used and free it

- To modify the page-fault service routine to include 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
       - Write the victim frame to the disk; change the page and frame tables accordingly
  3. Read the desired page into the newly free frame; update the page and frame tables
  4. Restart the process

Page Replacement (cont)



Page Replacement (cont)

- In the previous figure, if no frames are free, two page transfers (one out and one in) are required
- We can reduce this overhead by using **modify (dirty) bit**:
  - the hardware sets the modify bit of a page whenever any word/byte in the page in written into
  - then only modified pages are written to disk
- This technique is also applied to read-only pages

- Page replacement is basic to demand paging and completes the separation between logical memory and physical memory:
  - large virtual memory can be provided on a smaller physical memory

Demand Paging (cont)

- We must solve two major problems to implement demand paging:
  - frame-allocation algorithm
    - if we have multiple processes in memory, we must decide how many frames to allocate to each process
  - page-replacement algorithm
    - when page replacement is required, we must select the frames that are to be replaced
    - performance: we want an algorithm which will result in minimum number of page faults
      - same page may be brought into memory several times

- Even slight improvement in demand-paging methods may yield large gains in system performance

frame allocation is discussed on slide 8-35

Page Replacement Algorithms

- In general, we want the lowest page-fault rate
- We evaluate an algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - we can generate reference strings by using a random number generator
  - or we can trace a given system and record the address of each memory reference
- For example, if we trace a particular process, we might record the following address sequences:

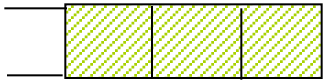
  0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, …

  At 100 bytes per page, the sequence is reduced to the following reference string:

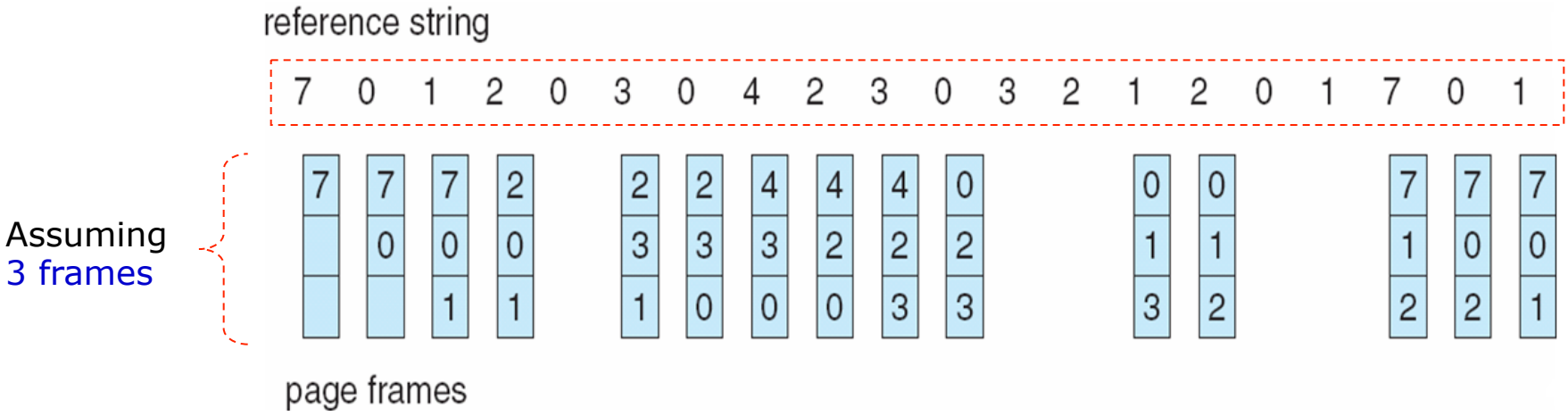
  1, 4, 1, 6, 1, 6, …

First-In-First-Out (FIFO) Algorithm

- The simplest page-replacement algorithm is first-in, first-out (FIFO) algorithm.
  - it associate with each page the time when that page was brought into memory,
  - when a page must be replaced, the oldest page is chosen
- Note that it is not necessary to record the time when a page is brought in.
  - We can create a FIFO queue to hold all pages in memory

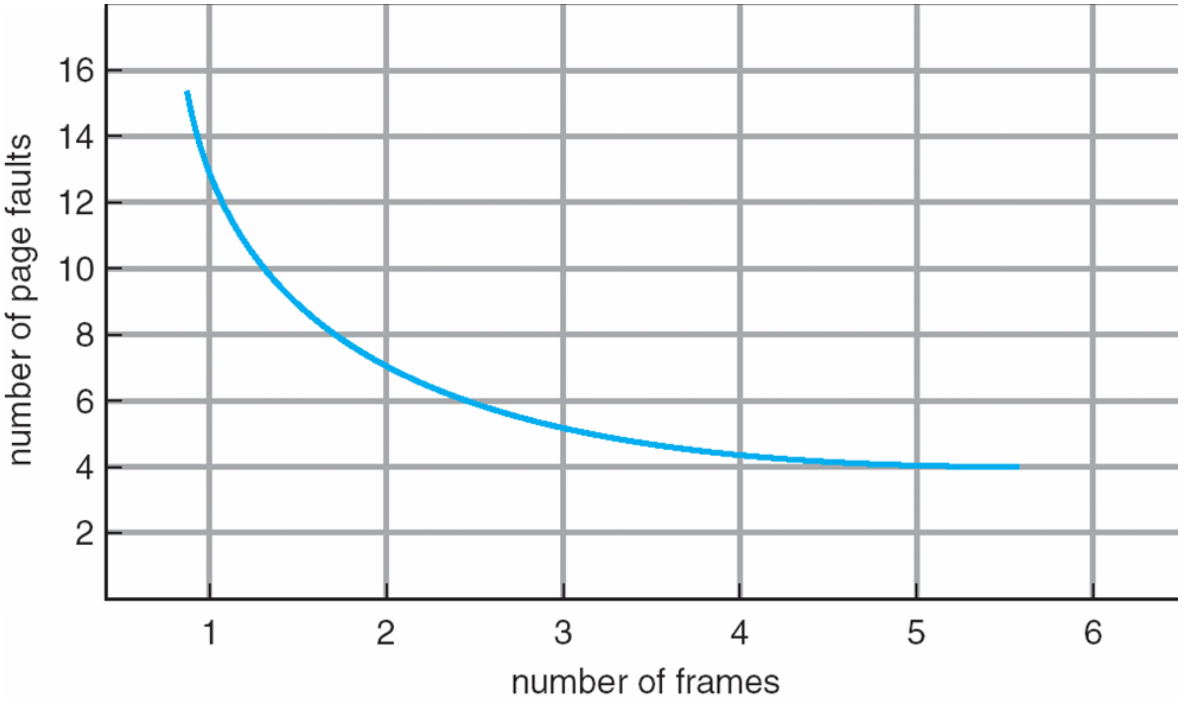


- Example: (15 page faults)



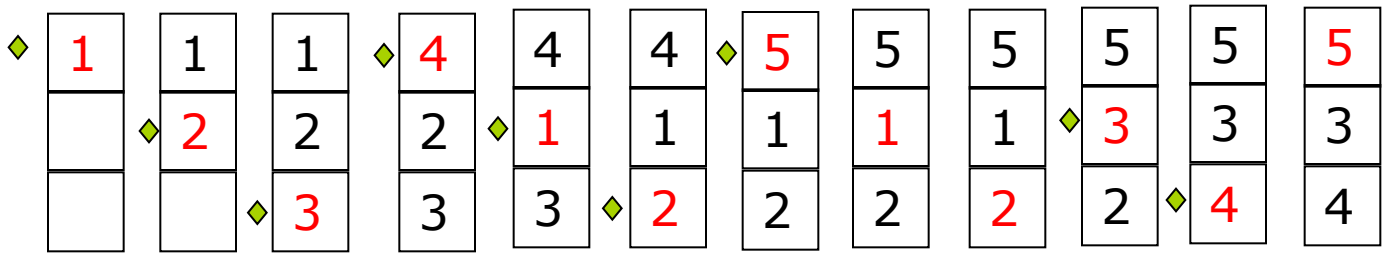
Graph of Page Faults Versus The Number of Frames

- To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available
- In general, we expect a curve such as the figure below:
  - as the number of available frames increases, the number of page faults decreases

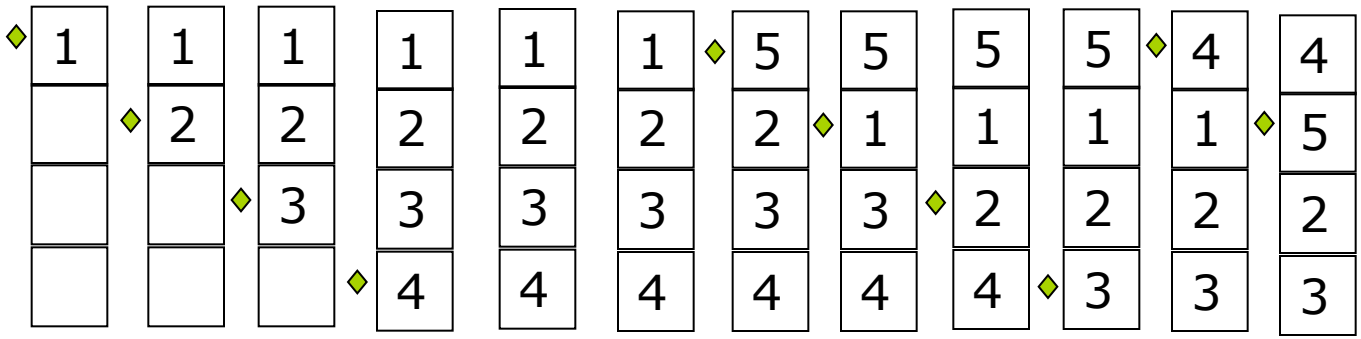


Example 2: FIFO Algorithm

- Consider another reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - if we assume 3 frames, how many page faults ?

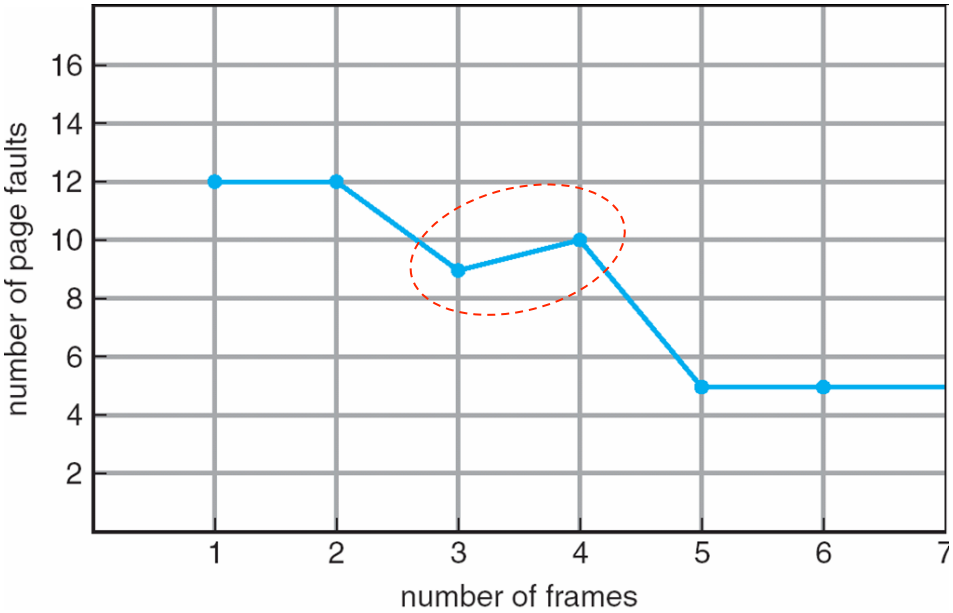


  - if we assume 4 frames, how many page faults ?



FIFO Illustrating Belady's Anomaly

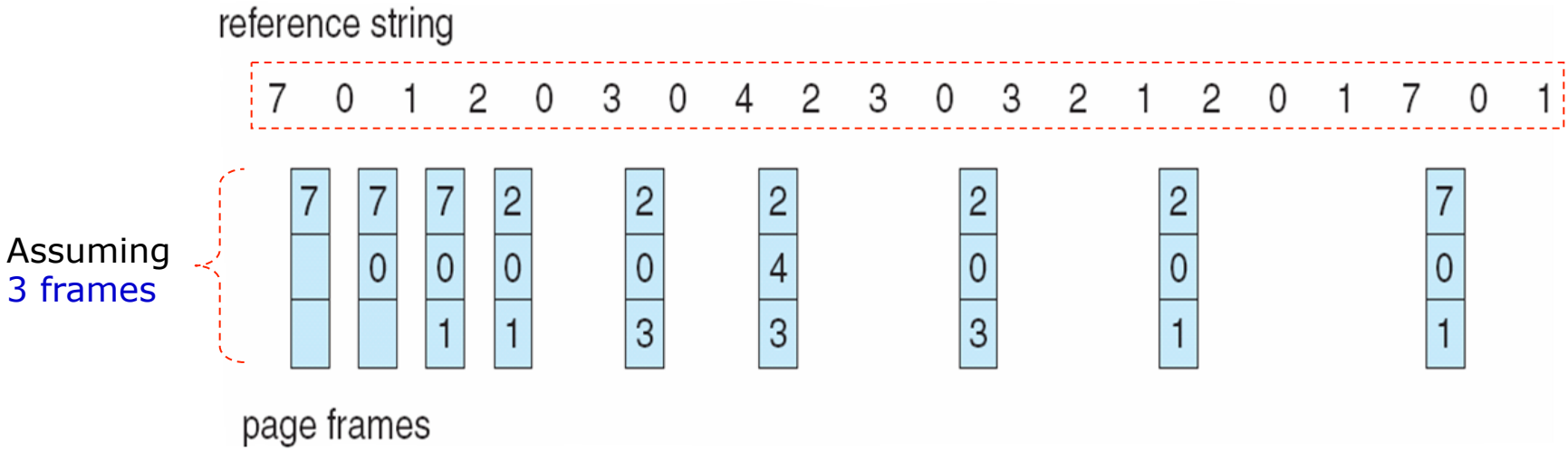
- If we draw the curve of page faults for this reference string versus the numbers of available frames, we have



- Belady's Anomaly:
  - sometimes more frames ⇒ more page faults!!
  - i.e. for some page replacement algorithms, the page-fault rate **may increase** as the number of allocated frames increases

Optimal Algorithm

- An optimal page-replacement algorithm:
  - has the lowest page-fault rate of all algorithms and
  - will never suffer from Belady's anomaly
- Such an algorithm is called OPT or MIN:
  - Replace the page that will not be used for the longest period of time
- Example
  - on our sample reference string, this algorithm would yield 9 page faults:

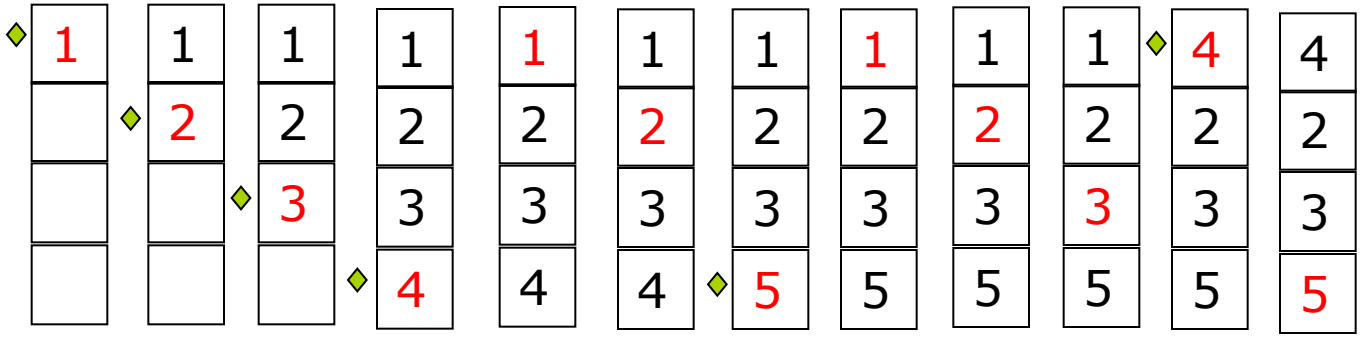


Optimal Algorithm (cont)

- As another example consider

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

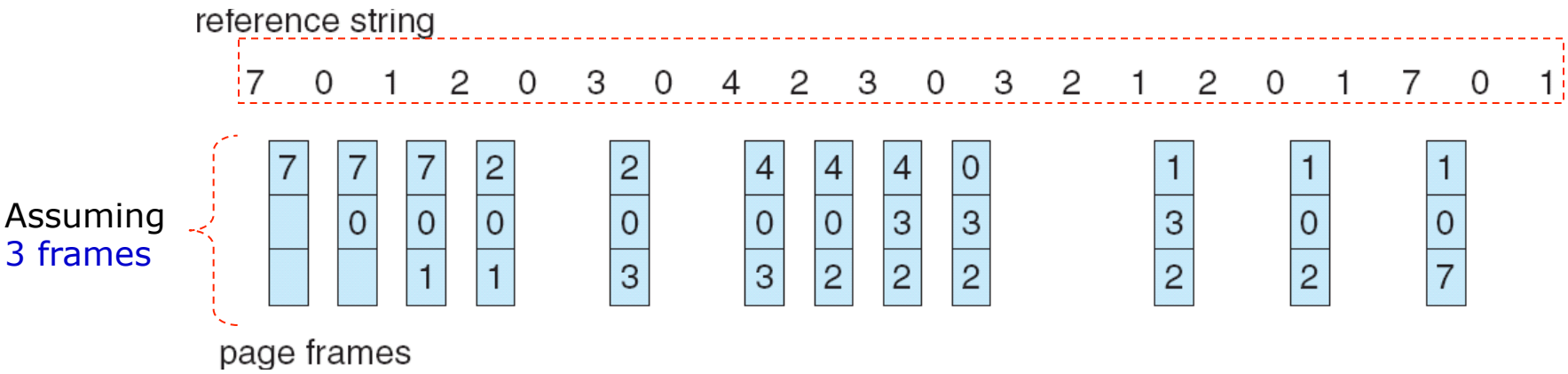
  - for 4 frames, we have 6 page faults



- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string
  - as a result, it is used mainly for comparison studies:
    - i.e. it is used to measure how well an algorithm performs
- The key difference between the FIFO and OPT (other than looking backward versus forward in time) is that :
  - the FIFO algorithm uses the time when a page was brought into memory
  - whereas the OPT algorithm uses the time when a page is to be used

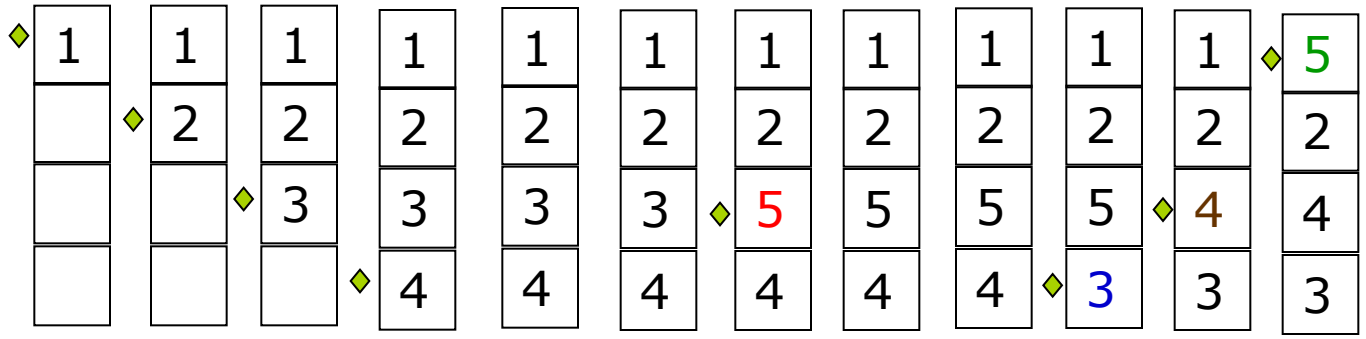
Least Recently Used (LRU) Algorithm

- If the optimal algorithm is not feasible, perhaps an approximation of it is possible.
  - If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period of time.
  - This approach is the least-recently-used (LRU) algorithm
  - LRU 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
- As an example:



LRU Algorithm (cont)

- As another example consider reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

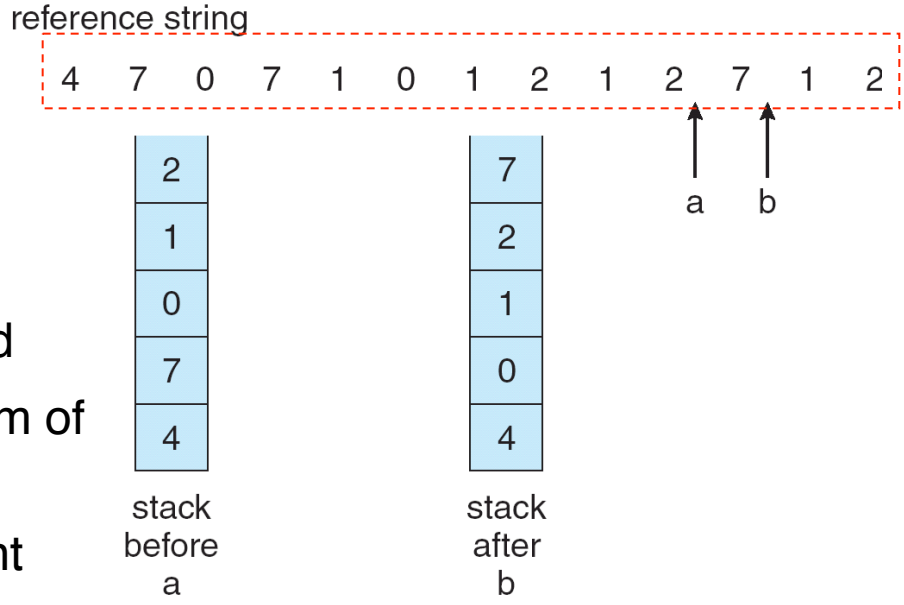


- The LRU policy is often used as a page-replacement algorithm and is considered to be good.
  - Like optimal replacement, LRU does not suffer from Belady's anomaly.
- Implementing LRU may require substantial hardware support:
  - the problem is to determine an order for the frames defined by the time of last use
- Two implementations are feasible:
  - counters
  - stack

LRU Algorithm (Cont.)

- **Counter implementation**
  - every page entry has a counter; every time the page is referenced through this entry, copy the clock into the counter
  - when a page needs to be changed, we replace the page with the smallest time value
  - though we need a search to find the LRU page and a write to memory for each memory access.

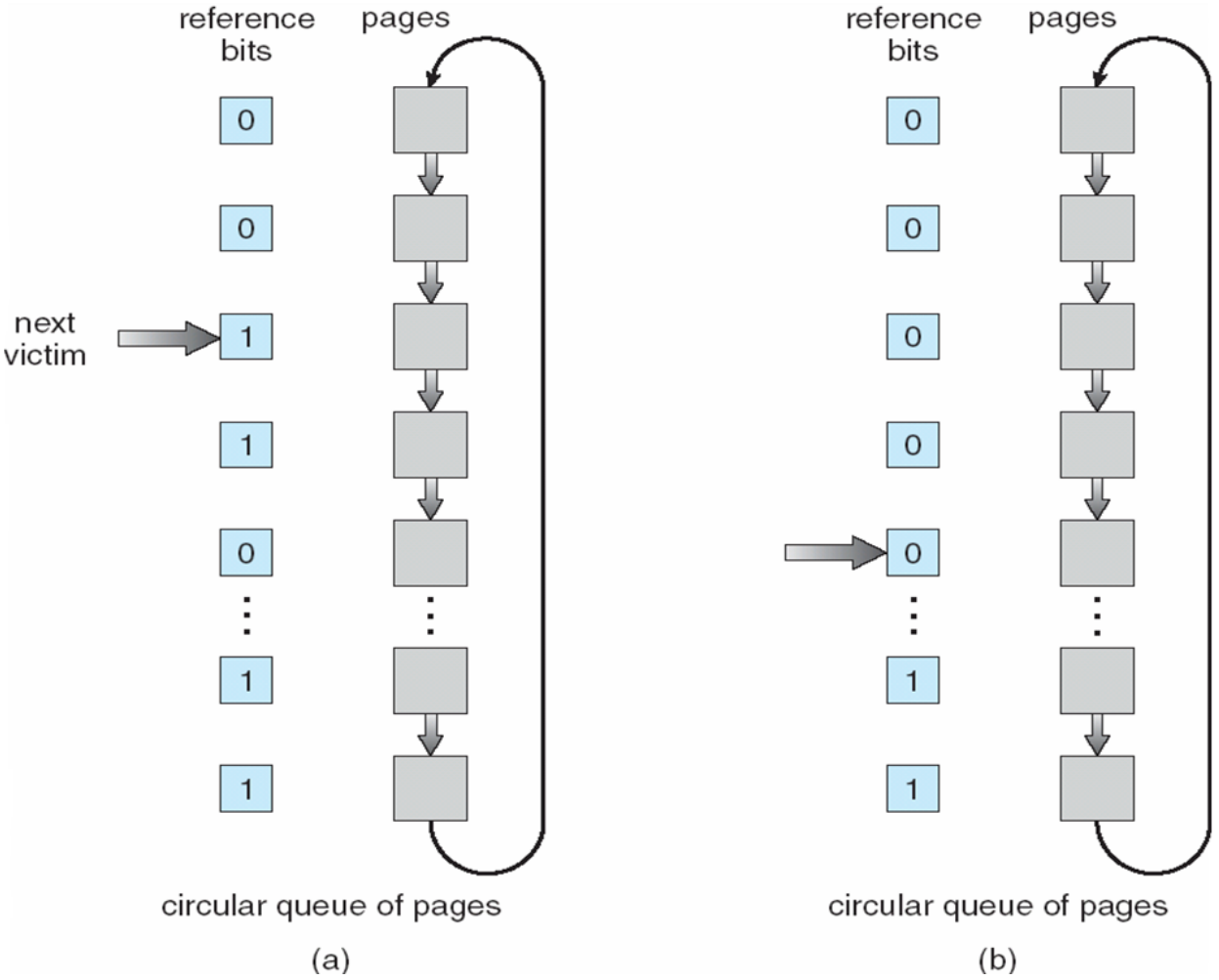
- **Stack implementation**
  - keep a stack of page numbers
  - whenever a page is referenced:
    - move it to the top
      - doubly linked lists are used
  - the tail pointer points to the bottom of the stack which is the LRU page.
  - there is no search for replacement



LRU Approximation Algorithms

- Few computer systems provide sufficient hardware support for true LRU page replacement. Many systems provide some help, however, in the form of a reference bit:
  - with each page associate a bit, initially = 0
  - when page is referenced, the bit set to 1
  - replace the one which is 0 (if one exists)
    - we do not know the order, however (approximate LRU)
  - additional-reference-bits algorithm uses an 8-bit byte for each page
- The basic algorithm of second-chance replacement is a FIFO:
  - it is based on using the reference bit based on a circular queue
  - inspect the reference bit of a selected page (in clock order)
    - If value is 0, replace this page
    - If reference bit = 1 then leave the page in memory (giving the page a second chance), but set reference bit 0
      - replace the next page (in clock order), subject to the same rules

Second-Chance (clock) Page-Replacement Algorithm



Counting Algorithms

- In counting-based algorithms, we can keep a counter of the number of references that have been made to each page, and develop the following two schemes
  - **Least Frequently Used (LFU) algorithm**: replaces the page with the smallest count
  - **Most Frequently Used (MFU) Algorithm**: is based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Neither MFU or LFU is common
  - they are expensive and do not approximate OPT well

Chapter 9: Virtual Memory

- Background

- Demand Paging
- Copy-on-Write
- Page Replacement

- **Allocation of Frames**
- Thrashing
- Allocating Kernel Memory

- Memory-mapped Files
- Other Considerations

Allocation of Frames

- We now turn to the issue of allocation:
  - how do we allocate the fixed amount of free memory among the various process?
- There are many variations
  - The simplest case is the single-user system under pure demand paging
- The strategies for the allocation of frames are constrained in various ways:
  - we can only allocate up to the total number of available frames (unless there is page sharing)
  - each process needs a *minimum* number of pages
    - performance: fewer # of frames => higher page fault rate
  - the minimum number of frames is defined by the computer architecture
    - e.g. if memory-reference instructions reference only one memory address: we need at least one frame for instructions and one frame for memory reference

Allocation Algorithms

- The easiest way to split *m* frames among *n* processes is to give everyone an equal share, *m/n* frames.
  - this scheme is called equal allocation
  - for example, if there are 93 frames and 5 processes, give each process 18 frames and use the 3 leftover frames as free-frame buffer pool
- An alternative is proportional allocation
  - different processes need differing amounts of memory
  - e.g. 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 $p_i = \frac{s_i}{S} \times m$

e.g.:

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Allocation Algorithms (cont)

- With either equal or proportional allocation, a high-priority process is treated the same as a low-priority process
  - by definition, however, we may want to give a higher priority process more memory to speed its execution (possibly to the detriment of low-priority processes)

- One solution is to use a proportional allocation scheme using priorities rather than size (or in addition to)
  - the ratio of frames depends not on the relative sizes of processes but rather on
    - priorities of processes or
    - a combination of size and priority

Global vs. Local Allocation

- Another important factor in the way frames are allocated to the various processes is page replacement
- We can classify page-replacement algorithms into two broad categories:
  - Global replacement
    - allows a process to select a replacement frame from the set of all frames
    - for example, if a high priority process generates a page fault,
      - the process can select for replacement one of its own frames
      - or it can select for replacement a frame from a process with a lower priority number
      - i.e. one process can take a frame from another to increase its frame allocation at the expense of a low-priority process
  - Local replacement
    - each process selects from only its own set of allocated frames
    - with this strategy, the number of frames allocated to a process does not change

Chapter 9: Virtual Memory

- Background

- Demand Paging
- Copy-on-Write
- Page Replacement

- Allocation of Frames
- **Thrashing**
- Allocating Kernel Memory

- Memory-mapped Files

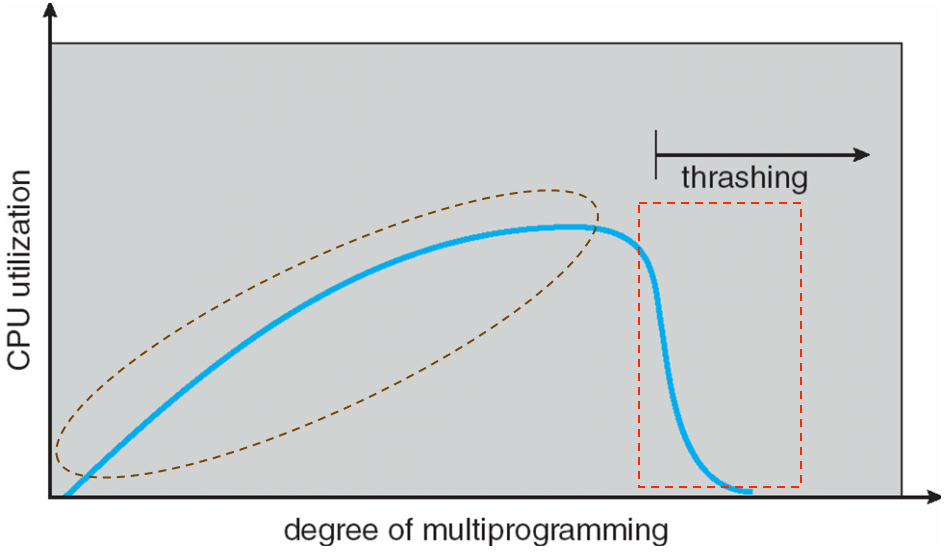
Thrashing

- Consider a process that does not have “enough” pages
  - if the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault
  - since all of 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
  - so, the page-fault rate becomes very high

- 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

Thrashing (cont)

- Consider the following scenario (based on early paging systems):
  - the OS monitors CPU utilization
  - if a process does not have “enough” pages, the page-fault rate becomes 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



Preventing Thrashing

- We can limit the effect of thrashing by using a local replacement algorithm
  - if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well
  - but the problem is not entirely solved
    - as the average service time for page fault will increase due to the longer average queue for the paging device

- To prevent thrashing, we must provide a process with as many frames as it needs
  - how do we know how many frames it needs?
  - One technique is working-set strategy:
    - starts by looking at how many frames a process is actually using
    - this approach defines the locality model of process execution
    - note that locality model is the unstated principle behind the caching discussions so far in this course

Locality Model

- 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 program is generally composed of several different localities
- For example,
  - in a program when a function is called, it defines a new locality
    - memory references are made to the instructions of the function call, its local variable and a subset of global variables
  - when we exit the function, the process leaves this locality
- Locality is defined by the program structure and its data structures

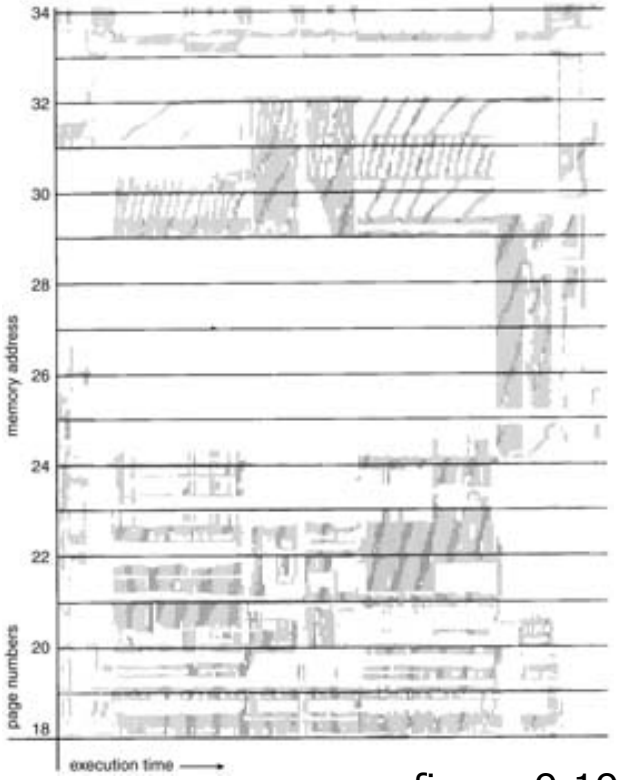
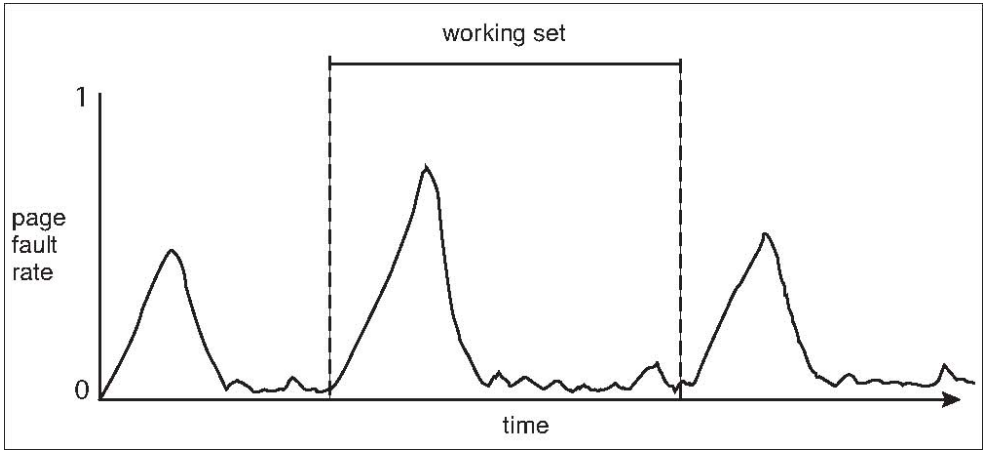


figure 9.19

Working-Set Model

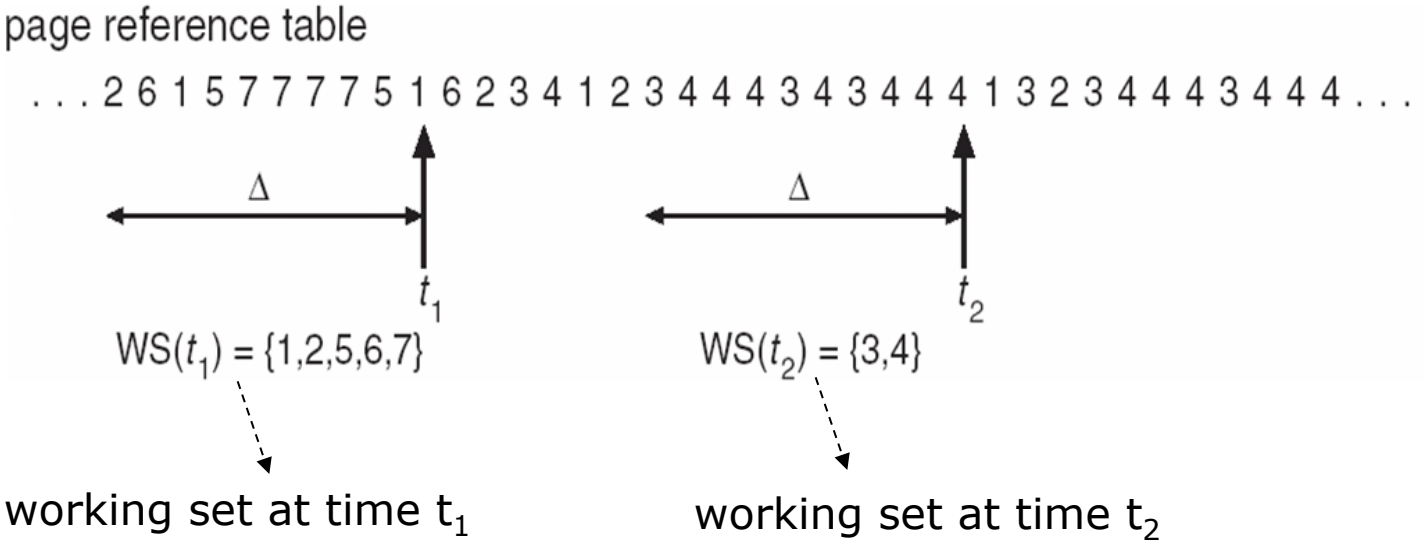
- The working-set model is based on the assumption of locality



- The model uses a parameter, Δ, to define the working-set window
  - the idea is to examine the most recent Δ page references
  - this working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible
  - the difficulty though is keeping track of the working set

Working-Set Model - Example

- Working set:
  - if a page is in active use, it will be in the working set
  - if it is no longer being used, it will drop from the working set Δ time units after its last reference
  - so the working set is an approximation of the program's locality
- For example:
  - if Δ is 10 memory references:



Working-Set Model (cont)

- The most important property of the working-set is its size
  - if Δ is too small, it will not encompass the entire locality
  - if Δ is too large, it may overlap several localities
    - in the extreme, if $\Delta = \infty$, it will encompass entire program

- Let WSS_i be the working set of Process P_i, *then* the total demand for frames is

$$D = \sum WSS_i$$

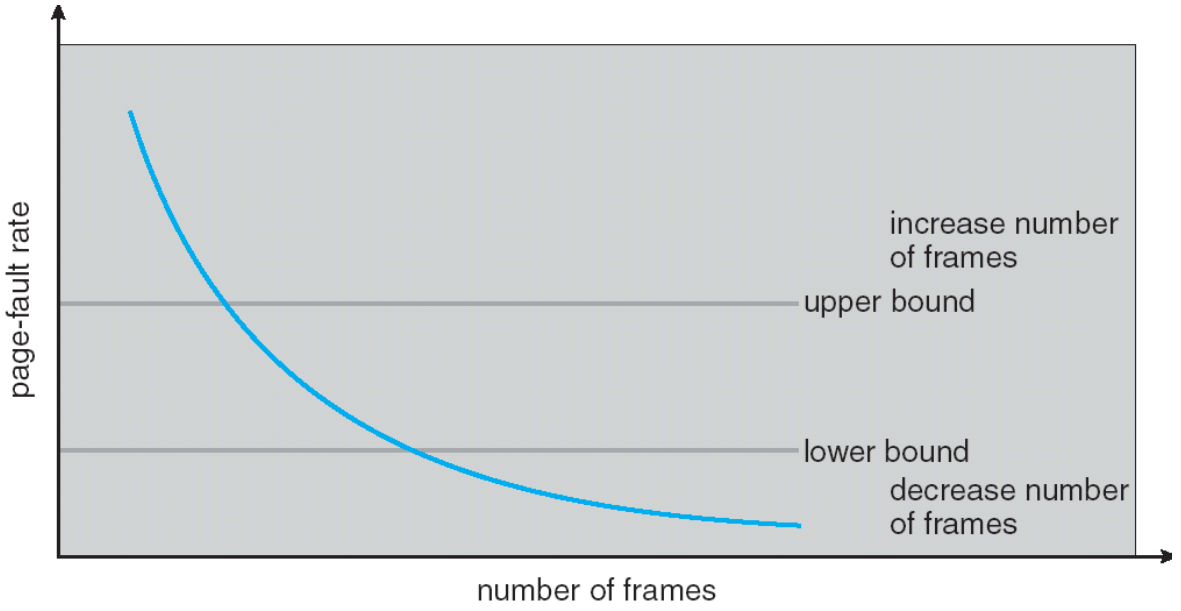
  - if the total demand is greater than the total number of available frames ($D > m$), thrashing will occur
- Once Δ has been selected, use of working-set model is simple:
  - the OS monitors the working-set of each process and allocates to that working-set enough frames to provide it with its working-set size
  - if there are enough frames, another process may be initiated
  - if the sum of working-set sizes exceeds the total number of available frames, the OS selects a process to suspend

Keeping Track of the Working Set

- We can approximate the working-set model with a fixed-interval timer interrupt and a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts, copy and clear the reference bit values for each page
  - If one of the bits in memory = 1 $\Rightarrow$ page is in working set
- Why is this not completely accurate?
  - because we cannot tell where, within an interval of 5000, a reference occurred
  - we can reduce the uncertainty by increasing the number of history bits and the frequency of interrupts (of course, at the cost)
    - e.g. in the above example:10 bits and interrupt every 1000 time units

Page-Fault Frequency Scheme

- Another strategy to prevent thrashing uses the page-fault frequency
  - the scheme takes a more direct approach
  - the objective is to establish an acceptable page-fault rate
    - If the page-fault rate is too high, the process needs more frames
    - If the page-fault rate is too low, the process may have too many frames
  - we can establish upper and lower bounds on the desired page-fault rate:



Chapter 9: Virtual Memory

- Background

- Demand Paging
- Copy-on-Write
- Page Replacement

- Allocation of Frames
- Thrashing
- **Allocating Kernel Memory**

- Memory-mapped Files

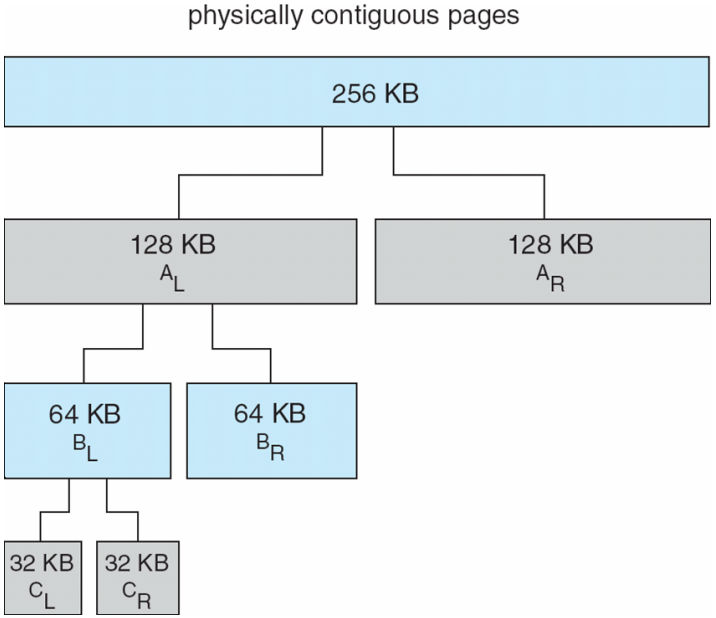
Allocating Kernel Memory

- Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode process for two reasons:
  1. the kernel requests memory for data structures of varying sizes, some of which are less than a page in size
  2. certain hardware devices interact directly with physical memory (without the benefit of a virtual memory interface) and may require memory residing in contiguous pages
     - recall that pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory

- One strategy for managing free memory for kernel processes is the Buddy system

Buddy System

- The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages
- Memory is allocated from this segment using a power-of-2 allocator
  - it satisfies requests in units sized as power of 2
    - 4 KB, 8 KB, 16 KB and so forth
  - a request is rounded up to next highest power of 2
    - this very likely causes fragmentation
  - When a smaller allocation is needed than is available, a current chunk is split into two buddies of next-lower power of 2
    - this will continue until an appropriate sized chunk is available
  - the advantage of this system is how quickly adjacent buddies can be combined to form a larger segment (coalescing)



Chapter 9: Virtual Memory

- Background

- Demand Paging
- Copy-on-Write
- Page Replacement

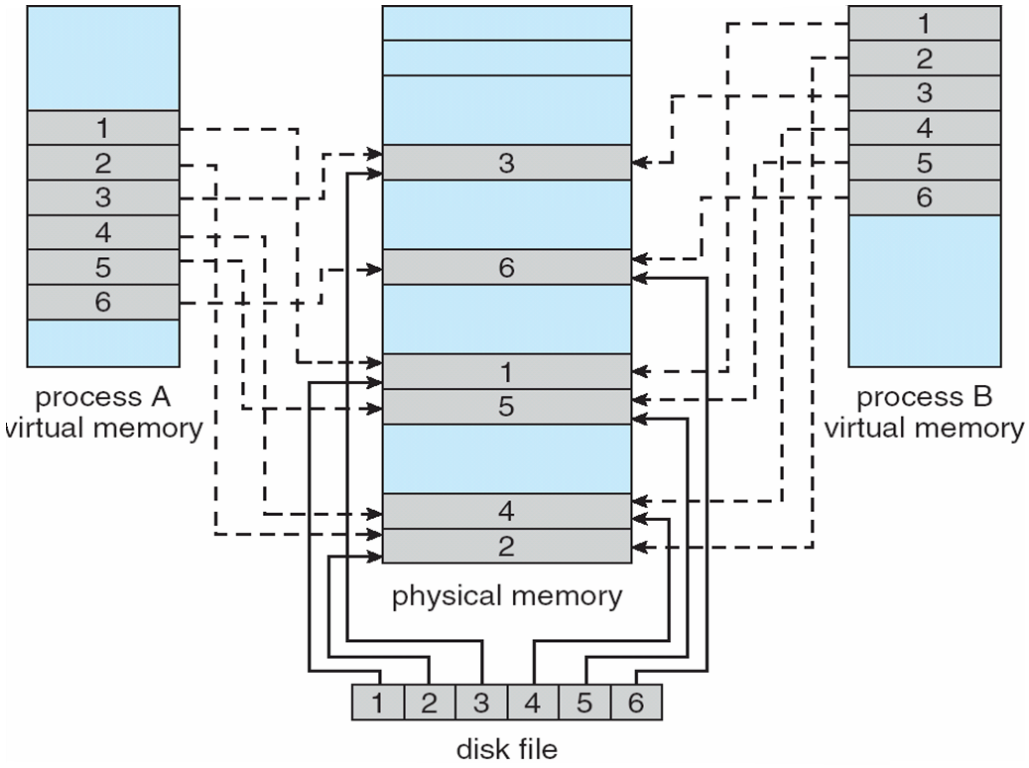
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory

- **Memory-mapped Files**
- Other Considerations

Memory-Mapped Files

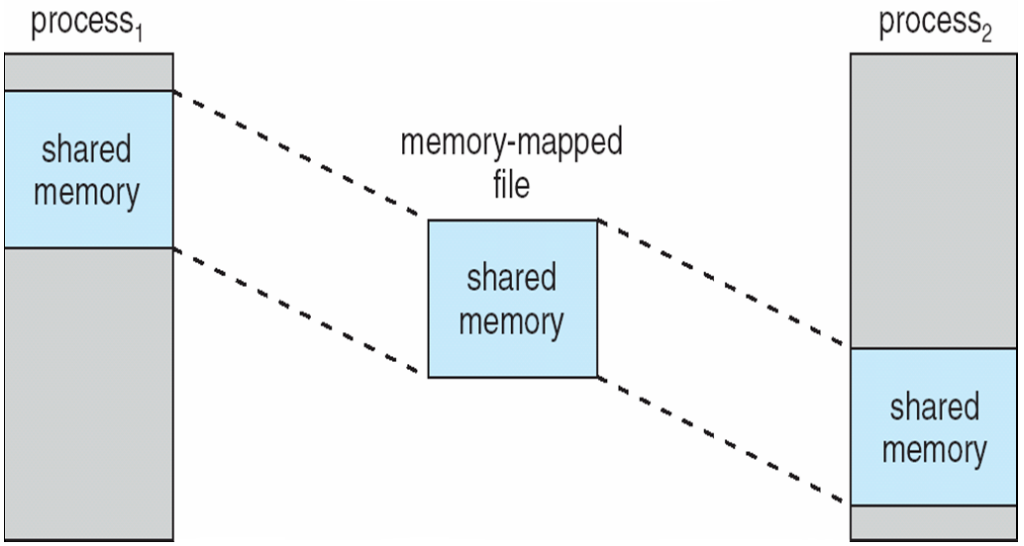
- **Memory-mapped file** I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory

- 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
- This simplifies file access by treating file I/O through memory rather than **`read()`** **`write()`** system calls



Memory-Mapped Shared Memory

- Memory mapped file mechanism also allows several processes to map the same file, allowing the pages in memory to be shared
- For example, on Windows systems, shared memory is accomplished by memory mapping files
  - on these systems, processes can communicate using shared memory by having the communicating processes memory-map the same file into their virtual address spaces, serving as the region of shared memory between the communicating processes



Memory-Mapped I/O

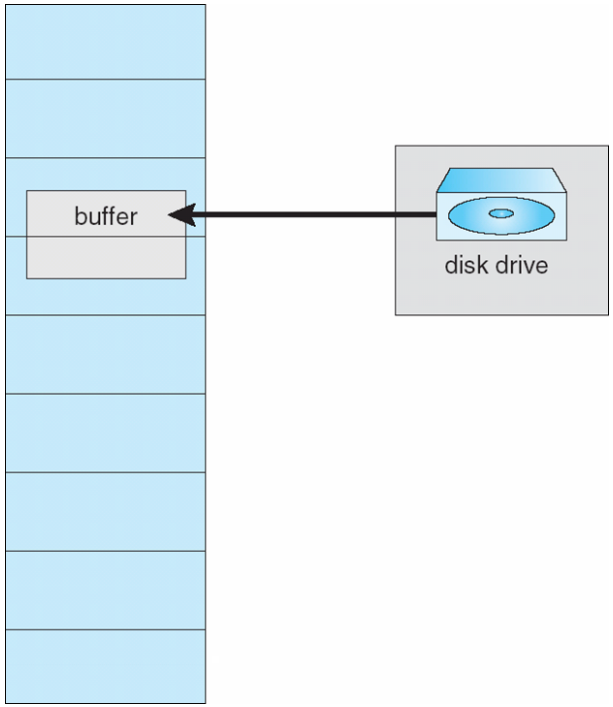
- Each I/O controller includes registers to hold commands and the data being transferred.
  - usually special IO instructions allow data transfer between these registers and system memory
- A more convenient access to I/O devices is **memory-mapped IO**
  - ranges of memory addresses are set aside and are mapped to the device registers
  - reads and writes to these memory addresses cause the data to be transferred to and from the device registers
    - in the IBM PC, each location on the screen is mapped to a memory location
    - in the case of printers and modems, the CPU transfers data through these kinds of devices by reading and writing a few device registers, called an I/O port (serial and parallel ports)

Memory-Mapped I/O (cont)

- cont.
  - The CPU may transfer one byte at a time
  - If the CPU uses polling to watch the control bit, this method is called programmed I/O
    - in this case, the CPU is constantly looping to see whether the device is ready
  - If the CPU does not poll the control bit, but instead receives an interrupt when the device is ready for the next byte, the data transfer is said to be interrupt driven.
  - direct memory access (DMA) is another method for I/O

I/O interlock

- When demand paging is used, we sometimes need to allow some of the pages to be locked in memory
  - **I/O Interlock** – one such situation occurs when I/O is done to or from user memory
- Consider this I/O situation: Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm



Chapter 9: Virtual Memory

- Background

- Demand Paging
- Copy-on-Write
- Page Replacement

- Allocation of Frames
- Thrashing
- Allocating Kernel Memory

- Memory-mapped Files