운영체제: 강의노트 12

A. Silberschatz, P.B. Galvin, G. Gagne
Operating System Concepts,
Sixth Edition, John Wiley & Sons, 2003.

Part III. Storage Management

11 파일시스템 인터페이스

11.1 파일 개념

- 우리는 정보를 다양한 저장 매체(디스크, 광디스크, 테이프 등)에 저장한다. 운영체제는 다양한 저장 매체에 대한 균일한 논리적 뷰를 제공한다. 이것이 **파일(file)**이다.
- 파일은 저장의 최소 단위이다. 우리는 파일 외에 다른 방법으로 데이터를 저장할 수 없다.
- 파일은 그것의 유형에 따라 특정 구조를 지닌다.

11.2 파일 속성

- 파일과 관련된 속성
  - 이름: 사용자에 의해 인식될 수 있는 형태로 유지되는 유일한 것
  - 식별자: 파일 시스템 내에서 각 파일을 구분하기 위한 식별자
  - 유형
  - 위치: 저장된 매체에 파일이 저장된 위치
  - 크기
  - 보호: 접근제어 정보
  - 시간과 날짜, 사용자 정보
- 파일과 관련된 이런 정보들은 디렉토리 구조 형태로 보조 저장장치에 유지된다.
- 전형적으로 디렉토리 항은 파일 이름과 그것의 유일한 식별자로 구성되며, 이 식별자는 다른 속성들이 있는 위치를 가리키는 포인터 역할을 한다.

11.2.1 파일 연산

- 파일과 관련된 연산
  - 파일 생성: 공간을 할당해야 하고, 파일과 관련된 정보를 디렉토리에 추가해야 한다.
  - 파일 쓰기
  - 파일 읽기
  - 파일 내의 위치 변경
  - 파일 삭제: 파일에게 할당되었던 공간을 회수하고, 디렉토리에서 파일과 관련된 정보를 삭제한다.
  - 파일 절단: 파일 내의 내용만 삭제한다. 즉, 파일에게 할당된 공간을 회수하고 파일 관련된 정보 중 크기만 0으로 바꾼다.
- 이 외에도 파일 끝에 첨가, 파일 이름 변경, 파일 복사 등을 제공한다.
- 매 번 디렉토리를 검색하는 비용을 줄이기 위해 파일 읽기나 쓰기를 하기 전에 파일을 열도록 한다. 운영체제는 열린 파일 테이블(open-file table)을 유지하며, 파일을 닫을 때까지 디렉토리 검색을 다시 하지 않고 파일을 조작할 수 있도록 해준다.
- 파일 열기: 파일 이름을 주면 디렉토리를 검색하여 열린 파일 테이블에 파일의 디렉토리 항을 복사한다. 파일을 열 때 접근 모드(생성, 읽기, 읽기-쓰기 등)를 지정할 수 있다. 보통 열기 시스템 호출은 이 테이블에 대한 포인터를 반환해준다.
- 다중 사용자 환경의 경우에는 두 단계 테이블을 사용한다.
  - **프로세스별 테이블**: 각 프로세스의 PCB에 유지되며, 이 프로세스가 연 파일들을 관리한다. 파일 포인터, 파일 접근 모드 등이 이 테이블에 유지된다.
  - **시스템 전체 테이블**: 프로세스와 독립적인 정보들이 유지된다. 디스크 내의 파일의 위치, 접근 날짜, 파일 크기 등이 이 테이블에 유지된다. 또한 열린 계수(open count)를 유지하여 이 파일을 연 프로세스의 수를 유지한다.

11.2.2 파일 유형

- 파일의 유형은 확장자를 이용하여 식별한다. 하지만 보통 운영체제는 일부의 확장자(보통 실행, 텍스트, 이진 정도만 구분)만 식별하며, 나머지는 사용자에게만 도움이 되는 구분자 역할을 한다.
- 최근에는 파일 유형과 관련 프로그램을 연관하여 파일 선택하면 자동으로 관련 프로그램이 실행되도록 한다.

  참고. 문서 중심 vs. 프로그램 중심

11.3 디렉토리 구조

- 디스크는 **파티션(partition)**으로 분할된다. 디스크는 보통 최소한 하나의 파티션을 가진다. 어떤 시스템은 여러 디스크를 하나의 파티션으로 사용할 수 있도록 해준다.

- 각 파티션은 저장되어 있는 파일과 저장된 파일들에 관한 정보로 구성된다. 파일에 관한 정보는 장치 디렉토리 또는 볼륨 테이블에 유지된다. 장치 디렉토리는 보통 디렉토리라 하며, 저장되어 있는 파일마다 디렉토리에 하나의 항이 존재한다.
- 디렉토리와 관련된 연산
  - 파일 검색: 특정 파일에 해당하는 디렉토리 항을 검색한다. 보통 파일 이름이 특정 패턴을 일치하는 모든 파일들을 찾을 수 있도록 해준다.
  - 파일 생성
  - 파일 삭제
  - 디렉토리 나열
  - 파일 재명명
  - 파일 시스템 순회(traverse): 파일 시스템 내에 있는 모든 디렉토리를 방문하는 것을 말한다.

11.3.1 단일 단계 디렉토리

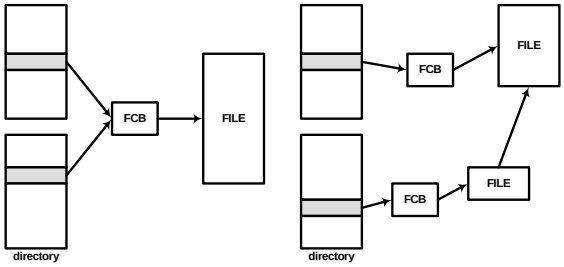
- 모든 파일이 하나의 디렉토리에 포함되는 구조를 말한다.
- 지원하기 가장 쉬운 방법이다.
- 문제점: 한 디렉토리 내에 있는 모든 파일의 이름은 독특해야 하므로 파일 이름 부여에 제약을 받는다.

11.3.2 두 단계 디렉토리

- 두 단계부터는 경로(path) 개념이 등장한다.
- 검색 경로(search path): 파일 이름이 주어졌을 때 이 파일을 검색하기 위해 검색하는 디렉토리의 목록을 말한다.

11.3.3 트리 구조의 디렉토리

- 트리 구조의 디렉토리에서 각 디렉토리는 하위 디렉토리를 가질 수 있으며, 트리의 최상위 디렉토리를 **루트 디렉토리**라 한다.
- 현재 디렉토리(current directory): 현재 사용자가 위치한 디렉토리로 파일 이름을 주면 이 디렉토리를 먼저 검색한다. 현재 디렉토리와 다른 디렉토리에 있는 파일을 접근하기 위해서는 그 디렉토리로 이동을 하거나 파일의 경로를 주어야 한다.
- 절대 경로(absolute path): 트리의 루트부터의 경로. 예) C:\windows\system32
- 상대 경로(relative path): 현재 디렉토리로부터의 경로. 예) ..\work
- 트리 구조에서 디렉토리 삭제



(a) 하드 링크 (b) 기호 링크

<그림 11.1> 유닉스에서 링크의 종류

  - 방법 1. 하위 디렉토리가 있으면 거부한다. 예) MSDOS
  - 방법 2. 삭제 명령에 옵션을 줄 수 있도록 하여 하위 디렉토리가 있어도 디렉토리를 삭제할 수 있도록 해준다. 예) UNIX

11.3.4 비순환 그래프 디렉토리

- 비순환 그래프란 주기가 없는 그래프를 말한다.
- 이 방식은 디렉토리나 파일의 공유를 쉽게 해준다.
- 구현 방법
  - 링크(link) 사용: 링크는 파일 또는 디렉토리에 대한 포인터이다. 유닉스는 기호 링크(symbolic link)와 하드 링크(hard link), 두 종류를 제공한다.
  - 파일 중복: 일관성을 유지하는 것이 어렵다.
- 문제점: 트리 구조보다는 유연한 방법이지만 다음과 같은 문제점이 있다.
  - 한 파일은 여러 경로를 가질 수 있다. 따라서 파일 시스템을 순회할 때 한 구조를 여러 번 방문할 수 있다.
  - 파일의 삭제: 링크되어 있는 파일을 삭제할 경우 이 링크가 더 이상 의미있는 곳을 가리키지 못하게 된다. 파일에 대한 링크 목록을 유지하지 않으면 이 링크들을 찾아 삭제하는 것은 많은 비용이 소요된다. 보통 파일이 삭제되어도 링크는 그대로 두며, 다만 이 링크를 나중에 사용하면 경고를 줄 수 있도록 고안한다. 또 다른 방법은 참조 계수를 유지하여 이 계수가 0이 되어야만 삭제할 수 있도록 한다.
- 기호 링크와 하드 링크의 차이점
  - 기호 링크는 파일 또는 디렉토리에 대한 포인터로서, 기호 링크의 삭제는 파일에 영향을 주지 않는다. 반대로 파일이 삭제되면 기호 링크는 의미없는 포인터(dangling pointer)가 된다.

  - 하드 링크는 파일에 대해서만 가능하고, 같은 디렉토리 항을 가리키는 포인터이며, 한 파일 시스템 내에서만 가능하다. 하드 링크를 삭제하면 참조 계수만 하나 감소한다.

11.3.5 일반 그래프 디렉토리

- 비순환 그래프 구조의 문제는 주기가 없도록 보장하는 것이다.
- 비순환 그래프의 장점은 파일 시스템 순회가 쉽다는 것이다.
- 일반 그래프 구조에서는 주기가 존재할 수 있다.
- 일반 그래프 구조의 문제점
  - 파일 시스템 순회의 어려움
  - 파일 삭제의 어려움: 참조 계수를 사용할 수 있지만 참조 계수가 0이 아님에도 불구하고 더 이상 의미 없는 링크가 존재할 수 있다. 이 경우에는 주기적으로 쓰레기 수집(garbage collection)을 해야 하지만 이것의 비용이 크므로 이런 기법을 사용하는 경우는 드물다.
- 실제 일반 그래프 디렉토리 구조를 사용하여도 주기는 보통 무시한다. 즉, 사용자 책임이며, 순회할 때에는 링크는 무시된다.

11.4 파일 시스템 마운팅

- 파일은 조작하기 전에 열어야 조작할 수 있다.
- 이와 마찬가지로 파일 시스템을 사용하기 위해서는 먼저 마운트되어야 한다.
- 마운트 절차
  - 운영체제는 장치의 이름과 파일 구조 내에 마운트할 위치를 받아 그 위치에 마운트한다.
  - 보통 마운트할 위치는 빈 디렉토리가 된다.
  - 운영체제는 파일 시스템을 마운트하기 전에 장치를 검증하여 유효한 파일 시스템이 장치에 포함되어 있는지 검사한다. 이 검사는 장치의 디렉토리 파일이 요구되는 형태를 갖추고 있는지를 검사하게 된다.
- 기능의 명확성을 보장하기 위해 시스템은 마운트에 대한 몇 가지 제약을 강요할 수 있다.
  - 예1) 파일을 포함하고 있는 디렉토리는 마운트 위치로 사용할 수 없다.
  - 예2) 파일 시스템은 오직 한번만 마운트되도록 제한할 수 있다.
- 윈도우 시스템은 부팅 때 시스템을 검색하여 모든 파일 시스템을 자동으로 마운트한다.

11.5 보호

- 물리적 손상과 불법적 접근으로부터 파일을 보호해야 한다.
- 물리적 손상은 백업을 통해 이루어지며 불법적 접근은 접근 제어(access control)를 통해 이루어진다.

11.5.1 접근 유형

- 파일에 대한 보호는 접근 유형을 제한하여 제어한다.
- 접근의 유형
  - 읽기
  - 쓰기
  - 실행
  - 첨가: 파일 끝에 내용 추가
  - 삭제
  - 파일 정보 열람

  이름 변경, 복사, 편집 등은 하위 수준의 기본 접근 유형에 따라 제어된다. 예를 들어 읽기 권한이 있는 사용자는 파일을 복사할 수 있다.

11.5.2 접근 제어

- 접근 제어는 보통 사용자 식별자를 근거로 접근 허용 여부를 결정한다.
- 각 파일마다 접근 제어 목록(access control list)를 유지하며, 사용자가 접근 요청하면 이 목록을 검토하여 접근 허용 여부를 결정한다.
- 접근 제어 목록에 각 사용자별 접근 권한을 유지하는 것은 현실성이 없다. 보통 사용자를 다음과 같이 분류하고, 분류에 대한 접근 권한만 유지한다.
  - 소유자(owner): 파일을 생성한 사용자
  - 그룹(group): 소유자가 속한 그룹
  - 일반(universe): 시스템에 등록된 모든 사용자

12 파일 시스템 구현

12.1 파일 시스템 구조

- 디스크의 특징
  - 재쓰기가 가능하다.
  - 직접 접근을 제공한다.
- 디스크 입출력은 보통 블록 단위로 이루어진다.
- 파일 시스템 설계 쟁점
  - 일반 사용자에게 어떻게 보여줄 것인가?

  - 논리적 파일 시스템을 물리적 보조저장장치에 어떻게 저장할 것인가?

- 파일 시스템은 보통 여러 단계로 구성되어 있다.
  - 단계 1. 장치 구동기(device driver)와 인터럽트 처리기: 장치 구동기는 번역기로서 상위 수준의 명령을 하드웨어 제어기가 이해하는 명령어로 바꾸어 하드웨어 제어기에 전달해주는 역할을 한다.
  - 단계 2. 기본 파일 시스템: 일반 명령을 장치 구동기에 전달한다. 일반 명령에는 블록 읽기와 블록 쓰기가 있으며, 블록 위치는 보통 장치번호, 실린더번호, 트랙번호, 섹터번호로 구성된다.
  - 단계 3. 파일 조직 모듈: 메타데이터를 관리한다. 메타데이터에는 파일의 내용을 제외한 파일 시스템 구조에 대한 모든 정보를 유지한다. 파일 구조는 보통 파일 제어 블록(FCB, File Control Block) 단위로 관리된다. FCB에는 파일의 소유권, 접근 제어, 파일의 위치 등을 유지된다.
- 대부분의 운영체제는 하나 이상의 파일 시스템을 지원한다. 윈도우 시스템은 FAT, FAT32, NTFS 파일 시스템을 지원하며, 유닉스 시스템은 UFS(Unix File System)을 지원한다.

12.2 파일 시스템 구현

12.2.1 개요

- 파일 시스템을 구현하기 위해 운영체제는 디스크와 메모리에 여러 정보를 유지한다.
- 디스크에 유지되는 정보
  - **부트 제어 블록(boot control block)**: 파티션으로부터 운영체제를 부팅하기 위해 필요한 정보를 여기에 유지한다. 전형적으로 부트 제어 블록은 파티션의 첫번째 블록에 저장되어 있다. 모든 파티션마다 있을 필요는 없다.
  - **파티션 제어 블록(partition control block)**: 파티션의 크기(블록 수), 블록의 크기, 빈 블록 수, 빈 블록 포인터, 빈 FCB 수, FCB 포인터 등과 같은 파티션의 정보를 유지하는 블록이다.
  - 디렉토리 구조
  - FCB: NTFS에 경우에는 파티션 제어 블록 내에 저장되지만 보통은 파타션 제어 블록과 별도로 저장된다.
- FCB를 UFS에서는 inode라 한다.
- 주기억장치에 유지되는 정보
  - 파티션 테이블: 마운트된 각 파티션에 대한 정보를 유지하는 테이블
  - 최근에 접근된 디렉토리 구조: 디렉토리에 대한 소프트웨어 캐시
  - 시스템 전체 연 파일 테이블(SWOFT, System-Wide Open-File Table): 연 파일의 FCB의 복사본을 유지하는 테이블
  - 프로세스 별 연 파일 테이블(PPOFT, Per Process Open-File Table): 시스템 전체 연 파일 테이블의 항을 가리키는 포인터를 유지하는 테이블
- 파일 생성
  - 새 FCB를 할당하고 해당 디렉토리를 주기억장치로부터 읽어 갱신한 다음 이 디렉토리를 디스크에 다시 쓴다.
  - 디렉토리를 파일과 동일하게 취급하는 시스템(예: 유닉스)이 있고, 별도의 시스템 호출을 사용하는 시스템(예: 윈도우)도 있다.
- 파일 열기
  - 파일 이름을 이용하여 디렉토리를 검색하여 파일에 해당하는 항을 찾아 그 파일의 FCB를 SWOFT에 복사한다. SWOFT는 각 파일마다 그것을 연 프로세스의 수를 나타내는 열기 계수(open count)을 유지한다. 그 다음 PPOFT에 새 항을 추가한다. 이 항에는 SWOFT에 유지되고 있는 FCB에 대한 포인터와 파일 포인터 등이 유지된다.
  - 열기 시스템 호출은 PPOFT에 새롭게 추가한 항에 대한 포인터를 반환하여 주며, 그 이후 파일 조작은 이 포인터를 이용한다. PPOFT의 항을 유닉스에서는 파일 설명자(file descriptor)라 하고, 윈도우에서는 파일 핸들(file handle)이라 한다.
- 파일 닫기
  - PPOFT의 해당 항을 삭제한다.
  - SWOFT의 열기 계수를 하나 감소한다.
  - 파일을 연 모든 프로세스가 파일 닫으면 갱신된 파일 정보를 디스크에 쓰고 SWOFT의 항을 제거한다.

12.2.2 파티션과 마운팅

- 각 파티션은 파일 시스템을 포함할 수도 있고 (“cooked”), 포함하지 않을 (“raw”) 수도 있다.
- 원시 디스크(raw disk)를 그대로 사용하는 경우도 있다.
  - 예1) 유닉스의 스왑 공간
  - 예2) 어떤 데이터베이스는 자체적인 형식을 사용하기 위해 원시 디스크를 사용하는 경우가 있다.
- 부트 정보를 별도 파티션에 저장하는 경우도 있다.

- 보통 부팅할 때에는 파일 시스템 장치 구동기가 실행되기 전 상태이므로 자체 형식을 사용한다.
- 다중 부팅을 하기 위해서는 부트 파티션에 여러 파일 시스템을 이해하는 부트 적재기(boot loader)가 있어야 한다.
- 운영체제 커널을 포함하는 루트 파티션은 시스템이 부팅될 때 자동으로 마운트되며, 다른 파티션은 부팅 때 자동으로 마운트되거나 나중에 수동으로 마운트된다.
- 운영체제는 주기억장치에 마운트 테이블을 유지한다.

12.3 디렉토리 구현

12.3.1 선형 리스트

- 가장 단순한 방법은 파일 이름과 데이터 블록에 대한 포인터로 구성된 선형 리스트를 유지하는 것이다.
- 선형 리스트에서 특정 파일을 찾기 위해서는 선형 검색이 필요하다. 또한 파일을 생성할 때에는 선형 검색을 하여 같은 이름의 파일이 없는지 검사하여야 한다. 이것이 선형 리스트의 가장 큰 단점이다.
- 디렉토리 항을 재사용하기 위해 삭제된 항은 빈 상태로 유지할 수 있다. 또한 검색 비용을 줄이기 위해서는 마지막 항을 빈 항에 옮길 수 있다. 또는 연결 리스트 형태로 유지할 수 있다.
- 선형 리스트의 단점을 극복하기 위해 사용된 항은 소프트웨어적으로 캐시한다.
- 정렬된 리스트를 유지하면 이진 검색을 할 수 있지만 정렬을 유지하는 비용이 추가로 소요된다. 다만, 파일 목록을 정렬된 형태로 출력할 때는 별도의 정렬 과정이 필요없다.

12.3.2 해시 테이블

- 해시 테이블을 이용하여 디렉토리를 구현할 수 있다.
- 검색 시간이 빠르며, 삽입과 삭제도 용이하다. 다만 해시 충돌 문제를 해결해야 한다.

12.4 할당 방법

12.4.1 연속 할당

- 이 방식에서는 각 파일을 연속된 공간에 할당한다. 동적 주기억장치 관리 문제와 유사하다.
- 빈 공간을 찾기 위해 홀들을 유지해야 하며, 주기억장치 관리와 마찬가지로 최초 적합, 최적 적합, 최악 적합 방법을 모두 사용할 수 있다.
- 단점
  - 주기억장치 관리와 마찬가지로 외부 단편화가 발생할 수 있다.
  - 파일을 생성할 때에는 파일의 크기를 알 수 없는 경우가 많다.
    - 예측을 하여 최적 적합 방식을 사용하면 공간이 부족할 수 있다. 이 처럼 예측되어 확보된 공간이 부족하면 보다 큰 공간으로 이동할 수 있지만 시간이 많이 소모된다.
    - 파일 복사처럼 크기를 정확하게 예측할 수 있어도 파일의 크기가 시간이 경과됨에 따라 변할 수 있다. 이것을 대비해 실제 크기보다 많은 공간을 미리 확보해주면 내부 단편화 문제가 발생할 수 있다.
    - 이런 문제를 극복하기 몇 공간을 연결 리스트 형태로 유지하여 공간을 할당해줄 수 있다.
- 장점
  - 파일 접근 시간이 빠르다.
  - 파일에 대한 직접 접근과 순차 검색을 모두 지원한다.
  - 디렉토리 항에는 시작 위치와 크기만 유지한다.

12.4.2 연결 할당

- 파일을 디스크 블록의 연결 리스트 형태로 할당하며, 디렉토리 항에는 첫번째 블록과 마지막 블록에 대한 포인터만 유지한다. 각 블록 내에는 다음 블록을 가리키는 포인터를 유지한다.
- 장점
  - 외부 단편화가 발생하지 않는다. 내부 단편화는 발생할 수 있지만 무시할 수 있다.
  - 파일의 생성과 삭제가 쉽다.
  - 이 방식은 파일의 크기 변화에 대해 쉽게 대처할 수 있다.
- 단점
  - 파일에 대한 순차 검색만 지원한다. 직접 접근은 연결 리스트에 따라 찾아가는 방법밖에 없다.
  - 많은 공간이 포인터 값을 저장하기 위해 사용된다. 이것을 극복하기 위해 블록들을 묶어 클러스터 단위로 할당할 수 있다. 클러스터 방법은 비록 내부 단편화 문제가 발생하지만 현재 널리 사용되고 있다.
  - 신뢰성도 떨어진다. 포인터 값이 하나만 손상되어도 파일 자체가 손상된다. 이것을 극복하기 위해 이중 연결 리스트로 유지할 수 있지만 이 경우에는 포인터를 저장하기 위한 오버헤드가 더 커진다.

- 예) FAT
  - MSDOS에서 사용한 연결 할당 방법
  - 디스크의 일부분에 블록에 대한 테이블을 유지한다. 디스크의 블록마다 이 테이블에 한 항이 유지되며, 블록 번호에 의해 색인된다.
  - 디렉토리 항에는 파일의 첫 번째 블록 번호를 유지한다. 이 블록 번호에 해당하는 테이블의 항에는 파일의 다음 블록 번호를 유지한다. 파일의 마지막 블록에 해당하는 테이블의 항에는 파일의 끝을 나타내는 특수한 값이 유지된다.
  - 사용되지 않는 블록에 해당하는 테이블 항의 값은 0이다. 블록의 할당은 테이블에서 값이 0인 항을 찾아 할당해준다.
  - 보다 빠른 검색을 위해 FAT을 캐시하여야 한다.

12.4.3 색인 할당

- 색인 할당은 디스크의 한 블록에 파일의 나머지 블록에 대한 포인터를 유지하는 방법이다. 이 블록을 **색인 블록**이라 한다.
- 색인 할당 방법은 파일에 대한 직접 접근을 용이하게 해준다.
- 장점: 색인 할당 방법은 연결 할당 방법의 장점을 모두 가지고 있으며, 여기에 추가로 빠른 직접 접근을 지원한다.
- 단점: 색인 블록 유지를 위한 공간 오버헤드가 존재한다.
- 색인 블록의 크기
  - 연결 방식: 처음에는 디스크의 한 블록만 색인 블록으로 할당한다. 만약 이 이상 필요하면 연결 리스트 형태로 색인 블록을 유지한다.
  - 다단계 색인 방식: 다단계의 색인 블록을 유지하는 방식이다.
  - 결합 방식: 색인 블록의 15개 정도의 포인터를 inode에 유지하는 방식으로, 이 중 대부분 항은 직접 색인을 유지하고, 나머지 항은 다단계 색인 방식을 사용하는 방식이다. 이것의 장점은 작은 크기의 파일은 별도의 색인 블록이 필요없다.

12.4.4 성능

- 디스크 공간 할당 방법의 성능은 저장 공간 효율과 접근 속도 두 가지 측면에서 고려된다.
- 접근 속도 측면에서 할당 방법을 비교하면 연속 할당이 가장 우수하다. 그러나 파일의 변화를 대처하기 어렵고, 디스크 공간 할당이 어렵다. 표 12.1 참조

<표 12.1> 디스크 할당 방법의 성능 비교: 접근 속도

	연속 할당	연결 할당	색인 할당
순차	우수	보통	보통
직접	우수*	나쁨	우수

*: 파일 전체가 연속 공간에 저장되어 있을 경우

- 파일 j번째 블록을 읽기 위해 읽는 블록의 수
  - 연속 할당: 1
  - 연결 할당: $j-1$
  - 색인 할당: $i+1$, 여기서 i는 j 블록을 읽기 위해 읽는 색인 블록의 수
- 할당 방법의 장단점을 모두 활용하기 위해 여러 방법을 함께 사용하는 경우가 있다.
  - 예) 작은 파일에 대해서는 연속 할당 방법을 사용하고 파일이 커지면 색인 할당 방법으로 전환한다.

12.5 빈 공간 관리

12.5.1 비트 벡터

- 비트 벡터(bit vector) 방법에서는 각 블록마다 1비트를 사용하여 블록의 할당 여부를 나타낸다. 블록이 빈 상태이면 이 값이 1이고, 할당되어 있으면 0이다.
- 장점: 첫 번째 빈 블록을 찾기가 단순하며 효율적이다. 이것은 한 워드 내에서 비트의 값이 1인 첫 위치를 찾아주는 기계 명령어가 보통 제공되기 때문이다.
- 비트 벡터 방법은 이 벡터를 모두 주기억장치에 유지할 경우에만 효율적으로 사용할 수 있다.

12.5.2 연결 리스트

- 빈 블록을 연결 리스트 형태로 유지하고, 첫 빈 블록에 대한 포인터를 운영체제가 주기억장치에 유지하는 방식이다.
- 빈 블록 목록에 대한 순회가 효율적이지 못하며, 주어진 크기의 빈 공간을 찾기 어렵다.
- 이 방식은 항상 빈 첫 블록을 할당에 사용하는 FAT과 같은 할당 방식에서만 사용할 수 있다.

12.5.3 그룹핑

- n개의 빈 블록 주소를 첫 빈 블록에 저장하고, n개 중 마지막 주소는 그 다음 $n-1$개의 빈 블록 주소를 포함하고 있는 블록을 가리키도록 유지하는 방식이다.

12.5.4 카운팅

- 연속 할당에서 사용할 수 있는 방법으로서, 빈 블록을 연결 리스트로 유지하기 보다는 빈 공간을 연결 리스트로 유지한다. 이 때 빈 공간마다 첫 블록 주소와 총 블록 수를 유지한다.

14 대용량 저장 구조

14.1 디스크 구조

- 디스크는 논리적으로 매우 큰 일차원 블록의 배열로 간주되며, 이 블럭은 전송의 최소 단위이다. 블록의 크기는 보통 512 KB이다.

- 이 블록들은 디스크의 섹터로 사상된다. 그러나 불량 섹터의 존재 가능성과 트랙 당 섹터의 수가 다를 수 있으므로 논리적 순서를 물리적 순서로 사상하는 것이 쉽지 않다.

14.2 디스크 스케줄링

- 디스크의 접근 시간에 가장 큰 영향을 주는 것은 탐색 시간(seek time)이다.

- 디스크 입출력 요청은 다음과 같은 정보를 지정한다.

  - 연산의 유형: 입력 또는 출력
  - 디스크 내의 위치 또는 주소
  - 주기억장치 내의 위치
  - 전송할 바이트의 수

- 디스크 드라이브와 제어기가 모두 사용 가능하면 입출력 요청은 즉시 처리될 수 있다. 그러나 둘 중에 하나가 다른 것을 서비스하고 있으면 요청은 큐에서 대기하여야 한다.

- 예) 디스크 헤드의 현재 위치가 53 트랙이고, 디스크의 총 트랙 수가 200이라고 하며, 디스크 큐에 있는 요청의 주소가 다음과 같다고 하자.

$$98, 183, 37, 122, 14, 124, 65, 67$$

14.2.1 FCFS 스케줄링

- 가장 단순한 방법이지만 헤드 움직임을 최적화하지 않기 때문에 성능이 나쁠 수 있다.

- 예) 스케줄링 결과

$$98, 183, 37, 122, 14, 124, 65, 67$$

  전체 움직임: 640

14.2.2 SSTF 스케줄링

- SSTF(Shortest-Seek-Time-First) 알고리즘은 탐색 시간이 가장 작은 것부터 먼저 처리하는 방식이다.

- 예) 스케줄링 결과

$$65, 67, 37, 14, 98, 122, 124, 183$$

  전체 움직임: 236

- 굶주림 현상이 발생할 수 있다.

- SSTF 알고리즘은 최적은 아니다.

$$37, 14, 65, 67, 98, 122, 124, 183$$

  전체 움직임: 208

14.2.3 SCAN 스케줄링

- 스캔 알고리즘은 한 쪽 끝에서 다른 쪽 끝으로 이동하면서 처리한 다음 방향을 바꾸어 처리하는 방식이다. 다른 말로 엘리베이터 알고리즘이라 한다.

- 예) 0으로 헤드가 움직이고 있을 때 스케줄링 결과

$$37, 14, 0, 65, 67, 98, 122, 124, 183$$

  전체 움직임: 236

- 끝에 가까워질 수록 새롭게 도착하는 요청을 바로 처리해주지 못한다. 따라서 끝에 가까워졌을 때 큐에서 대기하고 있는 요청은 대부분 반대쪽 끝 부분에 있을 것이다. 이 원리를 이용한 것이 C-SCAN이다.

- 예) 199으로 헤드가 움직이고 있을 때 C-SCAN 알고리즘의 스케줄링 결과

$$65, 67, 98, 122, 124, 183, 199, 0, 14, 37$$

  전체 움직임: 382

- 실제 구현에서는 0또는 199까지 이동하지 않는다. 이렇게 하는 것을 LOOK 스케줄링 알고리즘이라 한다.

- 예) 199으로 헤드가 움직이고 있을 때 C-LOOK 알고리즘의 스케줄링 결과

$$65, 67, 98, 122, 124, 183, 14, 37$$

  전체 움직임: 322

14.3 디스크 관리

14.3.1 디스크 포맷팅

- 물리적 포맷팅: 디스크를 디스크 제어기가 읽고 쓸 수 있도록 섹터 단위로 나누는 작업을 말한다. 다른 말로 저급 포맷팅(low-level formatting)이라 한다. 보통 물리적으로 포맷된 상태로 공장에서

출하된다. 섹터는 보통 헤더, 데이터 영역(512 바이트), 트레일러로 구성된다. 해더와 트레일러에는 섹터 번호, 오류 정정 코드 등을 기록하기 위해 사용된다.

- 논리적 포맷팅: 파티션을 만든 후에 초기 파일 시스템을 적재하는 과정을 말한다.

14.3.2 부트 블록

- 컴퓨터의 전원을 키면 부트스트랩 프로그램은 시스템의 모든 구성요소를 초기화하고, 디스크에서 운영체제 커널을 주기억장치에 적재한 다음 커널을 실행한다.
- 부트스트랩 프로그램은 ROM에 적재되어 있다. 이것의 장점은 다음과 같다.
  - 처음 부팅되었을 때 위치가 고정되어 있어 실행하기가 쉽다.
  - ROM은 읽기 전용 기억장치이므로 바이러스에 감염되지 않는다.

  그러나 ROM에 적재하면 부트스트랩 프로그램을 변경하기가 어렵다. 이 문제 때문에 부트스트랩 프로그램의 일부만 ROM에 적재하고, 나머지는 디스크에 유지한다. ROM에 있는 프로그램의 역할은 이 프로그램을 주기억장치에 적재하여 실행하는 것이다.
- 부트스트랩 프로그램이 적재되어 있는 디스크 블록을 부트 블록이라 하며, 부트 블록이 있는 디스크를 루트 디스크 또는 시스템 디스크라 한다.

14.3.3 불량 블록

- MS-DOS에 경우에는 FAT 테이블에 불량 블록을 기록하여 사용하지 않도록 한다. 사용 도중에 불량 블록을 발견하면 특수한 프로그램을 수행하여 이것을 불량 블록을 기록한 다음에 다시 사용해야 한다.
- SCSI와 같은 보다 복잡한 디스크는 여분 섹터를 유지하여 자동으로 교체하여 사용한다. 즉, 불량 섹터를 발견하면 그 사실을 기록하고, 이 섹터와 여분 섹터를 하나 연관해 놓는다. 그 다음부터 이 섹터를 접근하면 자동으로 연관한 여분 섹터를 사용한다.
- 여분 섹터가 멀리 떨어져 있으면 디스크 스케줄링 알고리즘의 나쁜 영향을 미친다. 따라서 섹터 옮기기(sector slipping) 기법을 사용한다. 트랙마다 여분 섹터를 두어 불량 섹터 이후의 모든 섹터를 그것의 다음 섹터로 사상되도록 사용한다.