

**CSE235**

**데이터베이스 시스템**

**(Database Systems)**

**Lecture 02: File Structure**

**담당교수: 전강욱(컴퓨터공학부)**

**kw.chon@koreatech.ac.kr**

\* 본 강의 자료는 정익사에서 제공한 화일구조(이석호 저) 관련 강의자료를 편집하였음.

# 개요

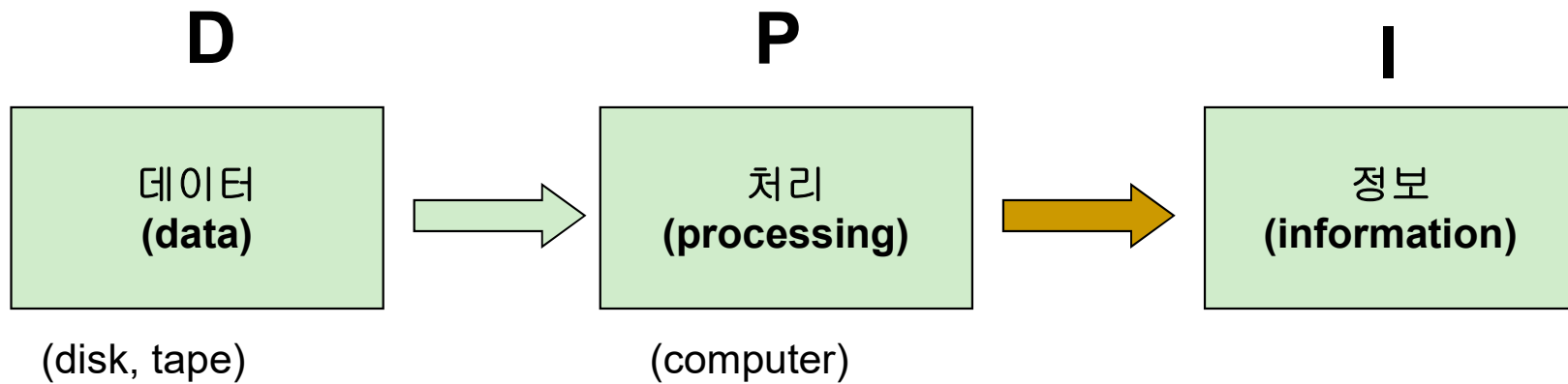
- 파일의 기본 개념
- 파일 저장 장치
- 파일의 입출력 제어
- 순차 파일

# 파일의 입출력 제어 환경

- 사용자 (프로그램) 관점: 논리적 구조의 파일을 사용함
  - 사용자 입장에서 파일은 정형화된 구조(예: 레코드의 집합, 프로그램 파일)를 가짐
  - 사용자는 실제로 파일이 디스크의 어디에 어떤 형태로 저장되어 있는지 확인하기 어렵고 확인할 필요가 없음
- 운영체제(with 파일 관리자) 관점: 물리적 구조의 파일을 사용함
  - 파일이 어떠한 논리적 구조를 가지고 있는지는 중요치 않음
  - 주어진 파일을 실제로 어떻게 찾고(디렉토리 관리), 어떻게 저장하고, 어떻게 해석하는지에 대한 책임을 가짐
  - Channel, Device Driver 등을 통하여 실제 I/O를 수행하는 주체이기도 하며, 수행된 I/O의 결과로 사용자에게 논리적 관점의 파일 구조를 전달함

# 파일의 종류

- 정보(Information) ≠ 데이터 (Data)



$$I = P(D)$$

# 주요 용어

- 데이터 필드(field), 속성 (attribute), 데이터 항목 (item)
  - 이름을 가진 논리적 데이터의 최소 단위 (예: 나이 필드, 이름 속성)
  - 특정 객체(object, entity)의 한 성질의 값
  - 테이블(table)의 attribute로 이해 할 수 있음
- 레코드 타입 (record type)
  - 논리적으로 서로 연관된 하나 이상의 데이터 필드(항목)들의 집합
  - 엔티티 타입 (예: 학생 = {이름 필드, 학번 필드, 성별 필드, ...})
- 레코드 인스턴스 (record instance)
  - 레코드 타입의 각 필드에 따라 실제 값이 들어가 어떤 특정 객체를 나타내는 것
  - 일반적으로 레코드(record)라고 함 (예: {홍길동, 2005012345, 남, ...})
  - “화일 구조”에서는 record occurrence라 정의하였음

# 주요 용어 (계속)

## ■ 파일 (file)

- (보조)기억장치에 저장된 같은 종류의 레코드 집합 (set of records)
- 하나의 응용 목적을 위해 함께 저장된 레코드 (예 : 급여, 인사, 재고, 재무, 회계 등)

## ■ 데이터의 집합을 (일반적으로는) 왜 파일로 구성하는가?

- 주기억 장치(main memory)에 전부 적재하기에는 데이터 양이 너무 많다.
- 프로그램은 특정 시간에 데이터 집합의 일부만을 접근한다.
- 자원 이용의 효율성을 위하여, 데이터 전부를 주기억 장치에 한꺼번에 저장시킬 필요가 없다.
- 데이터를 특정 프로그램의 수행과 독립적으로 보관시켜 데이터의 독립성 (independency) 유지하기 위함이다. (여러 응용 프로그램이 공용하기 쉬움)

# 기능에 따른 파일의 분류

## ■ 기능에 따른 파일의 종류

- 마스터 파일 (master file)
- 트랜잭션 파일 (transaction file)
- 작업 파일 (work file)
- 프로그램 파일 (program file)
- 기타: 보고서 파일(report file), 텍스트 파일(text file)

## ■ 마스터 파일 (mater file)

- 어느 한 시점에서 조직체의 업무에 관한 정적인 면을 나타내는 데이터의 집합
  - 제조 회사의 예: 급여 마스터 파일, 고객 마스터 파일, 인사 마스터 파일, 재고 마스터 파일, 자재 요청 마스터 파일
- 현재 시점의 정보를 표현하는 파일(레코드의 집합)로 볼 수 있음
- 비교적 영구적(permanent)인 데이터를 포함하기도 함 (특수 예: 사전)

# 기능에 따른 파일의 분류 (계속)

## ■ 트랜잭션 파일 (transaction file)

- 마스터 파일의 변경 내용을 모아 둔 파일 혹은 마스터 파일을 변경(update)하기 위한 데이터 파일
  - 새로운 레코드의 삽입(insert)
  - 기존 레코드의 삭제(delete)
  - 기존 레코드의 내용 수정 (modify, replace)

## ■ 트랜잭션 이란?

- 논리적인 작업 단위 (예: 입학 처리 트랜잭션, 이자율 계산 트랜잭션) 일련의 조회 및 변경 연산으로 구성됨
- 하나의 건수로 처리되어야 하며 분리될 수 없는 단일 작업 (All done or not done)



# 기능에 따른 파일의 분류 (계속)

## ■ 작업 파일 (work file)

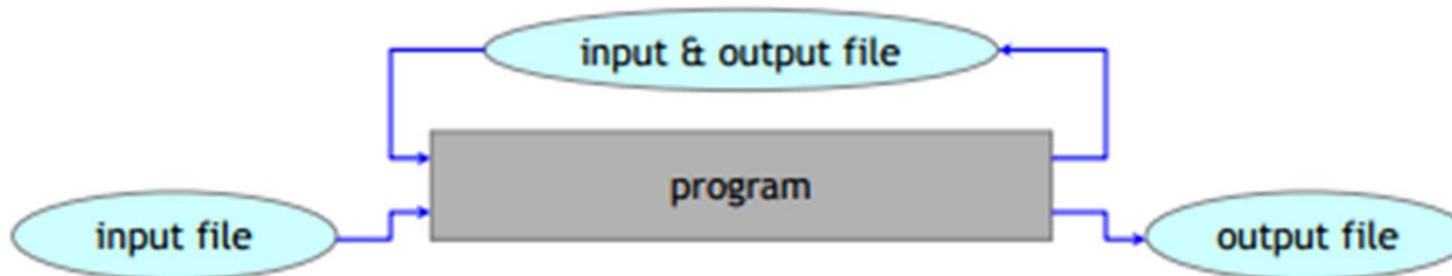
- 한 프로그램에서 생성한 (출력) 데이터를 다른 프로그램의 (입력) 데이터로 사용하기 위하여 임시로 만든 파일
- 임시 파일(temporary file)이라고 하며, 프로그래밍 과정에서 자연스럽게 자주 사용하는 기법임
- 작업 파일의 예
  - 시스템이 자동으로 만드는 작업 파일: Merge Sort 등의 정렬을 위한 파일
  - 프로그램이 만드는 작업 파일: 수강 신청 변경 파일

## ■ 프로그램 파일 (program file)

- 데이터를 처리하기 위한 명령어들을 저장하고 있는 파일
- 고급 언어(C, Java), 어셈블리어, 데이터베이스 질의어 (SQL 언어) 등

# 사용 목적에 따른 파일의 분류

- 입력 파일 (input file): 프로그램의 입력으로 사용되는 파일로서, 일반적으로 프로그램이 읽기(read)만을 하는 파일
- 출력 파일 (output file): 프로그램이 출력에 사용하는 파일로서, 일반적으로 프로그램이 쓰기(write)만을 하는 파일
- 입출력 파일(input & output file): 프로그램에서 입력 및 출력의 두 목적으로 사용하는 파일로서, 읽기 연산 및 쓰기 연산이 모두 적용되는 파일



# 파일의 기본 연산

- 생성 `fopen(..., O_CREATE)`
- 기록 `write(...), fprintf(...), ...`
  - 레코드 내용의 갱신 (update)
  - 새로운 레코드 삽입 (insert)
  - 기존 레코드 삭제 (delete)
- 판독 `read(...), fscanf(...), ...`
- 삭제 `unlink(...)`
- 개방과 폐쇄 `fopen(...), fclose(...), open(...), close(...), ...`
  - 버퍼의 할당과 반환

# 파일 구조 선정 요소

- 파일 구조 선정의 주요 자원 비교
  - 주기억 장치 (main memory)
    - 특정 데이터를 찾기 위한 최대 비교 연산 횟수 등으로 평가
    - 데이터 접근 시간은 모두 일정한 것으로 가정 (random access)
  - 보조 저장 장치 (secondary storage) (파일 구조 선정의 주요 고려 자원)
    - 데이터 액세스 시간이 주기억 장치에 비해 매우 긴 특성을 가짐
    - 일반적으로 보조 저장 장치의 접근 횟수가 프로그램 성능의 평가 요소가 됨
- 파일 구조 선정의 주요 요소
  - 가변성 (volatility)
  - 활동성 (activity)
  - 사용 빈도수 (frequency of use)
  - 응답 시간 (response time)
  - 파일 크기 (file size)
  - 파일 접근 유형

# 파일 구조 선정 요소 (계속)

## ■ 가변성 (volatility)

### □ 파일의 성격

- 내용이 변하지 않는 정적 파일 (주로 보관 및 검색이 목적인 과거의 기록)
- 내용이 자주 변하는 동적 파일 (자은 변경이 있는 현재의 상황 데이터)

### □ 가변성

- 전체 레코드 수에 대해 추가 혹은 삭제되는 레코드 수
- 가변성이 높은(추가 및 삭제가 많은) 동적 파일은 빠른 접근과 갱신이 필요

## ■ 활동성 (activity)

- 주어진 기간 동안에 파일의 총 레코드 수에 대해 액세스가 일어난 레코드 수의 비율
- 가변성이 Update 비율인 반면, 활동성은 액세스 비율을 나타냄
- 활동성이 높으면 (대부분의 레코드가 액세스 된다면) 순차 파일 구조가 유리

# 파일 구조 선정 요소 (계속)

- 사용 빈도수 (frequency of use)
  - 파일의 사용 빈도수
    - 일정 기간 동안의 파일 사용 빈도수 (파일이 얼마나 사용되는지를 나타냄)
    - 가변성과 활동성에 밀접히 관련 (가변성/활동성이 높으면 일반적으로 빈도수가 높음)
  - 사용 빈도수와 파일 구조
    - 빈도수가 낮으면 순차 파일 구조가 유리 (sequential access)
    - 빈도수가 높으면 임의 접근 구조가 유리 (random access)
- 응답 시간 (response time)
  - 검색이나 갱신에 대해 요구하는 지연 시간
  - 빠른 응답 시간의 요구 조건에는 임의 접근 구조를 선택
  - 응답 시간이 strict하지 않은 경우에는 순차 파일 구조를 선택
  - 실시간(real-time) 조건이 주어지는 경우, 응답 시간은 매우 중요한 설계 요소가 됨

# 파일 구조 선정 요소 (계속)

## ■ 파일 크기 (file size)

- 레코드 수와 각 레코드 길이가 파일의 크기를 결정 (trivial)
- 시간이 지남에 따라 파일 크기가 성장 (레코드 길이 확장, 레코드 개수 증가)
- 성장을 유연하게 수용할 수 있는 구조의 채택이 필요 (free space 유지 등)

## ■ 파일 접근 유형

- 파일에 적용되는 연산의 유형과 접근 형식에 따라 파일 구조를 결정
- 접근 유형의 예제
  - 판독 위주 접근 vs. 갱신 위주 접근
  - 순차 접근 위주 vs. 임의 접근 위주

# 파일 저장 장치의 특성

- 저장 장치 (storage device)
  - 저장 매체(medium) + 매체에 데이터를 저장하고 검색하기 위한 장치(device)
  - 예: physical disk + disk controller
- 저장 매체 (storage media)
  - 데이터를 저장하는 물리적 재료 (physical disk)
  - 소멸성(volatile) vs. 비소멸성(nonvolatile)
    - disk vs. memory
- 접근 장치 (access device or access mechanism)
  - 데이터를 판독(read)하거나 기록(write)하는 장치 (혹은 방법)
  - 예: disk controller(driver), tape driver



# 저장 장치

- 1차 저장장치(primary storage)
  - 주기억 장치 (main memory)
    - 데이터 액세스 시간이 일정하고 (디스크 등에 비해) 매우 빠름
    - 프로그램/데이터 처리를 위한 작업 공간
  - 캐시 메모리 (cache memory)
    - 주기억 장치의 성능을 향상 시키기 위한 목적으로, CPU Chipset 내에 구현되기도 함
    - 주기억 장치 보다 빠른 액세스 속도를 가지며, 최근에는 캐시 메모리 자체도 Primary(L1 Cache),
      - Secondary(L2 Cache)로 구분되기도 함
- 2차 저장장치(secondary storage)
  - 자기 디스크 (magnetic disk)
    - 데이터 액세스 시간이 일정하지 않고 (메모리에 비해) 액세스 시간이 느림
    - 용량이 크고 싸서 주로 파일의 저장에 사용됨
    - 저장된 데이터는 주기억 장치를 거쳐(메모리에 올라 와) CPU에 의해처리됨
  - 광디스크(optical disk), 자기 테이프(magnetic tape) 등

# 저장 장치의 유형

## ■ 캐시 메모리 (cache memory)

- 가장 빠르고 가장 비싼 저장 장치
- SRAM(Static Random Access Memory)을 주로 사용함
- 일부 캐시의 경우는 CPU Chipset 내에 구현되어 고성능을 보장함
- 현재 수십 KB ~ 수 MB 수준의 용량이 제공됨

## ■ 주기억 장치 (main memory)

- 프로그램 실행과 이에 필요한 데이터 유지 공간
  - 프로그램이 로딩되는 공간이며, 프로그램이 사용하는 데이터가 로딩되는 공간이기도 함
- DRAM(Dynamic Random Access Memory)을 주로 사용함
- 저용량, 소멸성(데이터 저장에는 부적합)
- 현재 수백 MB ~ 수십 GB 수준의 용량이 제공됨

# 저장 장치의 유형 (계속)

## ■ 플래시 메모리 (flash memory)

- 고밀도, 고성능 메모리로서 비소멸성(nonvolatile)의 특성을 가짐
- 주기억 장치와 비슷한 액세스 속도를 보임
- 핸드폰, MP3 Player, PDA 등에서 디스크/메모리 역할을 수행함
- 현재 수 MB ~ 수 GB 수준의 용량이 제공됨

## ■ 자기 디스크 (magnetic disk)

- 데이터(파일) 저장 장치의 주된 매체로 사용되고 있음
- 데이터 처리와 기록을 위해서는 주기억 장치를 거쳐야 함 (버퍼를 통해서 이루어짐)
- 고용량이며, 비소멸성의 특징을 가짐
- 현재 수십 GB ~ 수십 TB 수준의 용량이 제공됨 (disk array를 구성하여 사용)

# 저장 장치의 유형 (계속)

## ■ 광 디스크 (optical disk)

- 광학적으로 저장, 레이저로 판독
- 용량이 크고, 보존 기간이 긴 특성을 가짐
- CD-ROM, CD-R, CD-RW, DVD(digital video disk), DVD-ROM

## ■ 자기 테이프 (magnetic tape)

- 데이터의 백업과 보존을 위한 저장 매체
- 순차 접근이 적용되는 대표적인 저장 장치
- 테이프 쥬크 박스 (대용량 데이터 저장 가능)
- 은행, 통신 등에서는 주기적인 백업을 위하여 자기 테이프를 널리 활용함

# 저장 장치의 유형 (계속)

- 상위 층으로 갈수록 비트당 가격이 높아지고, 용량이 감소



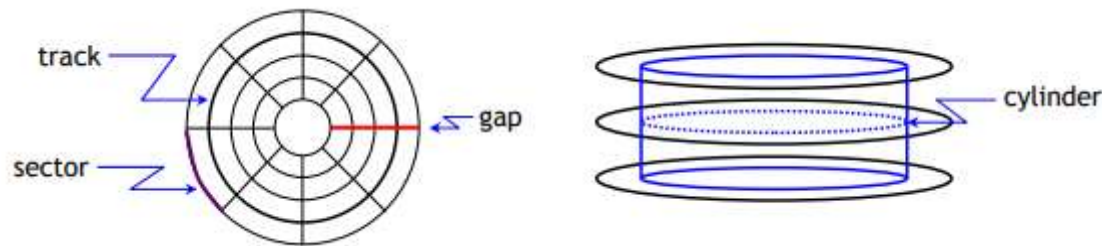
# 하드 디스크 (Hard Disk)

## ■ 자기 디스크의 종류

- 하드 디스크(hard disk): 1995년 IBM에서 개발 – 초기 5MB
- 유연한 디스크(flexible disk): floppy disk, diskette (~1.44 MB)

## ■ 디스크의 구성

- 갭(gap): 자기화 되지 않은 영역으로서 섹터를 구분하는 구분자 역할을 함
- 트랙(track): 갭(gap)으로 분리된 (하나의 원을 구성하는) 섹터들로 구성
- 섹터(sector): 기록과 판독 작업의 최소 단위 (block 크기에 대응됨)
- 실린더: 여러 디스크로 디스크 팩(disk pack)이 형성되었을 때, 지름이 같은 트랙의 모음



# 하드 디스크 (Hard Disk) (계속)

- 데이터 액세스 시간의 구분

- 디스크에서 데이터를 읽거나, 디스크에 데이터를 쓰기 위해서는
  - 1) 원하는 실린더(또는 트랙)을 찾아야 하고 (디스크 헤드의 좌우 이동),
  - 2) 원하는 섹터를 찾아야 하며 (디스크회전이동),
  - 3) 원하는 양 만큼 데이터를 전송해야 한다.
- Seek time(= s): 원하는 데이터가 있는 실린더(혹은 트랙)에 디스크 헤드를 위치 시키는데 걸리는 시간
- Rotational latency( = r): 실린더를 찾은 후, 원하는 섹터에 헤드를 위치 시키기 위해 디스크가 회전하는 시간
- Transfer time(= t): 섹터(와 갭)들이 헤드 밑을 회전하며 데이터를 전송하는 시간
  - Transfer rate: 초당 데이터가 전송되는 속도 (MBps)

- 파일 시스템(혹은 저장 시스템)의 설계자는 seek time과 rotational time을 최소화하도록 데이터 구조의 설계를 진행하여야 함 (clustering, locality, ...)

# 플로피 디스크 (Floppy Disk)

- 유연한 디스크(flexible) 저장 장치
  - 1970년경 IBM에서 개발
  - 직경: 5¼ inch, 3½ inch
  - 회전 속도: 360 rpm (c.f., 5400 ~ 10K rpm in hard disks)
  - 과거(수년 전): 네트워크가 연결되지 않은 컴퓨터 간의 파일 이동을 위한 가장 유용한 장치로 각광 받았음 (한 장에 1K원대)
  - 현재?: 용량 및 액세스 속도 문제로 인하여 CD-ROM 및 USB Driver에 그 자리를 거의 빼앗긴 상태임 (열 장에 수K원대)



# 블로킹 (Blocking)

## ■ 블록(block)

- 데이터 전송의 단위 (디스크와 메모리 사이의 최소 전송 단위)
- 트랙 길이 =  $b \times B$ 
  - $b$  = number of blocks in a track
  - $B$  = block size (블록 크기)
- 블록 크기  $\leq$  트랙 길이

## ■ 블록 크기

- 과거: 512 bytes, 1024 bytes, 2048 bytes
- 현재: 4096 bytes, 8192 bytes, more ...
- 단, 너무 크면 불필요한 데이터 전송 및 메모리 효율성 저하가 발생함
  - 예를 들어, 블록 크기가 8KB이면, 1 byte를 저장하려 해도 8KB가 필요할 수 있고, 1 byte만 전송하려 해도 8KB를 전송해야 한다.

# 블로킹 (Blocking) (계속)

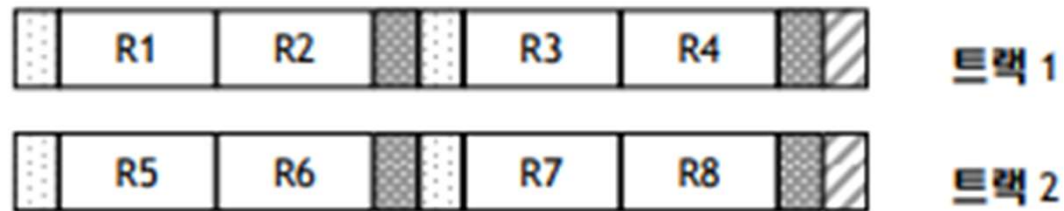
## ■ 블로킹 (blocking)

- 기억 공간과 I/O 효율을 위하여, 몇 개의 논리적 레코드를 하나의 물리적 레코드(블록)에 저장시키는 것
  - 즉, 레코드들을 블록에 저장시키는 것을 블로킹이라 함
- 블로킹 인수 (blocking factor)  $Bf = \lfloor B/R \rfloor$ 
  - $B$  = 블록 크기(block size)
  - $R$  = 레코드 크기 (fixed or variable)
- 블로킹은 I/O 시간을 감소시키는 장점(bulk read & write)이 있는 반면에, fragmentation에 의해 저장 공간이 감소하는 단점이 있음

## ■ 블로킹 방법

- 고정 길이 블로킹 (for fixed length records)
- 신장된(spanning) 가변 길이 블로킹: 가변 길이 레코드를 지원하되, 한 레코드가 인접한 블록에 걸쳐서 저장
- 비신장된 가변 길이 블로킹: 가변 길이 레코드를 지원하되, 한 레코드는 반드시 하나의 블록에 저장

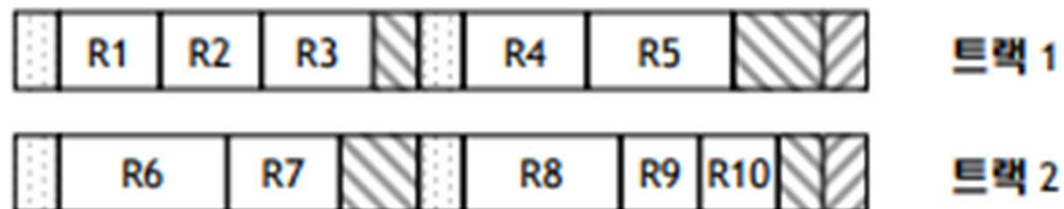
# 블로킹 (Blocking) (계속)



고정 길이 블로킹



신장된 가변 길이 블로킹



비신장 가변 길이 블로킹

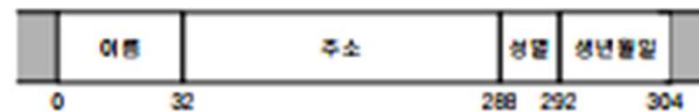
# 블로킹 (Blocking) (계속)

## ■ 레코드와 블록

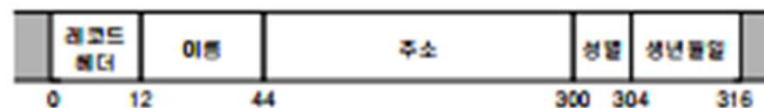
- 고정 길이 블로킹: 길이만 알면 레코드 구분이 가능함
  - $n$ 번째 레코드 시작 =  $n \times R$  or  $(n-1) \times R$
- 가변 길이 블로킹
  - 분리 표시: 레코드 끝 마크(end-of-record marker) 삽입
  - 각 레코드 앞에 길이 지시자 (length indicator) 관리
  - 위치 테이블(position table) 관리
    - 저장 시스템에서는 attribute에 대해서는 길이 지시자를 관리하고, record에 대해서는 위치 테이블을 관리하는 hybrid 방법을 주로 사용함

## ■ 레코드 설계 시 고려 사항

- 메모리 바이트 주소 특성
  - (4-byte 단위, 8-byte 단위)
- 레코드 헤더의 관리



(a) 학생 레코드 레이아웃



(b) 레코드 헤더가 추가된 학생 레코드 레이아웃

# 블로킹 (Blocking) (계속)

## ■ 블로킹의 고려 사항

### □ 적재 밀도 (loading density)

- 갱신을 위한 자유 공간 (free space) 할당
  - 레코드의 크기 증가 및 레코드의 삽입을 고려하여 일정 비율의 dummy space를 유지함
- 실제 데이터 저장 공간에 대한 (자유 공간을 포함한) 총 공간과의 비율

### □ 균형 밀도 (equilibrium density)

- 레코드 자체의 확장과 축소에 따른 레코드의 이동 빈도, 단편화 등의 척도
- 상당히 긴 기간 동안 시스템을 운용하고 안정시킨 뒤에 예상되는 저장 밀도

### □ 집약성 (locality)

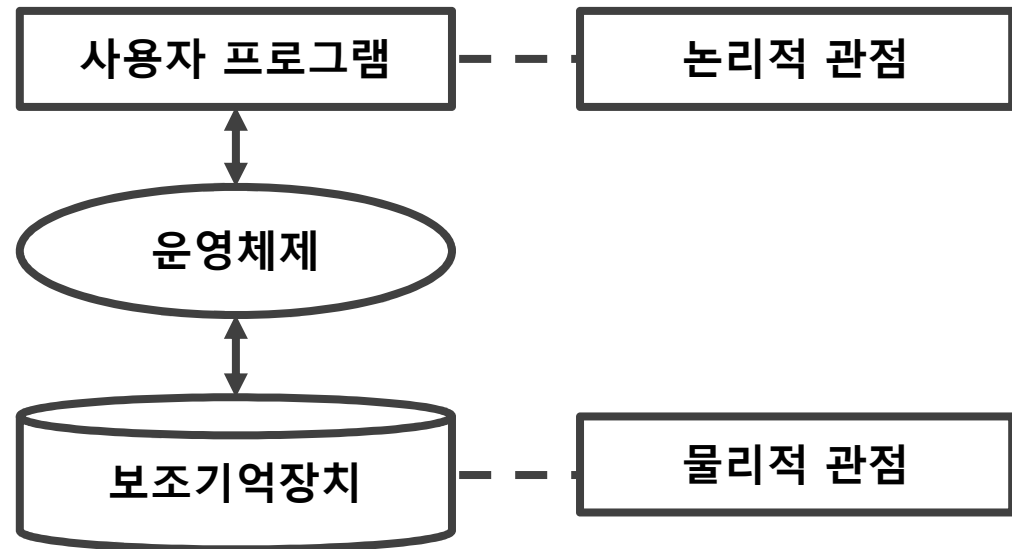
- 논리적으로 연관된 레코드들이 물리적으로 가깝게 위치하는 정도
- 예: 학번이 유사한 학생의 레코드는 동일 블록 내에 있거나 근접한 블록에 존재하는 경우, 집약성이 강하다(strong)고 이야기함

# 파일의 입출력 제어 환경

- 사용자 (프로그램) 관점: 논리적 구조의 파일을 사용함
  - 사용자 입장에서 파일은 정형화된 구조(예: 레코드의 집합, 프로그램 파일)를 가짐
  - 사용자는 실제로 파일이 디스크의 어디에 어떤 형태로 저장되어 있는지 확인하기 어렵고 확인할 필요가 없음
- 운영체제(with 파일 관리자) 관점: 물리적 구조의 파일을 사용함
  - 파일이 어떠한 논리적 구조를 가지고 있는지는 중요치 않음
  - 주어진 파일을 실제로 어떻게 찾고(디렉토리 관리), 어떻게 저장하고, 어떻게 해석하는지에 대한 책임을 가짐
  - Channel, Device Driver 등을 통하여 실제 I/O를 수행하는 주체이기도 하며, 수행된 I/O의 결과로 사용자에게 논리적 관점의 파일 구조를 전달함

# 파일의 입출력 제어 환경 (계속)

- 운영체제: 다수 사용자를 위해 컴퓨터 자원을 관리하는 SW

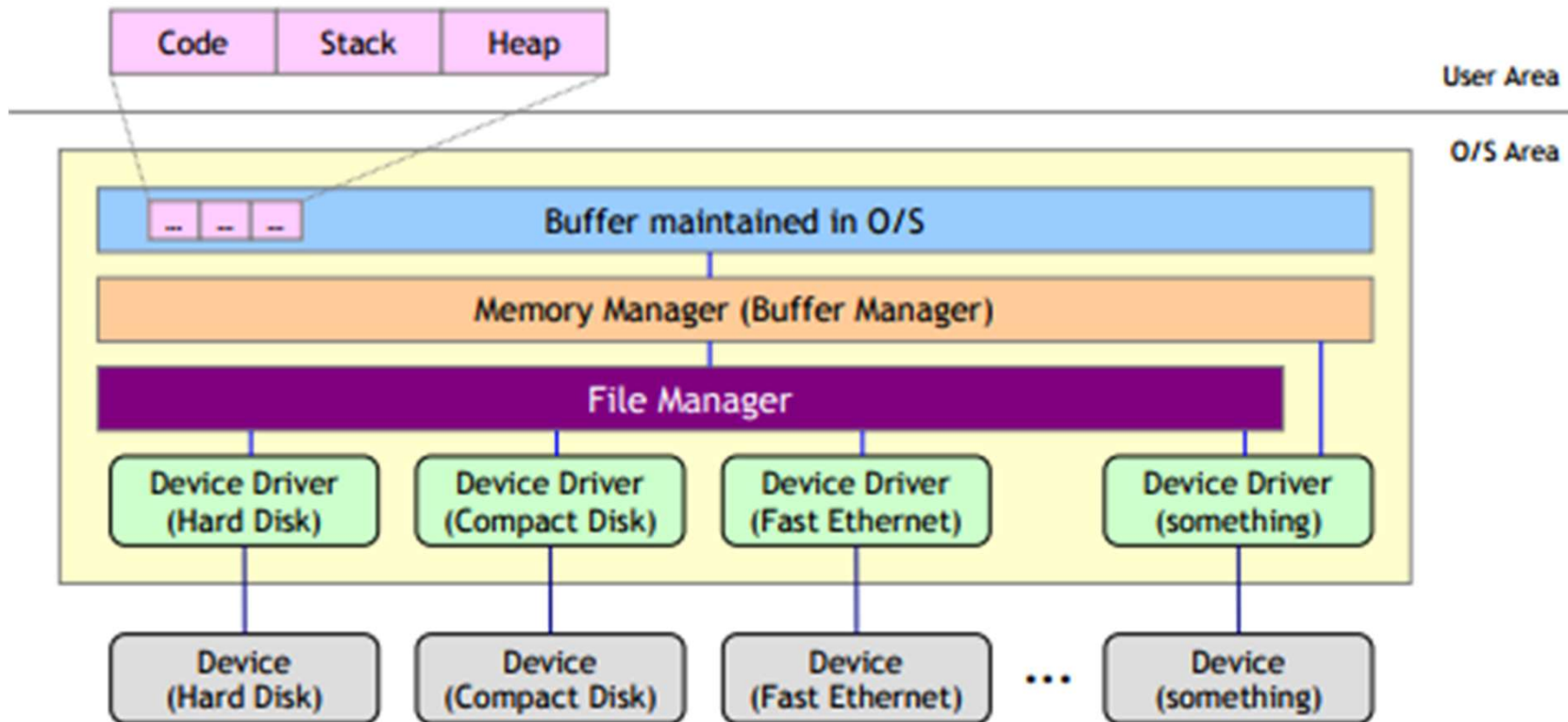


- 운영체제의 기능

- Main Memory Manager
- Process Manager
- Scheduler
- File Manager
  - 파일 조직 기법 제공 (블로킹, 분산 저장 등)
  - 사용자의 I/O 명령문(read, write, ...)에 대한 I/O 처리
- Device Manager: 물리적 저장 장치에 대한 접근 기능 제공

# 파일의 입출력 제어 환경 (계속)

- File Manager 중심의 운영체제 구조 (예제)





# Unix에서의 입출력

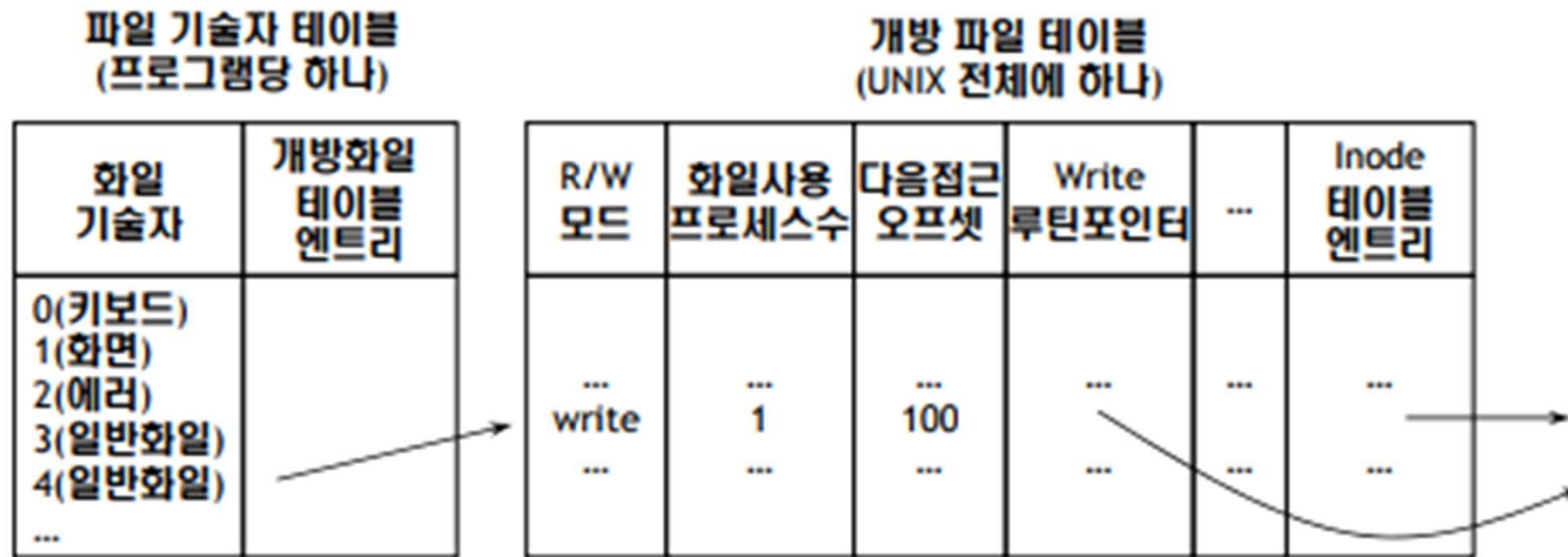
- UNIX에서는 파일을 단순히 바이트의 시퀀스(sequence of bytes)로 가정
  - 디스크 자체, 디스크 파일, 키보드, 콘솔 장치 등도 파일로 취급 (/dev/...)
- 파일 기술자 (file descriptor)
  - 사용자에게 파일을 접근할 수 있도록 하는 식별자(identifier)로서, 정수로 표현
  - 파일 세부 정보(경로, 권한, offset 등)를 저장한 배열의 인덱스 역할 수행
  - Special file descriptor
    - 표준 입력(키보드) = 0 (= STDIN)    fscanf(stdin,...);
    - 표준 출력(출력 화면) = 1 (= STDOUT)    fprintf(stdout,...);
    - 표준 에러 = 2 (= STDERR)    fprintf(stderr,...);
  - 사용자가 개방(open)한 파일은 3부터 부여
- 프로세스와 커널
  - 프로세스: 동시에 실행 가능한 프로그램 (모든 작업의 기본 단위)
  - 커널(kernel): 프로세스 이하의 모든 계층을 통합 (UNIX O/S와 동일한 개념)
    - 커널에서는 I/O를 일련의 바이트 상에서의 연산으로 여김

# Unix에서의 입출력 (계속)

- 커널의 I/O 시스템이 관리하는 테이블
  - 파일 기술자 테이블 (file descriptor table)
    - 각 프로세스가 사용하는 파일 기록 테이블 (매핑 테이블에 해당함)
  - 개방 파일 테이블 (open file table)
    - 현재 시스템이 사용 중인 개방된 파일에 대한 엔트리 저장
    - 각 엔트리는 판독/기록 허용 여부, 사용 프로세스 수, 다음 연산을 위한 파일 오프셋 등을 저장 (파일 오프셋의 경우는 파일 기술자 테이블에 저장될 수도 있음)
  - 인덱스 노드 (index node), 파일 할당 테이블 (file allocation table)
    - 파일의 저장 위치, 크기, 소유자 등의 정보를 관리
    - 디렉토리 엔트리의 개념으로 이해할 수 있음
  - 인덱스 노드 테이블 (index node table)

# Unix에서의 입출력 (계속)

- 파일 기술자 테이블과 개방 파일 테이블



# Unix에서의 입출력 (계속)

## ■ Inode (index node) 구조

inode 테이블의  
한 엔트리

소유자 ID
장치
그룹 이름
파일 유형
접근 권한
파일 접근 시간
파일 수정 시간
파일 크기 (블록수)
블록 카운트
파일 할당 테이블

파일 할당 테이블  
(계층 구조를 가질 수 있음)

데이터 블록번호 0
데이터 블록번호 1
...
데이터 블록번호 9

# Unix에서의 입출력 (계속)

## ■ UNIX에서의 파일 입출력 절차

□ 파일 기술자 값이 3인 파일에 대한 레코드 판독(read) 명령어를 처리한다고 가정

- 1. 프로그램의 파일 기술자 테이블에서 개방 파일 테이블을 이용하여, 개방 파일 테이블의 해당 엔트리를 검색
- 2. 개방 파일 테이블에서 Inode 테이블 포인터를 이용하여, Inode 테이블의 해당 엔트리를 검색
- 3. Inode 테이블에서 해당 파일의 데이터가 저장된 디스크 블록의 주소를 얻어 디스크에서 데이터 블록을 판독

# Unix에서의 입출력 (계속)

## ■ 디렉토리와 파일

- 디렉토리: 파일을 식별하는 Inode 번호와 그에 해당하는 파일 이름을 데이터로 저장한 파일 (디렉토리 자체도 파일로 관리됨)
- Inode에 대한 포인터는 파일에 대한 모든 정보를 참조함 (UNIX에서 파일 관리의 기본은 Inode 구조로 볼 수 있음)

## ■ 파일 이름과 Inode

- 여러 파일 이름이 같은 Inode를 포인터로 가리킬 수 있음 (hard link 구조)
- 이경우, 한 파일 이름이 삭제되더라도 포인터 수만 감소시킴

# 순차 파일 (Sequence File)

## ■ 정의

- 레코드들을 저장하는 가장 기본적인 방법으로서, 레코드들을 (지정된 순서에 따라) 연속적으로 저장하는 파일
- 레코드들을 접근할 때도 저장할 때의 순서대로 연속적으로 접근해야 함

## ■ 종류

- **입력 순차 파일 (entry-sequenced file):**
  - 레코드가 입력되는 순서대로 저장함 (즉, 입력 순서가 레코드의 저장 순서를 결정함)
  - Heap File(차레로 쌓는다는 의미)이라고도 함
- **키 순차 파일(key-sequenced file)**
  - 레코드의 특정 필드 값(키 값) 순서에 따라 저장
  - 특정 필드 값에 따라 정렬(sorting)된 상태를 유지함

# 스트림 파일(Stream File)

## ■ 정의

- 연속적인 판독(read) 연산을 통해 레코드가 파일에 저장되어 있는 순서에 따라 데이터를 액세스하는 파일
- 데이터가 하나의 연속된 바이트 스트림으로 구성

## ■ 종류

- 순차 접근 스트림 파일 (sequential access stream file)
  - 순차 접근만을 허용함 (예를 들어, 테이프의 경우 순차 접근만이 허용)
- 임의 접근 스트림 파일 (random access stream file)
  - 임의 접근이 허용됨 (예를 들어, 디스크의 경우 임의 접근도 허용함)

## ■ 접근 모드 (access mode)

- 파일에서 수행하려는 연산에 따라 판독(read), 기록(write), 갱신(read/write), 첨가(append) 등을 파일 개방(open) 시에 명시하여야 함



# 순차 접근 스트림 파일

## ■ 판독(read) 연산

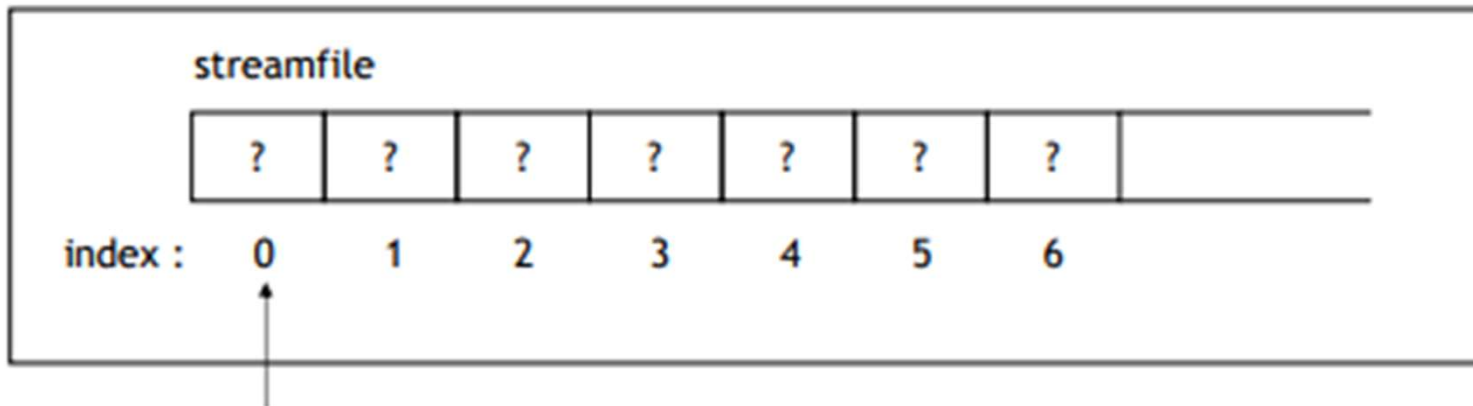
- 기본 스트림 화일을 판독(read) 모드로 열면 판독 포인터는 파일의 첫 번째 바이트를 가리킴
- 해당 위치에서 시작하여 해당 바이트 값을 전송하고, 판독 포인터를 스트림 파일의 다음 바이트 시작 위치로 변경함
- $n$ 번째 바이트 값을 판독하기 위해서는 반드시  $(n-1)$ 번째 바이트 값을 판독해야 함

## ■ 기록(write) 연산

- 파일을 기록(write) 모드로 열면 기록 포인터는 화일의 첫 번째 바이트가 기록될 위치를 가리킴
- 해당 위치에서 시작하여 해당 바이트 값을 기록하고, 기록 포인터를 다음 바이트가 기록될 위치로 변경함
- $n$ 번째 바이트 값을 기록하기 위해서는 반드시  $(n-1)$ 번 기록 연산을 수행해야 함

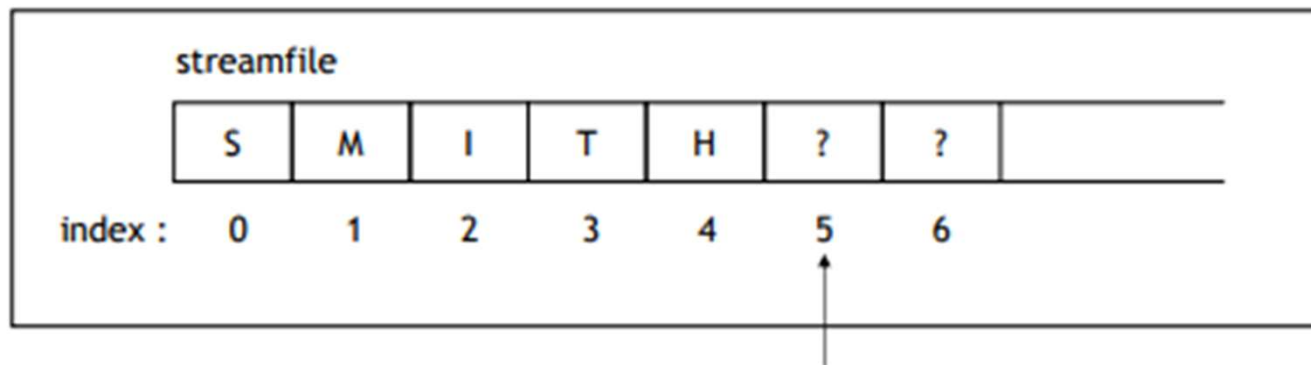
# 순차 접근 스트림 파일 (계속)

- C 언어를 이용한 스트림 파일의 생성
- `streamfile = fopen("stream.txt", "w");`
  - 이름이 "stream.txt"인 공백 스트림 파일이 생성되어 기록(write) 모드로 개방된다.
  - 아래 그림에서 화살표는 개방된 상태에서의 인덱스 값을 갖는 포인터를 나타낸다.



# 순차 접근 스트림 파일 (계속)

- C 언어를 이용한 스트림 파일의 생성 (계속)
- `fputc(ch, streamfile);`
  - 스트림 파일에 한 글자(character)를 기록한다.
  - `fputc()`를 연속적으로 사용하여 S, M, I, T, H 값을 파일에 기록
  - 아래 그림은 상기와 같이 다섯 글자를 기록한 이후의 포인터의 위치임

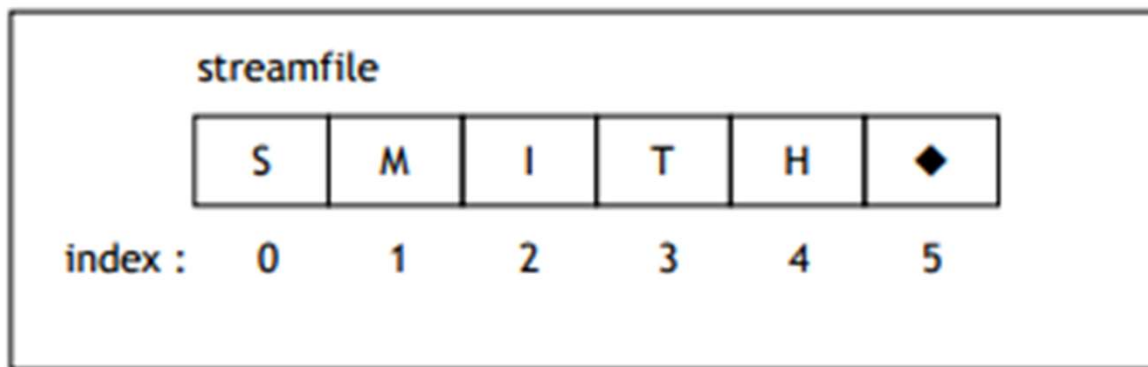


# 순차 접근 스트림 파일 (계속)

- C 언어를 이용한 스트림 파일의 생성 (계속)

- `fclose(streamfile);`

- 앞서 개방한 스트림 파일 `streamfile`을 닫음
- "◆"로 표현된 end-of-file 표시가 스트림 파일 끝에 첨가 (append)됨
- 지금까지 생성하고 기록한 스트림 파일은 "stream.txt"라는 파일 이름으로 저장됨



# 순차 접근 스트림 파일 (계속)

## ■ 순차 접근 스트림 파일

- 연속적으로 파일을 접근하고, 파일에 있는 모든 바이트를 처리하는 경우에 유용
- 화일을 순차적으로 접근하는 과정은 배열을 순차적으로 접근하는 것과 유사
- 특정 바이트 혹은 바이트 스트링만을 찾기 위한 방법으로는 좋지 않음

## ■ 임의 접근 스트림 파일

- 이원 탐색법 (binary search): 배열의 인덱스를 이용하여 배열의 원소를 직접(임의로) 액세스
- 이원 탐색법을 파일에 적용
  - 반드시 파일에 있는 바이트를 임의로 액세스할 수 있어야 함
  - 순차 접근 스트림 파일을 불가하며, 임의 접근 스트림 화일은 가능함

# 임의 접근 스트림 파일

## ■ 오프셋(offset) 값을 이용

- 순차적으로 바이트를 액세스하는 것이 아니라, 오프셋이 가리키는 위치에 대한 임의 접근을 수행

## ■ 임의 접근을 위한 함수

### □ fseek() 함수

- 파일 스트림에서 판독 또는 기록 포인터의 위치를 변경하는 데 사용
- 파일의 시작, 끝, 현재의 위치로부터 오프셋 크기 만큼 판독 또는 기록 포인터를 이동시킴

### □ ftell() 함수

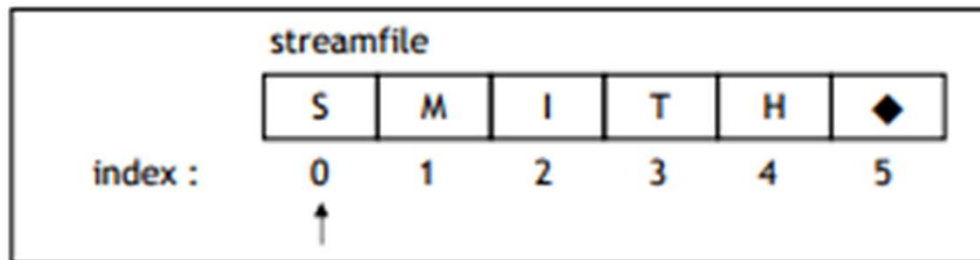
- 파일 스트림에서 판독 또는 기록 포인터의 인덱스 값을 반환하는 데 사용
- 즉, 파일에서 포인터가 가리키는 현재 위치(오프셋)를 리턴함

# 임의 접근 스트림 파일 (계속)

## ■ 판독 모드("r")로 개방한 스트림 파일

□ `streamfile = fopen("stream.txt", "r");`

- 파일 "stream.txt"를 판독 모드로 개방함
- 아래 그림과 같이, 파일이 열리면 판독 포인터는 파일의 첫 번째 바이트를 가리키도록 설정됨



□ `offset = ftell(streamfile);`

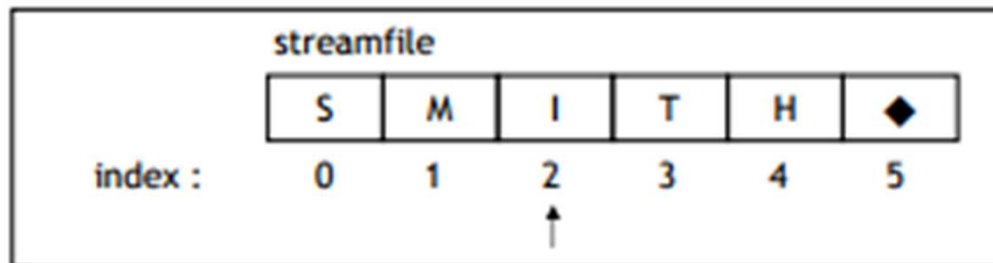
- 현재의 포인터 값을 반환해 주는 함수
- 현재 판독 포인터는 인덱스 값이 0이므로, 0이 반환됨

# 임의 접근 스트림 파일 (계속)

## ■ 판독 모드("r")로 개방한 스트림 파일 (계속)

### □ fseek(streamfile, 2, SEEK\_SET);

- 시작 위치(SEEK\_SET)로부터 판독 포인터를 2바이트 이동시킴
- SEEK\_SET: 시작 위치, SEEK\_END: 끝위치, SEEK\_CUR: 현재 위치
- 위 함수의 수행에 따라 판독 포인터의 위치는 다음과 같이 변경됨



### □ offset = ftell(streamfile);

- 현재 판독 포인터는 인덱스 값이 2이므로, 2가 반환됨



# 순차 파일의 유형

## ■ 입력 순차 파일 (entry-sequenced file)

- 레코드가 순서대로 쌓여 있다는 의미에서 heap file이라고도 함
- 레코드에 대한 분석, 분류, 표준화 과정을 거치지 않음
- 필드의 순서, 길이 등에 대해서도 제한 없음
- 레코드의 길이, 타입도 일정하지 않을 수 있음
- 레코드는 여러 개의 <필드, 값> 쌍으로 구성



# 순차 파일의 유형 (계속)

## ■ 입력 순차 파일의 갱신 작업

- 새로운 레코드 삽입, 기존 레코드 삭제, 기존 레코드 변경 등
- 레코드 삽입: 기존 파일 끝에 레코드를 첨가(append)
- 레코드 삭제 및 변경
  - 새로운 순차 파일을 생성하면서 동시에 수행
  - 작업 대상 레코드를 검색하면서 기존의 레코드를 새로운 파일로 출력
    - 해당 레코드 검색 시 삭제 혹은 변경 작업 수행 (변경 시, 변경 결과를 새로운 파일로 출력)
    - 나머지 남은 레코드들을 다시 새로운 파일로 모두 출력

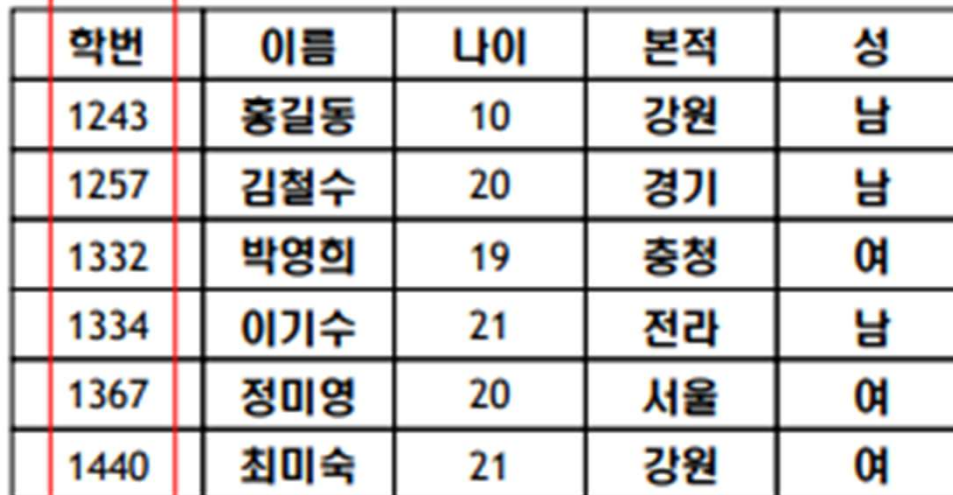
## ■ 입력 순차 파일의 검색 작업

- 주어진 필드 값(예: 학번 = 20051234)에 대응하는 레코드를 찾기 위하여, 첫 번째 레코드부터 차례로 검색하면서 원하는 필드 값을 순차적으로 검색(비교)
- 키 필드(key field): 검색을 수행하는 데 있어서, 레코드에서 비교 대상(검색 대상)이 되는 필드(예: 학생 테이블에서 학번 필드)
- 탐색 키 필드(search key field): 검색을 수행하는 데 있어서 비교 대상(검색 대상)으로 주어진 필드 (예: 찾고자 하는 학생의 학번인 20051234)

# 순차 파일의 유형 (계속)

## ■ 키 순차 파일(key-sequenced file)

- 저장 장치의 레코드 순서와 파일에 저장된 레코드들의 논리적 순서가 같은 구조의 파일
- 파일 내의 레코드들이 키 필드 값에 따라 정렬됨
- 데이터 필드, 즉, 애트리뷰트 이름은 개별 레코드가 아닌 파일 설명자에 한 번만 저장하면 됨
  - 앞서 예를 든 순차 파일의 경우 매번 필드 이름을 저장하고 있음



학번	이름	나이	본적	성
1243	홍길동	10	강원	남
1257	김철수	20	경기	남
1332	박영희	19	충청	여
1334	이기수	21	전라	남
1367	정미영	20	서울	여
1440	최미숙	21	강원	여

키 필드: 레코드의 순서를 결정하는 필드임

# 순차 파일의 유형 (계속)

## ■ 키 순차 파일(계속)

- 키 순차 파일은 정렬된 화일(sorted file)이라고도 함 (레코드들이 특정 키 필드 값에 따라 정렬된 파일을 “정렬된 파일”이라 함)
- 정렬 키(sort key): 정렬 순서 결정에 사용된 값의 필드
- 오름차순(ascending) / 내림차순(descending) 정렬

## ■ 키 순차 파일의 특징

- 일괄 처리(batch processing)에서 많이 사용
- 순서상 다음 위치에 해당하는 레코드를 신속하게 접근할 수 있음
- 하나의 순차 파일이 두 개의 상이한 정렬 순서를 만족시킬 수 없음
  - 예: 학번과 나이를 모두 키 필드로 사용할 수 없음, 나이 순으로 학번이 부여된다면?
- 여러 정렬 순서 파일이 필요한 경우에는 임시 파일을 생성했다가 용도가 끝나면 파일을 삭제함

# 순차 파일 설계 시 고려 사항

- 레코드 내의 필드 배치는 어떻게 할 것인가?
  - 활동 파일(active file)과 비활동 파일(inactive file)을 구분하여 저장
    - 활동 파일에 대한 크기를 감소시켜 액세스가 빠르게 함
  - 레코드 내의 필드 및 크기를 고려하여 고정 길이 혹은 가변 길이를 선택함
    - 레코드 구조 및 크기가 유사하면 고정 길이를, 그렇지 않으면 가변길이를 사용
- 키 필드는 어느 것으로 할 것인가?
  - 응용의 종류 및 특성에 따라 선정 (예: 전화번호부 – 가입자 이름)
  - 키 필드의 선정은 레코드 액세스 순서를 나타내므로 성능에 큰 영향을 미침
- 적정 블로킹 인수는 얼마로 하여야 하는가?
  - 순차 파일에서는 일반적으로 가능한 블록을 크게 하는 것이 바람직하며, 블로킹 인수 또한 큰 값을 사용하는 것이 유리함
    - 가능한 단번에 많은 데이터를 송수신하는 것이 유리하며, 레코드 중간에 레코드의 추가가 어려우므로 블로킹 인수도 큰 값이 유리함
  - 블록 크기는 버퍼 크기나 운영 체제가 지원하는 블록 크기에 의해 제한될 수 있음

# 감사합니다!

담당교수: 전강욱(컴퓨터공학부)

[kw.chon@koreatech.ac.kr](mailto:kw.chon@koreatech.ac.kr)