Virtual Memory

Operating Systems

Yu-Ting Wu

(with slides borrowed from Prof. Jerry Chou)

Outline

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

Background

Background

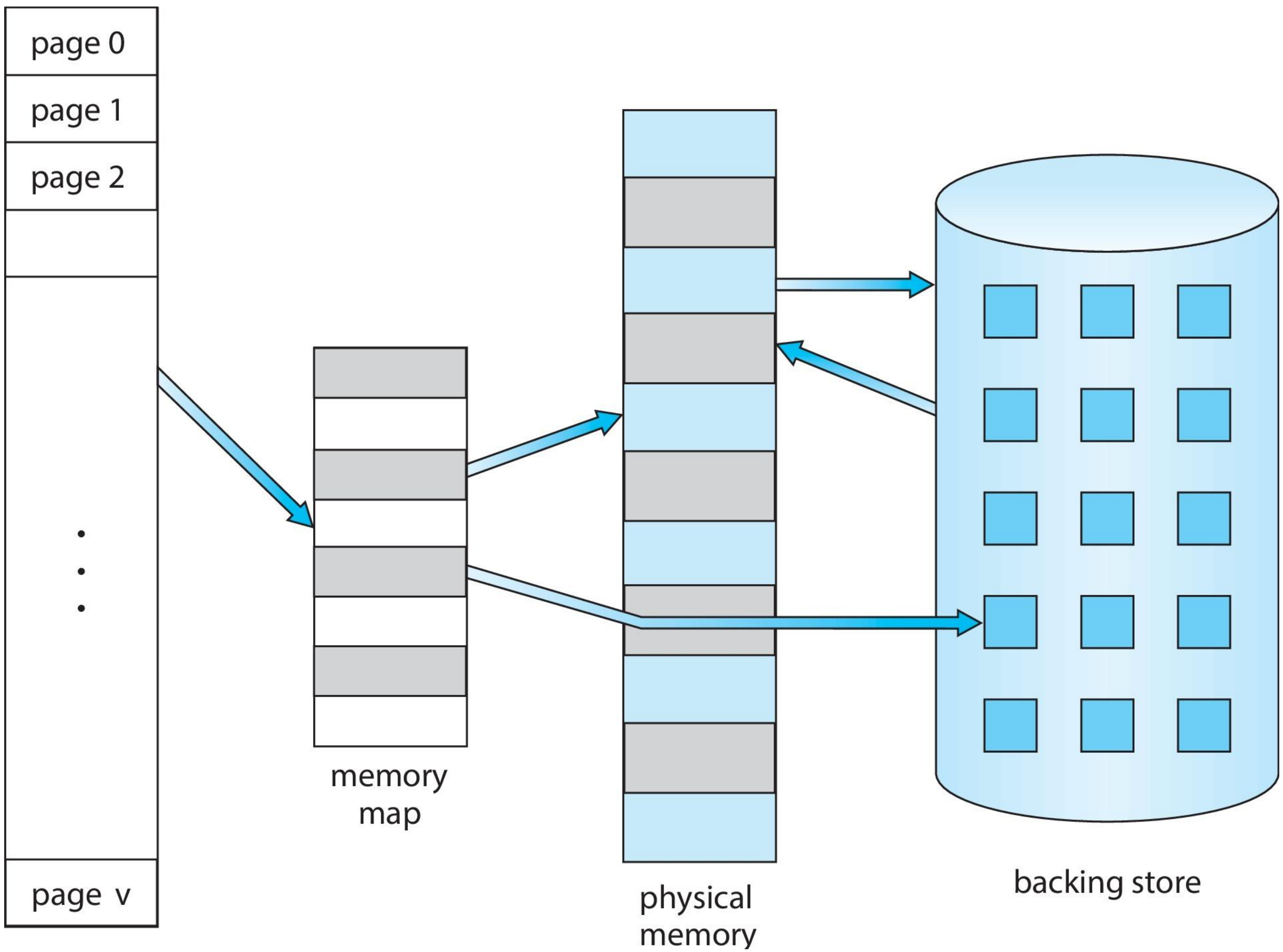
- Why we don’t want to run a program that is entirely in memory
  - Many code for handling unusual errors or conditions
  - Certain program routines or features are rarely used
  - The same library code used by many programs
  - Arrays, lists, and tables allocated but not used

➔ **We want better utilization**

Virtual Memory

- Separation of user logical memory from physical memory
  - To run an **extremely large process**
    - Logical address space can be much larger than physical address space
  - To increase **CPU/resource utilization**
    - Higher degree of multiprogramming degree
  - To simplify programming (compiler) tasks
    - Free programmer from memory limitation
  - To **launch** programs **faster**
    - Less I/O would be needed to load or swap
- Can be implemented via
  - **Demand paging**
  - Demand segmentation (more complicated due to variable sizes)

Virtual Memory v.s. Physical Memory



Demand Paging

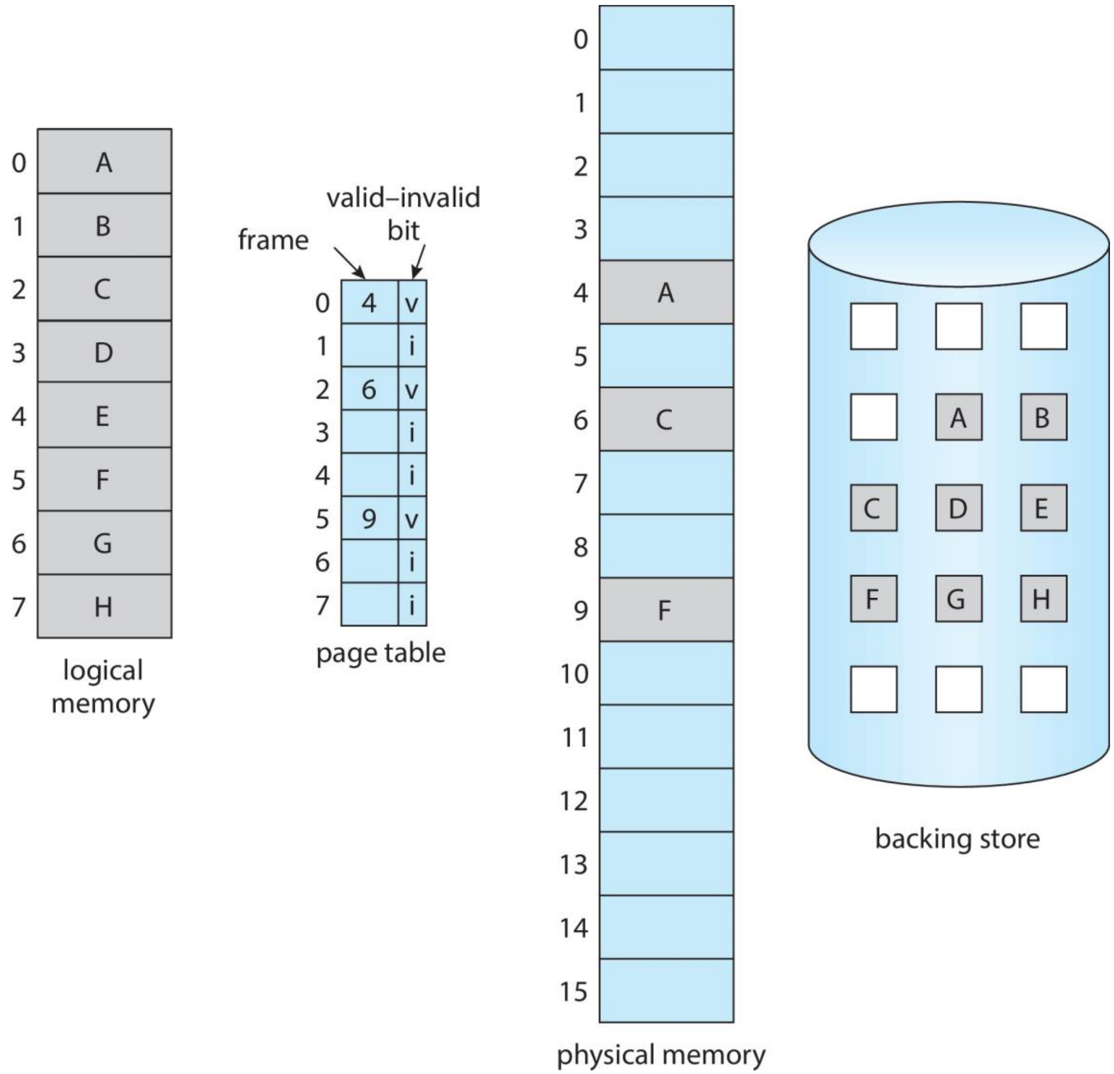
Demand Paging

- **A page rather than the whole process** is brought into memory only when it is needed
  - Less I/O needed ➔ fast response
  - Less memory needed ➔ more users
- Page is needed when there is a reference to the page
  - Invalid reference ➔ abort
  - Not-in-memory ➔ bring to memory via **paging**
- **Pure demand paging**
  - Start a process with no page
  - Never bring a page into memory until it is required

Demand Paging (cont.)

- **A swapper (midterm scheduler) manipulates the entire process**, whereas a **pager is concerned with the individual pages of a process**
- Hardware support
  - **Page table: a valid-invalid bit**
    - 1 ➔ page in memory
    - 0 ➔ page not in memory
    - Initially, all such bits are set to 0
  - Secondary memory (swap space, backing store): usually a high-speed disk (swap device) is used

Demand Paging (cont.)

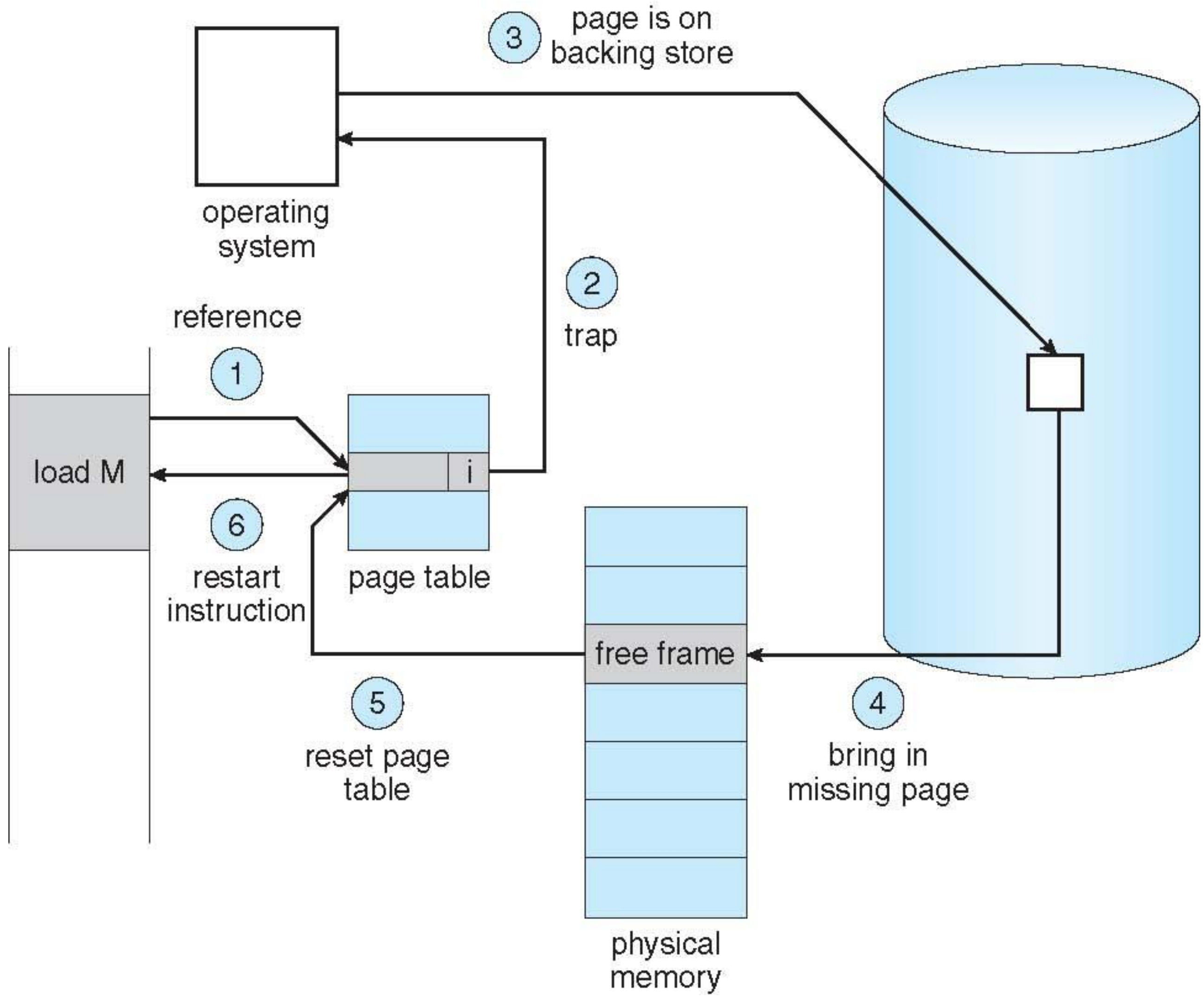


Page Fault

- First reference to a page will trap to OS

  ➔ **page-fault trap**
- OS looks at the internal table (in PCB) to decide
  - Invalid reference ➔ abort
  - Just not in memory ➔ continue
- Get an empty frame
- Swap the page from disk (swap space) into the frame
- Reset page table, valid-invalid bit = 1
- **Restart instruction**

Page Fault Handling



Page Replacement

- If there is no free frame when a page fault occurs
  - Swap a frame to backing store
  - Swap a page from backing store into the frame
  - Different **page replacement algorithms** pick different frames for replacement

Demand Paging Performance

- **Effective Access Time (EAT):** *(1-p) x ma + p x pft*
  - *p: page frame rate; ma: memory access time; pft: page fault time*
- Example: *ma = 200ns, pft = 8ms*
  - EAT = *(1 - p) x 200ns + p x 8ms*

    = 200ns + 7,999,800ns x p
- **Access time is proportional to the page fault rate**
  - If one access out of 1,000 causes a page fault, then

    EAT = 8.2 microseconds (slowdown by a factor of 40!)
  - For degradation less then 10%:

    220 > 200 + 7,999,800 x p

    p < 0.0000025 (one access out of 399,990 to page fault)

Demand Paging Performance (cont.)

- Programs tend to have **locality** of reference
- Locality means program often accesses memory addresses that are close together
  - A single page fault can bring in 4KB memory content
  - Greatly reduce the occurrence of page fault
- Major components of page fault time (about 8 ms)
  - Serve the page-fault interrupt
  - **Read in the page from disk (most expensive)**
  - Restart the instruction

  ➔ The 1st and 3rd can be reduced to several hundred instructions

  ➔ The page switch time is close to 8 ms

Process Creation

Process and Virtual Memory

- **Demand Paging**
  - Only bring in the page containing the first instruction

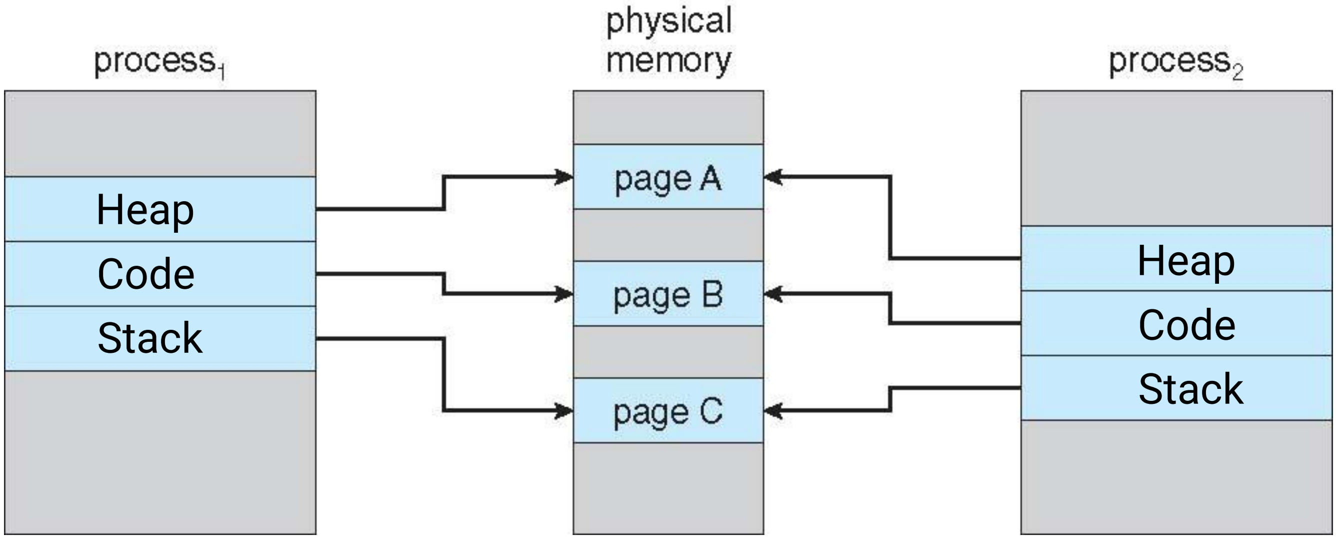
- **Copy-on-Write**
  - The parent and the child process share the same frames initially, and frame-copy when a page is written

Copy-on-Write

- Allow both the parent and the child process to **share the same frames in memory**
- If either process modifies a frame, then a frame is **copied**
- Copy-on-write allows efficient process creation
- Free frames are allocated from a pool of **zeroed-out** frames (security reason)
    - The content of a frame is erased to 0

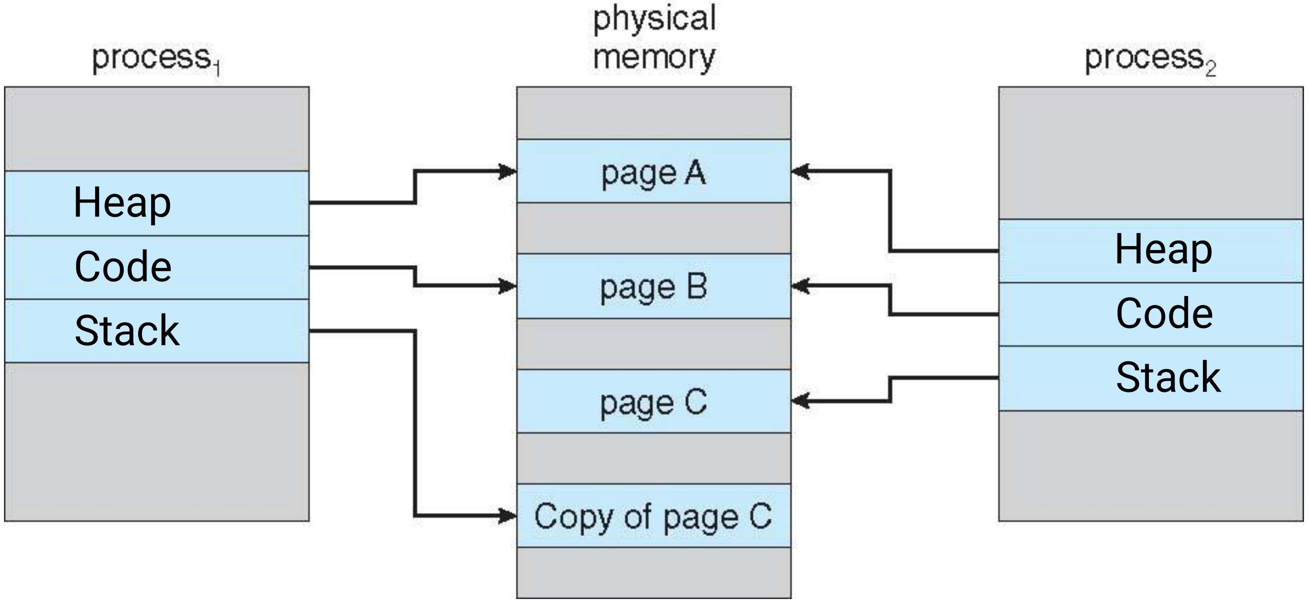
When a Child Process is Forked

```
#include <stdio.h>
void main( )
{
    int A;
    /* fork child process */
    A = fork( );

    if (A != 0) {
      /* parent process */
      int test1=0;
    }
    printf("process ends");
}
```



After a Page is Modified

```
#include <stdio.h>
void main( )
{
   int A;
   /* fork child process */
   A = fork( );

   if (A != 0) {
     /* parent process */
     int test1=0;
   }
   printf("process ends");
}
```



Page Replacement

Page Replacement Concept

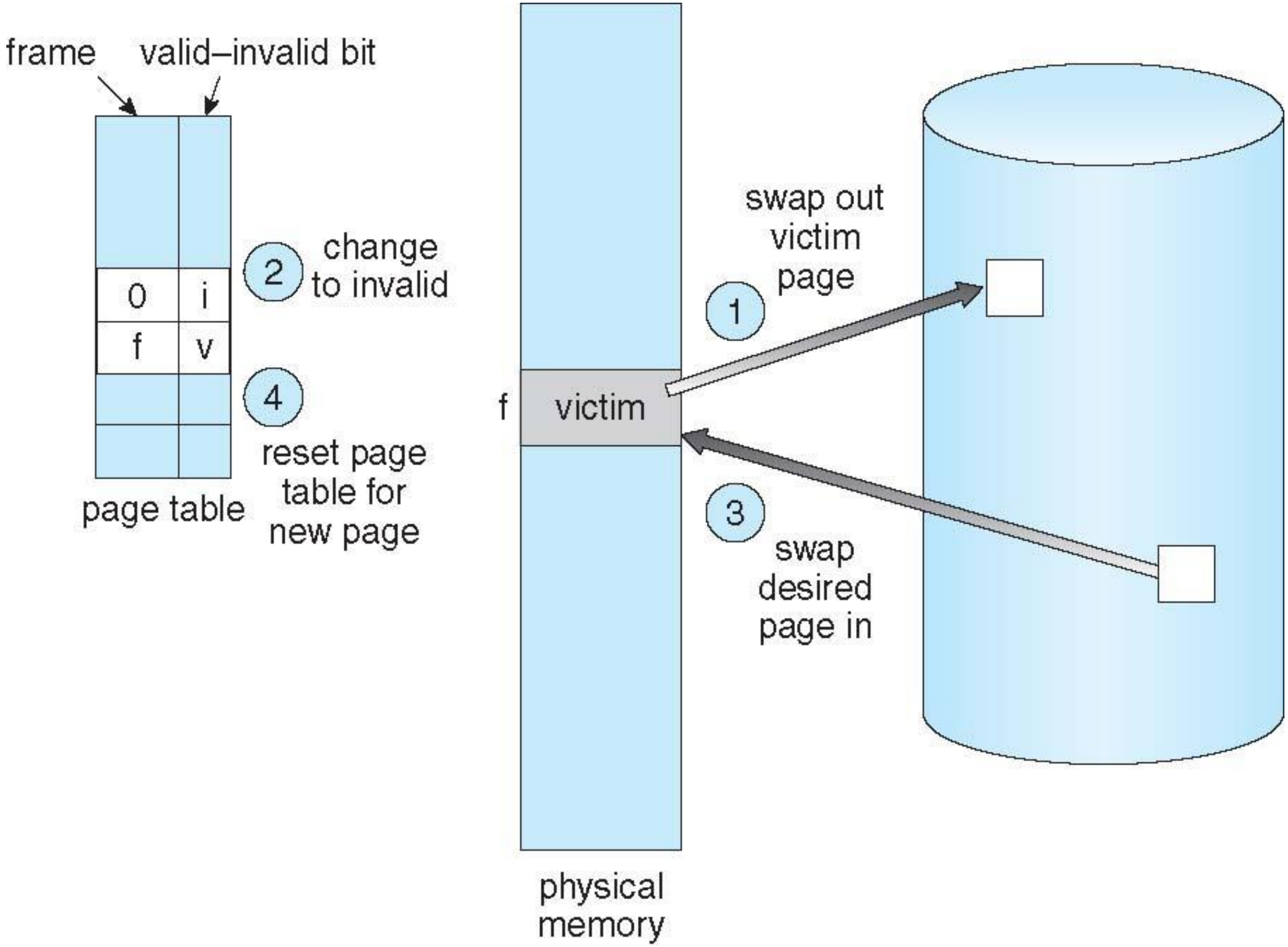
- When a page fault occurs with no free frame
  - **Swap out a process**, freeing all its frames, or
  - **Page replacement**: find one not currently used and free it

- Solve two major problems for demand paging
  - **Frame-allocation algorithm**
    - Determine how many frames to be allocated to a process
  - **Page-replacement algorithm**
    - Select which frame to be replaced

Page Replacement (Page Fault) Steps

- 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
- Read the desired page into the (newly) free frame
- Update the page and frame tables
- Restart the instruction

Page Replacement (Page Fault) Example



Page Replacement Algorithms

- **Goal:** **lowest page-fault rate**
- Evaluation: running against a string of memory references (**reference string**) and computing the number of page faults
- Reference string example:

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

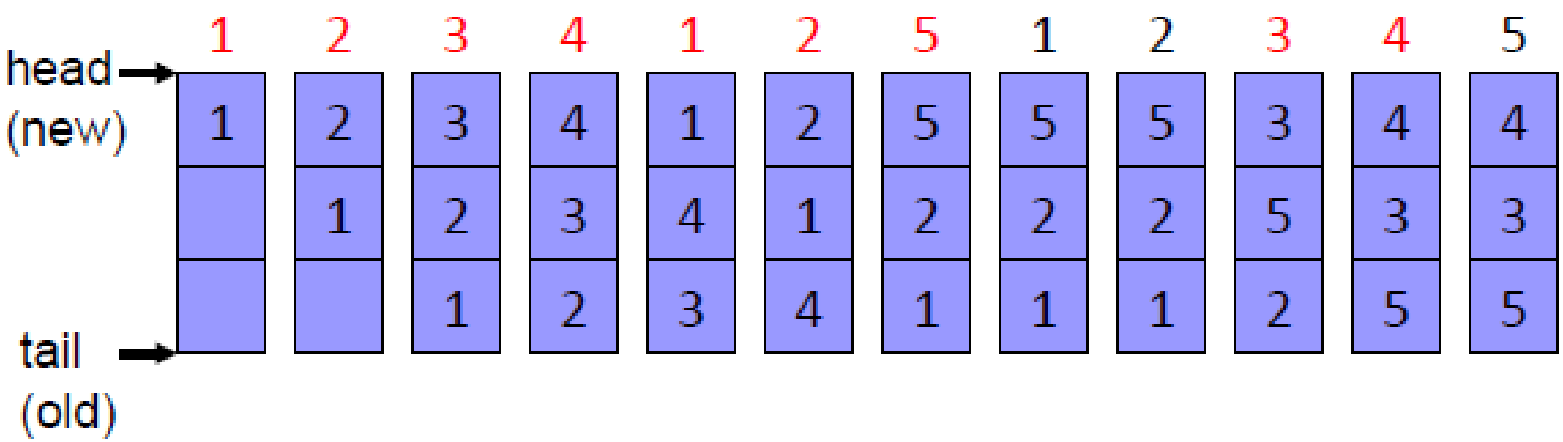
Replacement Algorithms

- FIFO algorithm
- Optimal algorithm
- LRU algorithm
- Counting algorithm
  - LFU
  - MFU

First-In-First-Out (FIFO) Algorithm

- The oldest page in a FIFO queue is replaced
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (available memory frames = 3)

➔ 9 page faults



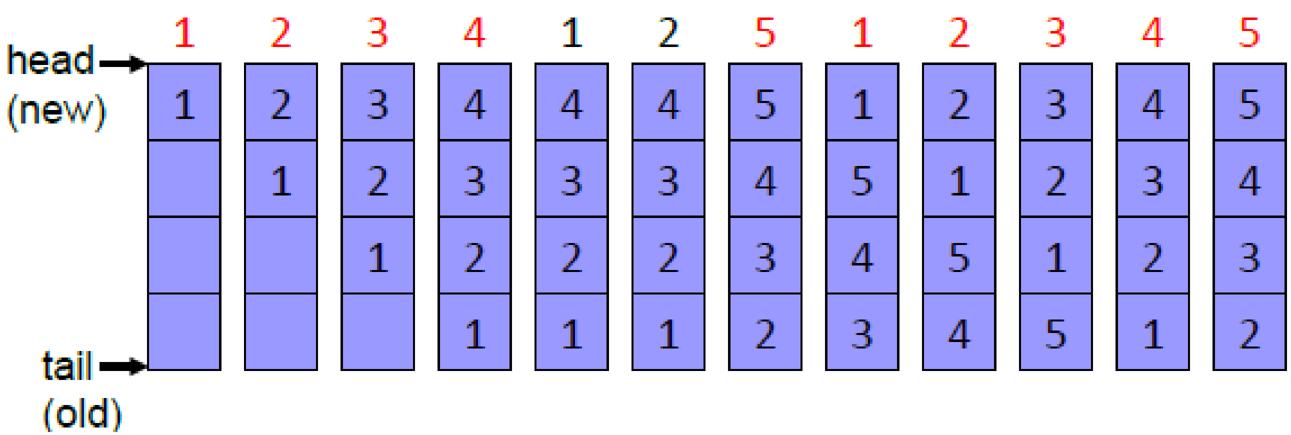
(example borrowed from Prof. Jerry Chou's slides)

FIFO Illustrating Belady's Anomaly

- Does more allocated frames guarantee less page fault?
  - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 4 frames (available memory frames = 4)

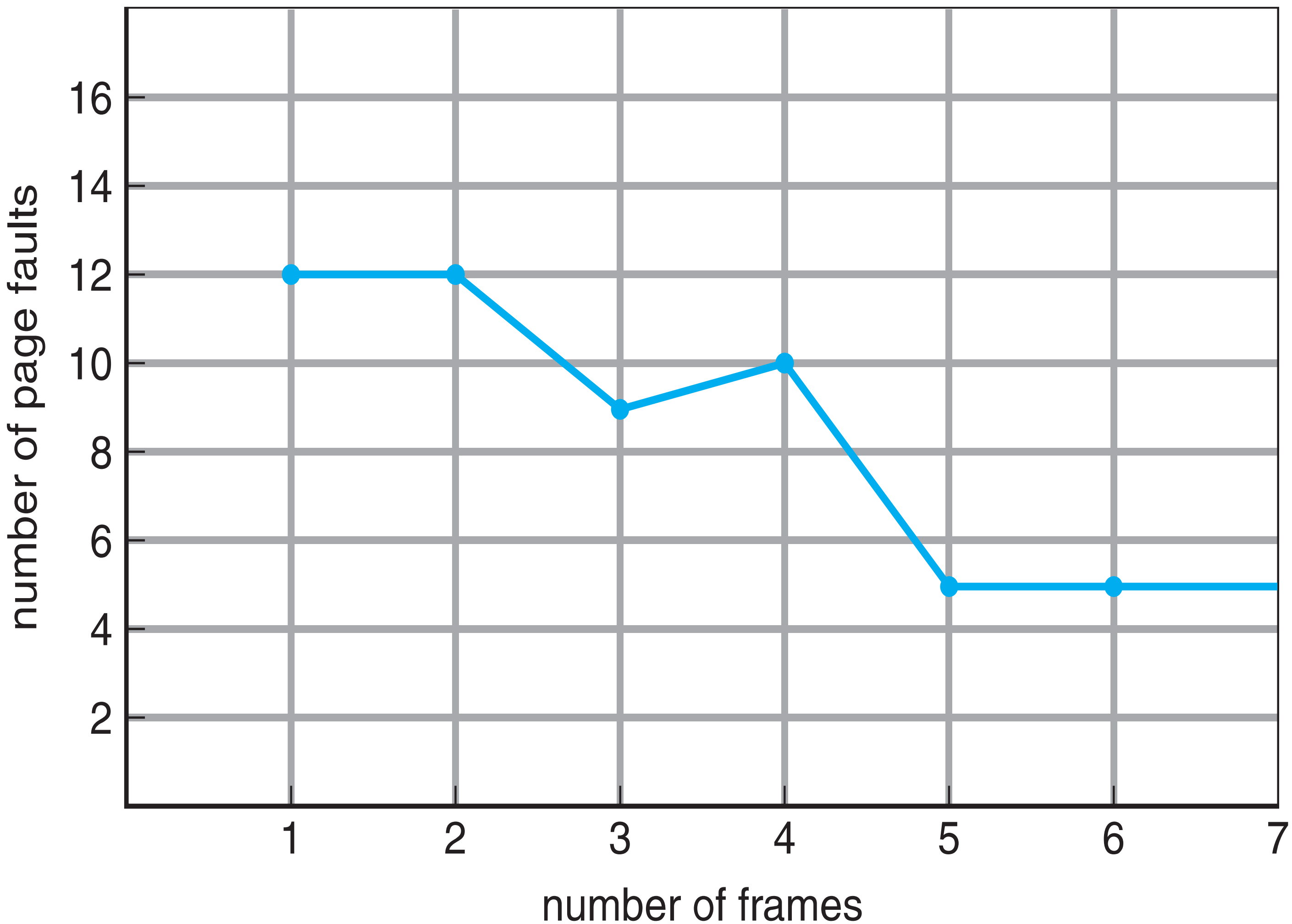
  ➔ **10** page faults !

- **Belady's anomaly**
  - More allocated frames could result in more page faults



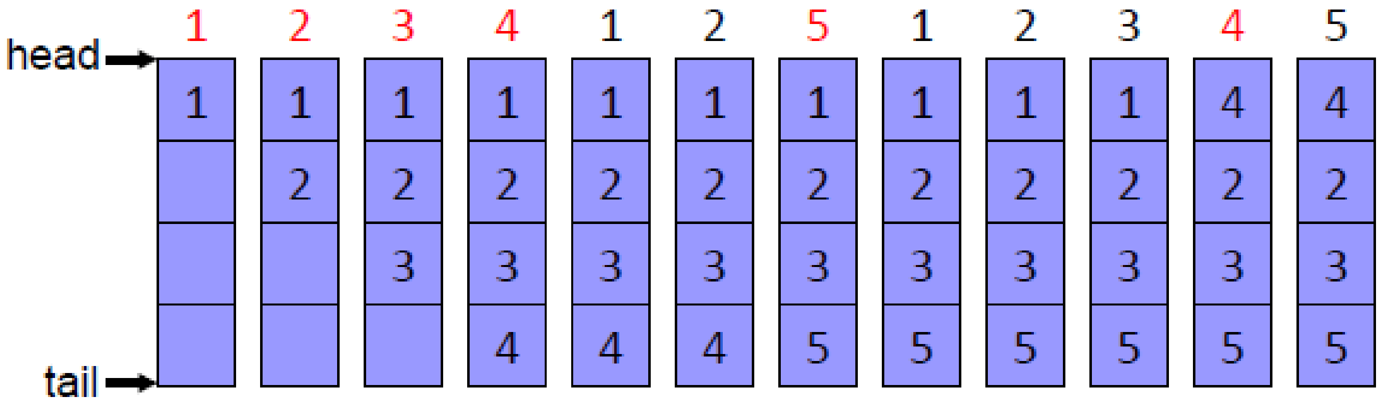
(example borrowed from Prof. Jerry Chou's slides)

FIFO Illustrating Belady's Anomaly (cont.)



Optimal (Belady) Algorithm

- Replace the page that will not be used for the **longest period of time**
  - Need future knowledge
- 4 frames: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 ➔ 6 page faults !
- In practice, we don't have future knowledge
  - Only used for reference and comparison



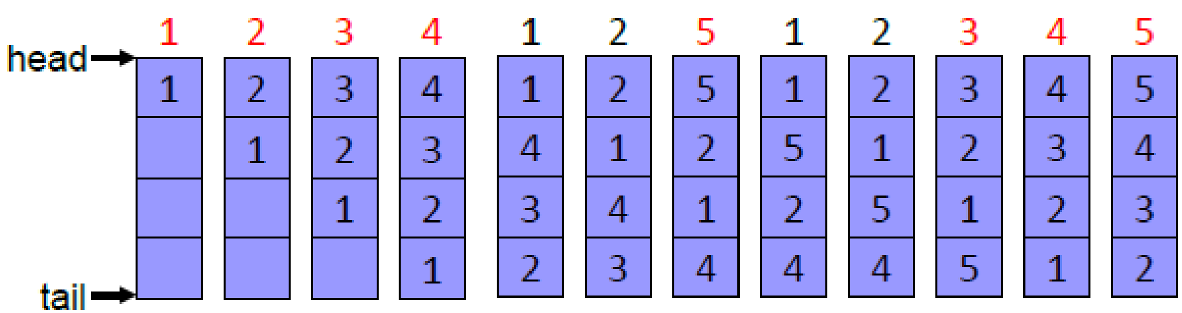
(example borrowed from Prof. Jerry Chou's slides)

LRU (Least Recently Used) Algorithm

- An approximation of optimal algorithm
  - **Looking backward** rather than forward
- It replaces the page that has **not been used for the longest period of time**
- It is often used, and is considered as **quite good**

LRU Algorithm Implementations

- Time stamp implementation
  - Page referenced: **time stamp** is copied into the counter
  - Replacement: remove the one with oldest counter
    - Linear search is required
- **Stack implementation**
  - Page referenced: move to top of the double-linked list
  - Replacement: remove the page at the bottom
  - 4 frames: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 ➔ 8 page faults !



(example borrowed from Prof. Jerry Chou's slides)

Stack Algorithm

- A **property** of algorithms
- **Stack algorithm**
  - The set of pages in memory for *n* frames is always a subset of the set of pages that would be in memory with *n+1* frames
- **Stack algorithms do not suffer from Belady's anomaly**
- Both **optimal** algorithm and **LRU** algorithm are stack algorithm

Counting Algorithm

- **LFU (Least Frequently Used) Algorithm**
  - Keep a counter for each page
  - Idea: an actively used page should have a large reference count
- **MFU (Most Frequently Used) Algorithm**
  - Idea: the page with the smallest count was probably just brought in and has yet to be used
- Both counting algorithms are not common
  - Implementation is expensive
  - Do not approximate OPT algorithm very well

Allocation of Frames

Frame Allocation

- **Fixed allocation**
  - **Equal allocation**

    Example: 100 frames, 5 processes ➔ 20 frame / process
  - **Proportional allocation**

    ➔ Allocate frames according to the size of the process

- **Priority allocation**
  - Using proportional allocation based on priority instead of size
  - If process P generates a page fault
    - Select one of its frame for replacement
    - Select from a process with lower priority for replacement

Frame Allocation (cont.)

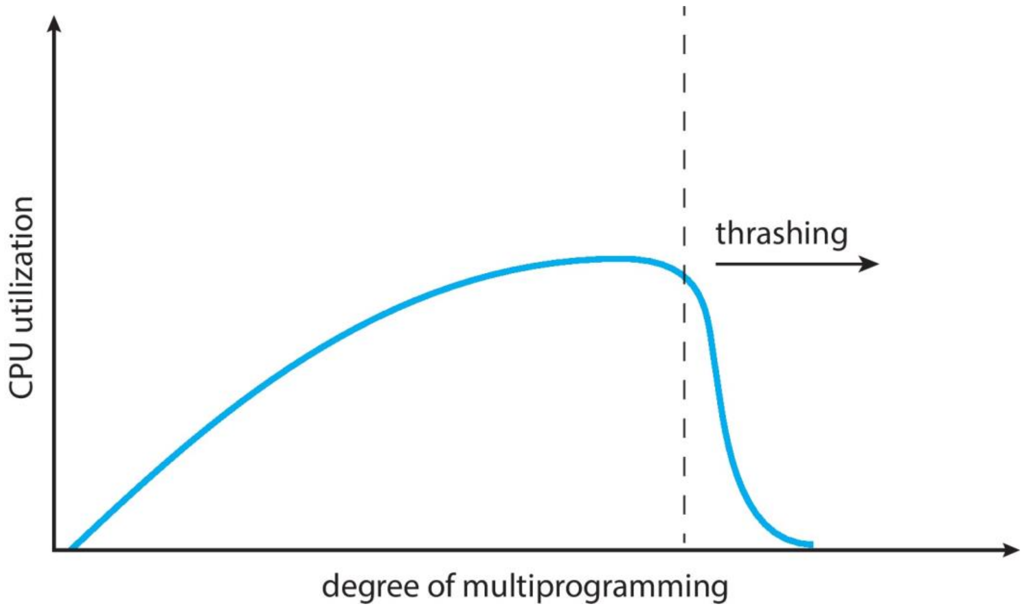
- **Local allocation**
  - Each process select from its own set of allocated frames
- **Global allocation**
  - Process selects a replacement frame from the set of all frames
  - One process can take away a frame of another process
    - E.g., allow a high-priority process to take frames from a low-priority process
  - Good system performance and thus is commonly used
  - **Need to prevent thrashing**

Thrashing

Definition of Thrashing

- If a process does not have enough **frames**
  - The process does not have # frames it needs to support pages in active use

    ➔ Very high paging activity
- **A process is thrashing if it is spending more time paging than executing**



Thrashing

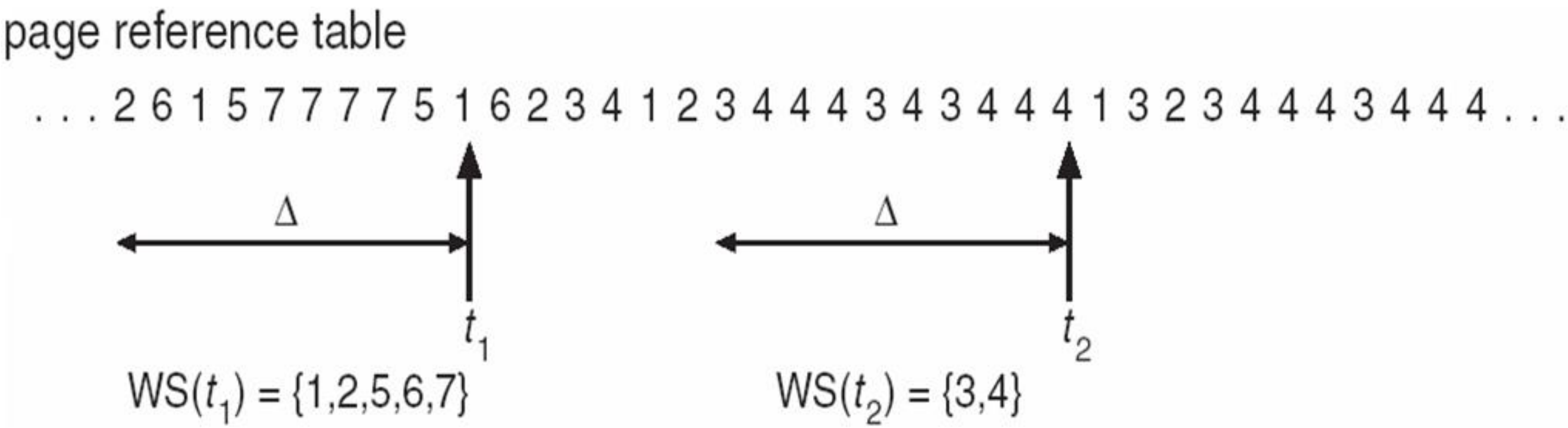
- Performance problem caused by thrashing (assume global replacement is used)
  - Processes queued for I/O to swap (page fault)
    - ➔ Low CPU utilization
    - ➔ OS increases the degree of multi-programming
    - ➔ New processes take frames from old processes
    - ➔ More page faults and thus more I/O
    - ➔ CPU utilization drops even further
- To prevent thrashing, must provide enough frames for each process
  - **Working-set model**
  - **Page-fault frequency**

Working-Set Model

- **Locality**: a set of pages that are actively used together
- Locality model: as a process executes, it moves from locality to locality
  - Program structure (subroutine, loop, stack)
  - Data structure (array, table)
- Working-set model (based on locality model)
  - Working-set **window**: a parameter Δ (delta)
  - Working-set: set of pages in most recent Δ page references (an approximation locality)

Working-Set Example

- If Δ (delta) = 10



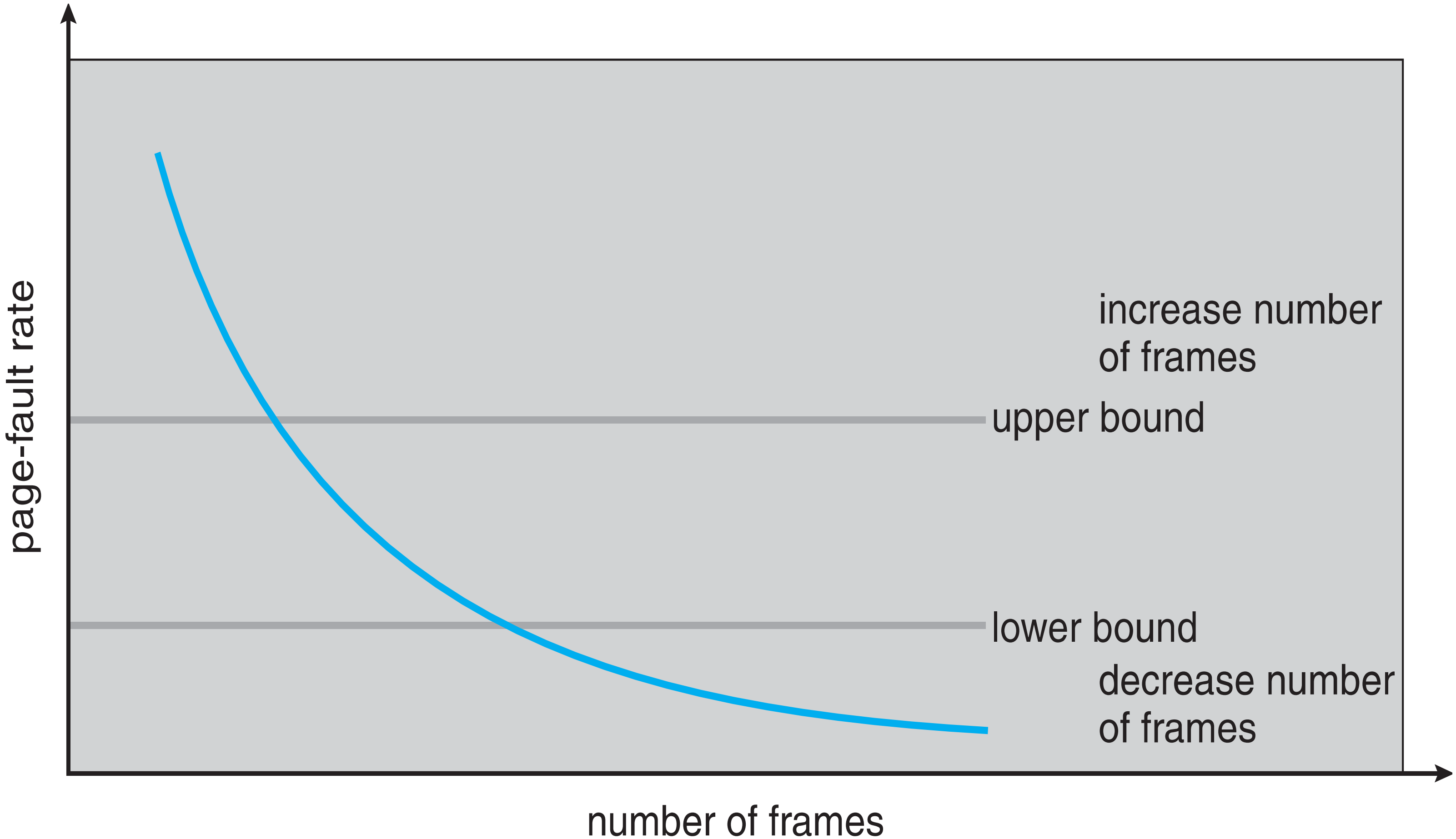
Working-Set Model (cont.)

- Prevent thrashing using the working-set size
  - WSS: working-set size for process *i*
  - $D = \sum WSS_i$ (total demand frames)
  - if D > m (available frames) ➔ thrashing
  - The OS monitors the WSS_i of each process and allocates to the process enough frames
    - if D << m, increase degree of MP
    - If D > m, suspend a process
- Prevent thrashing while keeping the degree of multiprogramming as high as possible
- Optimize CPU utilization
- **However, too expensive for tracking**

Page Fault Frequency Scheme

- **Page fault frequency** directly measures and controls the page-fault rate to prevent thrashing
  - Establish **upper** and **lower** bounds on the desired page-fault rate of a process
  - If page fault rate exceeds the upper limit
    - Allocate another frame to the process
  - If page rate falls below the lower limit
    - Remove a frame from the process

Page Fault Frequency Scheme (cont.)



Objective Review

- Define virtual memory and describe its benefits
- Illustrate how pages are loaded into memory using demand paging
- Apply the FIFO, optimal, and LRU page-replacement algorithms
- Describe the working set of a process and explain how it is related to program locality