國立陽明交通大學
NATIONAL YANG MING CHIAO TUNG UNIVERSITY

Institute of Artificial Intelligence Innovation
Department of Computer Science

Operating System

Lecture 08: Virtual Memory Management

Shuo-Han Chen (陳碩漢),
shch@nycu.edu.tw

Thur. 13:20 - 16:20 ED305

Course Schedule

W	Date	Lecture	Online	Homework
1	Sept. 4	Lec00: Couse Overview & Historical Prospective		
2	Sept. 11	Lec01: Introduction	V	
3	Sept. 18	Lec02: OS Structure	V	HW01 Due 10/5
4	Sept. 25	Lec03: Processes Concept	X	
5	Oct. 2	Typhoon – No class	V	
6	Oct. 9	Lec07: Memory Management	V	
7	Oct. 16	Lec08: Virtual Memory Management	V	HW02 Due 11/2
8	Oct. 23	Lec04: Process Scheduling	V	
9	Oct. 30	School Midterm Exam		
10	Nov. 6	Lec05: Process Synchronization	V	HW03
11	Nov. 13	Lec06: Deadlocks	V	
12	Nov. 20	School Event – No class		
13	Nov. 27	Lec09: File System Interface	V	HW04
14	Dec. 4	Lec10: File System Implementation	V	
15	Dec. 11	Lec11: Mass Storage System & Lec12: IO Systems	V	
16	Dec. 18	School Final Exam		

Overview

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

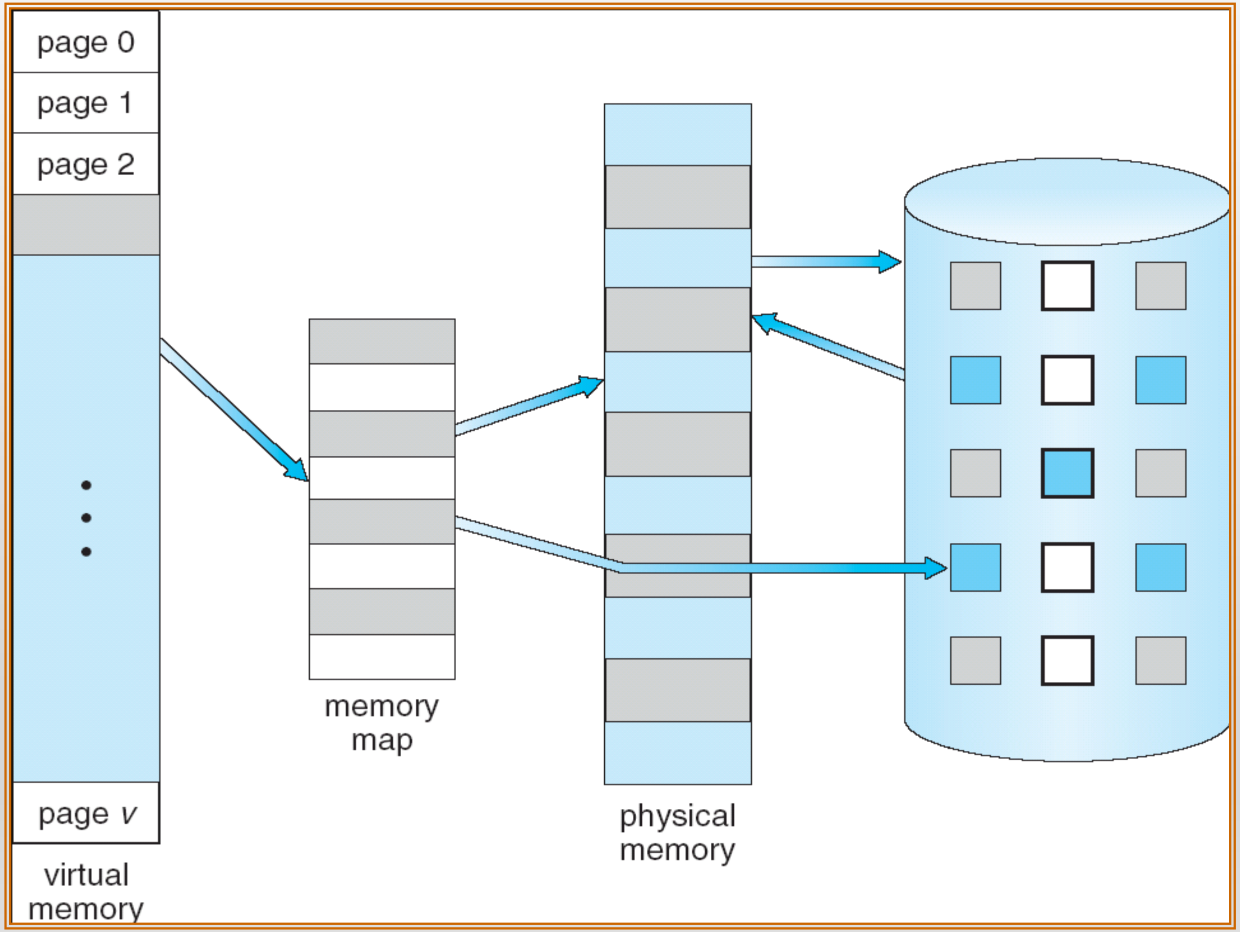
Background

- 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 a 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

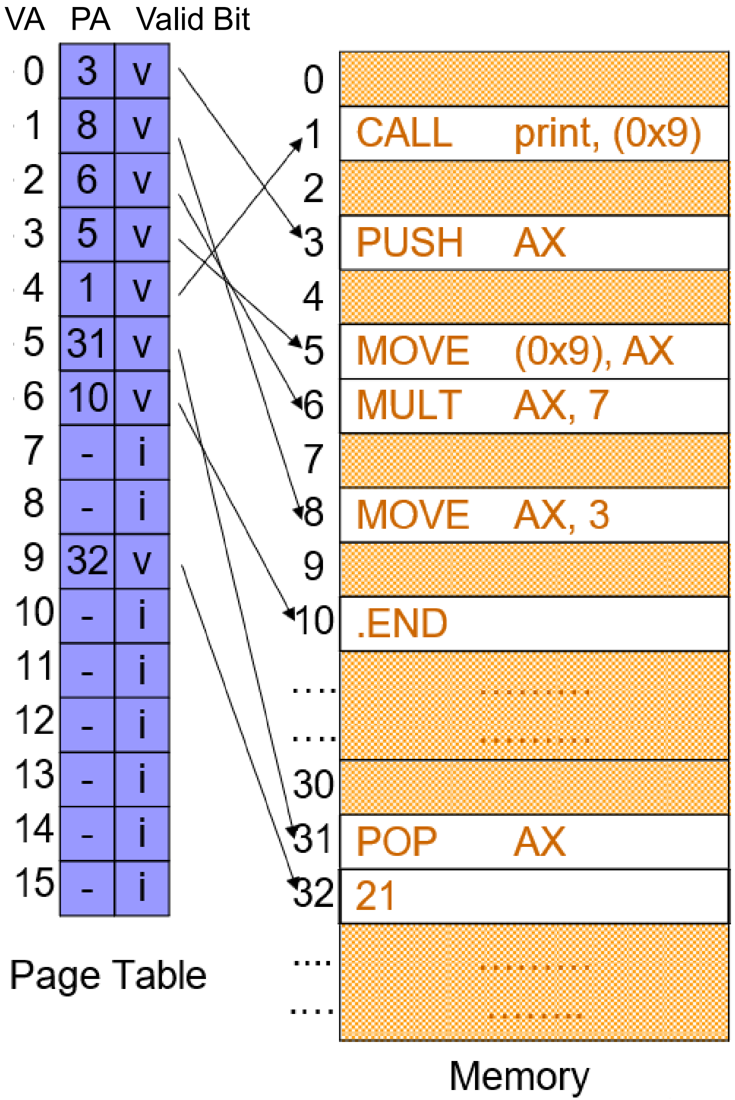
Virtual Memory vs. Physical Memory



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



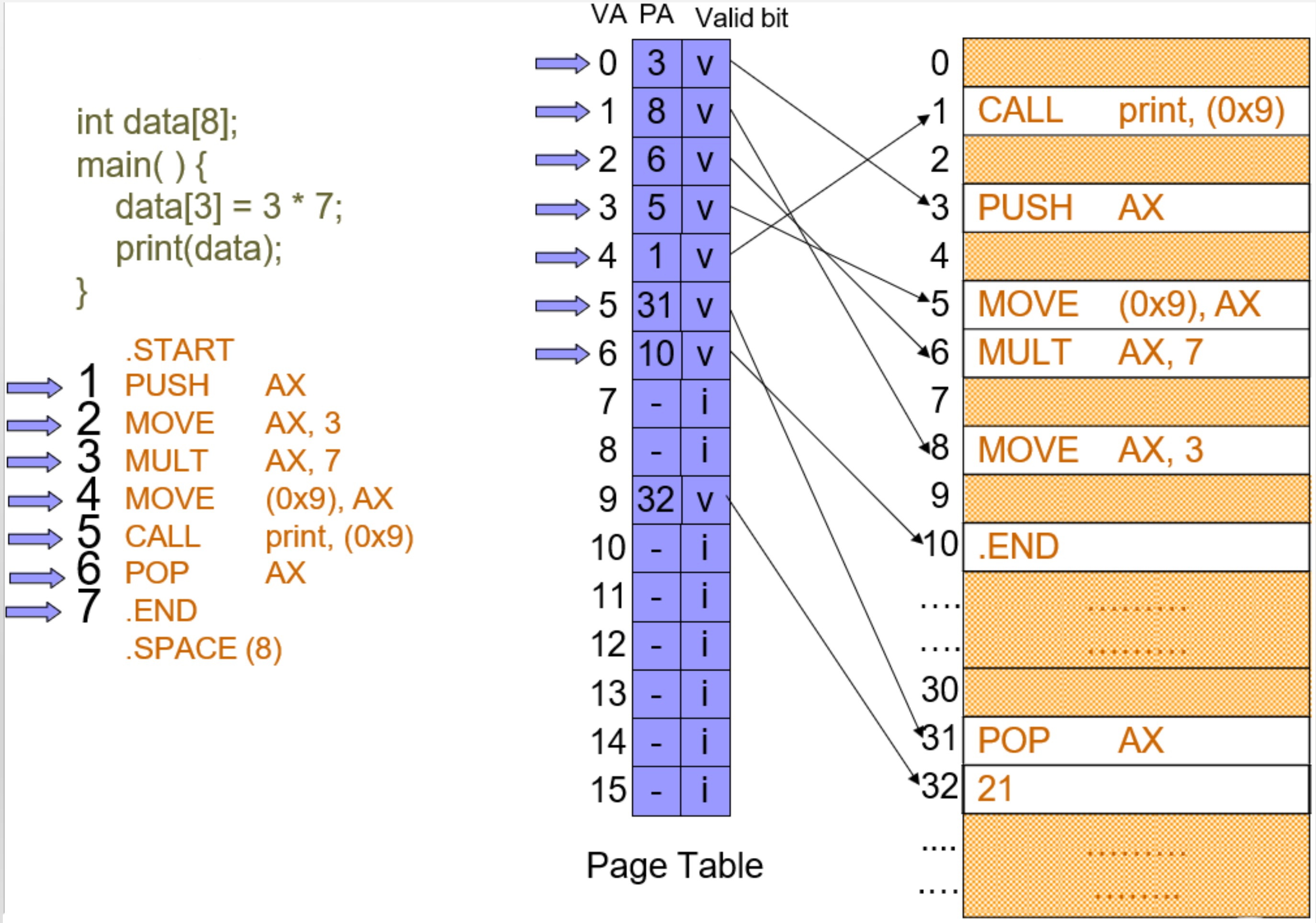
Demand Paging

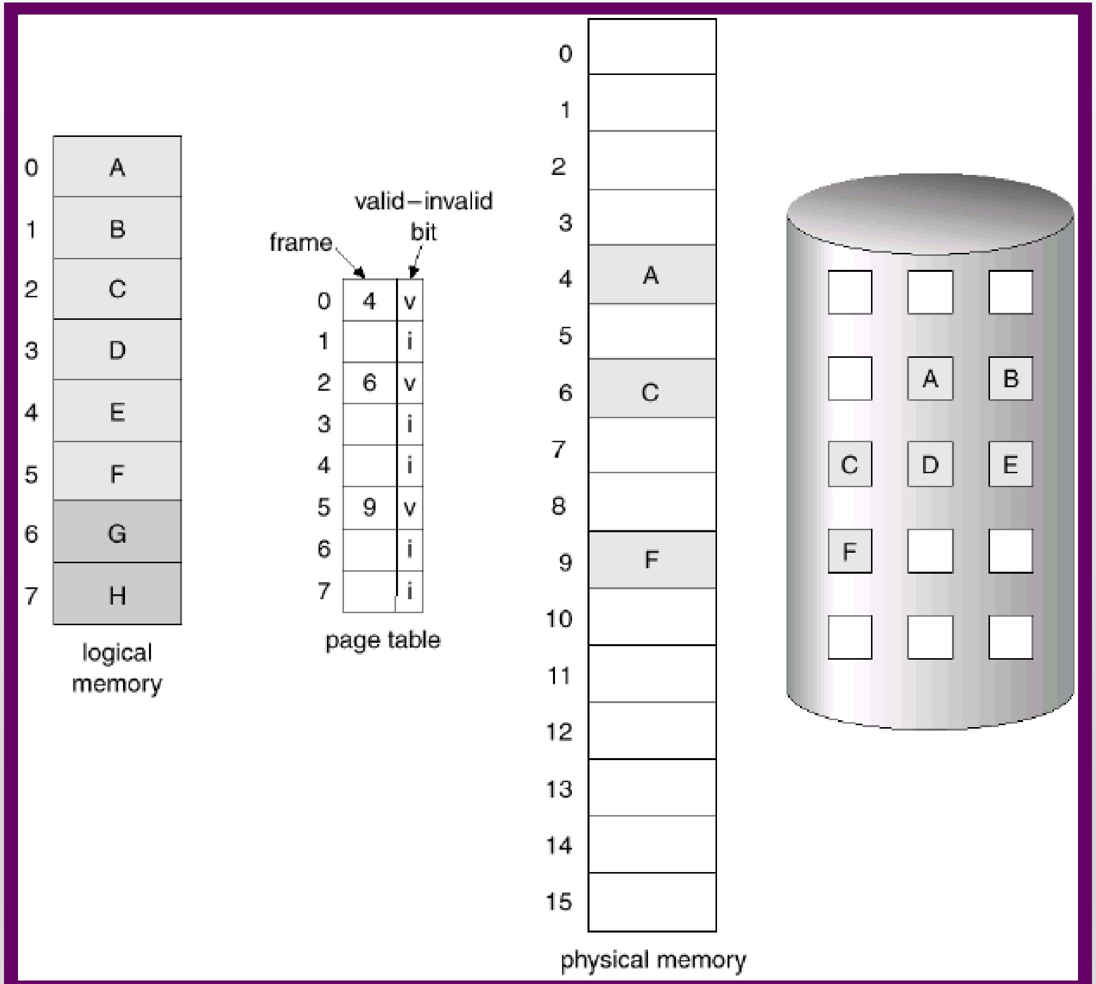
- A page rather than the whole process is brought into memory only when it is needed
  - Less I/O needed -> Faster response
  - Less memory needed -> More users
- Page is needed when there is a reference to the page
  - Invalid reference -> abort
  - Not-in-memory -> bring to memory via paging
- pure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

Demand Paging

- 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



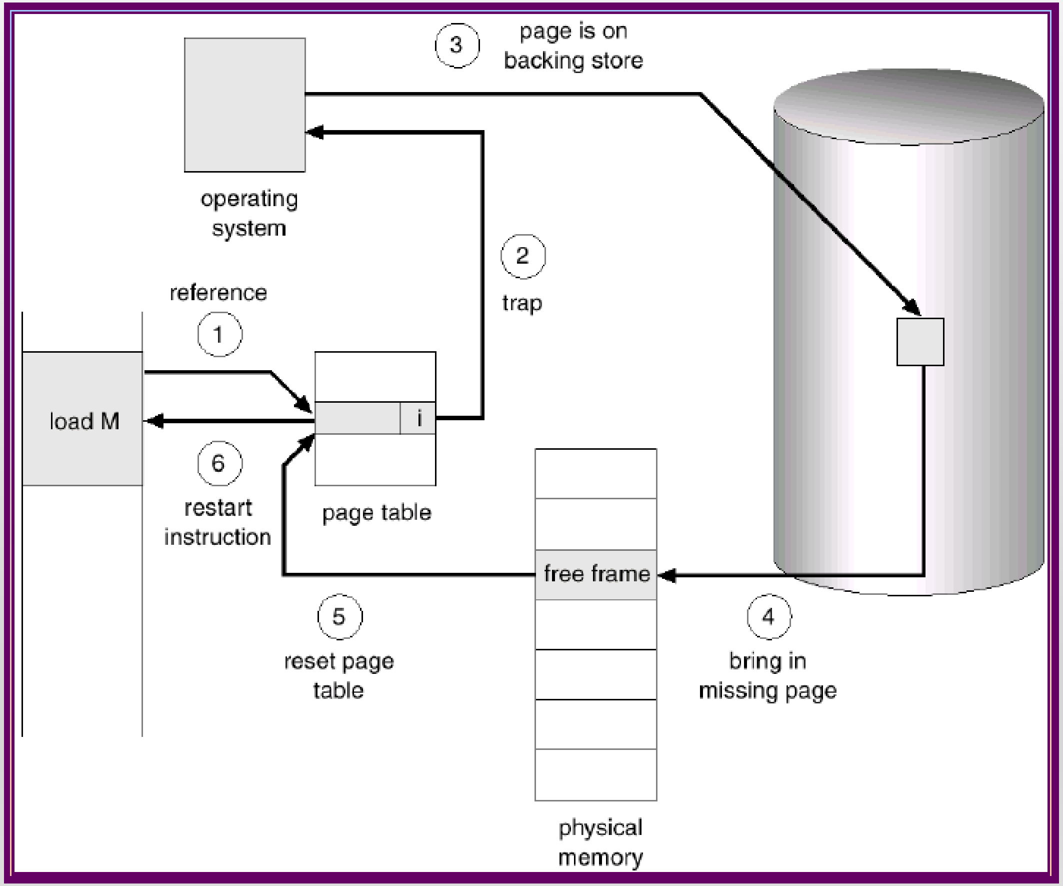


0
A
1
B
2
C
3
D
4
E
5
F
6
G
7
H
logical memory
valid–invalid bit
frame
0 4 v
1 i
2 6 v
3 i
4 i
5 9 v
6 i
7 i
page table
0
1
2
3
4 A
5
6 C
7
8
9 F
10
11
12
13
14
15
physical memory
A
B
C
D
E
F

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 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) x ma + p x pft
  - P: page fault rate, ma : mem. access time, pft : page fault time
- Example: ma = 200ns, pft = 8ms
  - EAT = (1 - p) * 200ns + p * 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 (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 1st and 3rd can be reduced to several hundred instructions
  - The page switch time is close to 8ms

Process & Virtual Memory

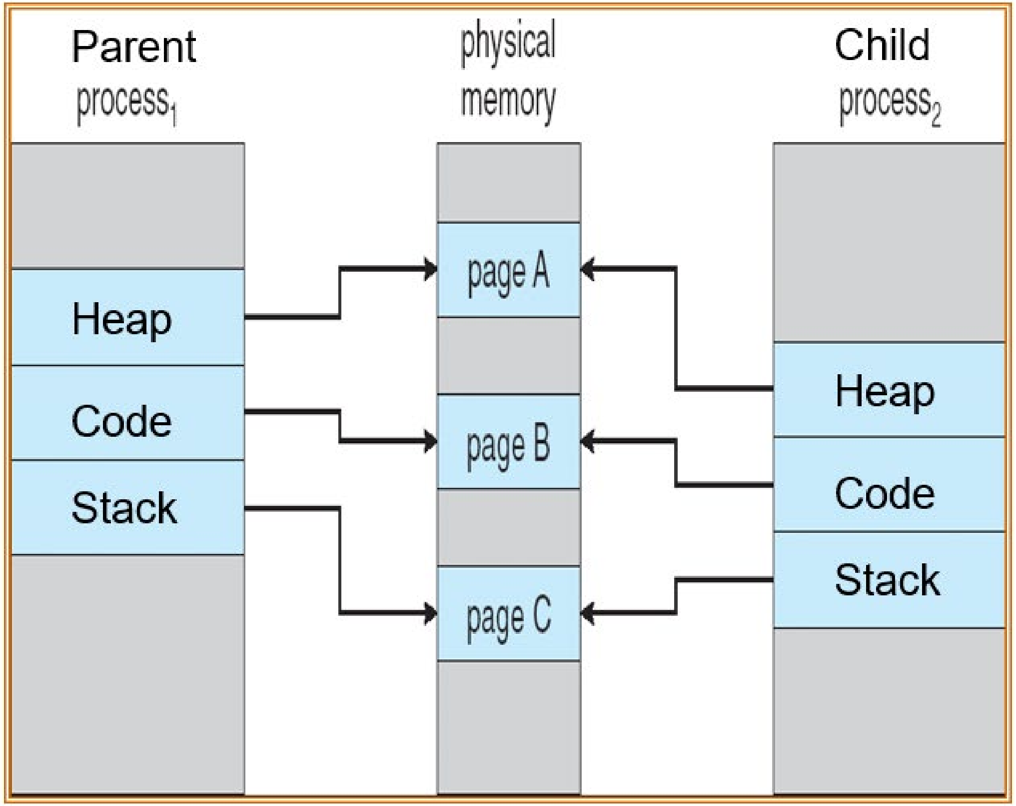
- Demand Paging: only bring in the page containing the first instruction
- Copy-on-Write: the parent and the child process share the same frames initially, and frame-copy when a page is written
- Memory-Mapped File: map a file into the virtual address space to bypass file system calls (e.g., read(), 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

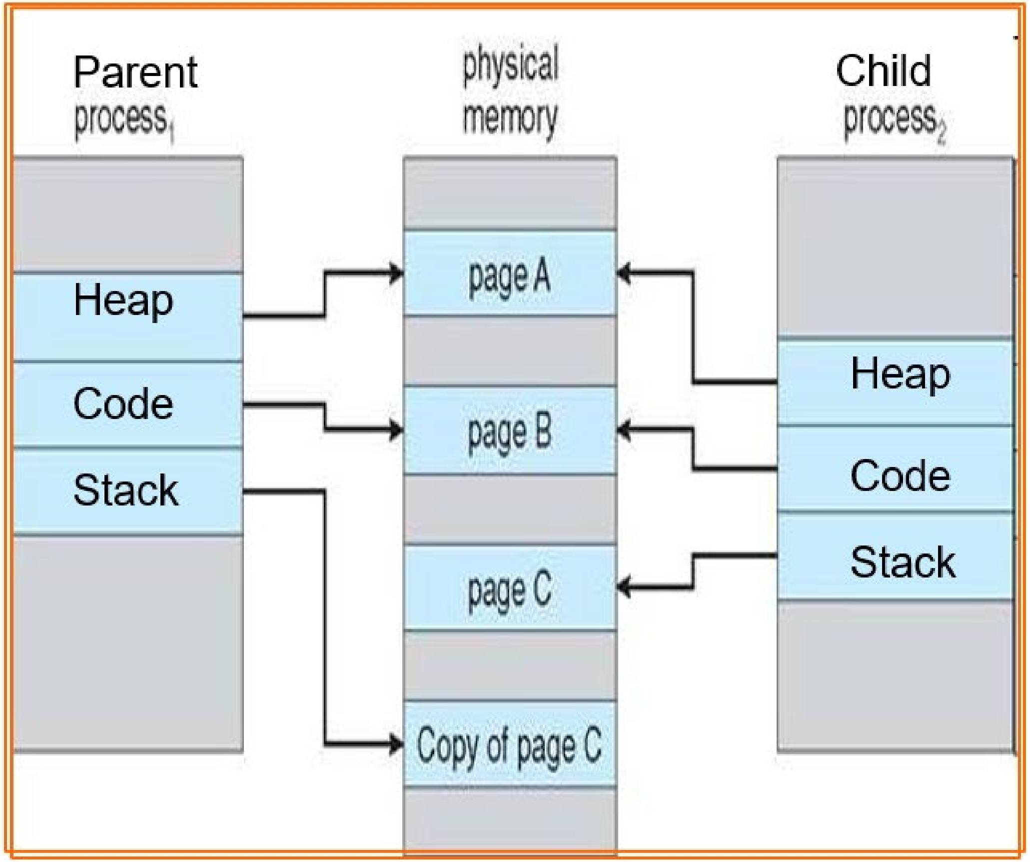
When a child process is forked

```
#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

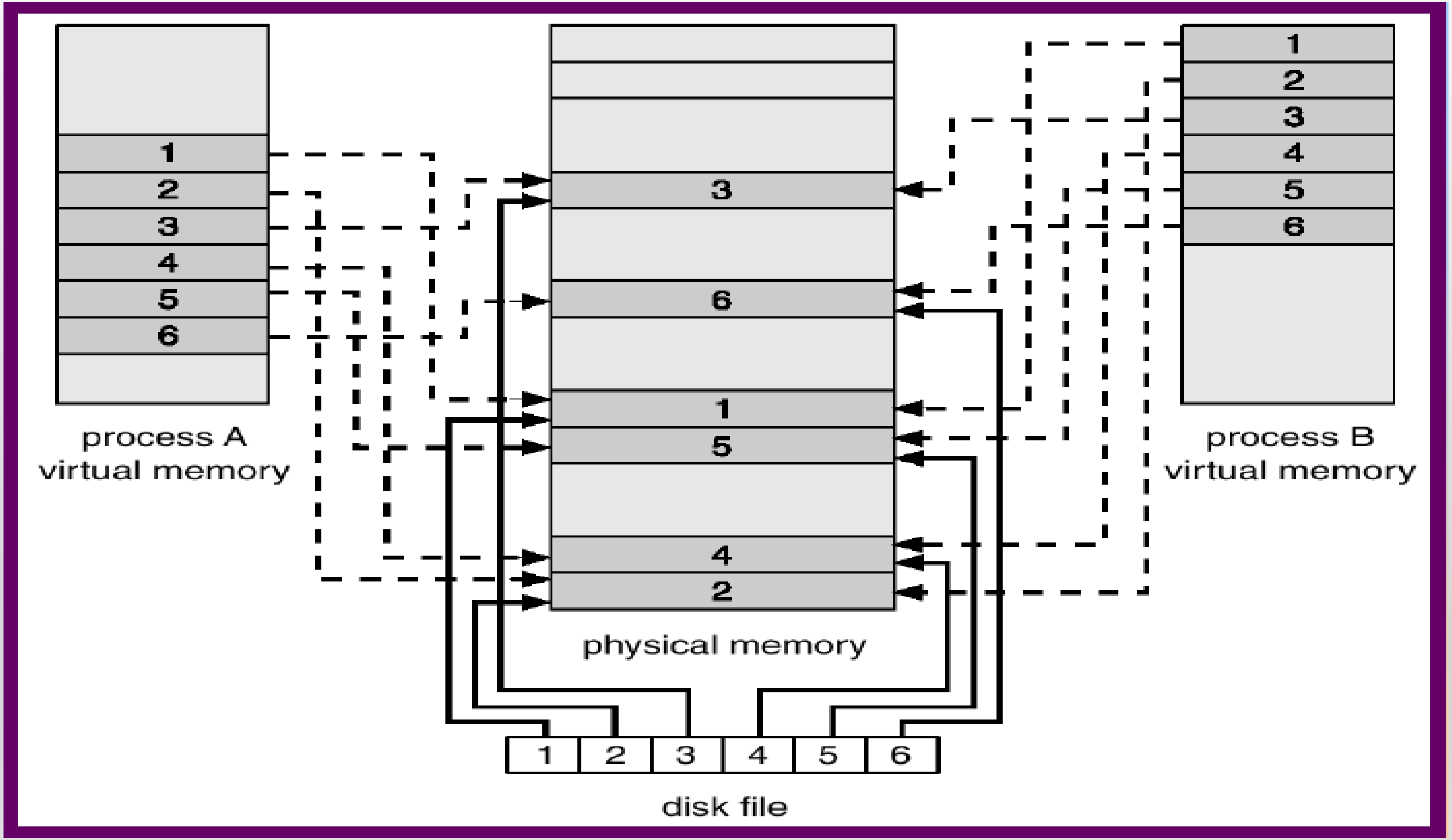
```
#include <stdio.h>
void main( )
{
   int A;
   /* fork child process */
   A = fork( );

   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, data lost, more programming efforts

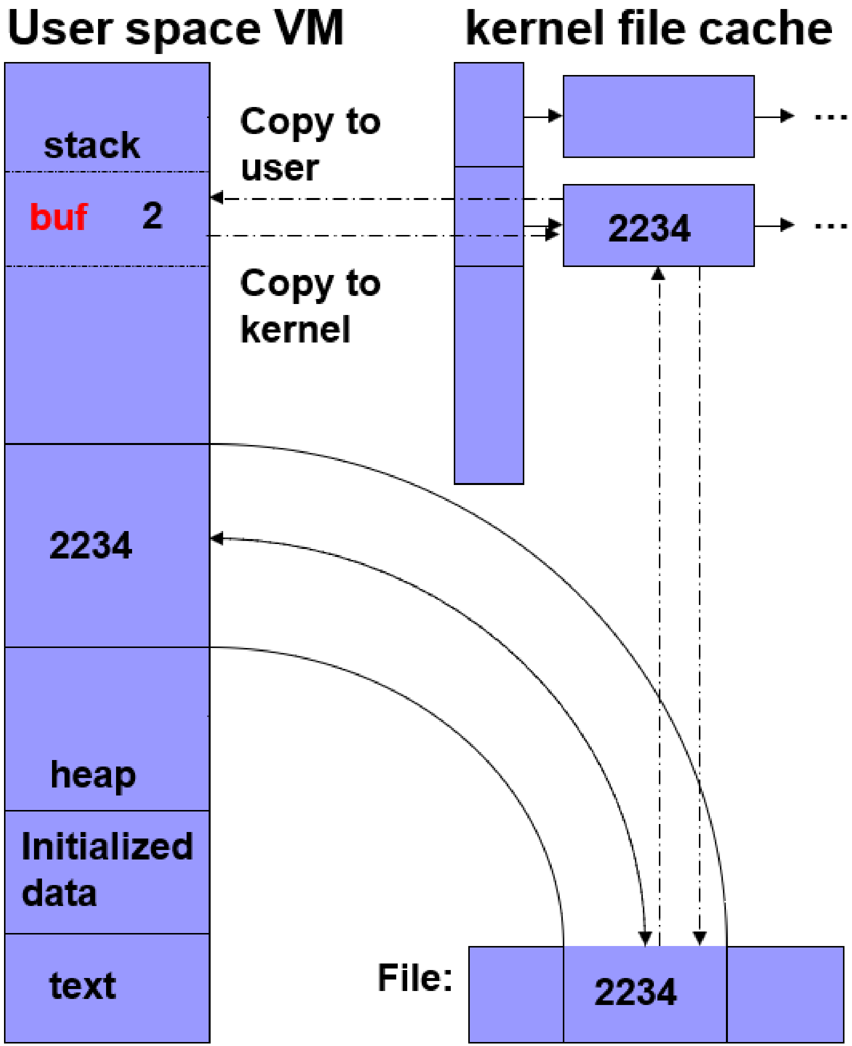
Memory-Mapped File Example



```
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?

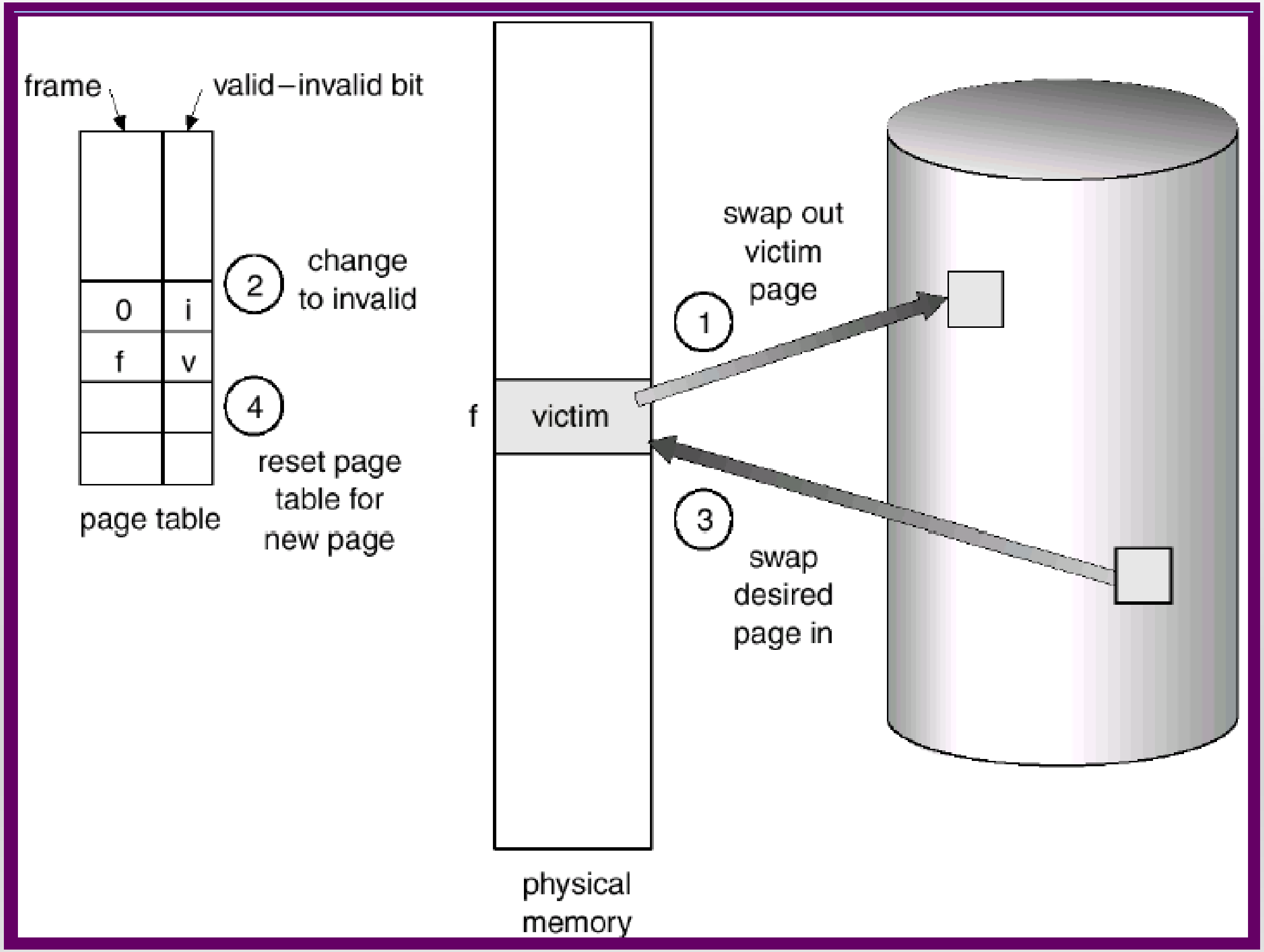
Page Replacement Concept

- 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

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

  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

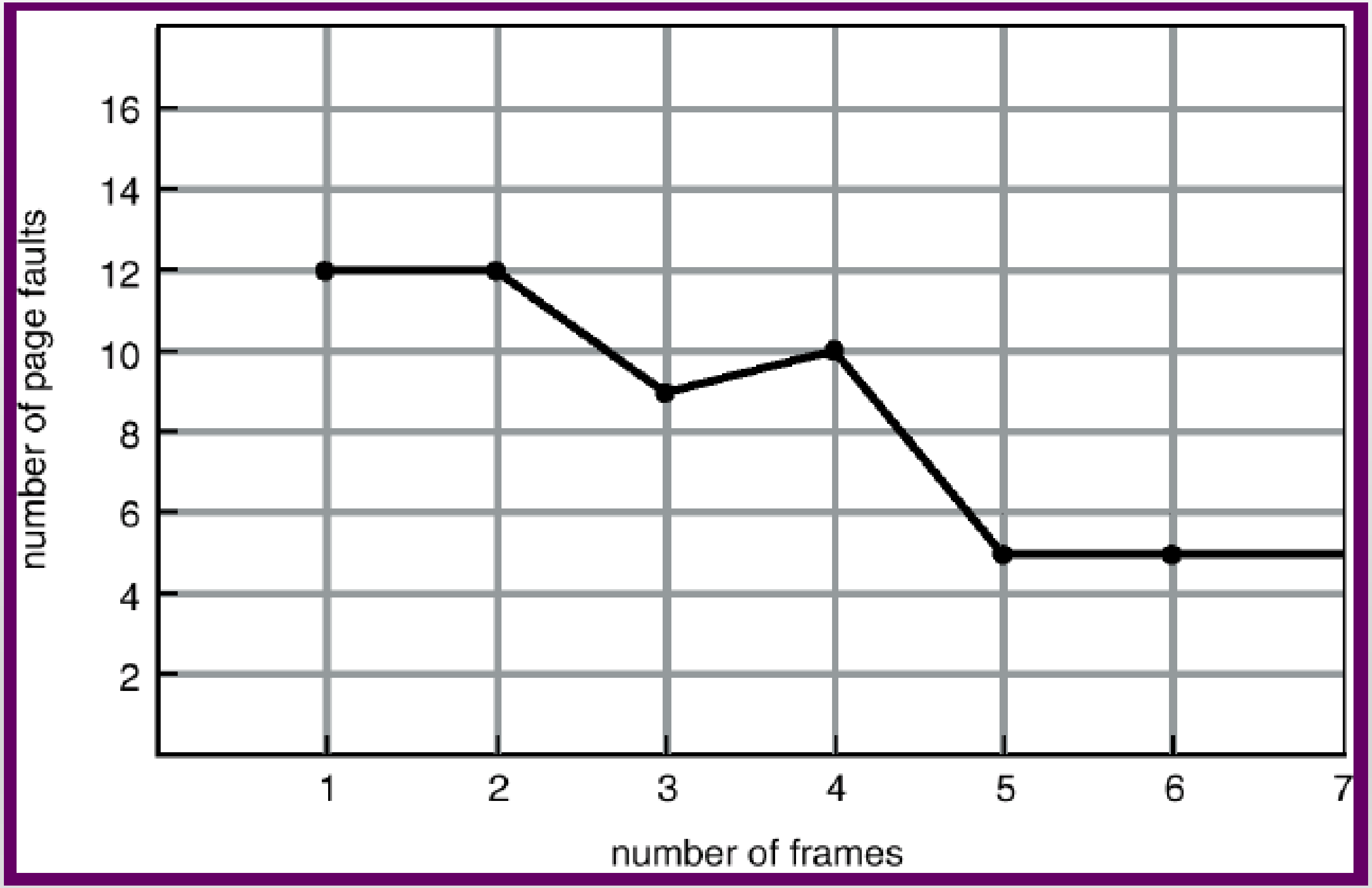
	1	2	3	4	1	2	5	1	2	3	4	5
head (new)	1	2	3	4	1	2	5	5	5	3	4	4
		1	2	3	4	1	2	2	2	5	3	3
tail (old)			1	2	3	4	1	1	1	2	5	5

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

	1	2	3	4	1	2	5	1	2	3	4	5
head → (new)	1	2	3	4	4	4	5	1	2	3	4	5
		1	2	3	3	3	4	5	1	2	3	4
			1	2	2	2	3	4	5	1	2	3
tail → (old)				1	1	1	2	3	4	5	1	2

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

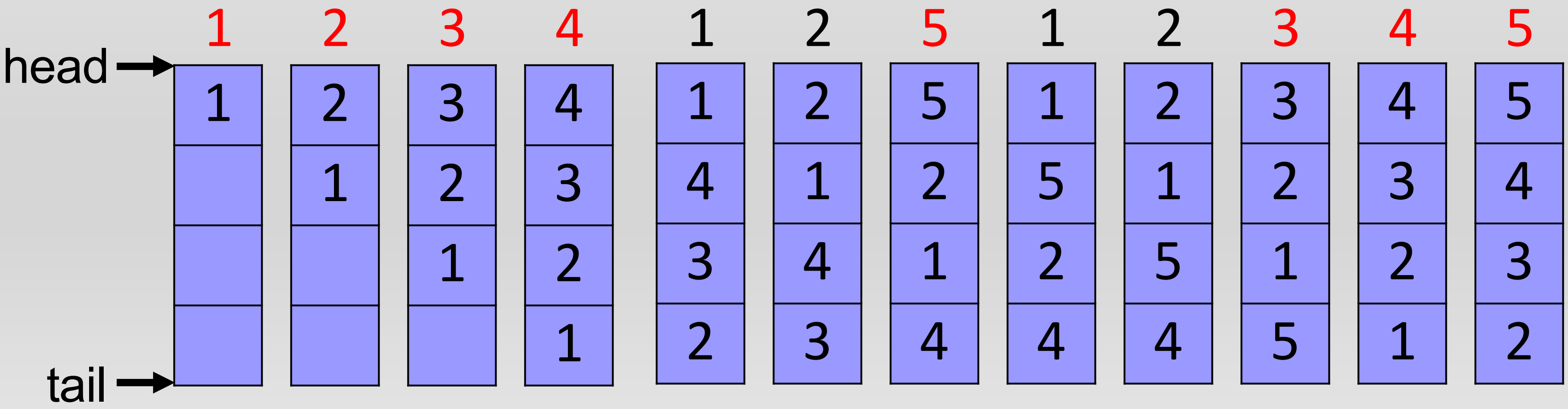
	1	2	3	4	1	2	5	1	2	3	4	5
head →	1	1	1	1	1	1	1	1	1	1	4	4
		2	2	2	2	2	2	2	2	2	2	2
			3	3	3	3	3	3	3	3	3	3
tail →				4	4	4	5	5	5	5	5	5

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

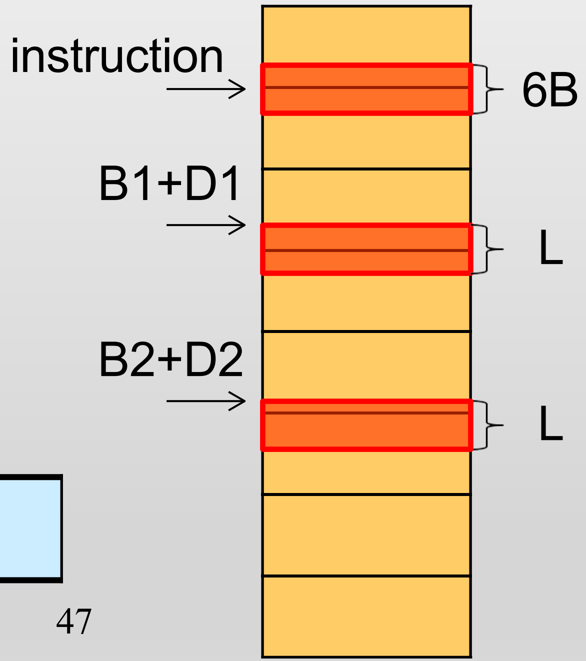
- Few systems provide sufficient hardware support for the LRU page-replacement
- additional-reference-bits algorithm
- second-chance algorithm
- enhanced second-chance algorithm

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

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



opcode	L	B1	D1	B2	D2

Bits 0 7 8 15 16 19 20 31 32 35 36 47

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

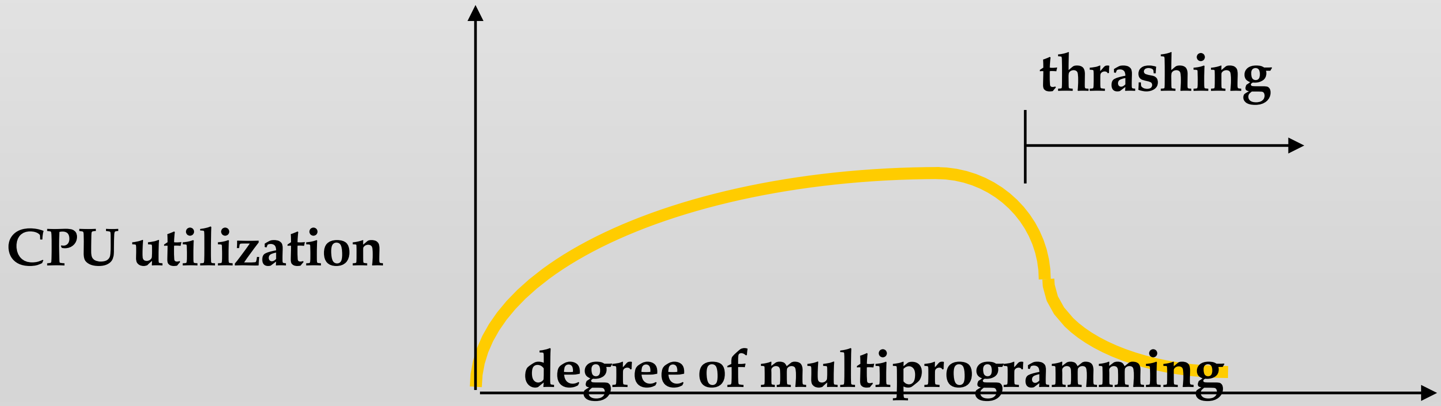
- Local allocation: each process select from its own set of allocated frames
- Global allocation: process selects a replacement frame from the set of all frames
  - one process can take away a frame of another process
  - e.g., allow a high-priority process to take frames from a low-priority process
  - 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?
- Place replacement algorithm goal?
- Dirty bit usage?
- Belady’s anomaly?
- FIFO? Optimal? LRU?
- Fixed vs. priority frame allocation?
- Global vs. local frame allocation?

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

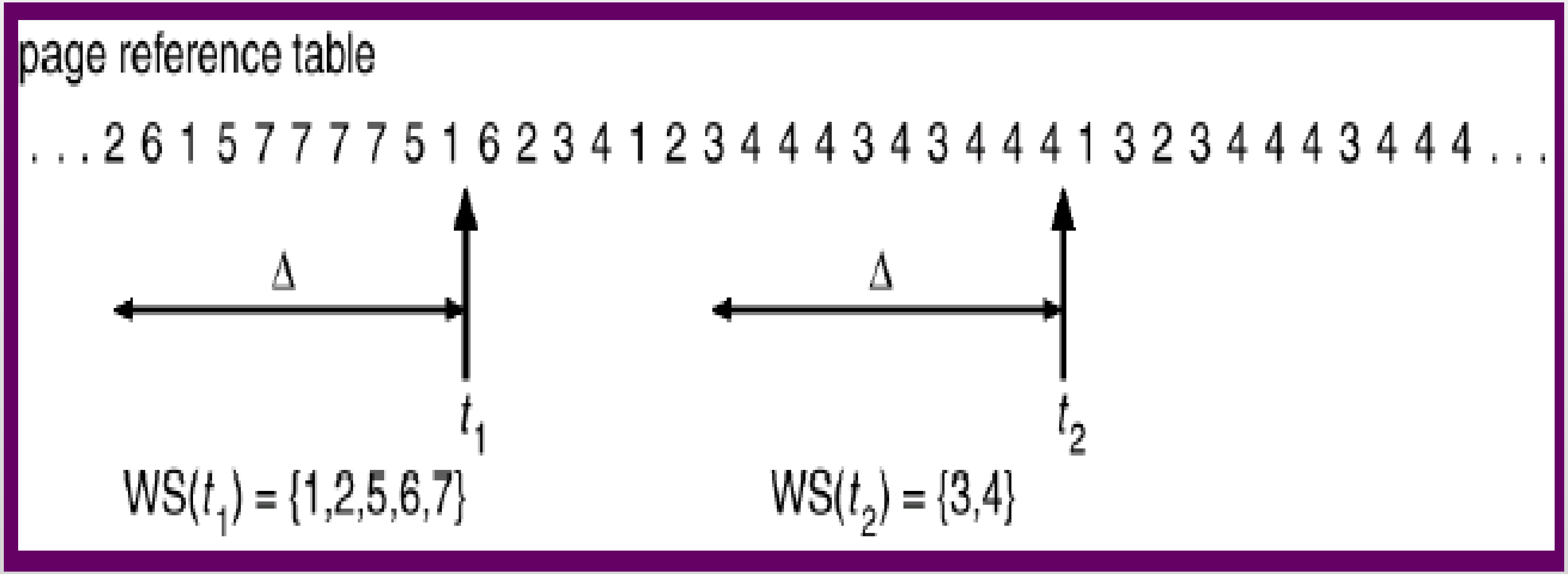
- Performance problem caused by thrashing (Assume global replacement is used)
  - processes queued for I/O to swap (page fault)
  - -> low CPU utilization
  - -> OS increases the degree of multiprogramming
  - -> new processes take frames from old processes
  - -> more page faults and thus more I/O
  - -> 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 Δ (delta)
    - working set: set of pages in most recent page Δ references (an approximation locality)

Working-Set Example

- If Δ = 10:



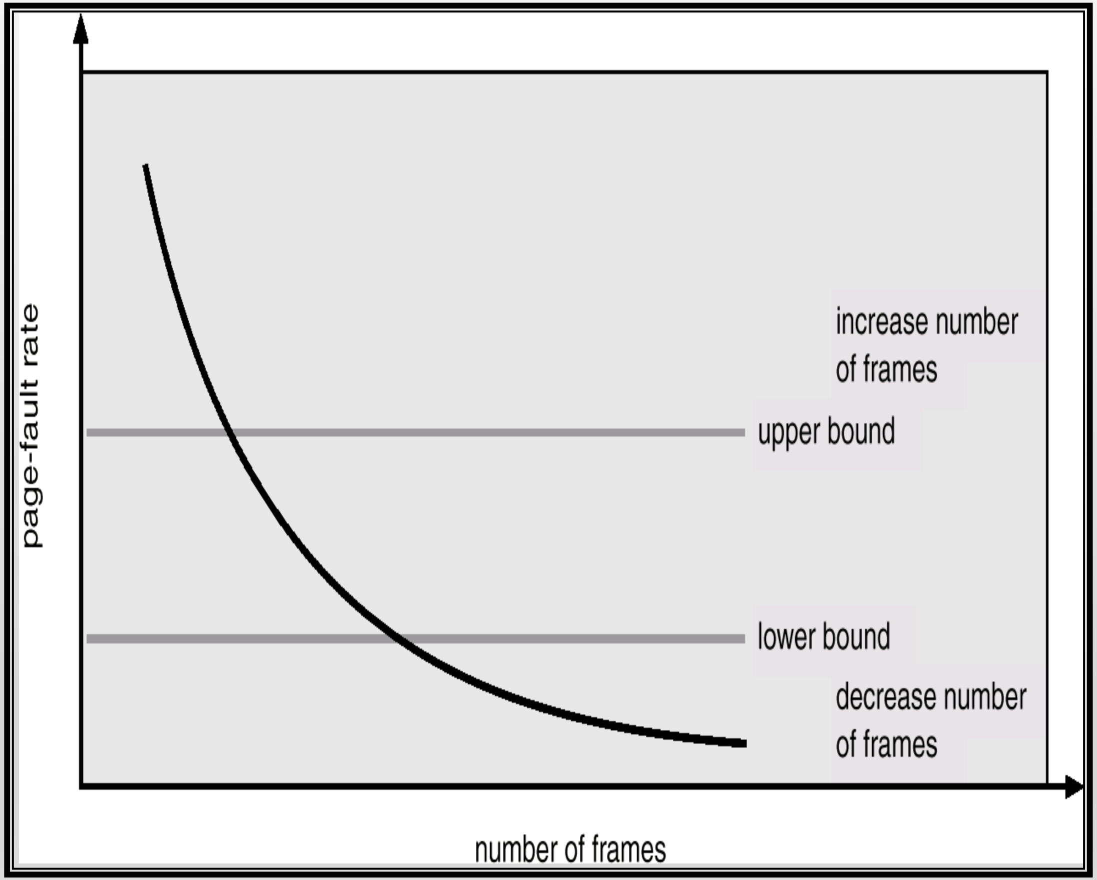
Working-Set Model

- Prevent thrashing using the working-set size
  - WSSi : working-set size for process i
  - D = $\sum$ WSSi (total demand frames)
  - If D > m (available frames) -> thrashing
  - The OS monitors the WSSi of each process and allocates to the process enough frames
    - if D << m, increase degree of MP
    - if D > m, suspend a process
- Good :
  1. prevent thrashing while keeping the degree of multiprogramming as high as possible
  2. optimize CPU utilization
- Bad : too expensive for tracking

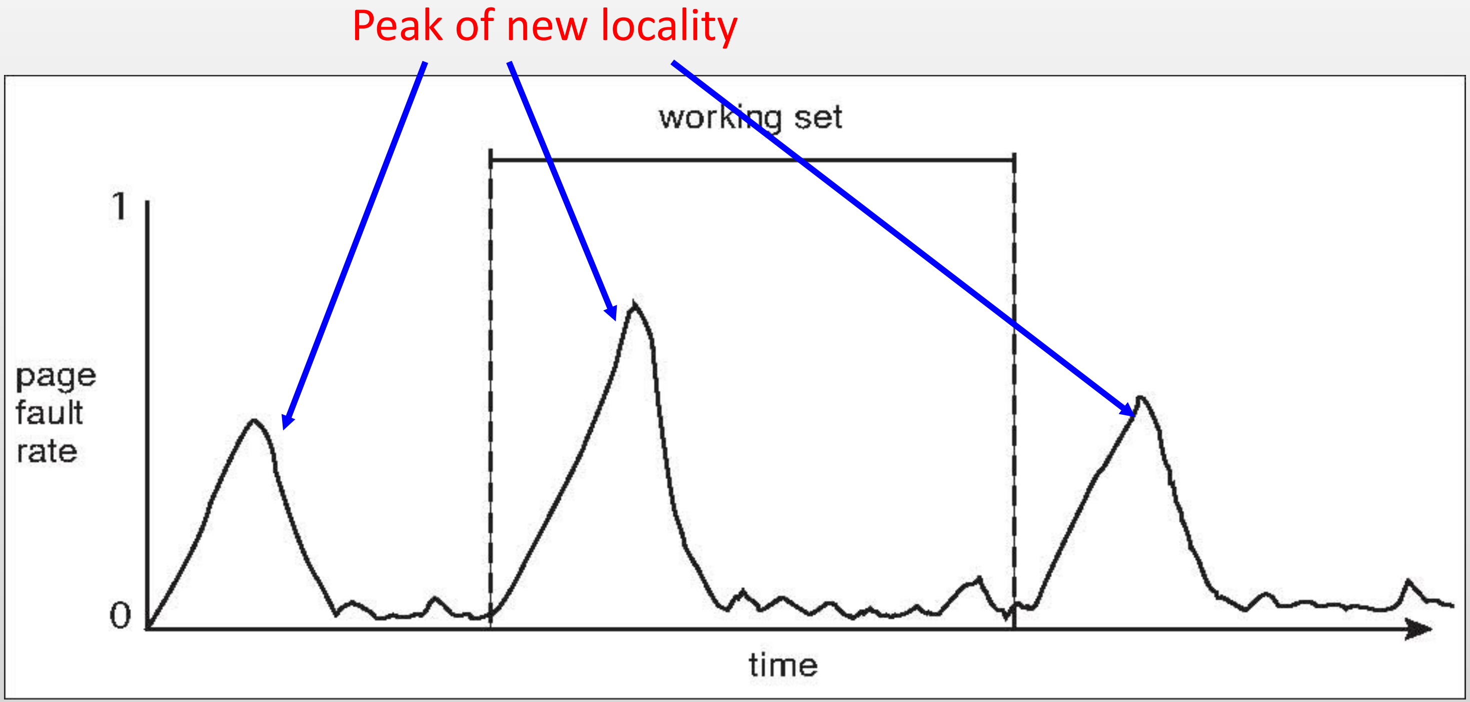
Page Fault Frequency Scheme

- 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



- 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?

Reading Material & HW

- Chap 9
- Problems
  - 9.2, 9.4, 9.6, 9.8, 9.9, 9.12, 9.14, 9.17, 9.19, 9.21

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

Q & A

Thank you for your attention