北京邮电大学软件学院
School Of software Engineering Of BUPT

Operating Systems

Lecture 8 Virtual Memory

Jinpengchen
Email: jpchen@bupt.edu.cn

Catalog Description

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing

Background

- Instructions must be loaded into memory before execution.
- Solutions in last chapter : Program entire→Physical memory
- Sometimes, jobs may be too big or too many.

How to expand the main memory?

- Physically? COST TOO HIGH!
- Logically? √

Background

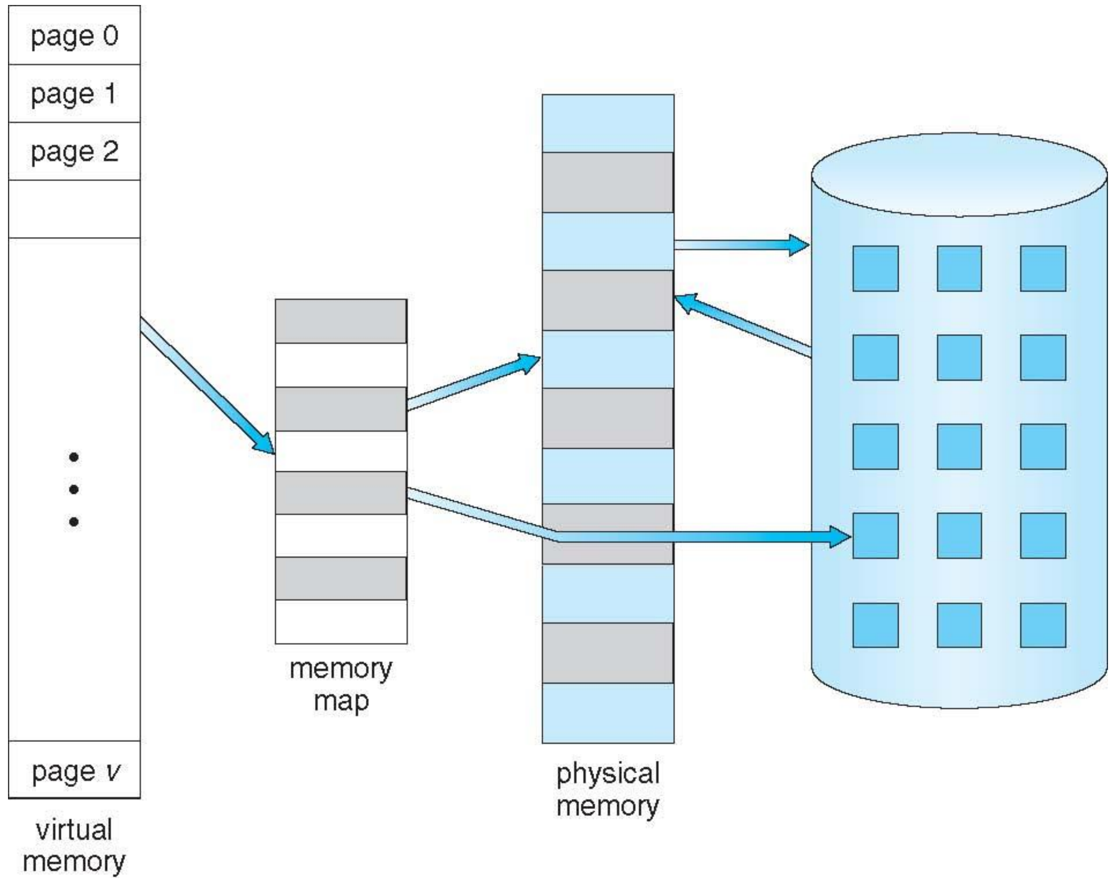
- Virtual memory: Why and How?
  - Some code may get no, or only little, opportunity of execution,
    for example, code for error handlers
  - Some data may get no opportunity of access
  - Locality of reference (程序的局部性原理), 1968, Denning
    - ✓ Temporal locality (时间局部性)
    - ✓ Spatial locality (空间局部性)
  - Idea: partly loading (部分装入)、demand loading (按需装入)、replacement (置换)

Background

- Virtual Memory(虚拟存储器)是指具有请求调页功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储器系统
  - Logical size:
    从系统角度看：内存容量＋外存容量
    从进程角度看：地址总线宽度范围内；内存容量＋外存容量
  - Speed: close to main memory
  - Cost per bit: close to secondary storage (disks)
- 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

Background

- Virtual Memory That is Larger Than Physical Memory



Background

- Virtual memory can be implemented via:
  - Demand paging
    - ✓ Paging technology + pager （请求调页） and page replacement
    - ✓ Pager VS. swapper

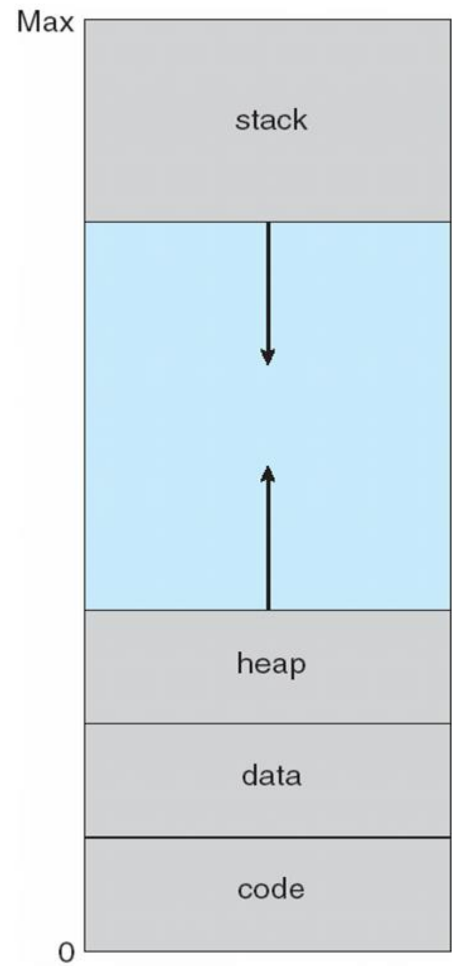
      the unit of swapping in/out is not the entire process but page.
  - Demand segmentation

Background

- 虚拟存储器的特征
  - 多次性：最重要的特征
    - ✓ 一个作业被分成多次装入内存运行
  - 对换性
    - ✓ 允许在进程运行的过程中，（部分）换入换出
  - 虚拟性
    - ✓ 逻辑上的扩充
  - 虚拟性是以多次性和对换性为基础的。
  - 多次性和对换性是建立在离散分配的基础上的

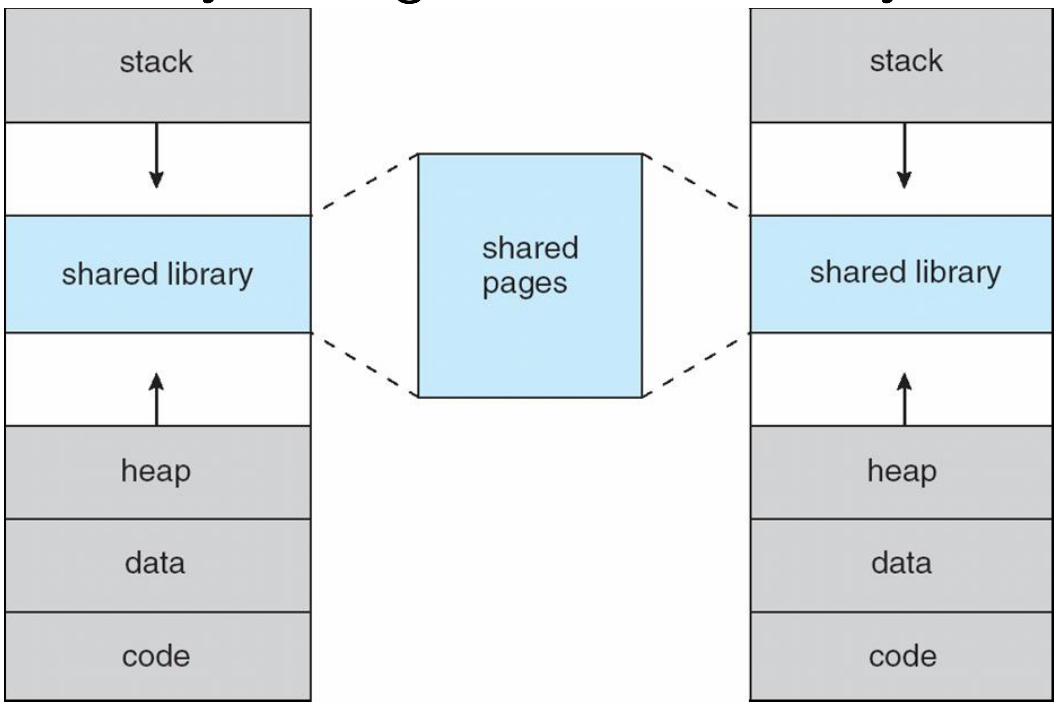
Background

- Virtual-address Space (虚拟地址空间)
  - The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory.
    - ✓ Typically: 0~xxx & exists in contiguous memory
  - In fact, the physical memory are organized (partitioned) in page frames & the page frames assigned to a process may not be contiguous ⇒ MMU



Background

- Some benefits
  - Shared library using virtual memory



  - Shared memory
  - Speeding up process creation

Background

- Some benefits
  - Programmers need not care about memory limitations
  - More programs could be run simultaneously
  - Less I/O needed, each user program would run faster

Catalog Description

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing

Demand Paging

- Do not load the entire program in physical memory at program execution time.

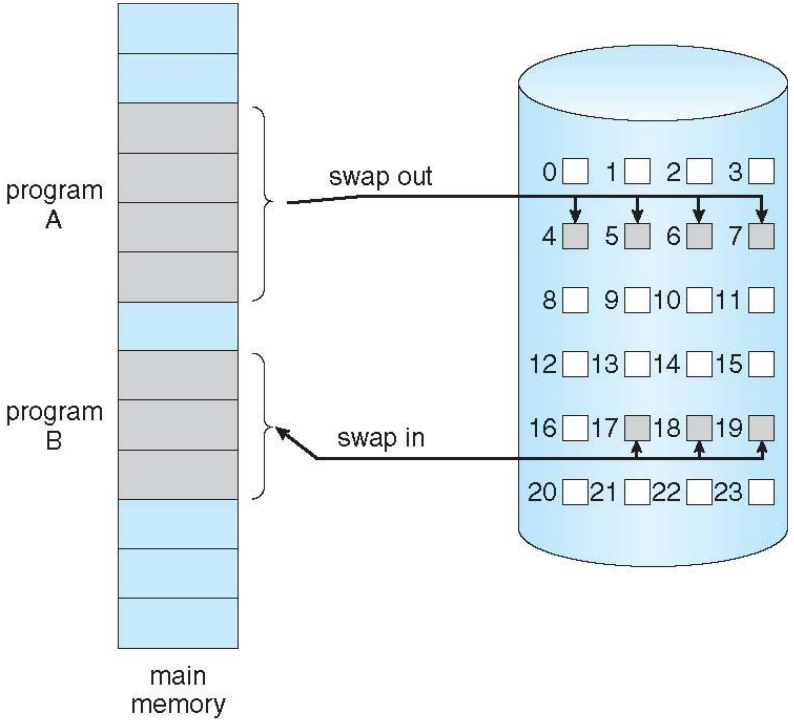
  NO NEED!
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- A page is needed ⇐ Reference to it
  - Invalid reference ⇒ Abort
  - Not-in-memory ⇒Bring to memory

Demand Paging

- Swapper VS. Pager
  - A swapper manipulates the entire processes
  - Lazy swapper

    Never swaps a page into memory unless the page will be needed
    - ✓ Swapper that deals with individual pages is a pager

Example： Transfer of a Paged Memory to Contiguous Disk Space

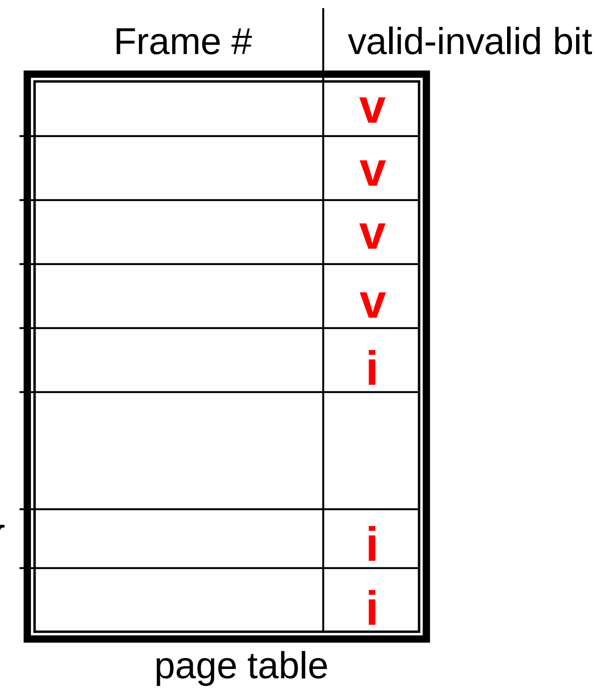


Demand Paging

- Basic Concepts (Hardware support)
  - The modified page table mechanism
  - Page fault
  - Address translation
  - Secondary memory (as swap space)

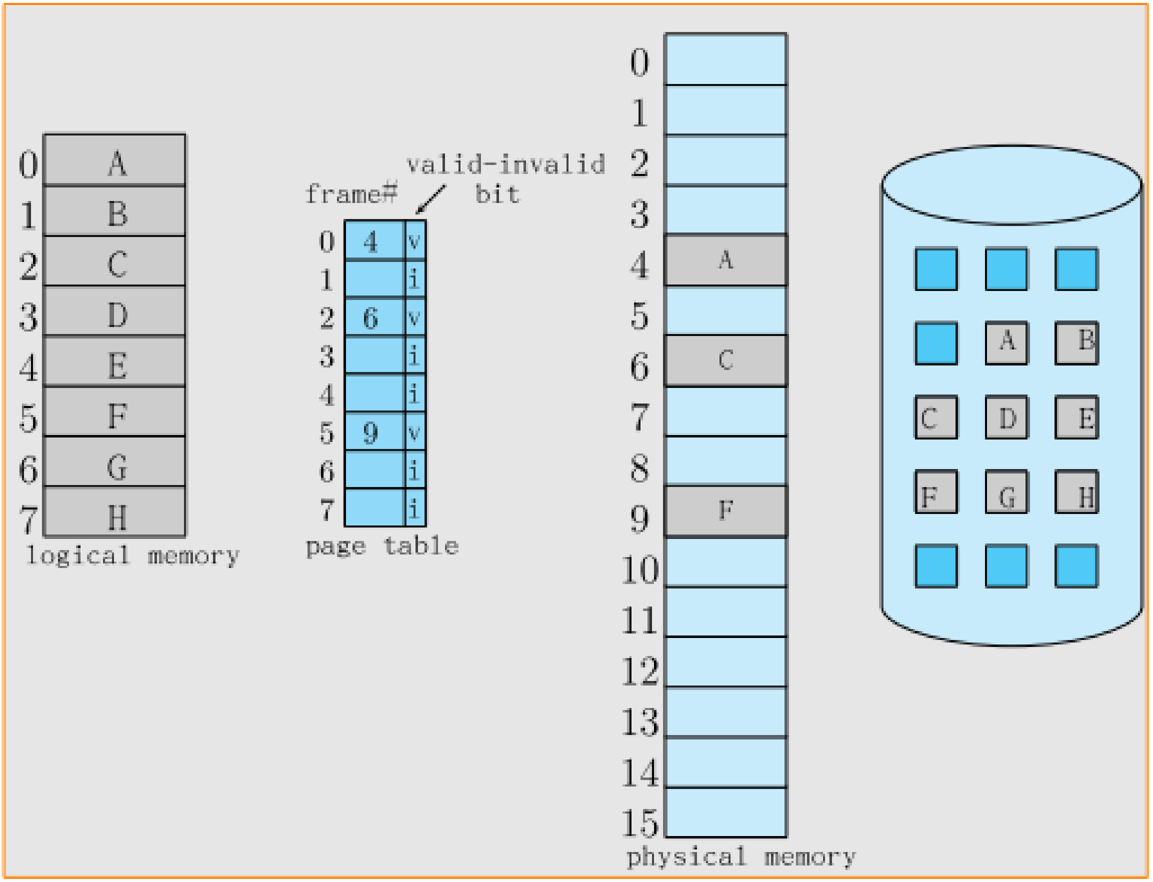
Demand Paging

- The modified page table mechanism
  - Valid-Invalid Bit (PRESENT bit)
    - ✓ Each page table entry is associated with a valid-invalid bit
      - v ⇒ in-memory &legal,
      - i ⇒ not-in-memory
    - ✓ Initially, the valid-invalid bit is set to i on all entries
    - ✓ During address translation, if the valid-invalid bit in page table entry is i ⇒ page fault
  - Reference bits (for pager out)
  - Modify bit (or dirty bit)
  - Secondary storage info (for pager in)



Demand Paging

- The modified page table mechanism
  - Page table when some pages are not in main memory

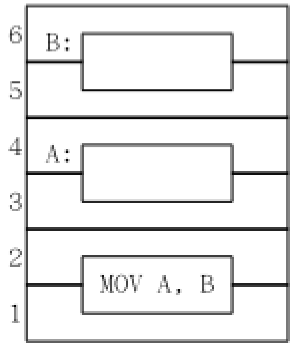


Demand Paging

- Page Fault (缺页故障)
  - First reference to a page will trap to OS:
    - ✓ page fault(缺页故障/异常/中断)
  - Page fault trap (缺页异常)
    - ✓ Exact exception (trap), 精确异常

      Restart the process in the exactly same place and state.

      Re-execute the instruction which triggered the trap
  - Execution of one instruction may cause multiply page faults



- Page fault may occur at every memory reference
- One instruction may cause multiply page faults while fetching instruction or r/w operators

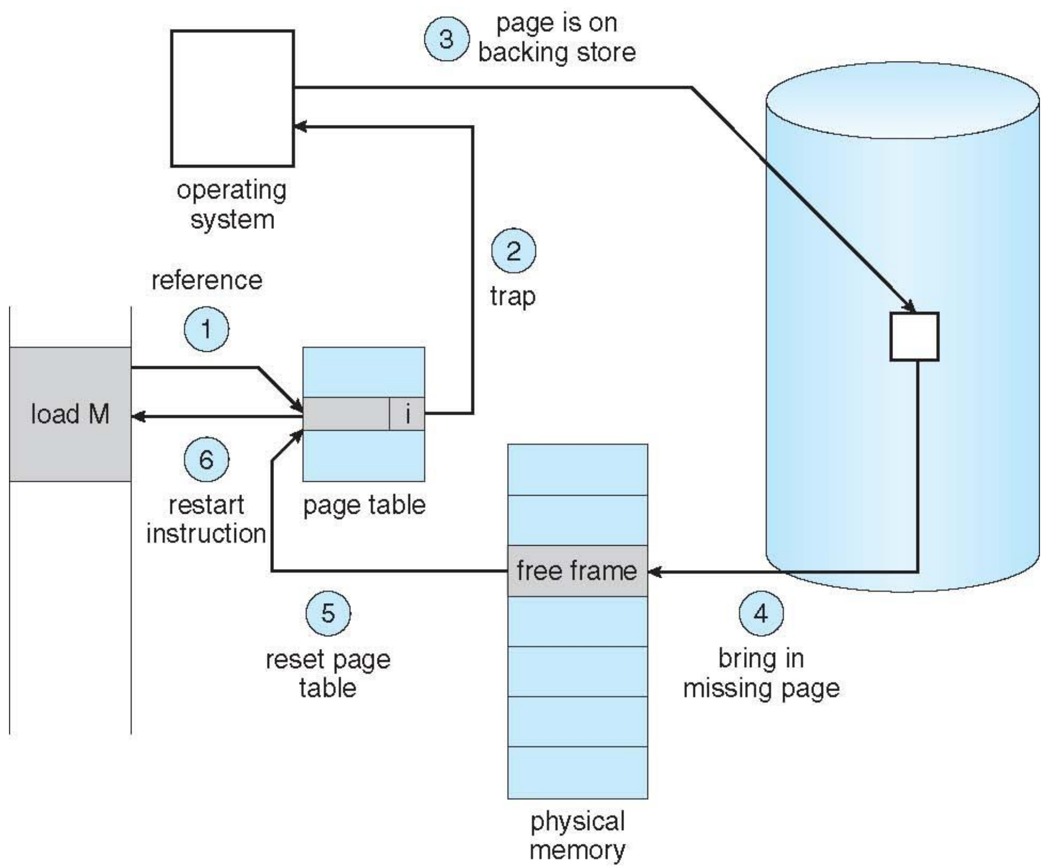
Example: One instruction and 6 page faults

Demand Paging

- Page Fault (缺页故障)
  - Page Fault Handling:
    - ✓ OS looks at an internal table to decide:
      - – Invalid reference ⇒ abort
      - – Just not in memory ⇒ bring to memory
    - ✓ Get an empty frame from the free frame list
    - ✓ Swap page into frame
      - – Pager out & pager in
    - ✓ Modify the internal tables & Set validation bit = v
    - ✓ Restart the instruction that caused the page fault

Demand Paging

- Page Fault (缺页故障)
  - Steps in Handling a Page Fault



Demand Paging

- Page Fault (缺页故障)
  - Restart instruction
    C = A + B
    - ✓ 1. fetch the instruction
    - ✓ 2. fetch A
    - ✓ 3. fetch B
    - ✓ 4. Add A and B
    - ✓ 5. Store the sum in C (C not in memory)
  - Problem can be ignored
    - ✓ Repetition
    - ✓ 1 instruction causes N page faults

Demand Paging

- Address translation
  - ✓ Address translation hardware + page fault handling
- Resume the execution
  - Context save (保存现场)

    Before OS handling the page fault, the state of the process must be saved
    - ✓ Example: record its register values, PC
  - Context restore (恢复现场)

    The saved state allows the process to be resumed from the line where it was interrupted.
  - NOTE: distinguish the following two situations
    - ✓ Illegal reference ⇒ The process is terminated
    - ✓ Page fault ⇒ Load in or pager in

Demand Paging

- Performance of Demand Paging
  - Let p = Page Fault Rate ($0 \leqslant p \leqslant 1.0$)
    - ✓ If p = 0, no page faults
    - ✓ If p = 1.0, every reference is a fault, Pure Demand Page
  - Effective Access Time (EAT)

    EAT = (1 – p) × memory access
    + p × page fault time

    page fault time = page fault overhead
    + swap page out (optional)
    + swap page in
    + restart overhead

Demand Paging

Performance of Demand Paging

- Example
  - ✓ Memory access time = 200ns
  - ✓ Average page-fault service time = 8ms

    EAT = (1−p)×200 + p×8ms= (1 − p)×200 + p×8,000,000

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

    p = 0.001

    EAT = 8, 199.8ns = 8.2μs

    This is a slowdown by a factor of 8.2us/200ns = 40!!
  - ✓ If we want performance degradation < 10%, then

    EAT = 200 + p × 7, 999, 800 < 200 (1 + 10%) = 220

    p × 7, 999, 800 < 20

    p < 20/7,999,800≈0.0000025

Demand Paging

- Performance of Demand Paging
  - Method for better performance
    - ✓ To keep the fault time low
      - – Swap space, faster than file system
      - – Only dirty page is swapped out, or
      - – Demand paging only from the swap space, or
      - – Initially demand paging from the file system, swap out to swap space, and all subsequent paging from swap space
    - ✓ Keep the fault rate extremely low
      - – Localization of program executing
        - Time, space

Catalog Description

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing

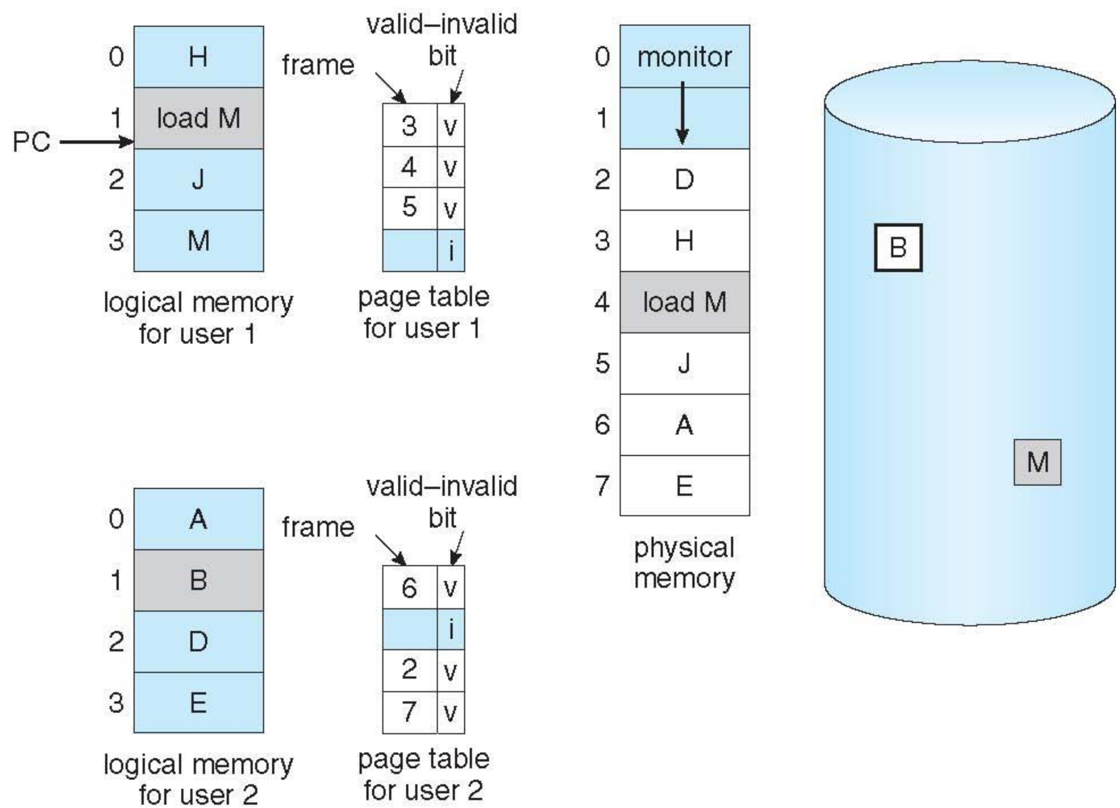
Page Replacement

- What happens if there is no free frame?
  - 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

- Need of Page Replacement (页面置换)
  - Free page frame is managed by OS using free-frame-list
  - Over-allocation: No free frames; All memory is in use.



Page Replacement

- What happens if there is no free frame?
  - Solution:

    Page replacement (页面置换)

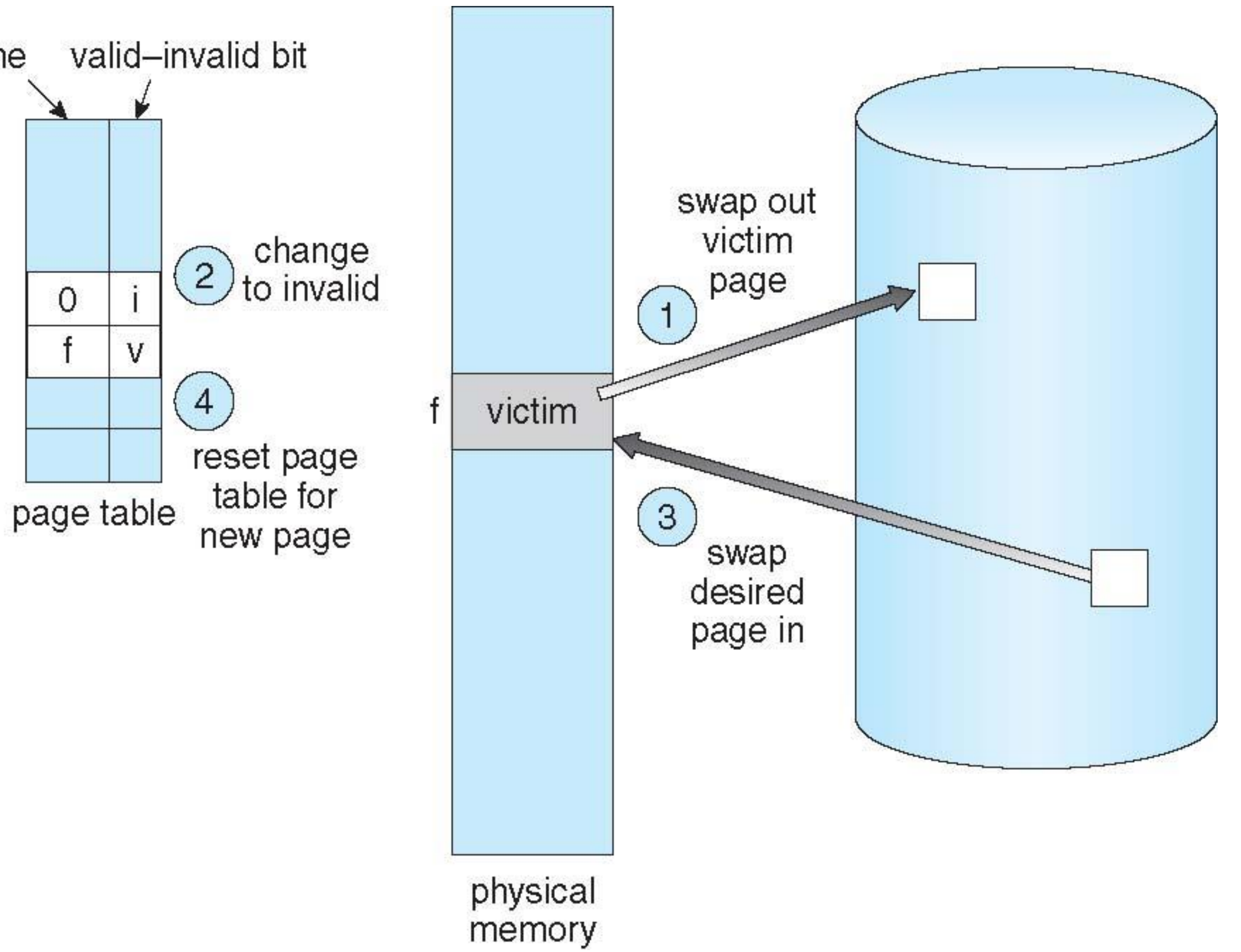
    Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

Page Replacement

- 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 and frame tables
  - Restart the process

Page Replacement

- Basic Page Replacement



Page Replacement

- Basic Page Replacement
  - NO MODIFY, NO WRITTEN (to disk/swap space)
    - ✓ Use modify (dirty) bit to reduce overhead of page transfers
      - – Only modified pages are written to disk
    - ✓ This technique also applies to read-only pages
      - – For example, pages of binary code
  - Page replacement completes separation between logical memory and physical memory
    - ✓ Large virtual memory can be provided on a smaller physical memory
  - Demand paging, to obtain the lowest page-fault rate, two major problems
    - ✓ Frame-allocation algorithms
    - ✓ Page-replacement algorithms

Page Replacement

- Page Replacement Algorithms
  - GOAL: to obtain the lowest page-fault rate
  - Different algorithms are evaluated by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - A reference string is a sequence of addresses referenced by a program

    Example:

    ✓ An address reference string:

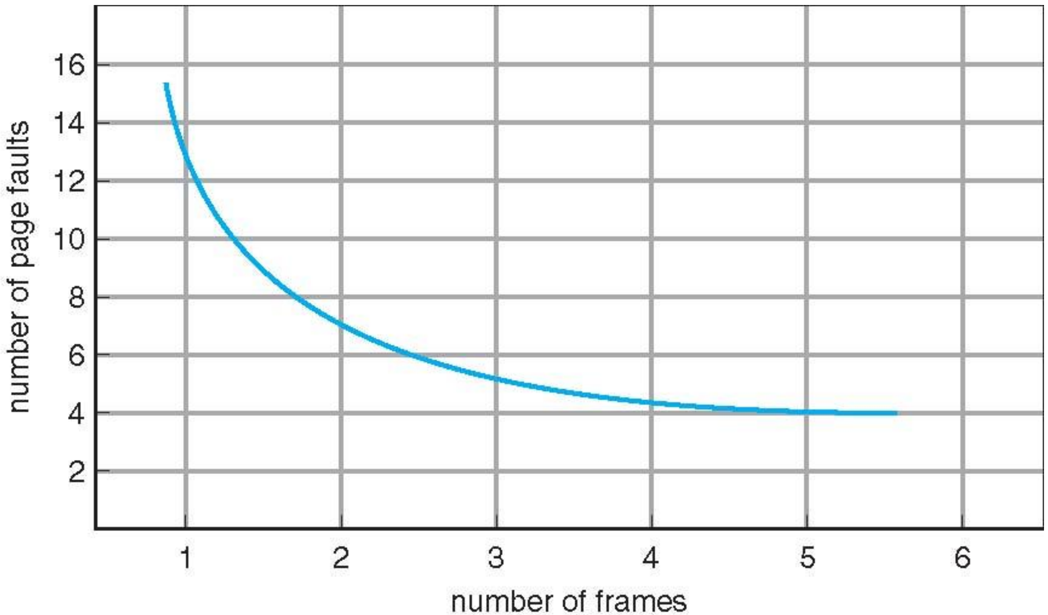
    0100 0432 0101 0612 0102 0103 0104 0101 0611 0103 0104 0101 0610 0102 0103 0104 0101 0609 0102 0105

    ✓ Assuming page size = 100B, then its corresponding page reference string is:

    1 4 1 6 1 6 1 6 1 6 1

Page Replacement

- Page Replacement Algorithms
  - How many page faults?
    - ✓ Determined by the number of page frames assigned to the process
    - ✓ For the upper example: 1 4 1 6 1 6 1 6 1 6 1
      - If ⩾ 3, then only 3 page faults
      - If = 1, 11 pages faults



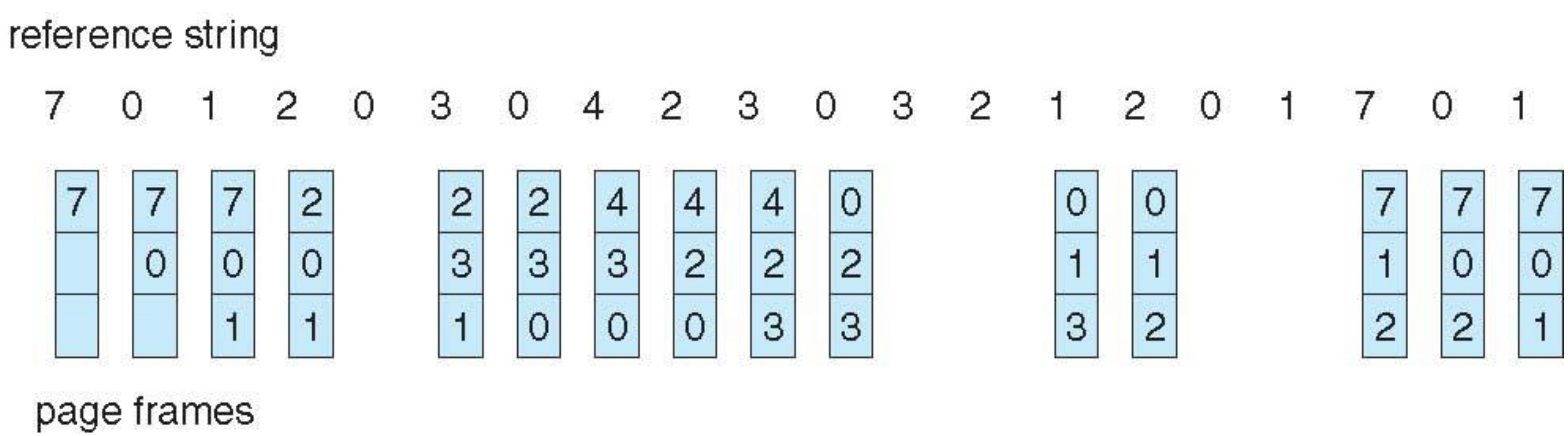
Graph of Page Faults Versus The Number of Frames

Page Replacement

- Page Replacement Algorithms
  - In all our examples, the reference strings are
    - ✓ 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
    - ✓ 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

First-In-First-Out (FIFO) Algorithm

- The simplest page-replacement algorithm: FIFO
  - For each page: a time when it was brought into memory
  - For replacement: the oldest page is chosen
  - Data structure: a FIFO queue
    - ✓ Replace the page at the head of the queue
    - ✓ Insert a new page at the end of the queue
  - Example 1: 15 page faults, 12 page replacements



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)

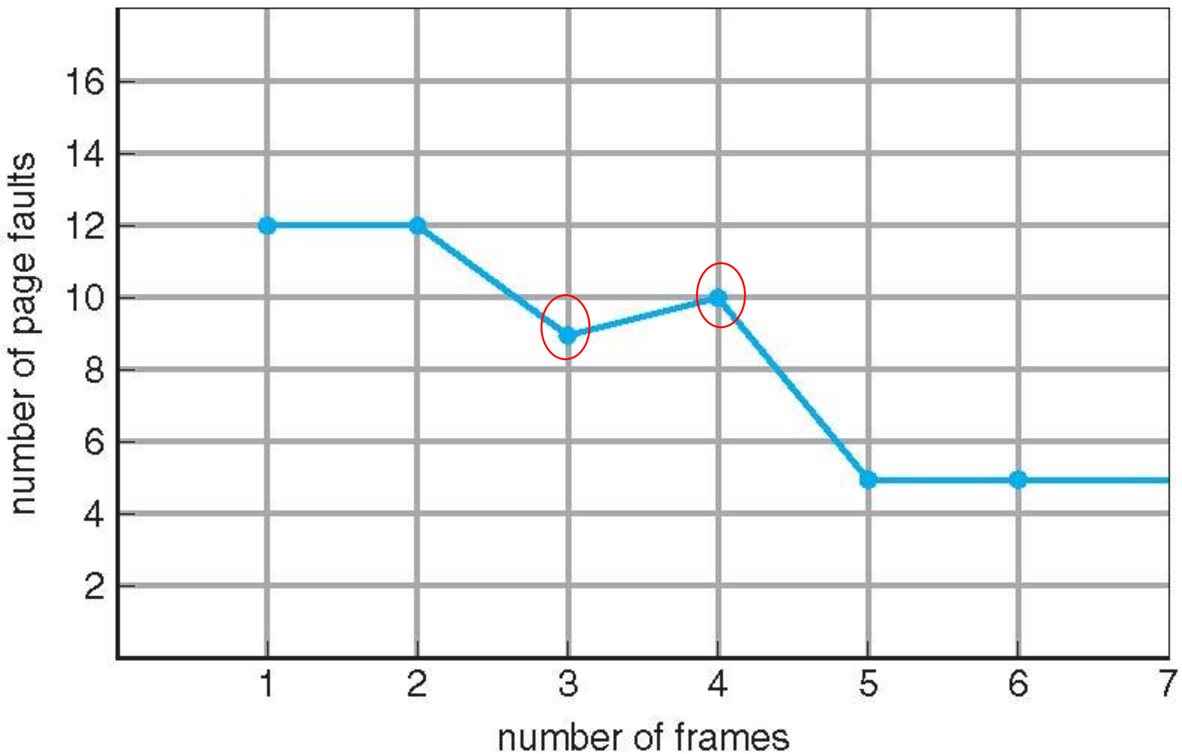
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

  - 4 frames

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

First-In-First-Out (FIFO) Algorithm

- More memory, better performance? MAY BE NOT!!
  - Belady's anomaly (贝莱迪异常现象):
    more frames ⇒ more page faults

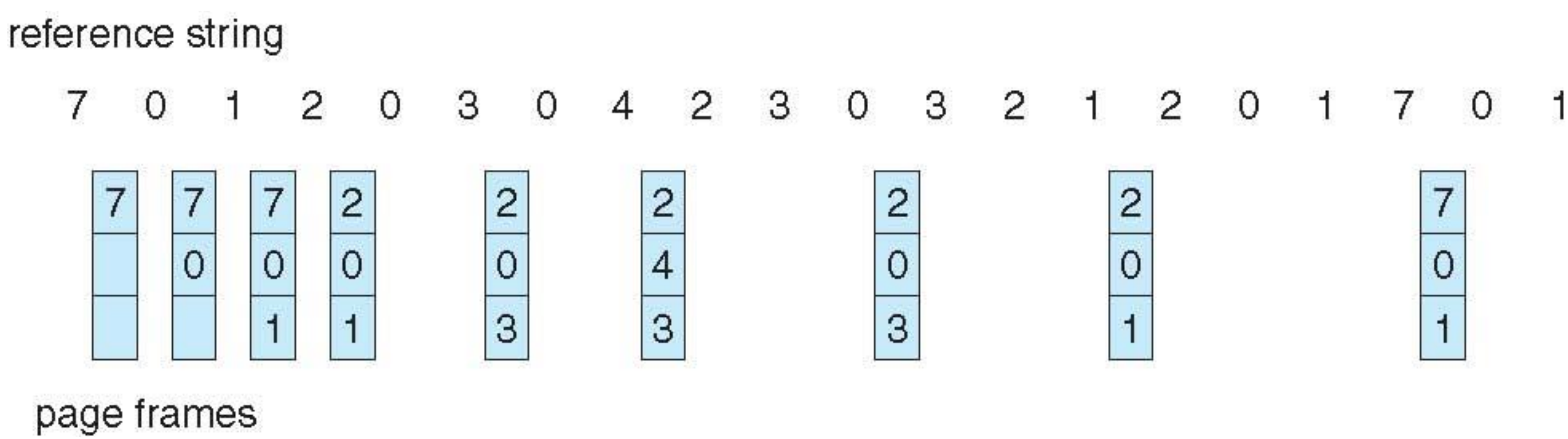


FIFO illustrating Belady's Anomaly

Optimal Algorithm

- Optimal page-replacement algorithm:

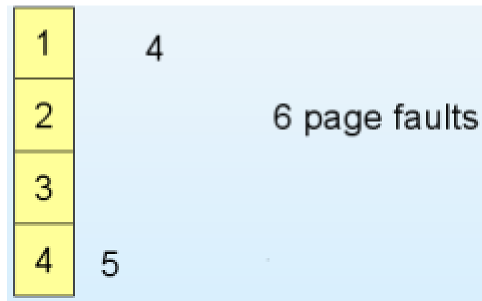
  Replace page that will not be used for longest period of time
  - It has the lowest page-fault rate
  - It will never suffer from Belady's anomaly
- Example 1: 9 page faults, 6 page replacements



Optimal Algorithm

- 4 frames example

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



- OPT: Difficult to implement
  - How to know the future knowledge of the reference string?
- So, it is only used for measuring how well other algorithm performs

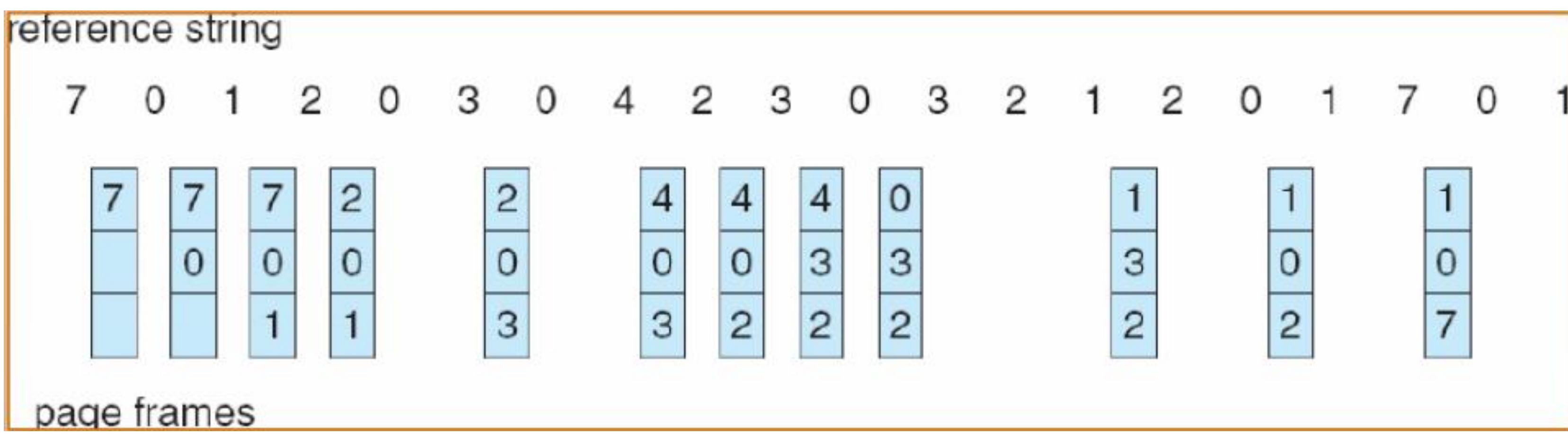
Least Recently Used (LRU) Algorithm

- LRU: an approximation of the OPT algorithm

Use the recent past as an approximation of the near future

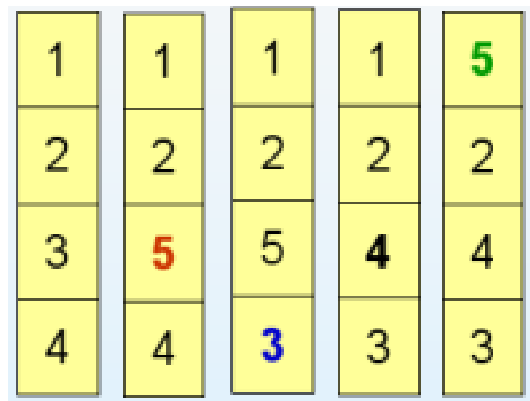
  - To replace the page that has not been used for the longest period of time
  - For each page: a time of its last use
  - For replace: the oldest time value

- Example 1: 12 page faults, 9 page replacements



Least Recently Used (LRU) Algorithm

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



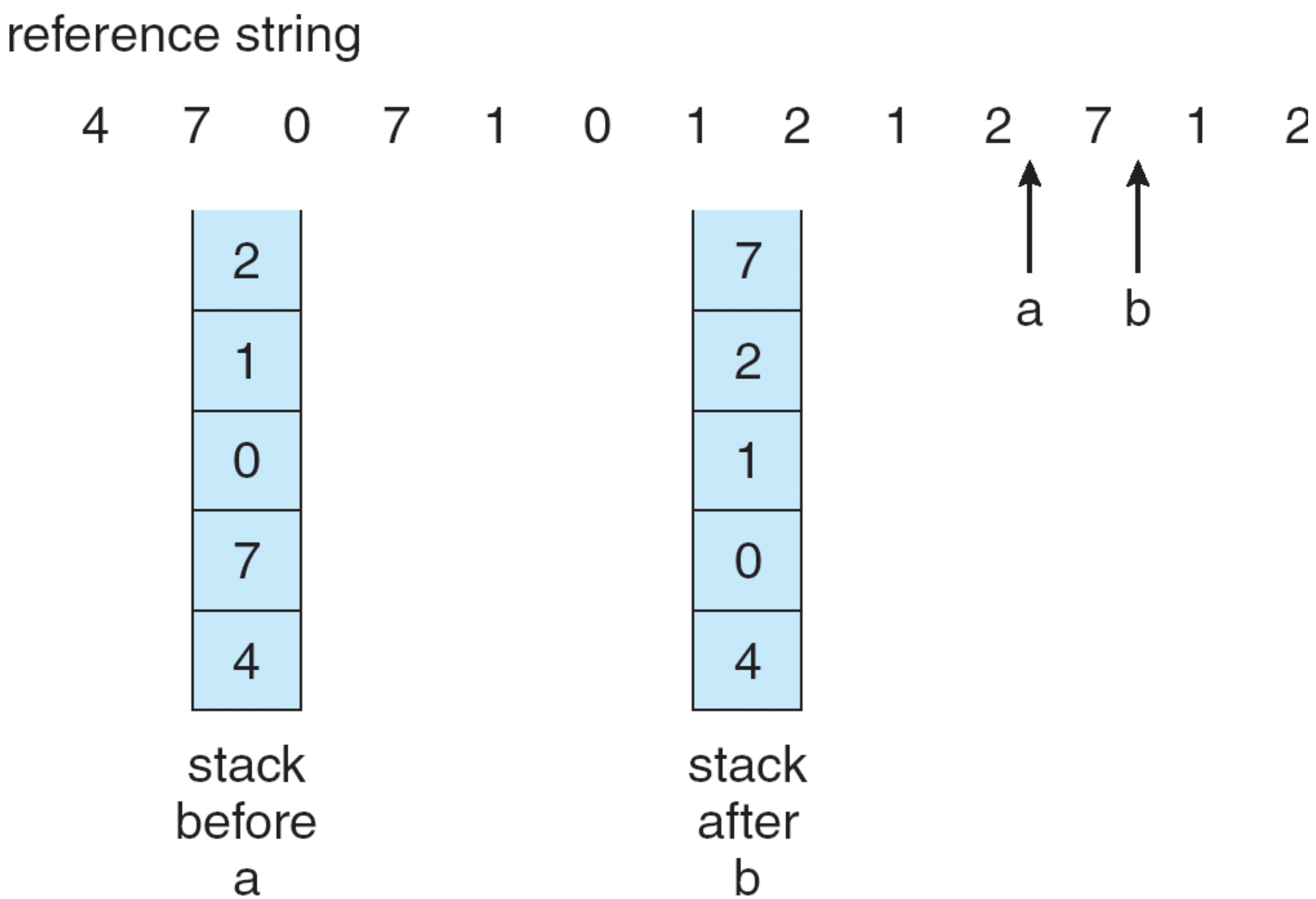
8 page faults

Least Recently Used (LRU) Algorithm

- HOW to implement LRU replacement?
  - Counter implementation
    - ✓ Every page entry has a counter;
      every time page is referenced through this entry, copy the clock into the counter
    - ✓ When a page needs to be changed, look at the counters to determine which are to be changed
  - Stack implementation – keep a stack of page numbers in a double link form:
    - ✓ When a page is referenced: Move it to the top
      - Requires 6 pointers to be changed
    - ✓ Each update is a bit more expensive
    - ✓ No search for replacement

Least Recently Used (LRU) Algorithm

- Use of a Stack to Record the Most Recent Page References



LRU Approximation Algorithms

- Reference bit
  - Each page is associated with a bit, initially = 0
  - When page is referenced, this bit is set to 1
  - Replace the one which is 0 (if one exists)
    - ✓ However, we do not know the order
- Additional-Reference-Bits Algorithm:
  - Record each entry of a page table with a 8-bit byte
  - At a regular interval, a timer interrupt transfers the control to the OS

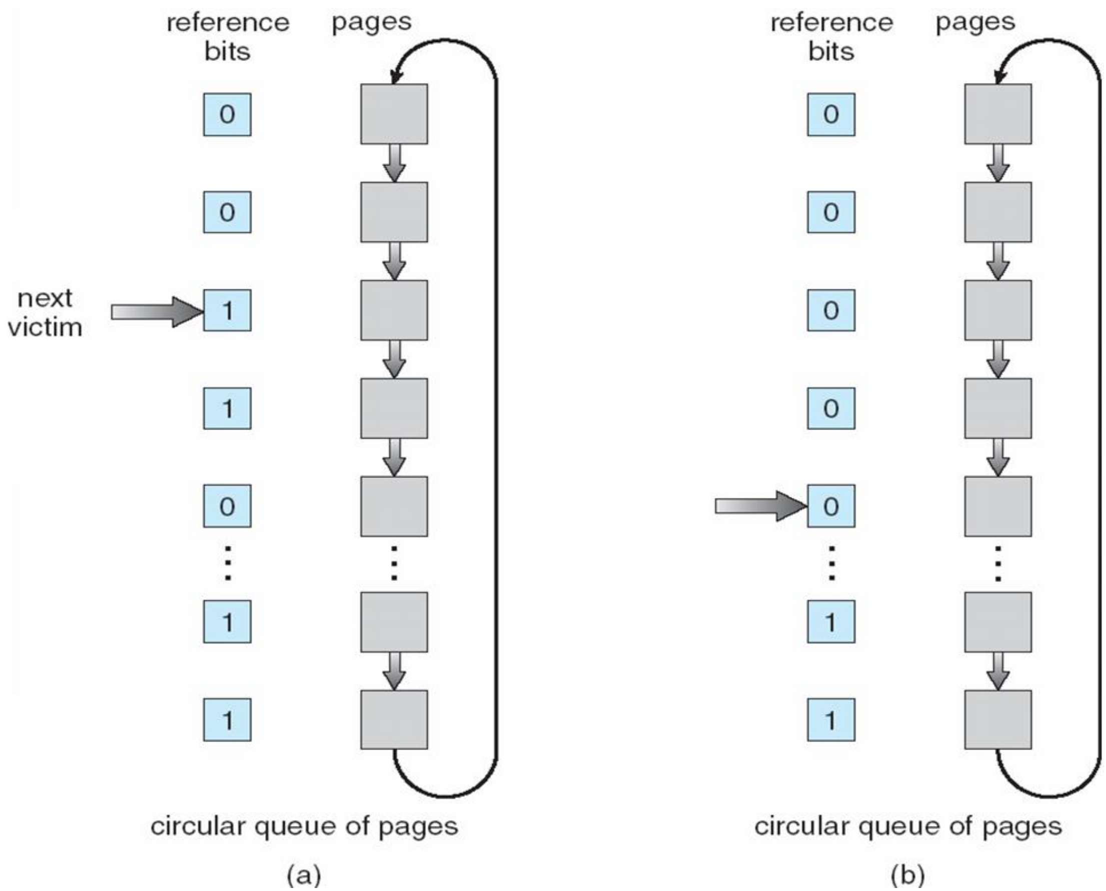
  Reference bits + time ordering, for example: 8 bits
    - ✓ HW modifies the highest bit, only
    - ✓ Periodically, right shift the 8 bits for each page
    - ✓ 00000000, ..., 01110111, ..., 11000100, ..., 11111111

LRU Approximation Algorithms

- Second chance (clock) Algorithm
  - Need only 1 reference bit, modified FIFO algorithm
  - Clock replacement
    - ✓ First, a page is selected by FIFO
    - ✓ If the page to be replaced (in clock order) has reference bit = 0, then replace
    - ✓ If the 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

LRU Approximation Algorithms

- Second chance (clock) Algorithm
  - Implementation: Clock replacement
    - ✓ Clock order



Counting Algorithms

- Counting algorithms:

  Keep a counter of the number of references that have been made to each page

  - LFU(Least Frequently Used) Algorithm: replaces page with the smallest count
    - ✓ Problem: heavily used initially, then never used
    - ✓ Solution: shift the counts right by 1 bit to decay the average usage count
  - MFU(Most Frequently Used) Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Catalog Description

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing

Allocation of Frames

- Minimum the number of pages
  - Each process needs the minimum number of pages
  - Determined by ISA (Instruction-Set Architecture)
    - ✓ We must have enough frames to hold all the different pages that any single instruction can reference
  - Example: IBM 370

    6 pages to handle MOVC instruction:
    - ✓ Instruction is 6 bytes, might span 2 pages
    - ✓ 2 pages to handle from
    - ✓ 2 pages to handle to
  - Two major allocation schemes
    - ✓ Fixed allocation; priority allocation
  - Two replacement policy
    - ✓ Global vs. local

Allocation of Frames

Fixed allocation

- Equal allocation
  - For example, if there are 100 frames and 5 processes, give each process 20 frames.

frame number for any process = m/n

m = total memory frames

n = number of processes

Allocation of Frames

Fixed allocation

- Proportional allocation
  - Allocate according to the size of process
    - ✓ example:

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

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

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

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

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

Allocation of Frames

- Priority Allocation
  - Use a proportional allocation scheme using priorities rather than size
  - If process Pi generates a page fault,
    - ✓ Select for replacement one of its frames
    - ✓ Select for replacement a frame from a process with lower priority number

Allocation of Frames

- Replacement policy: 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

    - ✓ Problem: a process cannot control its own page-fault rate
  - Local replacement
    - ✓ each process selects a frame from only its own set of allocated frames
    - ✓ Problem: less used pages of memory

Catalog Description

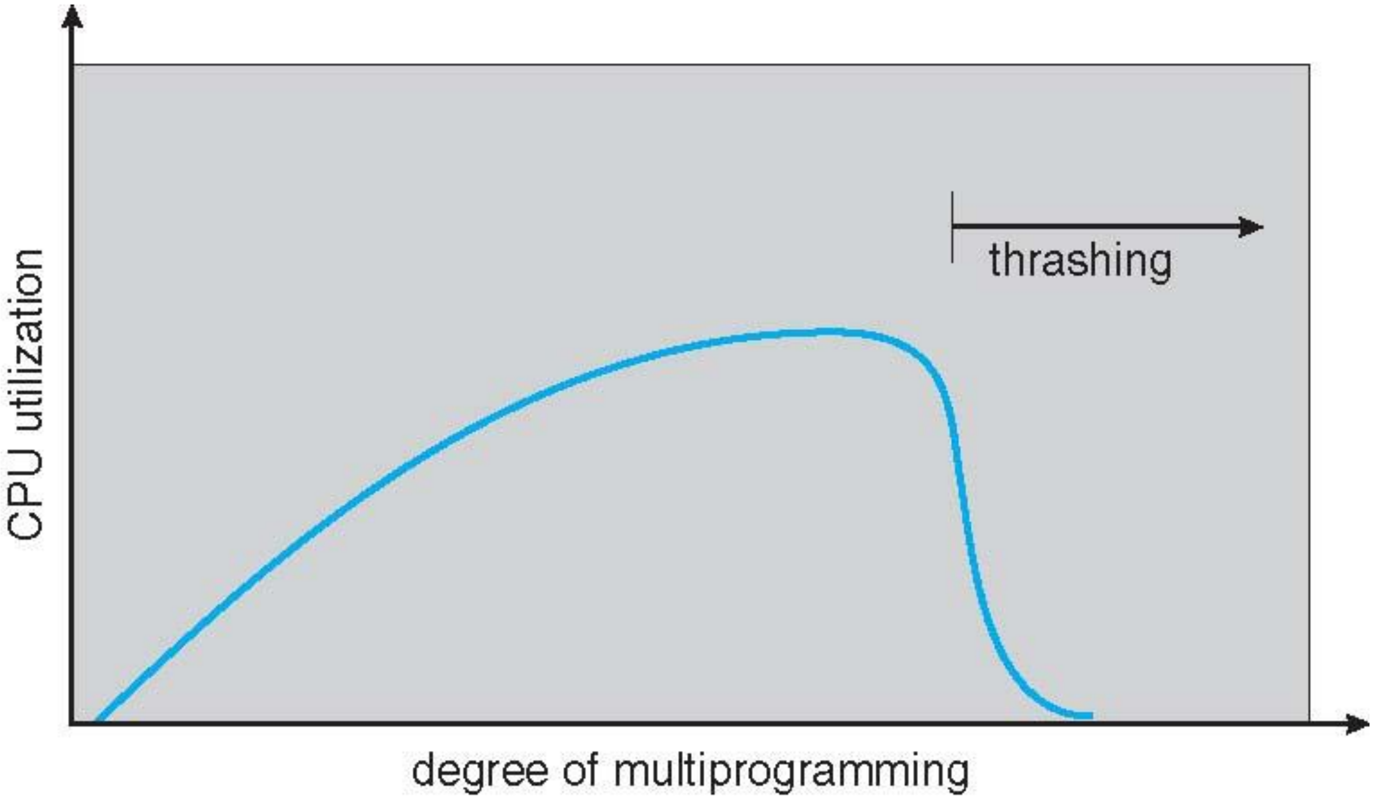
- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing

Thrashing (抖动)

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - Low CPU utilization
  - OS thinks that it needs to increase the degree of multiprogramming
  - Another process is added to the system, getting worse!
- Thrashing ≡ a process is busy swapping pages in and out
  - Spending more time in swapping than executing

Thrashing（抖动）

- Cause of trashing: unreasonable degree of multiprogramming (不合理的多道程序度)



Thrashing (抖动)

- How to limit the effects of thrashing
  - Local replacement algorithm? not entirely solved.
  - We must provide a process with as many frames as it needs--locality
  - How do we know how many frames are needed?
    - ✓ working-set strategy ⇐ Locality model
- Locality model: This is the reason why demand paging works
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?

  Σ size of locality > total memory size

Thrashing (抖动)

- Working-Set Model (工作集模型)
  - The working-set model is based on the assumption of locality.
  - Let

    $\Delta \equiv$ working-set window

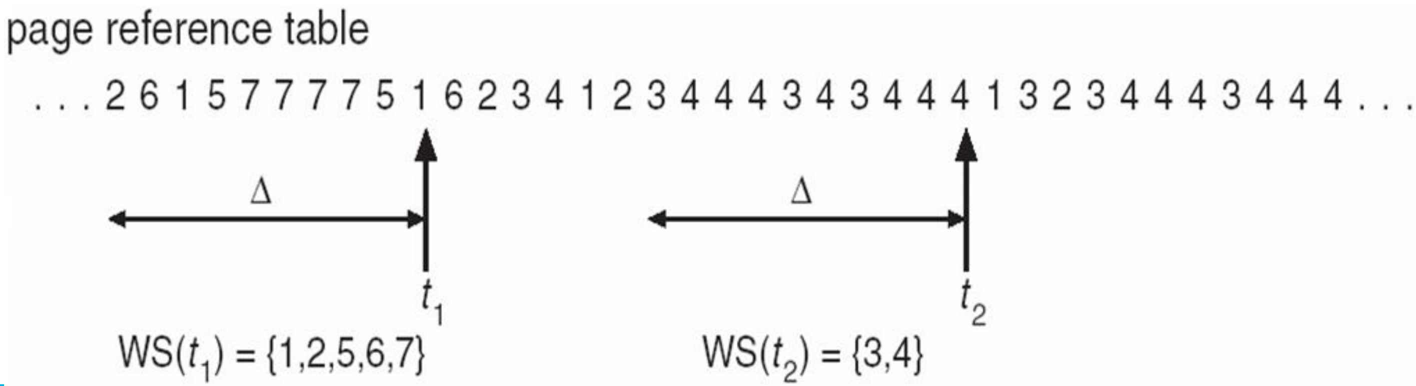
    $\equiv$ a fixed number of page references

    For example: 10,000 instructions
- Working set (工作集):

  The set of pages in the most recent Δ page references.
  - An approximation of the program's locality.

Thrashing (抖动)

- Working-Set Model (工作集模型)
  - Example: Δ = 10



  - Working set size:

WSS_i (working set of Process P_i) =
total number of pages referenced in the most recent Δ (varies in time)

- ✓ Varies in time, depend on the selection of Δ
  - if Δ too small ⇒ will not encompass entire locality
  - if Δ too large ⇒ will encompass several localities
  - if Δ = ∞ ⇒ will encompass entire program

Thrashing（抖动)

- For all processes in the system, currently

$$D = \Sigma\, WSS_i \equiv \text{total demand frames}$$

m：total number of available frames

  - D > m ⇒ Thrashing
  - Policy:

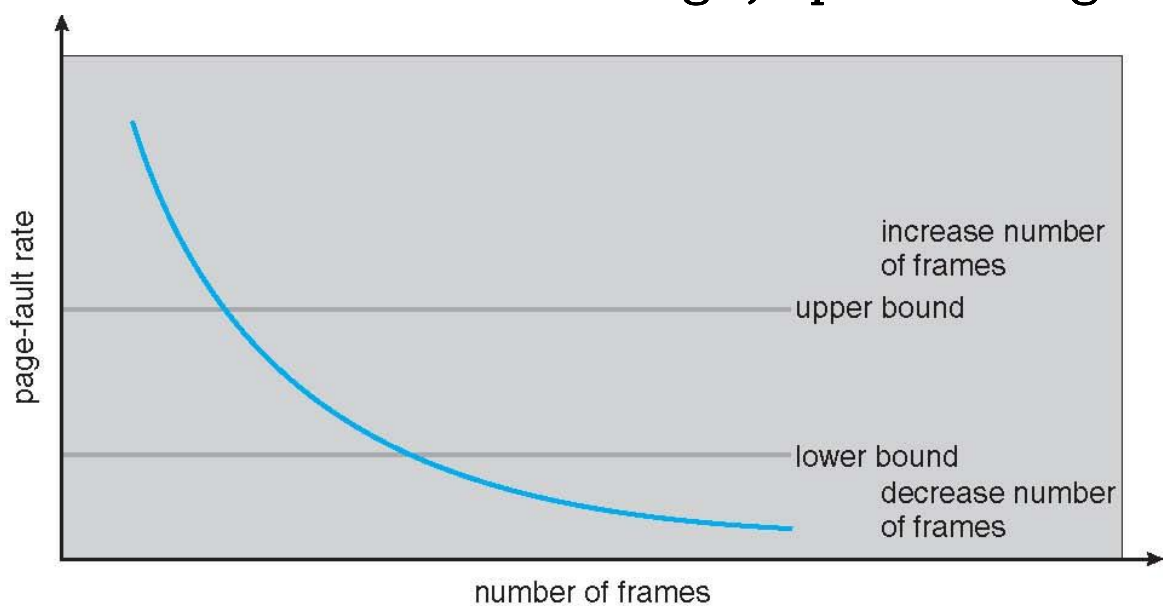
    if D > m, then suspend one of the processes

Thrashing (抖动)

- Keeping Track of the Working Set
  - Approximate with: fixed interval timer + reference bits
  - Example: $\Delta = 10{,}000$
    - ✓ Cause a timer interrupt after every 5000 time units
    - ✓ Keep two in-memory bits for each page
    - ✓ Whenever a timer interrupts, copy and set 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?
    - ✓ IN!! But where?
  - Improvement:
    - ✓ 10 bits and interrupt every 1000 time units

Thrashing (抖动)

- Page-Fault Frequency (缺页频率): helpful for controlling trashing
  - Trashing has a high page-fault rate.
  - Establish “acceptable” page-fault rate
    - ✓ If actual rate is too low, process loses frame
    - ✓ If actual rate is too high, process gains frame



End of Chapter 8