

Part 3. 저장 장치 관리

Chapter 9. 가상 메모리

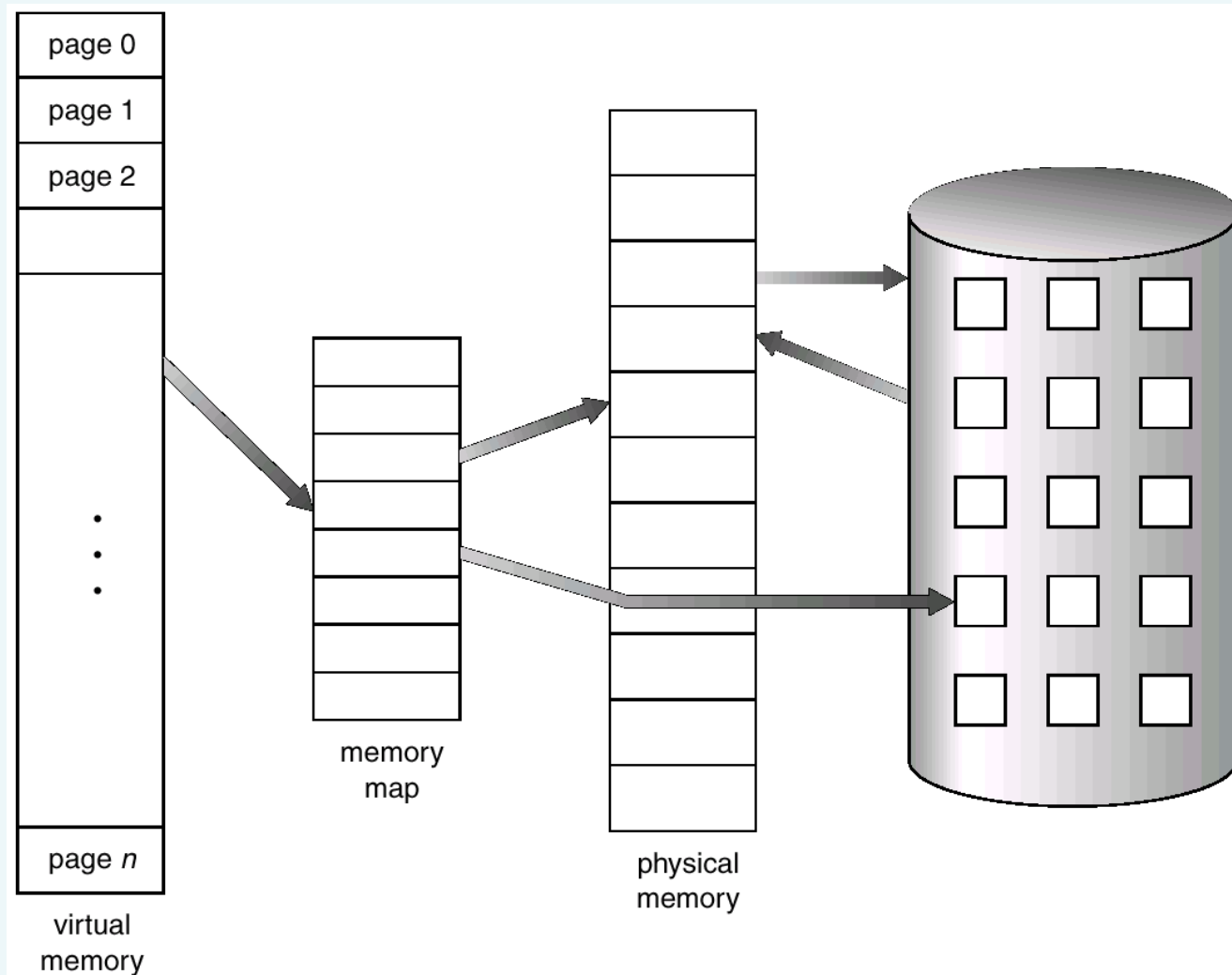
9.1 배경

- 요구 페이징(demand paging) 형태의 가상 메모리
 - 프로그램을 일부분만 적재하고 실행
 - 실제 메모리 크기에 관계 없이 프로그램이 가상의 큰 메모리 공간 사용이 가능
- 부분 적재 실행의 타당성
 - 항상 전체 프로그램 공간이 사용되지 않음
 - 오류 루틴 : 프로그램에서 오류는 거의 발생하지 않음
 - 행렬, 리스트, 테이블 등 : 실제 필요한 양보다 크게 선언되는 경향
 - 한 시점에 모든 프로그램 공간이 사용되지 않음
- 부분 적재 실행의 장점
 - 프로그램이 메모리 크기에 의해 제약받지 않음 (중첩 불필요)
 - 더 많은 프로세스들이 동시 수행 - CPU 활용도/처리율 개선
 - 프로그램 적재/교체(swap)를 위한 입출력 회수 감소

배경

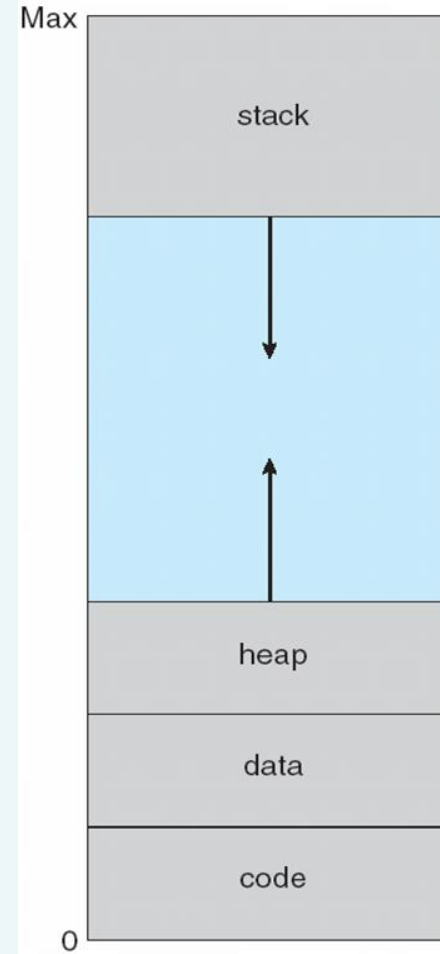
- 부분 적재 실행에서 해결해야 할 문제
 1. 디스크와 메모리 사이의 페이지 이동량이 과다해질 수 있으므로 교체(**swap**) 공간 운영.
 - 교체 공간은 일반 파일 시스템보다 상당히 효율적인 입출력 가능.
 2. 반입 정책 (**fetch policy**) : 언제 어떤 페이지를 적재하고 어느 경우에 페이지를 교체할 것인가 하는 페이지징 알고리즘을 선택.
 3. 교체 정책 (**replacement policy**) : 페이지 교체 알고리즘은 구체적으로 교체될 희생자를 선택하는 알고리즘.
 4. 할당 정책 (**allocation policy**) : 한 프로세스에 필요한 프레임의 수(물리적 공간)와 이 프레임을 차지하는 페이지의 구성 문제.
- 가상 메모리 구현
 1. 요구 페이지징 (**Demand Paging**)
 2. 요구 세그먼테이션 (**Demand Segmentation**)

물리 메모리보다 큰 가상 메모리

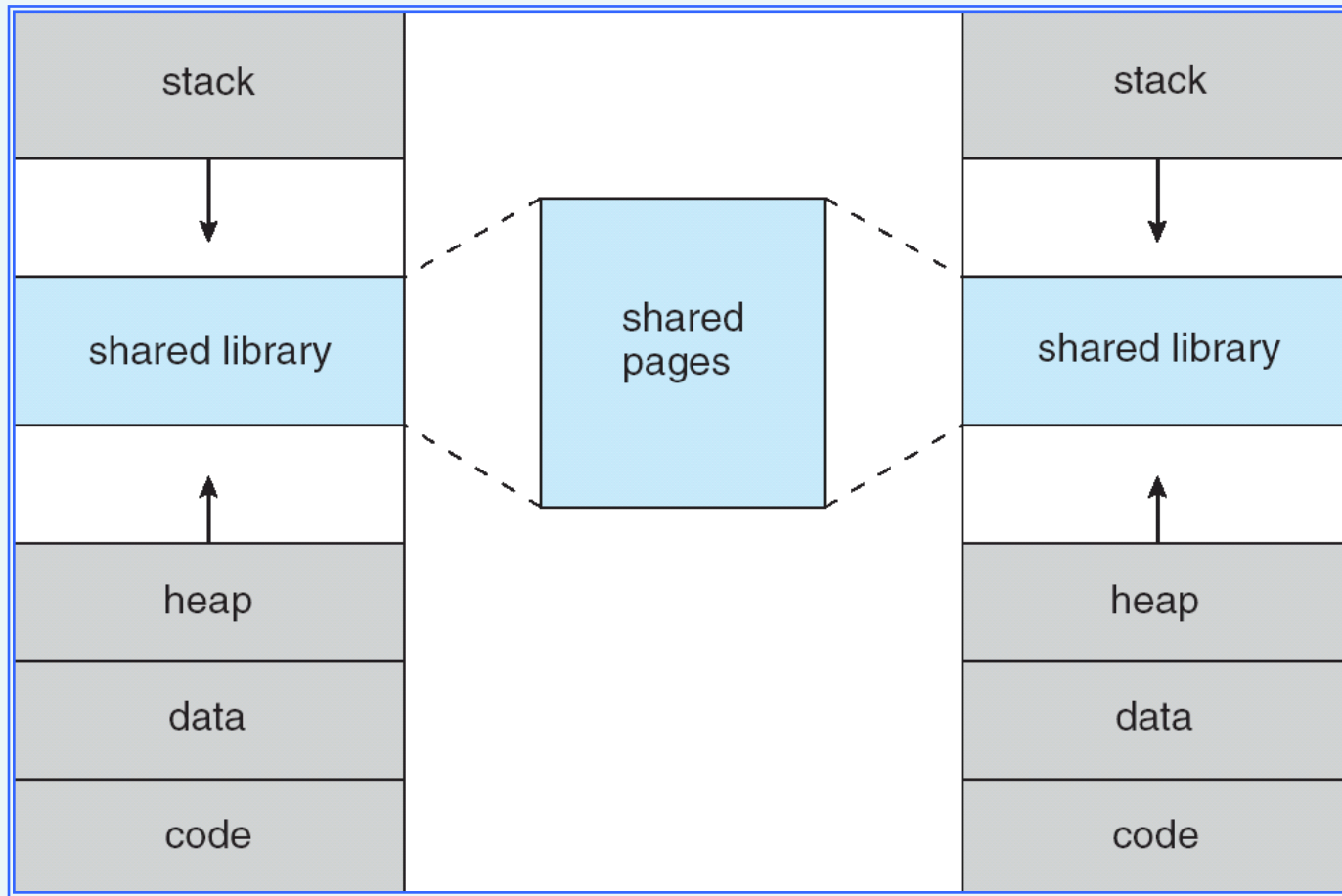


가상 주소 공간

- 일반적으로 스택은 가장 큰 주소에서 시작
- 힙은 위쪽으로 증가 (사용 부분은 분산됨)
- 가상 메모리의 공유
 - 공유되는 동적 라이브러리도 빈 공간 안에 할당
 - 공유되는 메모리
 - **fork()**를 사용한 프로세스 생성 시 메모리 공유도 지원



가상 메모리를 이용한 공유 라이브러리

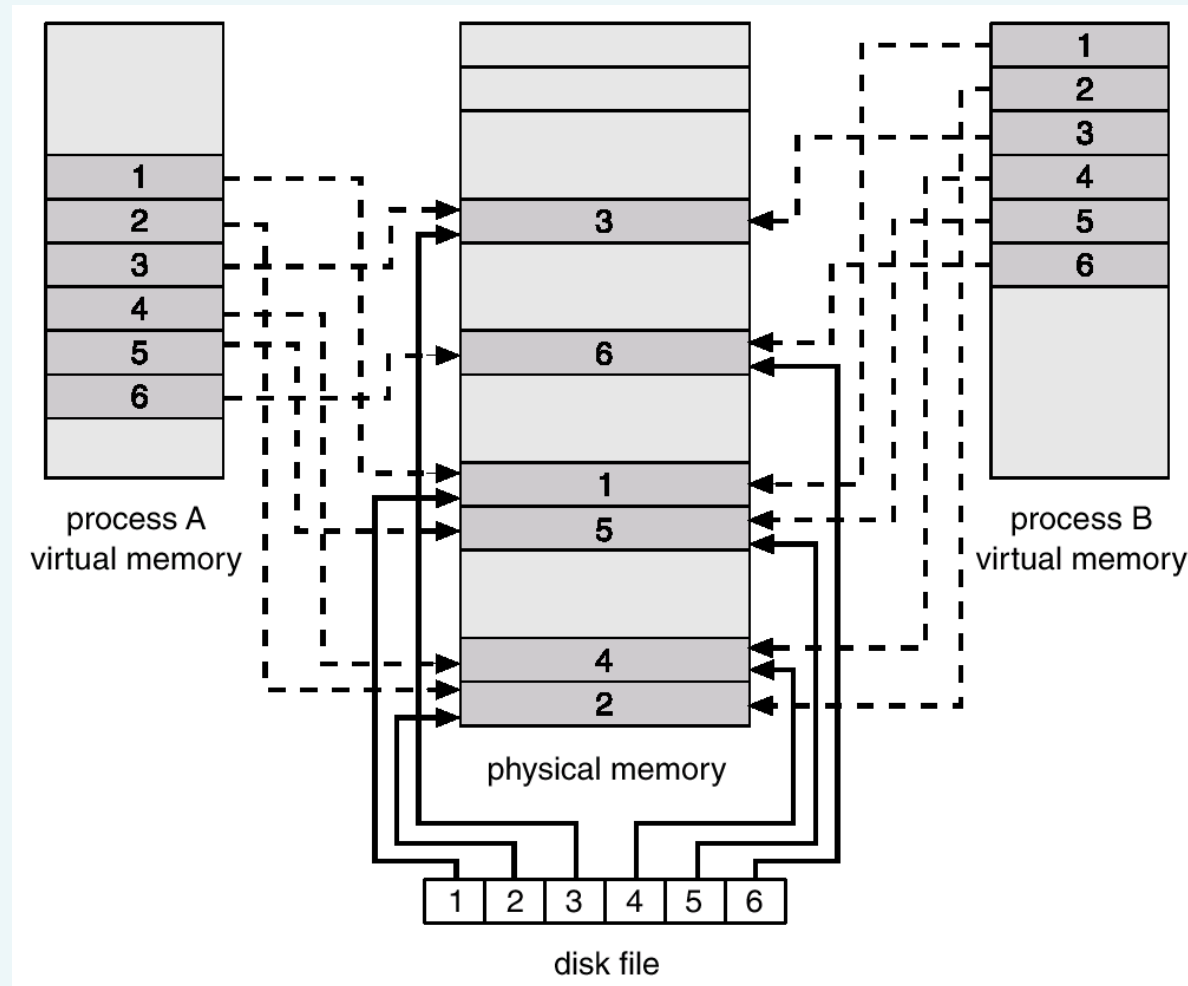


Memory-mapped File

■ Memory-mapped File

- 디스크 블록을 메모리의 한 페이지로 사상(mapping) 함으로써 파일 I/O 를 일반 메모리 접근으로 대신 처리.
- 파일이 요구 페이지징을 사용하여 페이지 단위로 메모리에 읽혀진 다음, 이후의 파일 읽기/쓰기는 일반 메모리 접근으로 처리.
- **read() write()** 시스템 호출 오버헤드를 줄임.
 - 적절한 주기 또는 프로세스가 파일을 close 할 때 디스크로 write.
- memory-mapped 파일을 공유 가능.
 - Solaris2 : mmap()

Memory-mapped File



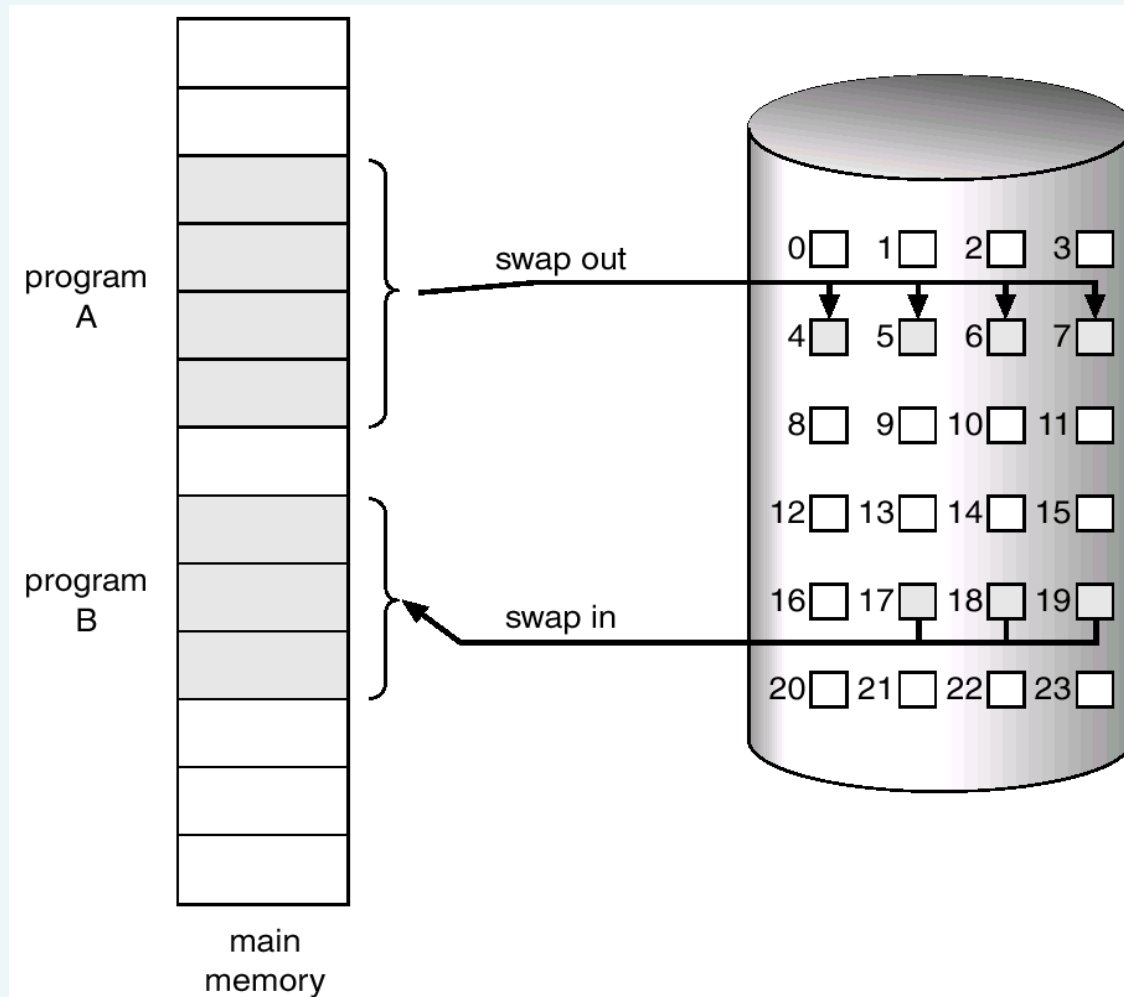
Mmap 예제

```
int main(int argc, char **argv) {
    int fd;
    char *file = NULL;
    struct stat sb;
    char buf[80] = {0x00};
    int flag = PROT_WRITE | PROT_READ;
    if (argc < 2) { fprintf(stderr, "Usage: input\n"); exit(1); }
    if ((fd = open(argv[1], O_RDWR|O_CREAT)) < 0) {
        perror("File Open Error");
        exit(1);
    }
    if ((file = (char *) mmap(0, 40, flag, MAP_SHARED, fd, 0)) == -1) {
        perror("mmap error"); exit(1);
    }
    printf("%s\n", file);
    *file = 'x';
    strcpy(file+2, "THIS IS TEST");
    close(fd);
}
```

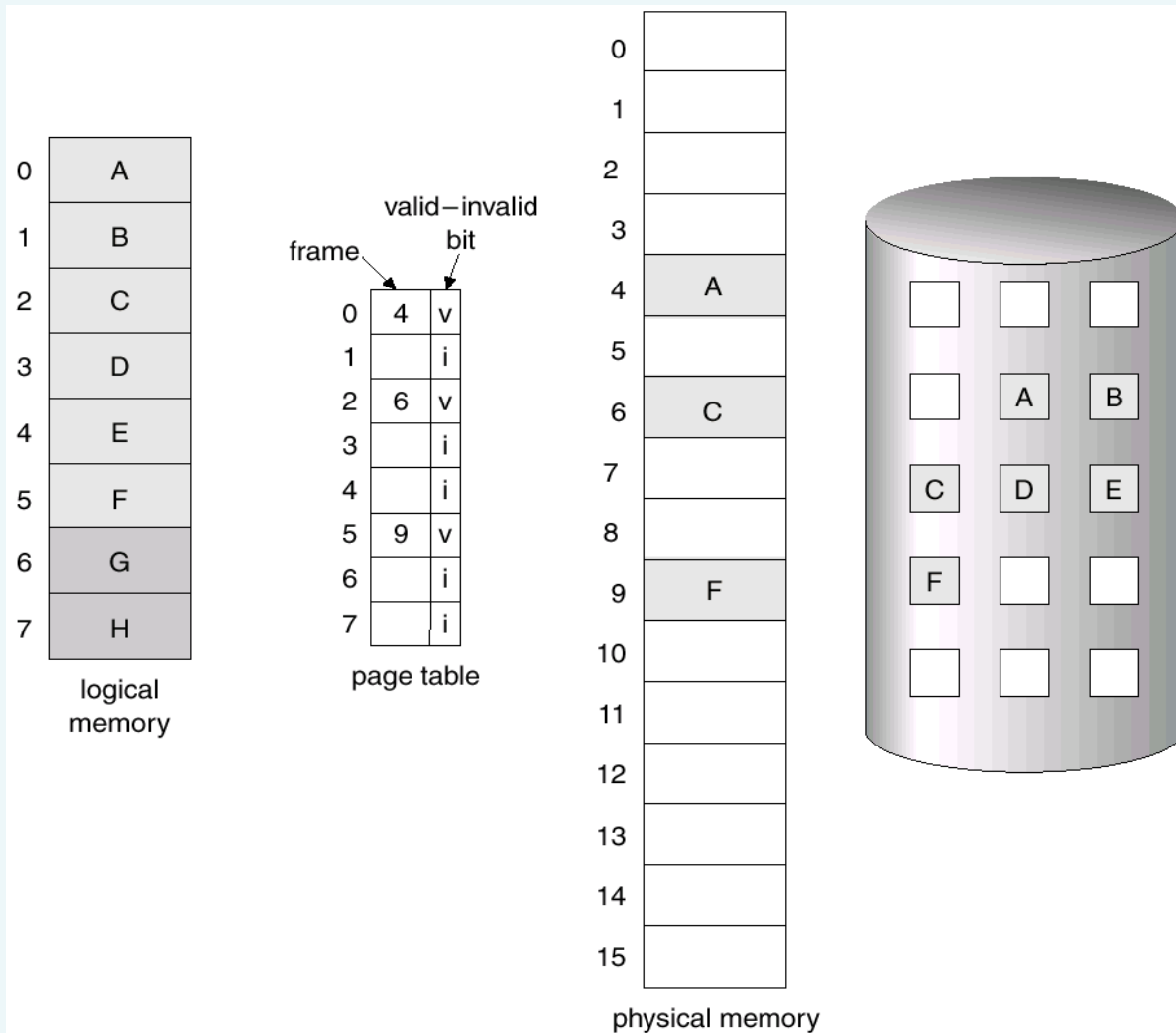
9.2 요구 페이징 (Demand Paging)

- 교체 기법을 사용하는 페이징 시스템과 유사
 - Swapper : 프로세서 전체를 올리고 내림
 - 요구 페이징
 - 게으른 스와퍼
 - 페이저(pager): 페이지 단위의 메모리 교체
 - » 부재 페이지의 참조시에만 페이지를 적재
 - » 주메모리가 부족한 경우에만 한 페이지를 교체
 - I/O 회수 감소, 적은 메모리 요구, 빠른 응답, 많은 사용자 가능
- 요구 페이징의 구현
 - 페이지 테이블의 유효/무효 비트를 사용
 - 무효 : 페이지가 메모리에 없거나 프로세스가 접근 불가
 - 유효 : 관련 페이지가 메모리에 있는 경우
 - 순수 요구 페이징: 모두 무효로 초기화

연속적인 디스크 공간으로 페이지화된 메모리 이동



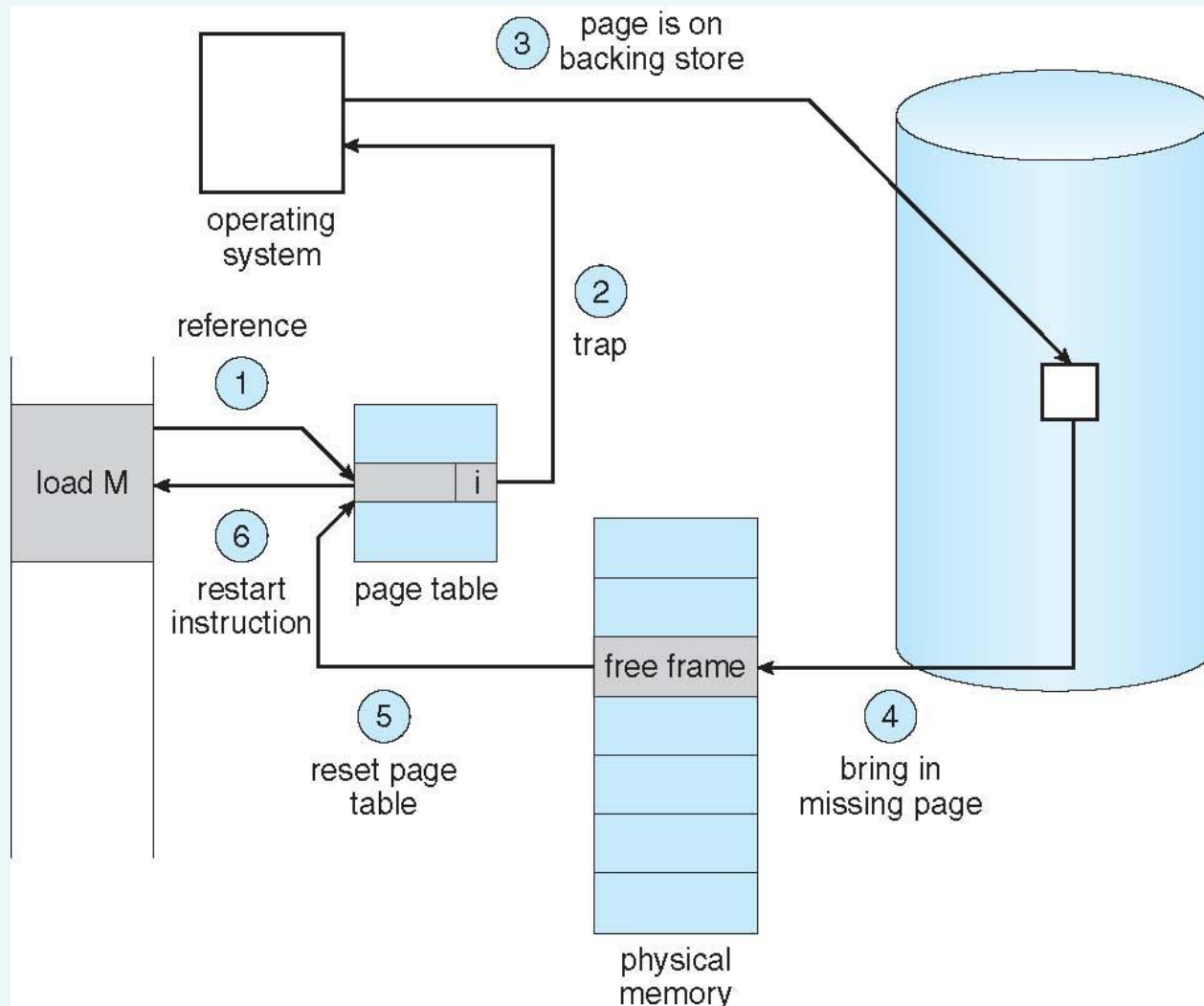
주 메모리에 몇몇 페이지가 없을 때의 페이지 테이블



페이지 부재 오류(Page Fault)의 처리

- 페이지 오류 처리 과정
 1. 프로세스 내부 테이블(internal table)의 검사를 통하여 해당 페이지에 대한 유/무효 비트를 검사
 2. 프로세스가 접근할 수 없는 페이지(논리적 주소 공간에 없는 경우 등)라면 프로세스는 중단(abort).
 3. 페이지가 주메모리에 올라오지 않았을 경우 하드웨어는 OS가 트랩을 발생시키도록 함
 4. OS는 페이지를 비어있는 프레임(free frame)으로 읽어 들임
 5. 페이지 테이블을 재설정하고, 참조 페이지의 유효 비트를 1로 수정.
 6. 명령어 재개

페이지 부재 오류(Page Fault)의 처리

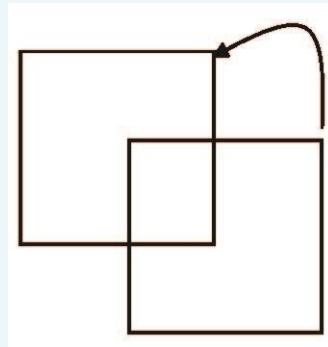


요구 페이징의 고려사항

- 극단적인 경우: 순수 요구 페이징
 - 프로세스가 메모리에 전혀 없는 상황에서 프로세스 수행 시작 → 페이지 폴트로 프로그램 시작
 - 여러 개의 페이지를 접근하는 명령어의 경우: 여러 번의 페이지 폴트
- 하드웨어 지원 필요
 - 페이지 테이블의 유효/무효 비트
 - 효율적인 보조 기억장치 지원
 - 명령어 재시작 문제

요구 페이징의 고려사항

- 페이지 폴트로 인한 명령어 재시작 문제
 - 여러 개의 페이지를 접근하는 명령어의 경우



- 해결책
 - 실제 메모리 변경 전에 페이지 유효 확인
 - 임시 레지스터에 저장될 값이 거치도록 함: 폴트 발생 시 메모리 복구

페이지 부재의 처리

- 페이지 부재 처리 과정
 1. 하드웨어 : 운영체제에게 트랩 요청
 2. 레지스터, CPU 내 각종 상태 저장
 3. 페이지 폴트 인터럽트 확인
 4. 페이지 참조의 유효성 검사 후 디스크 내 페이지 위치 검출
 5. 디스크로부터 필요 페이지를 빈 프레임에 로드
 - a. 해당 디스크를 위한 큐에서 대기
 - b. 해당 블록 접근을 위한 지연 시간 소요
 - c. 데이터 전송
 6. CPU는 다른 프로세스에 할당
 7. 디스크 입력 완료 인터럽트
 8. 수행중인 다른 프로세스 저장
 9. 인터럽트가 발신지/원인 확인(디스크/페이지 인터럽트)
 10. 페이지 테이블 갱신, 프로세스 상태 **ready** 상태로 전환
 11. CPU 할당
 12. 명령어 수행 재개

요구 페이징의 성능

- 페이지 부재율 $0 \leq p \leq 1.0$
 - if $p = 0$ – 페이지 부재 없음
 - if $p = 1$ – 매번 참조가 페이지 오류
- 실제 접근 시간 (Effective Access Time, EAT)
$$\text{EAT} = (1 - p) \times \text{메모리 접근 시간} + p \times (\text{페이지 부재 시간})$$
 - 페이지 부재 시간 = 인터럽트 처리 + 페이지 읽기 + 프로세스 재시작 시간
- 참조의 지역성 \rightarrow 일반적으로 p 는 0에 가까움

페이지 부재의 처리 예

- 페이지 부재 시간 25ms, 메모리 접근 시간 100ns
실제 접근 시간 = $(1-p) \times 100 + p \times 25000000$
 $= 100 + 24,999,900 \times p$
- 페이지 부재로 인한 성능저하를 10% 미만으로 유지
 $110 > 100 + 25,000,000 \times p$
 $10 > 25,000,000 \times p,$
 $p < 0.0000004$

→ 페이지 부재의 발생 빈도가 0.00004% 이하여야 함

스왑 공간의 처리

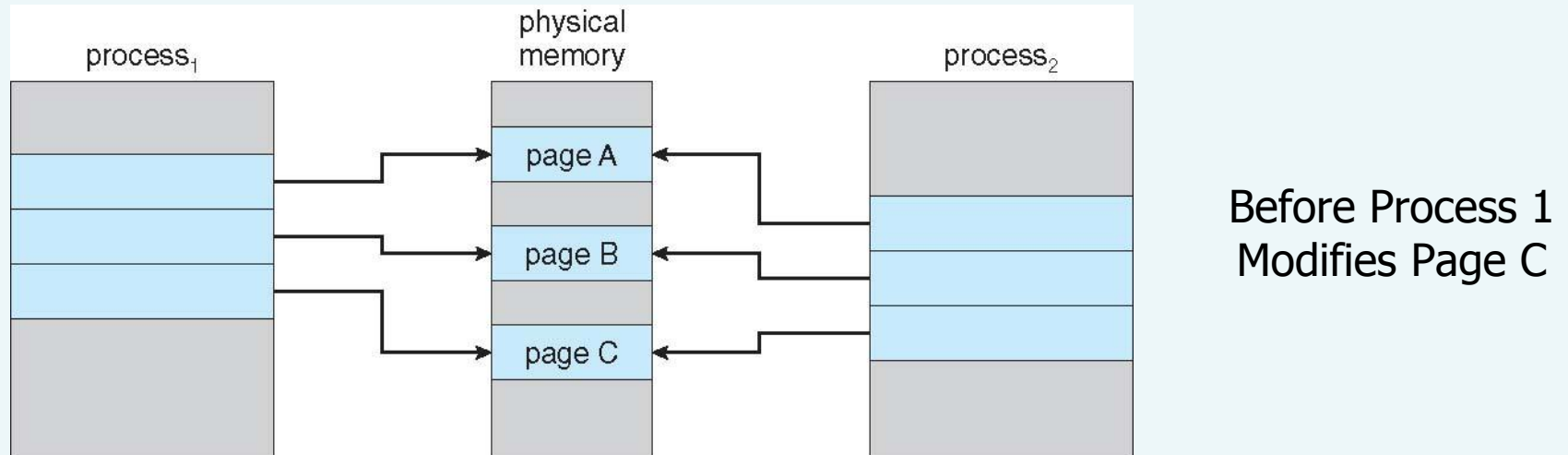
- 성능 향상을 위한 스왑 공간의 구현
 - 일반적인 파일 시스템보다 큰 블록을 사용
 - 파일 찾거나 간접 할당을 사용하지 않음
 - 스왑 공간의 할당 방법
 1. 프로그램의 전체 이미지를 스왑 공간에 복사하고 난 후 프로세스를 시작
 2. 파일로부터 요구 페이징 후 **replace** 시에 스왑 공간에 기록
 3. 이진 파일의 경우 일반 파일 시스템을 그대로 사용하고 스택이나 힙과 같은 자료 공간에 대해서만 스왑 공간 할당

9.3 Copy-on-Write

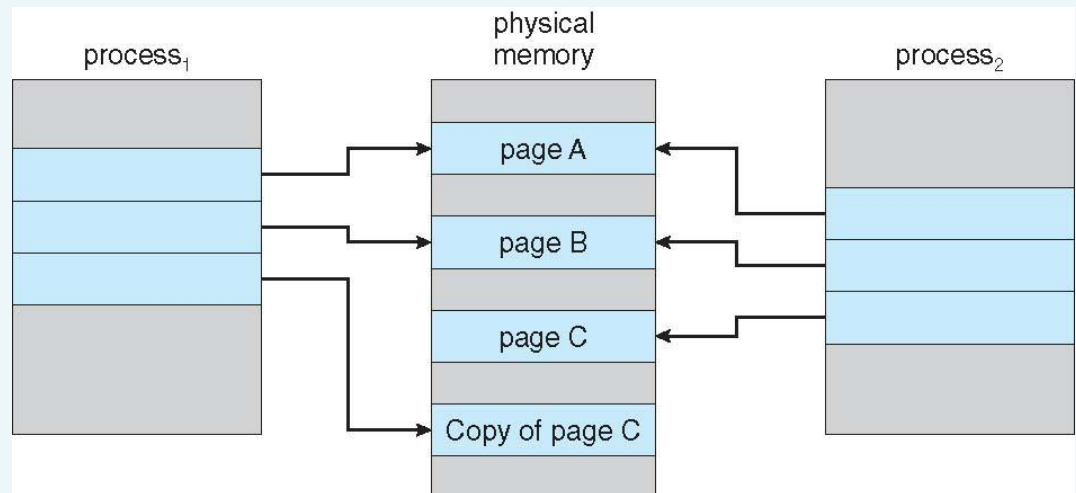
■ Copy-on-Write (COW)

- 부모와 자식 프로세스가 처음 생성 시에는 메모리의 같은 페이지를 공유하도록 허용.
- 프로세스 중 하나가 공유 페이지를 수정할 때, 페이지가 복사
- 수정된 페이지만 복사하므로 더욱 효율적인 프로세스 생성 가능
- 영으로 채워진(zero-filled) 페이지 풀(pool)로부터 free page 를 할당받아 페이지 복사.
- vfork
 - Child가 부모의 페이지에 write를 할 수 있음 (copy-on-write 작동 안함)
 - 일반적으로 exec을 수행할 경우에 효율을 위하여 사용

9.3 Copy-on-Write



After Process 1
Modifies Page C

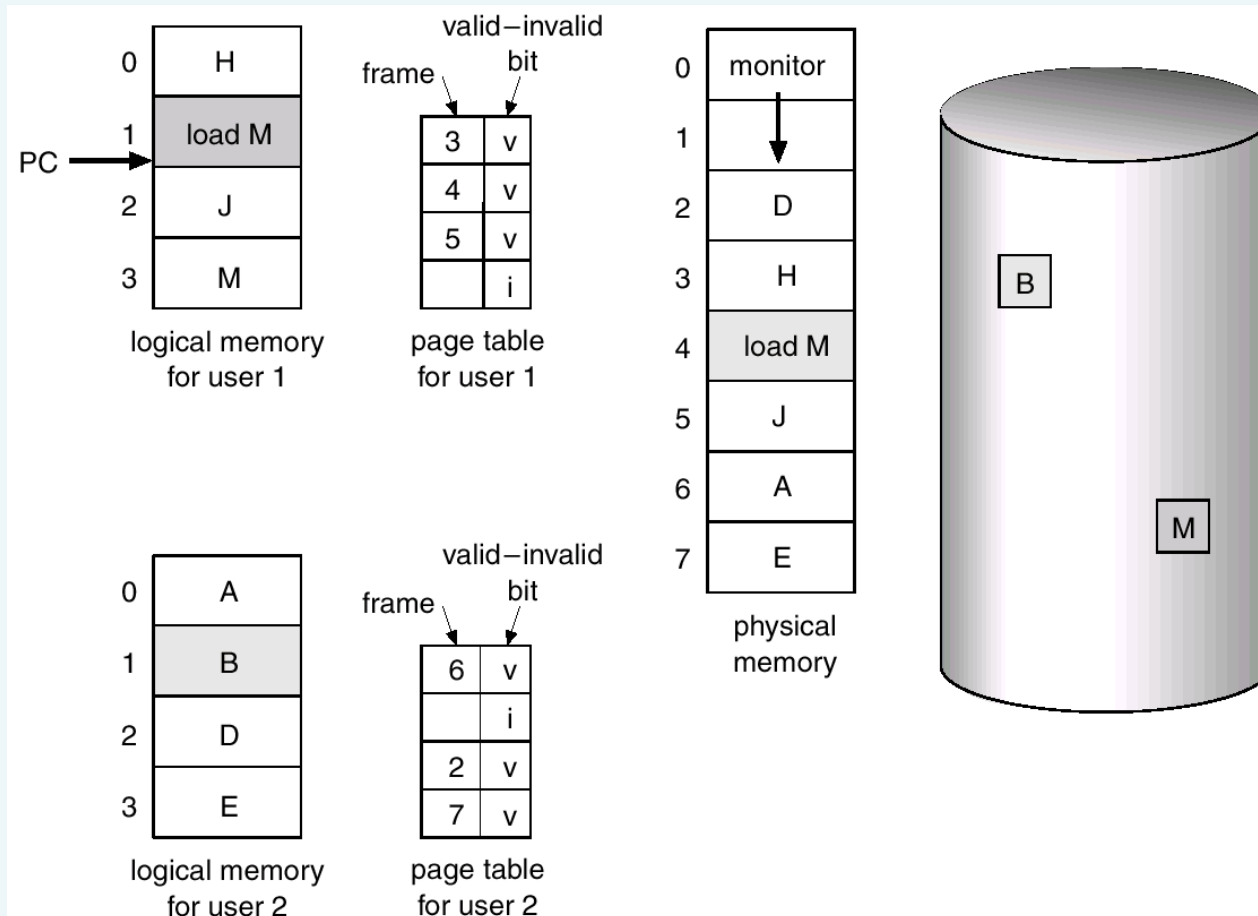


9.4 페이지 교체 (Page Replacement)

- 페이지 교체의 필요성
 - 프레임이 모두 할당되었을 경우의 선택
 - 페이지 폴트의 최소화가 필요
 - 같은 페이지가 여러 번 메모리에 진입할 수 있음
- 과도한 메모리 할당을 방지
 - **Modify** 비트를 사용하여 필요없는 부담을 제거
 - 논리 메모리와 물리 메모리의 분리를 최종적으로 구현: 물리 메모리 크기에 독립적인 논리 메모리 공간

9.4 페이지 교체 (Page Replacement)

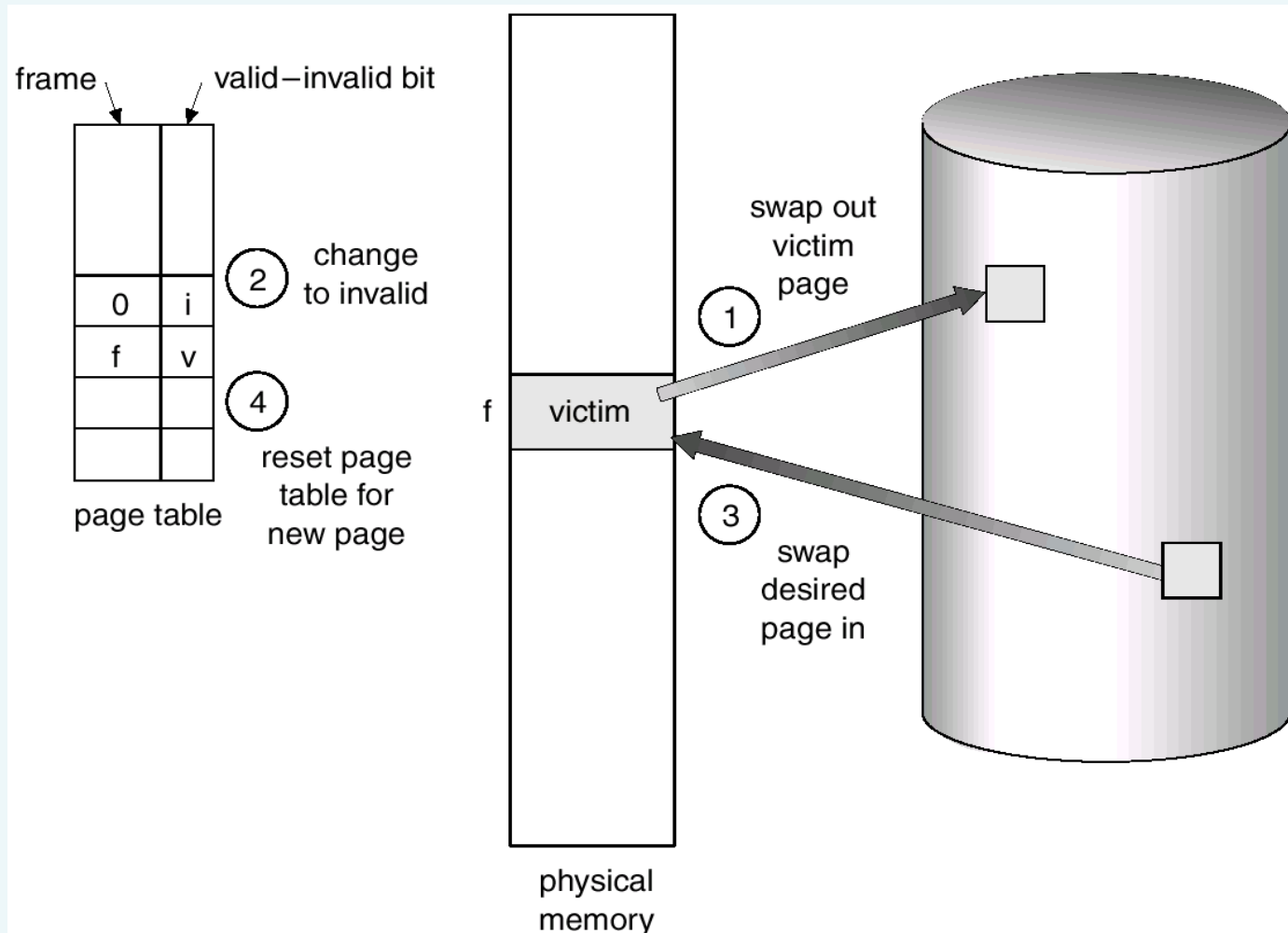
■ 페이지 교체의 필요성



페이지 교체 알고리즘 – 기본방법

1. 디스크에서 요구된 페이지의 위치를 알아낸다
 2. 비어 있는 프레임을 찾는다.
 - a. 만약, 빈 프레임이 있으면 그 프레임을 사용하고,
 - b. 그렇지 않으면 페이지 교체 알고리즘으로 희생자(victim) 페이지 선택
 - c. 희생자 페이지를 스왑 공간에 출력하고, 페이지 테이블에 기록
 3. 빈 프레임에 요구된 페이지를 불러들이고, 페이지와 프레임 테이블의 내용을 수정
 4. 사용자 프로세스를 재개
-
- 빈 프레임이 없는 경우 페이지 부재 처리 시간이 2배 소요 → 변경 비트 사용

페이지 교체 알고리즘 – 기본방법



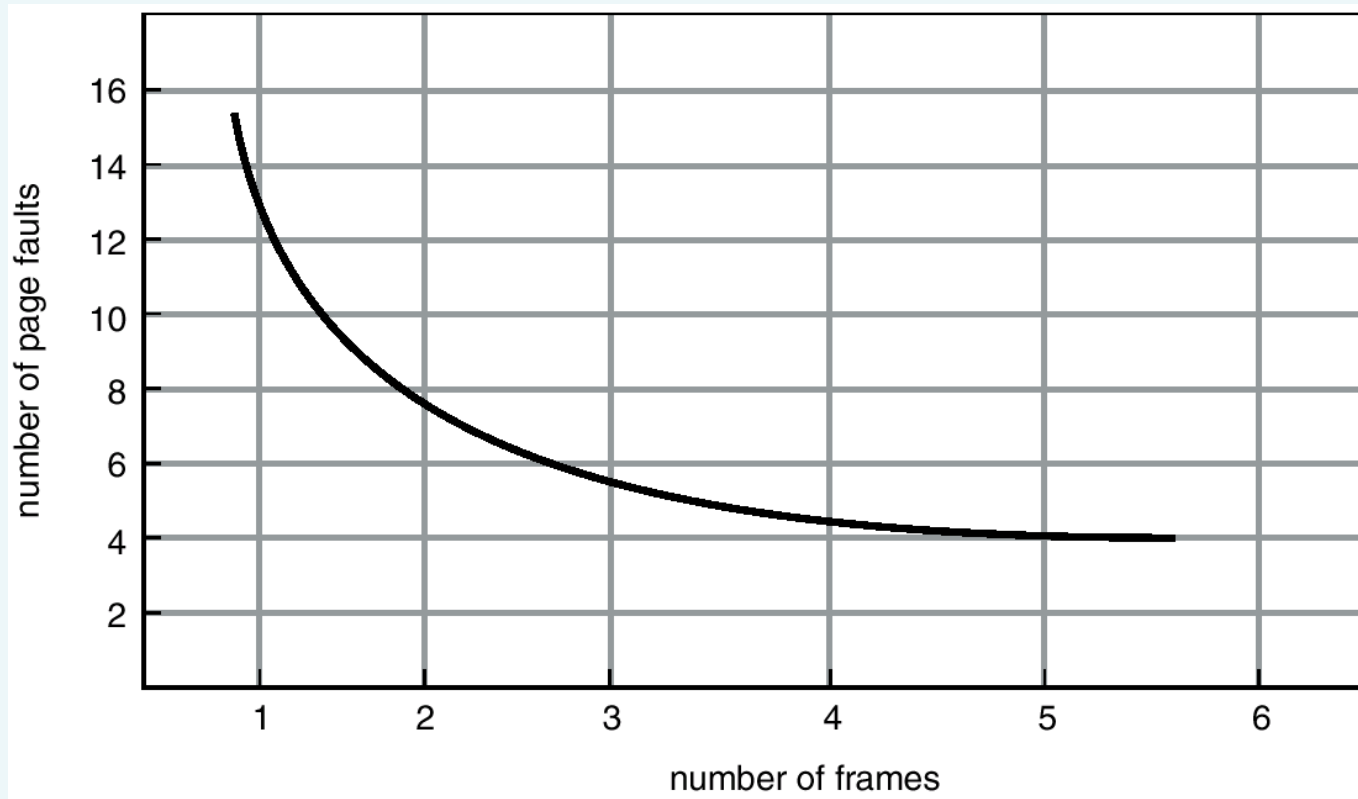
페이지 교체 알고리즘의 선택

- 페이지 교체 알고리즘
 - 페이지 폴트의 최소화가 목적임
 - 각 프로세스에게 할당할 프레임 수와, 교체할 프레임 결정
- 참조열 (reference string)
 - 페이지 부재 수를 계산하는 데 사용되는 기억장치 참조 순서
 - 예: 특정 프로세스의 참조 주소 추적
 - 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
 - 페이지 크기 = 100
 - 참조열: 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

가상 메모리의 성능

- 가상 메모리 시스템의 목적
 - 페이지 부재 오류율을 최소화
- 페이지 오류율
 - = 페이지 부재 오류의 수 / 참조열 길이
 - 페이지 교체 알고리즘 및 한 프로세스에 할당된 프레임의 수에 영향을 받는다.
 - 충분한 양의 프레임이 있으면 페이지 오류율은 ≈ 0
 - 페이지가 1개만 있으면, 페이지 오류율은 1에 가까워진다.

페이지 부재 수 대 프레임 수



FIFO 교체 알고리즘

- 가장 간단한 페이지 교체 알고리즘
 - 각 페이지에 기억 장치 안으로 들어온 시간을 연관
 - 가장 오래된 페이지가 교체되도록 선택
 - FIFO 큐 이용
- 이해하기 쉽고 프로그램 하기도 쉽다.

FIFO 교체 알고리즘

- 참조열: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 프레임

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		

- Belady의 변이 (Belady's Anomaly)
 - 할당되는 프레임의 수가 증가에 따른 오류율의 증가

FIFO 교체 알고리즘의 예

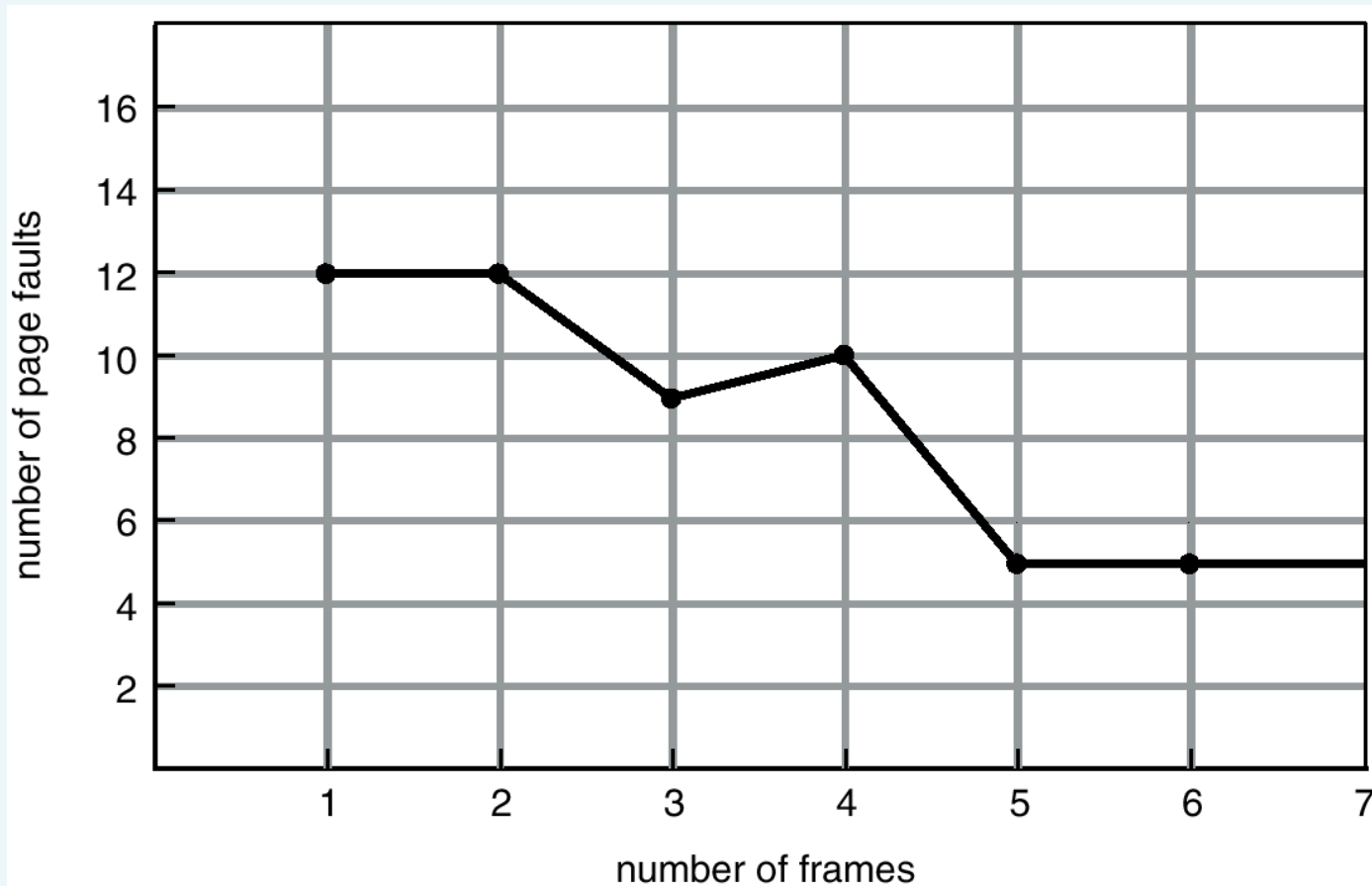
reference string

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

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

page frames

Belady의 변이 현상의 예



최적 교체 알고리즘 (OPT)

- 앞으로 가장 오래 사용되지 않을 페이지 교체
 - 모든 알고리즘 중에서 페이지 오류율이 가장 낮다.
- 예 : 4 프레임
 - 참조열: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	4
2	
3	
4	5

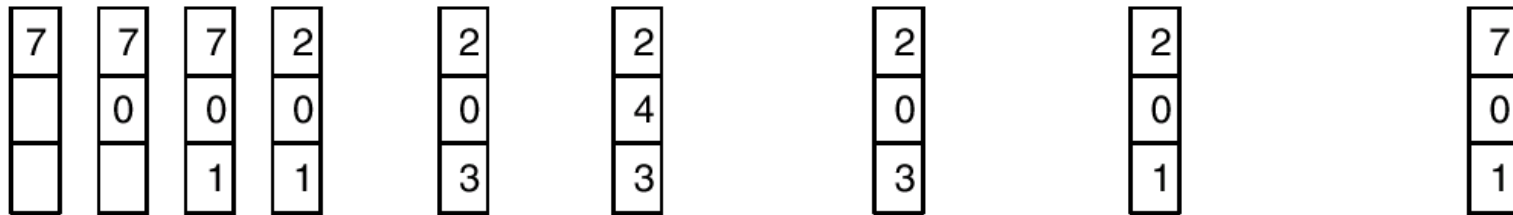
6 page faults

- 실제 구현은 불가능함
 - 교체 알고리즘 평가 기준으로 사용

최적 교체 알고리즘

reference string

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



page frames

LRU (Least Recently Used) 교체 알고리즘

- 최적 교체 알고리즘의 근사값
 - 가장 오랫동안 사용하지 않은 페이지 교체
 - 가까운 미래의 근사값으로 가장 최근의 과거를 사용
 - 최적 알고리즘과의 대칭성 때문에 성능이 유사
- LRU의 구현 방법
 - 최후의 사용 시간에 의해 정의되는 페이지 순서를 계수기 (counter) 또는 스택으로 관리

LRU 교체 알고리즘

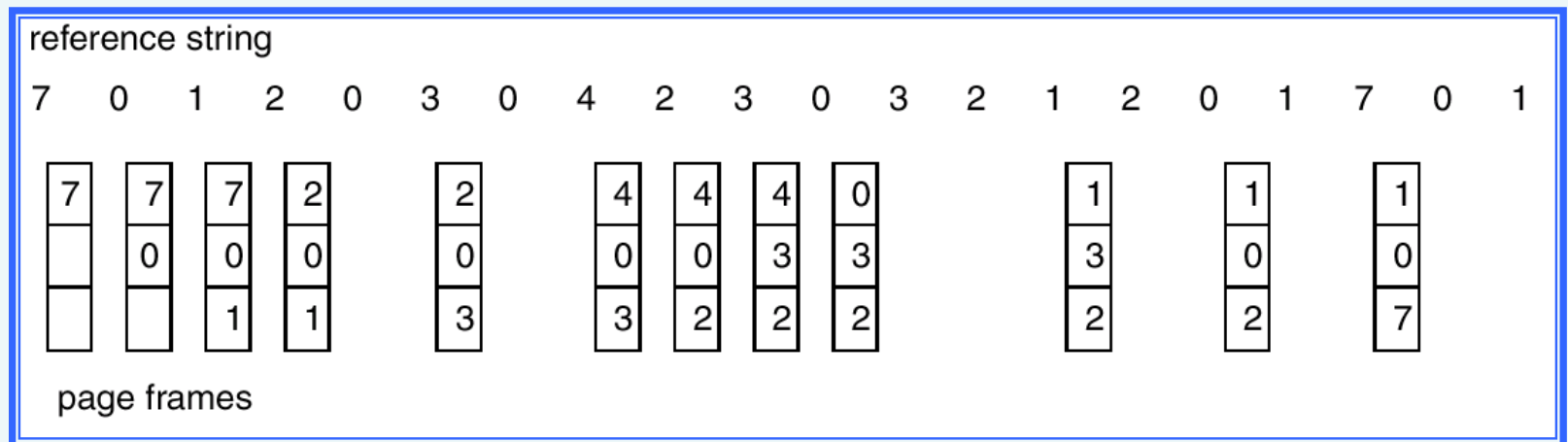
- 참조열: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 프레임

1	4	5	3
2	1		4
3	2		5

- 4 프레임

1		5	
2			
3	5	4	
4	3		

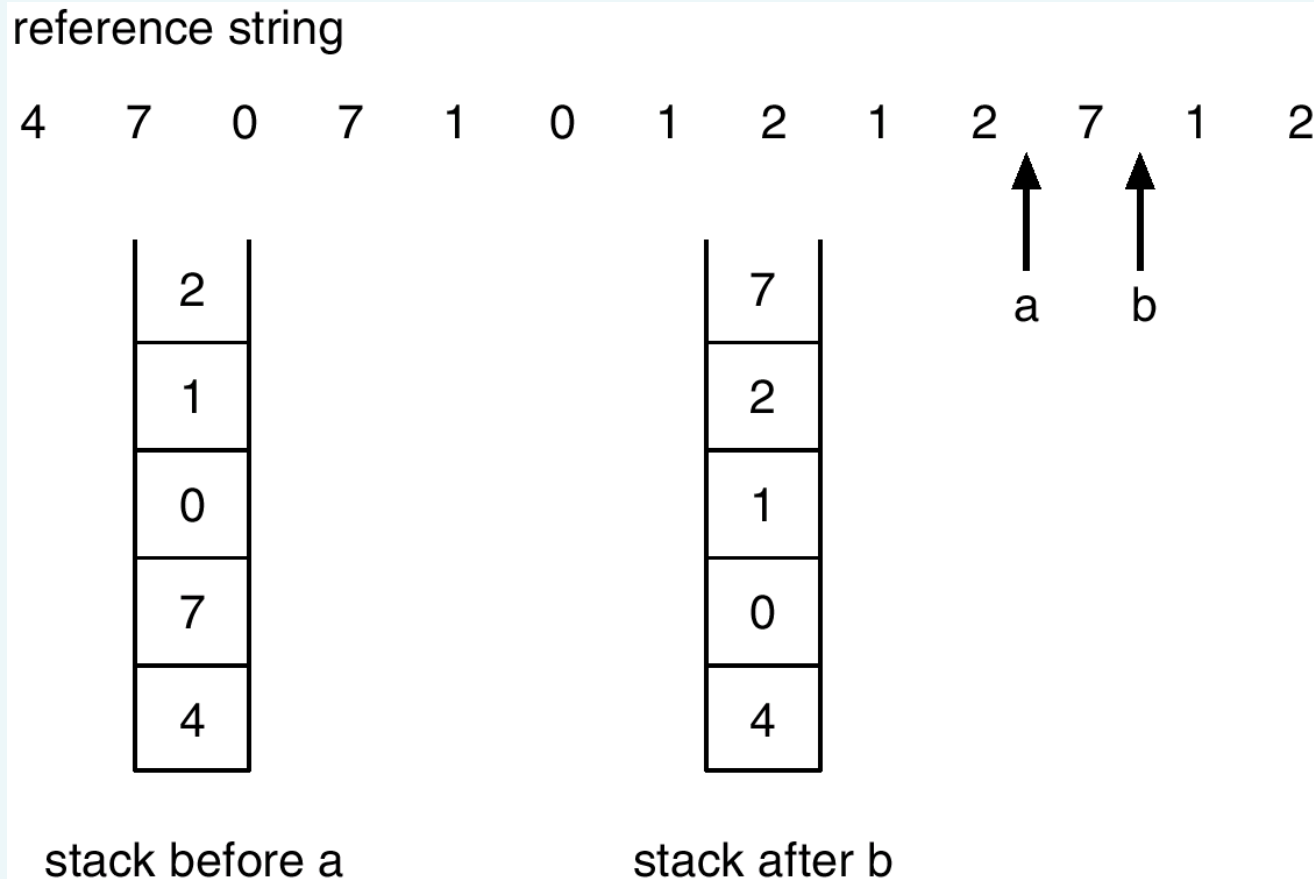
LRU 교체 알고리즘



LRU 교체 알고리즘의 구현

- 계수기의 사용
 - 각 페이지 테이블 항목마다 시간 필드를 둠
 - 페이지 에 대한 참조가 있을 때마다, 시계 레지스터의 내용을 해당 페이지 테이블 항목의 시간 필드에 복사
 - 가장 작은 시간값을 갖는 페이지를 교체
- 스택의 사용
 - 이중 연결 리스트를 사용한 페이지 번호의 스택을 유지
 - 페이지 접근시 스택 탑으로 이동
 - 오버헤드를 발생시키나 마이크로 코드로 구현 용이
- 두 방법 모두 하드웨어의 지원 필요
- 스택 알고리즘 : Belady Anomaly 가 발생하지 않는 알고리즘 ex) LRU, OPT

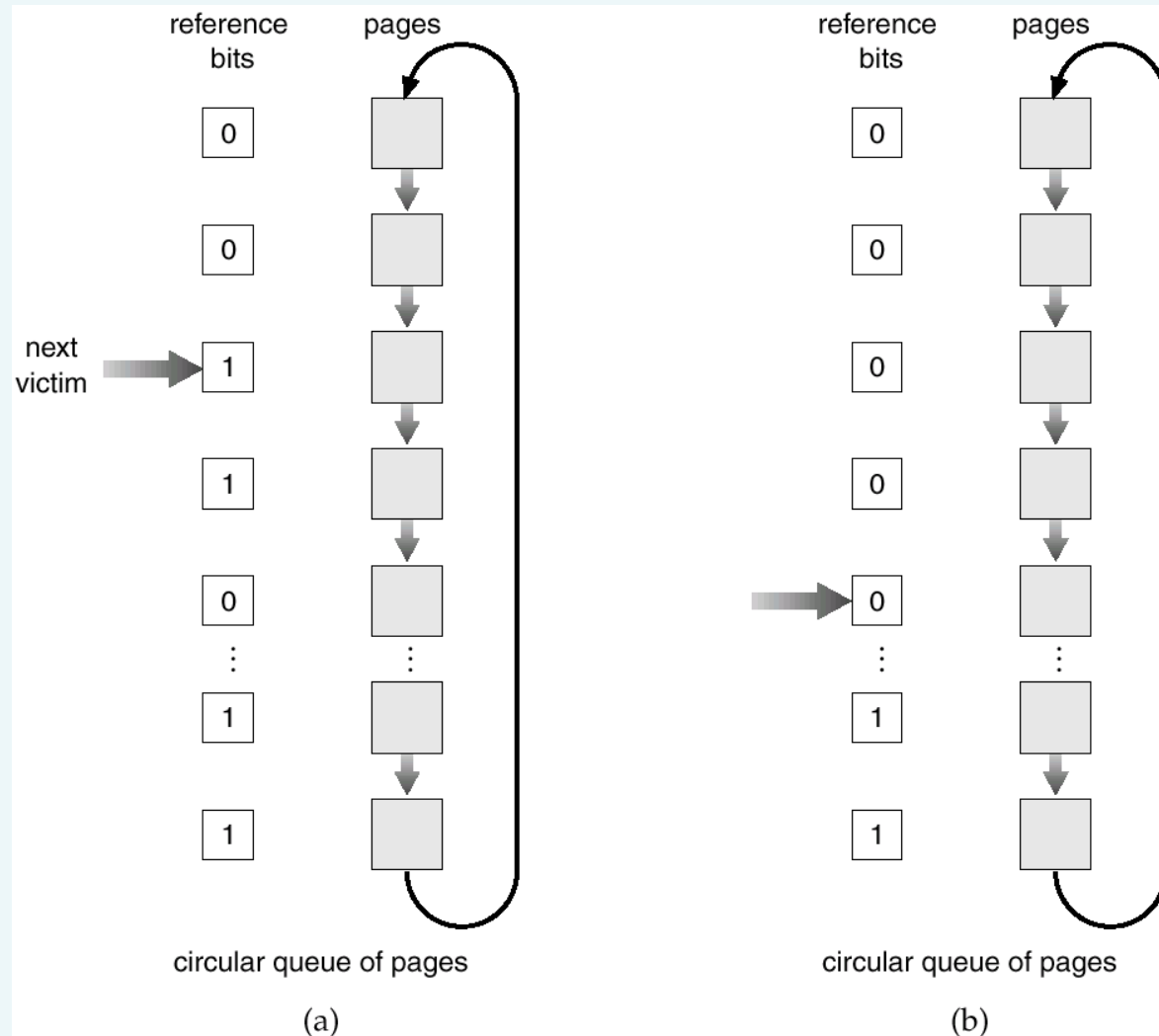
가장 최근의 페이지 참조를 기록하기 위한 스택의 사용



LRU 근접 알고리즘

- 참조 비트를 이용한 알고리즘
 - 각 페이지에 대해 8bit를 참조용으로 유지
 - 타이머 인터럽트->운영체제->8비트를 하나씩 오른쪽으로 이동시키고 참조 비트를 가장 높은 비트에 저장후 참조비트 리셋
 - 가장 낮은 번호를 가진 페이지를 교체
- 2차 기회(second-chance) 페이지 교체 알고리즘
 - 각 페이지에 대해 1 bit만 참조용으로 사용
 - 기본적으로 FIFO 교체 알고리즘 (순환 큐를 사용)
 - 참조 비트가 0이면 그 페이지를 교체
 - 참조 비트가 1이면 2차 기회를 주고 참조 비트를 지운 뒤 페이지 찾기를 계속
 - LRU에 근접한 성능을 보임

2차 기회 페이지 교체 알고리즘



2차 기회 페이지 교체 알고리즘

- 참조 비트와 오염(또는 modify) 비트 사용
 1. (0, 0) 참조되지도 변경되지도 않은 것
 2. (0, 1) 최근에 참조되지 않았으나 변경된 것
 3. (1, 0) 참조되었으나 변경되지 않은 것
 4. (1, 1) 참조되고 변경된 것
 - 등급의 번호가 낮은 페이지가 우선적으로 교체
 - Macintosh 가상메모리 시스템 에서 사용

기타 알고리즘

- LIFO
- LFU (least frequently used) 페이지 교체 알고리즘
 - 각 페이지마다 참조 횟수에 대한 **counter**를 두어 가장 작은 수를 가진 페이지를 교체
 - 초기에 많이 참조되고 이후 사용되지 않는 페이지?
→ 일정시간마다 **right shift** 수행
- MFU (most frequently used) 페이지 교체 알고리즘
 - 가장 작은 수의 페이지가 방금 들어온 것이어서 앞으로 더 사용될 것이라는 판단에 근거
- 페이지 버퍼링 알고리즘
 - 가용 프레임의 풀을 유지: 희생 페이지의 저장 전에 페이지 할당 가능
 - 변경된 페이지의 리스트를 유지 → 수시로 미리 저장해 놓음
 - 가용 프레임의 원래 페이지를 기억

9.5 프레임의 할당

- 일정한 사용 가능 메모리를 여러 프로세스에게 어떻게 할당할 것인가?
 - 최대한 가능한 메모리를 요구대로 할당?
 - 프로세스당 최소 프레임 수의 설정?
- 최소 프레임 수
 - 프로세스는 최소의 필수적으로 필요한 프레임의 수를 가짐
 - 명령어 집합 (instruction set)의 구조와 컴퓨터 구조에 의해 정의
 - PDP-11 : two byte for instruction + 2 indeirect addressing → 6 bytes
 - Infinite indirect addressing → 전체 프레임

할당 알고리즘

■ 고정 할당

— 균등 할당 (equal allocation)

- 모든 프로세스에 똑같이 m/n 개의 프레임을 할당

— 비례 할당 (proportional allocation)

- 프로세스의 크기(사용하는 페이지의 수)에 비례하여 각 프로세스에 프레임 할당

s_i = size of process p_i

$$S = \sum s_i$$

m = total number of frames

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

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \cdot 62 \gg 4$$

$$a_2 = \frac{127}{137} \cdot 62 \gg 57$$

할당 알고리즘

- 우선 순위의 고려
 - 우선 순위가 높은 프로세스에 큰 기억 장소를 할당
 - 우선 순위가 높은 프로세스가 교체할 프레임을 우선 순위가 낮은 프로세스에 할당된 프레임 중에서 선택

전역(Global) 대 지역(Local) 할당


- 전역 교체(global replacement)
 - 프로세스는 모든 프레임 집합으로부터 교체 프레임을 선택 → 다른 프로세스로부터 프레임을 뺏을 수 있다.
 - 우선 순위가 높은 프로세스가 낮은 프로세스의 프레임을 가져올 수 있음
 - 외부 요건에 의하여 실행 시간이 달라질 수 있음
- 지역 교체(local replacement)
 - 각 프로세스는 자신의 할당된 프레임 중에서 교체 프레임을 선택
 - 전역적인 면에서 프레임 낭비 발생 가능

Non-Uniform Memory Access

- 여러 개의 멀티프로세서 보드로 이루어진 시스템
 - 메모리 접근 시간이 일정하지 않음 (**NUMA**)
- 스레드가 실행하는 **CPU**에 가까운 프레임 할당 필요
- 스케줄러 고려사항: 같은 보드에서 수행되었든 스레드를 우선 배정
- Solaris: lgroups
 - CPU / Memory 지연시간 그룹
 - 한 프로세스의 스레드는 하나의 lgroup안에 할당 시도

9.6 쓰래싱 (Thrashing)

■ 쓰래싱

- 프로세스가 실제로 동시에 사용하는 프레임 수만큼의 프레임을 가지지 못해 계속적으로 페이지 부재 오류가 발생하여 페이지 교체가 계속 발생 
- 프로세스 실행 시간보다 페이지 교체 시간이 더 커짐

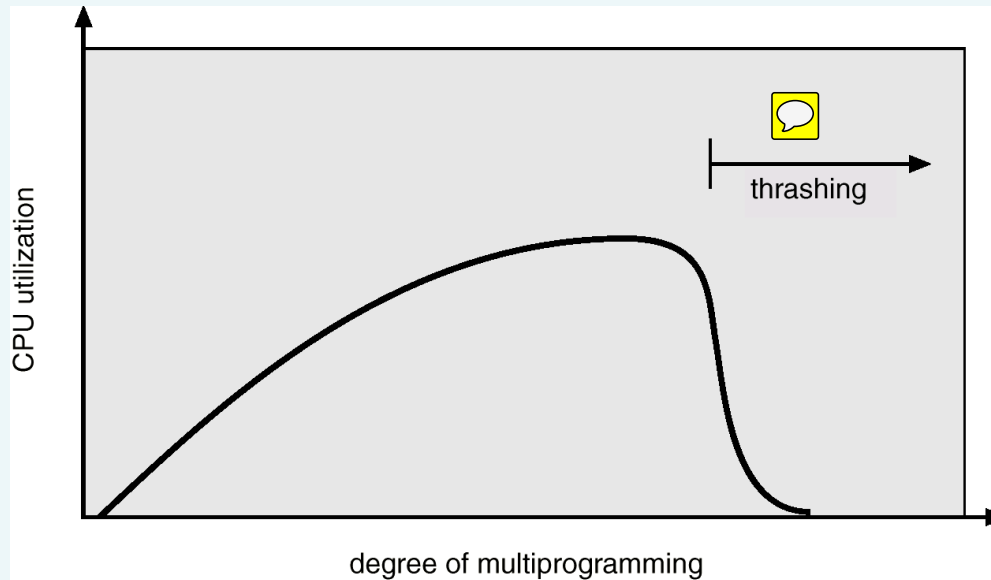
■ 쓰래싱의 원인

- 운영 체제는 **CPU** 이용률이 너무 낮으면 새로운 프로세스를 도입해서 다중 프로그래밍의 정도를 높인다.
- 프로세스가 페이지 오류를 발생하여 다른 프로세스들의 페이지를 내보내고 이들이 다시 페이지 부재 발생 반복
- 각 프로세스는 페이지징 처리 장치에서 대기
- **CPU** 이용률 저하 → 다중 프로그래밍의 정도를 높임

■ 쓰래싱 효과의 방지

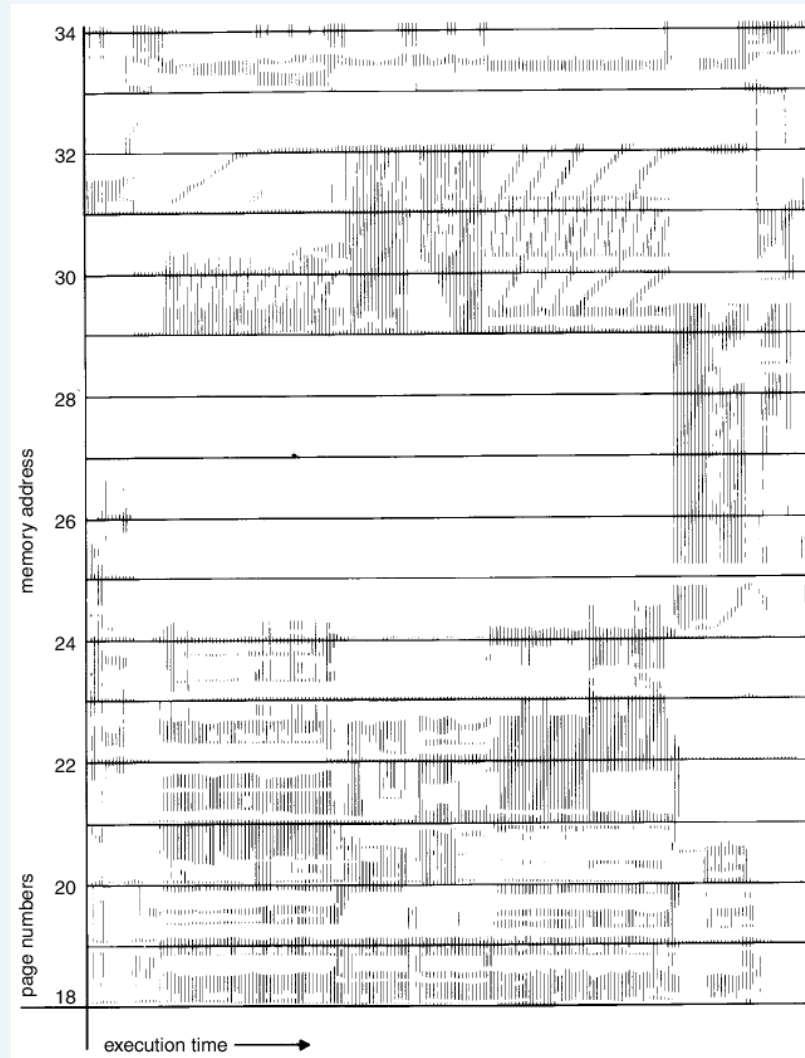
- 각 프로세스가 필요한 충분한 프레임 제공

쓰레싱



- 지역성 모델(Locality Model)
 - 프로그램 수행에 따라, 한 페이지 지역에서 다른 지역으로 이동
 - 지역성 = 실제로 함께 사용되는 페이지의 집합
- 쓰레싱의 발생 상황
 - Σ size of locality \rightarrow total memory size

메모리 참조 패턴의 지역성

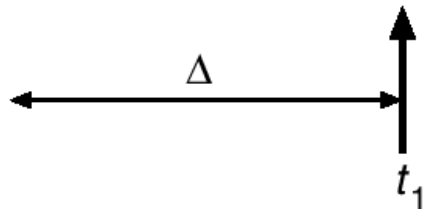


작업 세트(Working Set) 모델

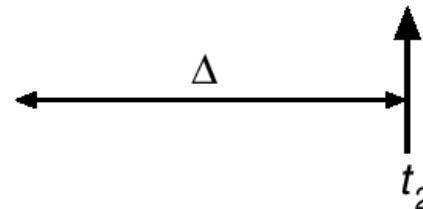
- 작업 세트 : 프로그램 페이지 접근 지역성의 근사값
- 페이지 참조의 추적

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 ...



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



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

- 가장 최근에 Δ 페이지 참조 조사
- Δ 가 너무 작으면 전체 작업 세트를 망라하지 못함
- Δ 가 너무 크면 여러 지역성 겹침
- $D = \sum WSS_i \equiv \text{total demand frames}$

작업 세트 모델

■ 사용법

- 운영 체제는 각 프로세스의 작업 세트를 감시하여 작업 세트 크기에 맞는 충분한 프레임 할당
- 작업 세트 크기의 합이 증가하여 총 유효 프레임 수보다 커지면 한 프로세스를 중지

■ 작업 세트의 추적

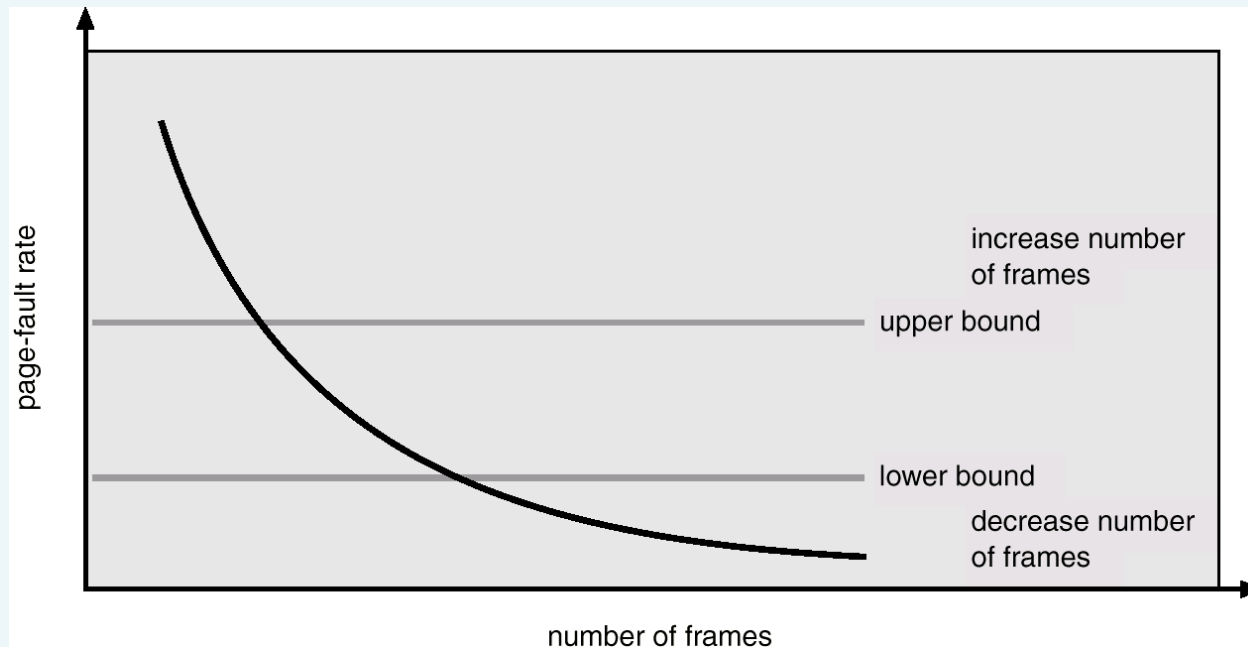
- 메모리 참조 시마다 계산되므로 많은 부담을 요구
- 일정 시간 간격의 타이머 인터럽트와 참조 비트를 사용하여 근사 가능
 - 일정 시간마다 참조 비트를 0으로 설정함으로써 최근의 일정 시간 동안의 작업 세트 크기를 알 수 있음

Working Set의 근사


- interval timer + a reference bit
- $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupts copy and sets the values of all reference bits to 0
 - If one of the bits in memory = 1 \Rightarrow page in working set (referenced in 10000~15000 references)
- 완전히 정확하지는 않음 (Why?)
 - Improvement = 10 bits and interrupt every 1000 time units

페이지 부재 빈도 (Page-Fault Frequency, PFF)

- 페이지 부재율 조절 - 쓰레싱 방지
 - 페이지 부재율의 상한과 하한을 정해, 상한을 넘으면 프레임의 수를 늘리고 하한보다 낮으면 프레임의 수를 줄인다



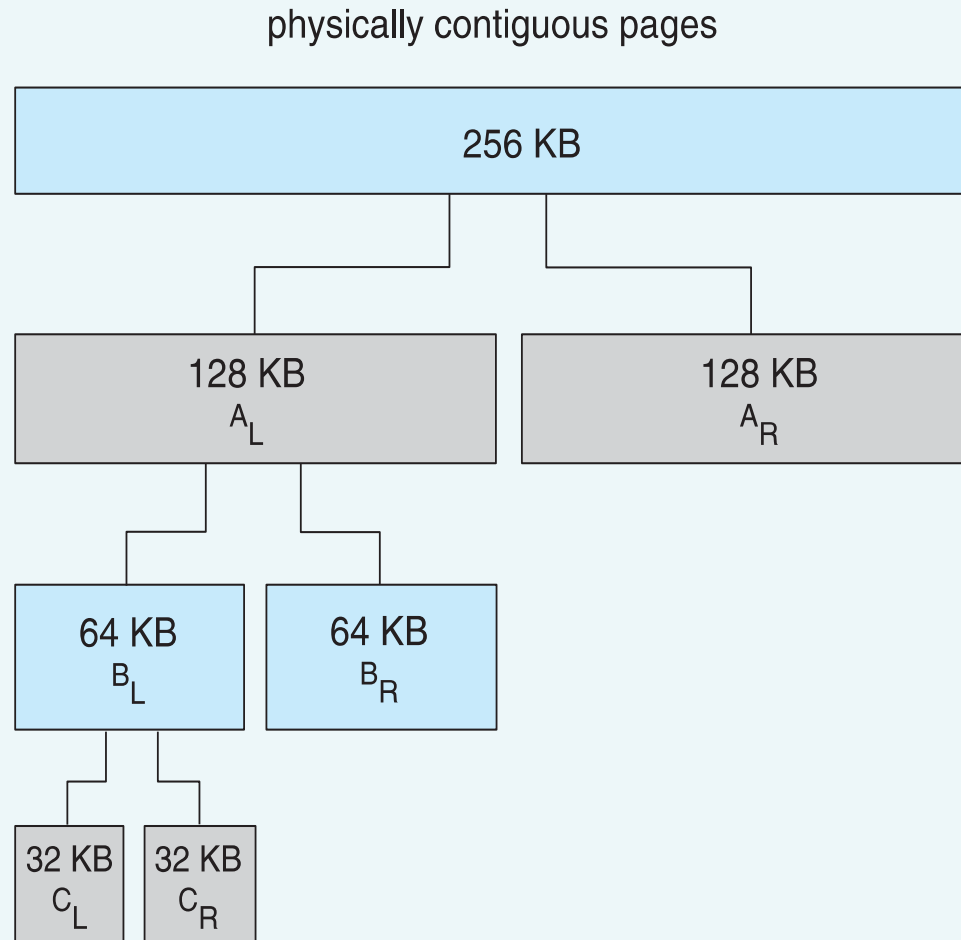
9.8 커널 메모리의 할당

- 사용자 메모리 할당을 위한 여유 메모리 구조를 공유하지 않음
- 별도의 메모리 풀 사용 
 - 다양한 크기의 자료구조를 위한 메모리 필요
 - 실제 프레임에서 연속적인 공간 필요 가능
 - I.e. for device I/O

Buddy System

- 물리적으로 **연속된 페이지**로 이루어진 고정 크기 세그먼트로부터 메모리 할당
- 2의 지수승 크기 메모리 할당
 - 필요한 메모리보다 큰 최소의 2의 지수승 크기 할당
 - 더 작은 메모리가 필요할 경우 반복하여 2개씩 분할하여 최적의 메모리 크기 생성
 - 예: 256KB 메모리 사용 가능 / 21KB 필요
 - A_L and A_R : 128KB each $\rightarrow B_L$ and B_R : 64KB each $\rightarrow C_L$ and C_R : 32KB
 - 사용하지 않는 덩어리를 쉽게 합침(**coalesce**)
- 단점: fragmentation

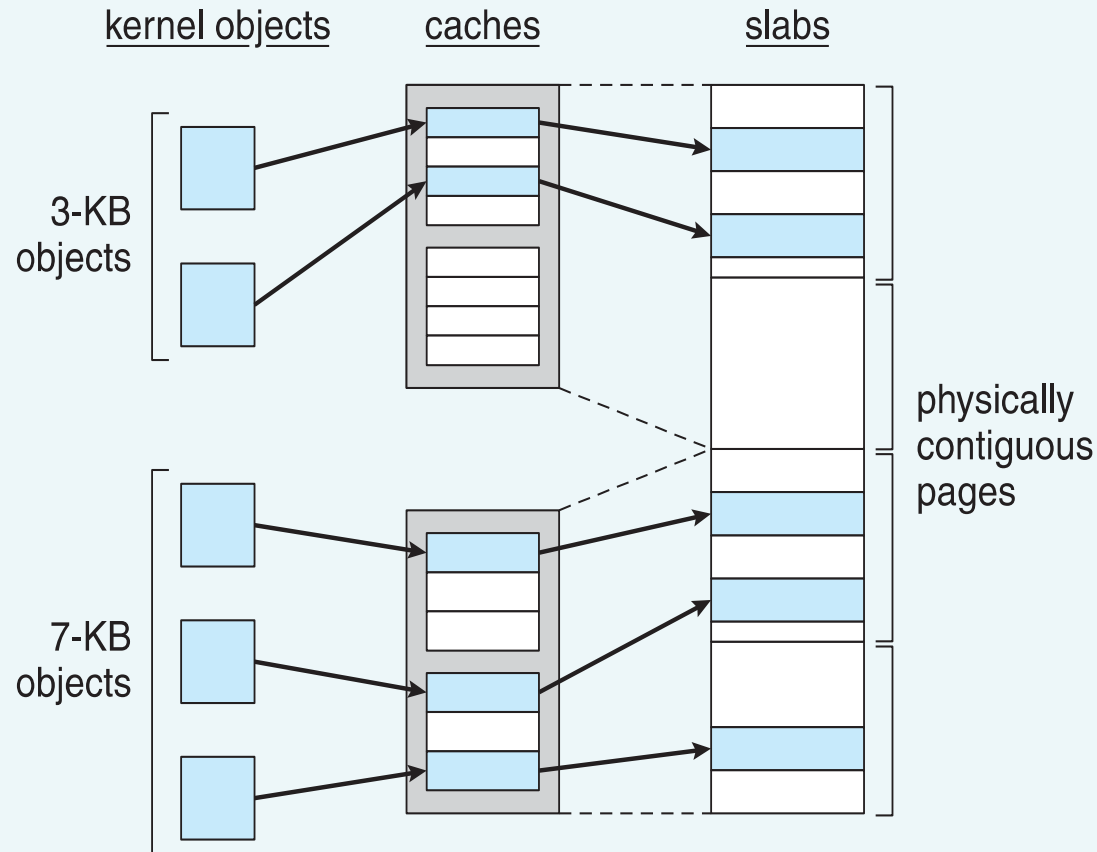
Buddy System Allocator



Slab Allocator

- Slab은 하나 이상의 연속된 페이지로 구성
- Cache는 하나 이상의 슬랩으로 구성
- 각 커널 자료구조 종류마다 하나의 캐쉬 운영
 - 각각의 캐쉬 내의 슬랩에 커널 자료구조의 실체를 저장: objects
- 캐쉬 생성시 각각의 비어있는 객체로 채움
- 자료구조 저장시 객체를 사용으로 표시
- 슬랩이 가득 차면 다음 객체는 비어있는 슬랩에서 할당 (필요시 슬랩 새로 할당)
- 장점
 - no fragmentation
 - 메모리(자료구조) 요청 처리 속도 향상

Slab Allocation




Slab Allocator in Linux


- 예: process descriptor
 - `Struct task_struct`
 - Approx 1.7KB of memory
 - New task -> allocate new struct from cache
 - Will use existing free `struct task_struct`
- Slab can be in three possible states
 1. Full – all used
 2. Empty – all free
 3. Partial – mix of free and used
- Upon request, slab allocator
 1. Uses free struct in partial slab
 2. If none, takes one from empty slab
 3. If no empty slab, create new empty

9.9 기타 고려사항

■ 프리 페이징 (pre-paging)

- 프로세스 시작시 또는 **swap out** 이후 재 시작시의 프로세스 시작시의 과도한 페이지 부재를 방지
- 관련 페이지를 사전에 메모리로 가져옴
 - Working set을 기억 후 **swap in** 시에 가져옴 
 - 가져오는 페이지의 주변 페이지를 가져옴 (**clustering**)

■ 페이지 크기

- 내부 단편화와 테이블 크기의 부담을 고려
- 디스크 전송시간
 - 회전 지연과 탐색 지연은 페이지가 클수록 유리 
 - CPU와 디스크의 속도차가 증가하면 페이지 크기 증가 경향

기타 고려사항

■ TLB Reach

- TLB 로부터 접근 가능한 메모리 양.
- $TLB\ Reach = (TLB\ Size) \times (Page\ Size)$
- 이상적으로, 각 프로세스의 작업 세트는 TLB 에 저장. 그렇지 않으면 페이징 효율 저하

■ 역페이지 테이블

- 참조 페이지가 부재시에 해당 페이지를 찾기 위하여 추가적인 확장된 페이지 테이블이 필요
- 확장된 페이지 테이블은 드물게 참조됨 → 그 자체가 페이징 가능

기타 고려사항

- 프로그램 구조 (Program structure)

- **int A[][] = new int[1024][1024];**

- 각 행은 한 페이지에 저장

- Program 1

```
for (j = 0; j < A.length; j++)  
    for (i = 0; i < A.length; i++)  
        A[i,j] = 0;
```


→ 1024 x 1024 page faults

- Program 2 

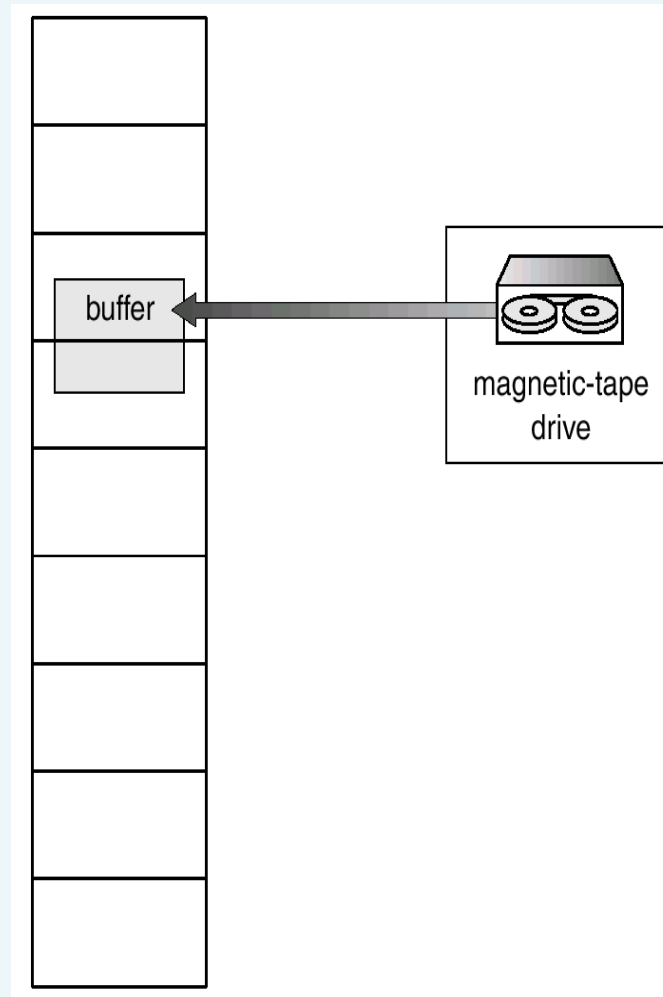
```
for (i = 0; i < A.length; i++)  
    for (j = 0; j < A.length; j++)  
        A[i,j] = 0;
```

→ 1024 page faults

기타 고려사항

- **I/O의 대상이 되는 페이지의 pageout**
 - 해결책
 - 커널 공간 사용 → 추가적인 메모리 복사 필요
 - **입/출력 상호잠금 (I/O Interlock)** – 페이지 교체 대상에서 제외
 - 커널은 대부분 메모리에서 잠금이 걸림
- 메모리 잠금의 다른 사용
 - **한번도 읽어지지 않은 페이지의 잠금**
- 실시간 처리 
 - 실시간 시스템은 가상메모리를 사용하지 않음
 - Solaris
 - 중요 페이지 정보를 프로세스가 줄 수 있음
 - Privileged user에게 특정 페이지의 lock을 허용

입/출력용 프레임이 메모리 내에 존재해야 하는 이유



9.10 운영체제의 예

- Windows
- Solaris 2

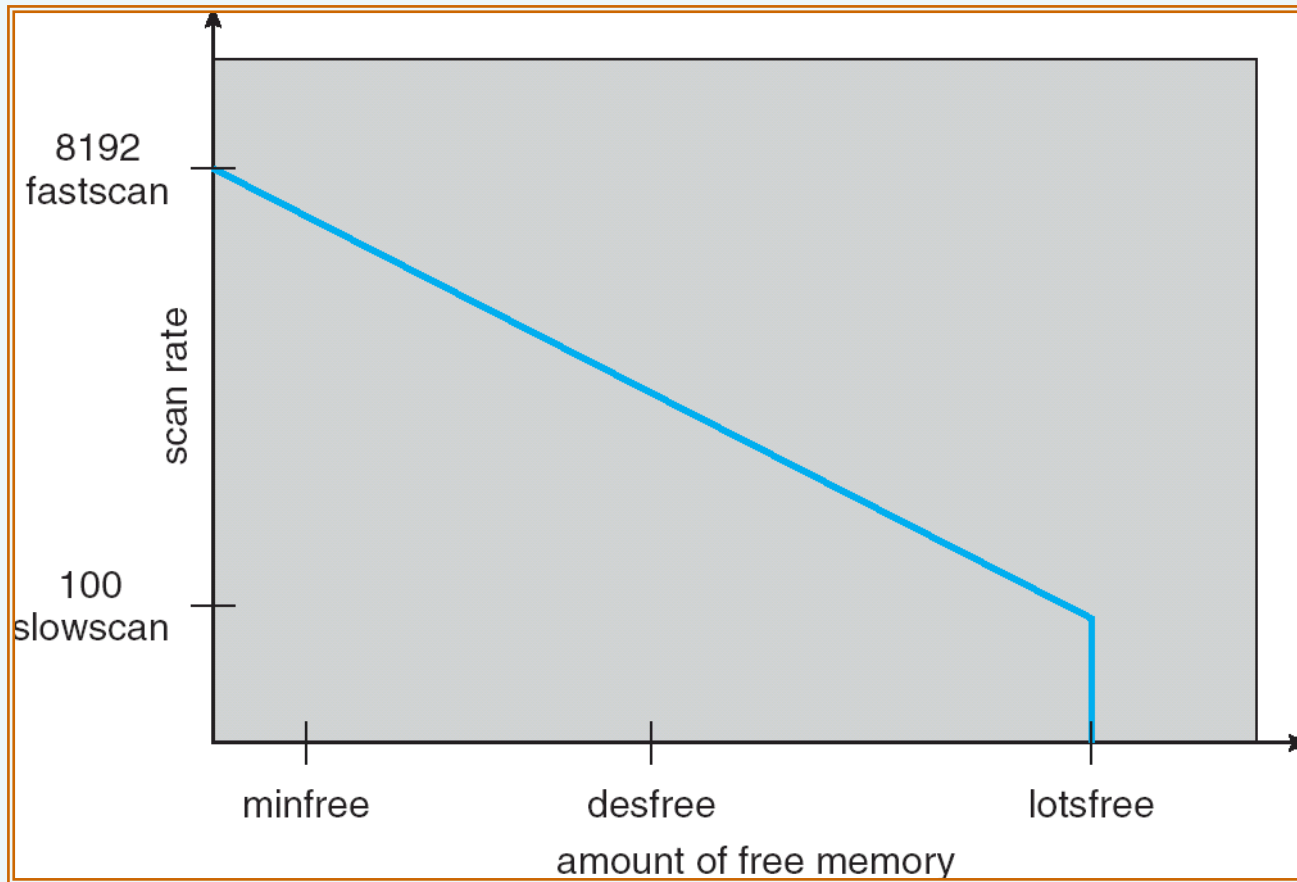
Windows

- 요구 페이징 : clustering (주변 페이지도 같이 가져옴)
- 프로세스의 작업집합(**working set**)의 하한과 상한 설정
 - 작업 집합 하한 : 보장된 최소 할당 페이지수
 - 작업 집합 상한 : 최대 할당 페이지수
- Working set 자동 조절
 - 가용 공간이 임계치보다 낮아졌을 때 공간 확보
 - 하한치보다 많은 공간을 가지고 있는 프로세스들의 페이지들을 회수 (FIFO)

Solaris

- 가용 페이지 관리
 - *Lotsfree* – 최소 확보 가용 페이지수 (1/64)
 - *Desfree* – *Lotsfree*의 1/2
 - *Minfree* – *Desfree*의 1/2
- *pageout* 프로세스
 - Pageout scans pages using modified clock algorithm
 - Front hand : reference 비트를 0으로 설정
 - Second hand : reference 비트가 0인 것을 free-list에 추가
 - *Scanrate* : 초당 페이지 스캔 속도
 - Slowscan(100)~Fastscan(8192)
 - 매초 4차례 검사하여 가용 페이지가 *Lotsfree*보다 적어지면 *pageout* 프로세스 가동)
 - 가용 페이지가 *DesFree*보다 작아지면 초당 100회 검사 수행
- 가용 페이지가 *Minfree*보다 작아지면 *swapping* 시작

Solaris 2



연습문제 과제

- 문항: 9.2, 9.6, 9.8, 9.9, 9.16
- Due date: 12/9 (수업시간 전 교탁에 제출)
- 손으로 써서 제출할 것