

# 8장. 가상 메모리

---

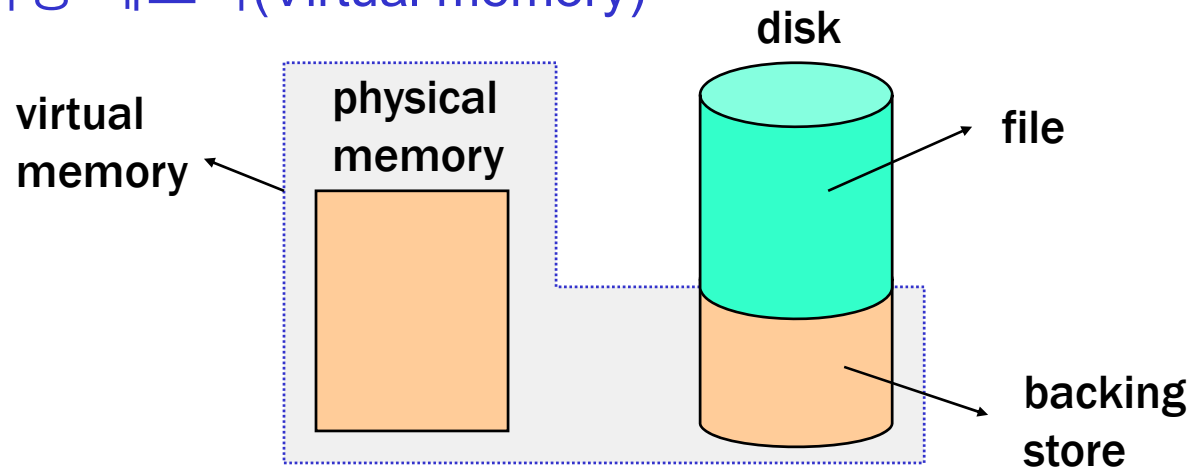
# 목표

---

- 가상 메모리의 이점(benefits) 소개
- 요구 페이징, 페이지 교체 정책, 페이지 프레임 할당 개념 설명
- working-set model의 원리 논의
- 공유 메모리와 memory mapped file 관계 탐구

## 8.1 배경 지식

- 실행되는 instruction은 물리 메모리에 있어야 함
- 이 요구조건의 충족 방법은?
  1. 프로그램의 전체 논리 주소 공간을 물리 메모리에 적재 후 실행
  2. 동적 적재(dynamic loading) – programmer's work
  3. 가상 메모리 사용 – 실행에 필요한 부분만 물리 메모리에 위치
- 가상 메모리(Virtual memory)



# 가상메모리(Virtual memory)

---

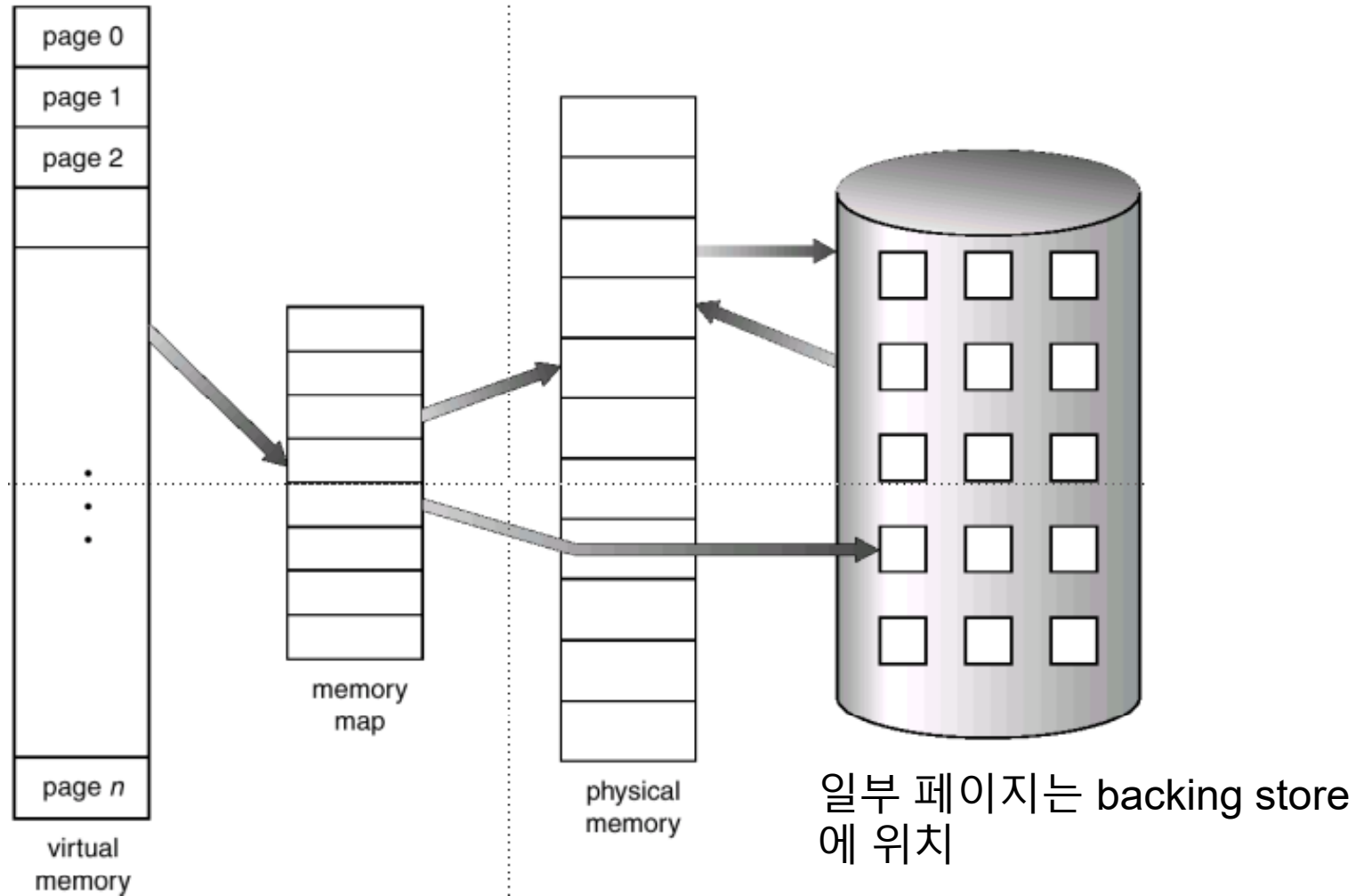
## ■ Virtual memory

- 프로그램의 논리 메모리와 물리적 메모리를 분리
- 프로세스가 완전히 메모리에 적재되지 않아도 프로세스 실행 허용
- 물리적 메모리보다 크기가 큰 가상 메모리 제공 가능  
(virtual memory size  $\leq$  physical memory size + backing store size)
- 프로그램 크기가 물리적 메모리보다 클 수 있음
- 페이지 단위의 swap-in/swap-out  $\rightarrow$  입출력 크기 감소

## ■ Virtual memory의 구현 방법

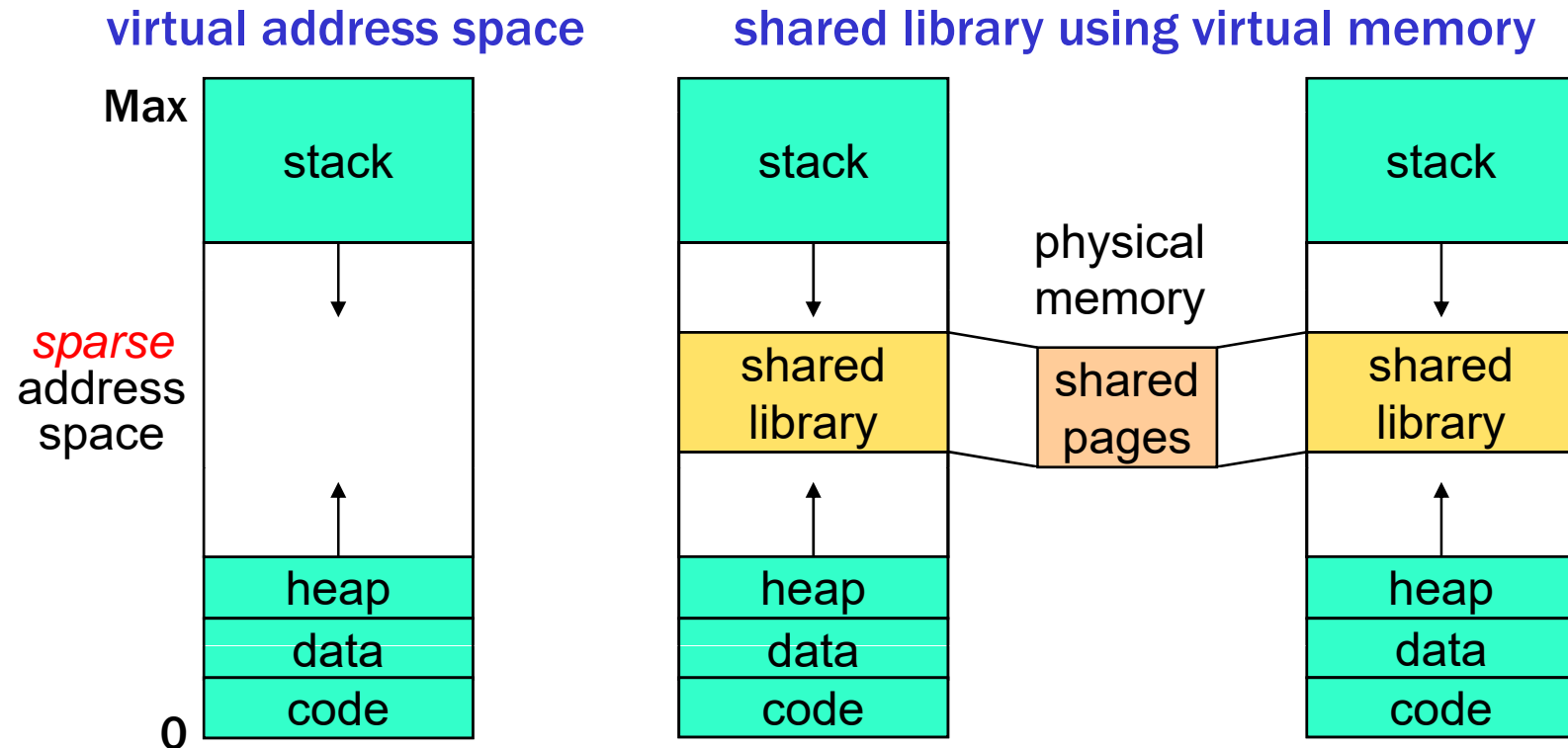
- 요구 페이징(Demand paging)
- 요구 세그먼테이션(Demand segmentation)  $\rightarrow$  가변 크기 때문에 더 복잡

# 물리적 메모리보다 큰 가상 메모리



# 가상 메모리와 메모리 공유

- 가상 메모리는 프로세스들이 메모리와 파일 공유를 가능하게 함
  - 공유 라이브러리(shared library)
  - 공유 메모리(shared memory)
  - fork()에 의한 프로세스 생성 동안 부모와 자식프로세스가 페이지 공유  
→ 효율적인 프로세스 생성



## 8.2 요구 페이징(Demand Paging)

---

### ■ 요구 페이징(Demand Paging)

- page가 필요할 때 page를 메모리로 가져옴 – 필요한 page들만 적재  
(cf) swapping – 프로세스를 실행할 때에 프로세스 전체 공간을 메모리에 적재하고, 메모리 공간을 비워줄 때에 전체 공간을 디스크로 이동함
- 요구 페이징을 수행하는 프로그램을 **lazy swapper** 또는 **pager**라고 부름

### ■ 요구 페이징의 장점

- 입출력 감소 → swap 시간 감소
- 적은 메모리 사용 → 메모리 공간 절약
- 빠른 응답 시간
- 더 많은 사용자 허용

# 요구 페이징 구현 - Valid Bit 사용

- 필요사항 – valid bit를 사용한 page table, 보조기억장치

- paging 하드웨어와 같음

- valid bit – 각 page table entry에 포함

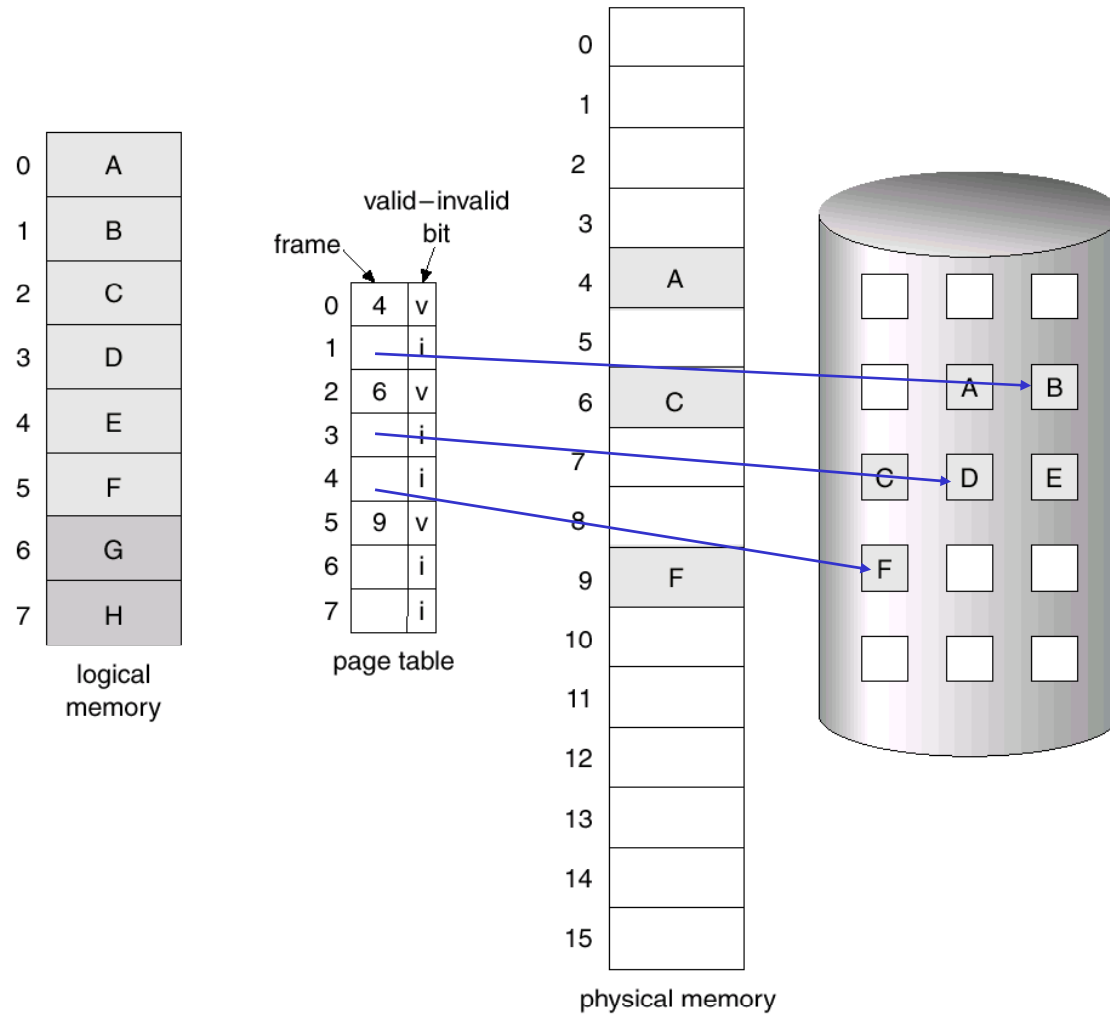
- 1 ⇒ in memory
- 0 ⇒ not in memory
  - (1) 디스크에 있음 (valid), 또는 (2) 유효하지 않음
- 초기값: 모든 valid bit = 0
  - page가 메모리에 적재될 때에 1로 설정

frame no.	V
page frame no.	1
page frame no.	1
...	
page # on disk	0
----	0

- Demand paging 구현

- paging table을 사용한 주소 변환 과정에서 (hardware 동작)
  - if valid bit = 1 (메모리에 적재된 유효한 페이지) → 메모리 참조
  - if valid bit = 0 (메모리에 없음) → page fault trap 발생
- page fault trap 처리 (운영체제 trap handler)
  - 메모리에 없지만, 디스크에 있으면 → 해당 page를 메모리로 적재 후 재실행
  - 유효하지 않은 참조(invalid reference) → 프로세스 중단(abort)

# 메모리에 적재되지 않은 페이지를 갖는 page table



# 페이지 부재(Page Fault) 처리

## ■ Page fault 처리 과정

1. page table의 해당 entry를 조사하여 invalid 원인을 알아냄
  - Invalid reference → abort.
  - not in memory, but on the disk → page it in.
2. free(empty) frame을 찾음
3. 디스크에 있는 page를 page frame으로 swap in → disk read 동작스케줄
4. disk read가 완료되면 page table 갱신
  - 적재된 page의 page table entry에서 valid bit = 1로 설정
5. page fault trap에 의해 중단되었던 instruction을 재시작

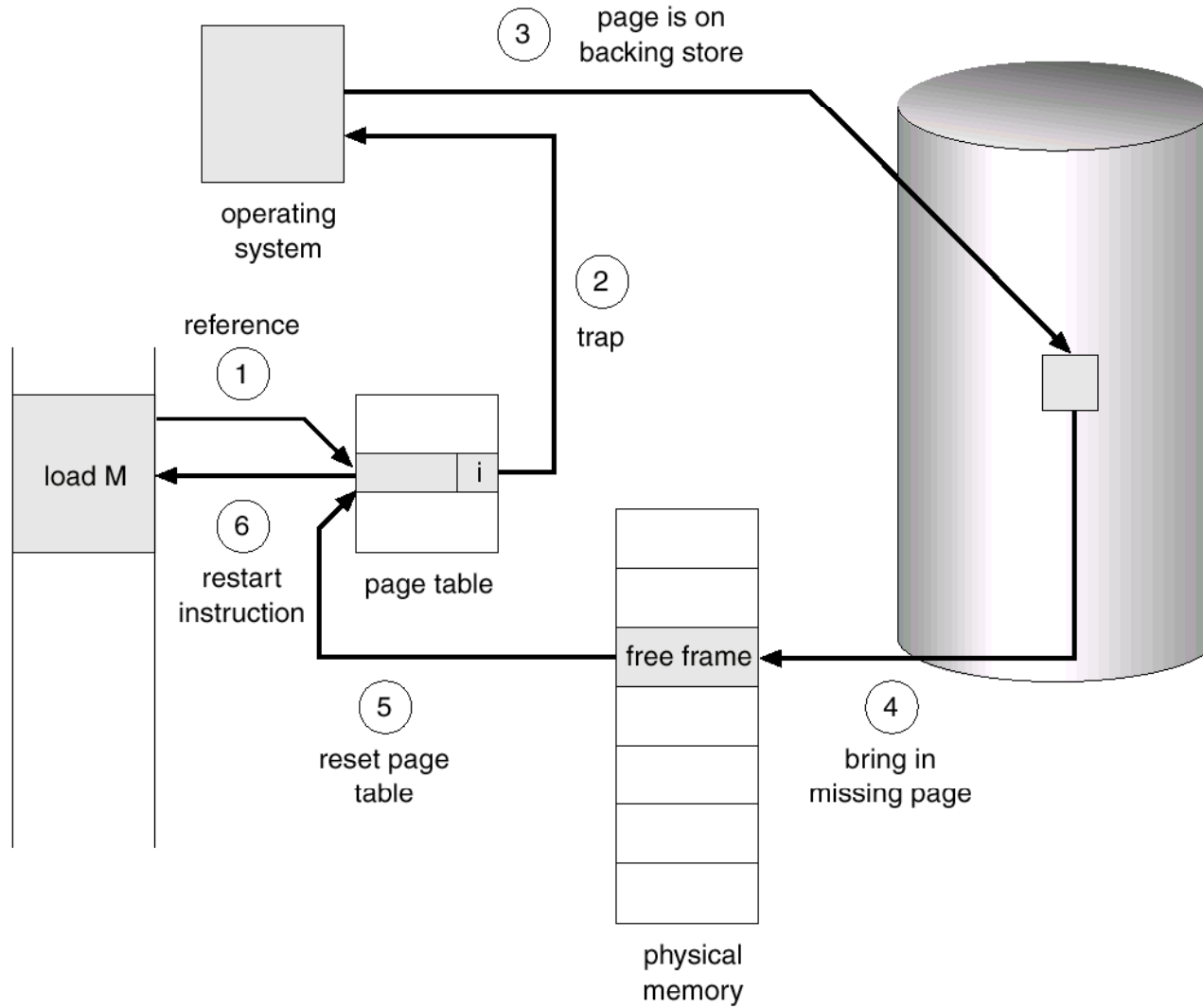
## ■ 순수 요구 페이징(Pure demand paging)

- 요구 받기 전에는 page를 메모리에 적재하지 않음 – 적재하지 않은 상태에서 프로그램 실행을 시작함
- 프로그램이 처음 실행될 때에 모든 페이지에 대해 page fault가 발생함

## ■ 메모리 참조의 지역성(Locality of reference) 존재

- 요구 페이징을 사용해도 만족할만한(reasonable) 성능을 보임

# Page Fault 처리 과정



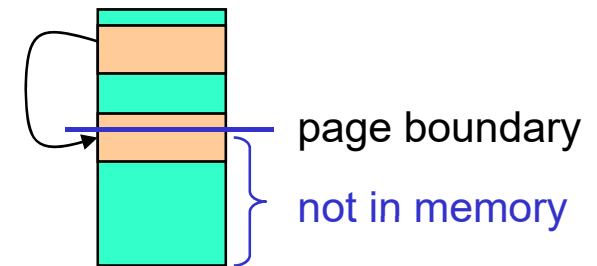
# Page Fault 처리 후 instruction 재실행

## ■ 처리하기 쉬운 예

- 3주소 명령어 (예) add A, B, C
  - 과정: 명령어 인출 → A인출 → B인출 → 덧셈 → C에 저장

## ■ 처리하기 어려운 예

- block move instruction
  - page fault trap이 발생하면 메모리에 trap 발생 이전의 메모리 내용을 복원해야 함 - overwrite 후에는 복원 불가능
  - 사전에 page fault 발생 가능성을 확인 → 발생이 가능하면 page fault trap 발생
- auto increment/decrement mode operation (예) MOV (r2)+, -(r3)
  - 명령어를 재 시작하기 전에 수정된 r2, r3값을 원래 값으로 복원



두 영역이 겹침

# Demand Paging의 성능

- 페이지 부재율(Page Fault Rate):  $p$  ( $0 \leq p \leq 1.0$ )
  - if  $p = 0$ , no page faults
  - if  $p = 1$ , every reference is a fault

$p$ 가 0에 근접하도록 해야 함
- Effective Access Time (EAT)
  - $EAT = (1 - p) \times [\text{memory access}] + p \times [\text{page-fault time}]$
  - page-fault time = [page-fault trap 처리]+[swap page in]+[process재시작]
- (예) memory access time=200ns, 평균 page-fault time=8ms
  - $EAT = (1-p) \times 200 + p \times 8,000,000 = 200 + 7,999,800 \times p$
  - $p=0.001 \rightarrow EAT = 8,200 = 8.2\text{ms}$
  - 성능 10% 미만으로 저하( $EAT < 220$ )시키는 page fault rate ?  
 $EAT < 220 \rightarrow 7,999,800 \times p < 20 \rightarrow p < 0.0000025 = 0.00025\%$
  - 페이지 부재율  $p$ 를 낮게 유지하는 것이 중요.  
그렇지 않으면 EAT는 급격히 증가할 수 있음

## 8.3 쓰기 시 복사(Copy-on-Write)

---

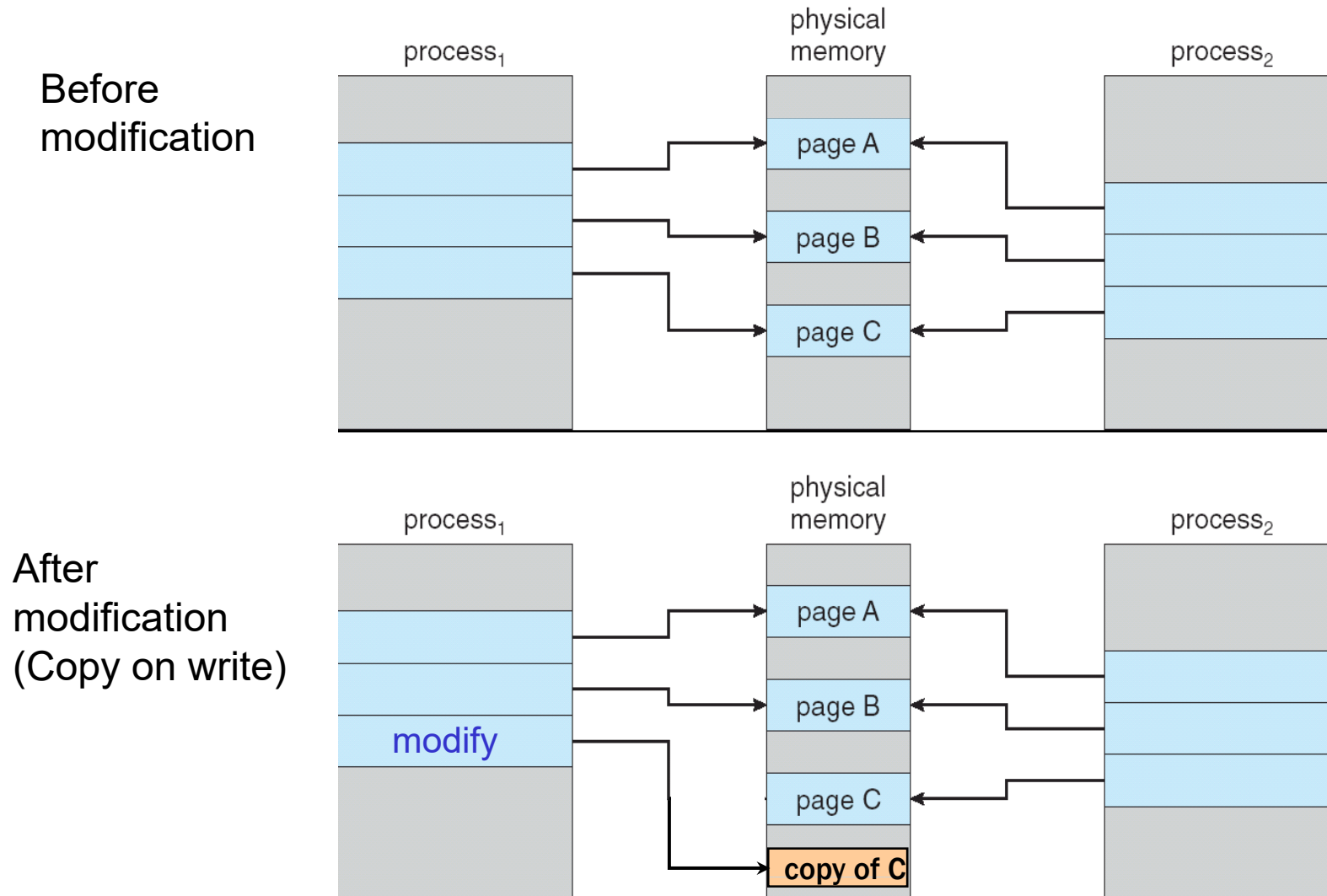
### ■ Copy-on-write (COW)

- fork()를 사용한 프로세스 생성시에 child process는 parent process와 같은 page 공유함 – page table 내용 복사, page는 복사하지 않음
  - exec()를 호출할 때 복사한 page가 쓸모 없게 되는 문제를 해결
- 공유 페이지를 수정(write)한다면, 수정되는 page만 복사한 후에 수정함 (다음 슬라이드 그림)
- 수정된 페이지만 복사되므로 효율적인 process 생성

### ■ Zero-fill-on-demand

- free page frame을 할당할 때에 0으로 채워 이전 내용을 지운 후 할당함
- stack 또는 heap이 확장되거나, copy-on-write를 수행할 때에 free page frame 할당 시에 사용

# Copy-on-Write



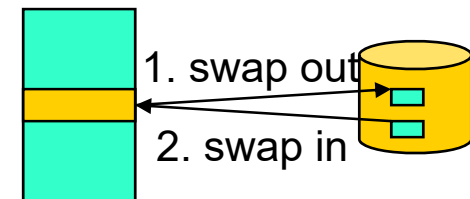
## 8.4 페이지 교체(Page Replacement)

---

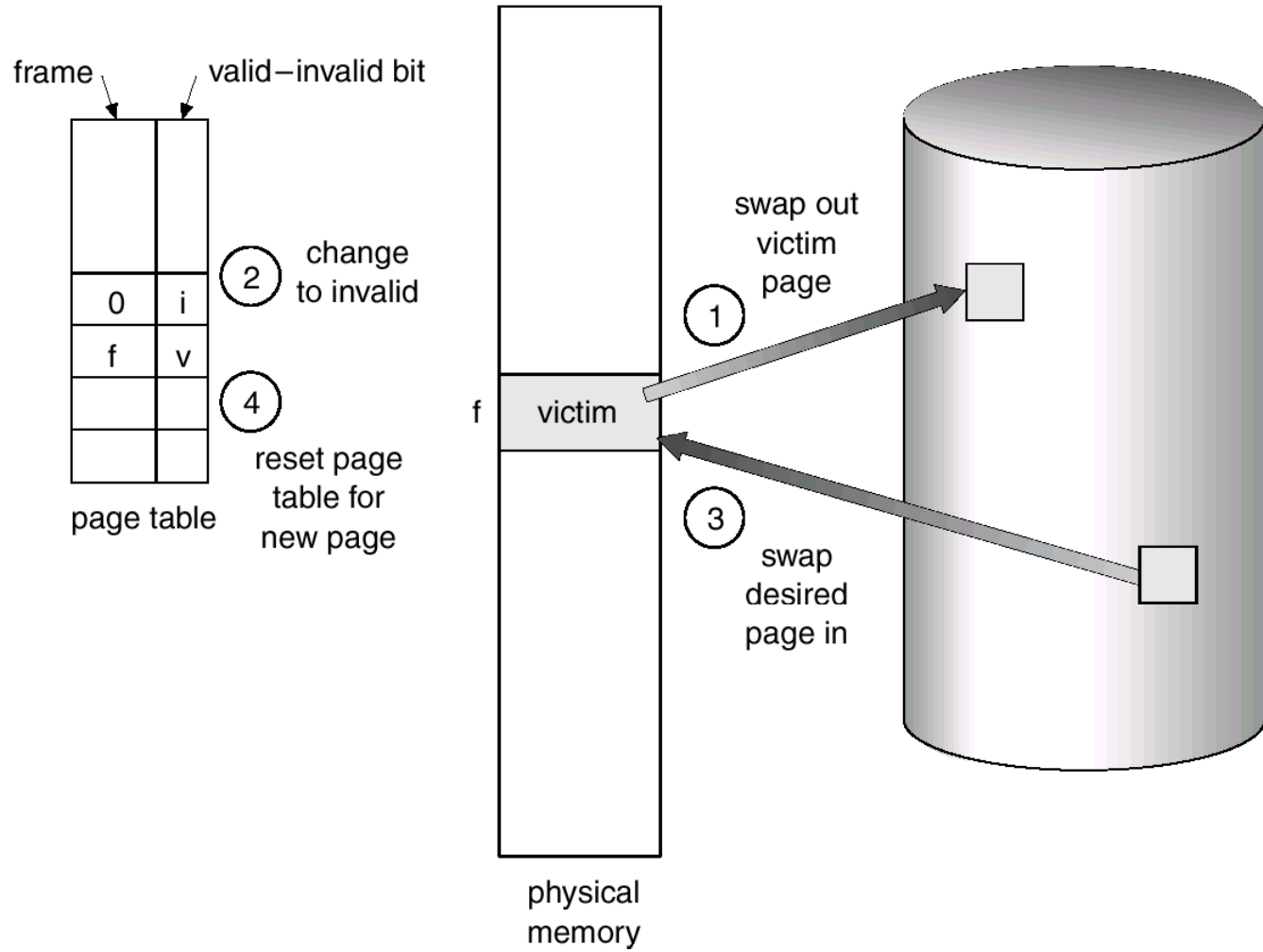
- free page frame이 없는 경우(over-allocation)의 해결책
  1. multiprogramming의 정도를 낮춤
    - 프로세스 종료 또는 프로세스 swap out 후 page frame을 해제함
  2. page replacement
- 페이지 교체(Page replacement)
  - free frame이 없으면
    - 메모리에 있는 사용 중이 아닌 page frame을 선택하여 swap out하고, 새로운 page를 비워진 page frame으로 swap in함
  - demand paging의 기본이 됨
  - 논리 메모리와 물리적 메모리의 분리가 완성됨
- 페이지 교체 알고리즘
  - 교체 대상으로 선택되는 page frame을 결정하는 알고리즘
    - victim frame
  - page fault 수를 최소화하는 알고리즘으로 선정

# 페이지 교체의 기본 과정

- page replacement을 포함하도록 Page-fault service routine을 수정
  1. 디스크에서 필요한 page 위치를 찾음
  2. free frame을 찾음
    - (a) 있으면, 그것을 사용함
    - (b) 없으면, **page replacement 알고리즘**을 사용하여 **victim frame** 선택  
→ victim frame을 디스크에 저장하고 page table과 frame table 수정
  3. 필요한 page를 free frame에 가져오고, page table과 frame table 수정
  4. page fault가 발생한 instruction 재시작
- free frame이 없다면 2번의 page 전송 필요(swap out, swap in)
- Modify bit (Dirty bit)를 사용한 page 전송 감소 방법
  - **Modify (Dirty) bit** – 페이지가 수정되었음을 표시
  - 페이지 교체 시에 수정된 페이지만 쓰기를 수행
    - 페이지의 원본이 디스크에 있을 때에 page가 변경되지 않았으면 swap out 불필요



# Page Replacement View



# Demand Paging and Page Replacement

---

- 요구 페이징 구현에 필요한 두 가지 중요 알고리즘
  - frame 할당 알고리즘
  - page 교체 알고리즘
  
- Frame 할당 알고리즘
  - 여러 프로세스가 존재하는 경우, 각 프로세스에게 얼마나 많은 frame 을 할당해야 하는 가?

# Page-Replacement 알고리즘

- 페이지 부재율이 낮은 것으로 선정

- 알고리즘 평가

- 특정 메모리 참조열(페이지가 참조되는 순서)에 대해서 수행하여 page fault 횟수를 계산
- 참조열(reference string)

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, (page크기=100)  
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105



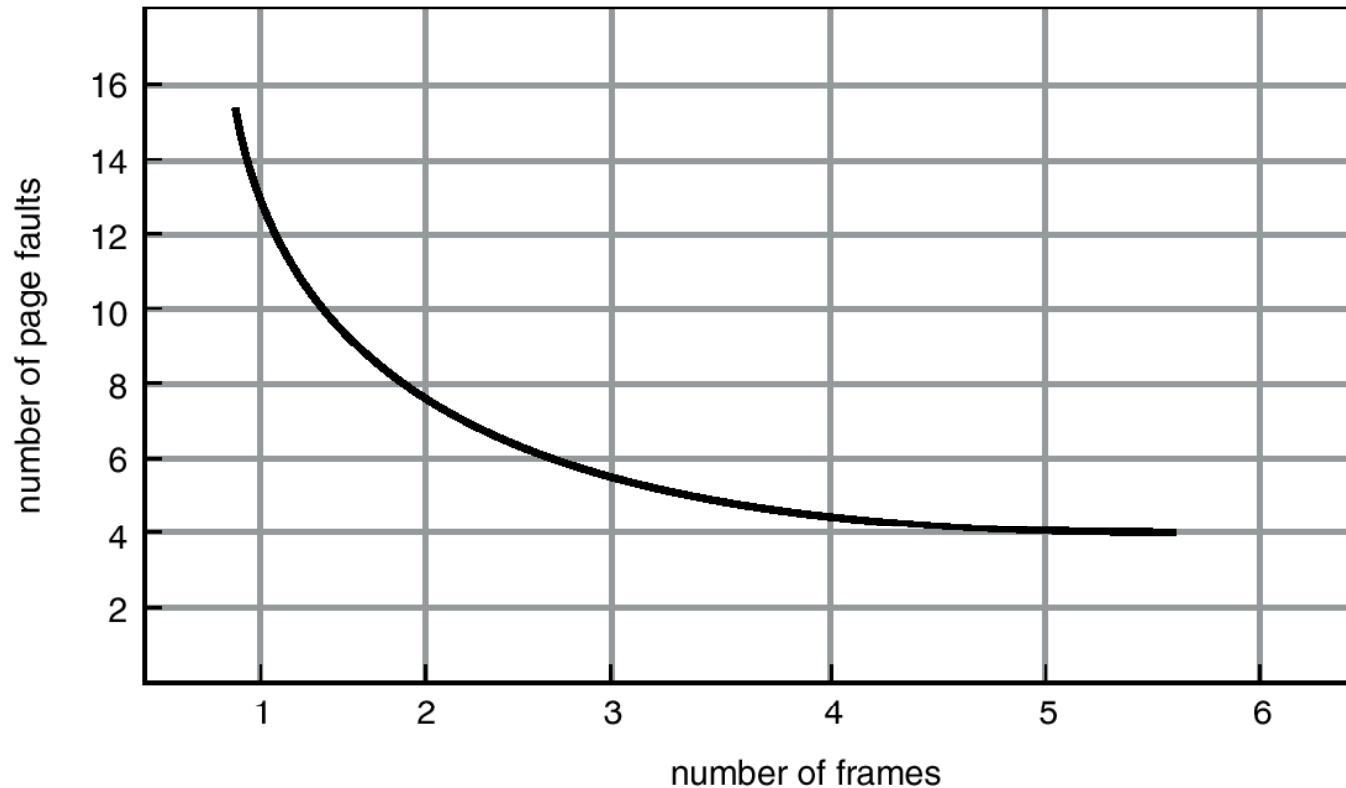
→ reference string: 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

- Page-Replacement 알고리즘

- **FIFO** : 가장 오래된 페이지를 교체
- **LRU** (least recently used) : 최근에 사용되지 않은 페이지를 교체
- LRU Approximation
- Counting-based : LFU(least frequently used), MFU(most FU)
- Page-buffering

# 가용 frame 수와 page fault와의 관계

- 가용 frame 수가 증가할 수록 page fault 수는 감소할 것으로 예상

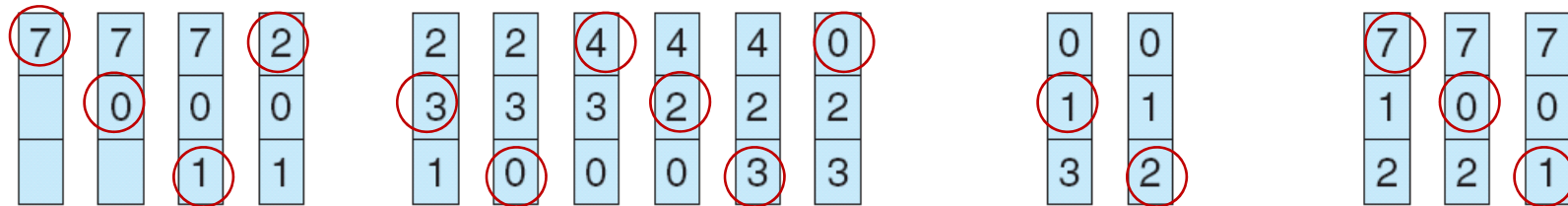


# FIFO Page 교체

## ■ 가장 오래된 page frame을 교체

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

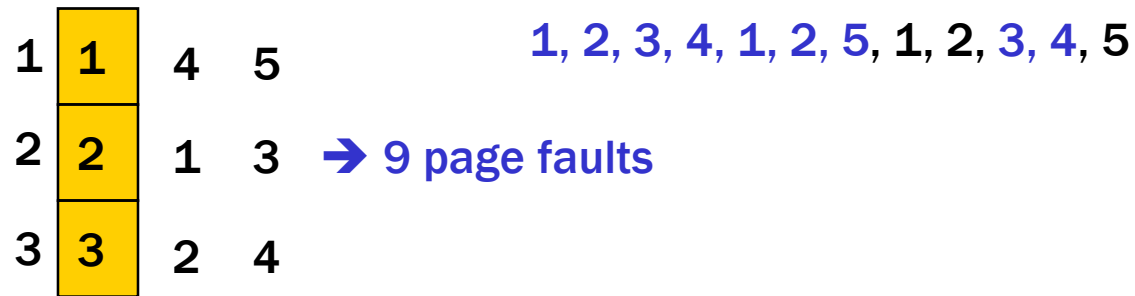


page frames

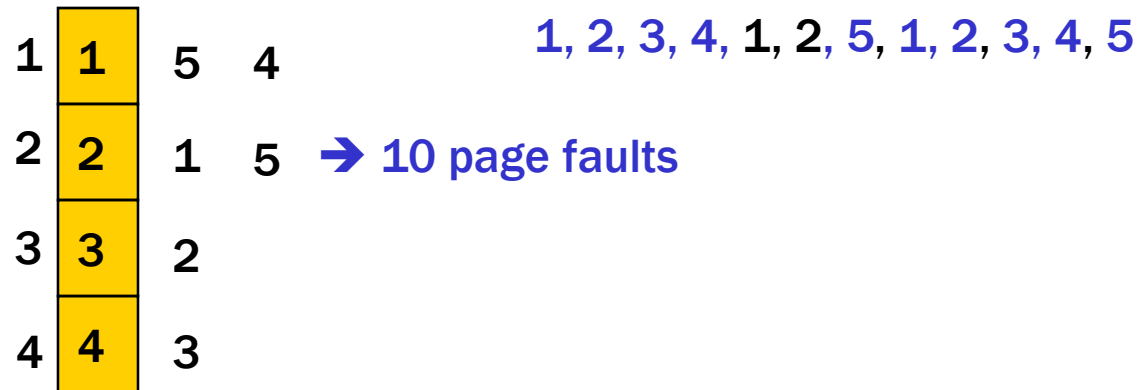
15 page faults

# FIFO 교체 알고리즘 – Belady 모순 발생 가능

- (예) Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 3 frames / process

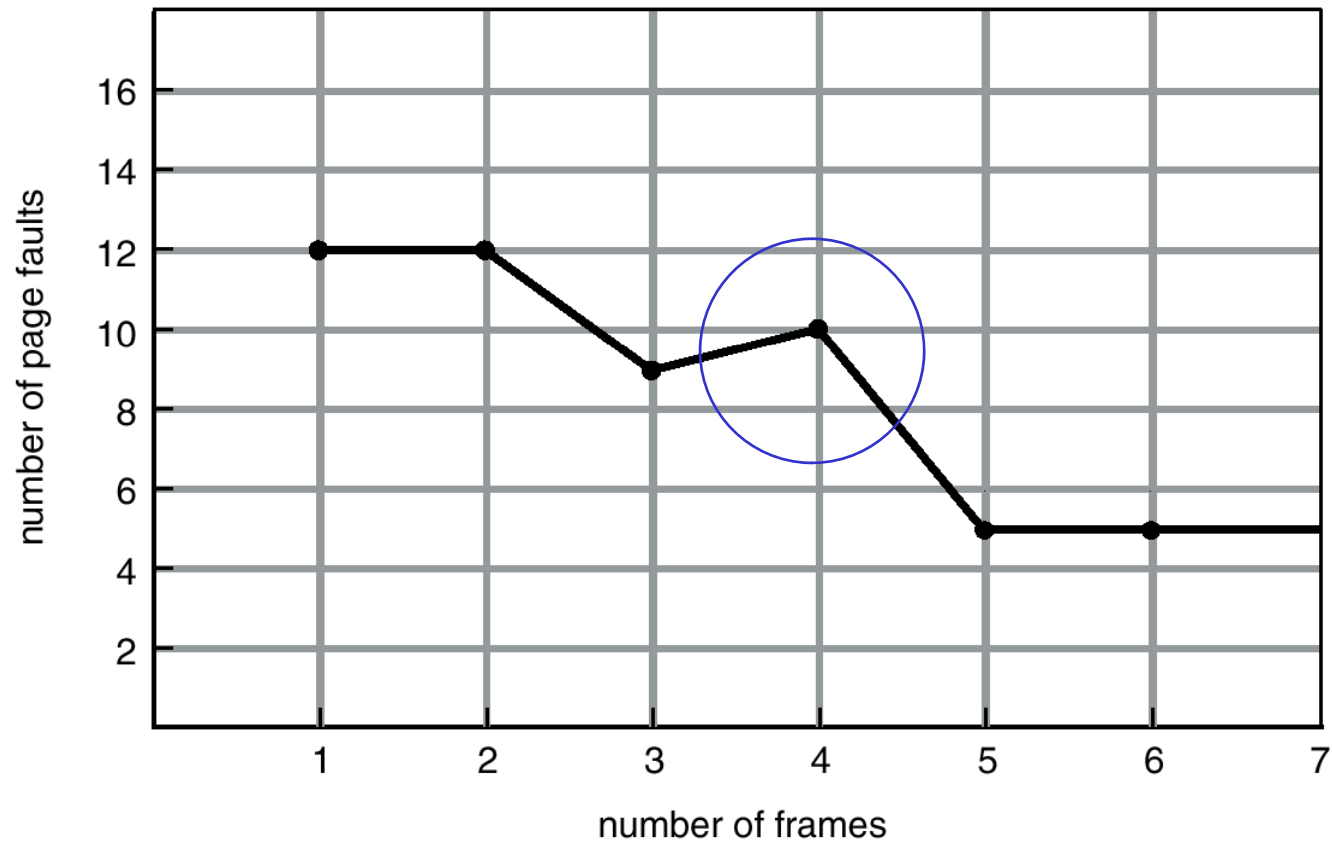


- 4 frames / process



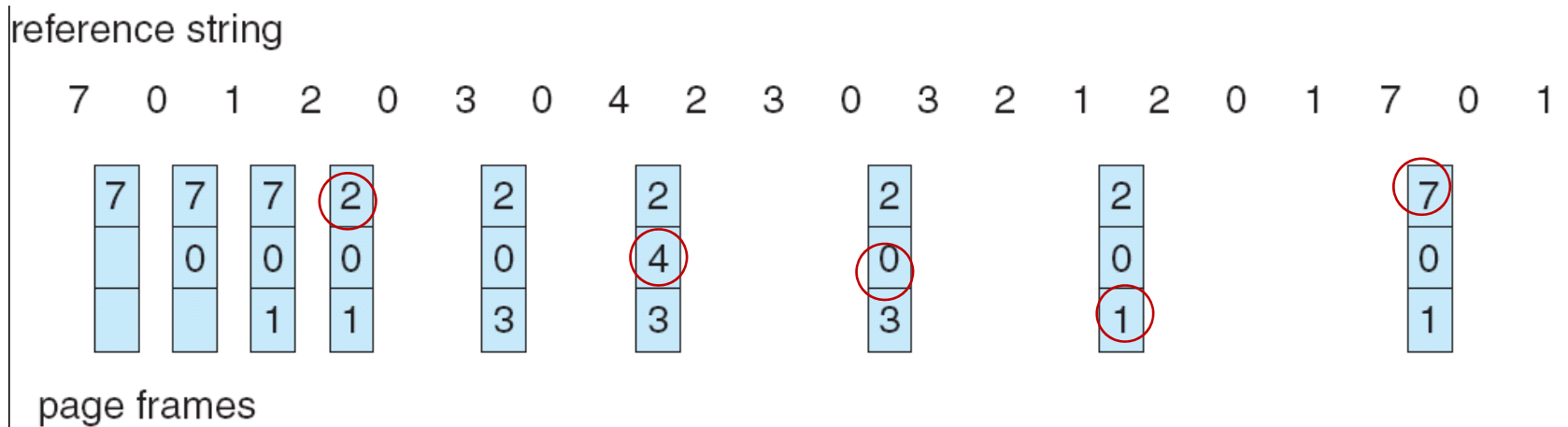
# Belady's Anomaly

- FIFO Replacement – Belady's Anomaly 발생 가능



# Optimal Algorithm

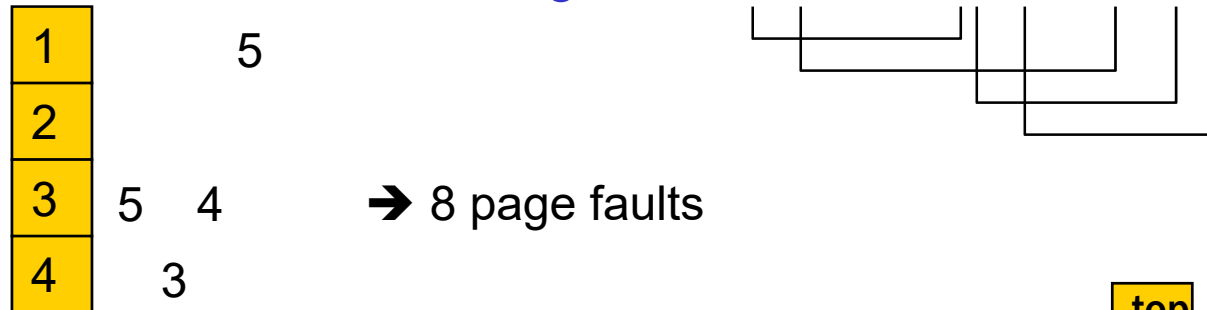
- 가장 오랫동안 사용되지 않을 page를 교체
  - 이러한 page을 알기 어려움 (SJF CPU 스케줄링 알고리즘과 유사)
- 주로 알고리즘의 비교 목적으로 사용됨
- (예)



9 page faults

# Least Recently Used (LRU) Algorithm

- 가장 덜 최근에 사용된 (가장 오래 전에 참조된) page를 교체
  - 최근의 과거를 가까운 미래의 근사치로 추정
- (예) 4 frames: reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- LRU 구현 → 하드웨어 도움이 필요
  - counter – page table entry에 counter 포함.
    - 메모리 참조마다 clock을 증가
    - 페이지 참조마다 clock을 counter로 복사
    - counter값이 최소인 page 교체
  - stack – 페이지 번호 저장용 stack을 유지
    - 페이지 참조마다, 참조된 페이지 번호를 stack top으로 이동
    - stack bottom의 페이지 교체

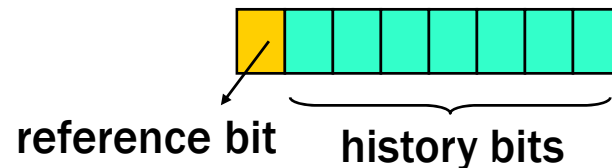
# LRU 근사 알고리즘

## ■ 참조 비트(Reference bit)

- 각 page entry에 1비트 reference bit 사용, 0으로 초기화
- page가 참조되면, 1로 설정
- page가 사용되는 순서는 알지 못함.
- 참조되지 않은(참조 비트=0) page 중 하나를 선택하여 교체

## ■ 부가적 참조 비트(Additional-Reference-Bit) algorithm

- 참조 비트 사용 – 일정 시간마다 참조 비트를 shift right




(LRU) 00000000 < 01110111 < 10000000 < 10001110 (MRU)

# LRU 근사 알고리즘 – 2차 기회 알고리즘

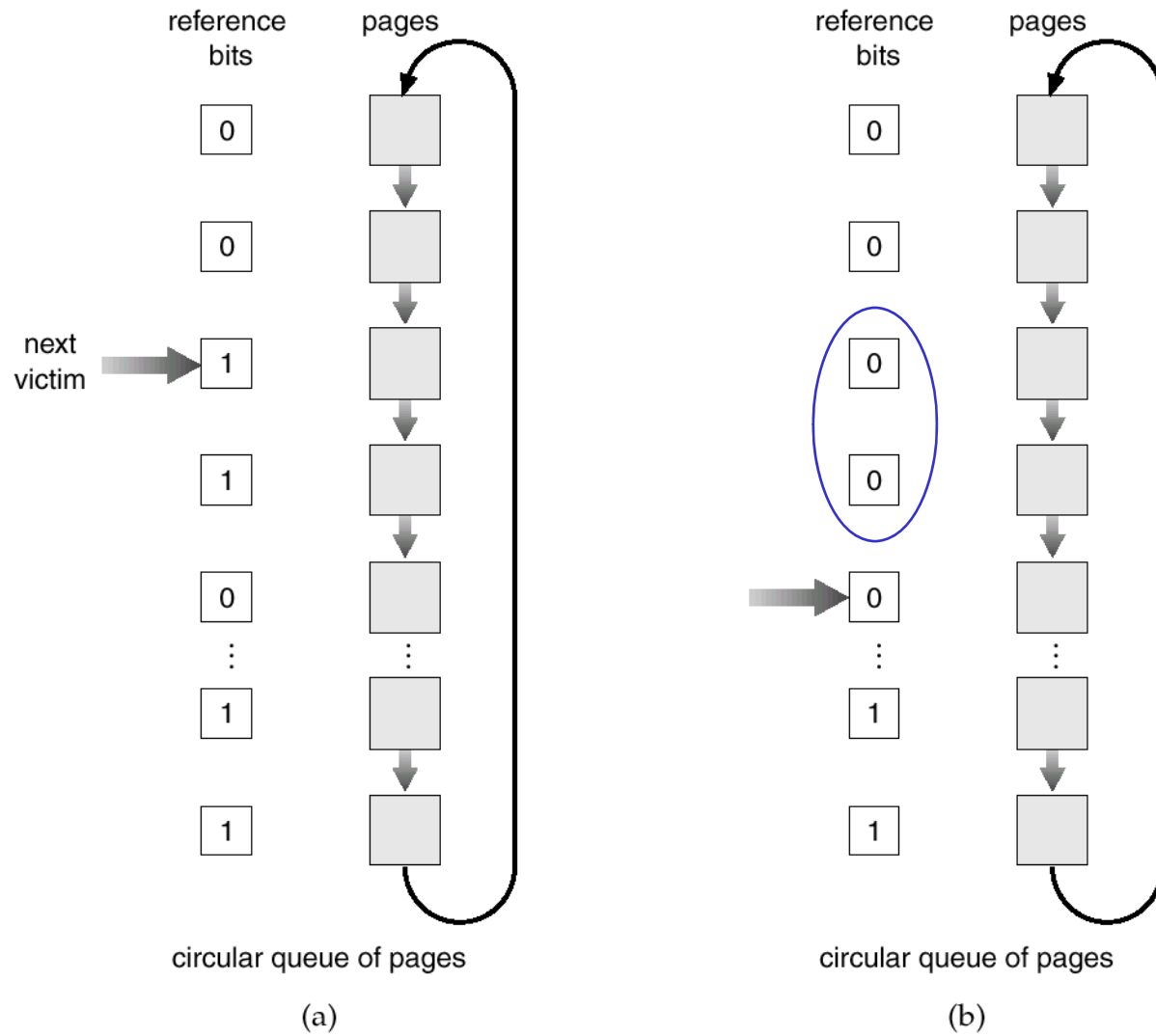
## ■ 2차 기회 (Second chance) 알고리즘

- 참조(reference) 비트만 사용
- page들을 circular queue로 구현 (next slide)
- 교체될 page를 순서대로 조사
  - 참조 비트가 1이면 0으로 설정. 기회를 한 번 더 줌
  - 참조 비트가 0이면 해당 페이지 교체
- clock 교체 알고리즘이라고도 함

## ■ 개선된 LRU 근사 알고리즘

- (reference bit, modify bit) 사용
  - 4가지 조합
    - (0,0) : neither recently-used nor modified (교체 대상 페이지)
    - (0,1) : not recently-used but modified
    - (1,0) : recently-used but not modified
    - (1,1) : recently-used and modified
- 

# Second-chance(clock) page replacement



# Counting Algorithms

---

## ■ Counting Algorithm

- 각 page entry에 참조 횟수 계수를 위한 counter 사용
- 참조 횟수를 기반으로 교체 page 선택 – 많이 사용되지 않음
  - LFU(least frequently used) Algorithm
  - MFU(most frequently used) Algorithm

## ■ Page-Buffering Algorithm

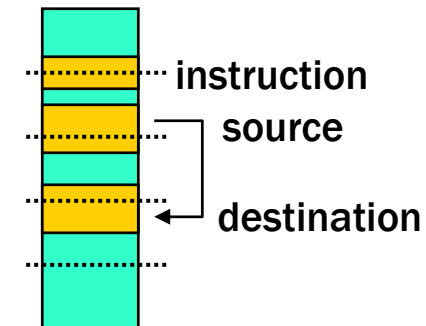
- 가용 frame의 pool을 유지
- 페이지를 먼저 free frame으로 읽은 후에, 편리한 시점에 victim 페이지를 디스크에 저장함

## ■ 수정된 page-buffering algorithm

- modified page list 유지함. paging device가 idle일 때 마다, 수정된 page를 선택하여 디스크에 저장 → victim page을 저장할 필요가 없음
- free frame을 사용했던 page를 기억함  
이 frame을 재사용시에 디스크 입력을 할 필요가 없음

## 8.5 Frame 할당

- process에게 할당하는 frame 수
  - 최소 frame 수 – 아키텍처에 의해서 결정
  - 최대 frame 수 – 가용 물리 메모리 크기에 의해서 결정
- (예) IBM 370 – SS MOVE instruction은 6 page 이상 필요
  - instruction is 6 bytes, might span 2 pages.
  - 2 pages to handle from.
  - 2 pages to handle to.
- 할당 알고리즘
  - fixed allocation
    - 균등할당
    - 비례할당
  - priority allocation



# Allocation Algorithms

---

## ■ 균등할당(Equal allocation)

- 모든 프로세스에게 똑같이 할당  
(예) 100 frames, 5 processes → 각 프로세스에게 20 pages씩 할당

## ■ 비례할당(Proportional allocation)

- 프로세스의 크기에 비례하여 할당
- Let  $m$  = total # of frames,  $s_i$  = size of process  $p_i$   
allocation for process  $p_i$ ,  $a_i = (s_i / S) \times m$   
, where  $S = \sum s_i$

(예)  $m = 64$ ,  $s_1=10$ ,  $s_2=127$  →  $S = s_1+s_2 = 137$

$$a_1 = (10/137) \times 64 \approx 5, a_2 = (127/137) \times 64 \approx 59$$

## ■ 우선순위 할당(Priority allocation)

- 프로세스의 크기 대신에 priority 또는 크기 및 priority의 조합을 사용하여 비례할당 사용

# Global 대 Local Allocation

---

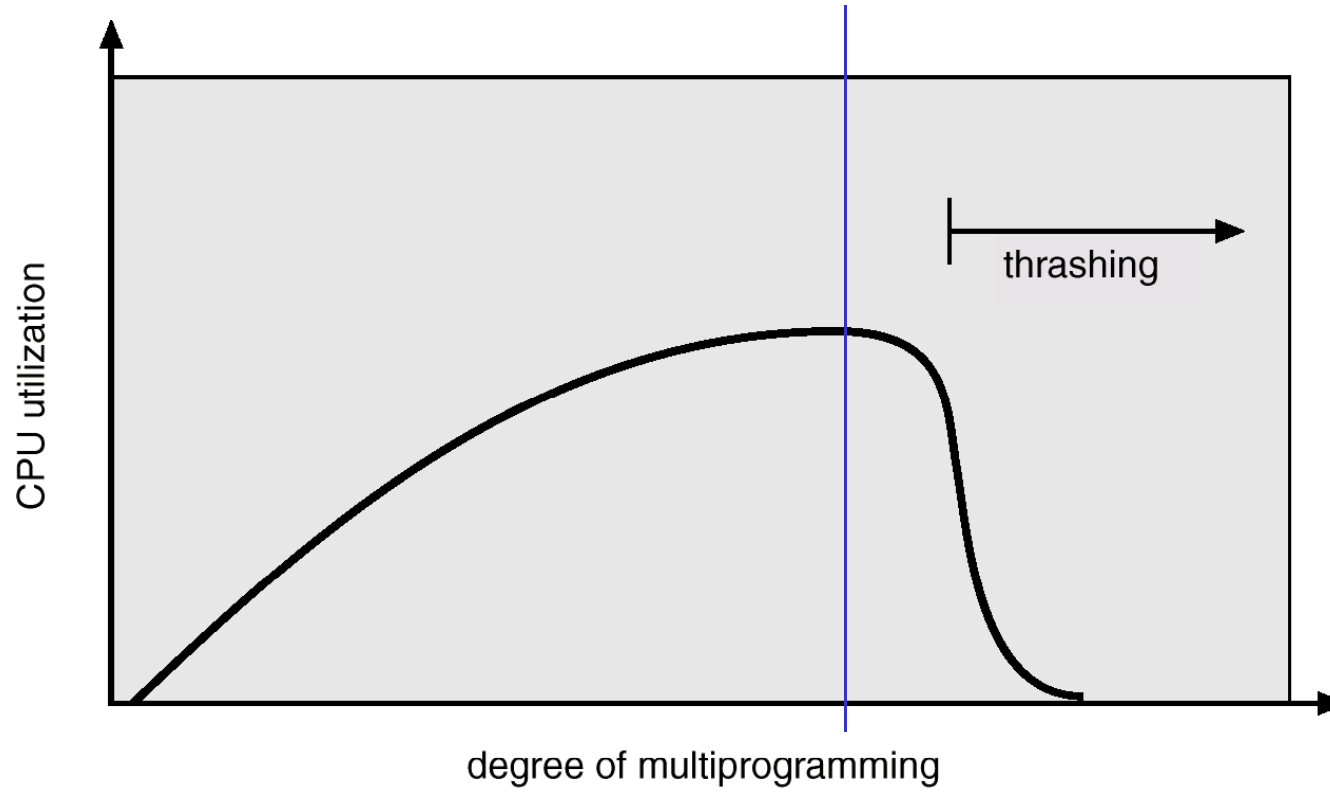
- 전역교체(Global replacement)
  - 모든 frame 집합에서 교체할 frame을 선택함
  - 다른 프로세스로부터 frame을 가져올 수 있음
- 지역교체(Local replacement)
  - 각 프로세스는 자신에게 할당된 frame 집합에서 교체할 frame을 선택
- 일반적으로 전역교체가 시스템 throughput이 더 좋음

## 8.6 쓰레싱(Thrashing)

---

- 프로세스가 충분한 page를 할당 받지 못하면, page-fault rate가 매우 높아지게 됨
  - 프로세스가 swap-in/out에 시간을 소비 →
  - 낮은 CPU utilization. →
  - 운영체제가 multiprogramming의 정도를 증가시키도록 판단 →
  - 다른 프로세스가 시스템에 추가됨 →
  - 더 많은 page faults 발생 →
  - CPU utilization 더 악화됨(See next slide)
  
- Thrashing
  - 빈번한 page 교체로 프로세스가 반복적으로 swap in/out 하느라 바쁜 상황

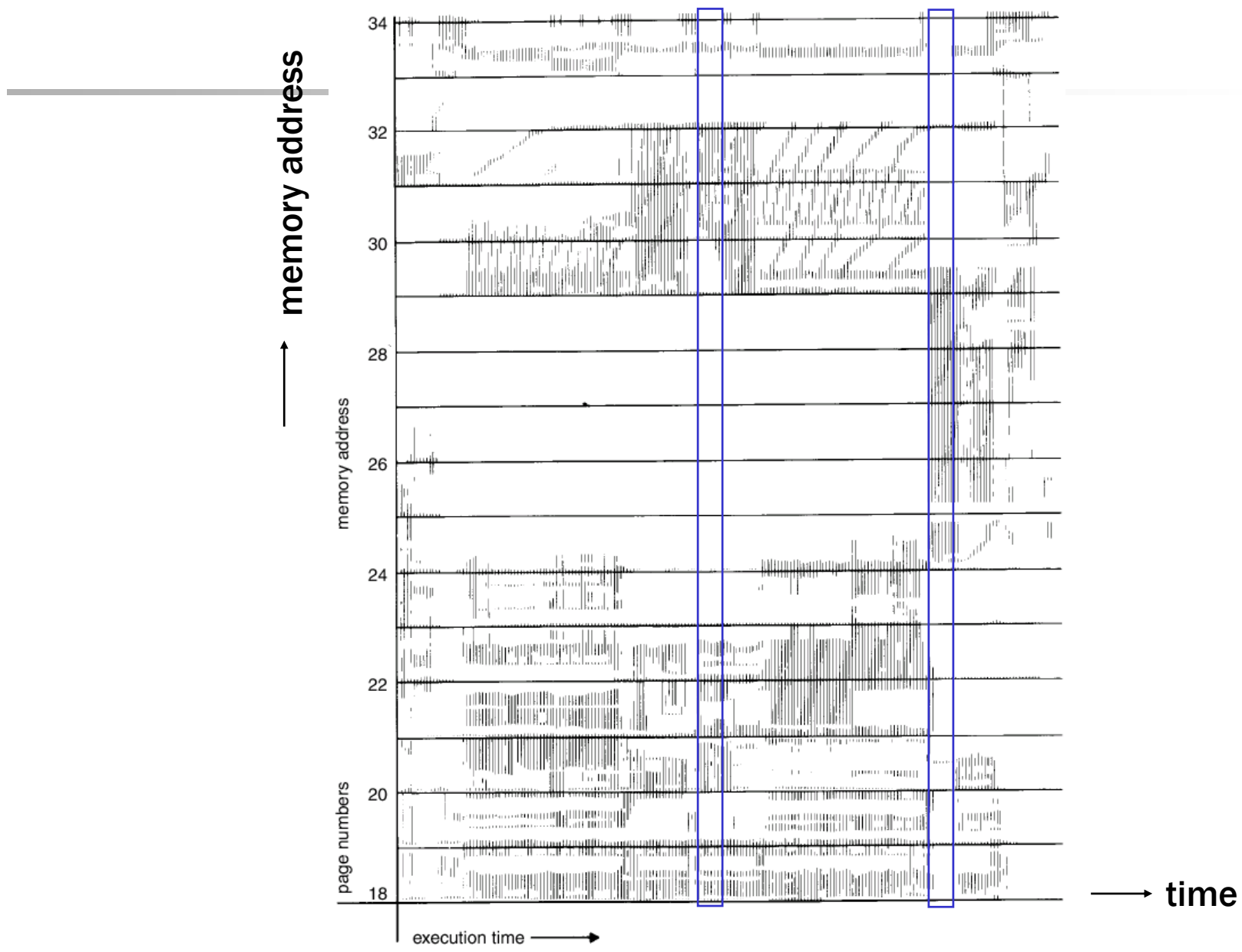
# Thrashing Diagram



# 지역성 모델(Locality Model)

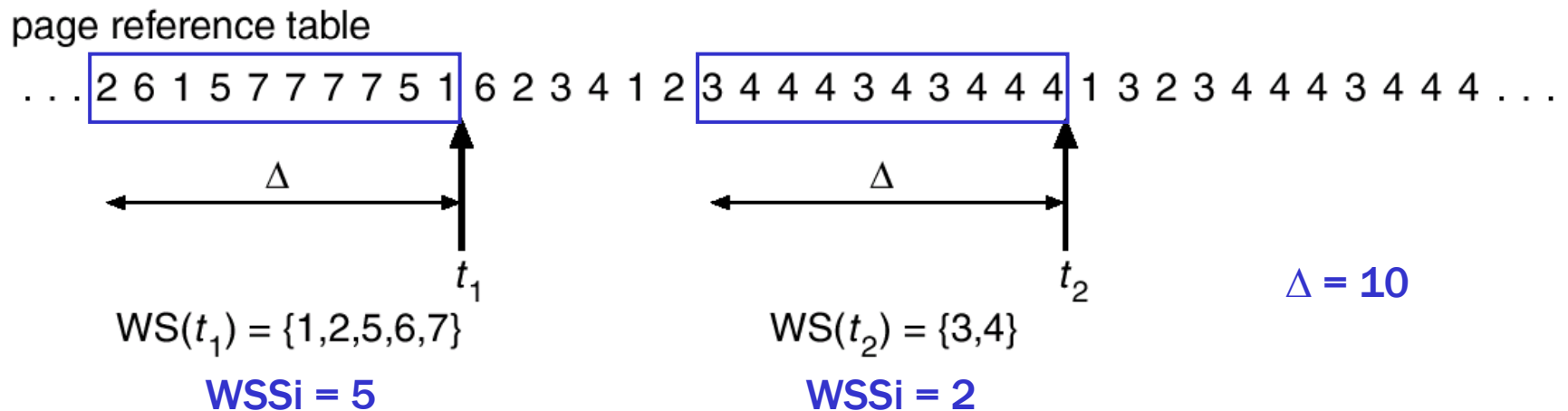
---

- thrashing 방지를 위하여
  - 프로세스에게 필요한 만큼의 frame을 제공해야 함
- 프로세스 실행의 지역성 모델
  - 지역성(locality) : 집중적으로 함께 사용되는 페이지들의 집합
  - 프로그램은 몇 개의 다른 지역성들로 구성됨
  - 프로세스가 실행됨에 따라서 지역성은 이주함
  - 지역성들은 중첩될 수 있음
- thrashing 발생 원인
  - $\sum \text{size of locality} > \text{allocated memory size}$  일 때 thrashing 발생
  - 함께 사용되는 페이지들을 모두 메모리에 적재할 수 없기 때문에



# Working-set model

- working-set window :  $\Delta$ 
  - 고정된 횟수의 페이지 참조 (예) 10,000 instructions
- working set (locality) :  $WS(t)$ 
  - 가장 최근의  $\Delta$ 번의 페이지 참조에 들어있는 서로 다른 page 집합
  - working set size :  $WSS_i = |WS(t)|$ 
    - if  $\Delta$  too small, will not encompass entire locality.
    - if  $\Delta$  too large, will encompass several localities.
    - if  $\Delta = \infty$ ,  $\rightarrow$  will encompass entire program.



## Working-set model(2)

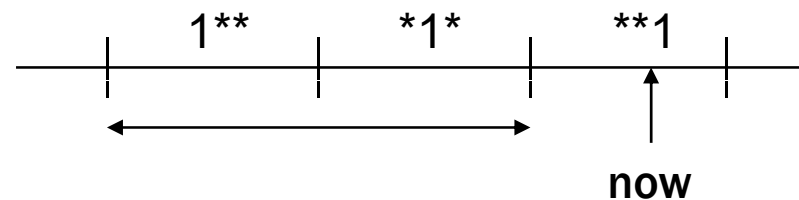
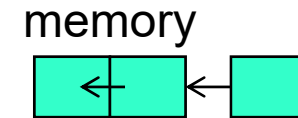
---

- 전체 frame 요구량  $D = \sum WSS_i$ 
  - if  $D > m$ , thrashing 발생 (m: # of available frames)
- Working-set strategy
  - if  $D > m$ , 한 프로세스를 중지하고 이 프로세스의 page들을 다른 프로세스에게 할당하여 thrashing을 방지함

# Working Set 추적

## ■ working set 추적 방법

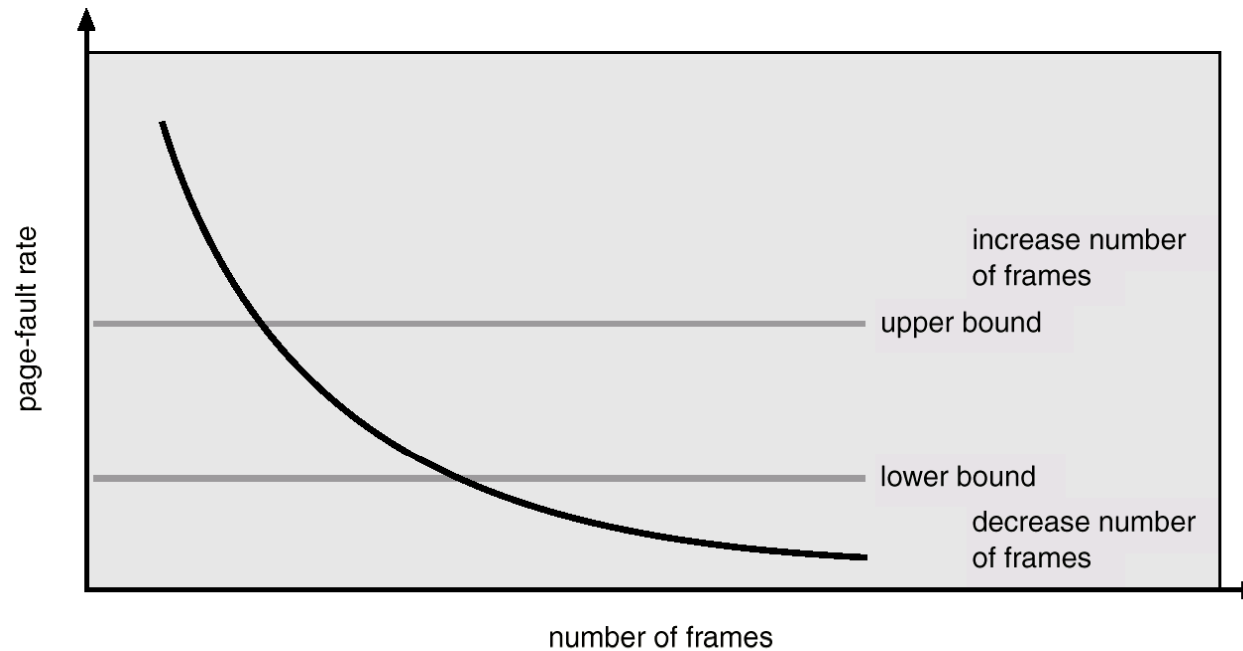
- 고정 시간간격 타이머와 2비트 참조 비트 사용
- (예)  $\Delta = 10,000$ 
  - 각 페이지마다 메모리에 2-bit history 비트 추가
  - 5000 time unit 마다 timer 인터럽트  
→ reference bit를 history로 복사 후 clear
  - 참조 비트와 history 비트 중 적어도 1비트가 1인 페이지는 working set에 있는 것임.



## ■ 정확도 향상 방법

- history bit 수 증가 및 타이머 인터럽트 빈도 증가

# Page-Fault Frequency(PFF) Scheme



## ■ Thrashing 제어 방식

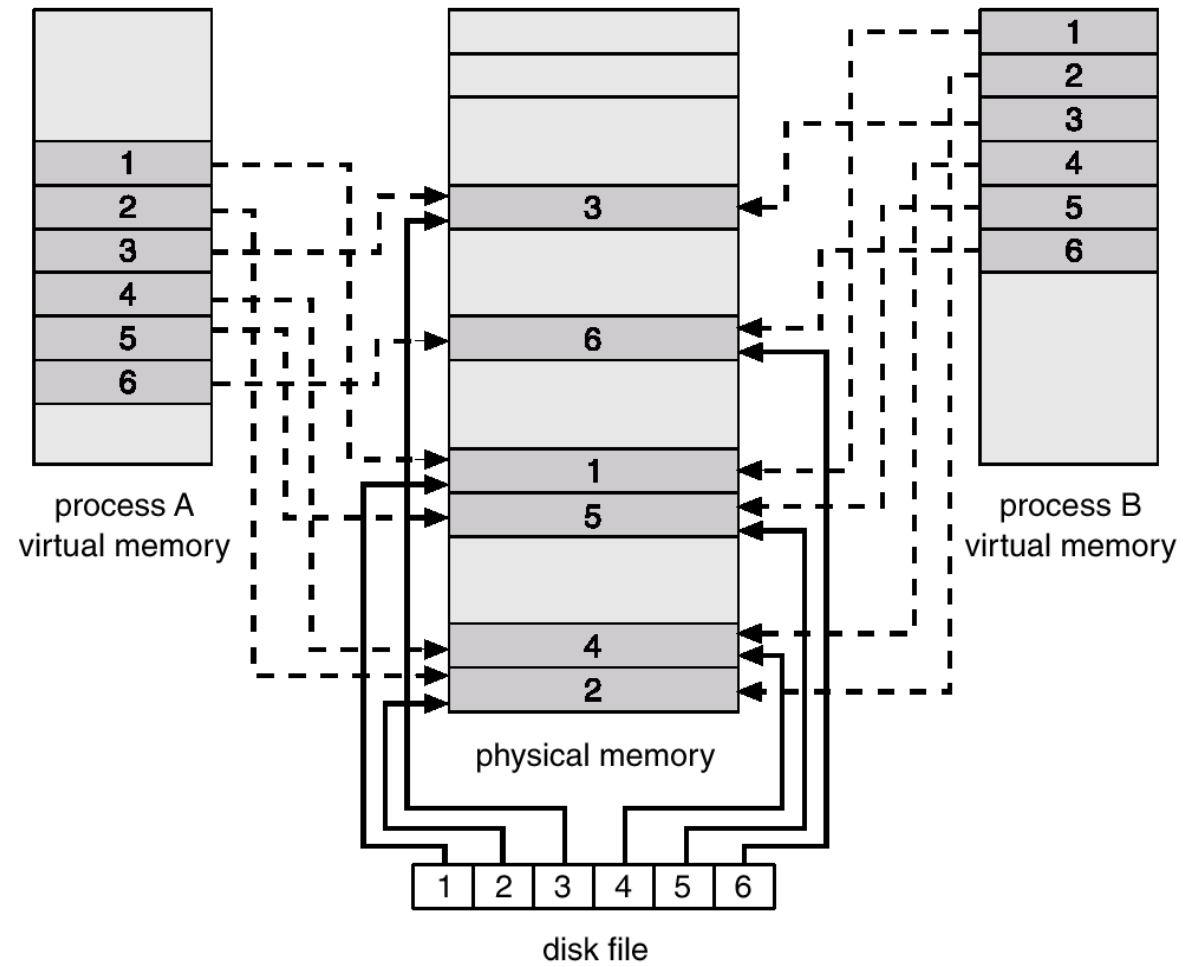
- "acceptable" page-fault rate 설정
  - If page-fault rate too low, 프로세스는 frame 수를 줄임
  - If page-fault rate too high, 프로세스는 frame 수를 늘임

## 8.7 Memory-Mapped Files

---

- Memory-mapped file I/O
  - 디스크 블록(또는 file)을 메모리에 있는 page에 맵핑  
→ file I/O는 보통의 메모리 접근으로 처리
  - 가상 주소 공간의 일부를 논리적으로 파일과 연관시킴
  - mmap() system call 사용 (UNIX와 Linux)
- Memory-mapped file operations
  - 파일 입출력을 open(), read(), write() 시스템 호출 대신에 메모리 접근을 통하여 단순화함.
  - demand paging을 사용하여 파일을 읽음
  - 파일의 page크기 부분이 file system에서 물리적 page로 읽혀짐
  - 지속적인 파일의 읽기/쓰기는 보통의 메모리 접근으로 처리
- 여러 프로세스가 페이지들을 공유하여 같은 파일에 맵핑할 수 있음

# Memory Mapped Files



## 8.9 Other Consideration

---

### ■ Program structure

- demand paging is **transparent** to the user program.  
however, system performance can be improved if the user(compiler) has an awareness the demand paging

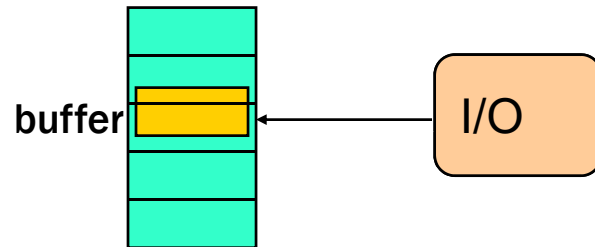
### ■ Example

- `int A[1024][1024] // array`  
page size=128 word  
Each row is stored in one page frames
- Program 1 `for j := 0 to 1023 do`  
`for i := 0 to 1023 do`  
`A[i][j] := 0;`  
1024 x 1024 page faults
- Program 2 `for i := 1 to 1023 do`  
`for j := 1 to 1023 do`  
`A[i][j] := 0;`  
1024 page faults

## Other Consideration (Cont.)

### ■ I/O interlock

- Pages must sometimes be locked into memory.



- Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

### ■ Another solution of I/O buffer problem

- never to I/O to user memory; Instead, I/O takes place only between system memory and I/O device

