교착 상태(deadlock)

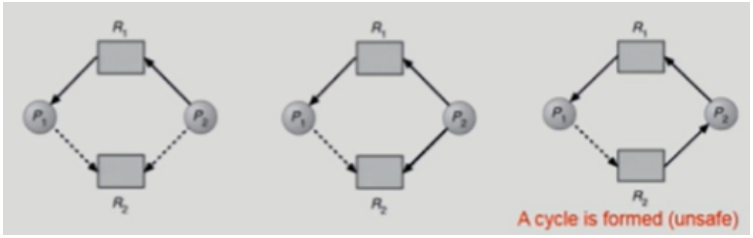
- 자원을 소유한 스레드들 사이에서, 각 스레드는 다른 스레드가 소유한 자원을 요청하여 무한정 대기하고 있는 현상

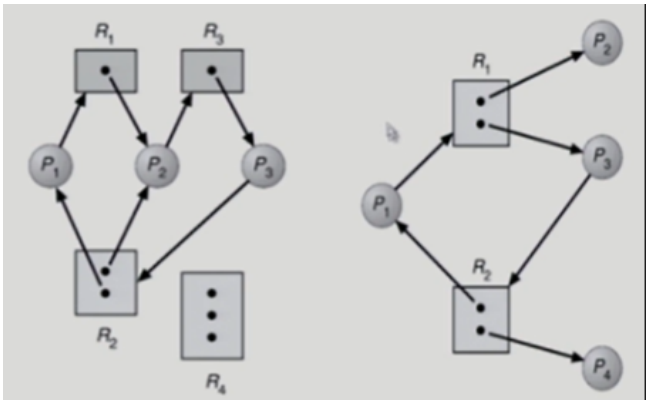
Deadlcok 발생의 4가지 조건

- Mutual exclusion (상호배제)
  - 매 수간 하나의 프로세스만이 자원을 사용할 수 있음
- No preemption (비선점)
  - 프로세스는 자원을 스스로 내어놓을 뿐 강제로 빼앗기지 않음
- Hold and wait (보유 대기)
  - 자원을 가진 프로세스가 다른 자원을 기다릴 때 보유 자원을 놓지 않고 계속 가지고 있음
- Circular wait (환형 대기)
  - 자원을 기다리는 프로세스간에 사이클이 형성되어야 함

Resource-Allocation Graph (자원할당그래프)

: Cycle 생성 여부 조사시 프로세스의 수가 n일 때 O(n^2) 시간이 걸린다



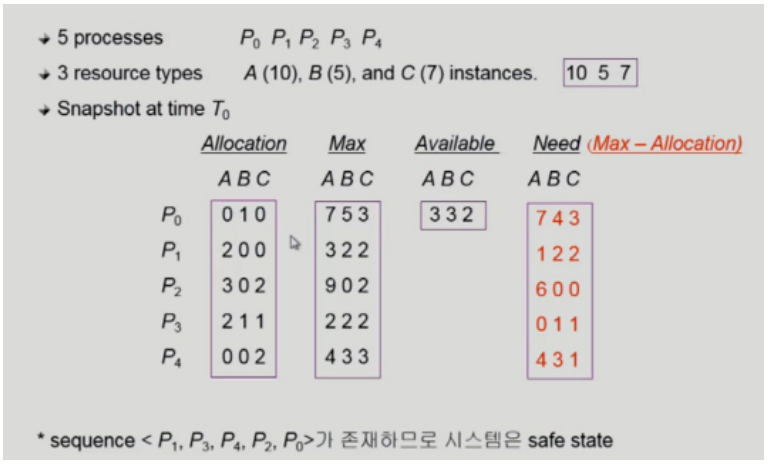


- 그래프에 cycle이 없으면 deadlock이 아니다
- 그래프에 cycle이 있으면
  - 자원 유형당 인스턴스가 하나뿐이라면 교착 상태 발생
  - 자원 유형당 여러 인스턴스가 있다면 교착 상태 발생 가능성 존재

Deadlock의 처리 방법

1. Deadlock Prevention (교착상태 예방)
   - Mutual Exclusion ⟶ 상호 배제 없애기
     - 공유해서는 안되는 자원의 경우 반드시 성립해야 함
   - Hold and Wait ⟶ 기다리지 않게 하기
     - 방법 1. 프로세스 시작 시 모든 필요한 자원을 할당받게 하는 방법
     - 방법 2. 자원이 필요할 경우 보유 자원을 모두 놓고 다시 요청
   - No Preemption ⟶ 선점 허용
     - State(문맥)를 쉽게 save하고 restore(복원)할 수 있는 자원에서 주로 사용
   - Circular Wait ⟶ 환형 대기 제거
     - **모든 자원 유형에 할당 순서**를 정하여 정해진 순서대로만 자원 할당
2. Deadlock Avoidance (교착상태 회피)

- 자원 요청에 대한 부가적인 정보를 이용해서 자원 할당이 deadlock으로부터 안전(safe)한지를 동적으로 조사해서 안전한 경우에만 할당
- safe state
  - 시스템 내의 프로세스들에 대한 safe sequence가 존재하는 상태
- safe sequence
  - 프로세스의 sequence <$P_1, P_2, \ldots, P_n$>이 safe하려면 P_i 의 자원 요청이 **"가용 자원 + 모든 $P_j(j < i)$ 의 보유 자원"** 에 의해 충족 되어야 함
- Banker's 알고리즘



3. Deadlcok Detection and recovery (교착상태 감지 및 복구)
   - 자원 강제 선점(preemption)
   - 롤백(rollback)
   - 스레드 강제 종료(kill process)
4. Deadlock Ignorance (교착상태 무시)
   - Deadlock이 일어나지 않는다고 생각하고 아무런 조치도 취하지 않음
   - 타조알고리즘

메모리 계층 구조

- CPU 레지스터 – CPU 캐시 – 메인 메모리 – 보조기억장치

메모리 관리가 필요한 이유

- 메모리는 **공유 자원이기 때문**
- 메모리 **보호되어야 하기 때문**
- 메모리 **용량 한계 극복할 필요**
- 메모리 **효율성 증대를 위해**

물리 주소와 논리 주소

- Logical Address (=virtual address)
  - 프로세스마다 독립적으로 가지는 주소 공간
  - 각 프로세스마다 0번지부터 시작
  - CPU가 보는 주소는 logical address임
- Physical address
  - 메모리에 실제 올라가는 0부터 시작하는 연속된 주소
- MMU (Memory Management Unit)
  - 논리 주소를 물리 주소로 바꾸는 하드웨어 장치
  - MMU는 CPU 패키지에 내장

연속 메모리 할당

: 프로세스별로 연속된 한 덩어리의 메모리 할당

- 고정 크기 할당 (외부, 내부 단편화 발생)
  - 메모리를 **고정 크기의 파티션**으로 나누고 프로세스당 하나의 파티션 할당
  - 파티션의 크기는 모두 같거나 다를 수 있음
  - 메모리가 파티션들로 미리 나누어져 있기 때문에 고정 크기 할당이라고 부름

- 가변 크기 할당 (외부 단편화 발생)
  - 메모리를 **가변 크기의 파티션**으로 나누고 프로세스당 하나의 파티션 할당
- 장점
  - 논리 주소를 물리 주소로 바꾸는 과정이 단순, CPU의 메모리 액세스 속도 빠름
  - 운영체제가 관리할 정보량이 적어서 부담이 덜함
- 단점
  - 메모리 할당의 유연성이 떨어짐. 작은 홀들을 합쳐 충분한 크기의 메모리가 있음에도, 연속된 메모리를 할당할 수 없는 경우 발생

단편화(fragmentation)

: 프로세스에게 할당할 수 없는 조각 메모리들이 생기는 현상, 조각 메모리를 홀이라 부름

- 내부 단편화(internal fragmentation)
  - 할당된 메모리 내부에 사용할 수 없는 홀이 생기는 현상

- 외부 단편화(external fragmentation)
  - 할당된 메모리들 사이에 사용할 수 없는 홀이 생기는 현상

홀 선택 알고리즘 (시험)

- first-fit(최초 적합)
  - 홀 리스트를 검색하여 처음으로 만나는, 요청 크기보다 큰 홀 선택
  - 할당 속도 빠름/단편화 발생 가능성
- best-fit(최적 적합)
  - 홀 리스트를 검색하여 요청 크기를 수용하는 것 중, 가장 작은 홀 선택
  - 크기 별로 홀이 정렬되어 있지 않으면 전부 검색
- worst-fit(최악 적합)
  - 홀 리스트를 검색하여 요청 크기를 수용하는 것 중, 가장 큰 홀 선택
  - 크기 별로 홀이 정렬되어 있지 않으면 전부 검색

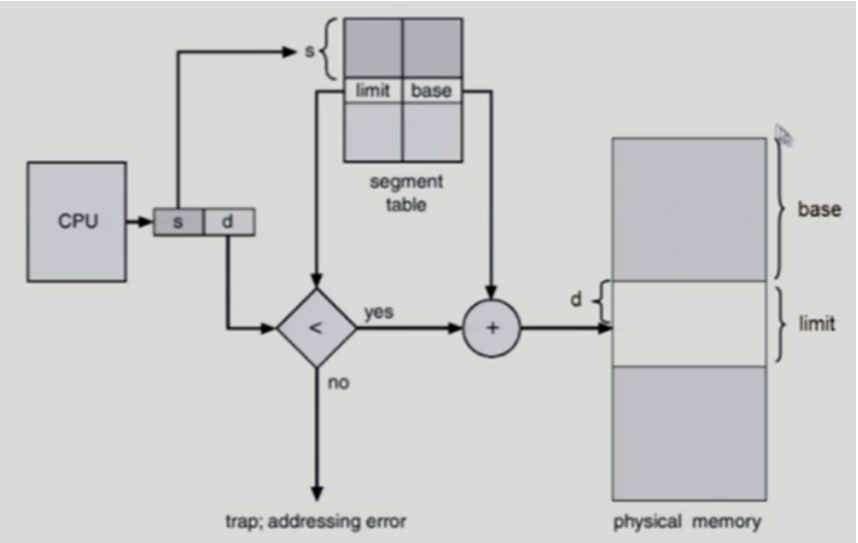
분할 메모리 할당

: 프로세스에게 여러 덩어리의 메모리 할당

- 고정 크기 할당
  - 고정 크기의 덩어리 메모리를 여러 개 분산 할당 ex) 페이지(paging) 기법
- 가변 크기 할당
  - 가변 크기의 덩어리 메모리를 여러 개 분산 할당 ex) 세그먼테이션 기법

Segmentation

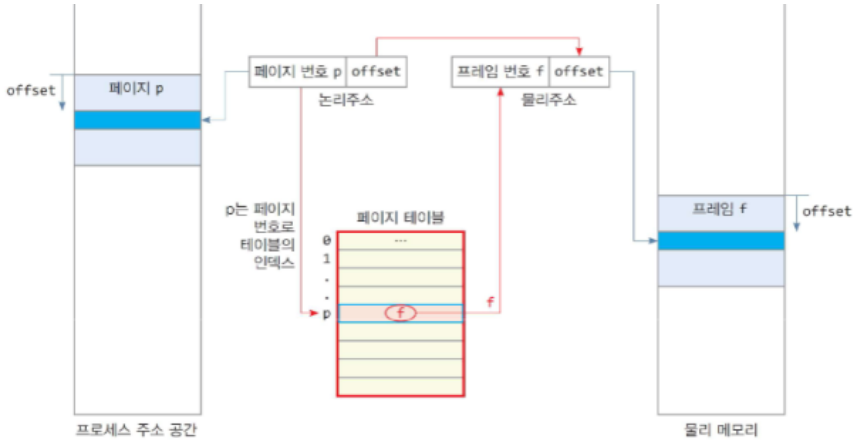
: 프로세스를 논리 세그먼트들로 나누고, 각 논리 세그먼트를 물리 메모리(물리 세그먼트)에 할당하는 메모리 관리 기법



- 논리주소의 구성
  - <세그먼트 번호, offset>
- 세그먼트 테이블 (Segment table)
  - 세그먼트별로 **시작 물리 주소(base)** 와 **세그먼트 크기,길이(limit)** 정보
- STBR (Segment-table base register)
  - 물리 메모리에서의 세그먼트 테이블의 위치를 가르킴
- STLR (Segment-table length register)
  - 프로그램이 사용하는 세그먼트의 수
- 시스템 전체에 1개의 세그먼트 테이블을 두고 논리 주소를 물리 주소로 변환
- 외부 단편화 발생
  - 세그먼트들의 크기가 같지 않기 때문에 세그먼트와 세그먼트 사이에 발생하는 작은 크기의 홀
- 내부 단편화 발생 없음
- 세그먼트는 의미 단위이기 때문에 공유나 보안에 있어 페이징보다 효과적이다

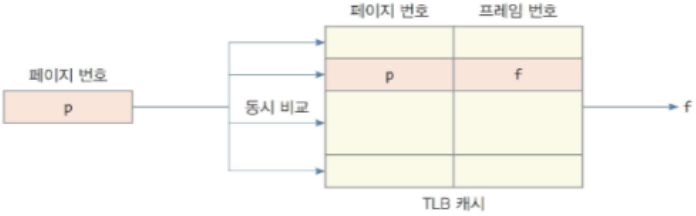
페이징

: 프로세스의 주소 공간과 물리 메모리를 페이지 단위로 분할하고, 프로세스의 각 페이지를 물리 메모리의 프레임에 분산 할당하여 관리하는 기법



- 구성 : <페이지 번호, offset>
- 페이지 테이블은 물리 메모리에 있음
- PTBR (Page-table base register) : 페이지 테이블을 가르킴
- PTLR (Page-table length register) : 테이블 크기
- 외부 단편화 발생 없음
- 내부 단편화 발생
  - 프로세스의 마지막 페이지에만 단편화 발생
  - 단편화의 평균 크기 = 페이지의 ½ 크기
- 모든 메모리 접근 연산에는 **2번의 물리메모리 엑세스** 필요
  - 페이지 테이블 접근 1번, 실제 데이터 접근 1번

- 속도 향상을 위해 TLB라 불리는 고속의 캐시 사용



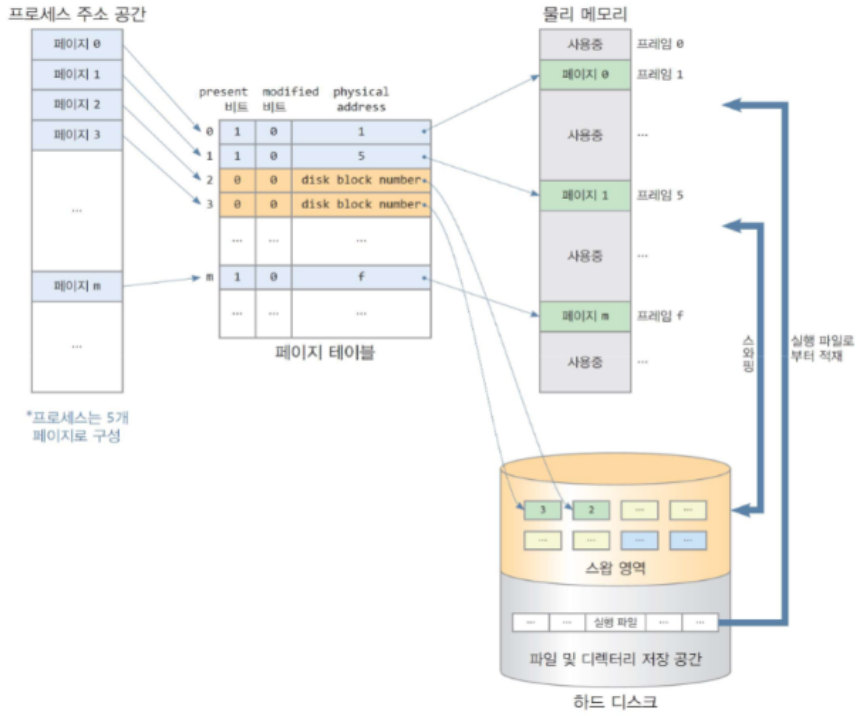
  - 페이지 번호와 TLB 내 모든 항목 **동시에 비교**
  - 페이지가 클수록 TLB 히트 증가 -> 실행 성능 향상
  - 페이기가 클수록 내부 단편화 증가 -> 메모리 낭비
  - TLB reach : TLB가 채워졌을 때, 미스없이 작동하는 메모리 액세스 범위
    - TLB 항목 수 x 페이지 크기
- 페이지 테이블 낭비의 해결책
  1. 역 페이지 테이블(inverted page table, IPT)
     - 각 프레임이 어떤 프로세스의 어떤 페이지에 할당되었는지를 나타내는 테이블
     - 역 페이지 테이블 항목 : <프로세스번호(pid), 페이지 번호(p)>
     - 역 페이지 테이블은 시스템에 1개 존재
  2. 멀티 레벨 페이지 테이블(multi-level page table)
     - 프로세스가 현재 사용 중인 페이지들에 대해서만 페이지 테이블을 만드는 방식

가상 메모리 개념

1. 운영체제는 물리 메모리 영역을 하드 디스크로 연장
2. 프로세스 실행 시 프로세스 전체가 물리 메모리에 적재되어 있을 필요 없음
3. 운영체제는 물리 메모리의 빈 영역이 부족하게 되면, 물리 메모리 일부분을 하드 디스크로 옮겨 물리 메모리의 빈 영역 확보 (다중 프로그래밍 정도를 높임, cpu 활용율 높임)
4. 물리 메모리를 확장하여 사용하는 디스크 영역을 스왑 영역이라고 부름
   - 스왑-아웃 : 물리 메모리의 일부를 스왑 영역으로 옮기는 작업
   - 스왑-인 : 스왑 영역에서 물리 메모리로 가지고 오는 작업

요구페이징 = 페이징 + 스와핑(swapping)

: 현재 실행에 필요한 일부 페이지만 메모리에 적재하고 나머지는 하드 디스크에 두고, 페이지가 필요할 때 메모리에 적재하는 방식
: 운영체제는 **첫 페이지만** 물리 메모리에 적재하고, 실행 중에 프로세스가 다른 페이지를 필요로 할 때 페이지 적재



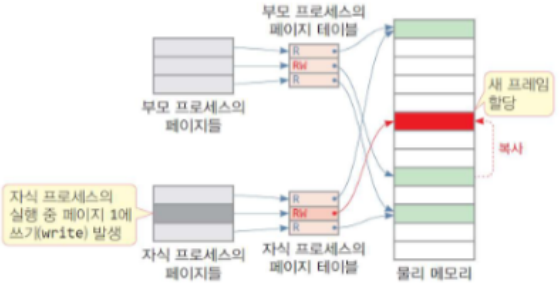
- **present/valid bit : 해당 페이지가 물리 메모리에 있는지 여부**
  - 비트가 1이면 , 해당 페이지가 프레임 번호의 메모리에 있음
  - 비트가 0이면, 해당 페이지가 디스크에 있음

- **modified/dirty bit : 해당 페이지가 수정되었는지 여부**
  - 비트가 1이면, 해당 페이지가 프레임에 적재된 이후 수정되었음, 나중에 쫓겨날 때 스왑-아웃
  - 비트가 0이면, 해당 페이지는 수정된 적이 없음. 나중에 쫓겨날 때, 스왑 영역에 저장 될 필요 없음
- physical address
  - present bit=1이면, 해당 페이지가 적재되어 있는 프레임 번호
  - present bit=0이면, 해당 페이지가 있는 디스크 블록 번호
- 페이지 폴트(page fault)
  - cpu가 액세스하려는 페이지가 물리 메모리에 없을 때, 페이지 폴트 발생
  - 스왑-인(swap-in) = page-in
    - 페이지를 스왑 영역에서 프레임으로 읽어 들이는 행위
  - 스왑-아웃(swap-out) = page-out
    - 프레임에 저장된 페이지를 스왑 영역에 저장하고 프레임을 비우는 행위

쓰기 시 복사(COW, copy on write)

(방법 1) 완전 복사

- 부모 프로세스의 모든 페이지를 완전히 복사
- **비효율적** – fork() 후 exec()하는 것이 일반적인 사례이기 때문
  (방법 2) **쓰기 시 복사**

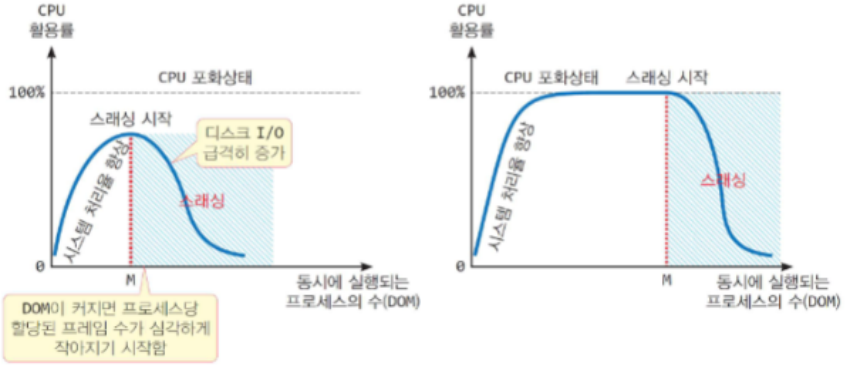


(c) 자식 프로세스가 페이지 1에 쓰기를 실행할 때, 운영체제는 새 프레임을 할당하여 쓰기가 발생한 페이지 1을 복사

- 자식 프로세스를 위해 **부모 프로세스의 페이지 테이블만 복사**
- 그러므로, 초기에 자식 프로세스는 부모 프로세스의 메모리 프레임을 완전 공유
- 자식이나 부모 중 누군가 **페이지를 수정할 때**, **새로운 프레임을 할당 받아 공유하고 있는 부모 프레임을 복사**
- 쓰기 시 복사의 장점
  - 프로세스 생성 시간 절약
  - 메모리 절약

스레싱(thrashing)

: 스레싱은 페이지 폴트가 계속 발생하여, 메모리 프레임에 페이지가 반복적으로 교체되고, 디스크 입출력이 심각하게 증가하고, CPU 활용율이 대폭 감소하는 현상



(a) 메모리가 작은 경우 (b) 메모리가 많은 경우

- 스레싱 원인
  1. 다중 프로그래밍 정도(DOM; degree of multiprogramming)가 과도한 경우
     - 메모리에 비해 너무 많은 프로세스가 실행되어서
     - 프로세스 당 할당되는 프레임 개수가 적을 때
     - 프로세스가 필요한 충분한 페이지가 적재되지 못하여 페이지 폴트 발생

2. 잘못된 메모리 할당/페이지 교체 알고리즘
3. 기본적으로 메모리 양이 적을 때
4. 우연히 특정 시간에 너무 많은 프로세스 실행

- 해결 및 예방
  - 다중 프로그래밍 정도(DOM) 줄이기
  - 메모리 늘리거나 SSD 사용

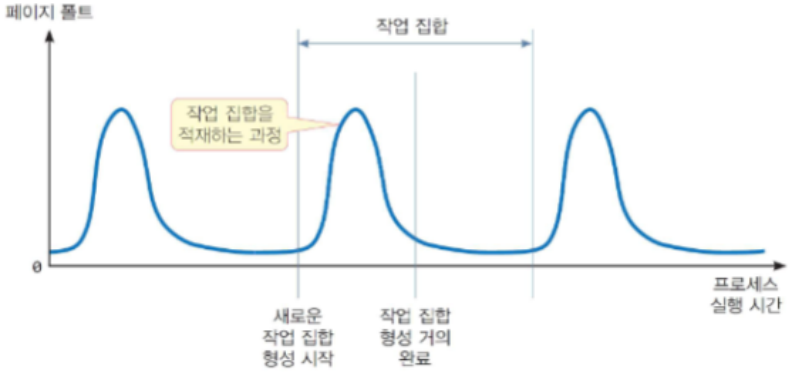
참조의 지역성(reference of locality)

: CPU가 짧은 시간 범위 내에 일정 구간의 메모리 영역을 반복적으로 참조하는 경향
-> 메모리 할당과 페이지 교체 전략에 활용

- **90/10 규칙** - 프로그램 코드의 10%에서 실행 시간의 90% 소비

작업 집합(working set)

: 일정 시간 범위 내에 프로세스가 액세스(참조)한 **페이지들의 집합**

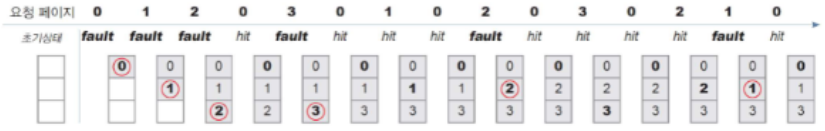


※ 새로운 작업 집합이 형성되는 과정에서 페이지 폴트가 급격히 발생하지만 곧 줄어들어 안정 상태가 된다.

- 프로세스가 실행되는 동안 계속 작업 집합 이동
  - 시간이 지나면 새로운 작업 집합 형성
  - **스래싱은 작업 집합이 메모리에 올라와 있지 않을 때 발생**

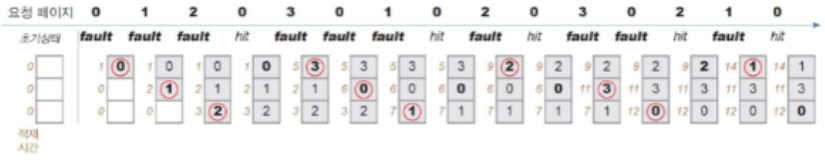
페이지 교체 알고리즘의 종류

- 최적 교체(Optimal Page Replacement)



페이지 폴트 횟수 : 6

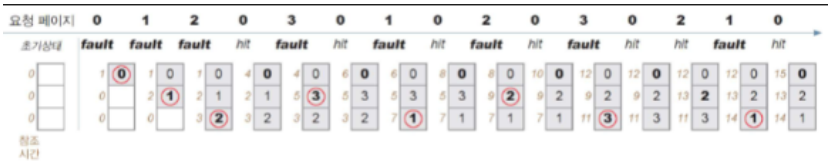
  - 가장 먼 미래에 사용될 페이지를 교체 대상으로 결정
- FIFO(First in first out)



페이지 폴트 횟수 : 10

  - 가장 오래전에 적재된 페이지 선택
  - 구현 단순
  - **벨라디의 이상현상**
    - Frame 3 : 페이지 폴트 9회, Frame 4 : 페이지 폴트 10회

- LRU(Least recently used)

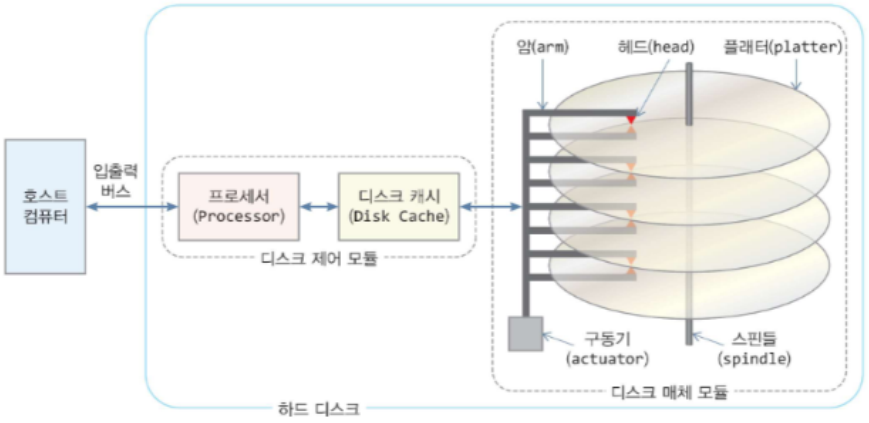


페이지 폴트 횟수 : 8

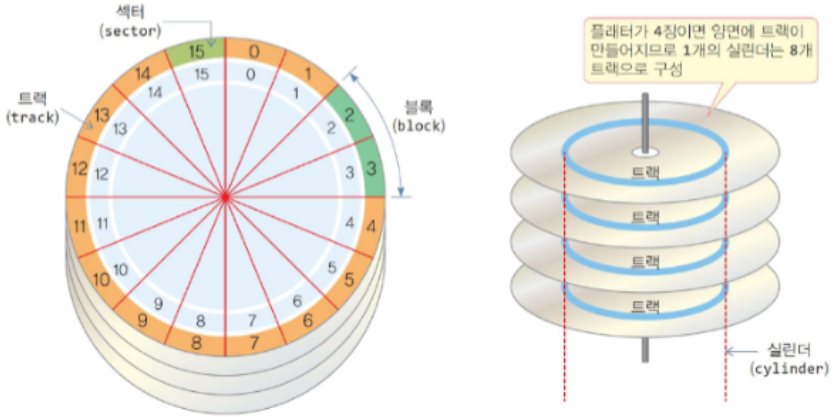
  - 가장 최근에 사용되지 않았던(가장 오래전에 사용된) 페이지 선택
  - 참조 비트(Ref)를 사용
- Clock

  - FIFO와 LRU를 섞은 알고리즘
  - 프레임을 원형 큐로 연결하고 원형 큐에서 검색을 시작하는 프레임 위치를 포인터
  - 프레임당 1비트의 참조 비트(reference bit/used bit) 사용
    - 참조 비트가 0이면, 그 프레임을 희생 프레임으로 선택
    - 참조 비트가 1이면, 0으로 바꾸고 다음 프레임으로 이동

저장 장치



- 플래터(platter)
  - 정보가 저장되는 매체, 원형 판(아래 윗면 모두 저장)
- 헤드(head)
  - 플래터 한 면당 하나의 헤드(플래터 한 장에 2개의 헤더)
  - 플래터에서 정보를 읽고 저장하는 장치



- 섹터 : 플래터에 정보가 저장되는 최소 단위, 512바이트 혹은 4096바이트

- 트랙 : 플래터에 정보가 저장되는 하나의 동심원, 여러 개의 섹터들 포함
- 실린더 : 같은 반지름을 가진 모든 트랙 집합
  - 예) 헤드가 8개인 디스크에서 8개의 트랙을 묶어 실린더라고 함
- 블록 : 운영체제가 파일 데이터를 입출력하는 논리적인 단위. 몇 개의 섹터로 구성

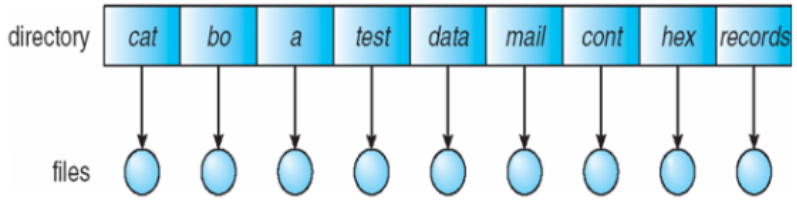
파일 입출력 주소

- 디스크 장치는 **디스크 물리 주소** 사용
  - 물리 주소(physical disk address)
    - 디스크의 섹터 위치를 나타내는 주소
    - CHS(Cylinder-head-sector) 물리 주소 = (cylinder 번호 , head 번호, sector 번호)
  - 물리 주소의 단위는 섹터
- 운영체제는 **논리 블록 주소** 사용
  - 논리 주소(Logical Block Address, LBA)
    - 저장 매체를 1차원의 연속된 데이터 블록들로 봄

디스크 구조

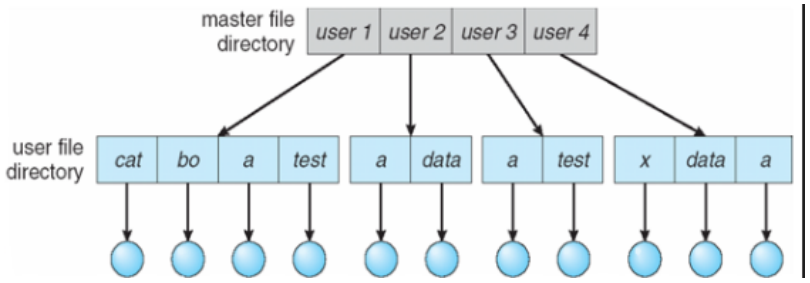
1. **파티션 (Partition)**:
   - 디스크를 여러 개의 논리적 구획으로 나눈 것입니다.
   - 파티션은 **미니디스크(minidisk)** 또는 **슬라이스**(slice)라고도 불립니다.
2. **파일 시스템(File System)**:
   - 데이터를 조직화하고 저장하기 위해 디스크를 포맷하여 사용합니다.
   - 파일 시스템이 없는 상태(raw)로도 디스크나 파티션을 사용할 수 있지만, 이 경우 데이터 관리가 복잡해질 수 있습니다.
3. **볼륨 (Volume)**:
   - 파일 시스템을 포함하는 엔터티를 **볼륨**이라고 합니다.
   - 각 볼륨은 파일 시스템을 포함하고 있으며, 디바이스 디렉토리 또는 **볼륨 테이블**(volume table of contents)에 해당 파일 시스템 정보를 기록합니다.

Single-Level Directory



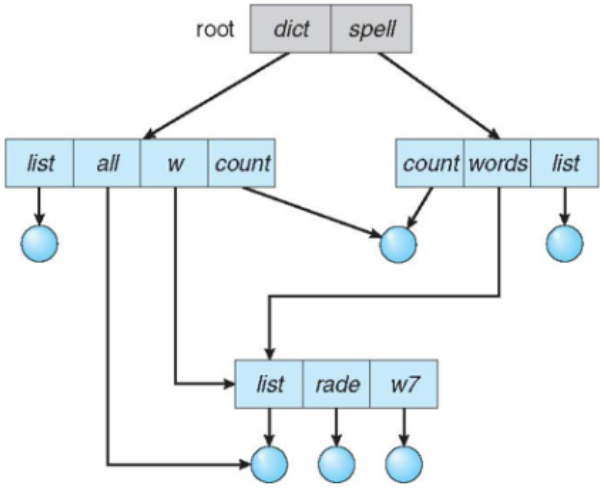
- Naming problem
- Grouping problem

Two-Level Directory



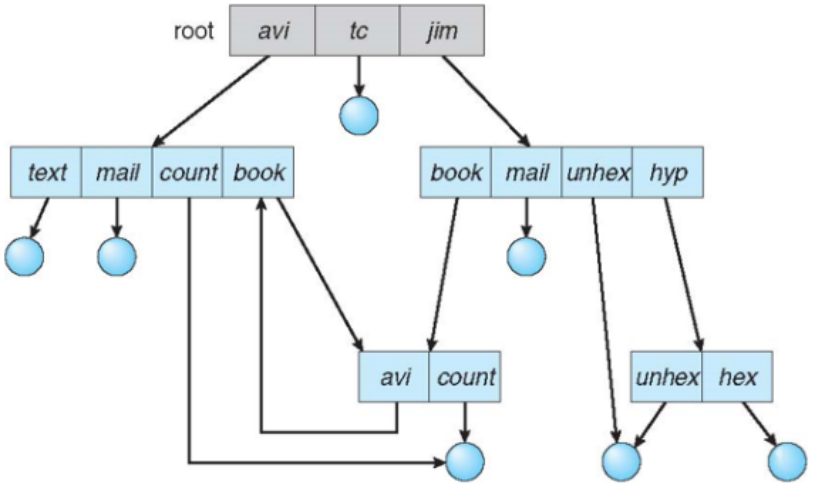
- **경로 이름(Path name)**
  - 파일을 찾기 위해 사용되는 전체 경로입니다.
- **다른 사용자가 동일한 파일 이름을 가질 수 있음**
  - 각 사용자는 자신의 파일을 가질 수 있으며, 파일 이름이 동일하더라도 서로 다른 사용자 영역에서 관리됩니다.
- **효율적인 검색**
  - 경로 이름을 사용하면 특정 파일을 빠르게 검색할 수 있습니다.
- **그룹화 기능 없음**
  - 파일이나 디렉토리를 논리적으로 묶는 기능이 제공되지 않습니다.

Acyclic-Graph Directories



- **두 개의 다른 이름(별칭, Aliasing)**
  - 동일한 파일이나 데이터에 대해 두 개의 서로 다른 이름을 사용하는 경우를 의미합니다.
- **딕셔너리에서 카운트(count)가 삭제되면 -> ==댕글링 포인터(dangling pointer) 발생==**
  - 파일이나 데이터에 대한 포인터가 존재하지만, 실제 데이터는 삭제되어 포인터가 유효하지 않은 상태가 됩니다.

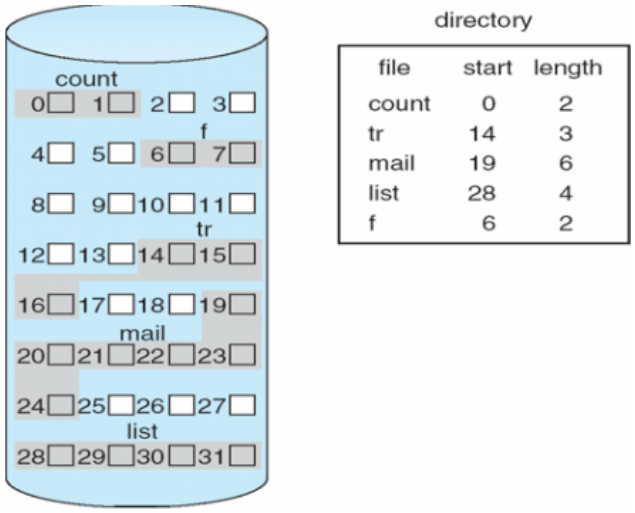
General Graph Directory



- **가비지(Garbage)**
  - 참조되지 않는 데이터나 리소스를 의미합니다.
- **참조 횟수(reference count)가 0이 될 때만 실제 삭제**
  - 데이터나 파일을 참조하는 모든 링크가 제거되었을 때 해당 데이터를 삭제합니다.
- **예시**
  - 루트(root)가 `avi`를 삭제하고, 짐(jim)이 `book`을 삭제하면 더 이상 참조되지 않는 데이터가 남아 가비지가 됩니다.
- **가비지 수집(Garbage Collection)**
  - 가비지를 탐지하고 제거하여 시스템 리소스를 효율적으로 관리합니다.

할당 방법 - 연속 할당

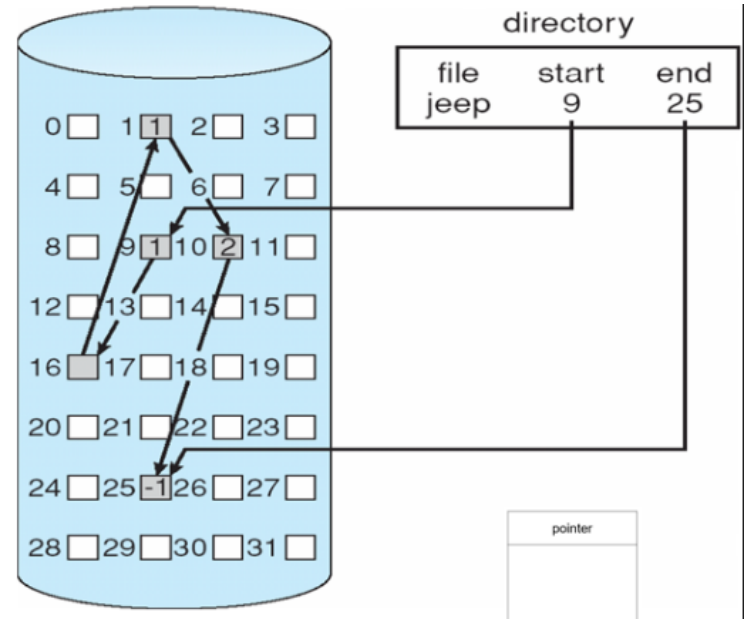
순차적 접근과 랜덤 접근 모두에 매우 적합함.
할당 방법은 파일에 대해 디스크 블록이 어떻게 할당되는지를 나타냅니다:



- **연속 할당(Contiguous allocation)**: 각 파일이 연속적인 블록 집합을 차지함
  - 대부분의 경우 **최상의 성능** 제공
  - 간단함 – 시작 위치(블록 번호)와 길이(블록 수)만 필요
  - 문제점:
    - 파일을 위한 공간 찾기
    - 파일 크기 예측
    - 외부 단편화 발생
    - 오프라인(중단 시간) 또는 온라인에서 압축(compaction)이 필요

할당 방법 - 연결 할당

순차적 접근에 적합하지만, 랜덤 접근에는 부적합함.

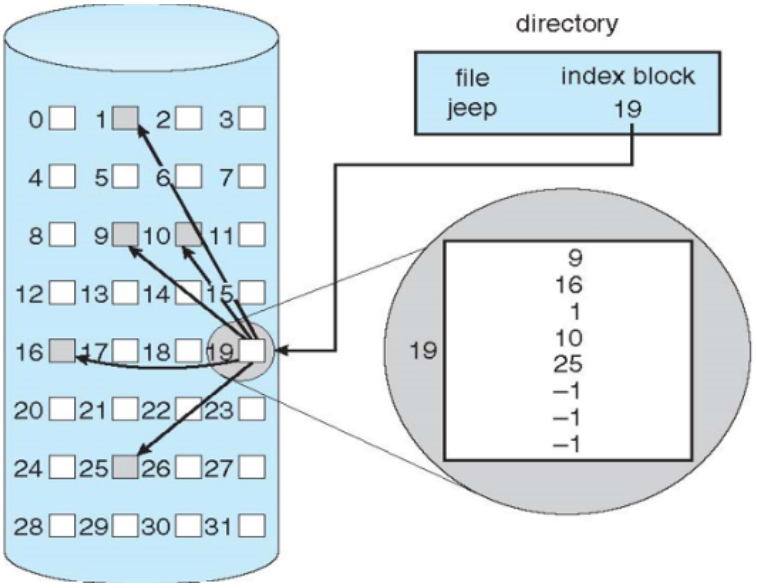


- **연결 할당(Linked allocation)**: 각 파일이 블록들의 연결 리스트로 구성됨
  - 외부 단편화 없음
  - 각 블록은 다음 블록을 가리키는 포인터를 포함
  - 압축(compaction) 불필요, 외부 단편화 없음
  - 블록 위치를 찾으려면 많은 I/O와 디스크 탐색 필요
- **FAT(File Allocation Table) -> 포인터들을 모아 놓은 파일블록**
  - 볼륨 시작 부분에 블록 번호로 인덱싱된 테이블 존재
  - 디스크 상에서 연결 리스트와 유사하지만 더 빠르고 캐시 가능
  - 새로운 블록 할당이 간단함

인덱스 할당(Index Allocation)

단일 블록 접근 시 2번의 인덱스 블록 읽기와 1번의 데이터 블록 읽기가 필요할 수 있음.
:인덱스 할당은 파일 시스템에서 파일 데이터를 저장하기 위해 디스크 블록을 관리하는 방법 중 하나입니다. 이 방식은 파일의 모든 데

이터 블록을 하나의 인덱스 블록에 저장된 포인터를 통해 추적합니다.



1. **구조**
   - 각 파일은 하나의 **인덱스 블록**을 가짐.
   - 인덱스 블록은 해당 파일에 속한 데이터 블록의 포인터를 저장.
   - 파일 데이터를 저장하는 블록들은 반드시 연속적일 필요가 없음(비연속적 할당 가능).
2. **장점**
   - **외부 단편화 없음**: 파일이 연속적으로 저장될 필요가 없으므로 외부 단편화 문제가 발생하지 않음.
   - **임의 접근(Random Access)** 지원: 특정 블록을 바로 찾을 수 있어 접근 속도가 빠름.
   - **파일 크기 동적 확장 가능**: 필요에 따라 블록을 추가적으로 할당 가능.
3. **단점**
   - **인덱스 블록 오버헤드**: 각 파일마다 인덱스 블록이 필요하므로 작은 파일이 많을 경우 공간 낭비 발생.
   - **큰 파일 관리 제한**: 인덱스 블록의 크기가 제한되면 포인터 수가 제한되어 큰 파일을 관리하기 어려움.

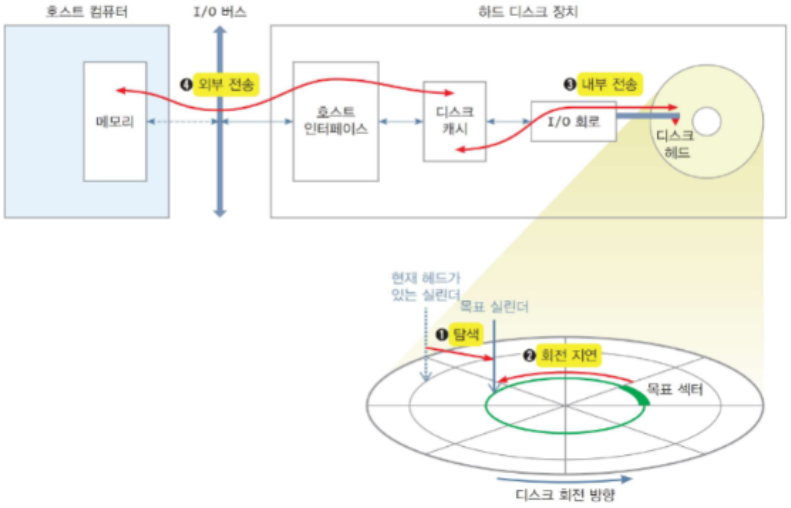
저장장치의 문제

- 입출력 병목을 줄이기 위한 방법
  - 주기억 장치 메모리 늘리기
  - 디스크 캐시 늘리기
  - 디스크 스케줄링
  - 병렬 저장 장치 사용 (RAID)
- 데이터 신뢰성
  - 저장 장치의 고장은 데이터 손실 초래 -> 심각한 문제
- 디스크 미러링(disk mirroring)
  - 2개의 동일한 디스크 사용
  - RAID 레벨 1로도 불림
- RAID(Redudant Array of inexpensive Disks)
  - 여러개의 디스크를 병렬로 연결
  - RAID 레벨 5
    - 디스크를 4개를 쓰지만 3개는 한번에 데이터를 쓰고 나머지 1개에 패리티 블록을 넣어 고장이 난 경우 블록을 복구 할 수 있음

디스크 용량

: **실린더 개수 x 실린더당 트랙 수 x 트랙당 섹터 수 x 섹터 크기**

디스크 입출력 명령을 처리하는 과정

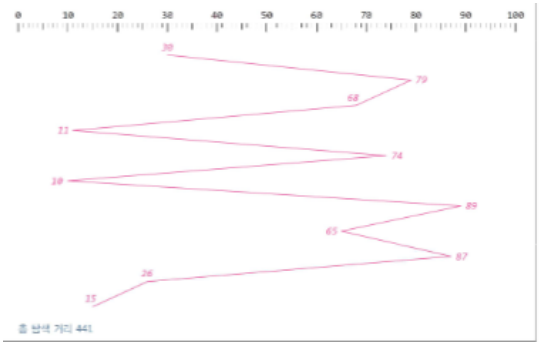


- 프로세서가 논리 블록 번호를 CHS 물리 주소로 바꾸고 장치 제어 및 입출력 시행
- 탐색(seek)
  - 디스크 헤드를 목표 실린더로 이동
  - 탐색 거리 : 이동하는 실린더 개수
- 회전 지연(rotational latency)
  - 플래터가 회전하여, 헤드 밑에 목표 섹터가 도달할 때까지 대기
  - **평균 회전 지연 시간 = ½ 회전 시간**
- 전송(transfer)
  - 디스크 헤드와 호스트 사이의 데이터 전송
  - 내부 전송과 외부 전송으로 나뉨
- 오버헤드(overhead)
  - 디스크 프로세서가 호스트에서 명령을 받고 해석하는 등의 부가 과정
- **디스크 액세스 시간**
  - 탐색 시간 + 회전 지연 시간 + 내부 전송 시간
- **입출력 응답 시간**
  - 탐색 시간 + 회전 지연 시간 + 전체 전송 시간 + 오버헤드

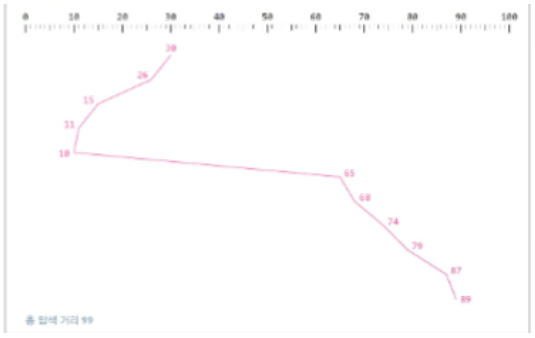
디스크 스케줄링 알고리즘

알고리즘	선택 방법	장점	단점
FCFS	도착순	구현이 쉽고 기아 없고 공평	처리율 낮음
SSTF	가장 가까운 요청 우선	처리율 최고 높음	요청이 도착하여 처리될 때까지의 응답 시간 편차가 크고, 기아 발생 가능
SCAN	한쪽 끝 실린더에서 다른 쪽 끝으로 와서 다시 반대 방향으로 이동하기를 반복. 왕복 이동	SSTF에 비해 균등한 서비스, 기아 없음	중간 실린더의 요청이 양쪽 끝 실린더 요청보다 높은 서비스 확률. SSTF보다 처리율 낮음
LOOK	SCAN과 같지만 움직이는 방향으로 더 이상 요청이 없으면 방향 전환. 왕복 이동	SSTF보다 더 균등한 서비스, 기아 없음. SCAN보다 처리율 높음	SSTF보다 처리율 낮음
C-SCAN	바깥쪽 실린더에서 시작하여 안쪽 끝까지 도착하면, 다시 바로 바깥쪽 실린더로 이동하여 반복. 한쪽 방향으로만 이동	SSTF보다 더 균등한 서비스, 기아 없음.	SST보다 처리율 낮음
C-LOOK	C-SCAN과 같지만 안쪽 방향으로 더 이상 요청이 없으면 대기 중인 가장 바깥쪽 요청에서 시작. 한쪽 방향으로만 이동	LOOK과 C-SCAN을 결합한 버전. 기아 없음	SSTF보다 처리율 낮음

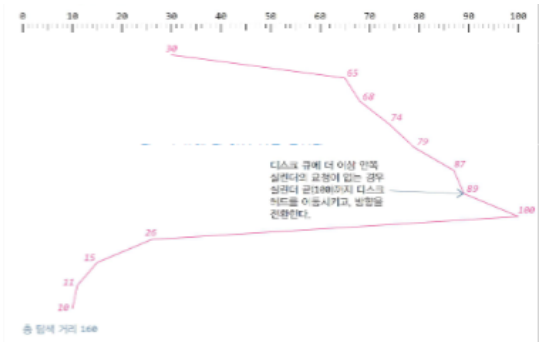
- FCFS : 디스크 큐에 도착한 순서대로 요청들을 처리



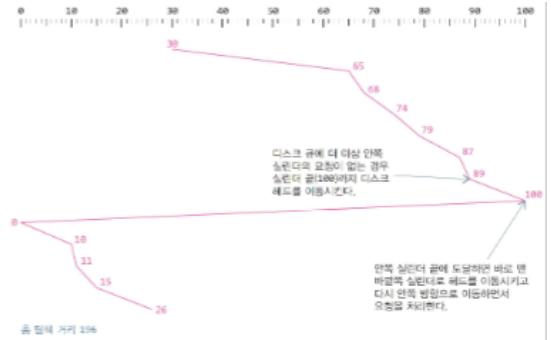
  - 디스크 큐를 검색할 필요 없어 구현이 쉽고 기아가 없음, 공평
  - 성능이 좋지 않음
- SSTF : 현재 디스크 헤드가 있는 실린더에서 가장 가까운 요청 선택



  - 탐색 거리가 가장 짧은 것을 선택하기 때문에 성능은 매우 우수
  - 디스크 헤드에서 멀리 있는 요청들에 기아 발생 우려
  - 바깥쪽 실린더에 대한 요청들은 오래 기다릴 수 있음 (응답 편차 큼)
- SCAN

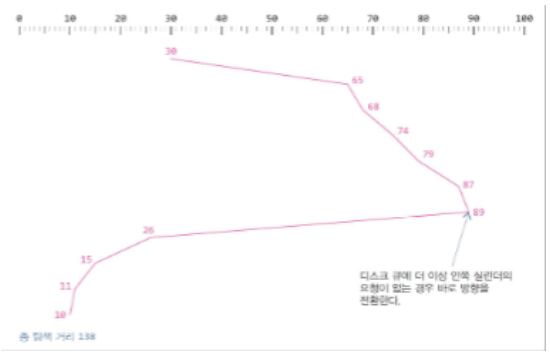


  - 기아 발생을 없애고 균등한 처리를 위해 고안된 알고리즘
  - 맨 바깥쪽 실린더의 요청에서 시작하여 안쪽 실린더로 요청을 처리 후 안쪽 끝까지 이동
    - 다시 바깥쪽 방향으로 요청을 처리하면서 끝 실린더까지 이동
  - SSTF에 비해 균등한 입출력 서비스
  - 하지만, 양 끝쪽 실린더의 요청들은 중간에 위치한 요청들보다 선택될 확률 낮음
- C-SCAN

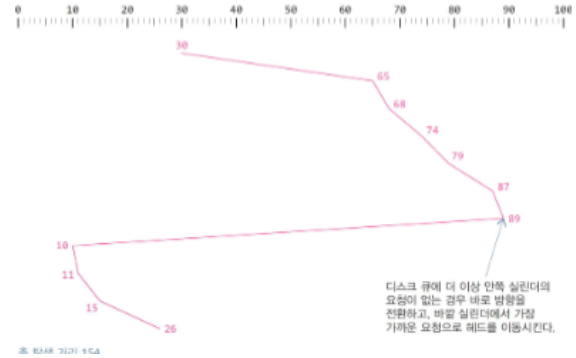


  - 중간 실린더의 요청들이 양끝보다 더 높은 확률로 서비스되는 SCAN의 단점 보완
  - **한 방향으로만** 이동하면서 요청 처리

- LOOK



  - SCAN 처럼 작동, 맨 끝 실린더로 이동하는 SCAN의 단점 보완
  - 현재 이동 방향에 더 이상의 요청이 없는 경우 즉시 이동 방향 변경
- C-LOOK



  - LOOK과 C-SCAN의 결합
  - **한 방향으로만** 서비스
  - 이동 방향에 요청이 없으면 즉각 이동 방향 수정

SSD 저장 장치

SSD의 특징

- 플래시 메모리(flash memory)를 저장소로 하는 비휘발성 기억 장치
- 메모리 계층 구조의 최하위 단에 위치하는 보조 기억 장치
- 읽기 속도보다 쓰기 속도가 더 오래 걸림

SSD의 용도

- 읽기가 많은 운영체제 코드나 프로그램의 설치에 적합
- 백업 저장 장치에 적합하지 않음
- 쓰기나 수정 작업이 많은 스왑 영역이나 많은 파일이 임시로 생성되었다가 지워지는 임시 파일 시스템에 적합하지 않음

웨어레벨링(wear leveling, 균등 쓰기 분배)

: 플래시 메모리의 모든 블록에 걸쳐 쓰기를 균등하게 분배, 특정 블록에 **과도한 쓰기를 막아**, 플래시 메모리의 **고장이나 데이터 손실 예방 기법**

- SSD 제어기에 의해 실행
  - 블록마다 지우기 횟수 기억, 쓰기 시 지우기 횟수가 가장 적은 블록 선택