07. Syncronization

IPC 종류	사용 시기	매개체	통신 단위	통신 방향	통신 범위
무기명 PIPE	부모 자식 간 단방향 통신	파일	stream	단방향	동일시스템
Named PIPE	다른 프로세스와 단방향 통신	파일	stream	단방향	동일시스템
Message Queue	다른 프로세스와 단방향 통신	메모리	구조체	단방향	동일시스템
Shared Memory	다른 프로세스와 양방향 통신	메모리	구조체	양방향	동일시스템
Memory Map	다른 프로세스와 양방향 통신	파일+메모리	페이지	양방향	동일시스템
Socket	다른 시스템과 양방향 통신	소켓	stream (packet)	양방향	동일+외부시스템

07.1. Race Condition, Syncronization, Thread-safe

07.2. 상호배제

만족 조건
- 상호 배제
- 한정 대기
- 진행의 융통성

HW 방법
1. 인터럽트 서비스 금지

```
cli  ; clear interrupt flag
...
sti  ; set interrupt flag
```

2. 원자 명령 사용: 2개 명령을 한번에 처리

```
mov ax, lock
mov lock, 1
```

→ Test and Set Lock

```
tsl ax, lock
```

07.3. Syncronization

- Mutex/Spinlock
  - 상호배제되도록 lock 사용
  - mutex: 동기화 대상이 1개일 때
  - 락 소유하지 않은 스레드, 락 해제까지 대기
  - busy-wait, sleep-wait(큐 존재)

  - 락 잠기는 시간 김: 뮤텍스
  - 단일 CPU 시스템: 뮤텍스
  - 커널: 스핀락, 사용자: 뮤텍스
  - 스핀락: 기아 발생 가능

- Semaphore
  - wait-signal

  - PV 연산
    - P: wait, 자원 요청 시 실행
    - V: signal: 자원 반환 시 실행
- mutex vs semaphore
  - mutex: 동기화 대상 1개, semaphore: N개
  - mutex: 자원 소유 가능 + 책임 가짐, semaphore: 자원 소유 불가
  - mutex: 소유 스레드만이 mutex 해제 가능

08. Deadlock

08.1. 교착상태

- 자원을 소유한 채 모두 상대방이 소유한 자원을 기다리면서 무한 대기 (러시아워)
  자원을 소유한 스레드 사이에서 각 스레드는 다른 스레드가 소유한 자원 요청, 무한 대기

- 원인: Circular wait

- 기아와 교착
  - 기아: 운영체제의 잘못된 정책 → 작업 지연
  - 교착: 여러 프로세스의 동시 작업 → 자연스럽게 발생
- 교착의 잠재 원인:
  1. 자원
  2. 자원과 스레드
  3. 자원과 운영체제
  4. 자원 비선점
- 자원 할당 그래프(Resource Allocation Graph)
  - P1 ← R1 : 자원 할당받음
  - P1 → R1 : 요청, 대기

08.2. 교착상태의 해결

교착상태 발생 필요충분조건

자원적 특성	행위적 특성
• 상호 배제	• 점유와 대기
• 비선점	• 환형 대기

- 넷 중 하나라도 성립되지 않으면 교착 발생 ×: 넷 중 하나를 깨자

해결 방법의 유형
1. 교착상태 예방
   - 위 네 개 조건 중 하나 이상의 조건 비성립되도록 시스템 구성
2. 교착상태 회피
   - 교착 가능성 판단 → 발생 않을 것이라 확신하면 자원 할당
   - 안전한 상태 / 불안전한 상태: 안전한 경우에만 자원 할당
   - 자원 할당 시마다 교착 가능성 검사: 시스템 성능 저하
3. 교착상태 감지 및 복구
   - 백그라운드에서 교착 감지 프로세스 상주해야함
4. 교착상태 무시
   - 무대책, 교착상태는 없다고 단정
   - 사용자가 이상을 느끼면 재실행할 것이다

08.2.1. 교착상태 예방

1. 상호 배제 없애기: 불가능
2. 선점 허용: 기아 발생 가능성, 구현 어려움, 오버헤드 큼
3. 점유와 대기: 대기 않도록 지정
   1. 운영체제, 스레드 실행 전 필요 자원 모두 파악, 한번에 할당
      다른 스레드, 필요 자원 할당 X, 대기
   2. 스레드 새로운 자원 요청시 할당받은 모든 자원 반환, 한꺼번에 요청
   - 프로세스: 자신이 사용하는 모든 자원 알기 어려움
   - 당장의 미사용 자원도 스레드에 할당: 활용률 떨어짐
   - 많은 자원 사용 프로세스: 적은 자원 프로세스보다 불리
4. 환형 대기 제거
   - 모든 자원에 숫자 부여: 숫자가 큰 방향으로만 자원 할당
   - 프로세스 작업 진행 유연성 떨어짐, 번호 부여는 어떻게?

08.2.2. 교착상태 회피

- 자원 할당 시, 미래에 환형 대기 발생 예상 시 할당 ×
- 교착 상태가 발생하지 않는 범위 내에서 자원 할당

banker's algorithm
- 은행에서의 대출 알고리즘
1. 각 프로세스가 시작 전, 필요한 전체 자원 수 OS에게 알림
2. 자원 할당 시, 할당하면 교착상태 발생하는가? 체크: 안전할때만 할당
3. 각 프로세스 필요한 자원의 개수, 할당 받은 자원 개수, 할당 가능한 자원의 개수 토대로 요청 자원 할당해도 안전한가? 판단
- 비현실적

08.2.3. 교착상태 감지 및 복구

- 교착상태 감지 프로세스 사용

교착상태 감지 전략
- 자원 할당 그래프 이용
- 타임아웃 이용

복구 방법
- 자원 강제 선점: 교착 상태에 빠진 스레드 중 하나의 자원 빼앗음
- 롤백: 주기적으로 관심병사 상태 저장, 교착 발생 시 롤백, 자원 다르게 할당
- **프로세스/스레드 강제 종료**
  - 교착 상태를 일으킨 모든 프로세스 동시 종료
  - 교착 상태를 일으킨 프로세스 중 하나를 골라 순서대로 종료

09. Memory

09.1. 메모리 관리

- 운영체제에 의해 메모리 관리 필요 이유
  - 메모리는 공유 자원
  - 메모리 보호
  - 메모리 용량 한계 극복
  - 메모리 효율성 증대

- 메모리 계층화
  - 계층화가 가능한 이유: 참조의 지역성, 짧은 시간 내에 코드, 데이터, 자원 재사용됨

- 메모리 관리의 이중성
  - 프로세스: 가능한 많은 메모리 사용
  - OS: 가능한 적은 메모리 제공

- 메모리 관리자
  - Fetch Policy: 적재 정책
    - 프로세스가 요구하는 데이터 언제 메모리로 로드할 것인가?
  - Placement Policy: 배치 정책
    - 가져온 프로세스, 메모리 어느 위치에 로드할 것인가?
  - Replacement Policy: 재배치 정책
    - 메모리 모두 차면 어떤 프로세스를 내보낼 것인가?

09.2. 메모리 주소

- 메모리 주소와 CPU 비트
  - 메모리 각 영역 1Byte로 나뉨, 0번지부터 시작
  - MAR: Memory Address Register
    CPU가 메모리 사용하기 위해 사용

물리 주소와 논리 주소
- 사용자, 프로세스: 절대 물리 주소를 알 수 없음
- 메모리 접근 시 상대 주소를 사용하면 절대 주소로 변환해야함: MMU가 수행

Address binding
- Compile time binding
- Load time binding
- Run time binding
  - MMU(Memory Management Unit)가 수행

상대주소 → 물리주소
- 상대주소 + `base` 레지스터 = 절대주소
- `limit` 레지스터: 작업 요청 시 경계를 벗어나는 작업 요청 프로세스 - 종료

09.3. 메모리 오버레이

- 프로그램의 크기가 실제 메모리보다 클 때 적당한 크기로 가져오는 기법 - "Swap"

09.4. 메모리 보호

- CPU, 사용자 모드에서 생성된 모든 메모리 액세스 확인, 기본값 제한값 사이에 있는가? 확인
  - 하드웨어(MMU)의 지원을 받음

ASLR: Address Space Layout Randomization

09.5. 메모리 할당

- Process partitioning: 연속 vs 불연속
  - 연속: 프로세스, 메모리 내 인접되어 연속되게 하나의 블록 차지하도록 배치
    - 장점: 논리-물리 변환 단순, 액세스 속도 빠름
    - 단점: 할당 유연성 떨어짐, 외부 단편화
    - 메모리 배치 기법: first-fit, best-fit, worst-fit
  - 불연속: 프로세스가 페이지나 세그먼트 단위로 분산 배치
- Memory partitioning: 가변 vs 고정
  - 가변: 프로세스의 크기에 맞게 메모리 분할
  - 고정: 프로세스의 크기에 무관하게 메모리 같은 크기로 나뉨

09.6. 단편화

- **외부 단편화** 할당된 메모리 사이에 사용할 수 없는 홀 발생
- **내부 단편화** 할당된 메모리 내부에 사용할 수 없는 홀 발생

- 조각 모음: 프로세스 정지 → 재배치 → 재시작

- Buddy System: 메모리 할당 판 Binary search
  - 메모리, 프로세스 크기대로 나뉨
  - 내부 단편화 발생
  - 조각모음 쉬움

세그멘테이션
- 세그먼트: 개발자 관점에서 보는 프로그램의 논리적 구성 단위

- 세그멘테이션: 프로세스, 논리 세그멘트 크기로 분할. 각 세그멘트를 한 덩어리의 물리 메모리에 할당

- 프로세스 주소 공간
  - 프로세스 주소 공간: 여러 개의 논리 세그멘트로 나누고 각 논리 세그멘트를 물리 세그멘트에 매핑
  - 시스템 전체 세그멘트 매핑 테이블 두고 논리 → 물리 변환

10. Paging

10.1. 페이지와 페이지 프레임

- 페이지: 프로세스의 주소 공간
- 프레임: 물리메모리를 페이지 크기로 나눔
- 페이지 테이블: 각 페이지에 대해 페이지 번호와 프레임 번호 1:1 매치

- 왜 페이징인가?
  - 용이한 구현
  - 높은 이식성
  - 높은 융통성
  - 메모리 활용과 시간 오버헤드
    - 외부 단편화 없음
    - 내부 단편화 매우 작음
      - 페이지 한 칸
      - 최대 내부 단편화: `단일 페이지 크기 - 1Byte`
    - 홀 선택 알고리즘 필요 없음

10.2. 페이징의 구현

하드웨어 지원
- CPU 지원
  - Page Table Base Register
- MMU 장치
  - 논리 주소의 물리 주소 변환
  - 페이지 테이블을 저장하고 검색하는 빠른 캐시
  - 메모리 보호(아래 사항 확인)
    - 페이지 번호가 페이지 테이블에 있는가?
    - 오프셋이 페이지의 범위를 넘어가는가?

운영체제 지원
- 프레임 동적 할당/반환
- **페이지 관리 기능**
  - 프로세스 생성/소멸 ⇒ 동적으로 프레임 할당/반환
  - 할당된 페이지 테이블, 빈 프레임 리스트 생성
  - 컨텍스트 스위칭 ⇒ 레지스터에 적절 값 로딩

32비트, 페이지 4KB인 경우
- 물리 메모리와 무관
- 주소 공간: 4GB
- 프로세스 당 최대 페이지: `4GB / 4KB` = $2^{32}/2^{12} = 2^{20}$
- 페이지 테이블 크기: `4Byte` × 2^{20} = 2^{22} = `4MB`

10.3. 페이지 주소 관리

`VA = <P, O>`
- VA: Virtual Address
- P: Page Number
- O: Offset

`PA = <F, O>`
- PA: Physical Address
- F: Frame Number
- O: Offset

페이지 논리 주소 구성
- `[#Page, Offset]` ⇒ `VA`
  - 페이지 크기가 `4KB`(4^{12})라면, 페이지 내부 표현에 12비트 필요
  - 오프셋: 12비트
  - 32비트 시스템에서: 상위 20비트 페이지 번호, 하위 12비트 오프셋

페이지 테이블 기준 레지스터 (Page Table Base Register)
- 컨텍스트 스위칭 시 PCB에 저장

페이지 테이블 매핑
- 페이지 테이블도 메모리에 있음: 스왑 대상
- 테이블 관리 방식에 따라 `VA`→ `PA` 방법 다름

직접 매핑
- 페이지 테이블 전체, 물리 메모리의 운영체제 영역에 존재
  - 추가 작업 없이 주소 변환 가능: 직접 매핑
- 모든 페이지, 물리 메모리: 속도 빠름, 용량 낭비

연관 매핑
- 일부 페이지 정보만 물리메모리 상 테이블에서 관리
  - 실제 메모리에 올라와있는 페이지만 관리
  - 나머지: 스왑 영역 (스왑 영역에 따로 전체 테이블 보관)
  - 연관 매핑 테이블: 저장된 페이지, 프레임 번호 저장
- 저장된 페이지 번호: 순서 × → 페이지 참조 시 테이블 전체 탐색 필요
  - 병렬 처리 로직 + TLB/연관레지스터로 캐시
  - 캐시 미스 ⇒ 검색 다시 필요: 느림

집합-연관 매핑
- 연관 매핑 문제점: 캐시 미스: 스왑영역의 모든 테이블 검색
1. 페이지 테이블을 여러 묶음으로 나눔
2. 묶음의 시작 주소를 Set table로
- 페이지 테이블 참조: 디렉토리 테이블 ⇒ 묶음 페이지 테이블 ⇒ 프레임 번호 매핑
- `VA = <P1, P2, O>`

10.4. 페이지 테이블 관리

- MM: 특정 프로세스가 실행될 때마다 해당 페이지 테이블 참조하여 VA ⇒ PA 작업 반복
  - 페이지 테이블: 자주 사용하므로 빠른 접근 필요
  - 물리 메모리 영역 중 커널 일부에 보관

무엇이 문제인가?
- 1회 메모리 액세스 위한 2회 물리 메모리 액세스
- 페이지 테이블 낭비
- 페이지 테이블도 스왑 대상

10.4.1. 1회 메모리 액세스 위한 2회 물리 액세스

TLB: Translation Look-aside Buffer

Address translation cache
- 논리 → 물리 변환 캐시: 페이지 접근에 캐시를 두자
- `[페이지 번호 p, 프레임 번호 f]`를 내용으로 저장: 연관매핑
  - 페이지 번호를 받아 전체 캐시 고속 검색
- **Content-Addressable Memory** (associative memory)

TLB 메모리 엑세스
1. CPU로부터 논리 주소 발생
2. 논리 주소의 페이지 번호가 TLB로 전달
3. 페이지 번호와 TLB 내 모든 항목 동시 비교
   - TLB hit: return
   - TLB miss: 페이지 테이블로부터 프레임 번호 로드, 로드 결과 TLB에 삽입

TLB의 성능
- 참조 지역성
  - 순차 액세스 시 속도 빠름, TLB hit 계속
  - TLB: 랜덤 액세스 or 반복X 시 느림
  - TLB 항목 개수 제한으로 자주 교체
- TLB hit 늘리기: TLB 키우기 (비용 ↑)
- TLB reach(TLB 도달 범위): TLB 채워졌을 때 미스 없이 작동 범위
  - `TLB 항목 수 * 페이지 크기`

10.4.2. 페이지 테이블 메모리 낭비

- 32비트 환경 프로세스 당 페이지 테이블 크기: 약 100만개 페이지
  - 10MB 사용 프로세스: 전체의 0.0024

⇒ 역 페이지 테이블(IPT), 멀티 레벨 페이지 테이블(n-lv page table)

Inverted Page Table, IPT
- 물리 메모리 프레임 번호 기준으로 테이블 작성
- 시스템 하나 당 IPT 하나
- 내용: `[PID, 페이지 번호 P]`
- 인덱스: `프레임 번호`
- IPT 사용 시 논리 주소 형식: `[PID, #page, offset]`
- IPT 사용 주소 변환: `VA=<PID, #page, offset>`
- 논리 주소에서 (PID, #page)로 역 페이지 테이블 검색
  인덱스가 `프레임 번호`이므로 항목 발견하면 인ㄷ게스 사용하면 됨

- IPT의 크기: `sizeof(pid) + sizeof(#page)`
  - `sizeof(pid) = 4, sizeof(#page) = 4 then 8`

Multi-level page table(Hierarchical Paging)
- **페이지 테이블을 수십 수백 개의 작은 페이지 테이블로 나눔: 여러 레벨로 구성**
- 2레벨 페이지 테이블 구성 시: `VA=<page-directory-index, #page, offset>`
  - 논리주소 상위 10비트: 페이지 디렉토리 인덱스
  - 논리주소 중위 10비트: 페이지 테이블 인덱스
  - 논리주소 하위 12비트: 페이지 내 오프셋

해시 페이지 테이블

10.5. 메모리 접근 권한

Segmentation Table

NX-bit Never eXecute Bit(a.k.a. DEP): 실행 방지 비트
- 프로세스 명령어, 코드, 데이터 저장을 위한 메모리 영역 따로 분리
- NX 특성으로 지정된 모든 메모리 구역: 데이터 저장에만 사용

10.6. 캐시 메모리

- 투명성
  - 하드웨어 단에서 처리, 프로그래머는 캐시 조작 불가
- `적중률 = 캐시 적중 회수 / 전체 메모리 참조 횟수`
- 평균 액세스 시간: $T_a = H \times T_c + (1 - H) \times Tm$

캐시 인덱싱
- 블록으로 구성됨: 각 블록, 데이터 가짐. 주소값을 키로 사용
  - 블록 개수, 블록 크기 ⇒ 캐시 크기 결정
  - 주소값 전체를 키로 사용 X, 일부만을 사용
    블록 1024개, 블록 사이즈 32바이트, 32비트 주소: `any, 10bit: index, 5bit: offset`

Tag Matching
- 인덱스 충돌 회피: 주소값의 일부를 태그로 사용

- 블록 개수 1024개, 블록 사이즈 32B, 32비트 주소 시
- `18: tag 10: index 4: offset`

Tag Overhead
- 태그 추가, but 캐시 사이즈 변함 없음:
  캐시 메모리 용량은 태그 메모리 빼고 데이터 메모리 용량만 의미
- 태그 공간: 블록 크기와 무관한 오버헤드 취급
- 1024개 32B 블록으로 구성된 32KB 캐시 태그 오버헤드
- 레이턴시 증가
  1. 태그 배열 접근: 히트 확인
  2. 데이터 배열에 접근, 데이터 가져옴

Tag Matching 장단점
- 장점
  - 태그 길이 짧음
  - CPU 태그, 하나의 캐시 태그와 비교: 하나의 비교기만 있으면 됨
  - 하드웨어 구현 단순, 접근 속도 빠름
- 단점
  - 적중률 저조
  - (Ping-pong) 동일한 캐시 블록에 사상되는 다른 메모리 블록 번갈아 참조: 심각한 충돌
  - 대용량 캐시 메모리일때만 직접 사상

Associateive Cache
- 핑퐁 문제 해결: 태그 배열, 데이터 배열 여러개 생성

- Direct mapped (직접 매핑): 인덱스가 가리키는 공간이 하나: 처리 빠름, 충돌 잦음
- Fully associative (완전 연관): 인덱스가 모든 공간 가리킴: 처리 느림, 충돌 적음
  - 모든 캐시 태그와 병렬 처리 필요
  - 태그 길이 김
  - 직접 사상에 비해 느린 속도, 추가 HW 필요
- Set associative(집합-연관 사상): 인덱스가 가리키는 공간 두 개 이상
  - 완전 연관 사상 + 직접사상: 전체 태그 대신 일부 태그에 대해 연관 탐색

10.6.1. Write on cache
- Write-through
  - 캐시 업데이트마다 메모리 데이터 업데이트
- Write-back
  - 블록 교체 시에만 업데이트

11. Demand Paging

11.1. Swapping
- 물리 메모리 영역을 하드디스크까지 연장
- 프로세스 전체가 물리 메모리에 적재될 필요 없음
  - 물리 메모리 일부분 하드디스크로 옮겨 메모리 빈 영역 확보
- 사용자: 컴퓨터 시스템에 무한대의 메모리가 있는걸로 착각
- 물리 메모리 확장하여 사용하는 디스크 영역: 스왑 영역

11.2. Demand Paging
- 일부만 메모리에 할당, 페이지가 필요할 때 메모리 할당받고 페이지 적재
- Demand Paging = Paging + Swapping

Page Table Entry

page number	a	m	v	r	w	x	v ? storage block address : frame number

Page Fault
- 빈 프레임 할당 → 스왑 영역, 실행 파일로부터 페이지 적재

프로세스의 실행
- 첫 페이지만 물리 메모리에 적재, 실행 중 다음 페이지가 필요하면 필요할 때 적재

Fork() 관련) Copy on Write
- 쓰기 시 복사: 자식 프로세스는 초기에 부모의 메모리 프레임 완전 공유
  - 수정할 때 새 프레임 할당받음

11.3. 페이지 교체 알고리즘
- Working Set
  - 일정 시간 범위 내에 프로세스가 액셋한 페이지들의 집합
  - 페이지 폴트: 작업 집합을 메모리에 적재하는 과정
  - 참조 지역성 → 페이지 폴트 → 메모리에 작업 집합 형성

  ∴ 일정 시간 동안 참조한 페이지 집합

- 페이지 교체
  - 메모리 프레임 중 하나 선택하여 비운 후, 요청된 페이지 적재
  - 범위
    - 지역 교체
      - 프로세스별 독립적 페이지 폴트 처리
      - 한 프로세스에서 발생한 스래싱, 다른 프로세스로 전파되지 않음
    - 전역 교체
      - 전체 메모리 프레임에서 선택

종류	알고리즘	특징
간단함	무작위	무작위로 페이지 스왑
	FIFO	First In First Out
이론적	최적	미래의 접근 패턴 → 대상 페이지 선정
최적 근접	LRU	시간적으로 멀리 떨어진 페이지 스왑 • 타임 스탬프 이용 • 참조 비트 시프트 사용
	LFU	사용 빈도가 적은 페이지 스왑 • 페이지 사용 횟수 기준
	NRU, NUR	Not Recently Used // 최근에 사용 않은 페이지를 스왑
	FIFO 변형	FIFO 변형하여 성능 높임

11.4. 스레싱
- Page fault ↑ ⇒ CPU Utilization ↓
- CPU, 이용률 떨어지므로 메모리에 더 많은 프로세스 로드
- 메모리 부족한데 더 로드 시도하므로 메모리 더더욱 부족
- 하드디스크 입출력 너무 많아짐: 잦은 페이지 부재로 작업 멈춘것 같이 보임

스레싱의 해결
- 다중 프로그래밍 정도 줄이기(프로세스 종료)
- IO빠른 SSD 사용
- 메모리 늘리기
- 프로세스에 너무 적은 프레임 할당: 페이지 부재 ↑
  프로세스에 너무 많은 프레임 할당: 메모리 낭비
  → 프레임 할당 문제

11.5. 프레임 할당
정적 할당
- 균등 할당: 프로세스에게 크기와 무관하게 동일 개수 프레임 할당
- 비례 할당: 프로세스 크기에 비례하여 프레임 할당
  - 페이지 폴트 수 줄일 수 있음
  - 프로세스 크기, 실행 중에 완벽히 알기 어려움: 실행 중에 작업 집합 판단 필요

동적 할당
- Working Set model
  - 최근 일정 시간동안 참조된 페이지, 집합화: 페이지들 물리 메모리에 유지
    - 작업 집합 크기: 작업 집합의 크기
    - 작업 집합 윈도우: 크기 범위 (내에 있는 작업)
- Page Fault Frequency
  - 페이지 부재 횟수 기록 ⇒ 페이지 부재 비율 계산
    - 상한 초과 시 프레임 추가
    - 하한 하회 시 프레임 회수
    - 페이지 부재 수를 적절한 선으로 조정

```
mov eax, 23
; 최소 1개의 페이지 (코드 1페이지)

mov eax, [mem_addr]
; 최소 2개의 페이지 (코드 1페이지 + 데이터 1페이지)

mov eax, [[mem_addr] + 5000]
; 최소 3개의 페이지 (코드 1페이지 + [mem_addr] 1페이지, [[mem_addr] + 5000] 1페이지)
```

12. File System

12.1. 저장장치

HDD
- 섹터: 정보가 저장되는 최소 단위
- 트랙: 정보가 저장되는 하나의 동심원 ∋ 섹터
- 실린더: 같은 반지름을 가진 모든 트랙 집합 ∋ 트랙
- 섹터 ∈ 트랙 ∈ 실린더

- 블록: 운영체제가 파일 데이터 io하는 논리 단위, 몇 개의 섹터로 구성

디스크의 용량: `실린더 개수×실린더당 트랙 수×트랙당 섹터 수×섹터 크기`

12.2. 디스크의 IO
디스크의 IO
1. 탐색 시간: 데이터가 위치한 트랙까지 헤드 이동
2. 회전 지연 시간: 트랙 내에서 데이터 시작지점 탐색
3. 데이터 전송 시간: 섹터에 나열된 만큼

디스크의 회전
- 각속도 일정 방식
  - 트랙마다 선속도 다름: 섹터 크기 다름 (바깥 섹터 > 안쪽 섹터)
- 선속도 일정 방식
  - 디스크 회전 속도 가변: 모터 제어 복잡

데이터 주소
- 디스크 물리 주소: `CHS = (cylinder #, head #, sector #)` (기본 단위: 섹터)
- 논리 주소 `LBA`: 저장 매체, 1차원 연속된 데이터 블록으로 봄 (바깥→안, 위→아래)

12.3. 파티션과 레이드
- Raid: Redundant Array of Independent Disks
- Raid 0: Striping
- Raid 1: Mirroring
- Raid 01: Striping before Mirroring
- Raid 10: Mirroring before Striping
- Raid 2: Hamming Code ECC (ECC 전용 디스크)
- Raid 3: Parallel Transfer with Parity (패러티 전용 디스크)
- Raid 4: Independent Data Disks with Shared Parity Disk
- Raid 5: Independent Data Disks with Distributed Parity Blocks
  패러티 비트, 여러 디스크에 분산 보관
- Raid 6: Independent Data Disks with Two Independent Distributed Parity Schemas
  패러티 비트 두개, 여러 디스크에 분산 보관

12.4. 디스크 스케줄링
- 탐색: 디스크 헤드를 목표 실린더로 이동
  - 탐색 거리: 이동 실린더 개수
  - 탐색 시간
- 회전 지연: 플래터 회전, 목표 섹터 도달할 때까지 대기
  - 평균 회전 지연 시간: 디스크 회전 시간의 1/2
  - 1회전 시간 = `60sec / rpm` (rotation per minute)
- 전송, 오버헤드

- 디스크 액세스 시간: 목표 섹터에 접근, IO까지 걸리는 시간
- 탐색 시간 + 회전 지연 시간 + 내부 전송 시간

알고리즘	특징	장점	단점
FCFS	도착순	구현 쉬움 기아 없음	처리율 낮음
SSTF	가장 가까운 요청 우선	처리율 최고 높음	응답 시간 편차 큼 기아 가능
SCAN	한쪽에서 다른쪽으로 훑듯	SSTF보다 균등한 서비스 기아 없음	중간 요청이 높은 서비스 확률 SSTF보다 처리율 낮음
LOOK	SCAN + 더 이상 요청 없으면 방향 전환	SSTF보다 균등한 서비스 기아 없음 SCAN보다 처리율 높음	SSTF보다 처리율 낮음
C-SCAN	한쪽 방향으로만 이동 (원형큐처럼)	SSTF보다 균등한 서비스	SSTF보다 처리율 낮음
C-LOOK	LOOK의 C-SCAN 판	기아 없음	SSTF보다 처리율 낮음

12.5. SSD
- SSD: 블록, 페이지로 구성, 저장단위: 페이지
- 페이지 ∈ 블록

FTL
- 하드디스크 기반 시스템과의 호환성을 위한 레이어

Garbage Collection
- 블록 내 빈 페이지가 없을 때, 블록 내 Dirty 페이지만 제거하여 새 블록에 내용물 재할당

12.6 파일 시스템

FAT(File Allocation Table)

Boot Sector	FAT1	FAT2(backup)	Root Directory Blocks..	Data Blocks...

FAT1

Reserved···	Root Directory Block #..	-1

Root Directory Blocks

Filename	Type	...	FAT #	Size
a	txt	...	52	9000
b	txt	...	320	200

xNix FileSystem
- 부트 블록: 부팅 시 적재, 실행 코드
- 슈퍼 블록: fs 메타정보 저장
- i-node & i-node 리스트
  - i-node: 파일당 1개의 i-node 필요, 메타데이터 저장

Boot Block	Super Block	i-node List	Data Blocks...

12.7. Formatting
- 저수준: at Factory
- 고수준: 파티션 및 파일시스템 구축, 부팅 코드 탑재

- 부팅: 바이오스가 디스크 포맷의 특정 영역(=부트 섹터) 호출하는 것

MBR: Master Book Record
- 첫 섹터: 부트 섹터(512Byte), 파티션 크기 2TB

GPT: GUID Partition Table

- UEFI 펌웨어 가진 컴퓨터에서 사용 가능
- 128개까지 파티션 분할 가능, 최대 18EB

12.8 File IO
- OS끼지 않고 디스크 접근 불가
- 자료구조
  - i-node 테이블
  - 오픈 파일 테이블: 시스템에서 열린 모든 파일에 대한 정보 (`offset, access mode, i-node, ...`)
  - 프로세스 별 오픈 파일 테이블
    항목 번호: `open`의 리턴 값(파일 디스크립터)
  - 버퍼 캐시