LG전자 BS팀

『3일차』: 오후

◈ 훈련과정명 : OS 기본

◈ 훈 련 기 간 : 2023.05.30-2[illegible]3.06.02

HS글로벌

목차

『9과목』

5-6교시 : Virtual Memory

Syllabus

학습목표

- 이 워크샵에서는 메모리 하드웨어를 구성하는 다양한 방법에 대한 자세한 설명 제공을 할 수 있다.
- 가상 메모리를 정의하고 이점을 설명을 할 수 있다.
- 요청 페이징을 사용하여 페이지를 메모리에 로드하는 방법을 보여을 할 수 있다.
- FIFO, 최적 및 LRU 페이지 교체 알고리즘을 적용을 할 수 있다.
- 프로세스의 작업 세트를 설명하고 이것이 프로그램 지역성과 어떻게 관련되는지 설명을 할 수 있다.
- Linux, Windows 10 및 Solaris가 가상 메모리를 관리하는 방법을 설명을 할 수 있다.
- C 프로그래밍 언어로 가상 메모리 관리자 시뮬레이션을 설계을 할 수 있다.

Syllabus

눈높이 체크

- 요구 페이징을 알고 계신가요?
- 기록 중 복사을 알고 계신가요?
- 페이지 교체을 알고 계신가요?
- 프레임 할당을 알고 계신가요?
- Thrashing을 알고 계신가요?
- 메모리 매핑된 파일을 알고 계신가요?
- 커널 메모리 할당을 알고 계신가요?
- 기타 고려 사항

1. Background

기본 개념

- 코드를 실행하려면 메모리에 있어야 하지만 전체 프로그램이 거의 사용되지 않음
  - 오류 코드, 비정상적인 루틴, 대규모 데이터 구조
- 전체 프로그램 코드가 동시에 필요하지 않음
- 부분적으로 로드된 프로그램 실행 능력 고려
  - 더 이상 물리적 메모리 제한에 의해 제한되지 않는 프로그램
  - 각 프로그램은 실행하는 동안 더 적은 메모리를 사용합니다 -> 더 많은 프로그램이 동시에 실행.
  - 응답 시간이나 처리 시간의 증가 없이 CPU 활용도 및 처리량 증가
  - 프로그램을 메모리에 로드하거나 교체하는 데 필요한 I/O 감소 -> 각 사용자 프로그램이 더 빠르게 실행

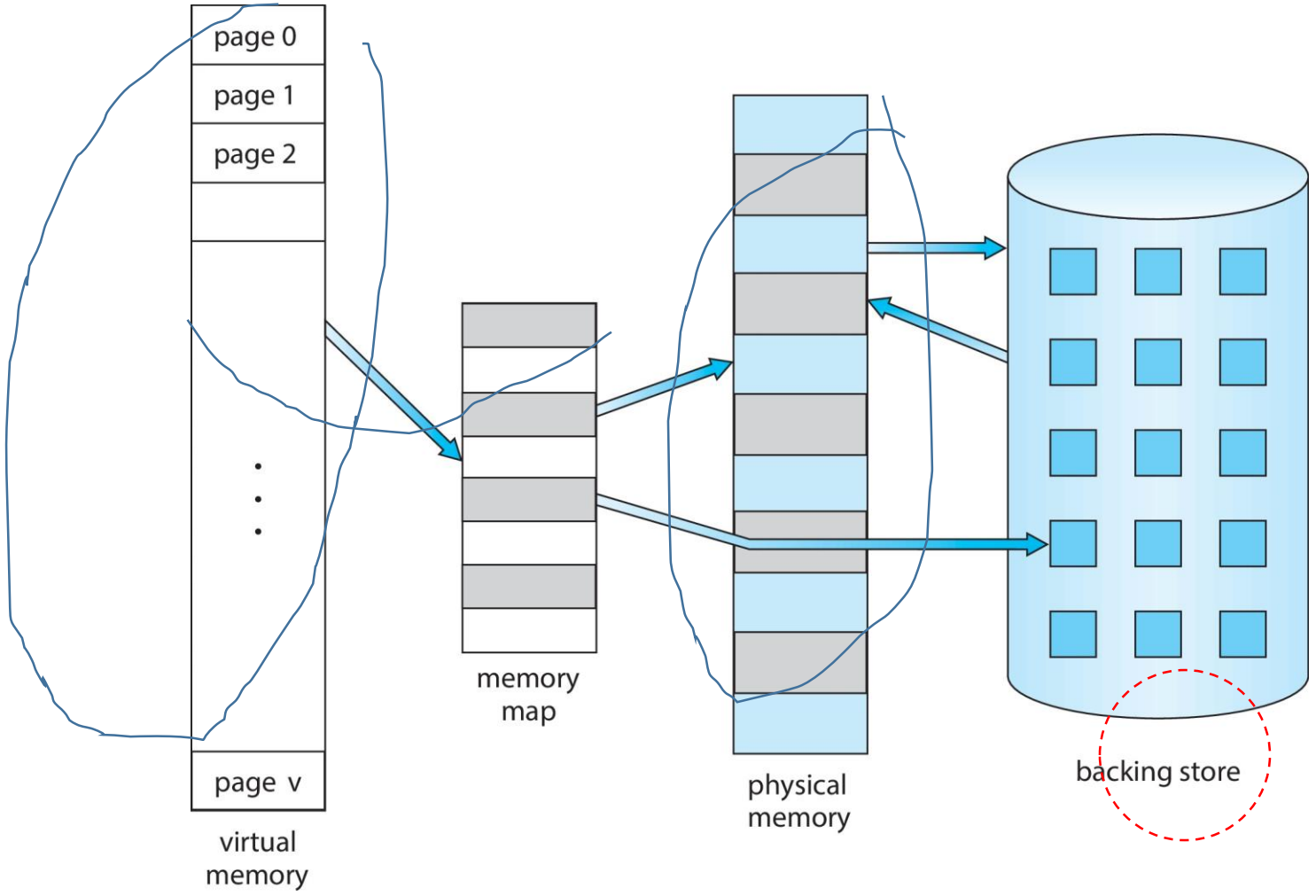
1. Background

Virtual memory

- 가상 메모리 – 사용자 논리 메모리와 물리적 메모리 분리
  - 실행을 위해 프로그램의 일부만 메모리에 있어야 함
  - 따라서 논리적 주소 공간은 물리적 주소 공간보다 훨씬 클 수 있다.
  - 주소 공간을 여러 프로세스에서 공유할 수 있다.
  - 보다 효율적인 프로세스 생성 가능
  - 동시에 실행되는 더 많은 프로그램
  - 프로세스를 로드하거나 교체하는 데 필요한 I/O 감소
- 가상 주소 공간 – 프로세스가 메모리에 저장되는 방법에 대한 논리적 보기
  - 일반적으로 주소 0에서 시작하여 공간이 끝날 때까지 연속 주소
  - 한편 페이지 프레임으로 구성된 물리적 메모리
  - MMU는 논리를 물리에 매핑해야 함
- 가상 메모리는 다음을 통해 구현할 수 있다.
  - 페이징 요청Demand paging
  - 수요 세분화Demand segmentation

1. Background

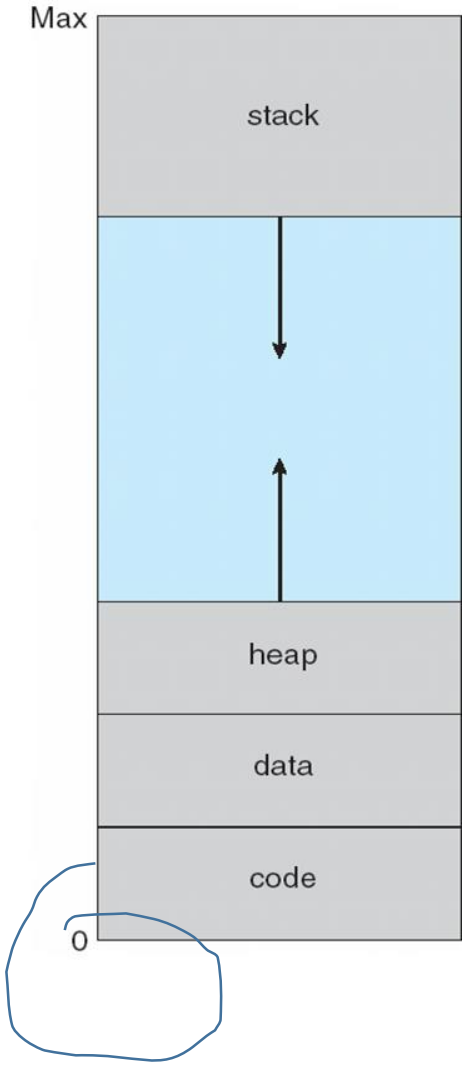
물리적 메모리보다 큰 가상 메모리



1. Background

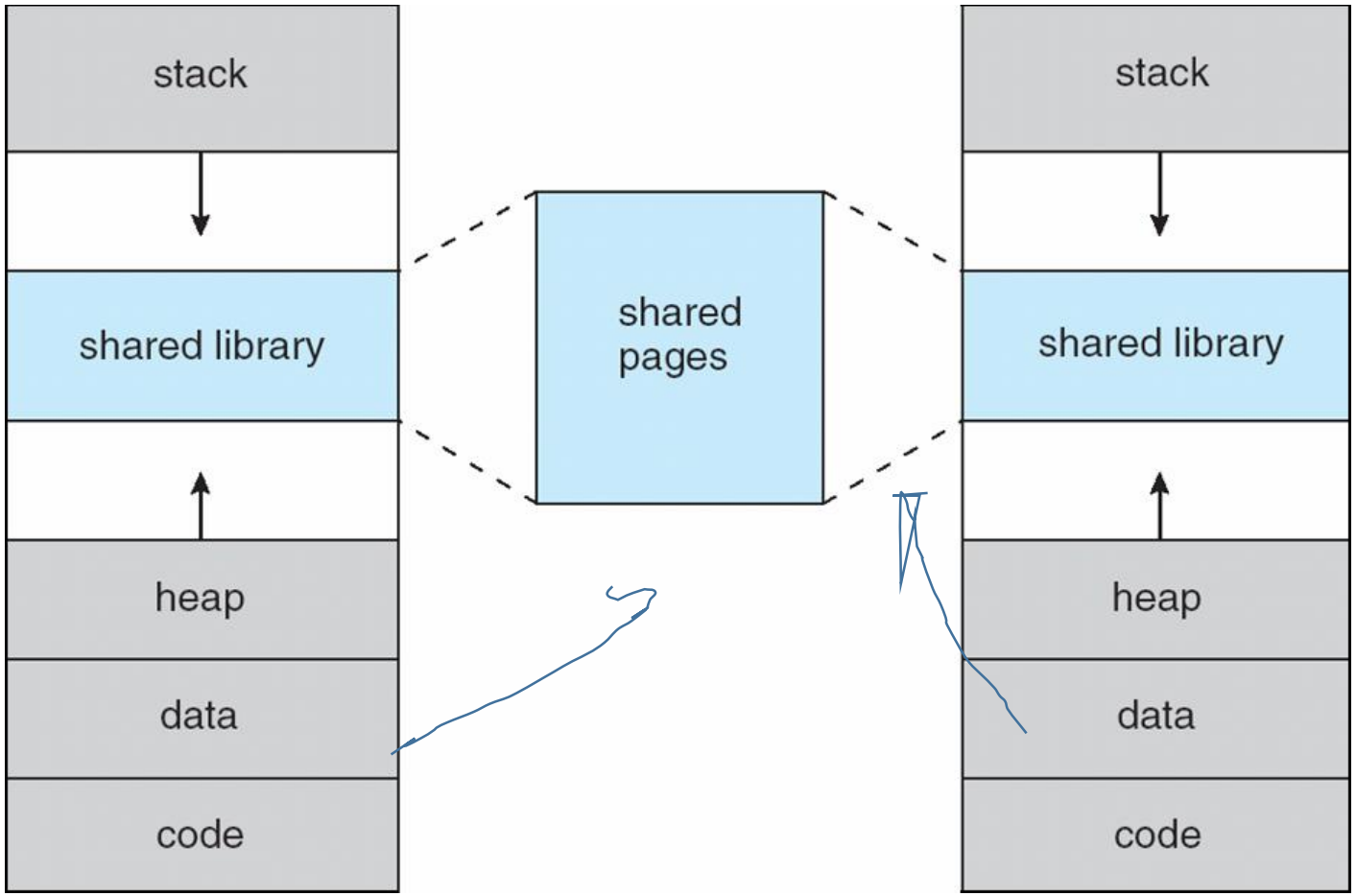
Virtual-address Space

- 일반적으로 최대 논리 주소에서 시작하여 힙이 "위로" 커지는 동안 스택이 "아래로" 커지도록 논리 주소 공간을 설계.
  - 주소 공간 사용 극대화
  - 둘 사이의 사용되지 않은 주소 공간은 **hole**.
  - 힙 또는 스택이 지정된 새 페이지로 커질 때까지 물리적 메모리가 필요하지 않다.
- 성장을 위해 남겨진 **hole**, 동적으로 연결된 라이브러리 등이 있는 희소 주소 공간을 활성화.
- 가상 주소 공간으로의 매핑을 통해 공유되는 시스템 라이브러리
- 페이지 읽기-쓰기를 가상 주소 공간으로 매핑하여 공유 메모리
- **fork()** 중에 페이지를 공유할 수 있으므로 프로세스 생성 속도가 빨라진다.



1. Background

가상 메모리를 사용하는 공유 라이브러리



2. Demand Paging

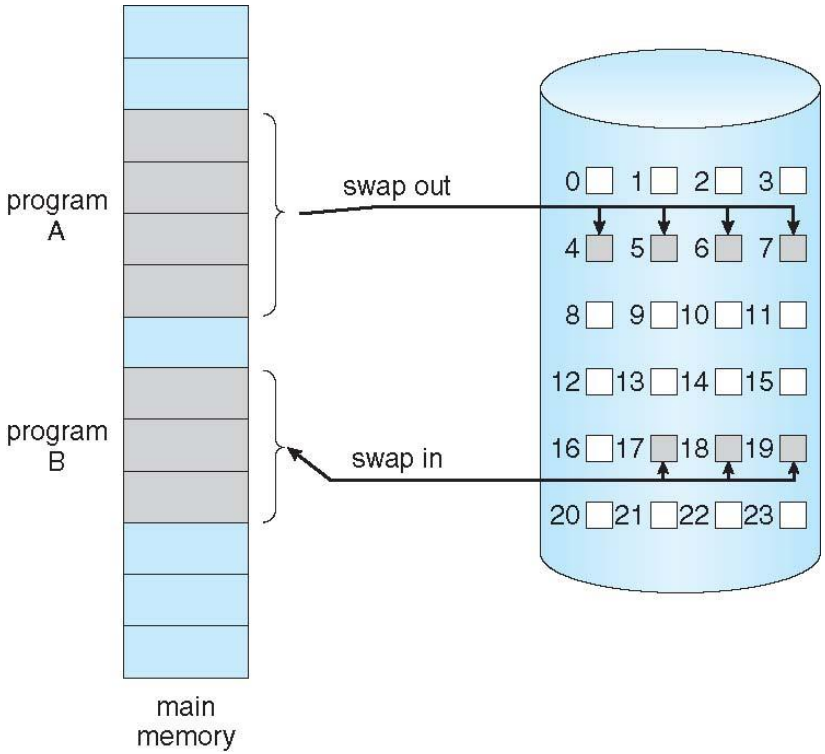
기본 개념

- 로드 시간에 전체 프로세스를 메모리로 가져올 수 있다.
- 또는 필요할 때만 페이지를 메모리로 가져온다.
  - 필요한 I/O 감소, 불필요한 I/O 없음
  - 더 적은 메모리 필요
  - 더 빠른 응답
  - 더 많은 사용자
- 스와핑이 있는 페이징 시스템과 유사(오른쪽 다이어그램)
- 페이지가 필요 ⇒ 참조
  - 잘못된 참조 ⇒ 중단
  - not-in-memory ⇒ 메모리로 가져오기
- 게으른 스와퍼 Lazy swapper – 페이지가 필요하지 않는 한 페이지를 메모리로 스왑하지 않습니다.
- 페이지를 다루는 스와퍼는 호출기pager

2. Demand Paging

기본 개념

- 로드 시간에 전체 프로세스를 메모리로 가져올 수 있다.
- 또는 필요할 때만 페이지를 메모리로 가져오시오.
  - 필요한 I/O 감소, 불필요한 I/O 없음
  - 더 적은 메모리 필요
  - 더 빠른 응답
  - 더 많은 사용자
- 스와핑이 있는 페이징 시스템과 유사(오른쪽 다이어그램)



2. Demand Paging

기본 개념

- 스와핑을 사용하면 페이저는 다시 스와핑하기 전에 어떤 페이지가 사용될지 추측.
- 대신 호출기는 해당 페이지만 메모리로 가져온다.
- 해당 페이지 세트를 결정하는 방법은 무엇일까?
  - 요구 페이징을 구현하려면 새로운 MMU 기능이 필요.
- 필요한 페이지가 이미 메모리 상주인 경우
  - 비수요 페이징demand-paging과 차이 없음
- 페이지가 필요하고 메모리에 상주하지 않는 경우
  - 저장소에서 메모리로 페이지를 감지하고 로드해야 함
  - 프로그램 동작을 변경하지 않고
  - 프로그래머가 코드를 변경할 필요 없이

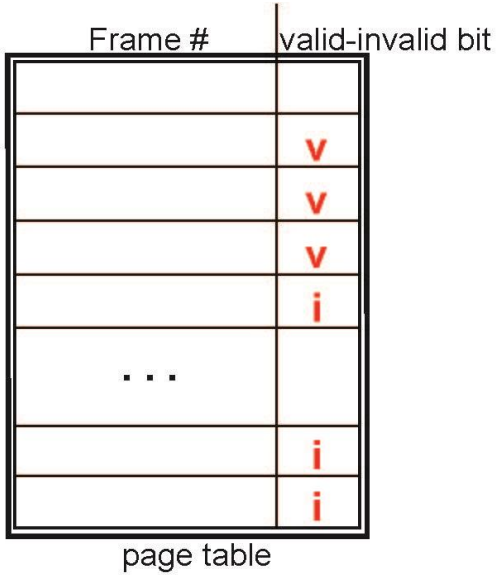
2. Demand Paging

유효-무효 비트 Valid-Invalid Bit

- 각 페이지 테이블 항목에는 유효-무효 비트가 연결

v ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory

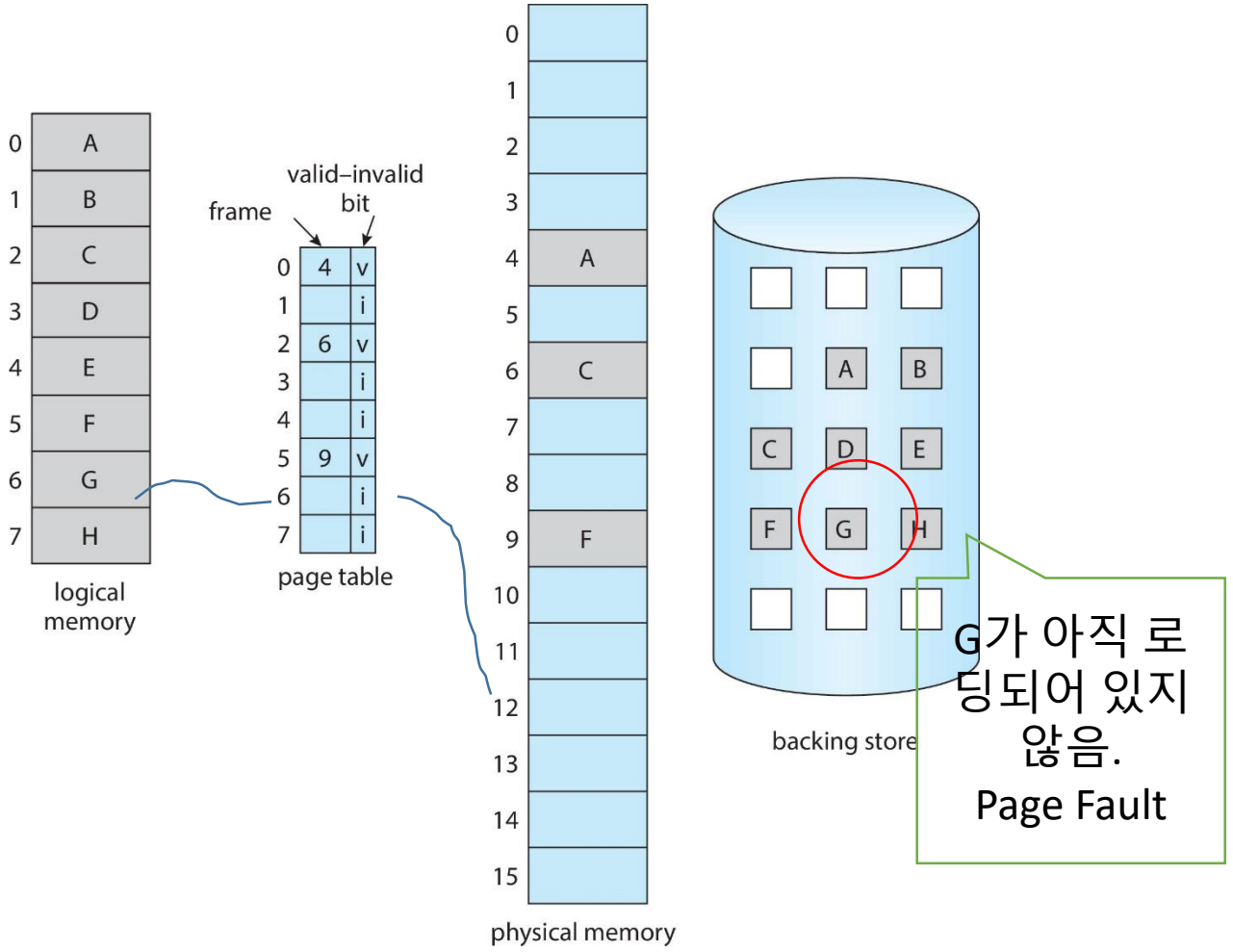
- 처음에 유효-무효 비트는 모든 항목에서 i로 설정.
- 페이지 테이블 스냅샷의 예:



- MMU 주소 변환 중 페이지 테이블 항목의 유효-무효 비트가 i ⇒ page 폴트인 경우

2. Demand Paging

일부 페이지가 메인 메모리에 없을 때의 페이지 테이블



2. Demand Paging

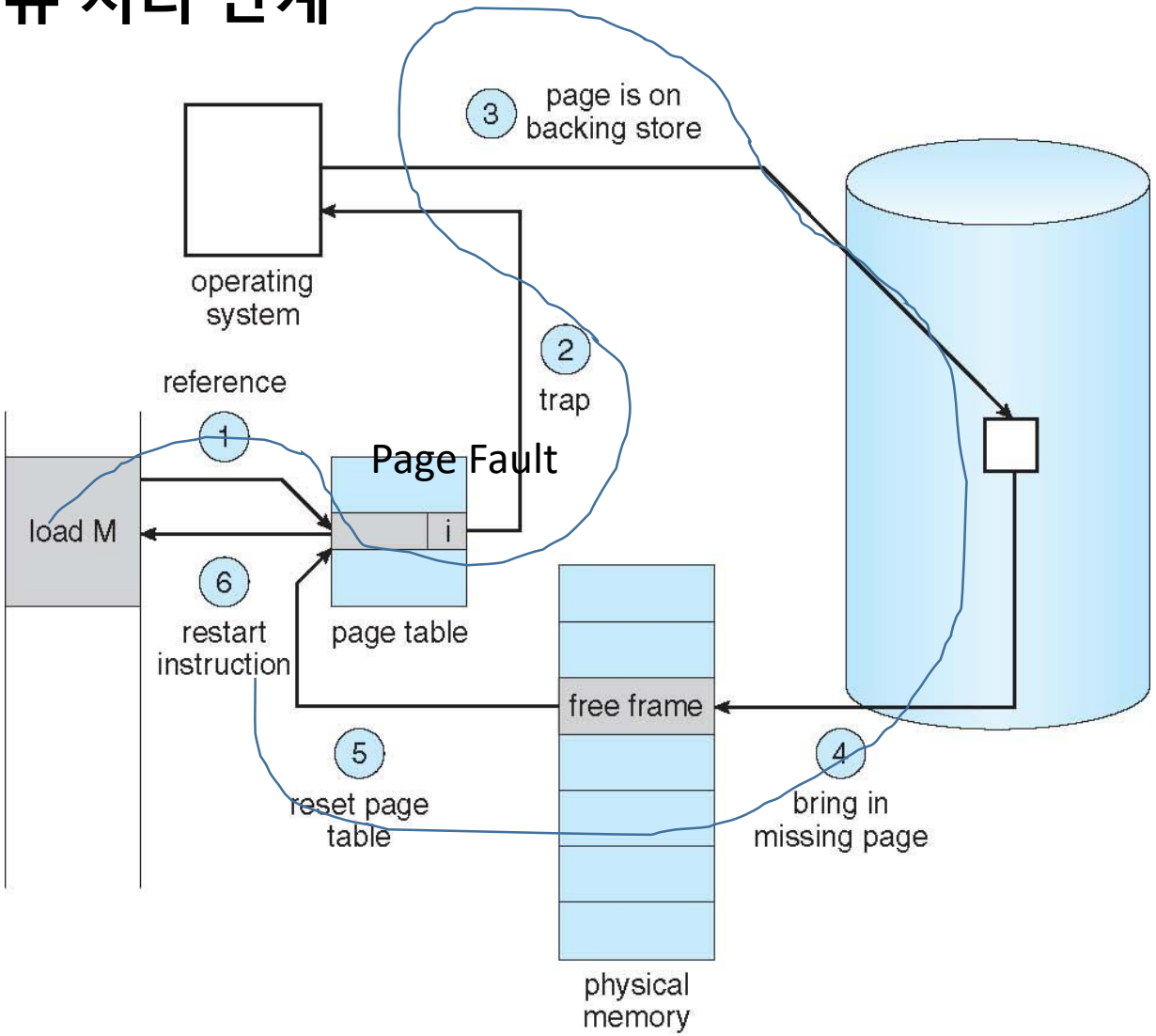
페이지 오류 처리 단계

page fault: 프로세스가 요청한 페이지가 물리적 메모리에 없는 경우 발생

- 페이지에 대한 참조가 있는 경우 해당 페이지에 대한 첫 번째 참조는 운영 체제에 트랩된다.
  Page fault
- 운영 체제는 다음을 결정하기 위해 다른 테이블을 확인.
  Invalid reference ⇒ abort
  Just not in memory
- 자유 프레임 찾기
- 예약된 디스크 작업을 통해 페이지를 프레임으로 교체
- 이제 메모리에 있는 페이지를 나타내도록 테이블을 재설정. 유효성 검사 비트 = v 설정
- 페이지 폴트를 일으킨 명령을 다시 시작한다

2. Demand Paging

페이지 오류 처리 단계



2. Demand Paging

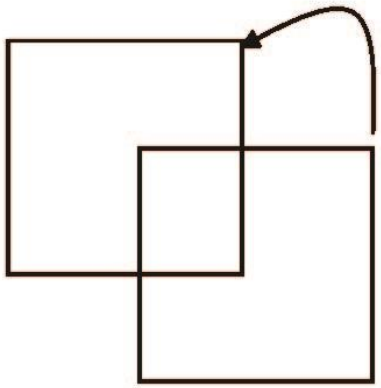
요구 페이징의 측면

- 극단적인 경우 – 메모리에 페이지가 없는 상태에서 프로세스 시
  - OS는 메모리에 상주하지 않는 프로세스의 첫 번째 명령에 대한 명령 포인터를 설정합니다. -> page fault
  - 그리고 처음 액세스할 때 다른 모든 프로세스 페이지에 대해
  - Pure demand paging
- 실제로 주어진 명령은 여러 페이지에 액세스할 수 있다 -> 여러 페이지 오류
  - 메모리에서 2개의 숫자를 추가하고 결과를 다시 메모리에 저장하는 명령의 가져오기 및 디코딩을 고려.
  - 참조 국부성(지역성) locality of reference로 인해 통증 감소
- 요구 페이징에 필요한 하드웨어 지원
  - 유효/무효 비트valid / invalid bit가 있는 페이지 테이블
  - 보조 메모리(스왑 공간 swap space 이 있는 스왑 장치)
  - 명령어 재시작

2. Demand Paging

명령어 재시작

- 여러 위치에 액세스할 수 있는 명령을 고려하십시오.
  - 블록이동



- 자동 증가/감소 위치
- 전체 작업을 다시 시작하시겠습니까?
- 소스와 대상이 겹치면 어떻게 되나요?

2. Demand Paging

Free-Frame List

- 페이지 오류가 발생하면 운영 체제는 원하는 페이지를 보조 저장소에서 주 메모리로 가져와야 합니다.
- 대부분의 운영 체제는 그러한 요청을 만족시키기 위한 자유 프레임 풀인 자유 프레임 목록 free-frame list 을 유지.

- 운영 체제는 일반적으로 주문형 제로 채우기(zero-fill-on-demand)라는 기술을 사용하여 무료 프레임을 할당. 즉, 할당되기 전에 프레임 내용이 0이 됩니다.
- 시스템이 시작되면 사용 가능한 모든 메모리가 자유 프레임 목록에 배치.

2. Demand Paging

수요 페이징의 단계 – 최악의 경우

1. 운영 체제에 트랩
2. 사용자 레지스터 및 프로세스 상태 저장
3. 인터럽트가 페이지 폴트인지 확인
4. 페이지 참조가 합법적인지 확인하고 디스크에서 페이지의 위치를 결정합니다.
5. 디스크에서 사용 가능한 프레임으로 읽기를 실행.

- 읽기 요청이 처리될 때까지 이 장치의 대기열에서 기다린다.
- 장치 검색 및/또는 대기 시간을 기다립니다.
- 자유 프레임으로 페이지 전송 시작

2. Demand Paging

수요 페이징의 단계 – 최악의 경우

6. 기다리는 동안 CPU를 다른 사용자에게 할당
7. 디스크 I/O 하위 시스템에서 인터럽트 수신(I/O 완료)
8. 다른 사용자를 위해 레지스터 및 프로세스 상태 저장
9. 인터럽트가 디스크에서 발생한 것인지 확인
10. 페이지가 현재 메모리에 있음을 표시하도록 페이지 테이블 및 기타 테이블 수정
11. CPU가 이 프로세스에 다시 할당될 때까지 기다립니다.
12. 사용자 레지스터, 프로세스 상태 및 새 페이지 테이블을 복원한 다음 중단 된 명령 재개

2. Demand Paging

수요 페이징의 단계 – 최악의 경우

- 세 가지 주요 활동
  - 인터럽트 서비스 – 신중한 코딩은 수백 개의 명령만 필요함을 의미.
  - 페이지 읽기 – 많은 시간
  - 프로세스를 다시 시작 – 다시 약간의 시간

Page Fault Rate $0 \le p \le 1$

- **p = 0이면 페이지 폴트 없음**
- **p = 1이면 모든 참조가 오류.**

- 유효 액세스 시간(EAT)

EAT = (1 – p) x memory access
+ p (page fault overhead
+ swap page out
+ swap page in)

$$EAT = (1 - p) \times ma + p * (page\ fault\ time)$$

2. Demand Paging

요구 페이징 예

- 메모리 액세스 시간 = 200나노초
- 평균 페이지 오류 서비스 시간 = 8밀리초

EAT = (1 – p) x 200 + p (8 milliseconds)
= (1 – p x 200 + p x 8,000,000
= 200 + p x 7,999,800

- 1,000개의 액세스 중 하나가 페이지 폴트를 일으키는 경우
- EAT = 8.2마이크로초.
- 이것은 40배나 느려진 것
- 성능 저하를 원하는 경우 < 10%

220 > 200 + 7,999,800 x p
20 > 7,999,800 x p
p < .0000025
< one page fault in every 400,000 memory accesses

2. Demand Paging

요구 페이징 최적화

- 동일한 장치에 있더라도 파일 시스템 I/O보다 빠른 공간 I/O 스왑
  - 더 큰 청크에 할당된 스왑, 파일 시스템보다 적은 관리 필요
- 프로세스 로드 시간에 전체 프로세스 이미지를 스왑 공간에 복사
  - 그런 다음 스왑 공간의 페이지 인 및 아웃
  - 이전 BSD Unix에서 사용됨
- 디스크의 프로그램 바이너리에서 페이지 인을 요구하지만 프레임을 해제할 때 페이지 아웃하지 않고 버립니다.
  - Solaris 및 현재 BSD에서 사용됨
  - 여전히 스왑 공간에 써야 합니다.
  - 파일과 연결되지 않은 페이지(예: 스택 및 힙) - 익명 메모리
  - 메모리에서 수정되었지만 아직 파일 시스템에 다시 기록되지 않은 페이지
- 모바일 시스템
  - 일반적으로 스와핑을 지원하지 않음
  - 대신 파일 시스템에서 페이지를 요구하고 읽기 전용 페이지(예: 코드)를 회수.

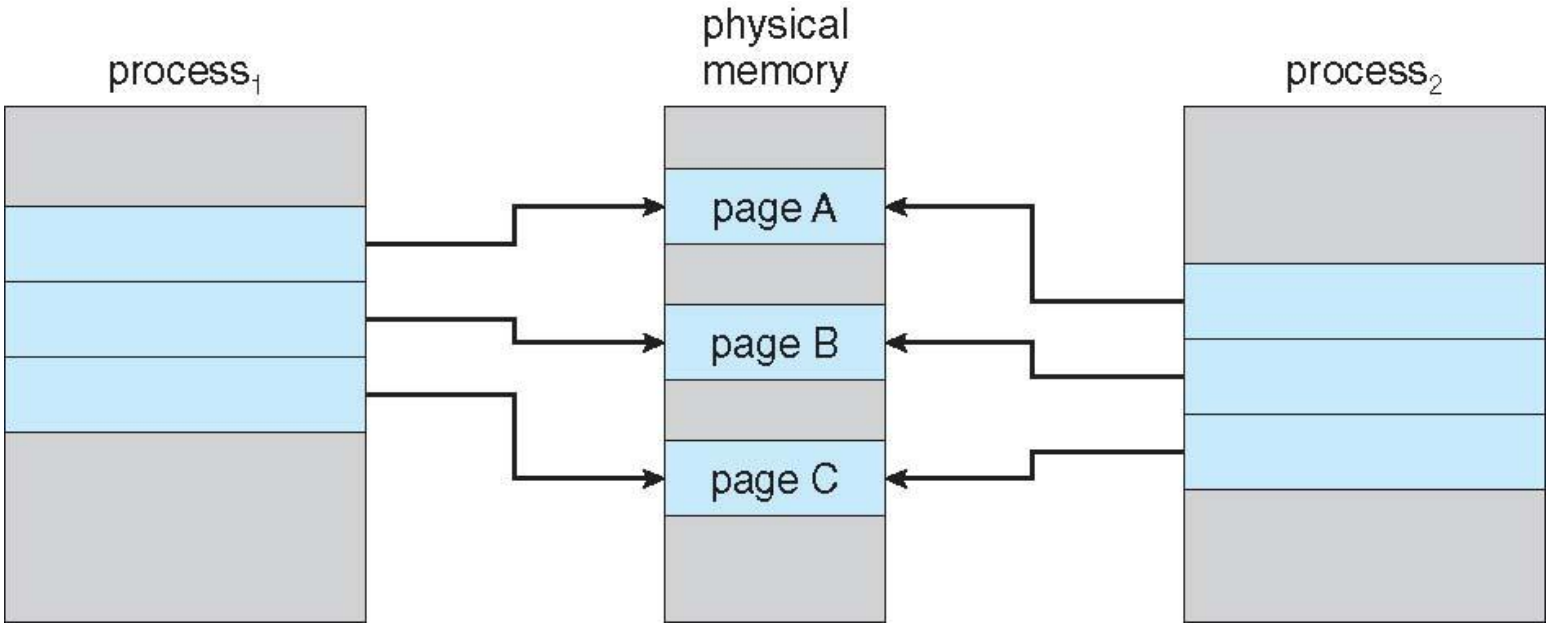
3. Copy-on-Write

기본 개념

- COW(Copy-on-Write)는 상위 및 하위 프로세스가 처음에 메모리에서 동일한 페이지를 공유할 수 있도록 한다.
  - 두 프로세스 중 하나가 공유 페이지를 수정하는 경우에만 페이지가 복사.
- COW는 수정된 페이지만 복사되므로 보다 효율적인 프로세스 생성이 가능.
- 일반적으로 사용 가능한 페이지는 제로 필 온디맨드 페이지 풀에서 할당.
  - 빠른 수요 페이지 실행을 위해 풀에는 항상 사용 가능한 프레임이 있어야 한다.
  - 페이지 오류에 대한 다른 처리뿐만 아니라 프레임을 해제하고 싶지 않다.
  - 페이지를 할당하기 전에 페이지를 0으로 만드는 이유는 무엇일까?
- fork() 시스템 호출의 vfork() 변형에는 부모 일시 중단 및 부모의 copy-on-write 주소 공간을 사용하는 자식이 있다.
  - 하위 호출 exec()를 갖도록 설계됨
  - 매우 효율적

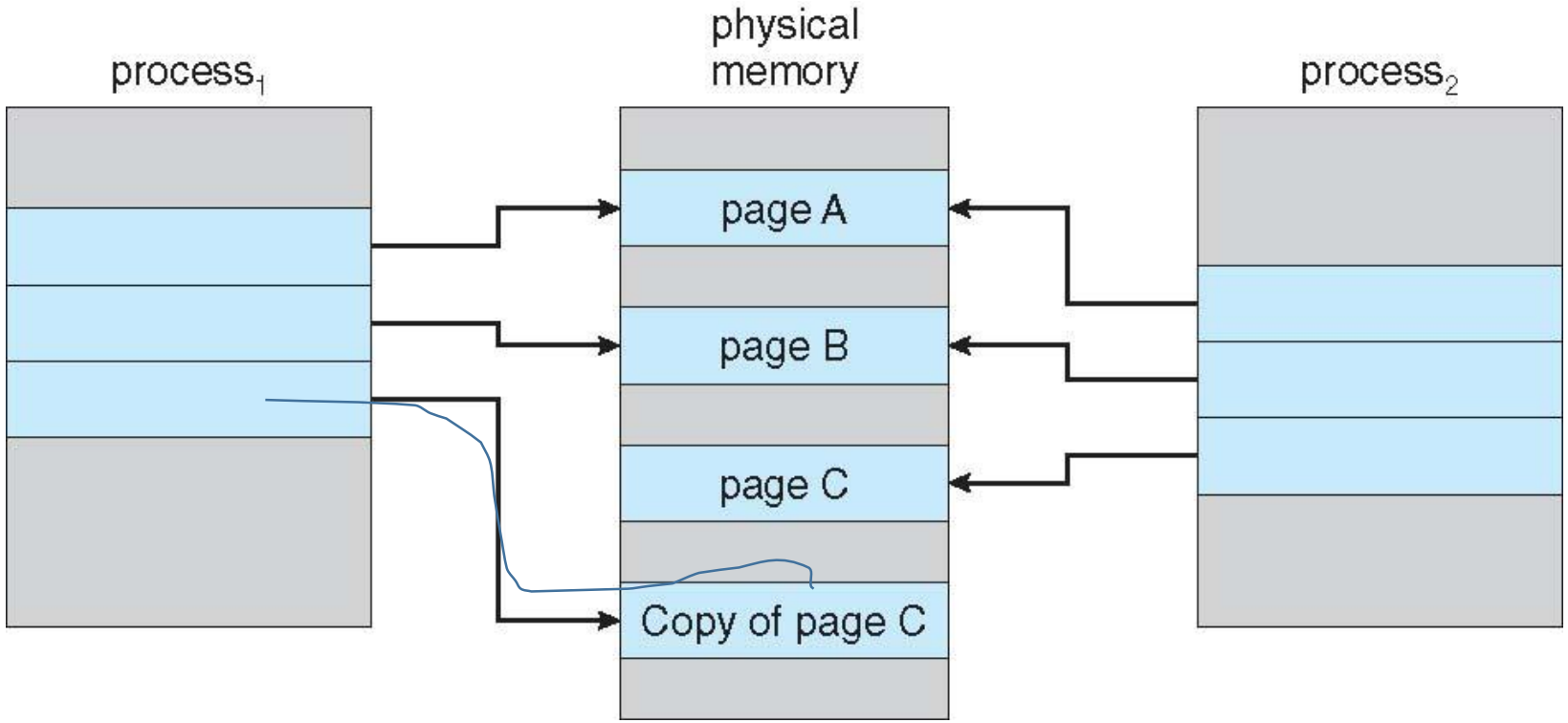
3. Copy-on-Write

프로세스 1이 페이지 C를 수정하기 전



3. Copy-on-Write

프로세스 1이 페이지 C를 수정한 후



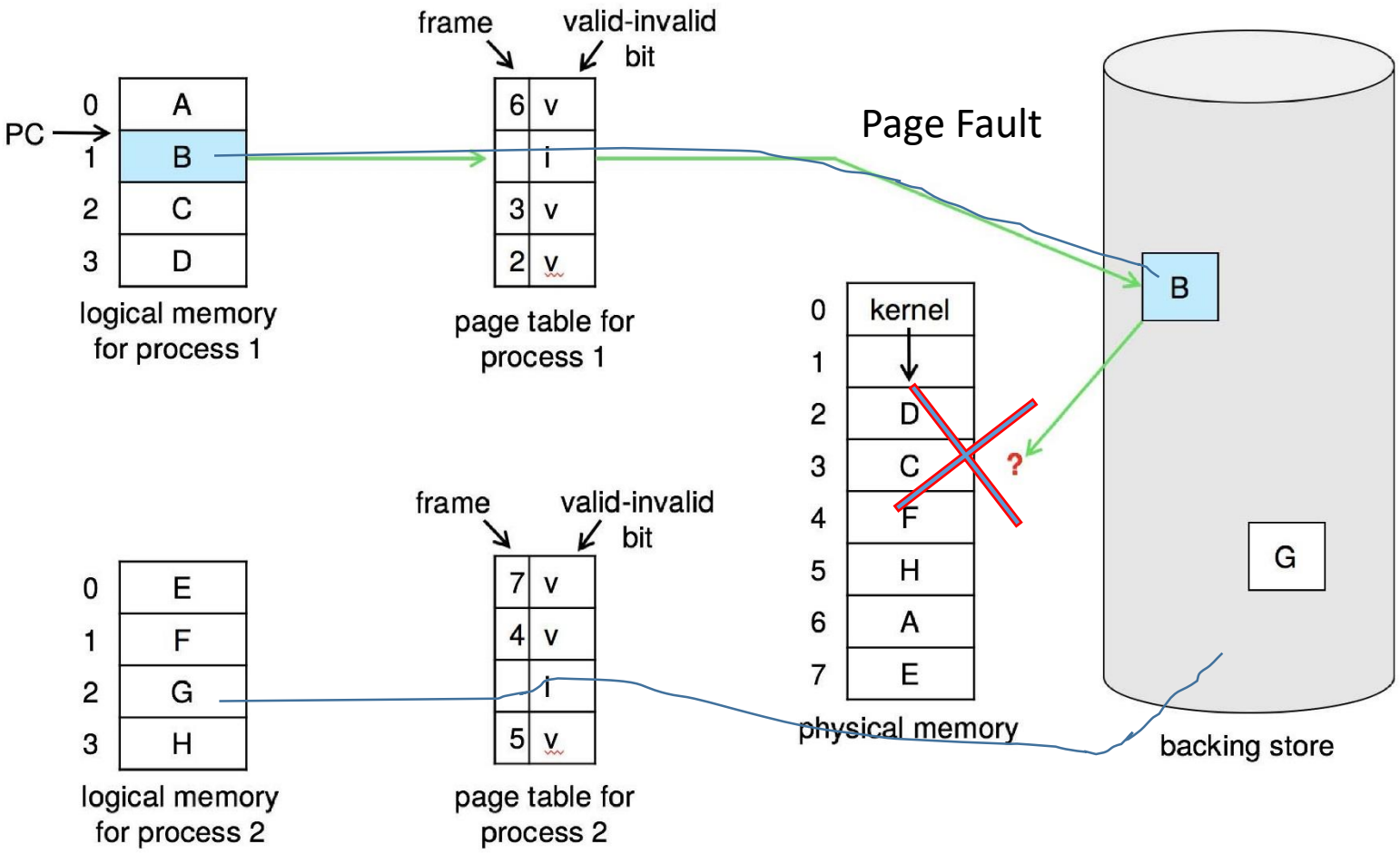
3. Copy-on-Write

여유 프레임이 없으면 어떻게 됩니까?

- 프로세스 페이지에서 사용됨
- 또한 커널, I/O 버퍼 등의 요구가 있다.
- 각각에 얼마를 할당할 것인가?
- 페이지 교체 – 메모리에서 일부 페이지를 찾지만 실제로는 사용하지 않고 페이지 아웃
  - 알고리즘 - 종료? 교환? 페이지를 교체?
  - 성능 – 페이지 폴트를 최소화하는 알고리즘이 필요.
- 동일한 페이지를 메모리에 여러 번 가져올 수 있다.

4. Page Replacement

페이지 교체 필요



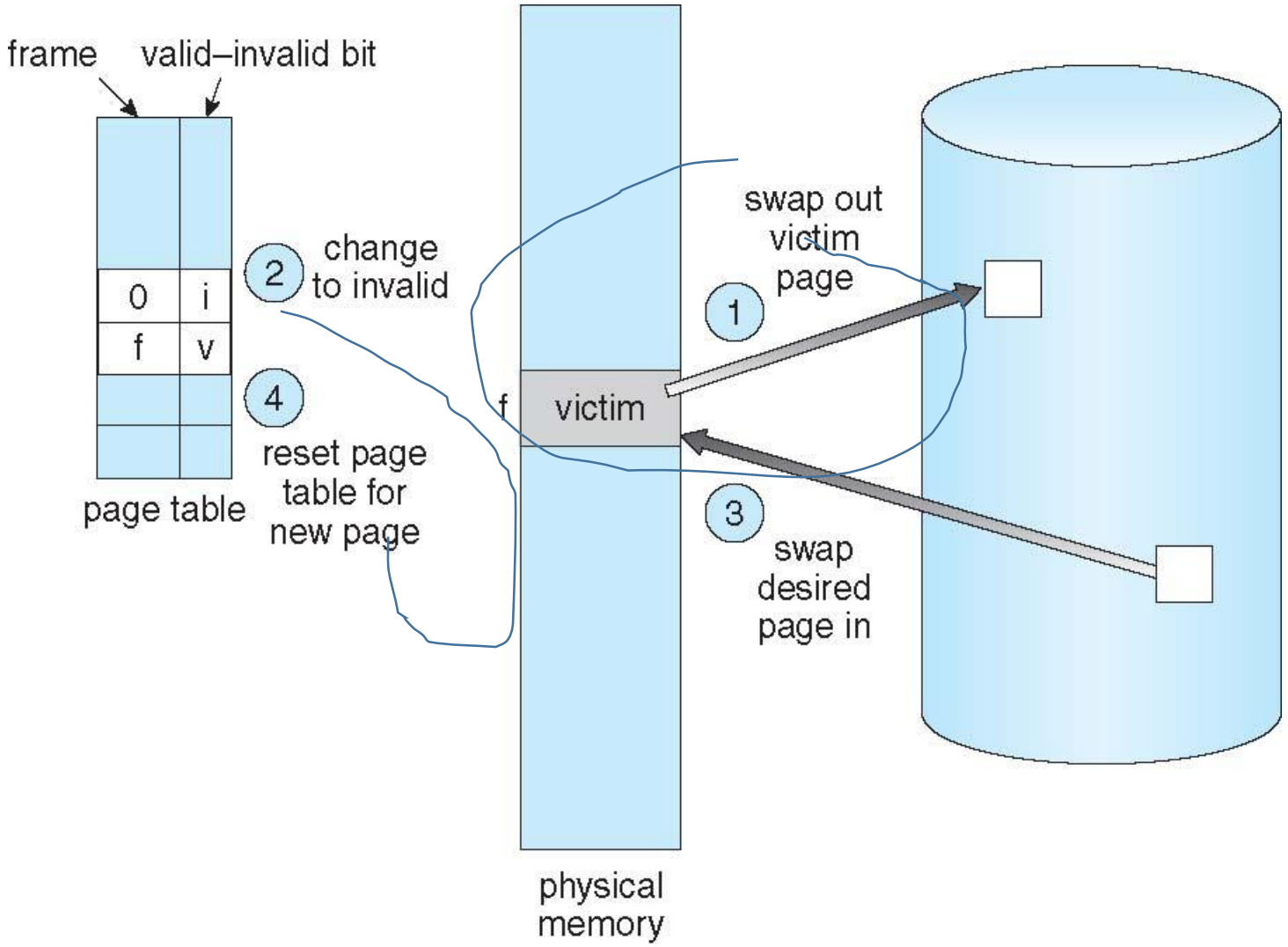
4. Page Replacement

기본 페이지 교체

1. 디스크에서 원하는 페이지의 위치 찾기
2. 빈 프레임 찾기:
   - 빈 프레임이 있으면 사용
   - 빈 프레임이 없으면 페이지 교체 알고리즘을 사용하여 희생 프레임victim frame 선택
   - 더티한 경우 희생 프레임을 디스크에 기록
3. 원하는 페이지를 (새로) 자유 프레임으로 가져옵니다. 페이지 및 프레임 테이블 업데이트
4. 트랩을 유발한 명령을 다시 시작하여 프로세스를 계속.

4. Page Replacement

기본 페이지 교체



4. Page Replacement

페이지 및 프레임 교체 알고리즘

- 프레임 할당 알고리즘 Frame-allocation algorithm 이 결정
  - 각 프로세스에 제공할 프레임 수
  - 교체할 프레임
- 페이지 교체 알고리즘Page-replacement algorithm
  - 최초 액세스와 재접속 모두에서 페이지 폴트율이 가장 낮기를 원함
- 특정 메모리 참조 문자열(참조 문자열)에서 실행하고 해당 문자열에서 페이지 폴트 수를 계산하여 알고리즘을 평가.
  - 문자열은 전체 주소가 아니라 페이지 번호일 뿐.
  - 동일한 페이지에 반복적으로 액세스해도 페이지 폴트가 발생하지 않다.
  - 결과는 사용 가능한 프레임 수에 따라 달라진다.

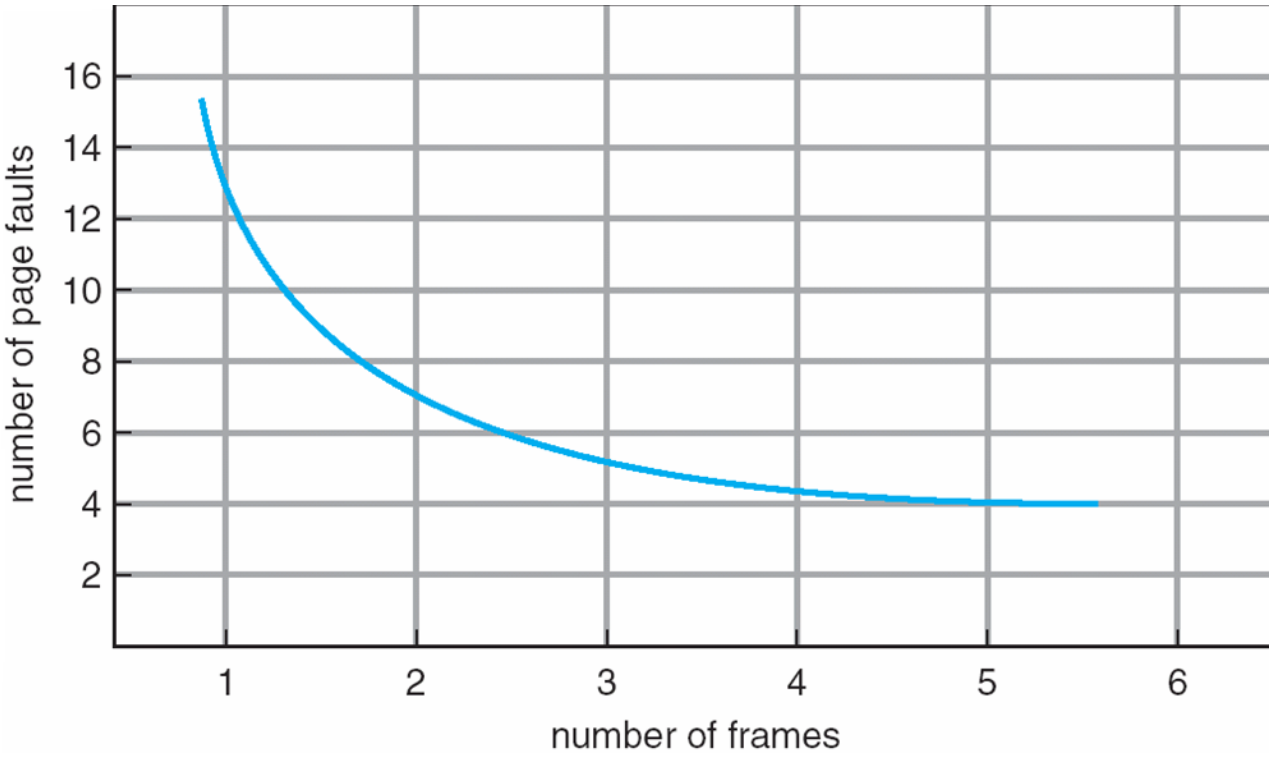
- 모든 예에서 참조된 페이지 번호의 참조 문자열은 다음과 같다.

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

4. Page Replacement

페이지 폴트 대 프레임 수 그래프

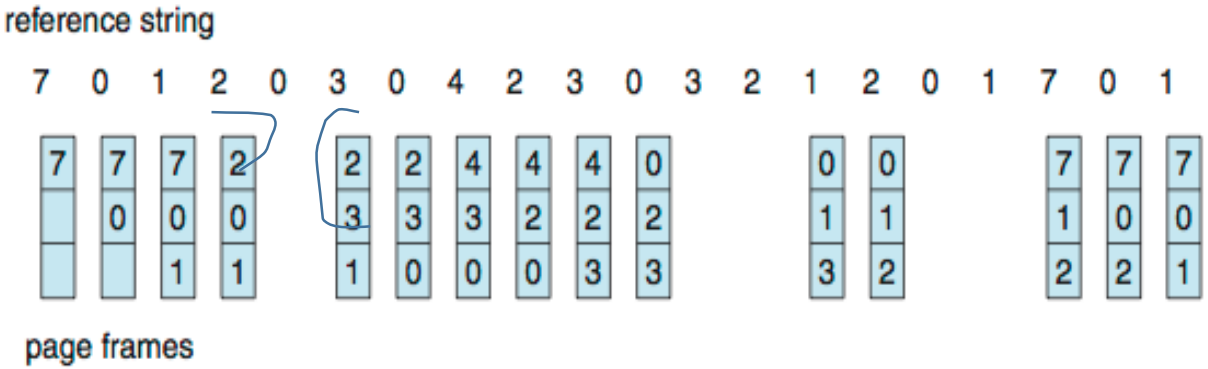
메모리를 너무 적게 분할하면(프레임수가 적으면) 올릴데가 없음



4. Page Replacement

First-In-First-Out (FIFO) Algorithm

- 참조 문자열: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 프레임(프로세스당 한 번에 3페이지가 메모리에 있을 수 있음)



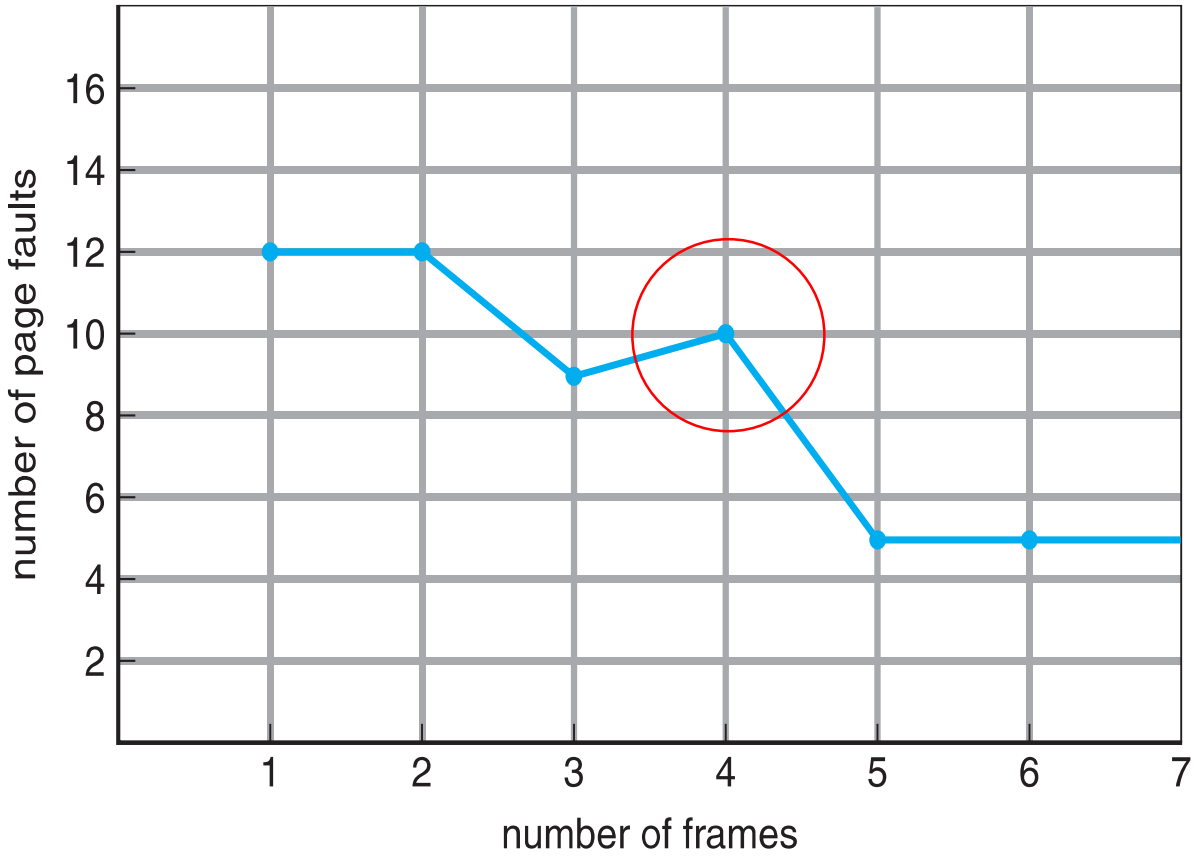
15 page faults

- 참조 문자열에 따라 다를 수 있음: 1,2,3,4,1,2,5,1,2,3,4,5 고려
  - 더 많은 프레임을 추가하면 더 많은 페이지 오류가 발생할 수 있다!
  - Belady의 변칙
- 페이지의 연령을 추적하는 방법은 무엇일까?
  - FIFO 대기열을 사용.

4. Page Replacement

Belady의 이상 현상을 보여주는 FIFO

- 페이지 프레임이 늘어났지만, Page-fault가 늘어나는 이상한 현상
- 123412512345

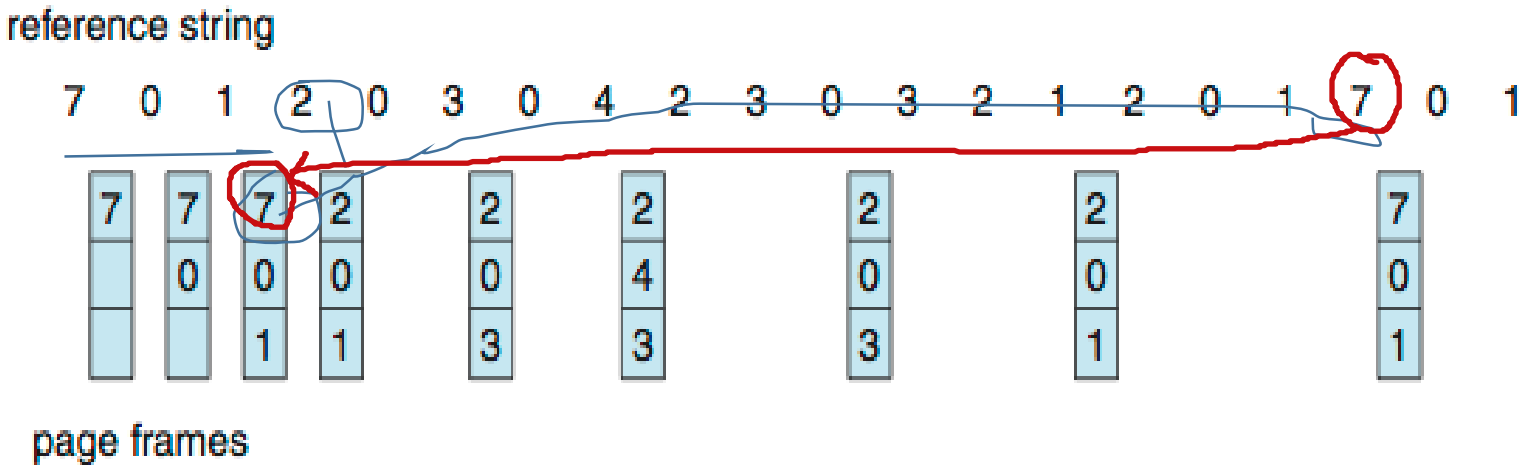


4. Page Replacement

최적 알고리즘 OPT

이거이거 백준 콘센트 알고리즘임 ㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋ

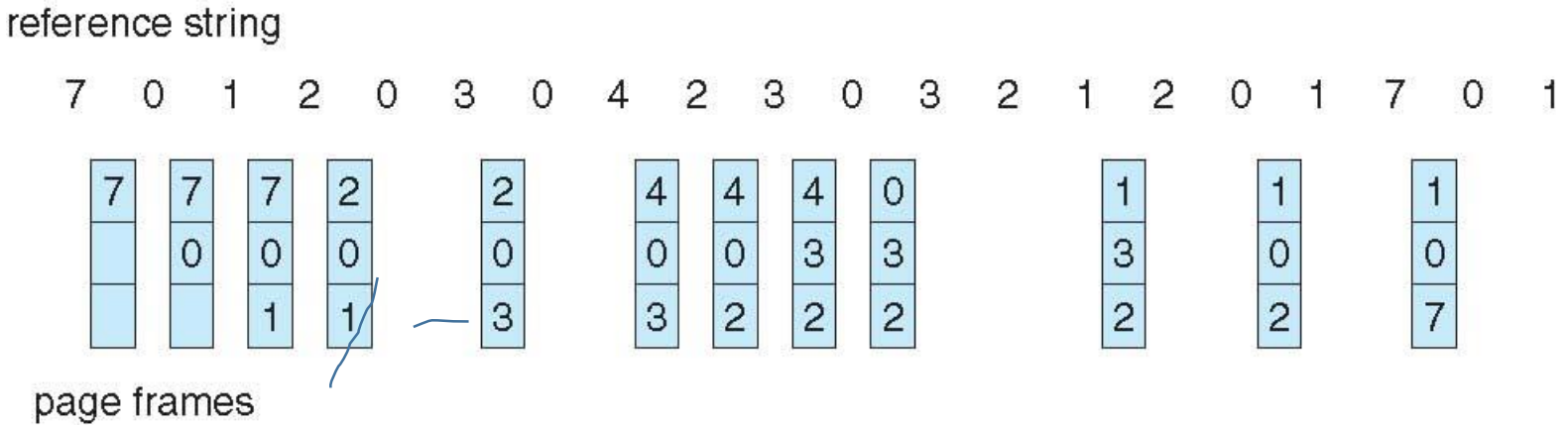
- 가장 오랫동안 사용하지 않을(사용안 될 것 같은) 페이지 교체
  - 예를 들어 9가 최적.
- 당신은 이것을 어떻게 알까?
  - 미래를 읽을 수 없다
- 알고리즘이 얼마나 잘 수행되는지 측정하는 데 사용.



4. Page Replacement

Least Recently Used (LRU) Algorithm

- 가장 최근에 사용됨 > 마지막 사용 시간을 각 페이지와 연결
- 미래보다는 과거의 지식을 사용하라
- 가장 오랫동안 사용하지 않은 페이지 교체



- 12개 오류 – FIFO보다 낫지만 OPT보다 나쁨
- 일반적으로 좋은 알고리즘이며 자주 사용됨

4. Page Replacement

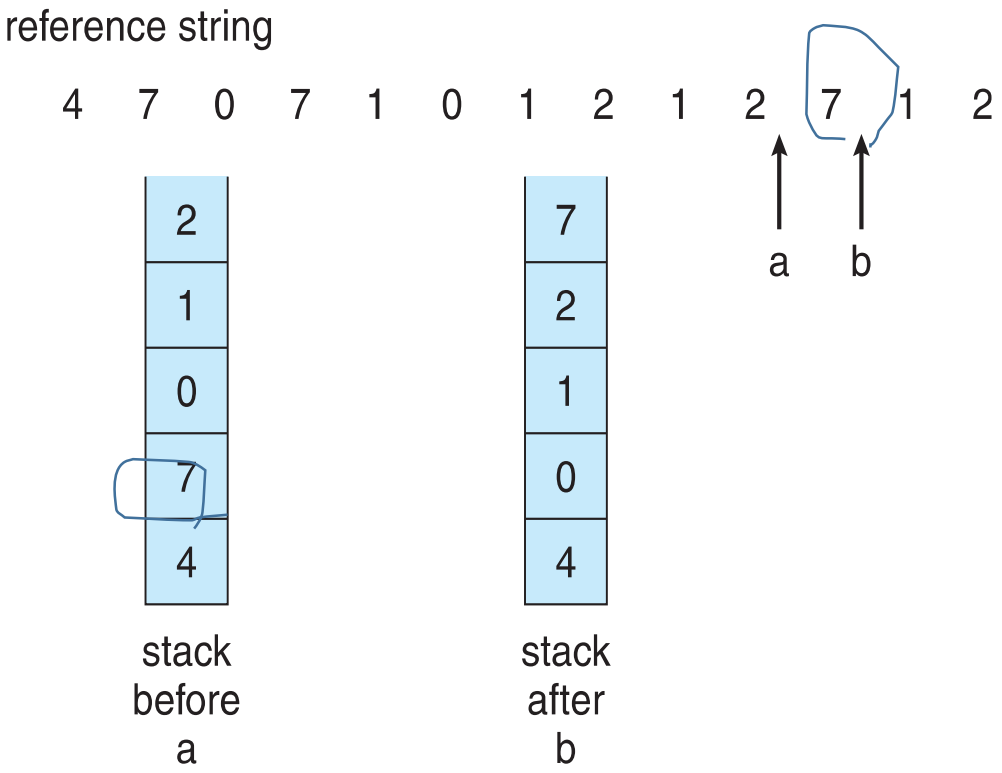
Least Recently Used (LRU) Algorithm

- **카운터 구현**
  - 모든 페이지 항목에는 카운터가 있다. 이 항목을 통해 페이지를 참조할 때마다 시계를 카운터에 복사.
  - 페이지를 변경해야 할 때 카운터를 보고 가장 작은 값을 찾는다.
  - 필요한 테이블을 통한 검색
- **스택 Stack 구현**
  - 이중 링크 형식으로 페이지 번호 스택 유지
  - **Page referenced**
    - ✓ 맨 위로 이동
    - ✓ 변경하려면 6개의 포인터가 필요.
  - 그러나 각 업데이트는 더 비싸다.
  - 대체 검색 없음

4. Page Replacement

Least Recently Used (LRU) Algorithm

- LRU와 OPT는 Belady's Anomaly가 없는 스택 알고리즘의 경우
- 스택을 사용하여 가장 최근 페이지 참조 기록



4. Page Replacement

LRU 근사 알고리즘 LRU Approximation Algorithms

- LRU는 특수 하드웨어가 필요하지만 여전히 느림
- Reference bit
  - 각 페이지는 초기에 = 0 비트를 연관.
  - 페이지를 참조할 때 비트가 1로 설정됨
  - 아무거나 참조 비트 = 0으로 교체(있는 경우)
  - 순서를 모르지만

4. Page Replacement

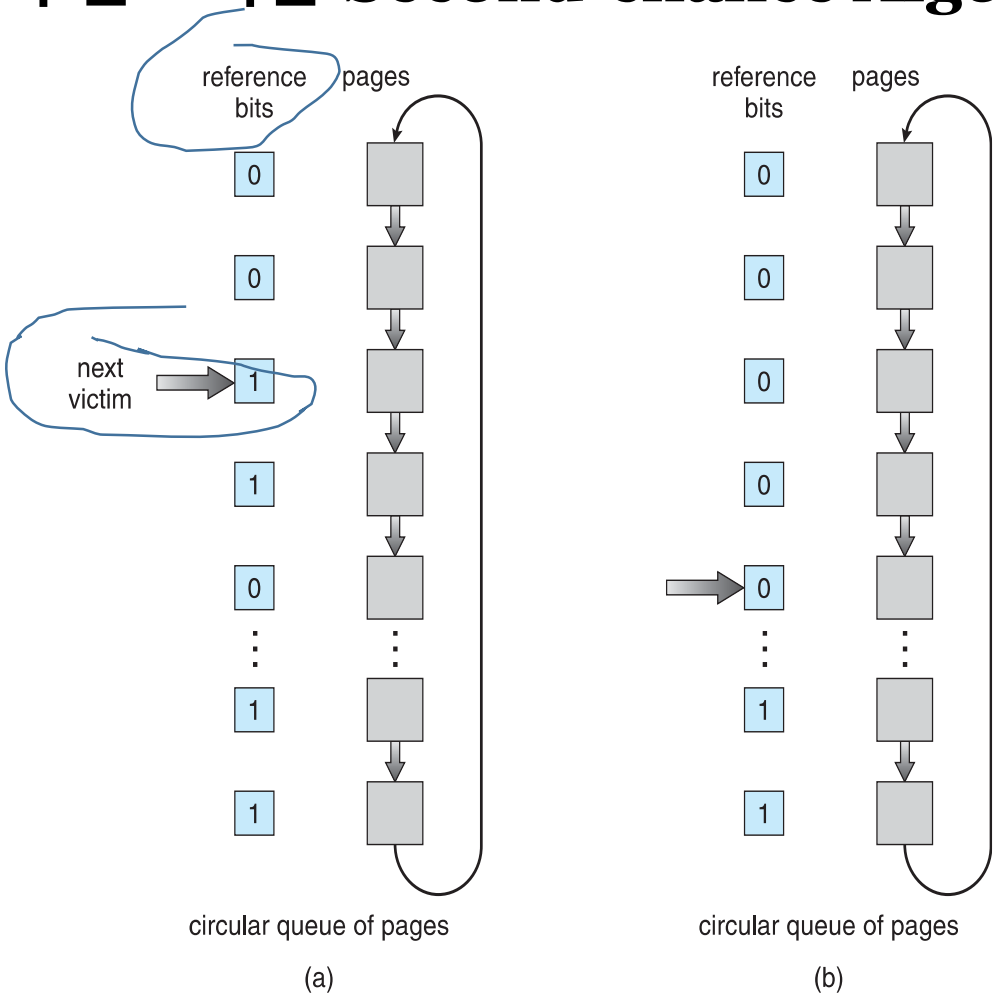
LRU 근사 알고리즘 LRU Approximation Algorithms

- 두 번째 기회 알고리즘
- 일반적으로 FIFO 및 하드웨어 제공 참조 비트
- **Clock replacement**
- 교체할 페이지가 있는 경우

Reference bit = 0 -> replace it
reference bit = 1 then:
set reference bit 0, leave page in memory
replace next page, subject to same rules

4. Page Replacement

두 번째 기회 알고리즘 Second-chance Algorithm



4. Page Replacement

향상된 두 번째 기회 알고리즘

- 참조 비트와 수정 비트(사용 가능한 경우)를 함께 사용하여 알고리즘을 개선.
- 순서쌍 취하기(참조, 수정):
  - (0, 0) 최근 사용되지 않음 수정되지 않음 - 교체할 최적의 페이지
  - (0, 1) 최근에 사용되지 않았지만 수정됨 – 좋지 않음, 교체 전에 작성해야 함
  - (1, 0) 최근에 사용되었지만 깨끗함 - 아마 곧 다시 사용될 것임
  - (1, 1) 최근에 사용 및 수정됨 – 아마도 곧 다시 사용될 것이며 교체하기 전에 작성해야 함
- 페이지 교체가 필요한 경우 클록 체계를 사용하지만 4개의 클래스를 사용하여 비어 있지 않은 가장 낮은 클래스의 페이지를 교체.
  - 순환 대기열을 여러 번 검색해야 할 수 있음

4. Page Replacement

Counting Algorithms

- 각 페이지에 대한 참조 횟수 카운터를 유지.
- 흔하지 않아
- **Lease Frequently Used (LFU) Algorithm :**
- 카운트가 가장 작은 페이지를 교체.
- **Most Frequently Used (MFU) Algorithm :**
- 카운트가 가장 적은 페이지가 아마도 방금 가져와 아직 사용되지 않았다는 주장에 근거.

4. Page Replacement

Page-Buffering Algorithms

- 항상 무료 프레임 풀 유지
  - 그런 다음 필요할 때 프레임을 사용할 수 있지만 오류 시간에는 찾을 수 없다.
  - 자유 프레임으로 페이지를 읽고 제거할 희생자를 선택하고 자유 풀에 추가
  - 편리할 때 피해자 퇴거victim
- 가능하면 수정된 페이지 목록을 유지하십시오.
  - 그렇지 않으면 유휴 저장소를 백업할 때 거기에 페이지를 쓰고 non-dirty로 설정.
- 가능하면 무료 프레임 내용을 그대로 유지하고 내용물을 기록.
  - 재사용하기 전에 다시 참조하면 디스크에서 콘텐츠를 다시 로드할 필요가 없다.
  - 일반적으로 잘못된 피해자 프레임을 선택한 경우 페널티를 줄이는 데 유용.

4. Page Replacement

응용 프로그램 및 페이지 교체

- 이러한 모든 알고리즘에는 향후 페이지 액세스에 대해 추측하는 OS가 있다.
- 일부 응용 프로그램은 더 나은 지식을 가지고 있다. 즉, 데이터베이스
- 메모리 집약적인 애플리케이션은 이중 버퍼링을 유발할 수 있다.
  - OS는 페이지 사본을 I/O 버퍼로 메모리에 보관.
  - 응용 프로그램은 자체 작업을 위해 페이지를 메모리에 유지.
- 운영 체제는 응용 프로그램을 방해하지 않고 디스크에 직접 액세스할 수 있다.
  - Raw disk mode
- 버퍼링, 잠금 등을 우회.

5. Allocation of Frames

기본 개념

- 각 프로세스에는 최소한 minimum 의 프레임이 필요다.
- 예: IBM 370 – SS MOVE 명령을 처리하기 위한 6페이지:
  - 명령은 6바이트이며 2페이지에 걸쳐 있을 수 있다.
  - 처리할 페이지 2개
  - 처리할 2페이지
- 최대 Maximum 는 시스템의 총 프레임.
- 두 가지 주요 할당 체계
  - 고정 할당fixed allocation
  - 우선순위 할당priority allocation
- 다양한 변형

5. Allocation of Frames

고정 할당

- 균등 할당 Equal allocation – 예를 들어 100개의 프레임(OS에 프레임 할당 후)과 5개의 프로세스가 있는 경우 각 프로세스에 20개의 프레임을 할당.
  - 일부를 무료 프레임 버퍼 풀로 유지
- 비례할당 Proportional allocation – 프로세스 규모에 따라 할당
  - 멀티프로그래밍의 정도에 따라 동적, 프로세스 크기 변경

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

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

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

5. Allocation of Frames

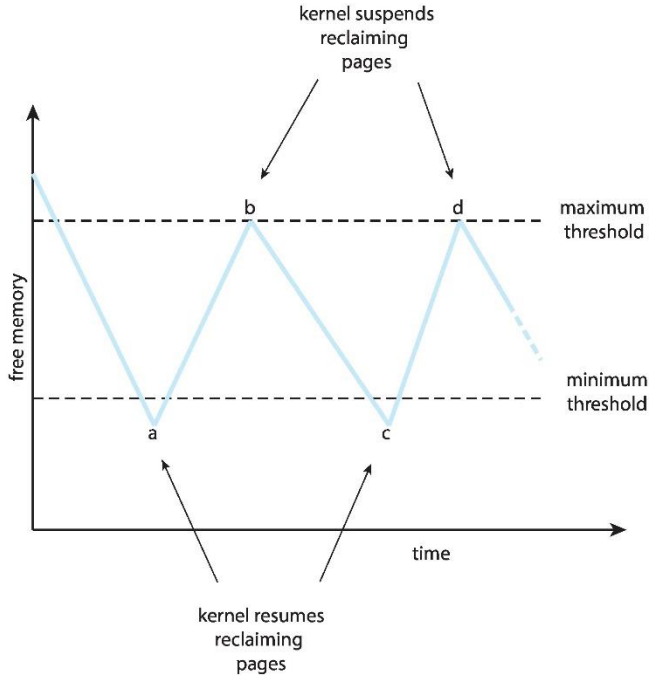
글로벌 대 로컬 할당

- 전체 교체 Global replacement – 프로세스는 모든 프레임 세트에서 교체 프레임을 선택합니다. 한 프로세스가 다른 프로세스에서 프레임을 가져올 수 있음
  - 프로세스 실행 시간은 크게 다를 수 있다.
  - 더 많은 처리량이 있으므로 더 일반적.
- 로컬 교체 Local replacement – 각 프로세스는 자신에게 할당된 프레임 집합에서만 선택.
  - 보다 일관된 프로세스별 성능
  - 하지만 사용률이 낮은 메모리

5. Allocation of Frames

페이지 회수 Reclaiming Pages

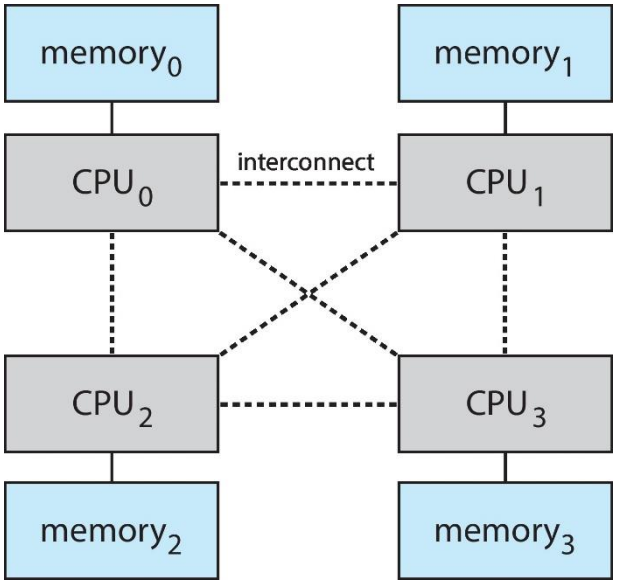
- 전역 페이지 교체 정책을 구현하기 위한 전략
- 교체할 페이지를 선택하기 전에 목록이 0으로 떨어질 때까지 기다리지 않고 자유 프레임 목록에서 모든 메모리 요청이 충족.
- 목록이 특정 임계값 아래로 떨어지면 페이지 교체가 트리거.
- 이 전략은 새로운 요청을 만족시키기에 항상 충분한 여유 메모리가 있는지 확인하려고 시도.



5. Allocation of Frames

균일하지 않은 메모리 액세스

- 지금까지 우리는 모든 메모리가 동일하게 액세스된다고 가정.
- 많은 시스템이 NUMA. 메모리 액세스 속도는 다양.
  - 시스템 버스를 통해 상호 연결된 CPU 및 메모리를 포함하는 시스템 보드를 고려.
- NUMA 다중 처리 아키텍처



5. Allocation of Frames

균일하지 않은 메모리 액세스

- 최적의 성능은 스레드가 예약된 CPU에 "가깝게" 메모리를 할당하는 것에서 비롯.
- 그리고 가능한 경우 동일한 시스템 보드에서 스레드를 예약하도록 스케줄러를 수정.
- lgroup을 생성하여 Solaris에서 해결

① CPU/메모리 지연 시간이 짧은 그룹을 추적하는 구조
② 내 일정 및 호출기를 사용.
③ 가능한 경우 프로세스의 모든 스레드를 예약하고 lgroup 내에서 해당 프로세스에 대한 모든 메모리를 할당.

6. Thrashing

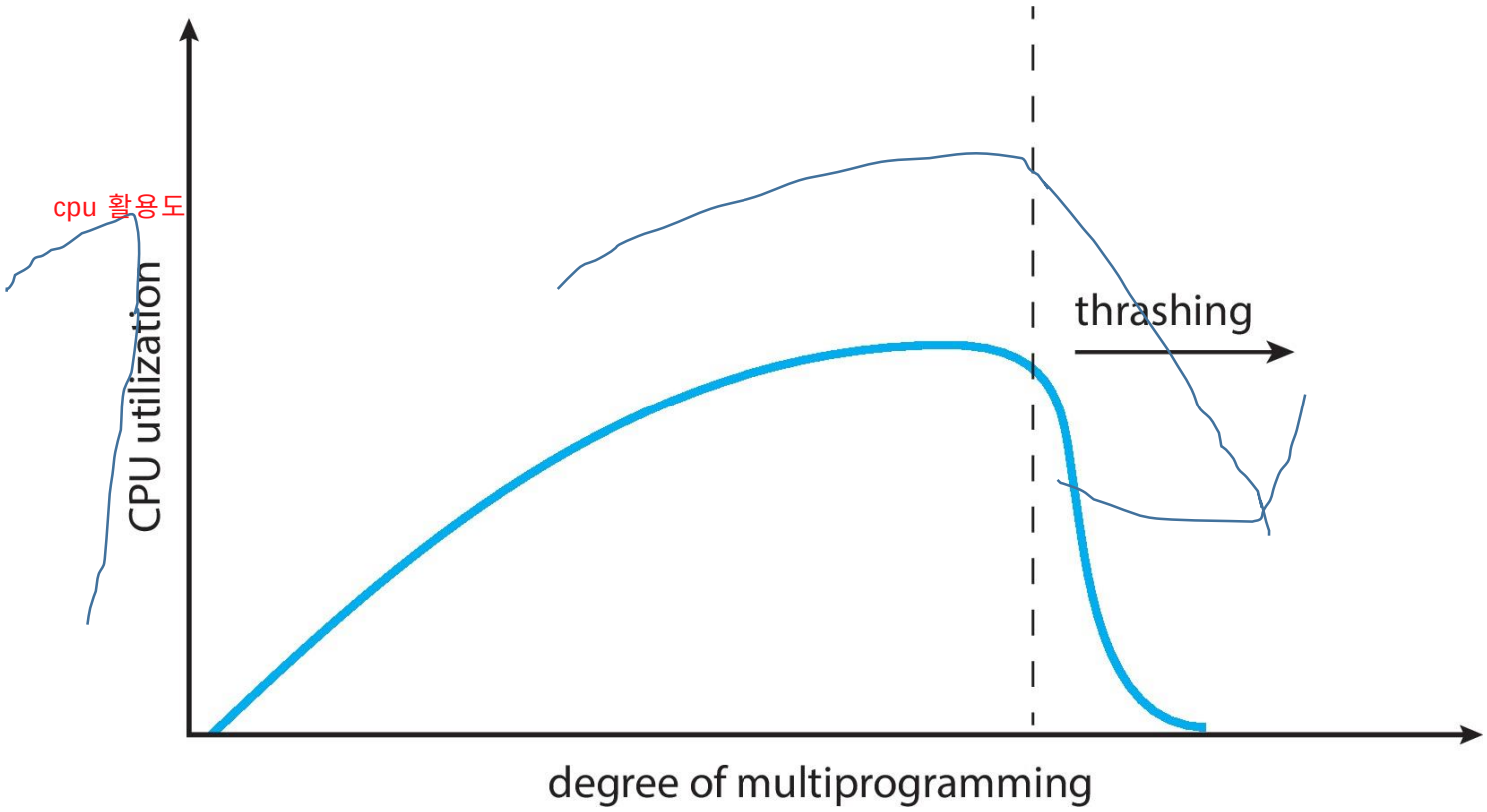
기본 개념

- 프로세스에 "충분한" 페이지가 없으면 페이지 오류율이 매우 높다.
  - 페이지를 가져오는 페이지 오류
  - 기존 프레임 교체
  - 그러나 빨리 프레임을 다시 교체.
  - 이로 인해 다음이 발생.
    - ✓ 낮은 CPU 사용률
    - ✓ 멀티프로그래밍 정도를 높여야 한다고 생각하는 운영체제
    - ✓ 시스템에 추가된 다른 프로세스

6. Thrashing

기본 개념

- **Thrashing. 프로세스가 페이지 안팎을 교환하느라 바쁘다.**



6. Thrashing

페이징 및 스레싱 요구

- 요구 페이징이 작동하는 이유는 무엇일까?

Locality model 지역성

- 프로세스가 한 지역에서 다른 지역으로 마이그레이션됨
- 지역이 겹칠 수 있음

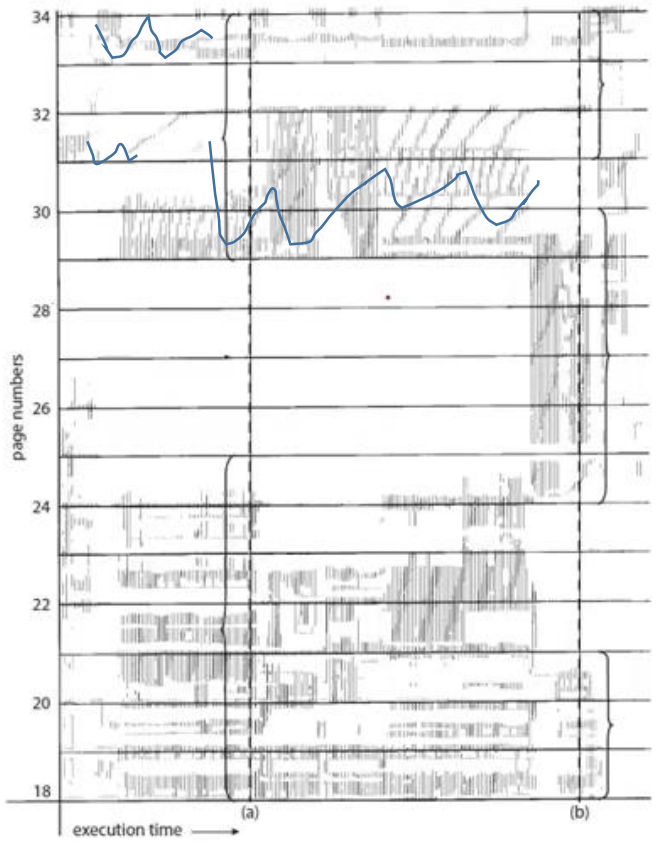
- 스래싱이 발생하는 이유는 무엇일까?

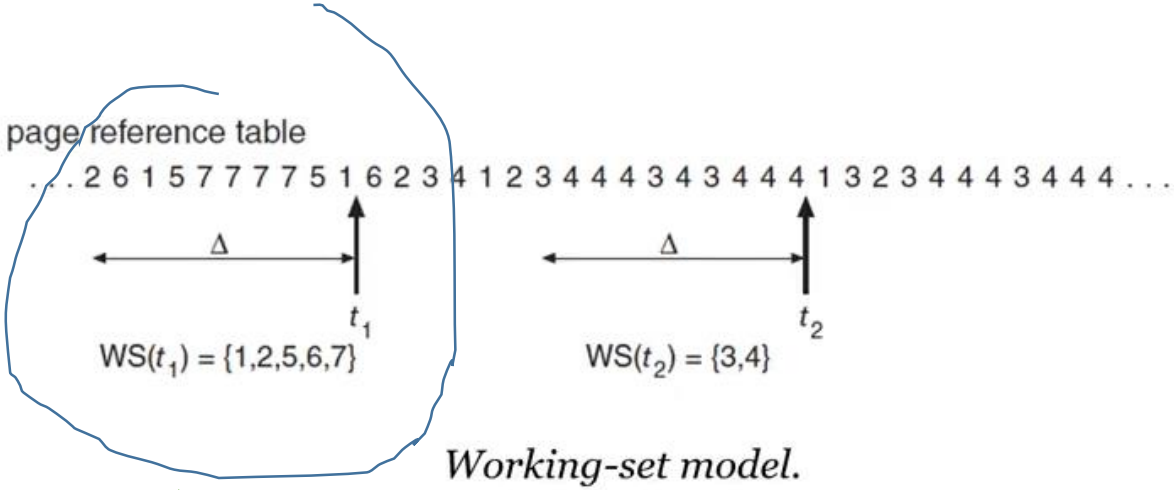
Σ size of locality > total memory size

- 로컬 또는 우선 페이지 교체를 사용하여 효과 제한

6. Thrashing

메모리 참조 패턴의 지역성





Working-set model.

특정 지역을 집중적으로 사용
인접한 것들을 자주 사용

6. Thrashing

작업 집합 모델 Working-Set Model

- Δ ≡ working-set window ≡ 고정된 수의 페이지 참조 예: 10,000 명령어
- WSSi(Process Pi 작업 세트) = 가장 최근 Δ 에서 참조된 총 페이지 수(시간에 따라 다름)
  - Δ 가 너무 작으면 전체 지역을 포함하지 않다.
  - Δ 가 너무 크면 여러 지역을 포함.
  - Δ = ∞ ⇒ 이면 전체 프로그램을 포함.
- D = Σ WSSi ≡ 총 수요 프레임
  - 지역의 근사

  - if D > m ⇒ 스레싱
  - 정책이 D > m이면 프로세스 중 하나를 일시 중지하거나 교체.

6. Thrashing

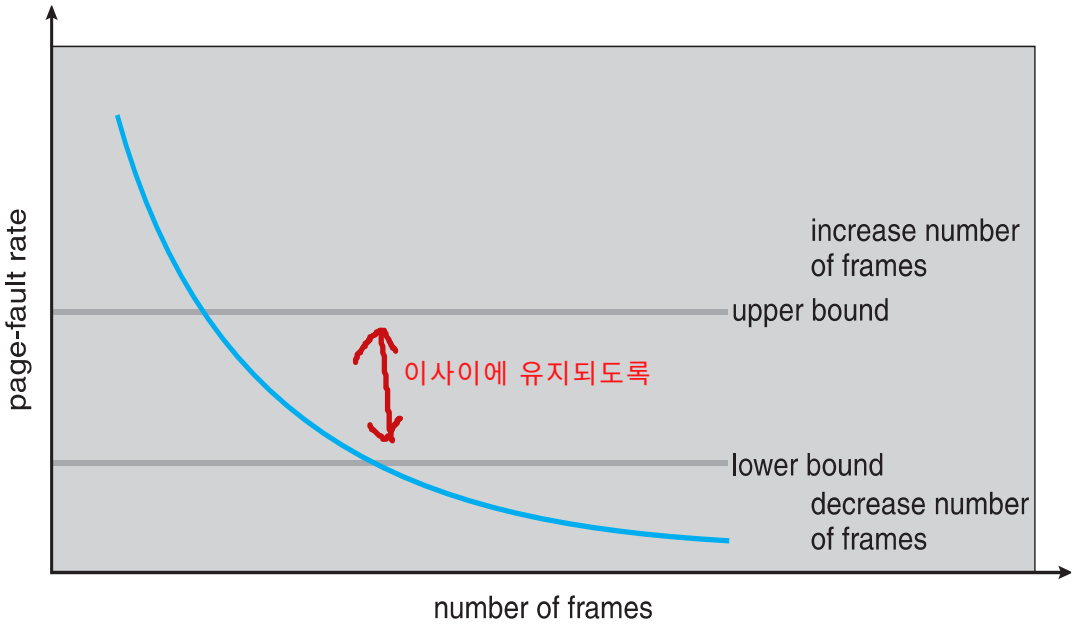
작업 세트 추적

- 간격 타이머 interval timer + 기준 비트reference bit로 근사
- 예: Δ = 10,000
  - 5000 시간 단위마다 타이머 인터럽트
  - 각 페이지에 대해 2비트를 메모리에 유지
  - 타이머가 복사를 중단하고 모든 참조 비트의 값을 0으로 설정할 때마다
  - 메모리의 비트 중 하나 = 1 ⇒ 작업 집합의 페이지
- 이것이 완전히 정확하지 않은 이유는 무엇일까?
- 개선 = 1000 시간 단위마다 10비트 및 인터럽트

6. Thrashing

Page-Fault Frequency

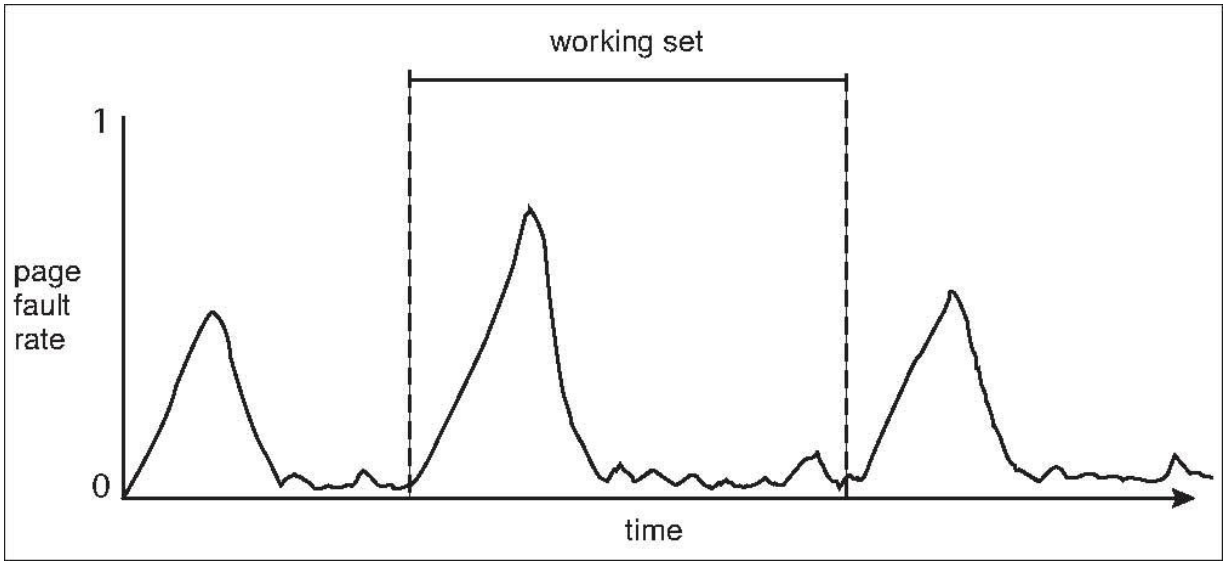
- WSS보다 직접적인 접근 방식
- "허용 가능한 acceptable " PFF(page-fault frequency 페이지 오류 빈도) 비율을 설정하고 로컬 교체 정책을 사용.
- 실제 속도가 너무 낮으면 프로세스에서 프레임 손실
- 실제 속도가 너무 높으면 프로세스가 프레임을 얻는다.



6. Thrashing

작업 집합 및 페이지 오류율

- 프로세스의 작업 집합과 해당 페이지 부재율 사이의 직접적인 관계
- 시간 경과에 따른 작업 세트 변경
- 시간이 지남에 따라 봉우리 **Peaks** 와 계곡 **valleys**



7. Allocating Kernel Memory

기본 개념

kernel: 운영체제를 동작시키기 위해 사용자에게 제공되는 인터페이스

- 사용자 메모리와 다르게 취급됨
- 종종 여유 메모리 풀에서 할당됨
- 커널은 다양한 크기의 구조에 대한 메모리를 요청.
- 일부 커널 메모리는 연속적이어야 합니다.
- 즉, 장치 I/O용

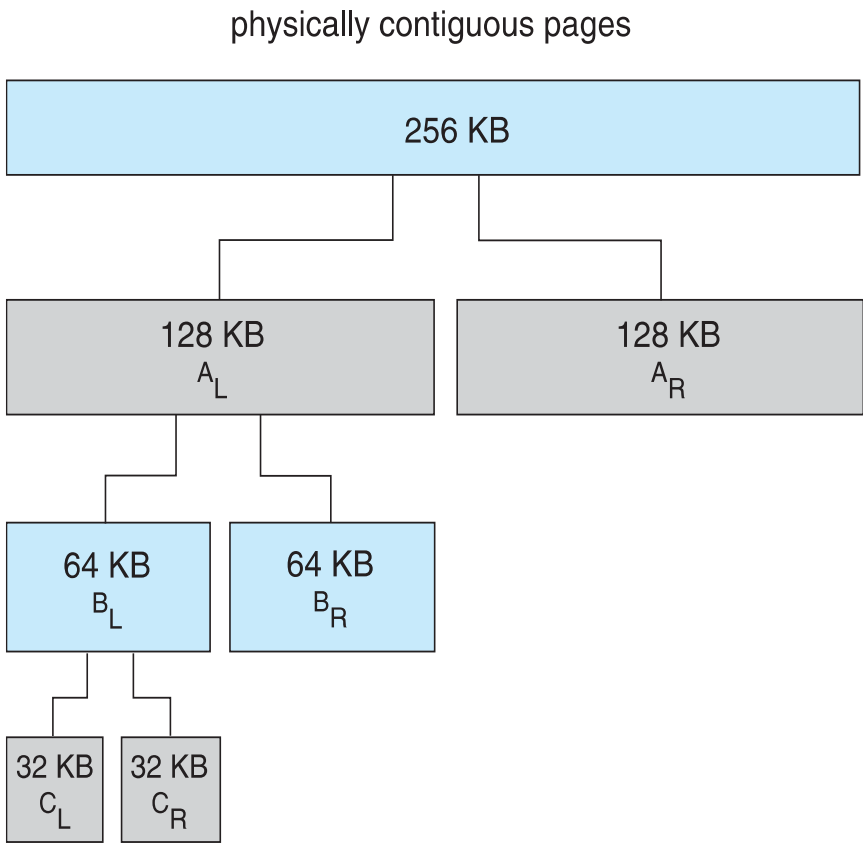
7. Allocating Kernel Memory

Buddy System

- 물리적으로 인접한 페이지로 구성된 고정 크기 세그먼트에서 메모리를 할당.
  - 2의 거듭제곱 할당자power-of-2 allocator를 사용하여 할당된 메모리
  - 2의 거듭제곱 단위로 요청을 충족.
  - 다음으로 높은 2의 거듭제곱으로 반올림된 요청
  - 사용 가능한 것보다 작은 할당이 필요한 경우 현재 청크는 다음으로 낮은 2의 거듭제곱의 두 버디로 분할.
  - 적절한 크기의 청크를 사용할 수 있을 때까지 계속
- 예를 들어 사용 가능한 청크가 256KB이고 커널 요청이 21KB라고 가정.
  - 각각 128KB의 AL 및 AR로 분할
  - 하나는 64KB의 BL과 BR로 더 나뉩니다.
  - 각각 32KB의 CL 및 CR에 추가로 하나씩 - 요청을 충족하는 데 사용되는 하나
- 장점 Advantage – 사용하지 않는 청크를 더 큰 청크로 빠르게 통합 coalesce
- 단점 Disadvantage - 단편화

7. Allocating Kernel Memory

Buddy System 할당자



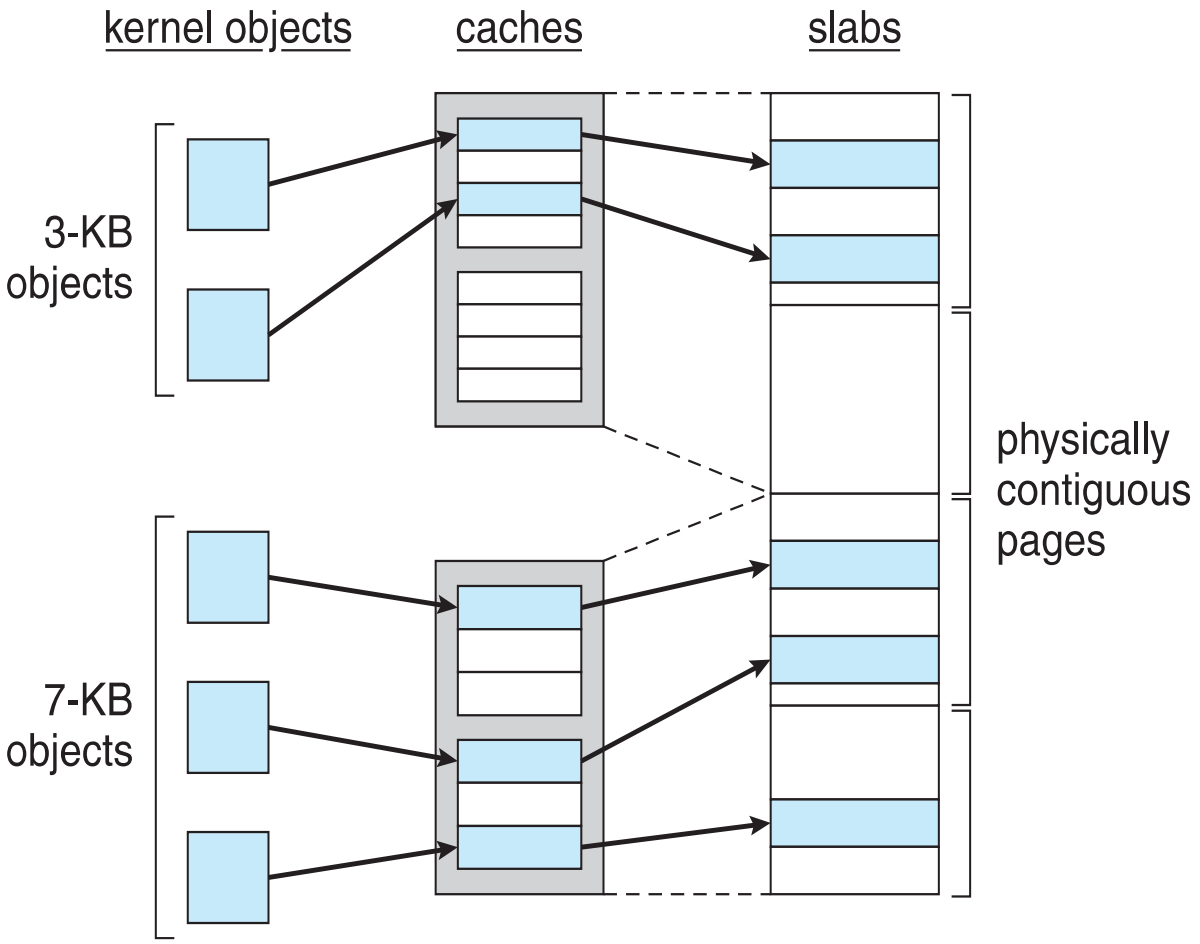
7. Allocating Kernel Memory

Slab Allocator

- 대체 전략
- 슬랩 Slab 은 물리적으로 인접한 하나 이상의 페이지.
- 캐시 Cache 는 하나 이상의 슬랩으로 구성.
- 각각의 고유한 커널 데이터 구조에 대한 단일 캐시
  - 개체로 채워진 각 캐시 – 데이터 구조의 인스턴스화
- 캐시가 생성되면 사용 가능한 것 free 으로 표시된 개체로 채워짐
- 구조가 저장되면 사용됨 used 으로 표시된 객체
- 슬래브가 사용된 객체로 가득차면 빈 슬래브에서 다음 객체 할당
  - 빈 슬래브가 없으면 새 슬래브가 할당됨
- 조각화 없음, 빠른 메모리 요청 만족 등의 이점

7. Allocating Kernel Memory

Slab Allocation



7. Allocating Kernel Memory

Slab Allocator in Linux

- 예를 들어 프로세스 설명자는 struct task_struct 유형.
- 약 1.7KB의 메모리
- 새 작업 -> 캐시에서 새 구조체 할당
- 기존 자유 `struct task_struct` 를 사용.
- 슬래브는 세 가지 가능한 상태일 수 있다.
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used

- 요청 시 슬랩 할당자

1. 부분 슬래브에서 자유 구조체 사용
2. 없는 경우 빈 슬래브에서 가져온다.
3. 빈 슬래브가 없으면 새 빈 슬래브를 만든다.

7. Allocating Kernel Memory

Slab Allocator in Linux

- lab은 솔라리스에서 시작하여 현재 다양한 OS의 커널 모드와 사용자 메모리 모두에 널리 사용.
- Linux 2.2에는 SLAB가 있었고 이제 SLOB 및 SLUB 할당자가 모두 있다.

• 메모리가 제한된 시스템의 SLOB

✓ 단순 블록 목록 – 소형, 중형, 대형 객체에 대한 3개의 목록 객체를 유지합니다.

• SLUB는 성능에 최적화되어 있다. SLAB는 페이지 구조에 저장된 메타데이터인 CPU별 대기열을 제거.

『10과목』

7-8교시 :
Mass-Storage Systems

Syllabus

학습목표

- 이 워크샵에서는 보조 저장 장치의 물리적 구조와 장치 구조가 용도에 미치는 영향 설명을 할 수 있다.
- 대용량 저장 장치의 성능 특성 설명을 할 수 있다.
- I/O 스케줄링 알고리즘 평가를 할 수 있다.
- RAID를 포함하여 대용량 저장소에 제공되는 운영 체제 서비스에 대해 논의를 할 수 있다.

Syllabus

눈높이 체크

- 대량 저장 구조을 알고 계신가요?
- HDD 스케줄링을 알고 계신가요?
- NVM 스케줄링을 알고 계신가요?
- 오류 감지 및 수정을 알고 계신가요?
- 저장 장치 관리를 알고 계신가요?
- 스왑 공간 관리를 알고 계신가요?
- 스토리지 첨부 파일을 알고 계신가요?
- RAID 구조를 알고 계신가요?

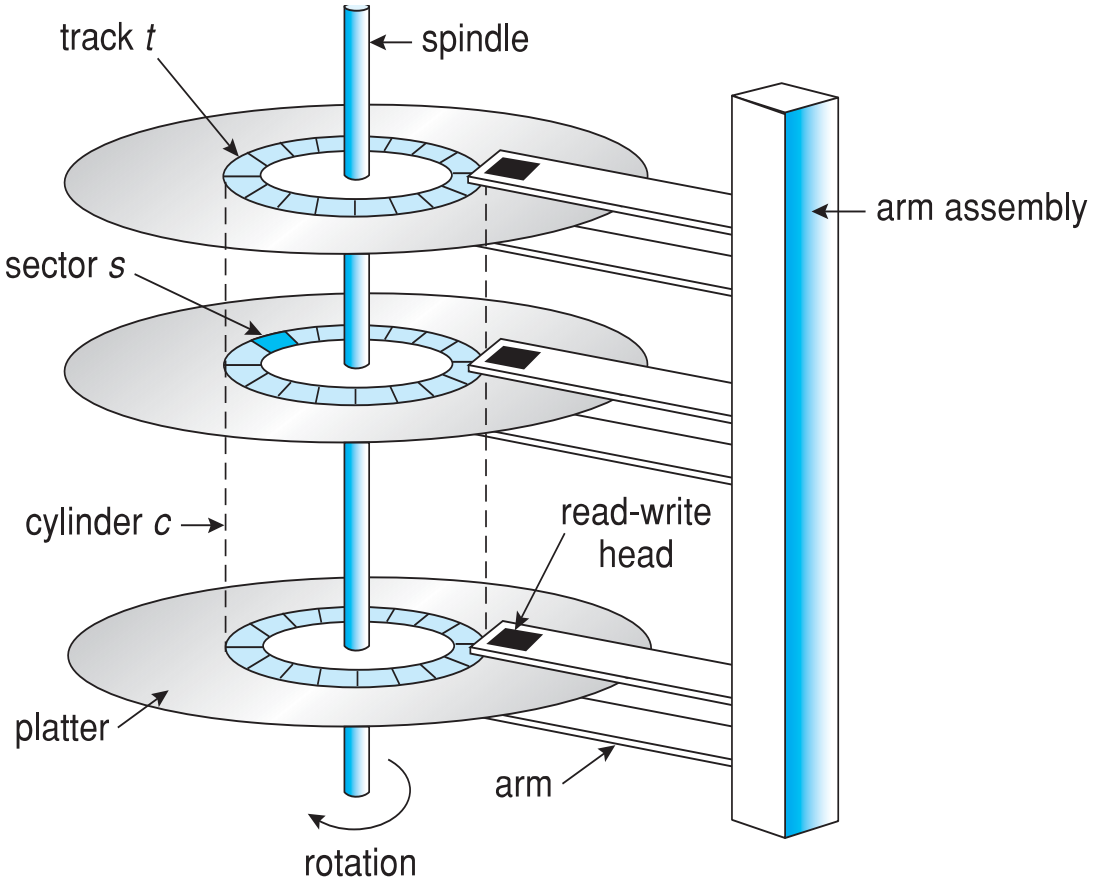
1. 대용량 저장소 구조 개요

대용량 저장소 구조 Mass Storage Structure

- 최신 컴퓨터의 대량 보조 스토리지는 HDD(hard disk drives 하드 디스크 드라이브) 및 NVM(nonvolatile memory 비휘발성 메모리) 장치.
- 움직이는 읽기-쓰기 read-write 헤드 아래에서 자기 코팅된 재료로 된 HDD 회전 플래터
  - 드라이브는 초당 60~250회 회전.
  - 전송 속도는 드라이브와 컴퓨터 간의 데이터 흐름 속도.
  - 포지셔닝 시간(랜덤 액세스 시간)은 디스크 암을 원하는 실린더로 이동하는 시간(탐색 시간)과 원하는 섹터가 디스크 헤드 아래에서 회전하는 시간(회전 대기 시간).
  - 헤드 충돌은 디스크 헤드가 디스크 표면과 접촉하여 발생. 좋지 않다.
- 디스크는 이동식일 수 있다.

1. 대용량 저장소 구조 개요

Moving-head Disk Mechanism



1. 대용량 저장소 구조 개요

Hard Disk Drives

- 플래터 범위는 .85"에서 14"(역사적으로)
- 일반적으로 3.5", 2.5" 및 1.8"
- 드라이브당 30GB~3TB 범위
- 성능

- 전송 속도 Transfer Rate – 이론적 – 6Gb/초
- 효과적인 전송 속도 Effective Transfer Rate – 실제 – 1Gb/초
- 3ms에서 12ms까지 탐색 시간 Seek time – 데스크탑 드라이브에 일반적으로 9ms
- 트랙의 1/3을 기준으로 측정 또는 계산된 평균 검색 시간 Average seek time
- 스핀들 속도에 따른 대기 시간 Latency
- 1 / (RPM / 60) = 60 / RPM
- 평균 대기 시간 Average latency = ½ 대기 시간

Hard Disk Performance

- Access Latency = Average access time = average seek time + average latency
  - For fastest disk 3ms + 2ms = 5ms
  - For slow disk 9ms + 5.56ms = 14.56ms
- Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead
- For example to transfer a 4KB block on a 7200 RPM disk with a 5ms average seek time, 1Gb/sec transfer rate with a .1ms controller overhead =
  - 5ms + 4.17ms + 0.1ms + transfer time =
  - Transfer time = 4KB / 1Gb/s * 8Gb / GB * 1GB / 1024^2KB = 32 / (1024^2) = 0.031 ms
  - Average I/O time for 4KB block = 9.27ms + .031ms = 9.301ms

1. 대용량 저장소 구조 개요

최초의 상용 디스크 드라이브

- 1956년
- IBM RAMDAC 컴퓨터에는 IBM Model 350 디스크 스토리지 시스템이 포함되어 있다.

5M(7비트) 문자
50 x 24" 플래터
액세스 시간 = < 1초

1. 대용량 저장소 구조 개요

비휘발성 메모리 장치 Nonvolatile Memory Devices

- 디스크 드라이브와 같은 경우 솔리드 스테이트 디스크(SSD)라고 함
- 다른 형태로는 USB 드라이브(썸 드라이브, 플래시 드라이브), DRAM 디스크 교체, 마더보드의 표면 장착, 스마트폰과 같은 장치의 기본 스토리지가 있다.
- HDD보다 안정적일 수 있음
- MB당 더 비쌈
- 수명이 짧을 수 있으므로 세심한 관리 필요
- 적은 용량
- 그러나 훨씬 더 빠름
- 버스가 너무 느릴 수 있음 -> 예를 들어 PCI에 직접 연결
- 움직이는 부품이 없으므로 탐색 시간이나 회전 대기 시간이 없다.

1. 대용량 저장소 구조 개요

비휘발성 메모리 장치 Nonvolatile Memory Devices

- 문제를 제시하는 특성을 가지고 있다.
- "페이지" 단위로 읽고 쓰지만(섹터 생각) 제자리에 덮어쓸 수 없음
  - 먼저 지워야 하며 지우기는 더 큰 "블록" 단위로 발생.
  - 닳기 전에 제한된 횟수만 지울 수 있다 – ~ 100,000
  - DWPD(drive writes per day 드라이브 쓰기/일)로 측정된 수명
  - 5DWPD 등급의 1TB NAND 드라이브는 보증 기간 내에 실패 없이 하루에 5TB를 기록할 것으로 예상.

1. 대용량 저장소 구조 개요

NAND 플래시 컨트롤러 알고리즘

- 덮어쓰지 않으면 페이지는 유효한 데이터와 유효하지 않은 데이터가 혼합되어 끝난다.
- 유효한 논리 블록을 추적하기 위해 컨트롤러는 FTL(Flash Translation Layer) 테이블을 유지합니다.
- 유효하지 않은 페이지 공간을 확보하기 위해 가비지 수집 garbage collection 도 구현.
- GC를 위한 작업 공간을 제공하기 위해 오버프로비저닝 overprovisioning 을 할당.
- 각 셀에는 수명이 있으므로 모든 셀에 균등하게 쓰기 위해서는 웨어 레벨링 wear leveling 이 필요.

1. 대용량 저장소 구조 개요

휘발성 메모리 Volatile Memory

- 대용량 저장 장치로 자주 사용되는 DRAM
  - 휘발성 때문에 기술적으로 보조 스토리지는 아니지만 파일 시스템을 가질 수 있고 매우 빠른 보조 스토리지처럼 사용할 수 있다.
- RAM 드라이브(RAM 디스크를 포함하여 많은 이름 포함)는 원시 블록 장치로 제공되며 일반적으로 파일 시스템 형식
- 컴퓨터에는 RAM을 통한 버퍼링, 캐싱 기능이 있는데 왜 RAM이 구동할까?
  - 프로그래머, 운영 체제, 하드웨어에 의해 할당/관리되는 캐시/버퍼
  - 사용자가 제어하는 RAM 드라이브
  - 모든 주요 운영 체제에서 발견됨
  - ✓ Linux /dev/ram, 생성할 macOS diskutil, 파일 시스템 유형 tmpfs의 Linux /tmp
- 고속 임시 저장소로 사용
  - 프로그램은 RAM 드라이브에 읽기/쓰기를 통해 대량 날짜를 빠르게 공유할 수 있다.

1. 대용량 저장소 구조 개요

Disk Structure

- 디스크 드라이브는 논리 블록의 큰 1차원 어레이로 주소가 지정되며 논리 블록은 전송의 최소 단위.
  - 로우 레벨 포맷은 물리적 미디어에 논리 블록 logical blocks 을 생성.
- 논리 블록의 1차원 배열은 디스크의 섹터에 순차적으로 매핑.
  - 섹터 0은 가장 바깥쪽 실린더에 있는 첫 번째 트랙의 첫 번째 섹터입니다.
  - 매핑은 해당 트랙, 해당 실린더의 나머지 트랙, 그리고 가장 바깥쪽에서 가장 안쪽으로 나머지 실린더를 통해 순서대로 진행.
- 물리적 주소에 대한 논리는 쉬워야 한다.
  - 불량 섹터 제외
  - 일정한 각속도를 통한 트랙당 섹터 수가 일정하지 않음

1. 대용량 저장소 구조 개요

Disk Attachment

- I/O 버스와 통신하는 I/O 포트를 통해 액세스되는 호스트 연결 스토리지
- ATA(Advanced Technology Attachment), SATA(Serial ATA), eSATA, SAS(Serial Attached SCSI), USB(Universal Serial Bus) 및 FC(Fibre Channel)를 비롯한 여러 버스를 사용할 수 있다.
- 가장 일반적인 것은 SATA.
- NVM은 HDD보다 훨씬 빠르기 때문에 PCI 버스에 직접 연결되는 NVMe(NVM Express)라는 NVM을 위한 새로운 고속 인터페이스
- 컨트롤러(또는 호스트 버스 어댑터 host-bus adapters, HBA)라고 하는 특수 전자 프로세서에 의해 수행되는 버스의 데이터 전송

- 버스의 컴퓨터 끝에 있는 호스트 컨트롤러, 장치 끝에 있는 장치 컨트롤러
- 컴퓨터는 메모리 매핑된 I/O 포트를 사용하여 호스트 컨트롤러에 명령을 내립니다.
- 호스트 컨트롤러는 장치 컨트롤러에 메시지를 보냅니다.
- 장치와 컴퓨터 DRAM 간에 DMA를 통해 전송되는 데이터

2. Scheduling

HDD Scheduling

- 운영 체제는 하드웨어를 효율적으로 사용할 책임이 있다. 디스크 드라이브의 경우 이는 빠른 액세스 시간과 디스크 대역폭을 의미.
- 탐색 시간 최소화
- 탐색 시간 Seek time ≈ 탐색 거리seek distance
- 디스크 대역폭 Disk bandwidth 은 전송된 총 바이트 수를 서비스에 대한 첫 번째 요청과 마지막 전송 완료 사이의 총 시간으로 나눈 값.

2. Scheduling

Disk Scheduling

- 많은 디스크 I/O 요청 소스가 있다.
  - 운영체제
  - 시스템 프로세스
  - 사용자 프로세스
- I/O 요청에는 입력 또는 출력 모드, 디스크 주소, 메모리 주소, 전송할 섹터 수가 포함.
- OS는 디스크 또는 장치별로 요청 대기열을 유지.
- 유휴 디스크는 I/O 요청 시 즉시 작업 가능, 디스크 사용량이 많으면 작업이 대기해야 함
  - 최적화 알고리즘은 대기열이 존재할 때만 의미가 있다.
- 과거에는 대기열 관리, 디스크 드라이브 헤드 스케줄링을 담당하는 운영 체제
  - 이제 저장 장치에 내장된 컨트롤러
  - LBA 제공, 요청 정렬 처리

2. Scheduling

Disk Scheduling

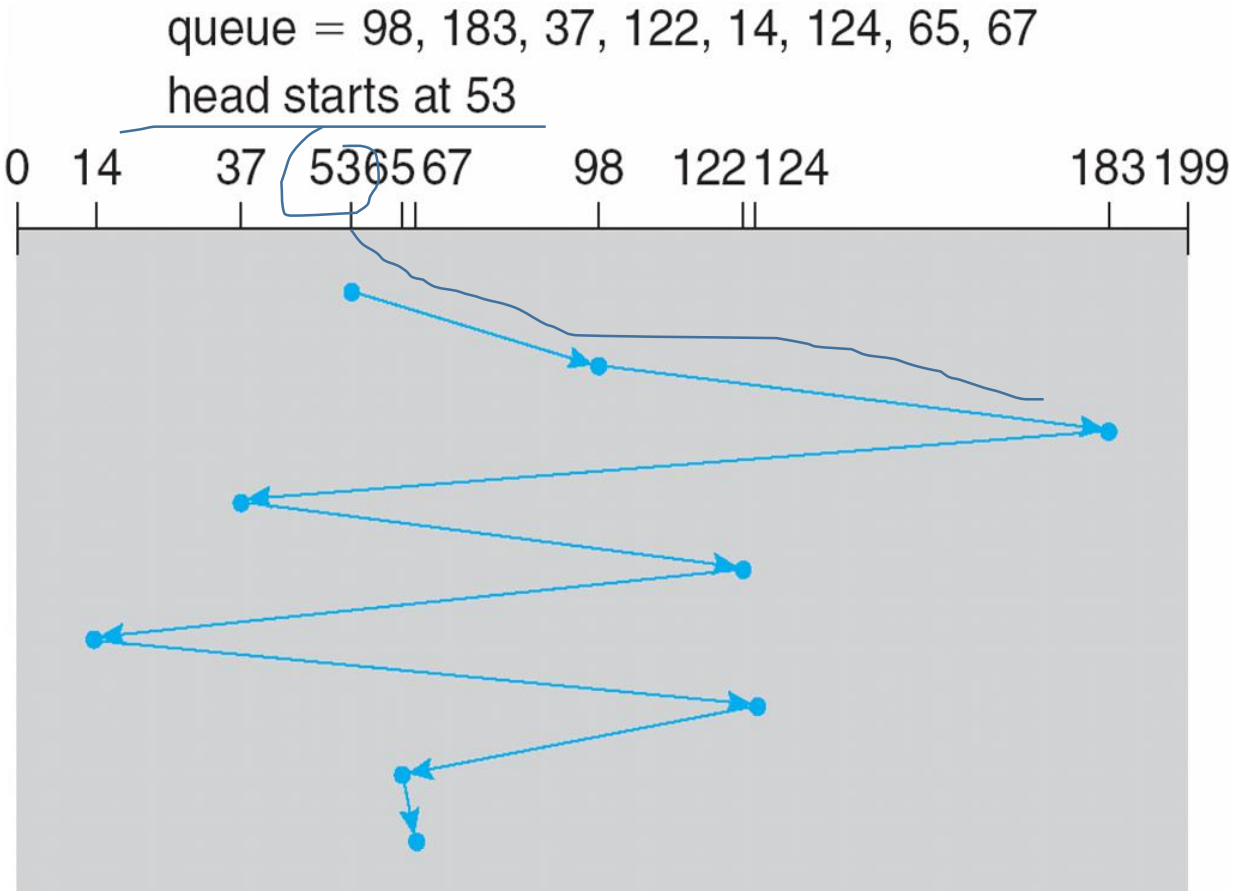
- 드라이브 컨트롤러에는 작은 버퍼가 있으며 다양한 "깊이"의 I/O 요청 대기열을 관리할 수 있다.
- 디스크 I/O 요청 서비스를 예약하기 위한 여러 알고리즘이 있다.
- 분석은 하나 이상의 플래터에 대해 사실.
- 요청 대기열(0-199)로 스케줄링 알고리즘을 설명.

98, 183, 37, 122, 14, 124, 65, 67
헤드 포인터 53

2. Scheduling

FCFS

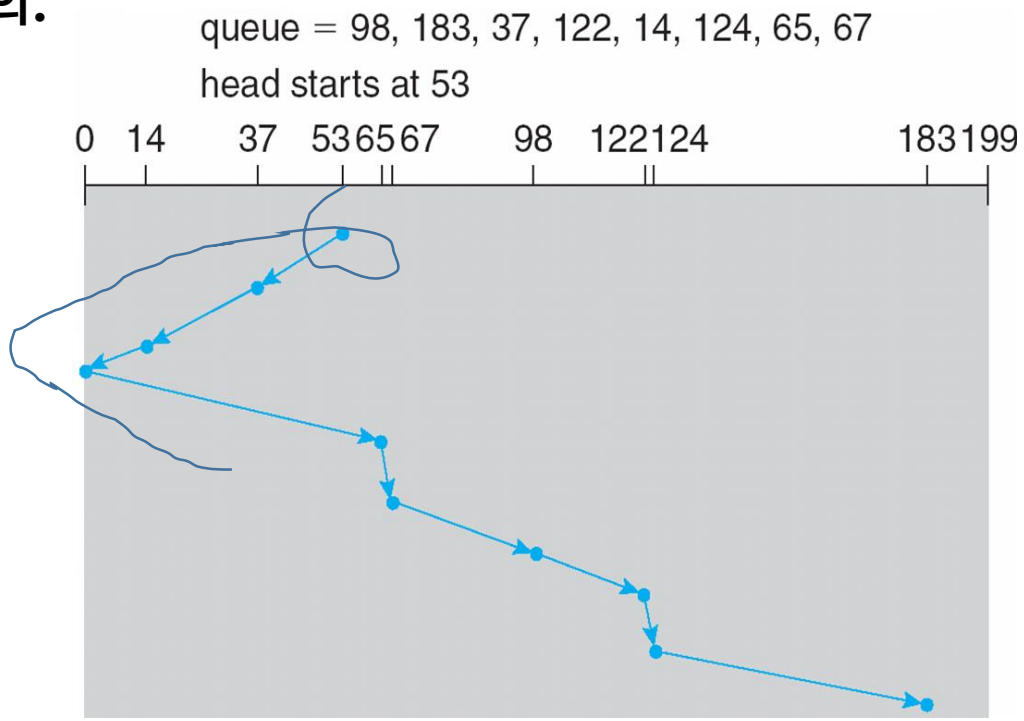
- 그림은 640개 실린더의 전체 헤드 움직임을 보여준다.



2. Scheduling

SCAN

- 디스크 암은 디스크의 한쪽 끝(0으로 이동해서) 에서 시작하여 다른 쪽 끝으로 이동하여 디스크의 다른 쪽 끝에 도달할 때까지 요청에 서비스를 제공. 여기서 헤드 이동은 역전되어 서비스가 계속.
- SCAN 알고리즘 엘리베이터 알고리즘이라고도 함
- 그림은 208개 실린더의 전체 헤드 이동을 보여준다.
- 그러나 요청이 균일하게 밀집된 경우 디스크의 다른 쪽 끝에서 밀도가 가장 크고 대기 시간이 가장 길다는 점에 유의.



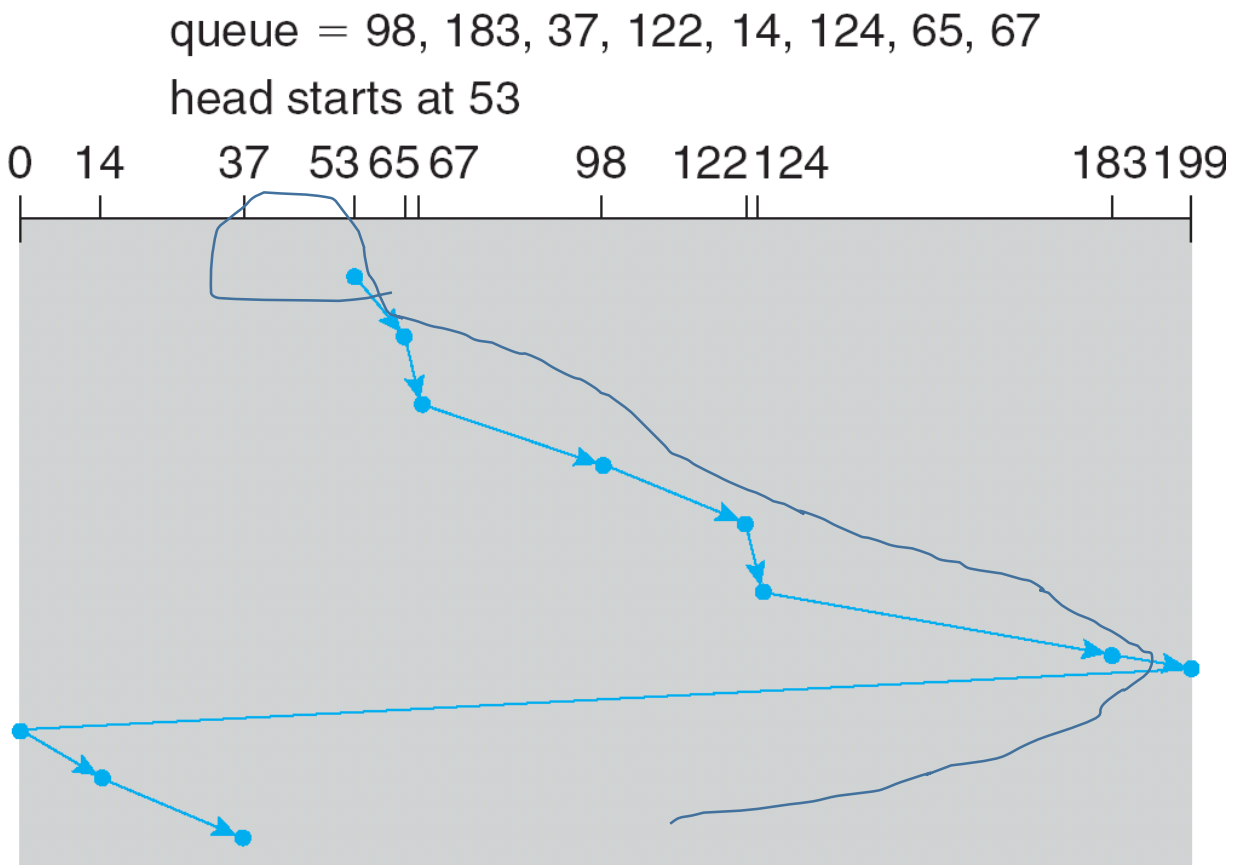
2. Scheduling

C-SCAN

- SCAN보다 균일한 대기 시간 제공
- 헤드는 디스크의 한쪽 끝에서 다른 쪽 끝으로 이동하면서 요청을 처리.
- 그러나 다른 끝에 도달하면 반환 여행에서 요청을 서비스하지 않고 즉시 디스크의 시작 부분으로 돌아간다.
- 실린더를 마지막 실린더에서 첫 번째 실린더로 둘러싸는 순환 목록으로 처리.
- 총 실린더 수?
- 183개 실린더의 전체 헤드 이동

2. Scheduling

C-SCAN



2. Scheduling

디스크 스케줄링 알고리즘 선택

- SSTF는 일반적이며 자연스러운 매력이 있다.
- SCAN 및 C-SCAN은 디스크에 과부하가 걸리는 시스템에서 더 잘 수행.
  - starvation 이 적지만 여전히 가능
- 기아를 피하기 위해 Linux는 데드라인 deadline 스케줄러를 구현.
  - 별도의 읽기 및 쓰기 대기열을 유지하고 읽기 우선 순위 부여
  - 프로세스가 쓰기보다 읽기를 차단할 가능성이 높기 때문에
  - 4개의 대기열 구현: 2 x 읽기 및 2 x 쓰기
  - LBA 순서로 정렬된 1개의 읽기 및 1개의 쓰기 대기열, 기본적으로 C-SCAN 구현
  - FCFS 순서로 정렬된 1개의 읽기 및 1개의 쓰기 대기열
  - 해당 큐의 순서대로 정렬되어 배치로 전송된 모든 I/O 요청
  - 각 배치 후 FCFS에 구성된 수명(기본값 500ms)보다 오래된 요청이 있는지 확인.
  - 그렇다면 해당 요청을 포함하는 LBA 대기열이 다음 I/O 배치를 위해 선택.
- RHEL 7에서는 NOOP 및 완전 공정 대기열 스케줄러(CFQ)도 사용할 수 있으며 기본값은 저장 장치에 따라 다르다.

2. Scheduling

NVM Scheduling

- 디스크 헤드 또는 회전 대기 시간이 없지만 여전히 최적화할 여지가 있음
- RHEL 7에서는 NOOP(스케줄링 없음)가 사용되지만 인접한 LBA 요청이 결합.

- NVM은 랜덤 I/O에서 최고, HDD는 순차에서 최고
- 처리량은 비슷할 수 있다.
- NVM을 사용하면 초당 입출력 작업 Input/Output operations per second (IOPS)이 훨씬 더 높아집니다(수십만 대 수백).
- 그러나 쓰기 증폭 write amplification (한 번의 쓰기, 가비지 수집 및 많은 읽기/쓰기 유발)은 성능 이점을 감소시킬 수 있다.

3. Error Detection and Correction

기본 개념

- 컴퓨팅의 여러 부분(메모리, 네트워킹, 스토리지)의 기본 측면
- 오류 감지 Error detection 는 문제가 발생했는지 확인합니다(예: 비트 뒤집기).
  - 감지되면 작업을 중단할 수 있다.
  - 패리티 비트를 통해 자주 수행되는 감지
- 패리티 체크섬 checksum 의 한 형태 – 모듈식 산술을 사용하여 고정 길이 단어의 값을 계산, 저장, 비교.
- 네트워킹에서 일반적인 또 다른 오류 감지 방법은 다중 비트 오류를 감지하기 위해 해시 함수를 사용하는 CRC(Cyclic Redundancy Check)입니다.
- 오류 수정 코드 Error-correction code (ECC)는 감지할 뿐만 아니라 일부 오류를 수정할 수 있다.
- 수정 가능한 소프트 오류, 감지되었지만 수정되지 않은 하드 오류

4. Storage Device Management

기본 개념

- 저수준 포맷 또는 물리적 포맷 Low-level formatting, or physical for matting - 디스크 컨트롤러가 읽고 쓸 수 있는 섹터로 디스크 분할
- 각 섹터는 헤더 정보, 데이터, 오류 수정 코드(ECC)를 포함할 수 있다.
- 일반적으로 512바이트의 데이터이지만 선택 가능
- 파일을 보관하기 위해 디스크를 사용하려면 운영 체제는 여전히 자체 데이터 구조를 디스크에 기록해야 합니다.

- 각각 논리 디스크로 취급되는 하나 이상의 실린더 그룹으로 디스크를 분할 Partition 합니다.
- 논리적 서식 Logical formatting 또는 "파일 시스템 만들기 making a file system "
- 효율성을 높이기 위해 대부분의 파일 시스템은 블록을 클러스터 clusters 로 그룹화합니다.
- 블록 단위로 수행되는 디스크 I/ODisk I/O done in blocks
- 클러스터에서 수행되는 파일 I/OFile I/O done in clusters

4. Storage Device Management

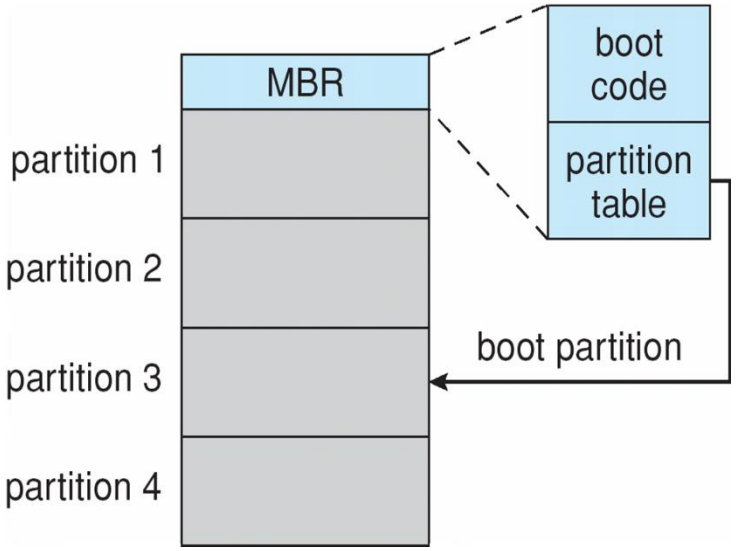
Boot block

- 루트 파티션 Root partition 은 OS를 포함하고, 다른 파티션은 다른 Oses, 다른 파일 시스템을 보유하거나 원시 파티션일 수 있다.
  - 부팅 시 마운트됨
  - 다른 파티션은 자동 또는 수동으로 마운트할 수 있다.
- 마운트 시 파일 시스템 일관성 확인
  - 모든 메타데이터가 정확합니까?
  - 안되면 수정하고 다시 시도
  - 그렇다면 마운트 테이블에 추가하고 액세스를 허용.
- 부트 블록 Boot block 은 파일 시스템에서 커널을 로드하는 방법을 알기에 충분한 코드를 포함하는 블록의 부트 볼륨 또는 부트 로더 세트를 가리킬 수 있다.
  - 또는 다중 OS 부팅을 위한 부팅 관리 프로그램

4. Storage Device Management

Boot block

- 자체 블록 관리를 수행하려는 앱에 대한 원시 디스크 액세스, OS를 방해하지 않음(예: 데이터베이스)
- 부팅 블록이 시스템을 초기화.
- ✓ 부트스트랩은 ROM, 펌웨어에 저장.
- ✓ 부트 파티션의 부트 블록에 저장된 부트스트랩 로더 Bootstrap loader 프로그램
- 불량 블록을 처리하는 데 사용되는 섹터 스페어링 sector sparing 과 같은 방법



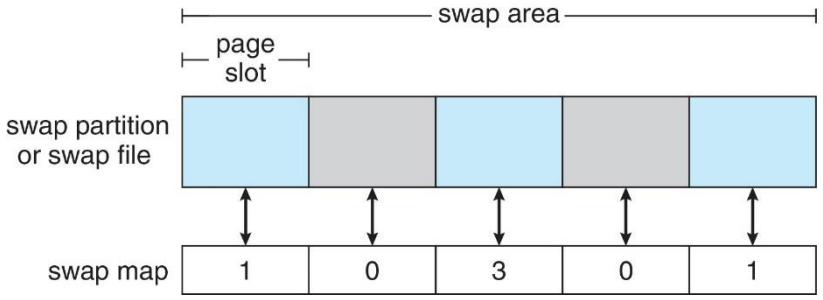
Booting from secondary storage in Windows

5. Swap-Space Management

기본 개념

- DRAM이 모든 프로세스에 비해 충분히 크지 않을 때 DRAM에서 보조 스토리지로 전체 프로세스(스왑) 또는 페이지(페이징)를 이동하는 데 사용.
- 운영 체제는 스왑 공간 관리를 제공.

- DRAM보다 느린 보조 스토리지, 성능 최적화에 매우 중요
- 일반적으로 여러 개의 스왑 공간 가능 - 주어진 장치에서 I/O 로드 감소
- 전용 장치를 사용하는 것이 가장 좋다.
- 원시 파티션 또는 파일 시스템 내의 파일에 있을 수 있음(추가의 편의를 위해)
- Linux 시스템에서 스와핑을 위한 데이터 구조:



6. Storage Attachment

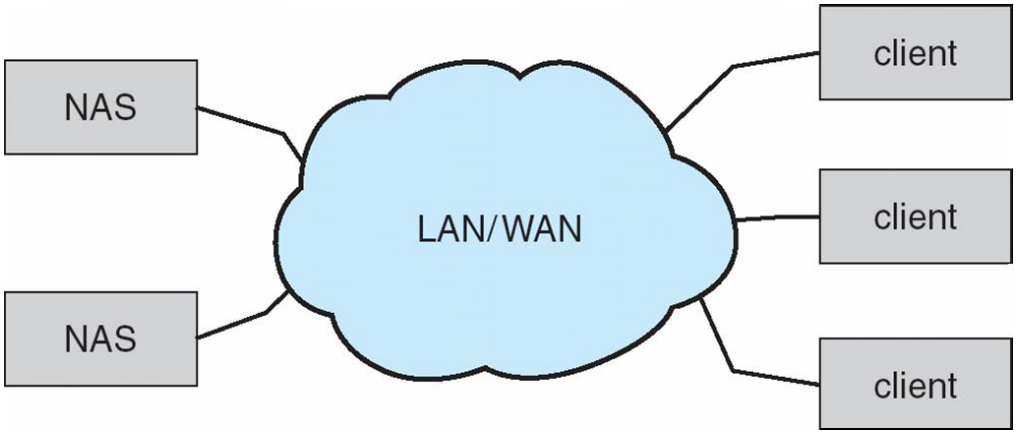
스토리지 첨부 파일

- 컴퓨터는 세 가지 방법으로 스토리지에 액세스.
  - 호스트 연결
  - 네트워크 연결
  - cloud
- 여러 기술 중 하나를 사용하여 로컬 I/O 포트를 통한 호스트 연결 액세스
  - 많은 장치를 연결하려면 USB, Firewire, Thunderbolt와 같은 스토리지 버스를 사용.
  - 하이엔드 시스템 High-end systems 은 파이버 채널(fibre channel FC)을 사용.
  - 광섬유 또는 구리 케이블을 사용하는 고속 직렬 아키텍처
  - 여러 호스트 및 스토리지 장치를 FC 패브릭에 연결할 수 있다.

6. Storage Attachment

네트워크 연결 스토리지

- NAS(Network-attached storage 네트워크 연결 스토리지)는 로컬 연결(예: 버스)이 아닌 네트워크를 통해 사용할 수 있는 스토리지.
  - 파일 시스템에 원격으로 연결
- NFS 및 CIFS는 일반적인 프로토콜.
- IP 네트워크에서 일반적으로 TCP 또는 UDP를 통해 호스트와 스토리지 간의 RPC(remote procedure calls 원격 프로시저 호출)를 통해 구현됨
- iSCSI 프로토콜은 IP 네트워크를 사용하여 SCSI 프로토콜을 전달.
- 장치(블록)에 원격으로 연결



6. Storage Attachment

Cloud Storage

- NAS와 유사하게 네트워크를 통해 스토리지에 대한 액세스를 제공.
  - NAS와 달리 인터넷 또는 WAN을 통해 원격 데이터 센터에 액세스
- NAS는 또 다른 파일 시스템으로 제공되는 반면 클라우드 스토리지는 API를 기반으로 하며 액세스를 제공하기 위해 API를 사용하는 프로그램이 있다.
  - 예: Dropbox, Amazon S3, Microsoft OneDrive, Apple iCloud
  - 대기 시간 및 장애 시나리오로 인해 API 사용(NAS 프로토콜이 제대로 작동하지 않음)

6. Storage Attachment

Storage Array

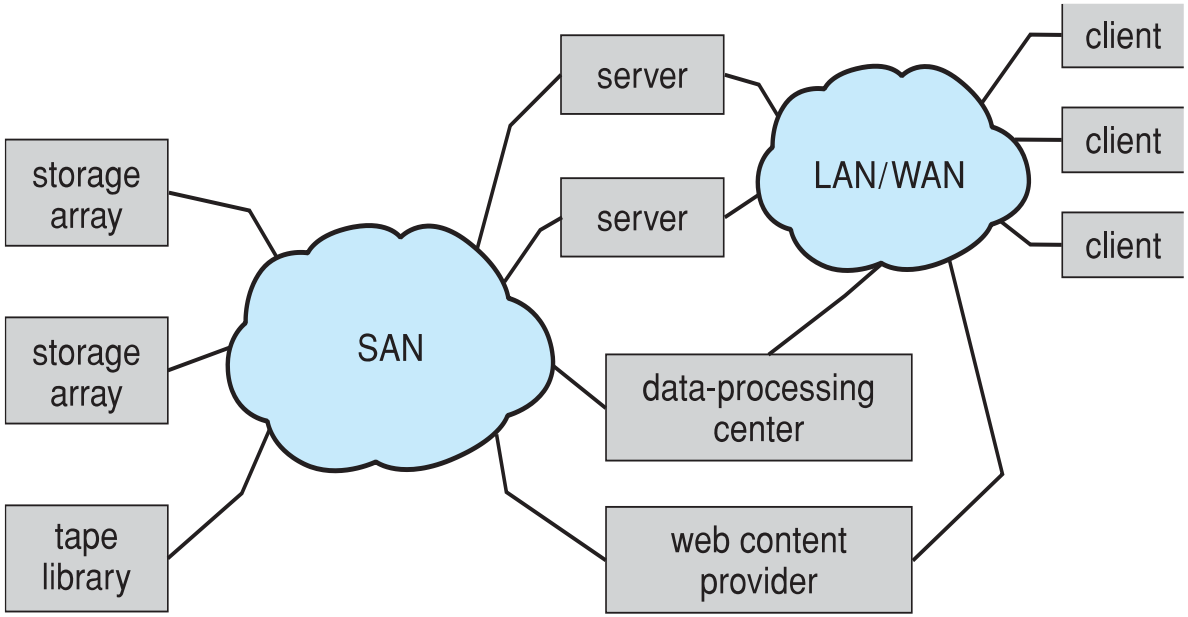
- 디스크 또는 디스크 어레이만 연결할 수 있음
- 네트워크 대역폭을 사용하는 NAS 단점을 방지.
- 스토리지 어레이에는 컨트롤러가 있고 연결된 호스트에 기능을 제공.

- 호스트를 어레이에 연결하기 위한 포트
- 메모리, 소프트웨어 제어(때때로 NVRAM 등)
- 몇 개에서 수천 개의 디스크
- RAID, 핫 스페어, 핫 스왑(나중에 설명)
- 공유 스토리지 -> 효율성 향상
- 일부 파일 시스템에서 발견되는 기능
- 스냅샷, 클론, 씬 프로비저닝, 복제, 중복 제거 등

6. Storage Attachment

Storage Area Network

- 대규모 스토리지 환경에서 공통
- 여러 스토리지 어레이에 연결된 여러 호스트 - 유연성



6. Storage Attachment

Storage Area Network

- SAN은 하나 이상의 스토리지 어레이.
- 하나 이상의 파이버 채널 스위치 또는 IB(InfiniBand) 네트워크에 연결됨
- 호스트도 스위치에 연결
- 특정 어레이에서 특정 서버로 LUN 마스킹을 통해 사용 가능한 스토리지
- 간편한 스토리지 추가 또는 제거, 새 호스트 추가 및 스토리지 할당
- 별도의 스토리지 네트워크와 통신 네트워크가 있는 이유는 무엇일까?
- iSCSI, FCOE

7. RAID Structure

기본 개념

- RAID redundant array of inexpensive disks– 저렴한 디스크의 중복 어레이
- 다중 디스크 드라이브는 중복성을 통해 안정성을 제공.
- 평균 고장 시간 증가mean time to failure
- 평균 수리 시간 Mean time to repair – 또 다른 장애로 인해 데이터 손실이 발생할 수 있는 노출 시간
- 위 요인을 기반으로 한 평균 데이터 손실 시간 Mean time to data loss
- 미러링된 디스크가 독립적으로 실패하는 경우 평균 실패 시간 mean time to failure 이 1300,000이고 평균 복구 시간이 10시간인 디스크를 고려.
- 평균 데이터 손실 시간은 $100,000^2 / (2 * 10) = 500 * 10^6$ 시간 또는 57,000 년
- NVRAM과 자주 결합하여 쓰기 성능 향상
- 디스크 사용 기술의 몇 가지 개선 사항에는 공동으로 작동하는 여러 디스크의 사용이 포함

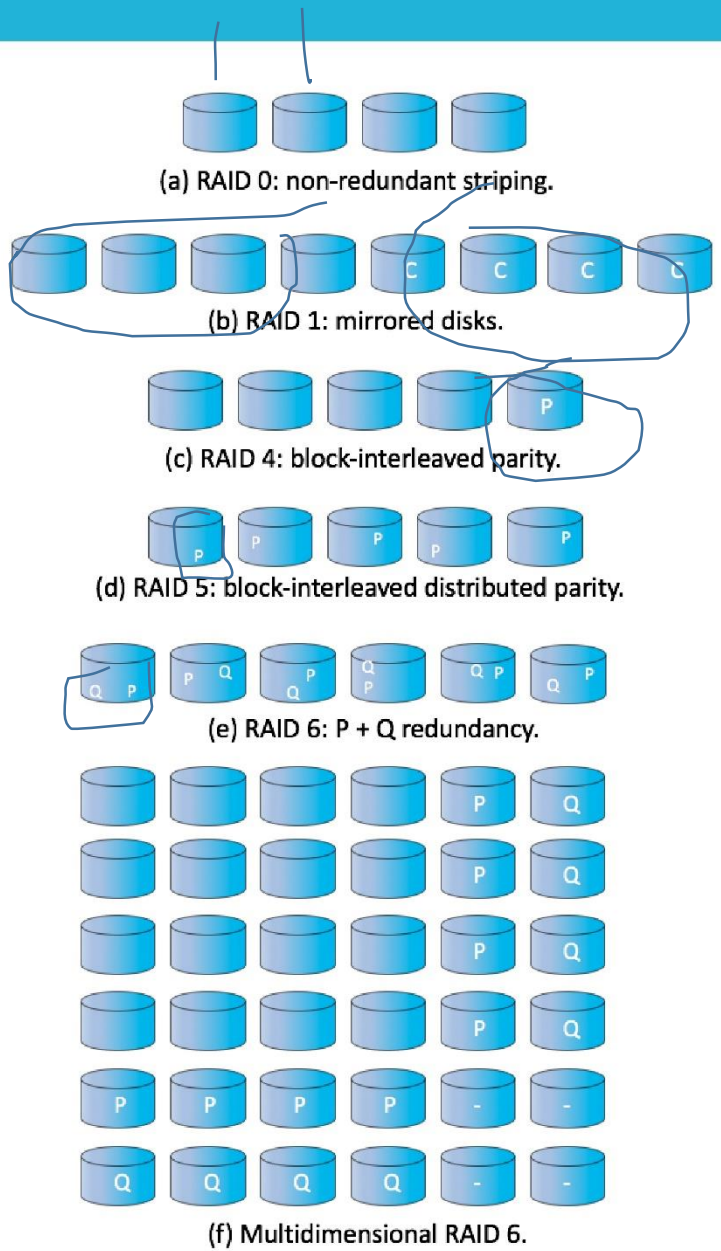
7. RAID Structure

기본 개념

- 디스크 스트라이핑은 디스크 그룹을 하나의 스토리지 단위로 사용.
- RAID는 6개의 다른 레벨로 배열.
- RAID 방식은 중복 데이터를 저장하여 스토리지 시스템의 성능과 안정성을 향상.
  - Mirroring or shadowing(RAID 1)은 각 디스크의 복제본을 유지.
  - Striped mirrors(RAID 1+0) 또는 mirrored stripes (RAID 0+1)는 고성능과 높은 안정성을 제공.
  - Block interleaved parity(RAID 4, 5, 6)는 훨씬 적은 중복성을 사용.
- 스토리지 어레이 내의 RAID는 어레이에 장애가 발생해도 여전히 장애가 발생할 수 있으므로 어레이 간 데이터 자동 복제가 일반적.
- 종종 적은 수의 핫 스페어 hot-spare 디스크가 할당되지 않은 상태로 남아 있어 장애가 발생한 디스크를 자동으로 교체하고 데이터를 재구성.

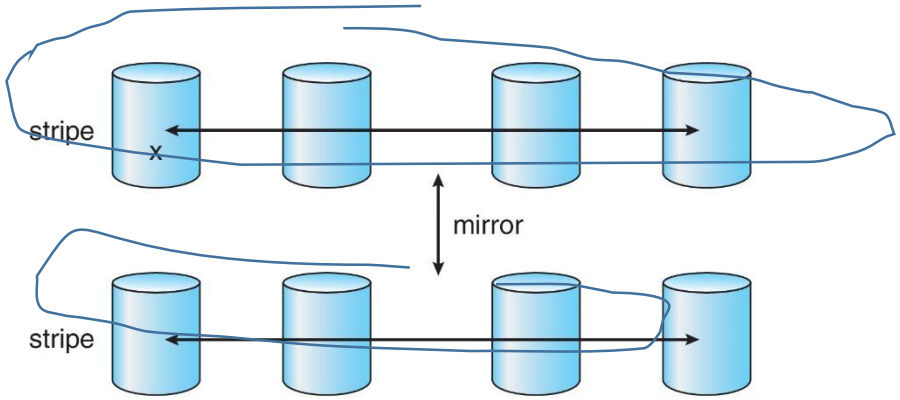
7. RAID Structure

RAID Levels

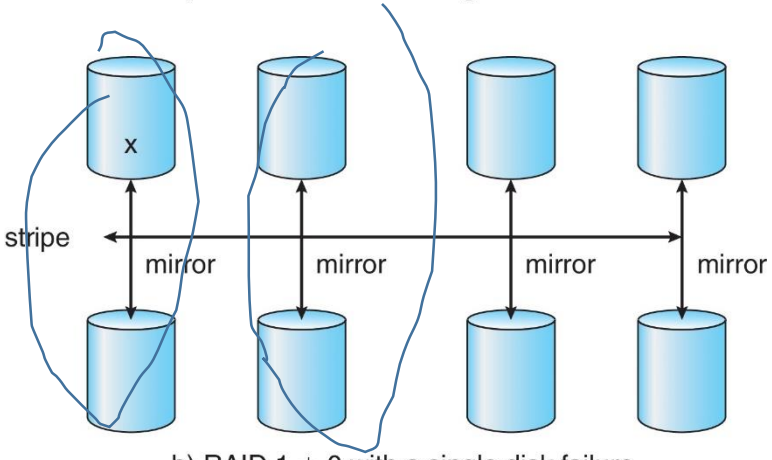


7. RAID Structure

RAID (0 + 1) and (1 + 0)



a) RAID 0 + 1 with a single disk failure.



b) RAID 1 + 0 with a single disk failure.