

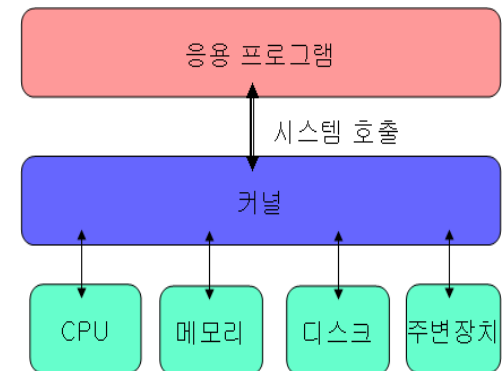
4장 파일 입출력

4.1 시스템 호출

커널과 컴퓨터 시스템 구조

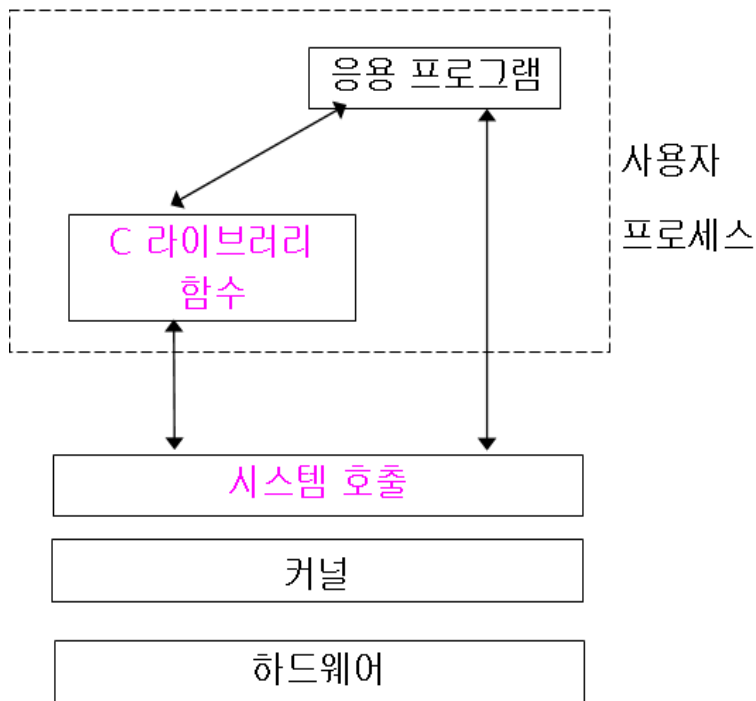
- 유닉스 커널(kernel)

- 하드웨어를 운영 관리하여 다음과 같은 서비스를 제공
 - 하드웨어는 CPU, 메모리, 저장장치, 주변장치로 구성
- 프로세스 관리(Process management)
- 메모리 관리(Memory management)
- 파일 관리(File management)
- 주변장치 관리(Device management)
- 통신 관리(Communication management)



시스템 호출 (system call)

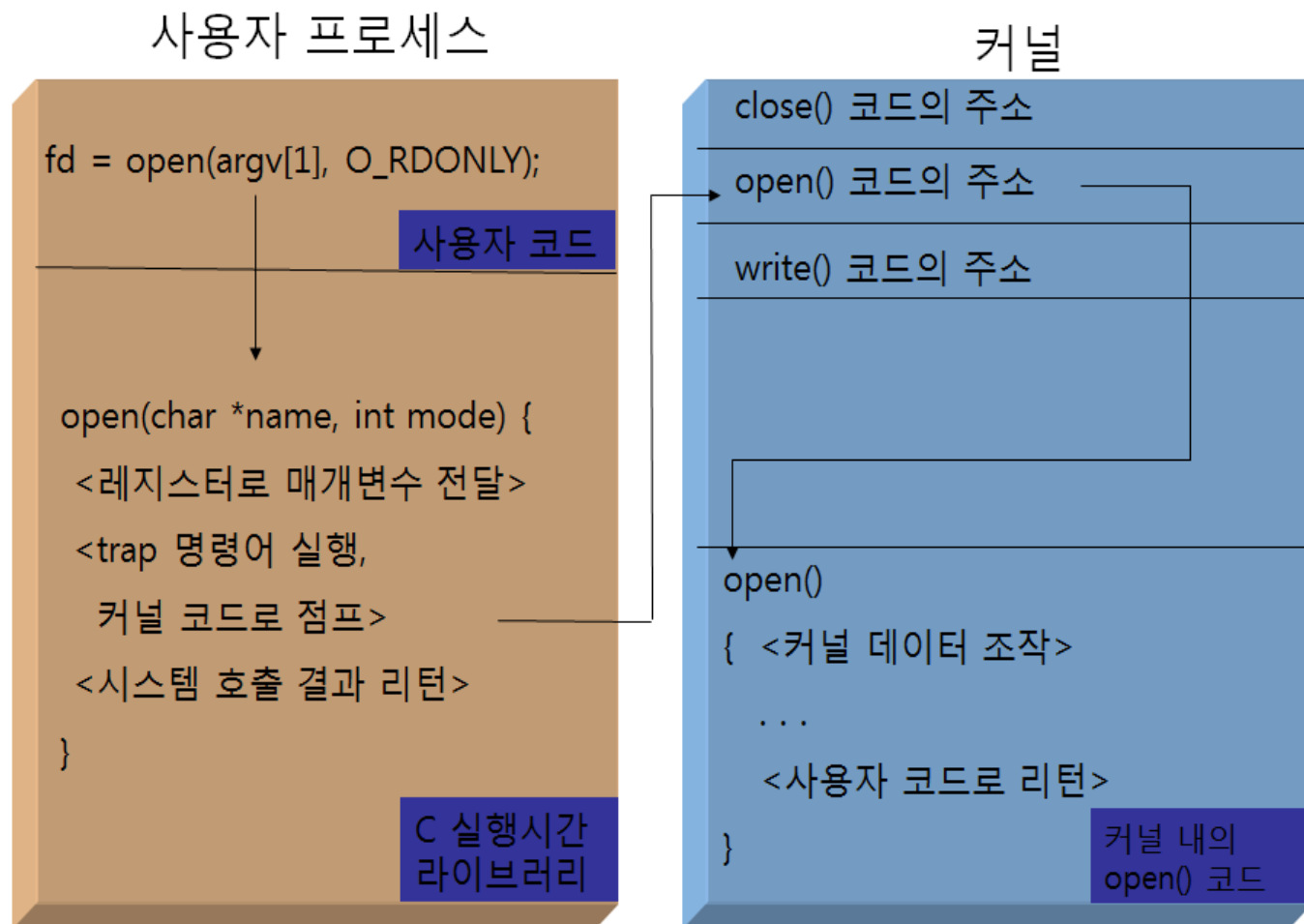
- **시스템 호출**은 커널에 서비스 요청을 위한 프로그래밍 인터페이스
- 응용 프로그램은 시스템 호출을 통해서 커널에 서비스를 요청한다.
- 응용 프로그램과 커널 사이의 interface



응용 프로그램이 라이브러리 함수를 호출할 수도 있으나, 이 역시 함수 내에서 관련된 시스템 호출을 수행 함

사용자 모드 vs. 커널 모드
사용자 프로세스 vs. 커널 프로세스

시스템 호출 과정



open() 시스템 호출을 수행하면 “C 실행시간 라이브러리”를 통해 커널내의 해당 코드로 점프
참고) C 실행 시간 라이브러리 != C 라이브러리 함수

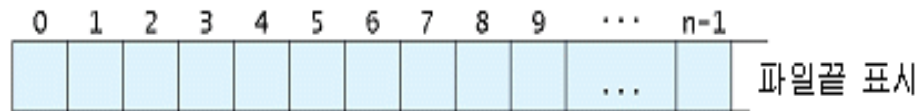
주요 시스템 호출 요약

주요 자원	시스템 호출
파일	open(), close(), read(), write(), dup(), lseek() 등
프로세스	fork(), exec(), exit(), wait(), getpid(), getppid() 등
메모리	malloc(), calloc(), free() 등
시그널	signal(), alarm(), kill(), sleep() 등
프로세스 간 통신	pipe(), socket() 등

4.2 파일

유닉스와 리눅스에서 파일

- 유닉스에서 **파일**이란 연속된 **바이트의 나열 (byte sequence)**
 - 모든 파일 데이터들은 결국은 바이트로 바뀌어서 파일에 저장
 - 이들 바이트들을 어떻게 해석하느냐는 전적으로 프로그래머(운영 체제)의 책임
- 유닉스와 리눅스는 그 이외의 파일별 특별한 다른 포맷을 정의하지 않음
- 디스크 파일뿐만 아니라,
부 장치에 대한 인터페이스 (즉, 장치 파일) 역시 파일로 구현

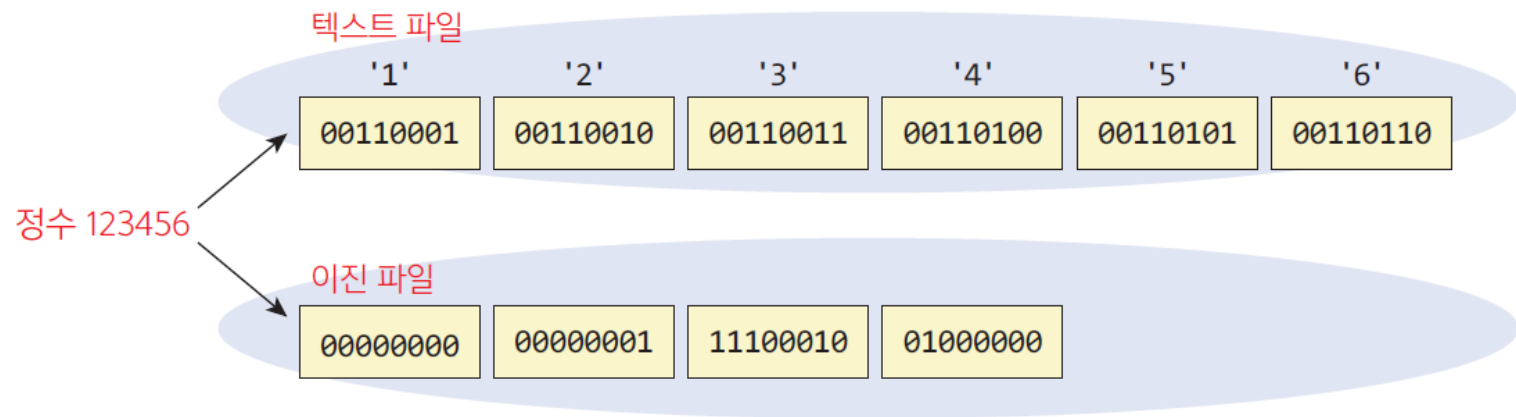


파일 종류

- 텍스트 파일(text file)
 - 사람들이 읽을 수 있는 **문자들을 저장**하고 있는 파일
 - ASCII 코드 값으로 저장되고 해석된다.
- 이진 파일(binary file)
 - 모든 데이터는 있는 그대로 **바이트의 연속으로 저장**
 - 이진 파일을 이용하여 메모리에 저장된 변수 값 형태 그대로 파일에 저장할 수 있다.

프로그램 파일 (text file) vs. 목적 파일 (object file, binary file)

파일 종류



파일 입출력

- C 언어의 파일 입출력 과정

1. 파일 열기

`fopen()` 라이브러리 함수 혹은 `open()` 시스템 호출 사용

2. 파일 입출력 수행

다양한 파일 입출력 함수 사용 (file read and write)

3. 파일 닫기

`fclose()` 혹은 `close()` 사용

파일 열기: open()

- 파일을 사용하기 위해서는 먼저 open() 시스템 호출을 이용하여 파일을 열어야 한다.
 - 파일을 사용하기 전 왜 열어야 하는가?

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
int open (const char *path, int oflag, [ mode_t mode ]);
```

파일 열기에 성공하면 파일 디스크립터를, 실패하면 -1을 리턴

- open()는 path가 나타내는 파일을 연다
 - 성공 시, 파일 디스크립터(file descriptor) 반환, 이는 열린 파일을 나타내는 번호이다. 실패 시, -1 반환
 - 즉, open 함수는 path name을 file descriptor로 변환하는 역할

파일 열기: open()

- 두번째 매개 변수 oflag : 파일을 어떤 형태로 open 할 것인가?
 - O_RDONLY
읽기 모드, read() 호출은 사용 가능
 - O_WRONLY
쓰기 모드, write() 호출은 사용 가능
 - O_RDWR
읽기/쓰기 모드, read(), write() 호출 사용 가능
 - O_APPEND
데이터를 쓰면 파일 끝에 첨부된다.
 - O_CREAT
해당 파일이 **없는 경우에 생성**하며
세번째 매개 변수 mode는 O_CREAT의 경우 생성할 파일의 사용 권한을 나타내게 된다.

파일 열기: open()

- oflag
 - O_TRUNC
파일이 이미 **있는 경우 내용을 지운다.**
 - O_EXCL
O_CREAT와 함께 사용되며 **해당 파일이 이미 있으면 오류 출력**
 - O_NONBLOCK
non-blocking 모드로 입출력 (버퍼 사용 모드) 하도록 한다
 - O_SYNC
write() 시스템 호출을 하면 디스크에 물리적으로 쓴 후 반환된다
즉, 현재 버퍼에 저장된 데이터를 다 쓴 후 반환토록 한다

파일 열기: 예

O_CREAT을 사용하는 경우에만 세번째 매개변수 사용

- `fd = open("account", O_RDONLY);`
- `fd = open(argv[1], O_RDWR);` (첫번째 명령줄 인수)
- `fd = open(argv[1], O_RDWR | O_CREAT, 0600);`
 - 읽기 쓰기 모드로 열고, 해당 파일이 없으면 새로 생성, 파일의 권한은 600
- `fd = open("tmpfile", O_WRONLY|O_CREAT|O_TRUNC, 0600);`
 - 쓰기 전용으로 열고, 없으면 새로 생성하고, 기존 내용이 있으면 지운다. 없으면 600 권한을 생성
- `fd = open("/sys/log", O_WRONLY|O_APPEND|O_CREAT, 0600);`
 - 첨부용 쓰기 전용으로 연다. 없으면 600 권한으로 생성
- `if ((fd = open("tmpfile", O_WRONLY|O_CREAT|O_EXCL, 0666)) == -1)`
 - 파일이 없는 경우에만 생성하고, 기존의 파일이 있으면 오류, 없으면 666 권한으로 생성

fopen.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
    int fd;
    if ((fd = open(argv[1], O_RDWR)) == -1)
        perror(argv[1]); /* 오류 메시지 출력 함수 */
    printf("파일 %s 열기 성공\n", argv[1]);
    close(fd);
    exit(0);
}
```


creat()은 open()으로 묘사가 가능하다.
따라서 부차적, 이차적 함수이다.

파일 생성: creat()

- creat() 시스템 호출

- path가 나타내는 파일을 생성하고 쓰기 전용으로 연다.
- 생성된 파일의 사용권한은 mode로 정한다.
- 기존 파일이 있는 경우에는 그 내용을 삭제하고 연다. //O_TRUNC
- creat()은 다음 open() 시스템 호출과 완전히 동일
`open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);`

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
int creat (const char *path, mode_t mode );
```

파일 생성에 성공하면 파일 디스크립터를, 실패하면 -1을 리턴

파일 닫기: close()

- close() 시스템 호출은 fd가 나타내는 파일을 닫는다.
 - 파일을 사용한 후에는 왜 닫아야 하는가?

```
#include <unistd.h>
```

```
int close( int fd );
```

close는 파일 위치를 끊음

fd가 나타내는 파일을 닫는다.

성공하면 0, 실패하면 -1을 리턴한다.

- 0. stdin -> 키보드
- 1. stdout -> 화면
- 2. stderr -> 화면
- 3. 파일 위치
- ...

데이터 읽기: read()

- read() 시스템 호출
 - fd가 나타내는 파일에서
 - nbytes 만큼의 데이터를 읽고
 - 읽은 데이터는 buf에 저장한다.
- 실제 읽은 크기 (nbytes)를 반환

```
#include <unistd.h>
```

```
ssize_t read ( int fd, void *buf, size_t nbytes );
```

파일 읽기에 성공하면 읽은 바이트 수, 파일 끝을 만나면 0,
실패하면 -1을 리턴

fsize.c (명령줄 인수로 받은 파일의 크기 계산)

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
#include <fcntl.h>
```

```
#define BUFSIZE 512
```

파일의 크기를 계산하는 다양한 명령어들의
내부 구현 형태라 생각할 수 있음

```
/* 파일 크기를 계산 한다 */
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    char buffer[BUFSIZE];
```

```
    int fd;
```

```
    ssize_t nread;           //signed integer type
```

```
    long total = 0;
```

```
    if ((fd = open(argv[1], O_RDONLY)) == -1)
```

```
        perror(argv[1]);     //오류 메시지를 stderr로 출력하는 함수
```

fsize.c

```
/* 파일의 끝에 도달할 때까지 반복해서 읽으면서 파일 크기 계산 */
while( (nread = read(fd, buffer, BUFSIZE)) > 0) // nread ← 실제 읽은 크기
    total += nread;    //512씩 total에 더함
close(fd);
printf ("%s 파일 크기 : %ld 바이트 \n", argv[1], total);
exit(0);
}
```

BUFSIZE가 512바이트, 한번 읽을 때 마다 512씩 (읽은 바이트 수만큼) 증가

512씩 읽다가 마지막 블록이 (가령) 510바이트면,
510이 total에 더해지고,
더 읽을 것이 없으면 0이 반환

데이터 쓰기: write()

- write() 시스템 호출
 - buf에 있는 nbytes 만큼의 데이터를 fd가 나타내는 파일에 쓴다

```
#include <unistd.h>
```

```
ssize_t write (int fd, void *buf, size_t nbytes);
```

파일에 쓰기를 성공하면 실제 쓰여진 바이트 수를 리턴하고,
실패하면 -1을 리턴

write() 함수 : buf → fd

c.f., read() 함수 : fd → buf

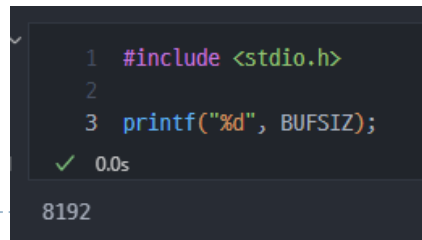
(Note)

\$copy a b 와 같은 명령어의 내부 구현 사례

다수의 exit() 함수 사용법 (c.f., exit(0) 정상 종료)

copy.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
/* 파일 복사 프로그램 */
main(int argc, char *argv[])
{
    int fd1, fd2, n;
    char buf[BUFSIZ]; //사전 정의 상수 8192
    if (argc != 3) {
        fprintf(stderr, "사용법: %s file1 file2\n",
            argv[0]);
        exit(1);
    }
```



```
1 #include <stdio.h>
2
3 printf("%d", BUFSIZ);
✓ 0.0s
8192
```

```
    if ((fd1 = open(argv[1], O_RDONLY)) ==
        -1) {
        perror(argv[1]);
        exit(2);
    }
    fd2 = open(argv[2], O_WRONLY | O_TRUNC | O_CREAT, 0644)
    if ((fd2 = open(argv[2], O_WRONLY |
        O_CREAT | O_TRUNC, 0644)) == -1) {
        perror(argv[2]);
        exit(3);
    }
```

```
    while ((n = read(fd1, buf, BUFSIZ)) > 0)
        write(fd2, buf, n); // 읽은 내용을 쓴다.
    exit(0);
}
```

파일 디스크립터 복제

- dup()/dup2() 호출은 기존의 파일 디스크립터를 복제한다.

```
#include <unistd.h>
```

```
int dup(int oldfd);
```

oldfd에 대한 복제본인 새로운 파일 디스크립터를 생성하여 그 번호를 반환한다.

실패하면 -1을 반환한다.

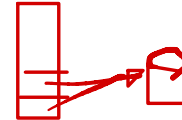
```
int dup2(int oldfd, int newfd);
```

oldfd을 newfd에 복제하고, 복제된 새로운 파일 디스크립터 번호를 반환한다.

실패하면 -1을 반환한다.

두 시스템 호출은 사용상의 차이일 뿐 기능은 동일

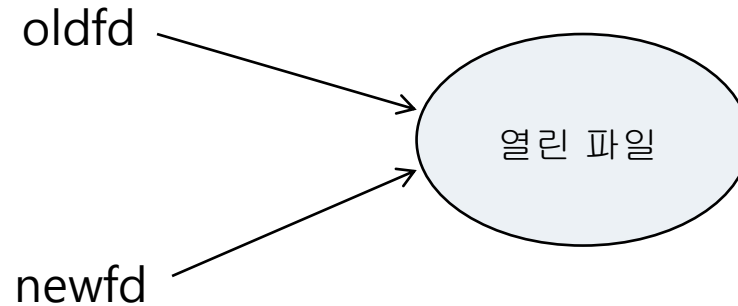
fd2 = dup (fd1); vs. dup2 (fd1, fd2);



- oldfd와 복제된 새로운 디스크립터는 하나의 파일을 공유한다.
- 입출력 재지정에 유용하게 사용
 - 표준출력을 통해 나오는 출력을 dup()을 통해 공유된 파일로 전달

파일 디스크립터 복제

- 열려있는 하나의 파일을 공유



dup.c

```
1 #include <unistd.h>
2 #include <fcntl.h>
3 #include <stdlib.h>
4 #include <stdio.h>
6 int main()
7 {
8     int fd, fd2;
9
10    if((fd = creat("myfile", 0600)) == -1)    (creat 함수가 fd 반환)
11        perror("myfile");
12
13    write(fd, "Hello! Linux", 12);    //fd is myfile
14    fd2 = dup(fd);                    // fd2 is now myfile
15    write(fd2, "Bye! Linux", 10);    // write to fd2(i.e., myfile)
16    exit(0);
17 }
```

```
$ ./dup
$ cat myfile (파일 내용 확인)
Hello! LinuxBye! Linux
```

4.3 임의 접근 파일

파일 위치 포인터(file position pointer)

- 파일 위치 포인터는 파일 내에 읽거나 쓸 위치인 현재 파일 위치(current file position)를 가리킨다



- 순서대로 출력가능하며, 임의의 위치로의 접근이 불가능
 - 임의의 위치로 이동 시킨 후, 출력 하게 하는 함수 → `lseek()`
 - C 표준 라이브러리 함수 `fseek()`과 유사

파일 위치 포인터 이동: lseek()

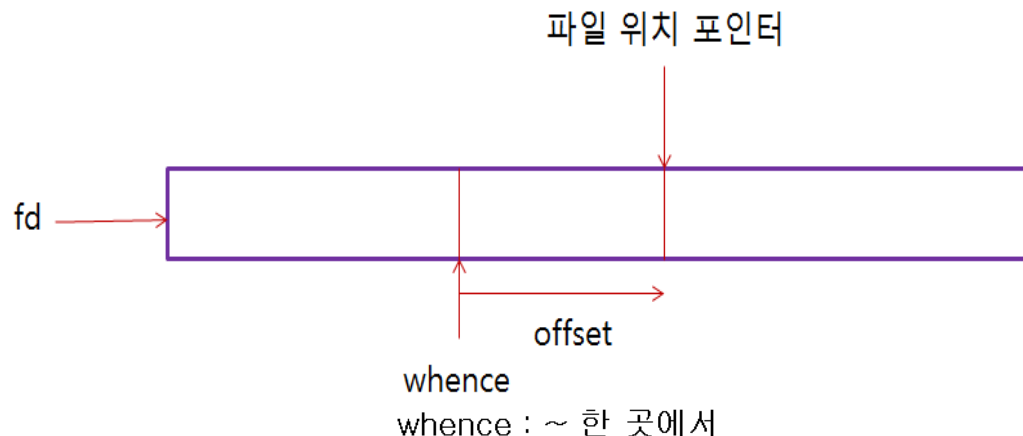
- lseek() 시스템 호출
 - 임의의 위치로 파일 위치 포인터를 이동시킬 수 있다.

```
#include <unistd.h>
```

Whence에서부터 offset 만큼 이동시킴

```
off_t lseek ( int fd, off_t offset, int whence );
```

이동에 성공하면 현재 위치를 리턴하고 실패하면 -1을 리턴한다.



Whence의 종류

SEEK_SET : 처음

SEEK_CUR : 현재

SEEK_END : 끝

파일 위치 포인터이동: 예

- 파일 위치 이동

- `lseek(fd, 0L, SEEK_SET);`
- `lseek(fd, 100L, SEEK_SET);`
- `lseek(fd, 0L, SEEK_END);`

파일 시작으로 이동(rewind)

파일 시작에서 100바이트 위치로

파일 끝으로 이동(append)

offset은 long type (L)

- 레코드 단위로 이동

- `lseek(fd, n * sizeof(record), SEEK_SET);` 데이터 단위 n+1번째 레코드 시작위치로
- `lseek(fd, sizeof(record), SEEK_CUR);` 다음 레코드 시작위치로
- `lseek(fd, -sizeof(record), SEEK_CUR);` 전 레코드 시작위치로

- 파일끝 이후로 이동

- `lseek(fd, sizeof(record), SEEK_END);` 파일 끝에서 한 레코드 다음 위치로

(이처럼 필요 시, 파일 끝 이후로도 이동 가능

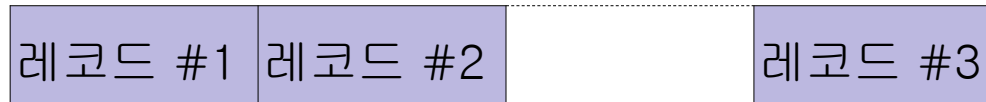
하며, 이 경우 레코드공간만큼 공백 발생)

`lseek(fd, 10L, SEEK_END)`

10뒤로 이동, 빈 데이터로 채워짐

레코드 저장 예

```
write(fd, &record1, sizeof(record));    // #1 쓰기  
write(fd, &record2, sizeof(record));    // #2 쓰기  
lseek(fd, sizeof(record), SEEK_END);    // 끝 이후로 이동  
write(fd, &record3, sizeof(record));    // 파일의 끝에서 다음 레코드에 #3이 위치
```



(실습) dbcreate.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include "student.h" // 학생 데이터베이스용 구조체 파일 →
/* 학생 정보를 입력받아 데이터베이스 파일에 저장한다. */
int main(int argc, char *argv[])
{
    int fd;
    struct student record;
    if (argc < 2) {
        fprintf(stderr, "사용법 : %s fileWn", argv[0]);
        exit(1);
    }
```

```
struct student {
    char name[MAX];
    int id;
    int score;
};
```


dbcreate.c

```
if ((fd = open(argv[1], O_WRONLY|O_CREAT|O_EXCL, 0640)) == -1) {  
    perror(argv[1]);  
    exit(2);  
}  
printf("%-9s %-8s %-4s\n", "학번", "이름", "점수");  
while (scanf("%d %s %d", &record.id, record.name, &record.score) == 3) {  
    lseek(fd, (record.id - START_ID) * sizeof(record), SEEK_SET);  
    write(fd, (char *) &record, sizeof(record) );  
}  
close(fd);  
exit(0);  
}
```

해당 파일이 있으면 오류 출력

scanf()의 반환값? 형식 인자의 개수
&없음

구조체 데이터를 이진 형식으로 저장

scanf로 입력받은 내용이 구조체 record에 저장되고
해당 record 내용을 write()을 이용해 파일에 씀

student.h

```
#define MAX 24
```

```
#define START_ID 1401001    //학번은 1401001부터 시작한다고 가정
```

```
struct student {  
    char name[MAX];  
    int id;  
    int score;  
};
```

dbquery.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include "student.h"
/* 학번을 입력받아 해당 학생의 레코드를 파일에서 읽어 출력한다. */
int main(int argc, char *argv[])
{
    int fd, id;  char c;
    struct student record;
    if (argc < 2) {
        fprintf(stderr, "사용법 : %s fileWn", argv[0]);
        exit(1);
    }
    if ((fd = open(argv[1], O_RDONLY)) == -1) {
        perror(argv[1]);
        exit(2);
    }
    }
```

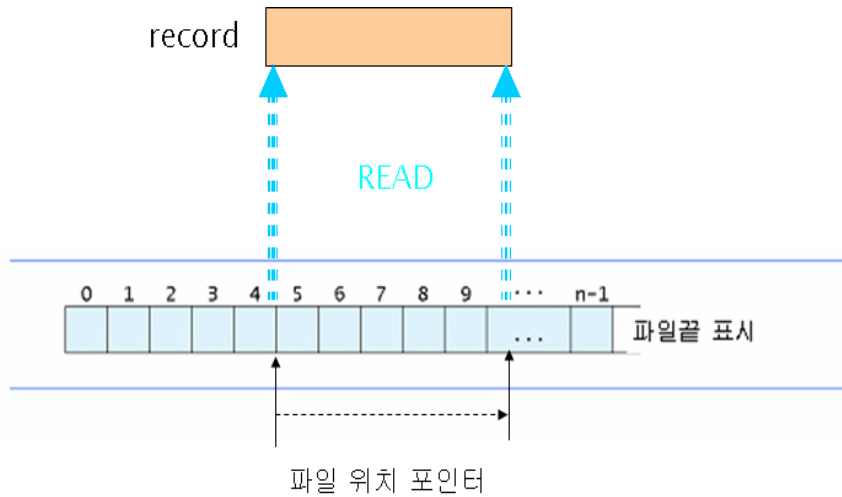
dbquery.c

```
do {
    printf("\n검색할 학생의 학번 입력:");
    if (scanf("%d", &id) == 1) {
        lseek(fd, (id-START_ID)*sizeof(record), SEEK_SET);
        if ((read(fd, (char *) &record, sizeof(record)) > 0) && (record.id != 0))
            printf("이름:%s\t 학번:%d\t 점수:%d\n", record.name, record.id,
                record.score);
        else printf("레코드 %d 없음\n", id);
    } else printf("입력 오류");
    printf("계속하겠습니까?(Y/N)");
    scanf(" %c", &c);
} while (c=='Y');
close(fd);
exit(0);
```

레코드 수정 과정

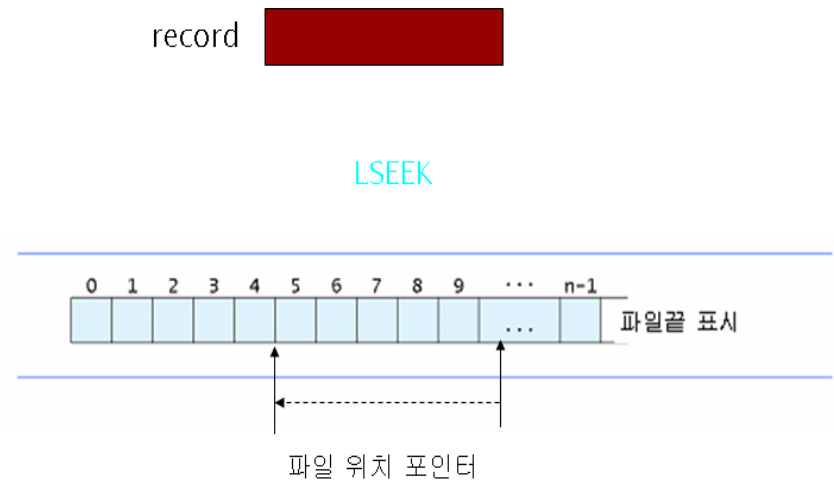
- (1) 파일로부터 해당 레코드를 읽어서
- (2) 이 레코드를 수정한 후에
- (3) 수정된 레코드를 다시 파일 내의 원래 위치에 써야 한다.

레코드 수정

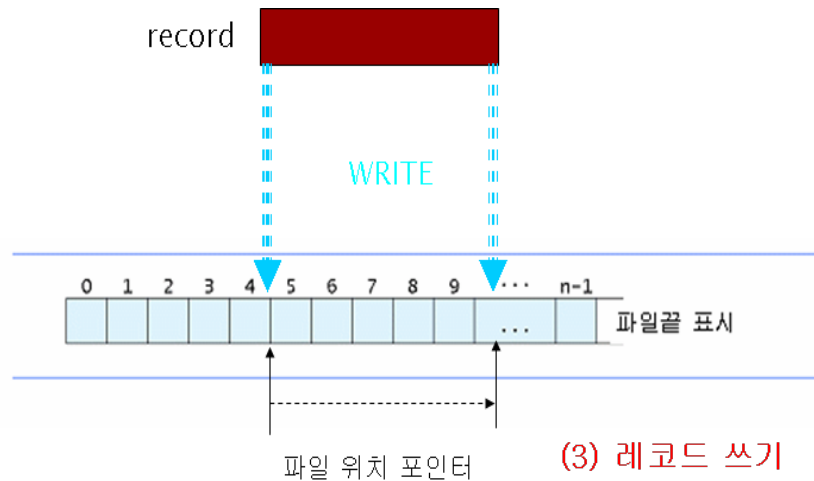


(1) 레코드 읽기

레코드 읽기



(2) 파일 위치 포인터 재조정
앞으로 다시 이동



(3) 레코드 쓰기

dbupdate.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include "student.h"
/* 학번을 입력받아 해당 학생 레코드를 수정한다. */
int main(int argc, char *argv[])
{
    int fd, id;
    char c;
    struct student record;
    if (argc < 2) {
        fprintf(stderr, "사용법 : %s fileWn", argv[0]);
        exit(1);
    }
    if ((fd = open(argv[1], O_RDWR)) == -1) {    //읽기 쓰기 권한 모두 획득
        perror(argv[1]);
        exit(2);
    }
}
```

dbupdate.c

```
do {
    printf("수정할 학생의 학번 입력: ");
    if (scanf("%d", &id) == 1) {
        lseek(fd, (id-START_ID)*sizeof(record), SEEK_SET);
        if ((read(fd, (char *) &record, sizeof(record)) > 0) && (record.id != 0)) {
            printf("학번:%8d\t 이름:%4s\t 점수:%4d\n",
                record.id, record.name, record.score);
            printf("새로운 점수: ");      // 학번 이름은 변경 없이 점수만 수정 하자
            scanf("%d", &record.score);
            lseek(fd, -sizeof(record), SEEK_CUR); //현재 위치에서 전 레코드로 이동
            write(fd, (char *) &record, sizeof(record));
        } else printf("레코드 %d 없음\n", id);
    } else printf("입력오류\n");
    printf("계속하겠습니까?(Y/N)");
    scanf(" %c",&c); %d에 있는 엔터를 scanf가 가져감
} while (c == 'Y');
close(fd);
exit(0);
}
```

핵심 개념

- **시스템 호출**은 커널에 서비스를 요청하기 위한 프로그래밍 인터페이스로 응용 프로그램은 시스템 호출을 통해서 커널에 서비스를 요청할 수 있다.
- **파일 디스크립터**는 열린 파일을 나타낸다.
- `open()` 시스템 호출을 파일을 열고 열린 파일의 파일 디스크립터를 반환한다.
- `read()` 시스템 호출은 지정된 파일에서 원하는 만큼의 데이터를 읽고 `write()` 시스템 호출은 지정된 파일에 원하는 만큼의 데이터를 쓴다.
- 파일 위치 포인터는 파일 내에 읽거나 쓸 위치인 현재 파일 위치를 가리킨다.
- `lseek()` 시스템 호출은 지정된 파일의 현재 파일 위치를 원하는 위치로 이동시킨다.