File System

파일 시스템

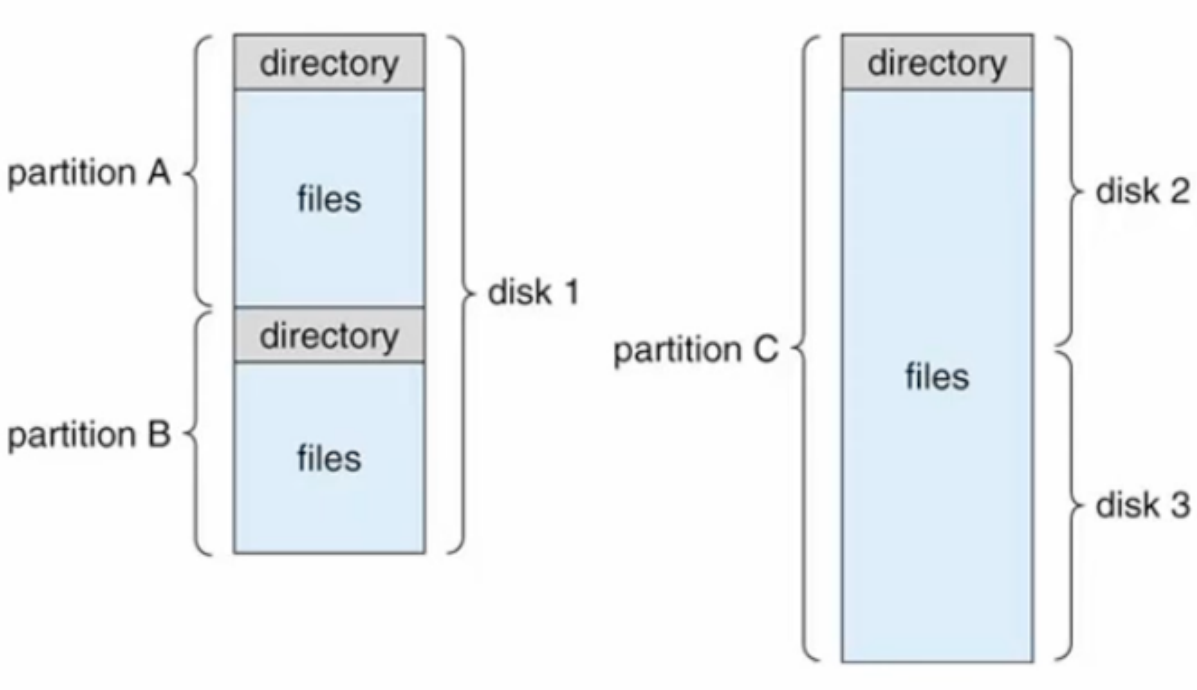
- **저장 공간을 효율적으로 사용하기 위한 조건으로 무엇이 있을까?**
  - **저장한 데이터를 빠르게 찾을 수 있어야 한다.**
  - **새로운 데이터를 빠르고 효율적으로 저장**할 수 있어야 한다.
  - **주어진 공간은 언제나 한정적**이기 때문에 주어진 공간을 최대한 체계적으로 정리해 **빈 공간이 없도록 관리**해야 한다.
  - **데이터 백업 및 복구 기능을 제공해야 한다.**

    ⇒ **운영체제에서 위 조건을 만족시키는 파일 시스템 기능을 제공해 저장 공간의 효율적 사용을 돕는다.**

🤔 **파일 시스템이 없다면, 사용자는 파일이 저장된 모든 주소값 범위(블록)을 전부 기억하고 있어야만 한다.**

- 하지만, 파일 시스템이 있기에 디렉터리 아래 파일을 쉽게 찾을 수 있게 된 것이다.

일반적인 파일 시스템의 구조



- **디스크를 여러 파티션으로 나누고, 각각의 파티션에 파일 시스템이 포함되어 있다.**
  - 여러 파티션으로 나눠진 경우, 파티션 마다 다른 파일 시스템이 운영될 수 있다.

파티션(partition)

- **디스크, 즉 물리적 공간을 논리적으로 나눠놓은 것을 의미한다.**
  - **하나의 디스크를 서로 분리된 여러 개의 디스크처럼 쓸 수 있다.**
  - 혹은, **여러 하드디스크가 있더라도 하나의 파티션으로 만들 수 있다.**
- 하나의 공간을 여러 개로 나눠 관리가 용이해지고, 한 파티션이 손실되더라도 다른 파티션 영역은 보호할 수 있게 된다.
- 보통 **파일 시스템이 포함된 파티션**의 경우 **볼륨(volume)** 이라고 칭한다.

파일

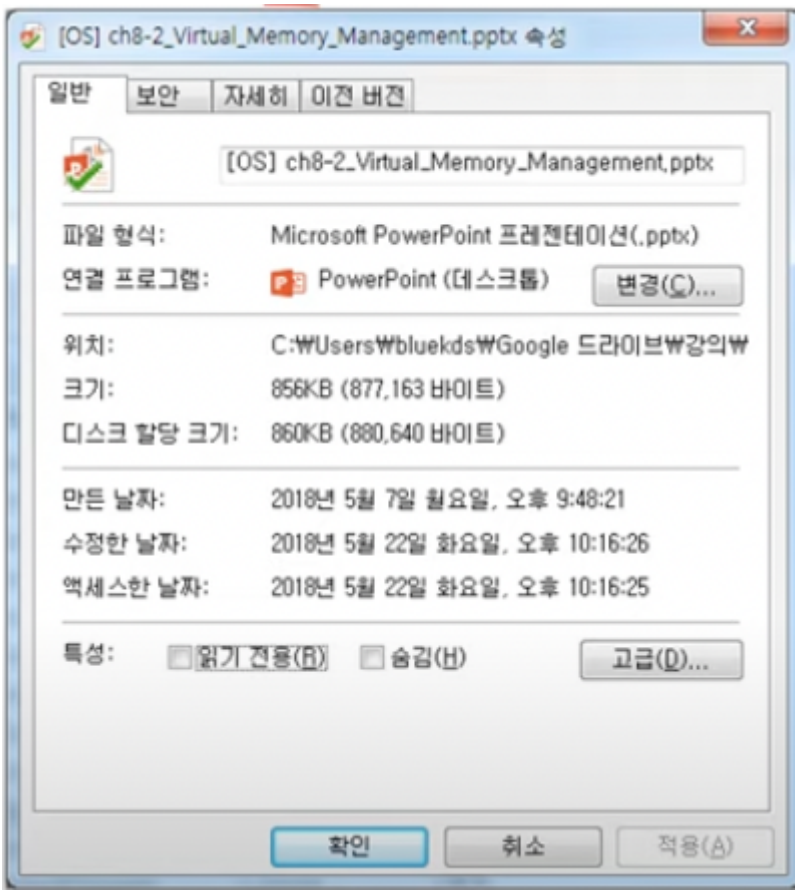
- **보조 기억 장치(디스크)에 저장된, 연관된 정보들의 집합을 의미한다.**
  - **보조 기억 장치 할당의 최소 단위**
  - 연속적인 바이트들의 집합 (물리적 정의)
  - **이름을 가지고 있는 저장 장치상의 논리적 단위**

- **내용에 따른 분류**
  - **프로그램 파일**
    - 소스 프로그램, 오브젝트 프로그램, 실행 파일
  - **데이터 파일**

- **형태에 따른 분류**
  - 읽을 수 있는 문자열로 구성된 텍스트(아스키) 파일
  - 이진 파일 (0, 1)

- 일반적으로 **실제 위치, 크기 등의 속성이 기록된 메타 데이터 영역**과 **실제 데이터가 기록된 데이터 영역**으로 나뉜다.

파일이 가진 속성



- **이름**
  - 사용자들이 이해할 수 있는 형태, 보통 아스키 코드로 붙인다.
- **식별자**
  - 식별을 위해 파일에 할당한 고유 번호로, 사용자가 판독이 불가능하다.
- **유형**
  - 다양한 파일 형식을 지원하는 시스템에 필요하다.
- **저장 위치**
  - 파일이 저장된 장치와, 그 장치 위치를 표시하는 포인터이다.
- **크기**
  - 파일의 현 크기로, 허용 가능한 최대 크기를 포함할 수도 있다.
- **액세스 제어 데이터**
  - 파일 읽기, 쓰기, 실행 등 권한 정보이다.
- **소유자**
  - 파일 최초 생성자를 의미한다.
- **시간, 날짜**
  - 생성 시간, 수정 변경 시간, 최근 사용 시간
- **사용자 식별 정보**
  - 파일을 보호, 보안하고 사용자를 감시하는 데 사용

파일의 연산

- **파일 생성**
  1. 파일 시스템에 있는 공간 탐색
  2. 새로 생성한 파일을 디렉터리에 만들어 파일 이름과 파일 시스템 내의 위치 기록
- **파일 쓰기**
  - 파일 이름과 파일에 기록할 정보를 표시하는 시스템 콜 수행
- **파일 읽기**
- **파일 재설정** (파일 옮기기)

- **파일 삭제**
- **기타**

  → **OS는 파일 연산에 대한 시스템 콜을 제공해야 한다.**

파일 디스크립터(File descriptor)

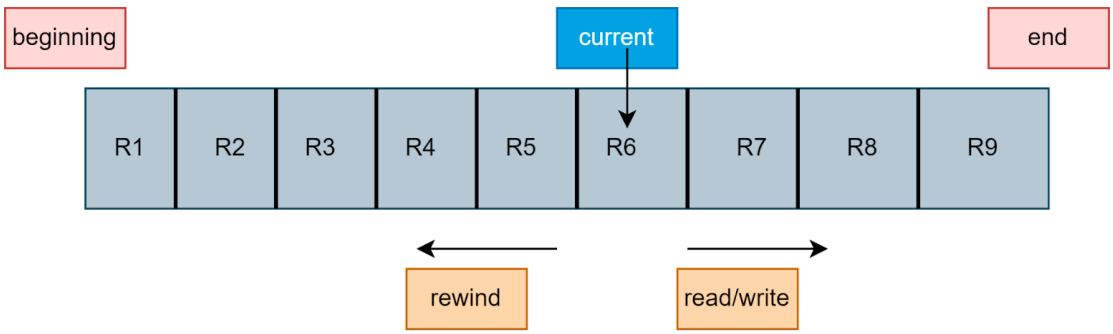
- **운영체제가 필요로 하는 파일에 대한 정보를 가지고 있는 제어 블록**을 의미한다.
- 파일마다 독립적으로 존재하며, 파일 시스템이 관리하므로 사용자가 직접 참조할 수 없다.
- 파일 디스크립터의 내용
  - 파일 이름, 크기, ID
  - 액세스 제어 정보
  - 생성 날짜
  - 저장장치 정보 등

파일에 접근하는 방법들

- 파일은 시스템에 필요한 많은 정보가 포함되어 있고, 컴퓨터 메모리는 실행 중 특정 파일을 요구할 수 있다.

  → **가능한 빠르게 파일에 접근해 정보를 탐색하는 효율적 방법이 필요하다.**

- **순차 접근(Sequential access)**



  - **파일에 레코드(바이트) 단위로 순서대로 접근한다.**

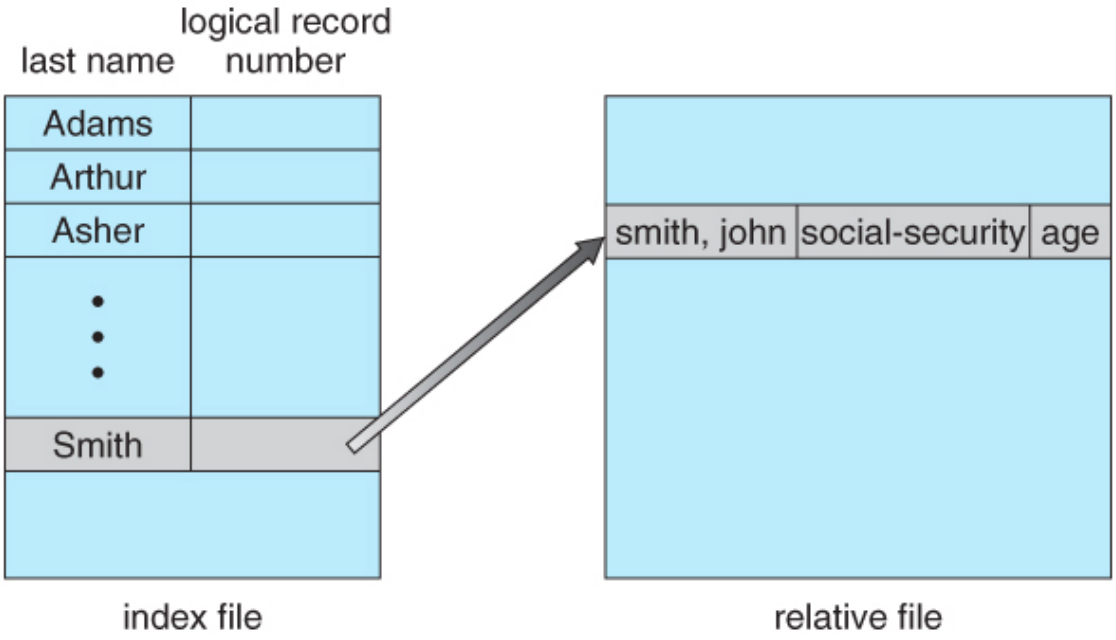
- 현재 포인터가 접근 중인 레코드를 가리키고 있고, **이 방식에서는 어떤 레코드로도 직접 이동할 수 없다.**
  - R8에 접근하려면 R6, R7에 접근해야 한다.
- 연결 리스트와 유사하며, 구현이 매우 간단하다.
- 레코드에 임의로 접근할 수 없기 때문에 효율적이지 않고, 속도가 느리다.

- **직접 접근(Directed access)**

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;
write_next	write cp; cp = cp + 1;

- **모든 블록을 직접 읽거나 쓸 수 있으며, 읽기나 쓰기 순서가 존재하지 않는다.**
  - 블록 14를 읽은 후 블록 57을 읽을 수 있고, 이어서 블록 7에 쓸 수도 있다.
- 대규모 데이터베이스에 유용하다.
- 순차 접근 방법보다 빠르고, 모든 블록을 통과할 필요가 없다.
- 구현 역시 쉽다.

- **인덱스 순차 접근(Indexed access)**



- **직접 액세스를 기반으로 디스크의 물리적 특성에 따라 인덱스를 구성한다.**
- **인덱스를 참조, 원하는 블록을 찾은 후 데이터에 접근한다.**
- 큰 파일도 적은 입출력으로 탐색이 가능하다.

디렉터리

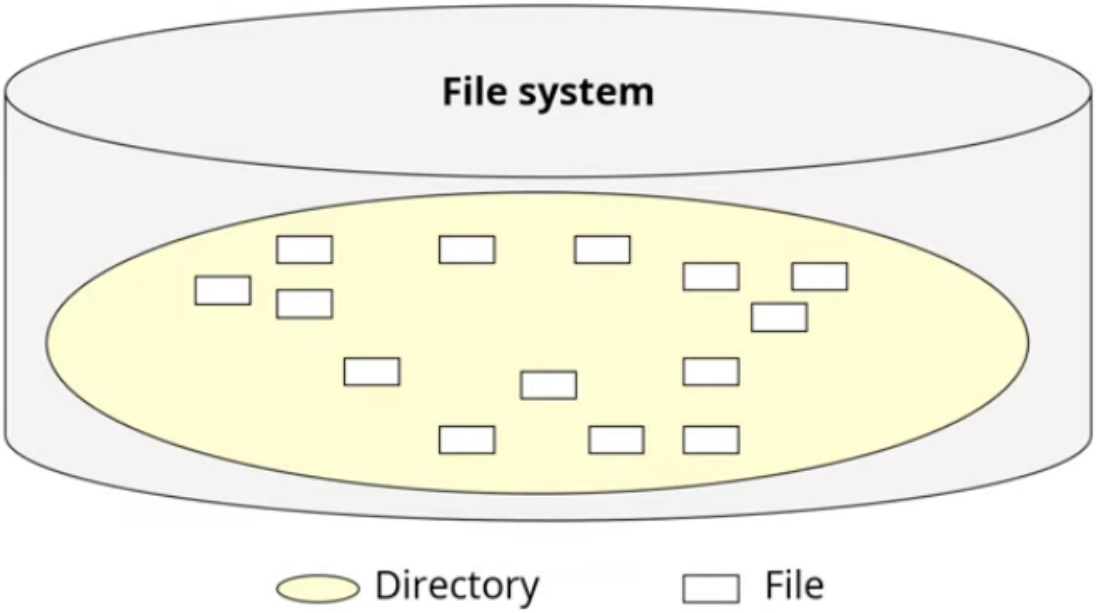
- **디렉터리**는 파일 시스템 내부에 있는 것으로 **효율적 파일 사용을 위해 디스크에 존재하는 파일에 대한 여러 정보를 가지고 있는 특수한 형태의 파일**이다.
  - **각 파일의 위치, 크기등의 정보**를 가지고 있다.

디렉터리의 연산

- 파일 탐색
- 파일 생성
- 파일 삭제
- 파일 열람
- 파일 이름 변경
- 파일 시스템 순회

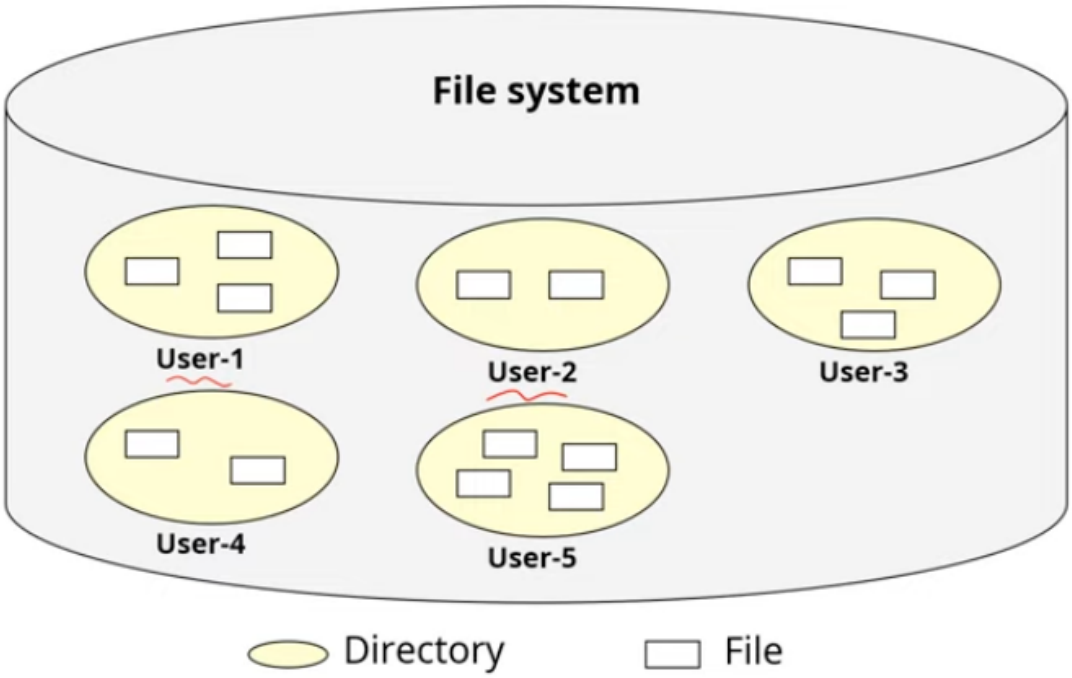
디렉터리 구조

- **1단계 디렉터리 구조**



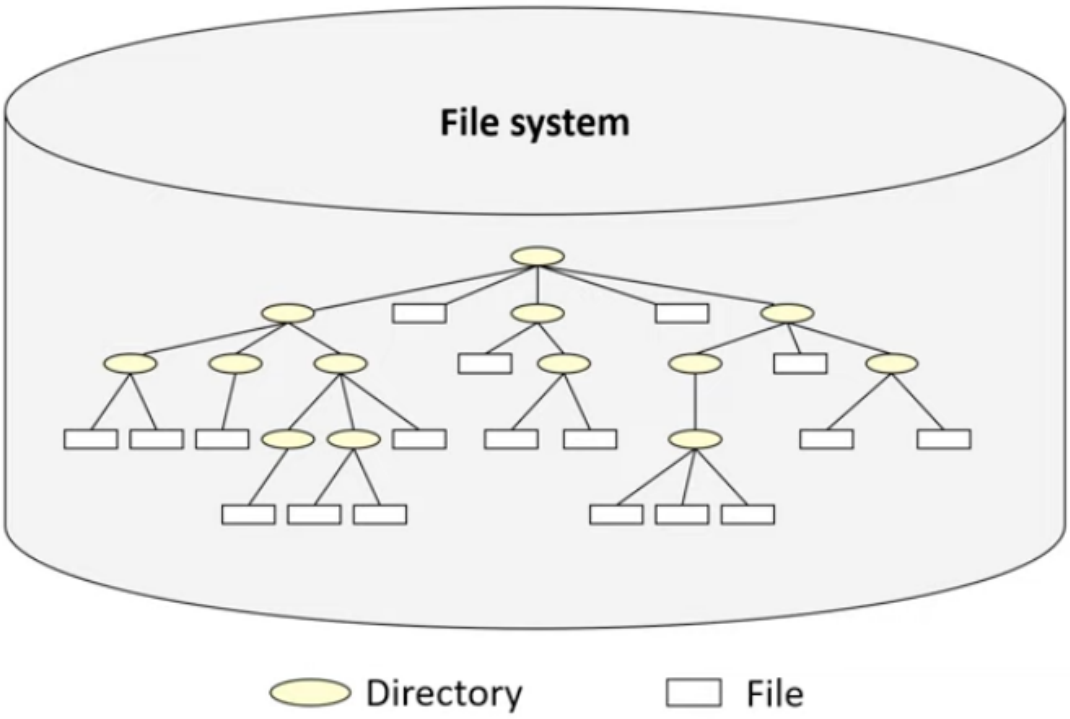
  - **가장 간단한 구조**로, **파일 시스템 내에 하나의 디렉터리만 존재하는 것**을 의미한다.
  - 모든 파일이 동일한 디렉터리에 있어 **유지하고 이해하기 쉽다.**
  - 그러나 **모든 파일이 동일한 디렉터리에 있으므로 모두 고유한 이름을 지녀야하는 단점**이 있다.
    - 네이밍 충돌을 피하기 위해 사용자는 모든 파일 이름을 기억해야만 한다.
    - 만약 동일한 네이밍으로 지정하면, 기존 파일이 덮어씌워질 위험이 있어 파일 보호가 어렵다.
    - **다중 사용자 환경에서 문제가 더욱 커진다.**

- **2레벨 디렉터리 구조**



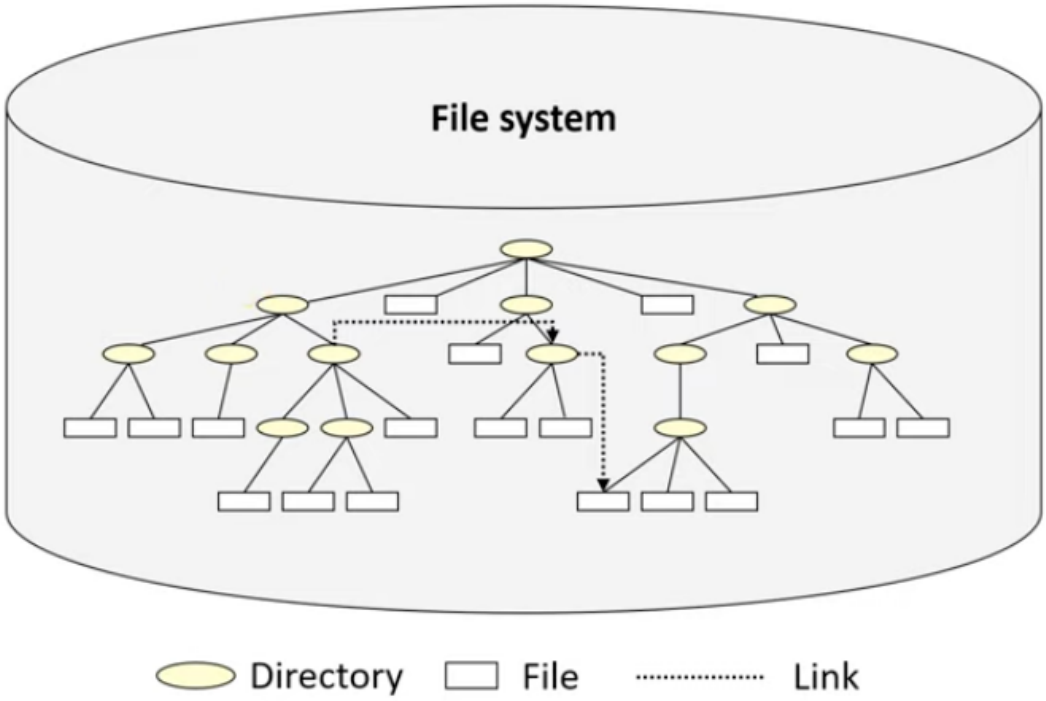
- **사용자마다 하나의 디렉터리를 배정하는 것**을 의미한다.
- **루트는 마스터 파일 디렉터리이고, 아래로 사용자 파일 디렉터리, 그 아래로 파일이 존재한다.**
- 사용자 디렉터리 내부적으로 하위 디렉터리 생성이 불가능해 파일 네이밍 이슈가 발생한다.
- 사용자 간 파일을 공유할 때 **사용자 파일 디렉터리 전체 액세스를 허용**해주어야 하므로 불편하고, 보안도 어렵다.

- **트리(계층적) 디렉터리 구조**



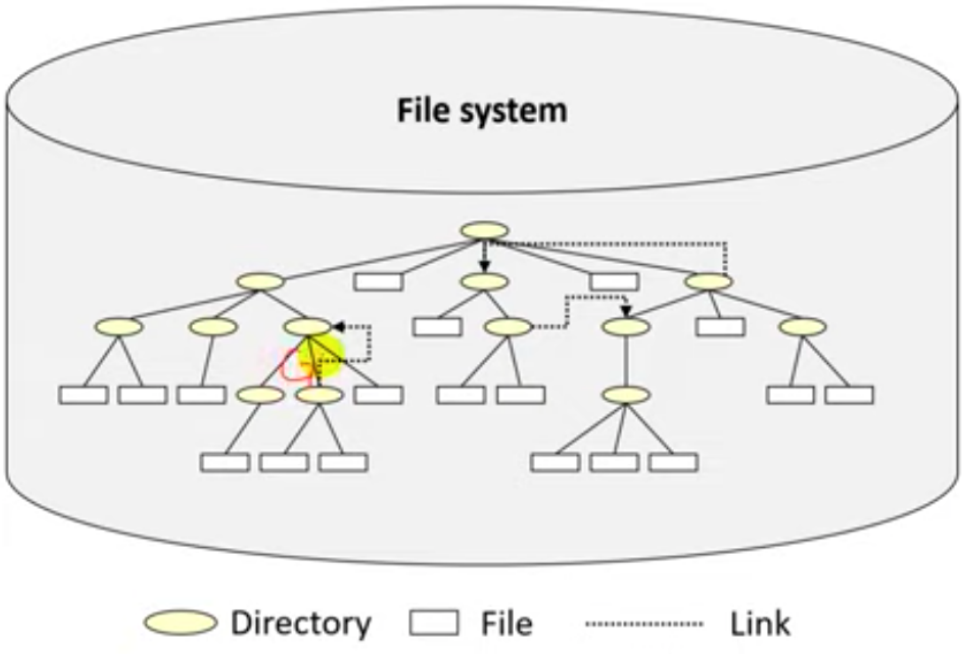
- **트리 형태의 계층적 디렉터리 배정이 가능하다.**
- 루트 디렉터리가 하나 있고, **시스템 내 모든 파일의 경로명이 유일**하다.
- **사용자가 하위 디렉터리를 생성 및 관리할 수 있다.**
  - OS에서 이와 관련된 시스템 콜이 제공되어야만 한다.
- **대부분의 OS에서 사용**하면서, 새로운 개념이 대두되었다.
  - 홈(루트) 디렉터리, 현재 디렉터리 개념
  - 절대 경로(Absolute pathname), 상대 경로(Relative pathname) 개념

- **비순환 그래프 디렉터리 구조**



- **트리 디렉터리 구조를 확장하여 일반화한 것**을 의미한다.
  - 트리 디렉터리 구조에서는 파일 혹은 디렉터리의 공유를 금지한 반면, 비순환 그래프 디렉터리에서는 공유를 허용한다.
- **링크(Link)**의 개념을 사용한다.
  - **다른 파일이나 서브 디렉터리를 가리키는 포인터**를 의미하며, 경로명으로 구현 가능하다.
  - 윈도우에서 사용하는 **바로가기**의 개념이다.
- A → B → A 와 같은 **사이클의 발생을 막는다.**
- **공유 파일이 삭제됐을 때 파일을 가리키는 포인터가 남아있는 고아 포인터 (dangling pointer)가 발생**할 수 있다.

- **일반 그래프 디렉터리 구조**

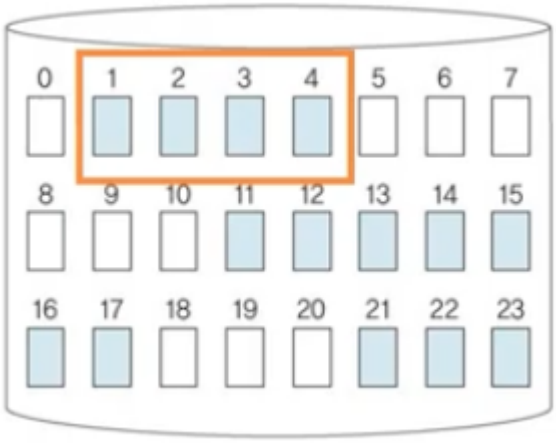


- **비순환 그래프 디렉터리 구조의 일반화로, 사이클을 허용한다.**
- 파일 탐색 시 무한 루프가 발생할 수 있다.
  - 사이클 발생 시 중복 탐색을 피하기 위해 한 번의 사이클에 하나의 디렉터리에 접근할 수 있는 횟수를 제한한다.

파일의 디스크 할당

- 운영체제는 **디스크 공간에 파일을 할당**할 때 어떤 방식으로 해야 **공간적 효율과 접근 속도를 개선**할 수 있는지 늘 고민해야 한다.

연속 할당



- **하나의 파일이 디스크의 연속된 블록에 할당되는 것을 의미한다.**
- **장점**
  - **순차적으로도, 직접적으로도 접근이 가능하기 때문에 파일 접근 시 매우 효율적**이다.
- **단점**
  - **외부 단편화 문제**가 발생한다.
    - 전체 남는 공간이 10개가 넘더라도 연속적이지 않으면 파일 할당이 불가능하다.
  - **파일 공간 크기**를 결정하기 어렵다.
    - 초기 6개의 블록을 할당했으나 파일의 크기가 늘어났을 때 대응하기 어렵다.
    - 이를 방지하기 위해 6개의 블록보다 더 큰 블록을 할당해야 하나 그 크기도 정하기 어렵다.

연결 할당

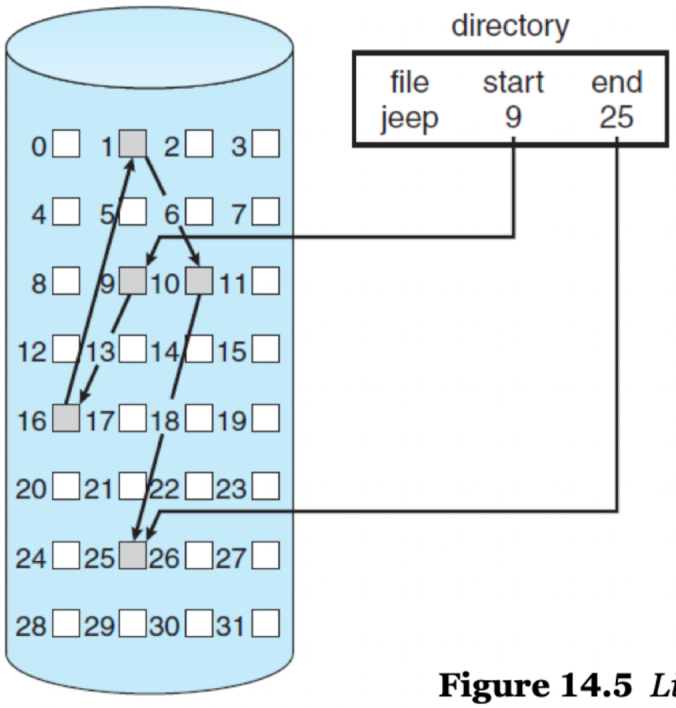
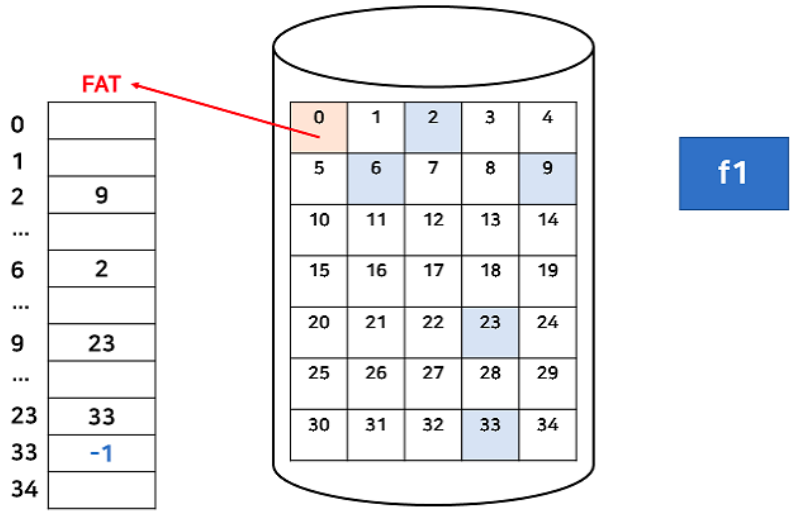


Figure 14.5 *Linked allocation of disk space.*

- **파일이 저장된 블록들을 연결 리스트로 연결하는 것**을 의미한다.
  - 즉, 비연속 할당이 가능함을 의미한다.
  - **디렉터리 항목은 각 파일에 대한 첫 번째(시작) 블록에 대한 포인터를 가진다.**
- **장점**
  - **구현이 쉽고**, 외부 단편화 문제가 발생하지 않아 **저장 공간 낭비가 발생하지 않는다.**
- **단점**
  - 파일 블록이 모두 흩어져 있으므로 시작 블록 번호만으로는 **직접 접근이 불가능**하다.
  - 포인터 저장을 위한 **별도 공간이 필요**하다.
  - **사용자가 포인터를 실수로 건드리는 경우 신뢰성 문제가 발생할 수 있다.**
    - 포인터가 완전히 사라진다면, 해당 파일은 사용할 수 없게 된다.

연결 할당의 변종 → File Allocation Table(FAT)



- 사용자가 **해당 블록의 포인터를 실수로 지우지 않게 하고, 블록 접근을 빠르게 하기 위해 포인터를 모아 놓은 테이블**을 의미한다.
  - 한 블록에 저장된다.
- **각 블록의 시작 부분에 다음 블록의 번호를 기록한다.**
  - 0번 블록에 저장된 FAT을 보면, 테이블 인덱스는 전체 디스크의 블록 번호이며 각 인덱스마다 다음 블록 번호를 저장하고 있다.
- **기존 연결 할당의 문제점 대부분을 해결할 수 있다.**
  - FAT을 한 번만 읽으면 직접 접근이 가능하다.
  - 중간 블록에 문제가 생겨도 그 다음 블록을 여전히 읽을 수 있다.
- **매우 중요한 정보이므로, 손실 시 복구를 위해 이중으로 저장한다.**
- MS-DOS, Windows 등에 사용된다.

인덱스 할당(Indexed Allocation)

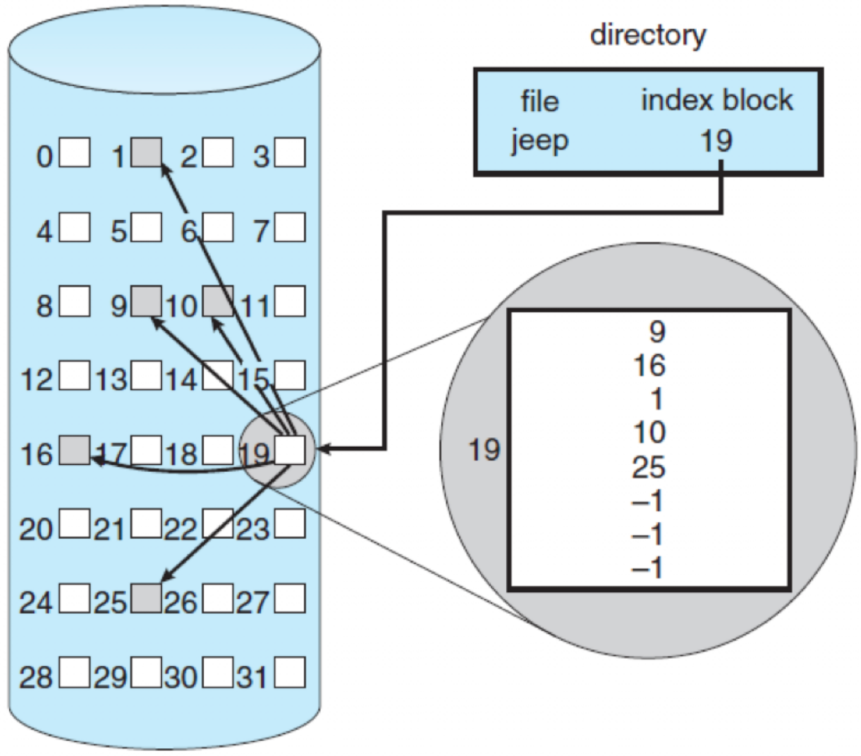


Figure 14.7 *Indexed allocation of disk space.*

- **각 파일에 포인터가 순서대로 저장된 인덱스 블록을 할당**한다.
  - **디렉터리 항목은 각 파일에 대한 인덱스 블록 번호를 저장한다.**
- **장점**
  - **외부 단편화가 발생하지 않는다.**
  - **직접 접근에 효율적**이고, **순차 접근 역시 가능**하다.
- **단점**
  - 인덱스 블록 할당에 따른 **저장 공간의 손실이 발생**한다.
  - **인덱스 블록의 크기에 따라 파일 최대 크기가 제한**된다.

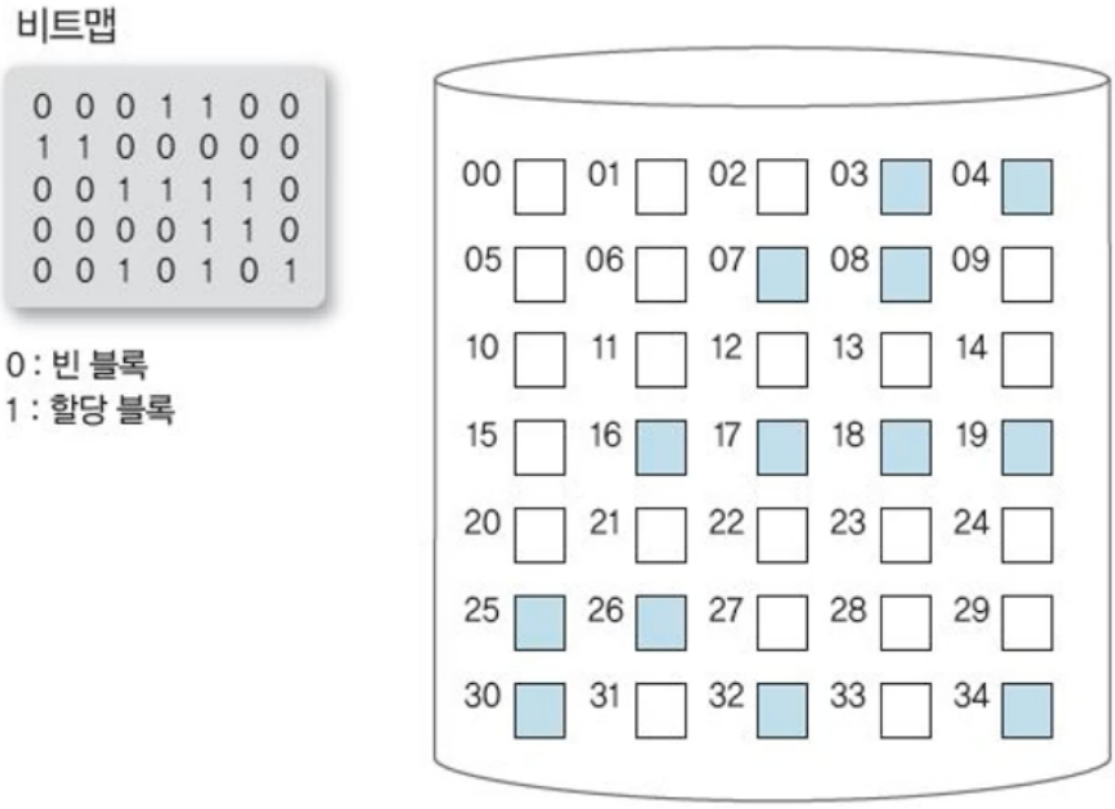
    → 인덱스 블록이 단 10개면, 파일 크기도 이에 맞춰져야 한다.
  - 순차 접근에 효율적인 방식은 아니다.

    → 블록 접근 후 다시 인덱스 블록으로 돌아와 다음 번호를 확인해야 한다.
- Unix 등에 사용된다.

빈 공간 관리

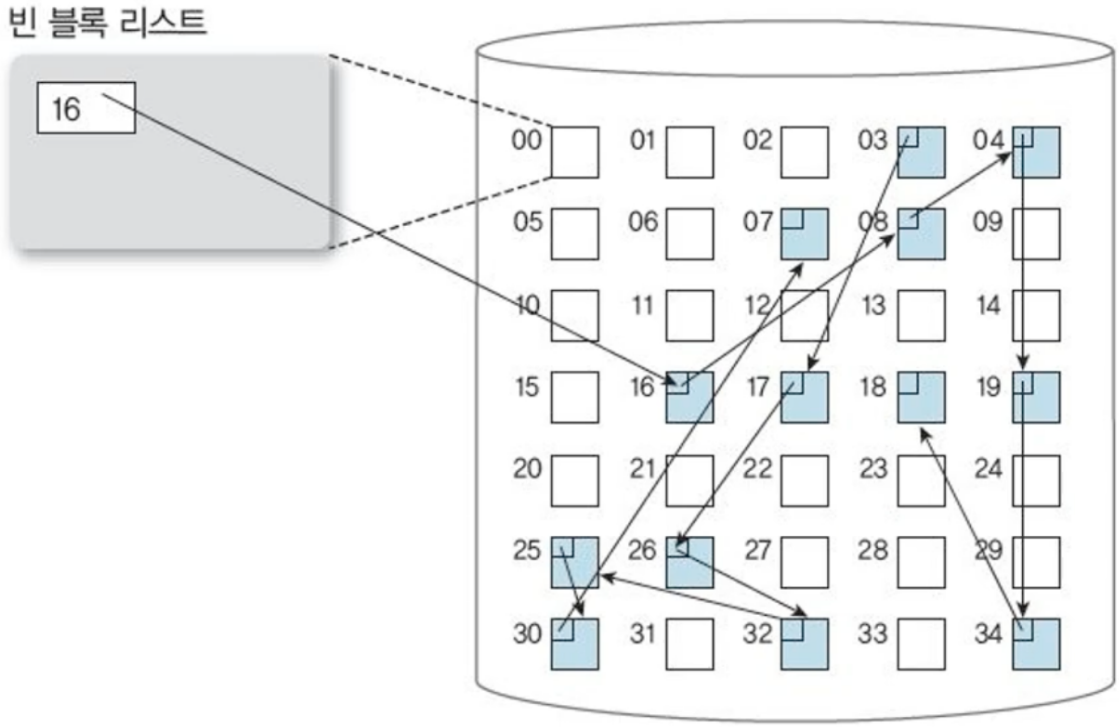
- 디스크 공간은 기본적으로 제한되어 있기 때문에 **삭제된 파일이 차지하던 공간을 새로운 파일들이 사용할 수 있도록 해야한다.**
- **디스크는 빈 공간을 관리하고 있으며**, 관리하는 방법으로 리스트를 사용할 수도 있다.
  - **새 파일을 만들 때 이 관리 공간에서 새 파일을 위한 공간을 할당받는다.**
  - **할당된 공간은 빈 공간에서 삭제된다.**

비트 벡터(Bit vector)



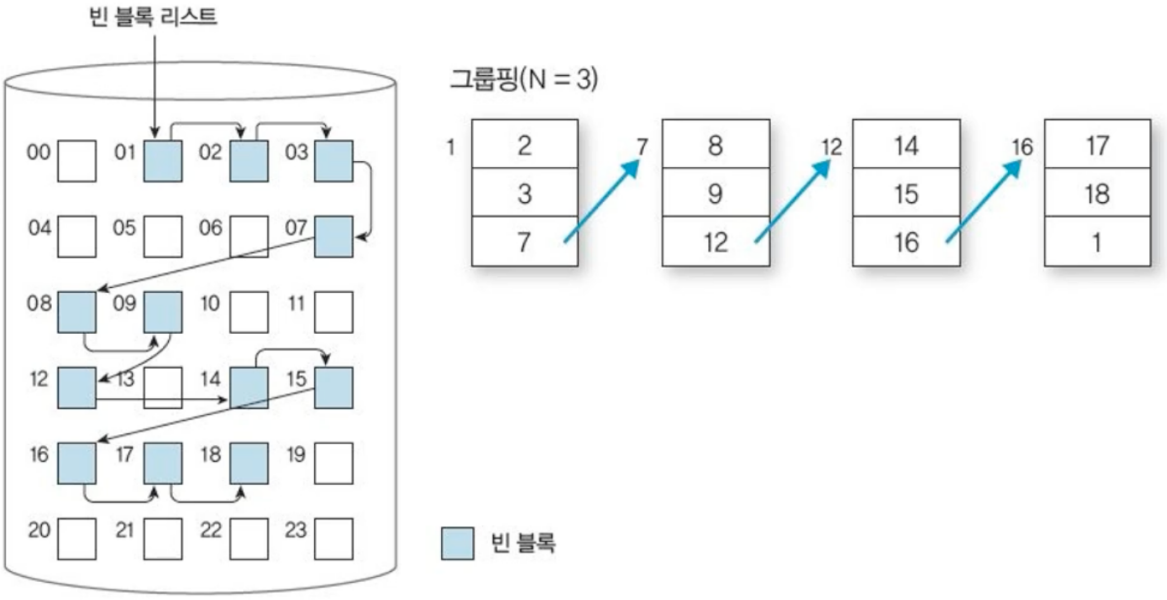
- **시스템 내 모든 블록들에 대한 사용 여부**를 **1 비트 플래그**로 표시한다.
  - 간단하고 효율적이다.
  - 그러나, **비트 벡터 전체를 메모리에 보관**해야 한다.
    - **대형 시스템에 부적합**하고, 마이크로 컴퓨터에 적합하다.

연결 리스트



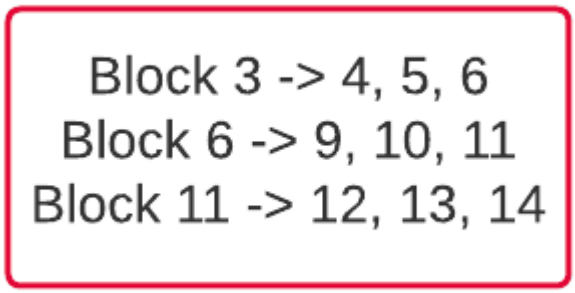
- **빈 블록**을 **연결 리스트로 연결**한다.
- **특정 위치의 빈 공간을 찾아가기 위해 많은 링크를 거쳐야하므로 비효율적**이다.

그루핑



- **n개의 빈 블록을 그룹으로 묶고, 그룹 단위별로 연결 리스트를 사용**한다.
  - 맨 처음 빈 블록에 n개 블록에 대한 주소를 기입한다.
- **단순 연결리스트보다 빠르게 빈 공간을 찾을 수 있다.**

카운팅



- **연속된 빈 블록들 중 첫 블록의 주소와 연속된 블록의 수를 테이블로 유지**한다.
  - **"몇 번째 블록 뒤로는 몇 개의 자유 공간이 있다."**
  - **연속 할당 시스템에 유리한 기법**이다.
    - 만약 연속적으로 할당된 메모리가 단 하나밖에 없다면 테이블 공간을 낭비하게 된다.

파일 보호

- **파일에 대한 부적절한 접근 방지가 필요**하다.
  - 다중 사용자 시스템에서 필요성이 더욱 부각된다.
- **파일은 허용하는 접근 권한을 제한하여 보호**할 수 있다.
  - 허용할 수 있는 접근 권한 → **읽기, 쓰기, 실행, 추가, 삭제**

파일 보호 기법

- 시스템 사이즈 혹은 응용 분야에 따라 다를 수 있지만, 일반적으로 아래의 방법들을 사용한다.

- **패스워드 기법**
  - 각 파일들에 **패스워드를 부여**하는 가장 간단한 기법이다.
  - 그러나, 사용자들이 파일 각각에 대해 패스워드를 기억해야 하기 때문에 **비현실적**이다.

- **접근 행렬 기법**

Object / Domain	F1	F2	F3	F4	F5
D1	R	R		RW	
D2	RW			RA	
D3		R		RW	X
D4	RW		X		

  - **범위(domain)와 개체(object) 사이의 접근 권한을 명시하는 기법**이다.
    - 범위 → 사용자 혹은 그룹
    - 개체 → 파일

References

- https://www.youtube.com/watch?v=bRwjKvmeyZQ&list=PLBrGAFAIyf5rby7QylRc6JxU5lzQ9c4tN&index=39
- https://www.codingninjas.com/codestudio/library/file-access-methods
- https://velog.io/@codemcd/운영체제OS-18.-파일-할당