



# Operating System: Chap9 Virtual Memory Management

National Tsing Hua University  
2021, Fall Semester

# Overview

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

# 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 memory utilization

# Background

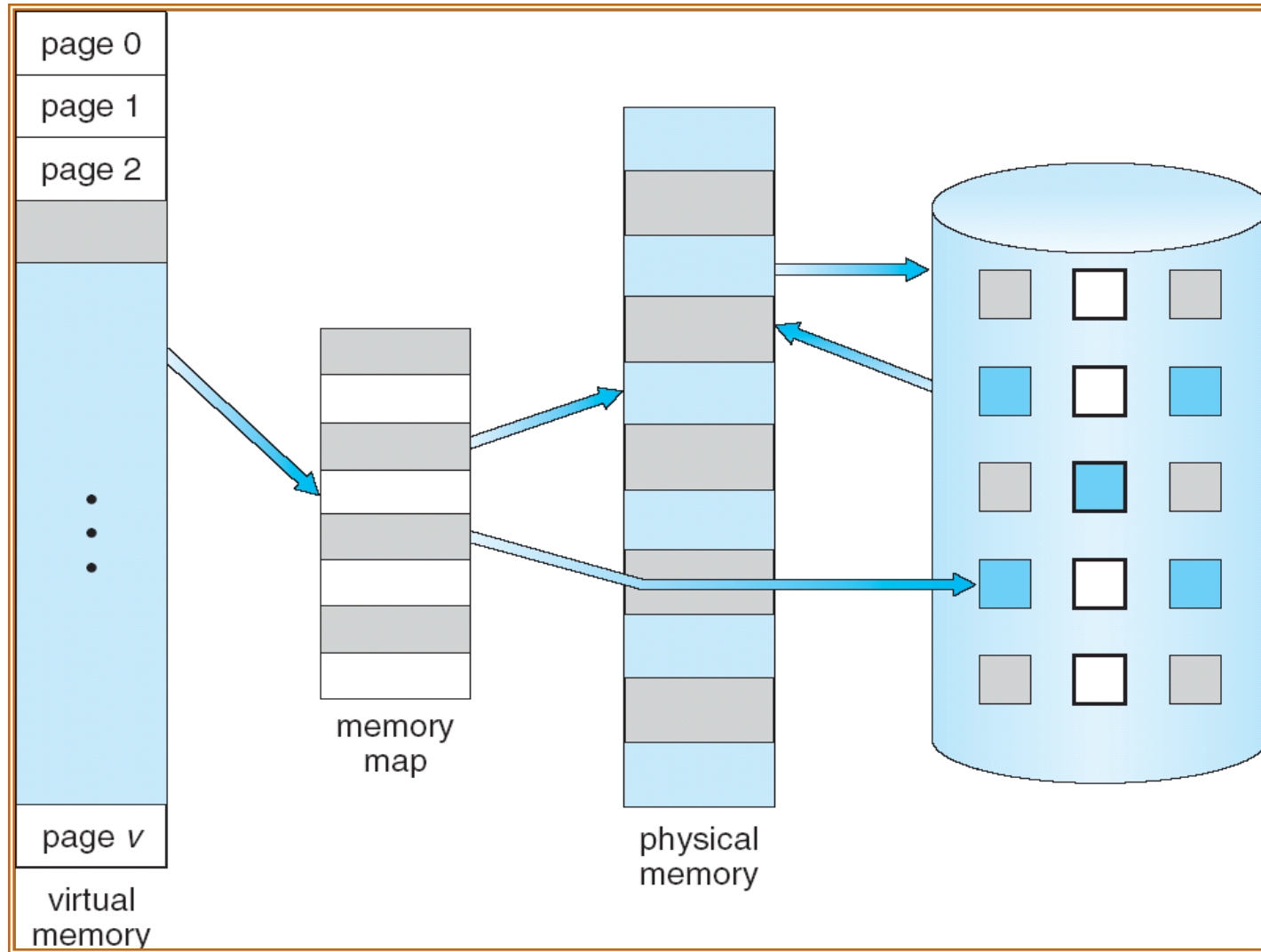
## ■ **Virtual memory** – separation of user logical memory from physical memory

- To run an extremely **large process**
  - ◆ Logical address space can be much larger than physical address space
- To increase **CPU/resources utilization**
  - ◆ higher degree of multiprogramming degree
- To **simplify programming** tasks
  - ◆ Free programmer from memory limitation
- To run programs **faster**
  - ◆ less I/O would be needed to load or swap

## ■ Virtual memory can be implemented via

- **Demand paging**
- Demand segmentation: more complicated due to variable sizes

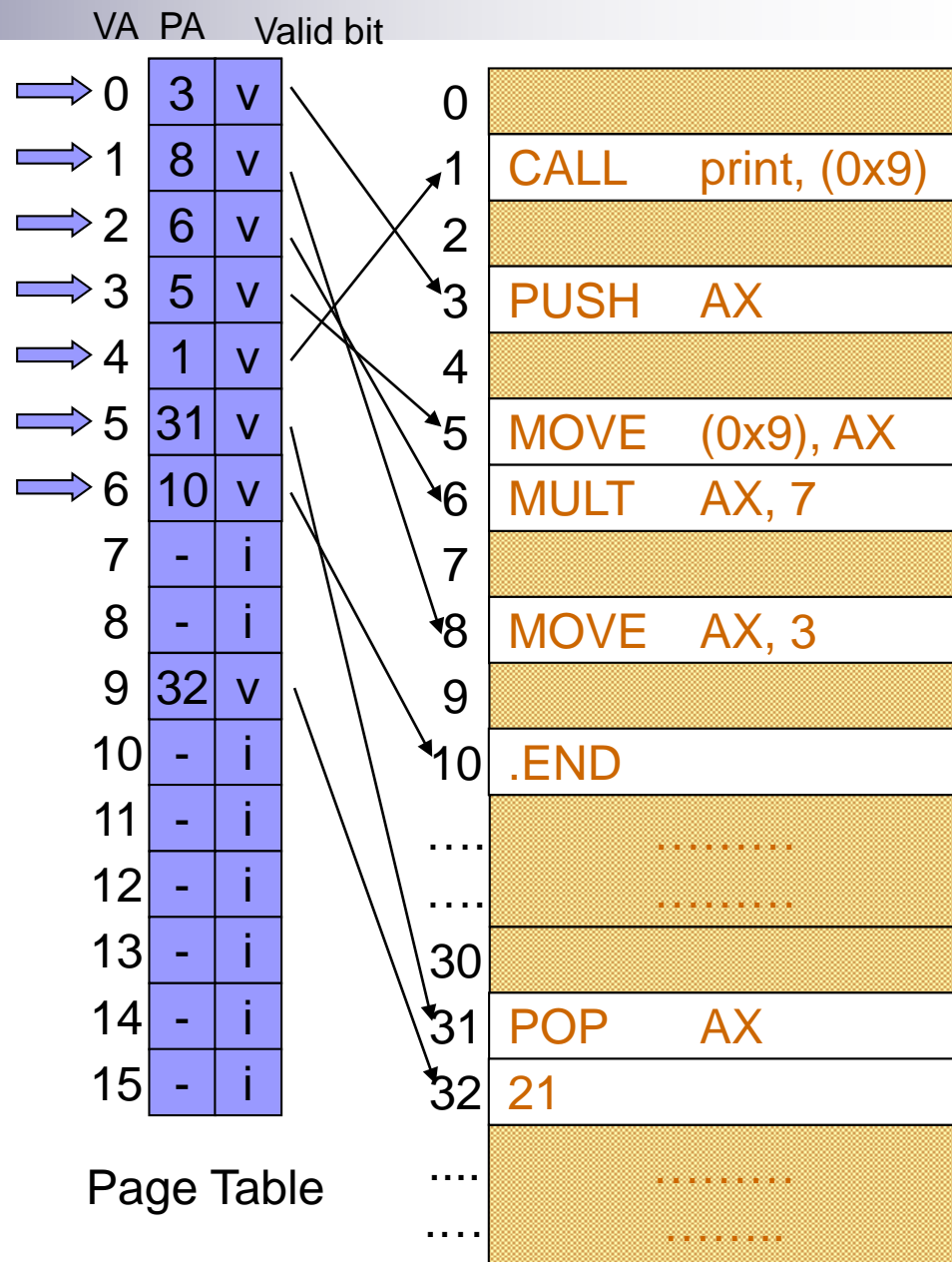
# Virtual Memory vs. Physical Memory



1. Initialize PCB, PC registers and Page Table.
2. Load Code into memory.
3. Running
4. Finish.

```
int data[8];
main( ) {
    data[3] = 3 * 7;
    print(data);
}
```

```
.START
0 PUSH    AX
1 MOVE    AX, 3
2 MULT    AX, 7
3 MOVE    (0x9), AX
4 CALL    print, (0x9)
5 POP     AX
6 .END
.SPACE (8)
```



Page Table

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 → Shorter process creation time
  - 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

- 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 use

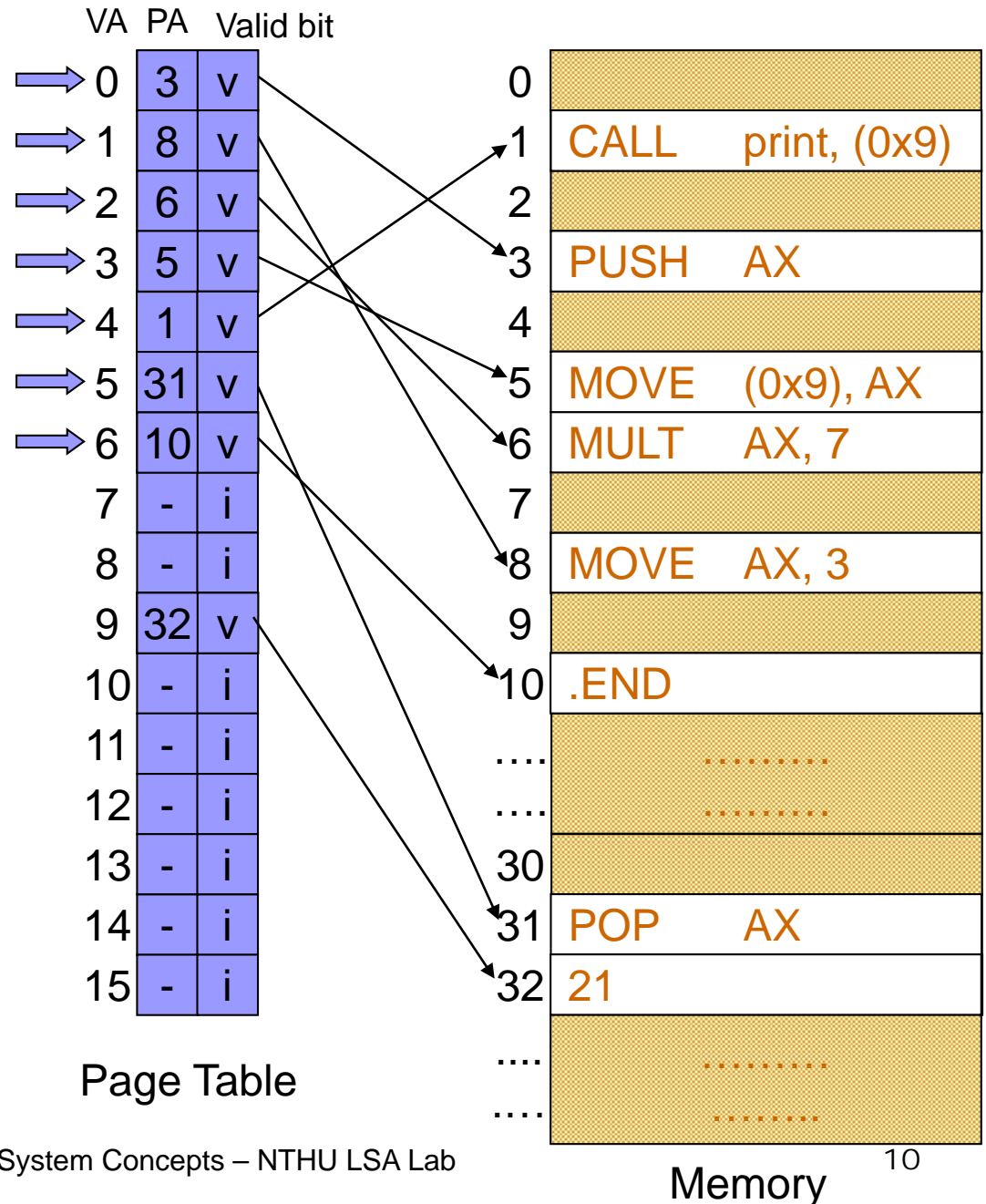
# Demand Paging

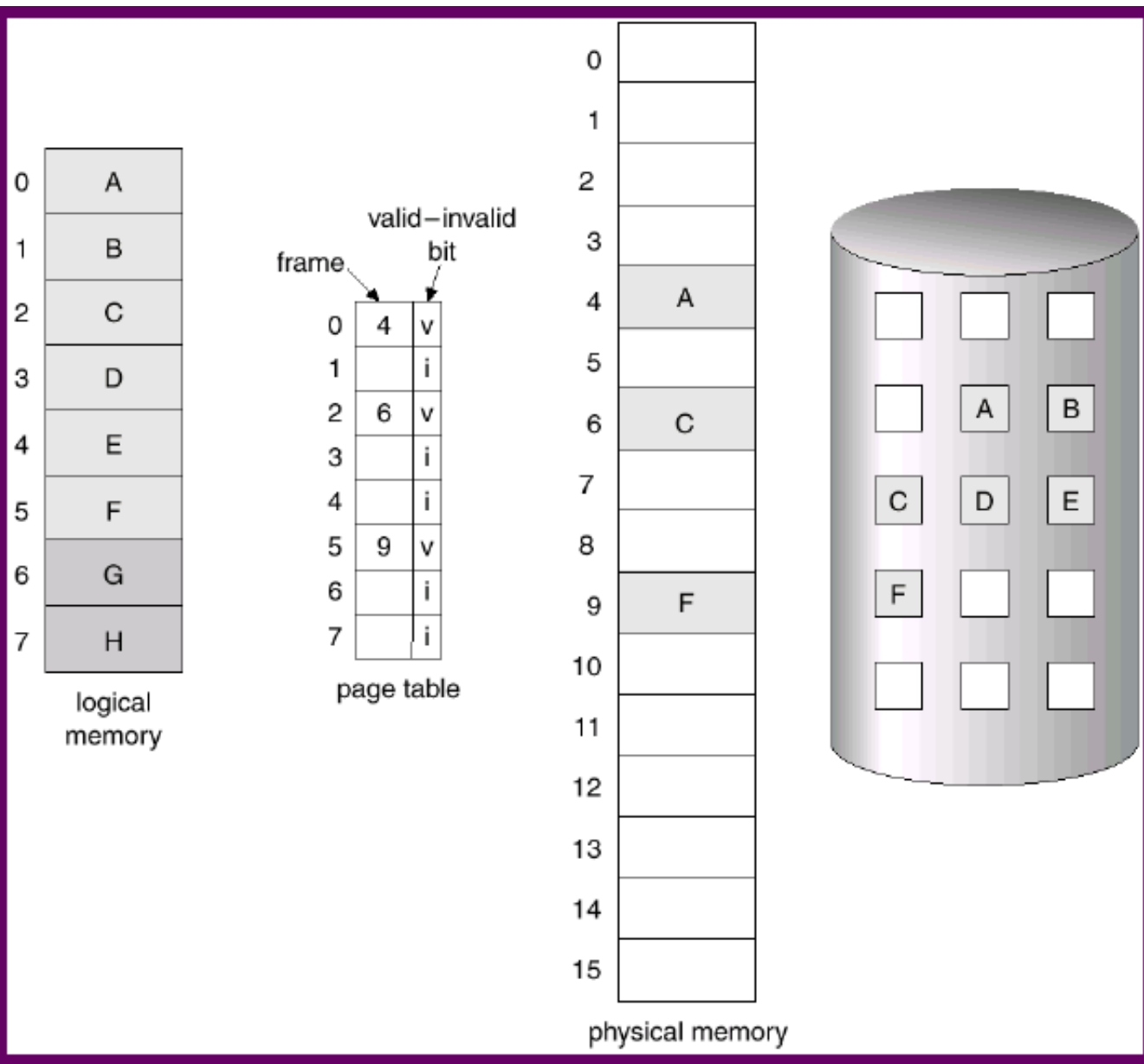
```
int data[8];
main() {
    data[3] = 3 * 7;
    print(data);
}
```

```

→ 0 .START
→ 0 PUSH    AX
→ 1 MOVE    AX, 3
→ 2 MULT    AX, 7
→ 3 MOVE    (0x9), AX
→ 4 CALL    print, (0x9)
→ 5 POP     AX
→ 6 .END
    .SPACE (8)

```

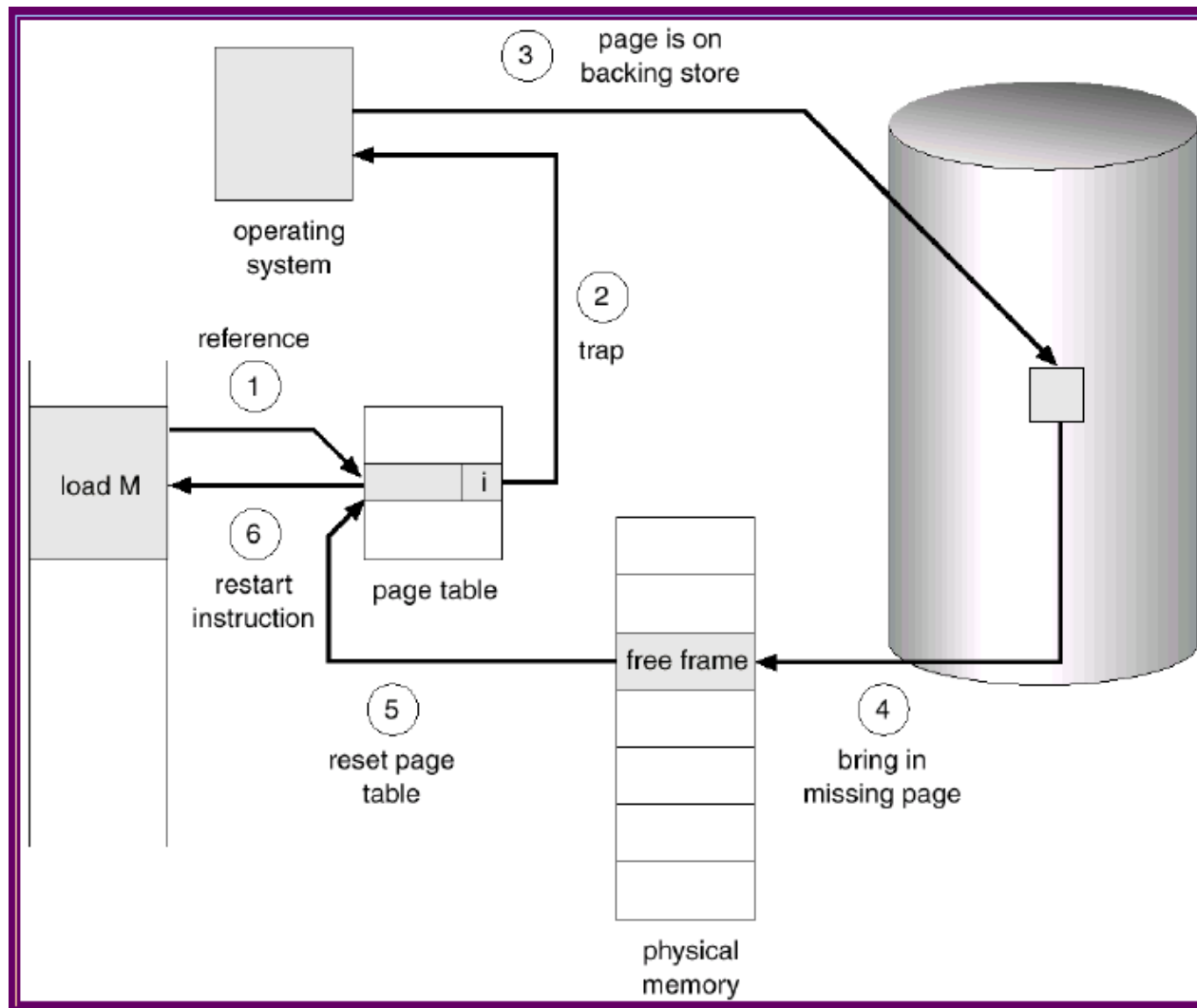




# Page Fault

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

# Page Fault Handling Steps



# 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):**  $(1 - p) \times ma + p \times pft$ 
  - $P$ : **page fault rate**,  $ma$  : mem. access time,  $pft$  : page fault time
- **Example:**  $ma = 200\text{ns}$ ,  $pft = 8\text{ms}$ 
  - $EAT = (1 - p) * 200\text{ns} + p * 8\text{ms}$   
 $= 200\text{ns} + 7,999,800\text{ns} \times p$
- **Access time is proportional to the page fault rate**
  - If one access out of 1,000 causes a page fault, then  
 $EAT = 8.2 \text{ microseconds}$ . ➔ **slowdown by a factor of 40!**
  - For degradation less than 10%:  
 $220 > 200 + 7,999,800 \times p$ ,  
 $p < 0.0000025$  ➔ one access out of 399,990 to page fault

# Demand Paging Performance (Con't)

- 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)
  1. serve the page-fault interrupt
  2. **read in the page from disk (most expensive)**
  3. restart the process
  - The 1<sup>st</sup> and 3<sup>rd</sup> can be reduced to several hundred instructions
  - The page switch time is close to 8ms





# Process Creation

# Process & 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
- **Memory-Mapped File:** map a file into the virtual address space to bypass file system calls (e.g., read(), write())

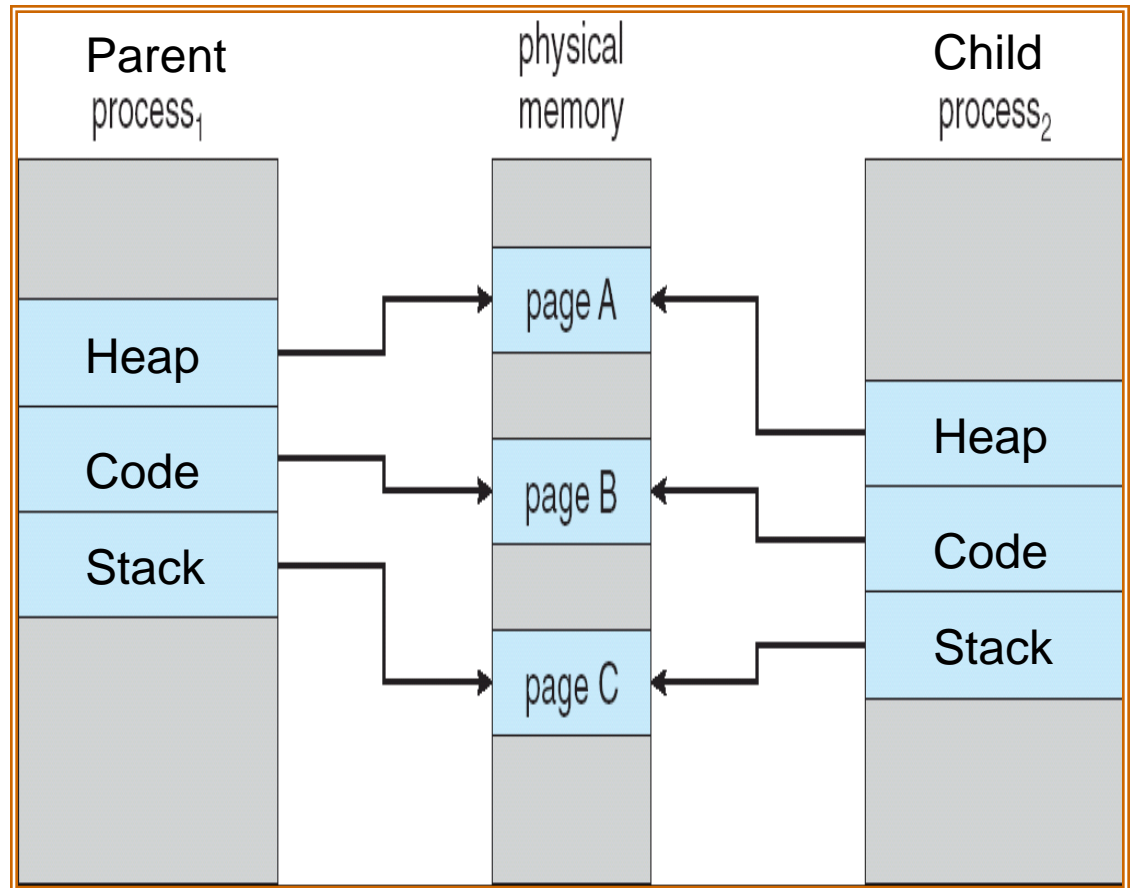
# Copy-on-Write

- Allow both the parent and the child process to share the same frames in memory
- If either process modifies a frame, only then a frame is copied
- COW allows efficient process creation (e.g., `fork()` )
- 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( );

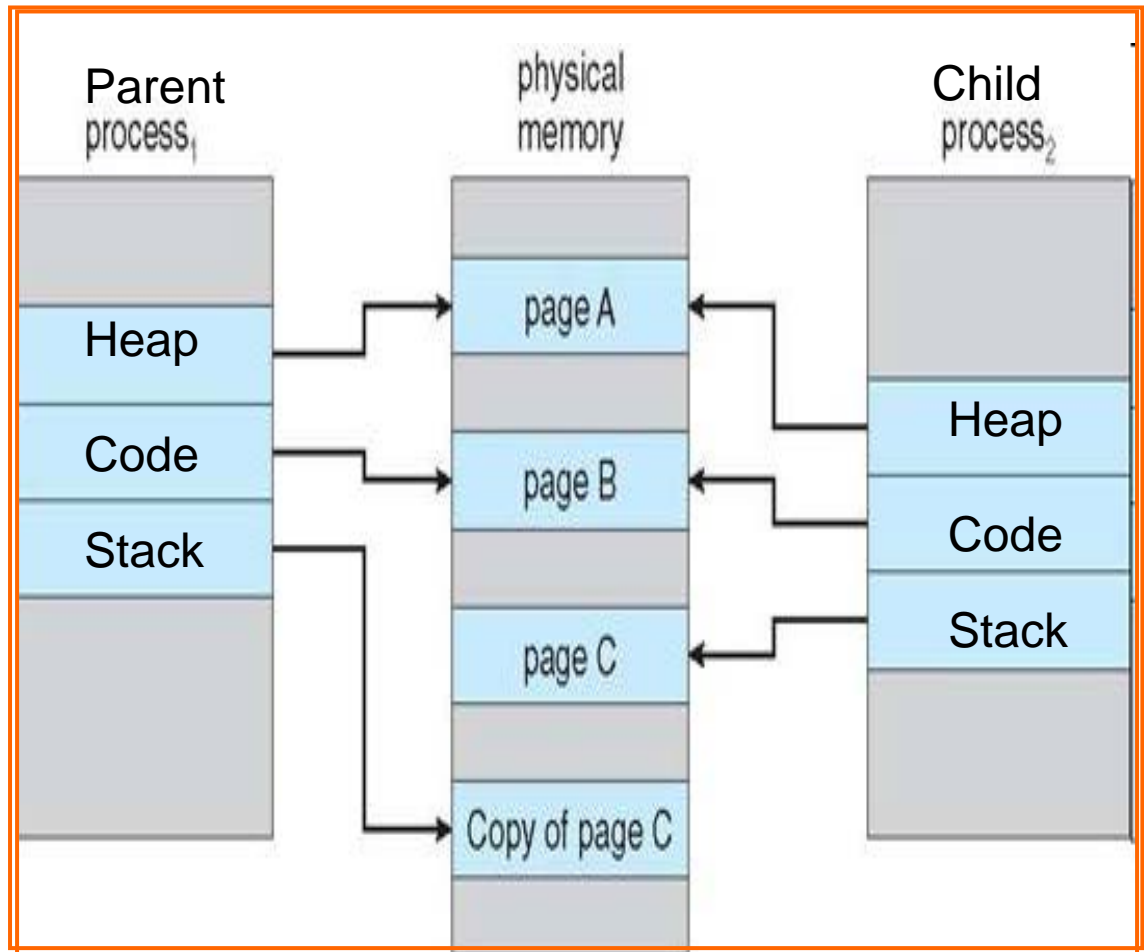
    if (A != 0) {
        /* parent process */
        int test1=0;
    }
    printf("process ends");
}
```



# 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");
}
```



# Memory-Mapped Files

## ■ Approach:

- MMF allows file I/O to be treated as *routine memory access* by mapping a disk block to a memory frame
- A file is initially read using demand paging. Subsequent reads/writes to/from the file are treated as ordinary memory accesses

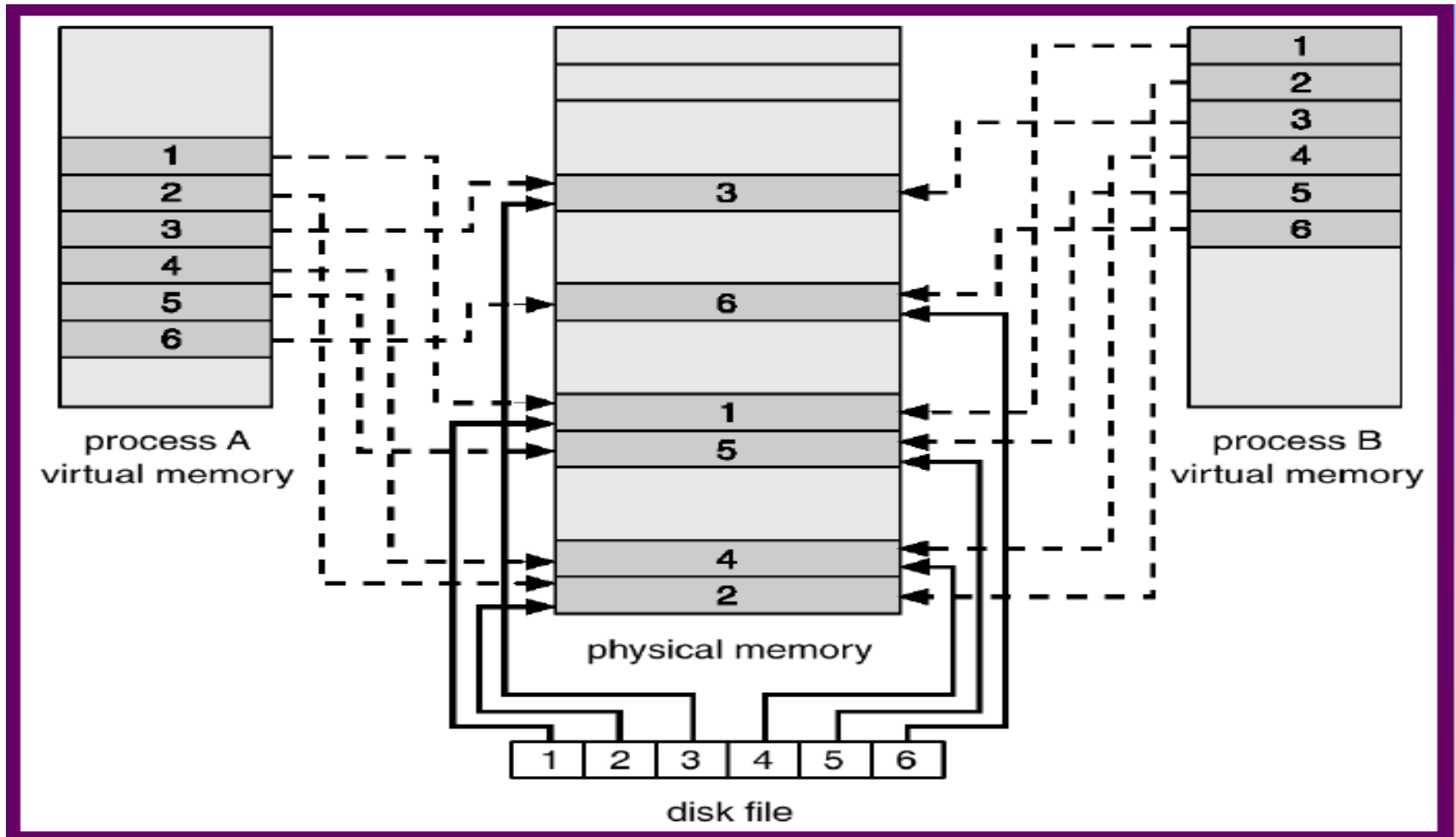
## ■ Benefit:

- **Faster file access** by using memory access rather than **read()** and **write()** system calls
- Allows several processes to map the SAME file allowing the pages in memory to be **SHARED**

## ■ Concerns:

- **Security**(access control), data lost, more programming efforts

# Memory-Mapped File Example



```

int buf;
int fd = open( filename, O_RDWR );
lseek( fd, 1024, SEEK_SET );
→ read( fd, &buf, sizeof(int) );
→ buf ++;
lseek( fd, 1024, SEEK_SET );
→ write( fd, &buf, sizeof(int) );
→ close(fd);

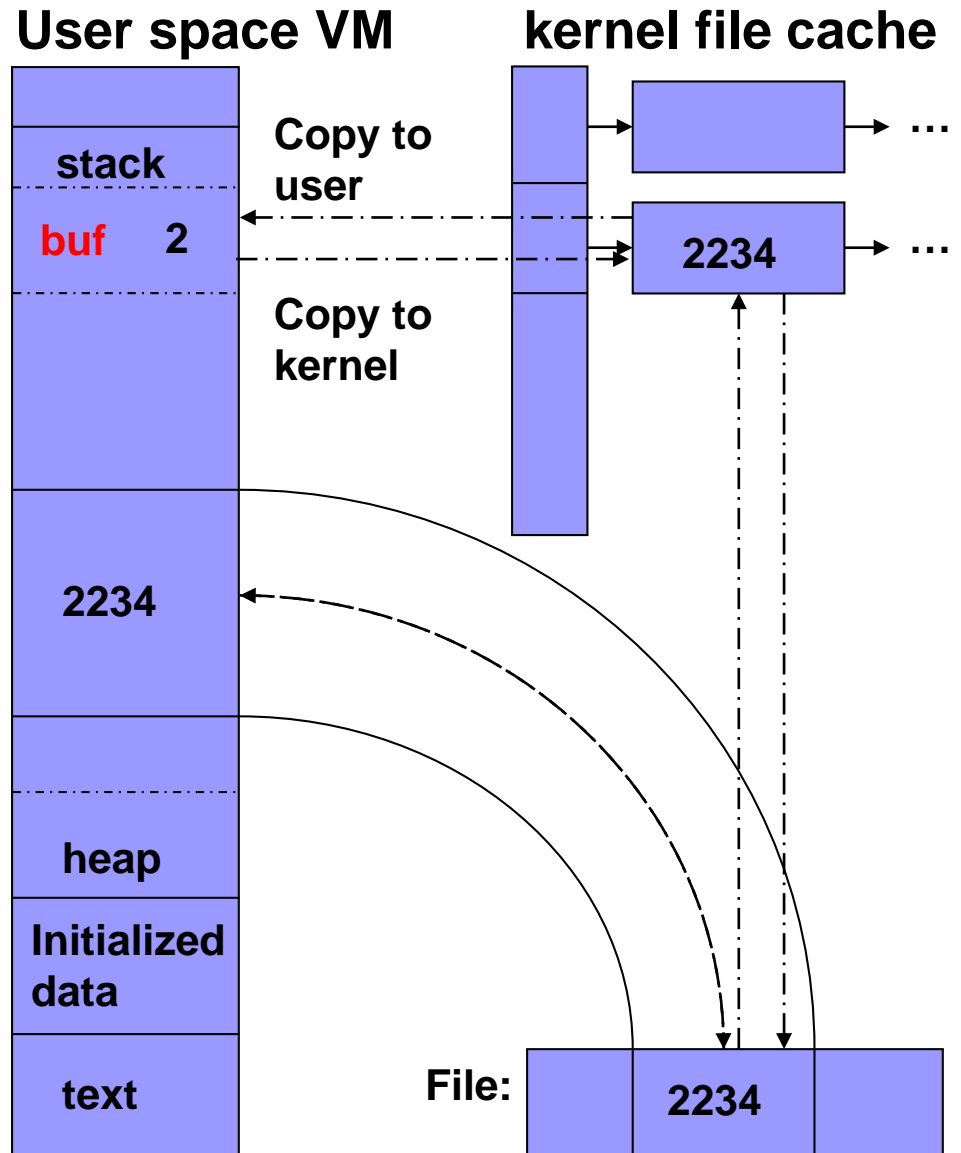
```

**Memory mapped portion**

```

int fd = open( filename, O_RDWR );
→ int* area = mmap( 0, BUFSIZE,
    PROT_READ | PROT_WRITE,
    MAP_SHARED, fd, 1024 );
→ area[0]++;
close(fd);
→ munmap( area, BUFSIZE );

```





# Review Slides ( I )

- Virtual memory? Physical Memory?
- Demand paging?
- Page table support for demand paging?
- OS handling steps for page fault?
- Page replacement?
- Copy-on-write? Usage?
- Memory-mapped file? Usage?



# Page Replacement

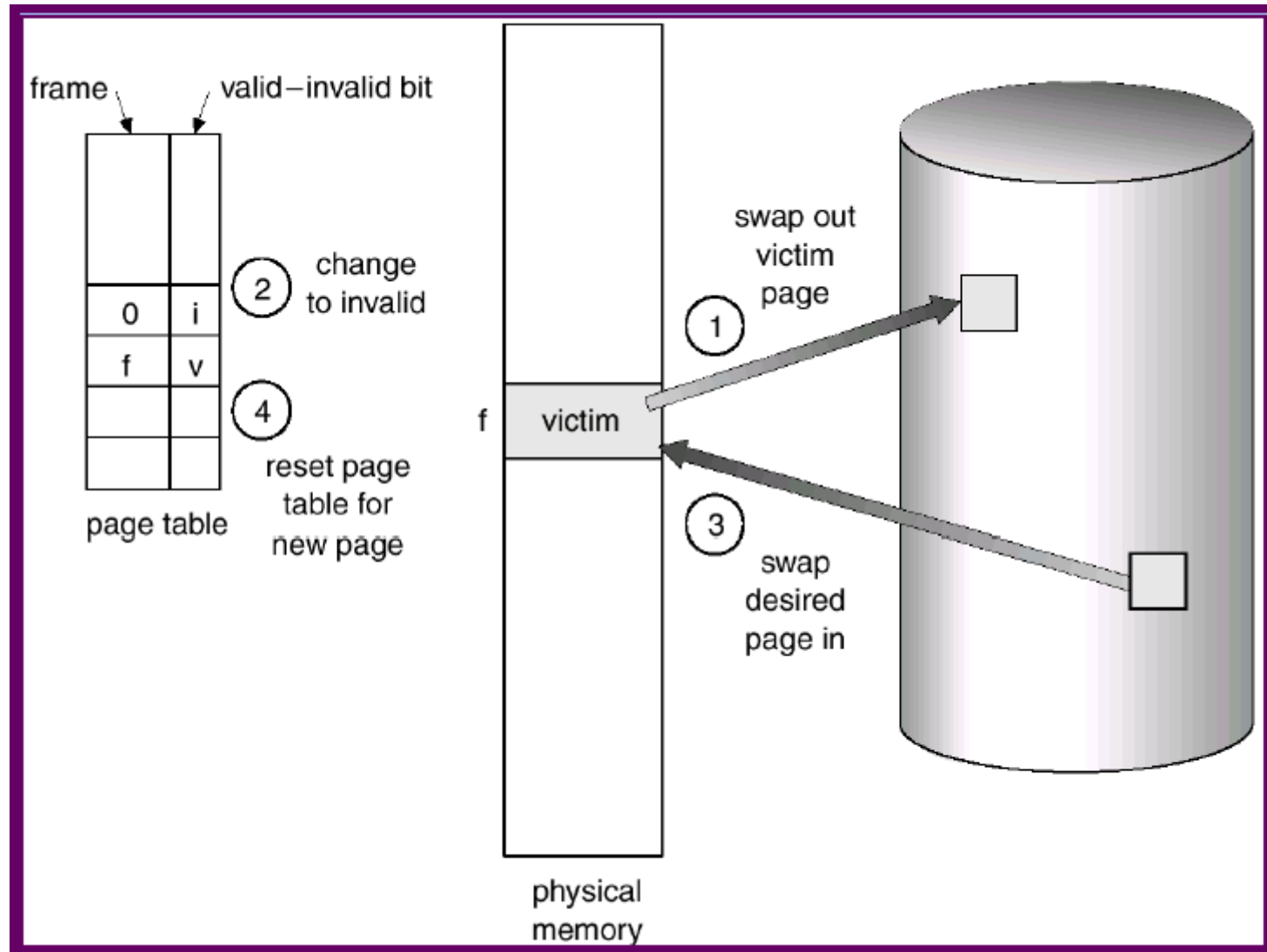
# Page Replacement Concept

- 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
    - ◆ Use *dirty bit* to reduce overhead of page transfers – only modified pages are written to disk
- 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

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
3. Read the desired page into the (newly) free frame. Update the page & frame tables
4. Restart the process

# Page Replacement (Page Fault) Example



# Page Replacement Algorithms

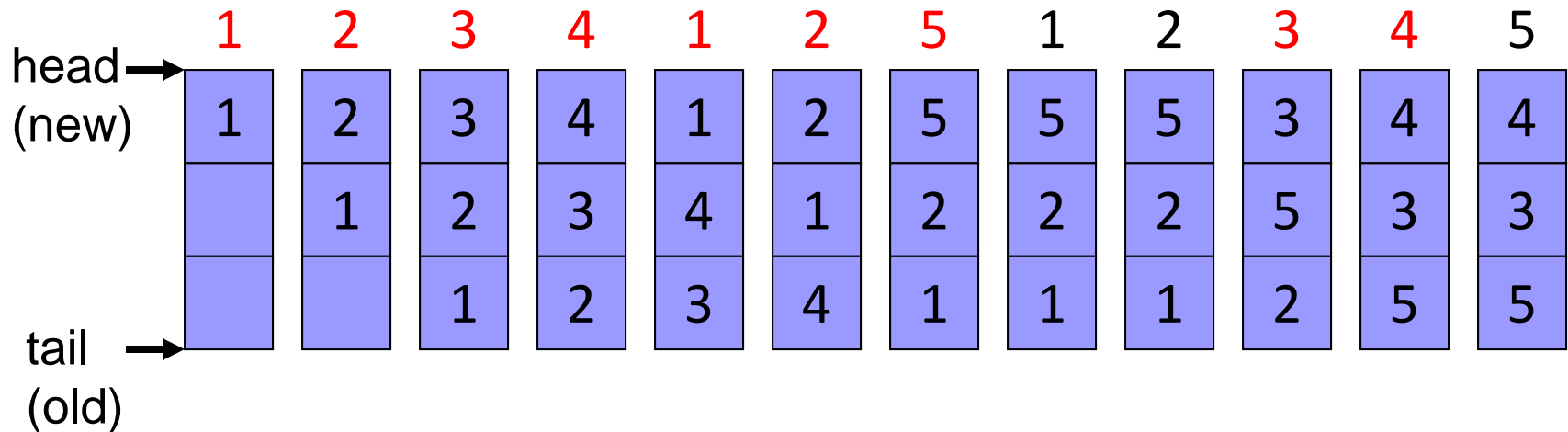
- Goal: **lowest page-fault rate**
- Evaluation: running against a string of memory references (**reference string**) and computing the number of page faults
- Reference string:  
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

# 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





# FIFO Illustrating Belady's Anomaly

## ■ Does more allocated frames guarantee less page fault?

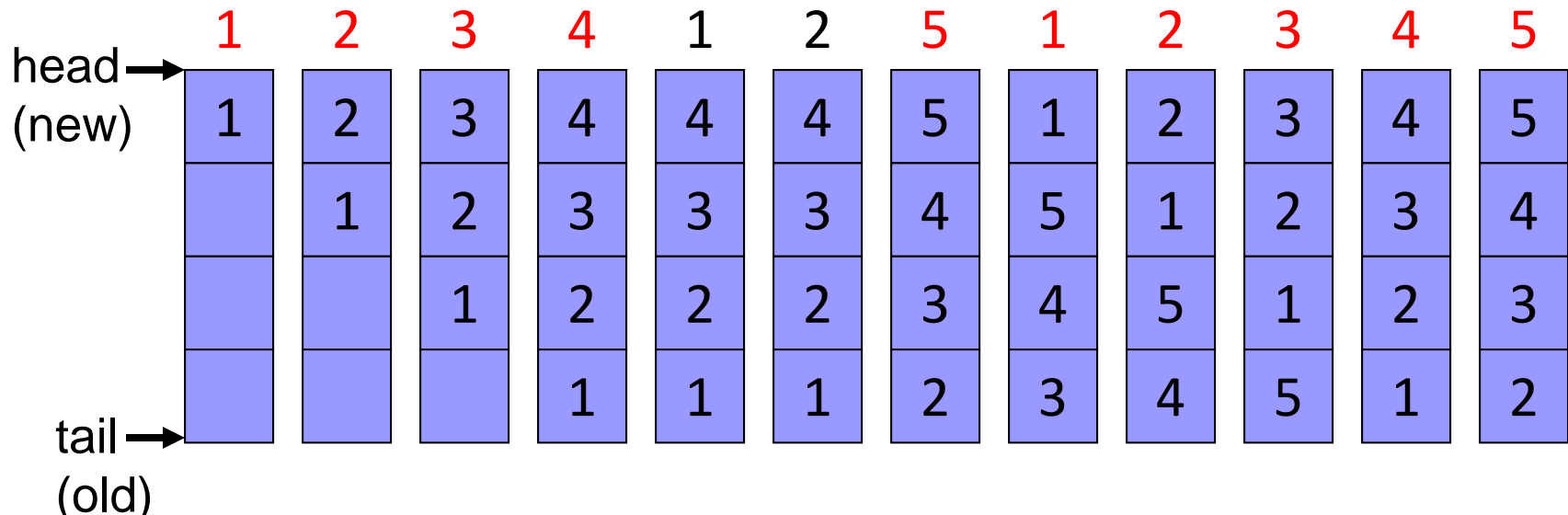
➤ Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

➤ 4 frames (available memory frames = 4)

➔ 10 page faults!

## ■ Belady's anomaly

➤ Greater allocated frames ➔ more page fault

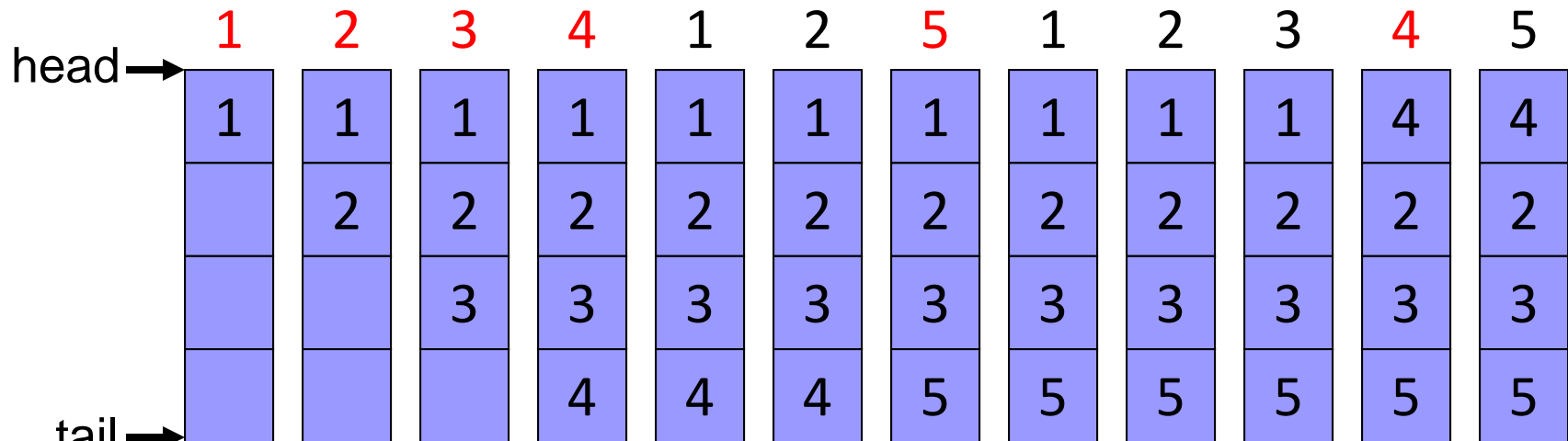


# FIFO Illustrating Belady's Anomaly



# 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 & comparison



# LRU Algorithm (Least Recently Used)

- 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

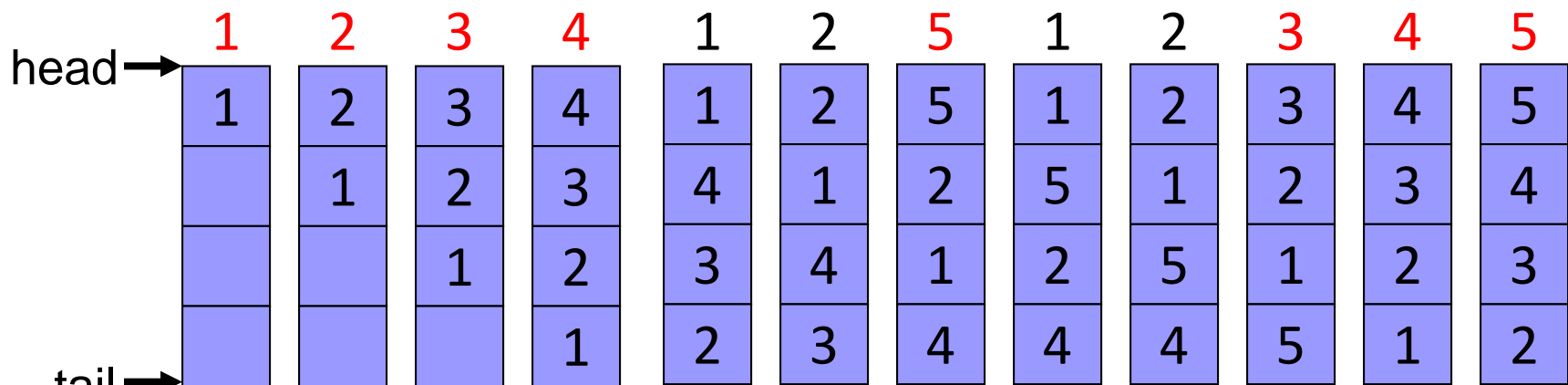
# LRU Algorithm Implementations

## ■ Counter 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!



# 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 suffers from Belady's anomaly
- Both **optimal** algorithm and **LRU** algorithm are stack algorithm

# LRU approximation algorithms

- **Few systems** provide sufficient hardware support for the LRU page-replacement
  - additional-reference-bits algorithm
  - second-chance algorithm
  - enhanced second-chance algorithm

# Counting Algorithms

## ■ LFU Algorithm (least frequently used)

- keep a counter for each page
- Idea: An actively used page should have a large reference count

## ■ MFU Algorithm (most frequently used)

- Idea: The page with the smallest count was probably just brought in and has yet to be used

## ■ Both counting algorithm are not common

- implementation is expensive
- do not approximate OPT algorithm very well



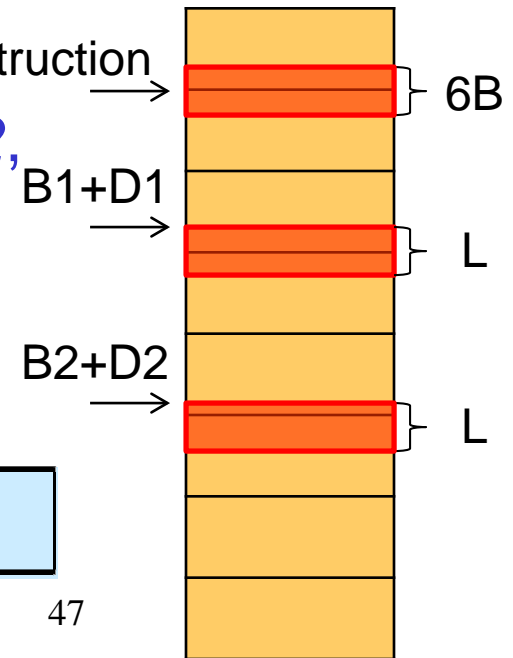
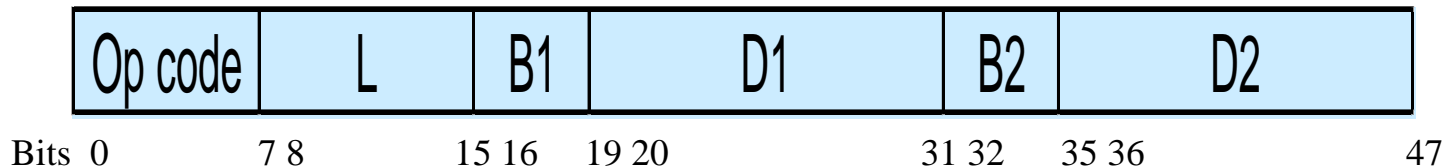


# Allocation of Frames

# Introduction

- Each process needs **minimum** number of frames
- E.g.: IBM 370 – **6 pages** to handle Storage to Storage MOVE instruction:

- Both operands are in main storage, the first operand is  $B1(Reg.ID)+D1$ , the second operand is  $B2(Reg.ID)+D2$ ,  $L$  plus 1 is the length.
- instruction is 6 bytes, may span 2 pages
- Moving content could across 2 pages



# Frame Allocation

## ■ Fixed allocation

- Equal allocation – 100 frames, 5 processes → 20 frames/process
- Proportional allocation – Allocate 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 for replacement one of its frames
  - ◆ select for replacement from a process with lower priority

# Frame Allocation

- **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 common used
  - A **minimum** number of frames must be maintained for each process to prevent **trashing**

# Review Slides ( II )

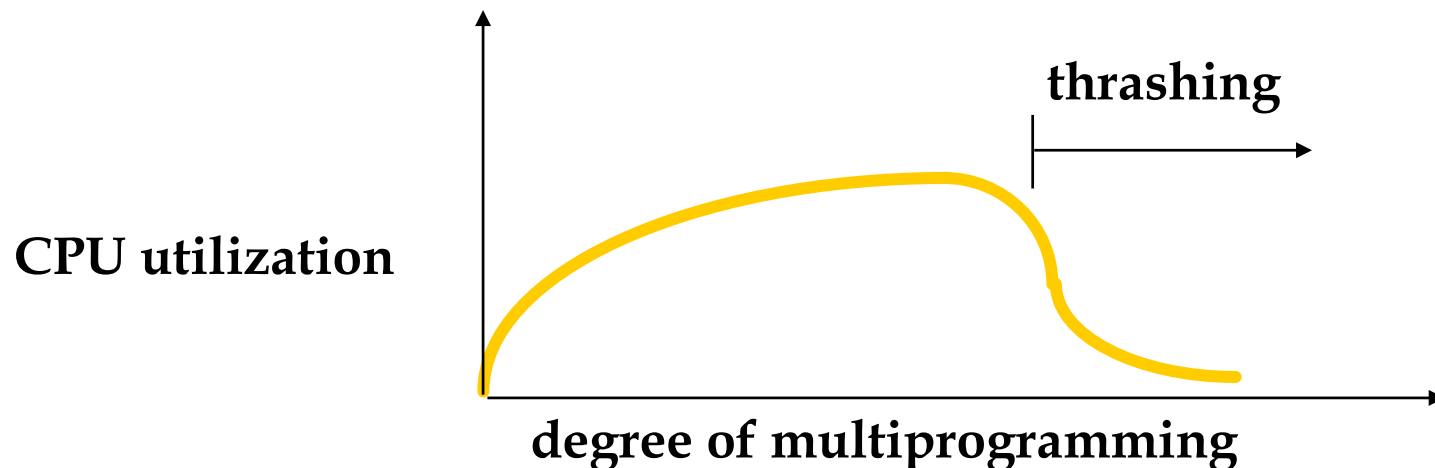
- Page replacement steps?
- Page replacement algorithm goal?
- Dirty bit usage?
- Belady's anomaly?
- FIFO? Optimal? LRU?
- Fixed vs. priority frame allocation?
- Global vs. local frame allocation?



# 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 multiprogramming**
  - ➔ 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

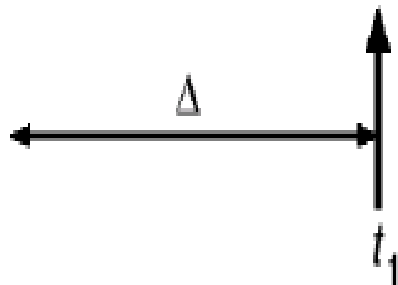
- **Locality**: a set of pages that are **actively used together**
- Locality model: as a process executes, it **moves from locality to locality**
  - program structure (subroutine, loop, stack)
  - data structure (array, table)
- **Working-set model** (based on locality model)
  - working-set **window**: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

# Working-Set Example

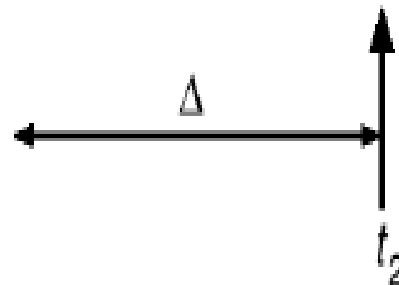
- If  $\Delta = 10$ :

page reference table

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



$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



$$WS(t_2) = \{3, 4\}$$

# Working-Set Model

## ■ Prevent thrashing using the working-set size

- $WSS_i$  : working-set size for process  $i$
- $D = \sum WSS_i$  (total demand frames)
- If  $D > m$  (available frames)  $\Rightarrow$  thrashing
- The OS monitors the  $WSS_i$  of each process and allocates to the process enough frames
  - ◆ if  $D \ll m$ , increase degree of MP
  - ◆ if  $D > m$ , suspend a process

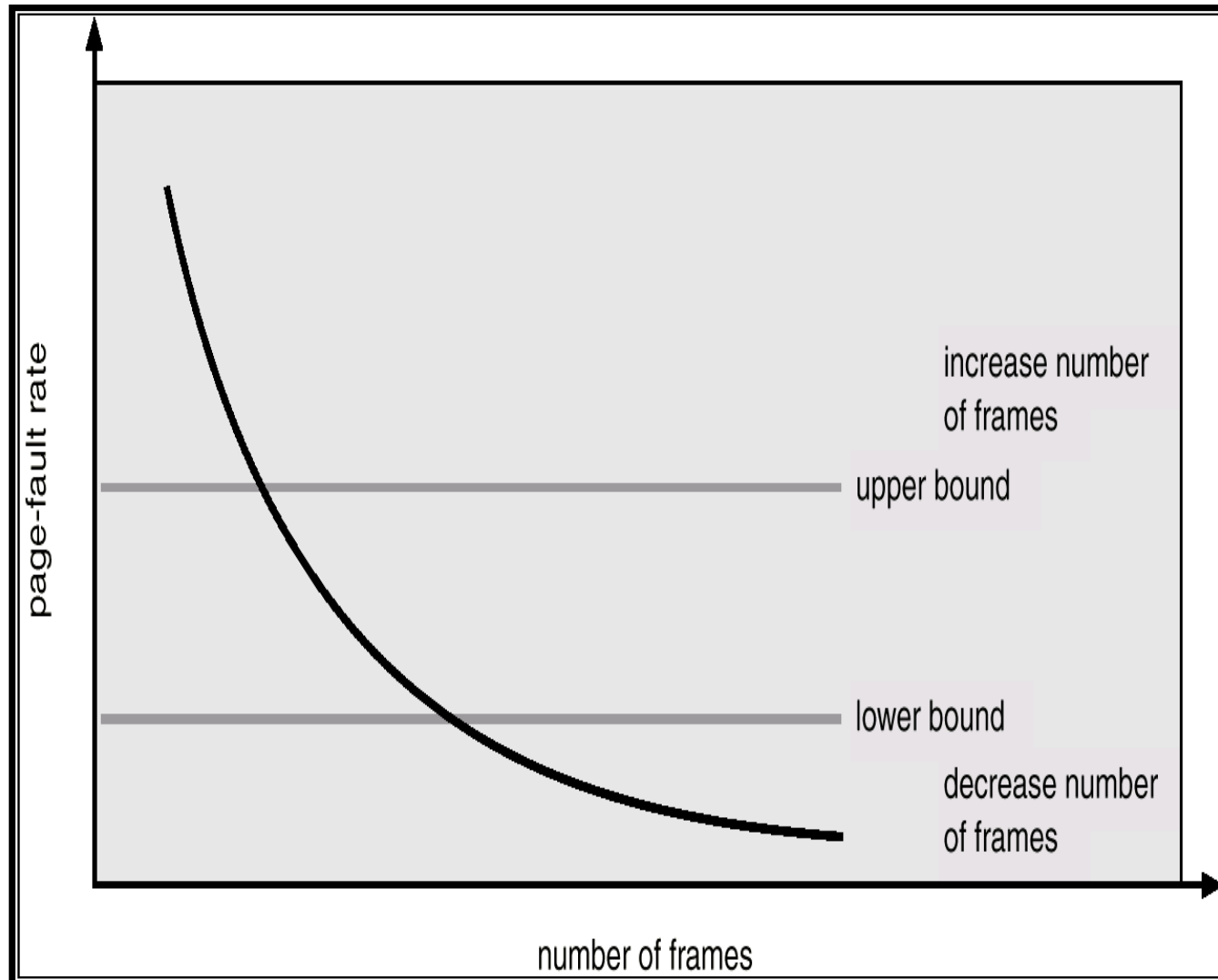
- ☺: 1. prevent thrashing while keeping the degree of multiprogramming as high as possible
2. optimize CPU utilization

☹ : too expensive for tracking

# Page Fault Frequency Scheme

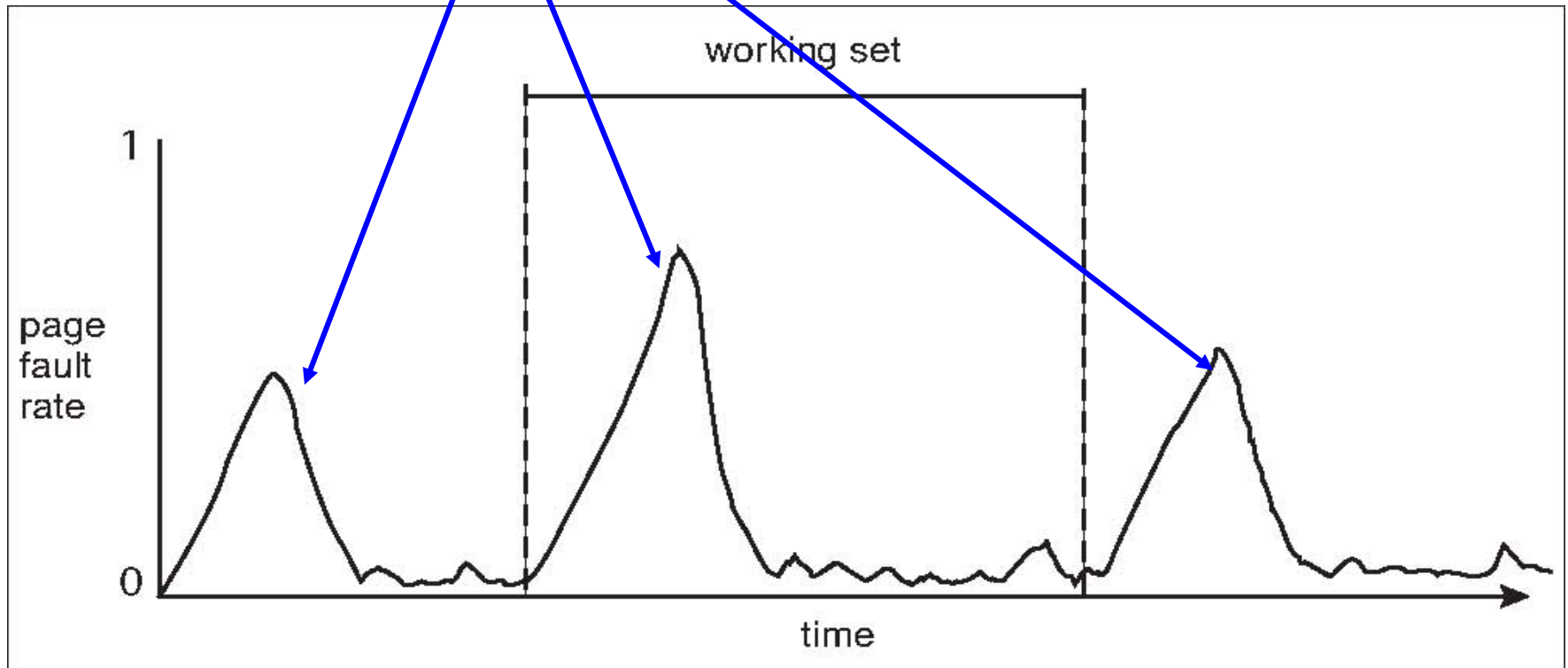
- **Page fault frequency** directly measures and controls the page-fault rate to prevent thrashing
  - Establish upper and lower bounds on the desired page-fault rate of a process
  - If page fault rate exceeds the upper limit
    - ◆ allocate another frame to the process
  - If page fault rate falls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

# Page Fault Frequency Scheme



# Working Sets and Page Fault Rates

peak of new locality



- Memory has locality property
- When the process moves to a new WS, the PF rate rises toward a peak

# Review Slides ( III )

- Thrashing definition?
- Process locality?
- When will thrashing happen? Solution?

# Problem Set

- 9.2: Consider the page table for a system with 12-bit virtual and physical addresses with 256-byte pages. The list of free page frames is D, E, F (that is, D is at the head of the list, E is second, and F is last). Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal.

(A dash for a page frame indicates that the page is not in memory.)

- 9EF
- 111
- 700
- 0FF

Page	Page Frame
0	-
1	2
2	C
3	A
4	-
5	4
6	3
7	-
8	B
9	0



# Problem Set

- 9.4: Consider a demand-paging system with the following time-measured utilizations: CPU utilization 20%, Paging disk 97.7%, Other IO devices 5%. For each of the following, say whether it will (or is likely to) improve CPU utilization. Explain your answers.
  - Install a faster CPU
  - Install a bigger paging disk
  - Increase the degree of multiprogramming
  - Decrease the degree of multiprogramming

# Problem Set

- 9.6: Consider a demand-paging system with a paging disk that has an average access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference if the page-table entry is in the associative memory. Assume that 80 percent of the accesses are in the associative memory and that, of those remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time?

# Problem Set

- 9.11: Discuss situations in which the most frequently used page-replacement algorithm generates fewer page faults than the least recently used page-replacement algorithm. Also discuss under what circumstances the opposite holds.
- 9.12: Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page faults occurs.
- .19,

# Problem Set

- 9.16: Consider the two-dimensional array A: `int A[][] = new int[100][100];` where `A[0][0]` is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in the page 0 (location 0 to 199). Thus every instruction fetch will be from page 0. For two page frames, how many page faults are generated by the following array-initialization loops, using LRU replacement and assuming that the first page frame contains the process and the other is initially empty?
  - `for (int j=0;j<100;j++) for (int i=0; i<100; i++) A[i][j] = 0;`
  - `for(int i=0; i<100; i++) for (int j=0; j<100; j++) A[i][j] = 0;`

# Problem Set

- 9.18: What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?
- 9.32: Is it possible for a process to have two working sets, one representing data and another representing code? Explain.



# Backup

# Windows NT

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned working-set minimums and working-set maximums
- WS minimum: the minimum # of pages the process is guaranteed to be in memory
- A process can have pages up to its WS maximum
- When the amount of free memory in the system falls below a threshold, automatic working set trimming is performed
- **Working set trimming** removes pages from processes that have pages in excess of their WS minimum