


## 2. 파일 디스크립터와 저수준 입출력

### 2.1 파일 디스크립터란?

#### 정의

파일 디스크립터(File Descriptor, FD)는 리눅스에서 열린 파일, 소켓, 파이프, 디바이스, 터미널 등 시스템 자원에 접근하기 위한 정수 번호이다.

 즉, `int` 형 숫자 하나로 운영체제의 거의 모든 I/O 리소스를 제어할 수 있는 **핸들(Handle)** 역할.

#### 구조적으로 보면?

- 운영체제는 각 프로세스마다 **파일 디스크립터 테이블**을 유지한다.
- 파일을 열면 커널은 이 테이블에 항목을 추가하고, **인덱스 번호(int)**를 FD로 반환한다.

```
1 User Program
2   fd = open("file.txt", O_RDONLY);
3       ↓
4 Kernel
5   fd_table[fd] = → inode, position, mode ...
```

#### 기본적으로 예약된 FD 번호 3개

번호	의미	설명
0	<code>STDIN_FILENO</code>	표준 입력 (keyboard 등)
1	<code>STDOUT_FILENO</code>	표준 출력 (터미널 화면)
2	<code>STDERR_FILENO</code>	표준 에러 (에러 출력용)

```
1 write(1, "Hello\n", 6); // STDOUT
2 write(2, "Error\n", 6); // STDERR
```

#### C 코드 예제 ①: open → read → close

```
1 #include <fcntl.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 int main() {
6     char buffer[64];
7     int fd = open("test.txt", O_RDONLY); // 파일 오픈
8 }
```

```
9      if (fd == -1) {
10          perror("open");
11          return 1;
12      }
13
14      int n = read(fd, buffer, sizeof(buffer)); // 파일에서 읽기
15      write(1, buffer, n);                     // STDOUT에 출력
16
17      close(fd); // 파일 닫기
18      return 0;
19  }
```

🔍 흐름 설명

- `fd` 는 커널이 내부 테이블에 부여한 번호 (예: 3)
- 이 번호를 통해 커널은 어떤 파일, 어느 위치에서 읽어야 할지를 식별함

🧠 **FD는 단지 “파일만” 다루는 게 아니다**

FD가 참조하는 자원	예시
일반 파일	<code>open("a.txt", ...)</code>
디바이스 파일	<code>open("/dev/ttyUSB0", ...)</code>
파이프	<code>pipe(fd)</code>
소켓	<code>socket()</code> 의 반환값
터미널	STDIN/STDOUT/STDERR

⚠️ **FD와 FILE\*의 차이점**

항목	<code>int fd</code> (저수준)	<code>FILE* fp</code> (고수준)
정의	파일 디스크립터	C의 스트림 포인터
함수군	<code>open</code> , <code>read</code> , <code>write</code> , <code>lseek</code>	<code>fopen</code> , <code>fread</code> , <code>fprintf</code> , <code>fclose</code>
버퍼링	없음 (커널 버퍼만)	있음 (표준 C 라이브러리 내부 버퍼)
성능	빠름, 제어력 높음	사용 편리, 이식성 좋음

## 🔧 FD를 복제하거나 공유하는 방법

`dup`, `dup2`, `fcntl(F_DUPFD)` 등을 통해 FD를 복제할 수 있다.

```
1 int fd = open("log.txt", O_WRONLY);
2 int new_fd = dup(fd); // 동일한 파일에 대한 또 다른 FD
3
4 write(new_fd, "duplicate\n", 10); // 원본과 같은 동작
```

→ 셸에서 `2>&1` 같은 리디렉션도 결국 `dup2()` 로 구현됨

## 📁 시스템 콜 정리

함수	설명
<code>open()</code>	파일 열기 (→ FD 반환)
<code>read()</code>	FD에서 읽기
<code>write()</code>	FD에 쓰기
<code>close()</code>	FD 반납
<code>dup()</code> / <code>dup2()</code>	FD 복제
<code>lseek()</code>	파일 위치 조정
<code>fcntl()</code>	FD 설정/제어 (비동기, close-on-exec 등)

## ✅ 요약

핵심 포인트	설명
파일 디스크립터는 시스템 리소스에 대한 숫자 핸들	
프로세스마다 독립적인 FD 테이블을 갖는다	
FD는 파일, 소켓, 파이프, 디바이스 등 모든 I/O 자원에 사용된다	
고수준 스트림(FILE*)과 달리 버퍼링 없는 정밀한 제어가 가능하다	

## 2.2 `open()`, `read()`, `write()`, `close()` 함수

### 🔧 이 함수들은 무엇인가?

이 함수들은 리눅스의 저수준 파일 I/O 함수로서, 모두 `glibc`를 통해 시스템 콜로 연결되어 커널 내부에서 자원을 직접 제어하게 된다.

모두 `<unistd.h>` 또는 `<fcntl.h>` 에 선언되어 있다.

## 1. open()

### 헤더

```
1 #include <fcntl.h>
2 #include <sys/types.h>
3 #include <sys/stat.h>
```

### 함수 원형

```
1 int open(const char *pathname, int flags, mode_t mode);
```

파라미터	설명
pathname	열 파일 경로
flags	O_RDONLY, O_WRONLY, O_RDWR, O_CREAT 등
mode	(옵션) 파일 생성 시 퍼미션 (0666 등)

### 예제

```
1 int fd = open("test.txt", O_WRONLY | O_CREAT, 0644);
```

### 내부 흐름

```
1 → glibc open()
2 → syscall(SYS_openat, ...)
3 → 커널의 sys_openat() 호출
4 → do_filp_open() → dentry/inode 검색
```

---

## 2. read()

### 헤더

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

### 함수 원형

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

파라미터	설명
fd	읽을 대상의 파일 디스크립터
buf	데이터를 저장할 버퍼

파라미터	설명
<code>count</code>	읽을 바이트 수

### 예제

```
1 char buffer[128];
2 int n = read(fd, buffer, sizeof(buffer));
```

### 내부 흐름

```
1 → read() → syscall(SYS_read)
2   → sys_read()
3     → vfs_read() → file->f_op->read()
```

## 3. `write()`

### 헤더

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

### 함수 원형

```
1 ssize_t write(int fd, const void *buf, size_t count);
```

파라미터	설명
<code>fd</code>	출력 대상 (파일, STDOUT 등)
<code>buf</code>	출력할 데이터의 버퍼
<code>count</code>	바이트 수

### 예제

```
1 write(1, "Hello\n", 6); // STDOUT
```

### 내부 흐름

```
1 → write() → syscall(SYS_write)
2   → sys_write() → vfs_write() → file->f_op->write()
```

## 4. close()

### 헤더

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

### 함수 원형

```
1 | int close(int fd);
```

파라미터	설명
<code>fd</code>	닫을 파일 디스크립터

### 예제

```
1 | close(fd);
```

### 내부 흐름

```
1 | → close() → syscall(SYS_close)
2 |   → sys_close() → fput() → FD 테이블 정리 및 해제
```

## 전체 예제 (파일 복사기)

```
1 | #include <fcntl.h>
2 | #include <unistd.h>
3 | #include <stdio.h>
4 |
5 | int main() {
6 |     char buf[128];
7 |     int in = open("input.txt", O_RDONLY);
8 |     int out = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
9 |
10 |    if (in == -1 || out == -1) {
11 |        perror("open");
12 |        return 1;
13 |    }
14 |
15 |    int n;
16 |    while ((n = read(in, buf, sizeof(buf))) > 0) {
17 |        write(out, buf, n);
18 |    }
19 |
20 |    close(in);
21 |    close(out);
22 |    return 0;
}
```

## 📌 주요 오류 코드 (공통)

errno	의미
EACCES	권한 없음
ENOENT	파일 없음
EEXIST	파일이 이미 존재
EBADF	잘못된 파일 디스크립터
EFAULT	잘못된 주소 전달

## ✅ 요약

함수	역할	시스템 콜
open()	파일 열기/생성	sys_openat()
read()	데이터 읽기	sys_read()
write()	데이터 쓰기	sys_write()
close()	파일 디스크립터 정리	sys_close()

이 네 함수만 제대로 쓰면 `cat`, `cp`, `echo`, `ls` 등 모든 CLI 프로그램을 직접 구현할 수 있다.

## 2.3 표준 스트림과 리디렉션

### ◆ 1. 표준 스트림(Standard Streams)이란?

리눅스에서 모든 프로세스는 시작할 때 기본적으로 3개의 파일 디스크립터(FD)를 갖고 실행된다.

파일 디스크립터 번호	상수 이름	의미	기본 연결 대상
0	STDIN_FILENO	표준 입력	키보드
1	STDOUT_FILENO	표준 출력	터미널 화면
2	STDERR_FILENO	표준 에러 출력	터미널 화면

💡 `write(1, ..., ...)` → 터미널 출력

💡 `read(0, ..., ...)` → 키보드 입력

## C 코드 예제: 표준 출력, 에러 출력

```
1 #include <unistd.h>
2
3 int main() {
4     write(STDOUT_FILENO, "Hello, stdout!\n", 15);
5     write(STDERR_FILENO, "Hello, stderr!\n", 15);
6     return 0;
7 }
```

📄 실행하면 두 줄 모두 터미널에 출력되지만, `STDERR` 은 항상 즉시 출력됨 (비버퍼링)

## 2. 표준 리디렉션의 개념

리디렉션(Redirection)이란, 입출력의 흐름을 기본 대상(터미널)에서 파일/다른 장치로 바꾸는 것이다.

### 쉘 명령어에서의 리디렉션 예시

명령어	의미
<code>command &gt; out.txt</code>	표준 출력 → 파일
<code>command &lt; in.txt</code>	표준 입력 ← 파일
<code>command 2&gt; err.txt</code>	표준 에러 출력 → 파일
<code>command &gt; out.txt 2&gt;&amp;1</code>	표준 출력과 에러 모두 파일로

🔗 이 모든 건 실제로는 C 함수 `open()`, `dup2()`, `close()` 로 구현됨

## 3. 내부 구현 방식 (시스템콜 레벨)

`ls > out.txt` 실행 시 내부적으로:

```
1 int fd = open("out.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
2 dup2(fd, STDOUT_FILENO); // 표준출력을 해당 파일로 변경
3 execve("/bin/ls", ...); // ls 프로그램 실행
```

- `dup2()` 는 기존 FD(STDOUT)를 강제로 새 FD로 덮어씀
- 이후 `ls` 내부의 `write(1, ...)` 호출은 모두 `out.txt` 로 함함



## C 예제: `printf()` 결과를 파일로 리디렉션

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <unistd.h>
4
5 int main() {
6     int fd = open("log.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
7     dup2(fd, STDOUT_FILENO); // STDOUT 변경
8     printf("This goes to file!\n"); // log.txt에 저장됨
9     close(fd);
10    return 0;
11 }
```

## `dup()`, `dup2()` 핵심 비교

함수	설명
<code>dup(fd)</code>	사용 가능한 가장 작은 FD 번호를 반환
<code>dup2(oldfd, newfd)</code>	<code>newfd</code> 를 강제로 <code>oldfd</code> 와 동일하게 만들

```
1 int fd = open("out.txt", O_WRONLY);
2 dup2(fd, STDOUT_FILENO); // STDOUT을 out.txt로 연결
```

## `STDERR` 리디렉션까지 포함하기

```
1 command > out.txt 2> err.txt
```

→ FD 1 → `out.txt`, FD 2 → `err.txt`

```
1 command > out.txt 2>&1
```

→ FD 1, FD 2 모두 → `out.txt` (순서 중요)

## C 예제: `stdout` + `stderr` 리디렉션

```
1 #include <fcntl.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 int main() {
6     int out = open("log.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
7     dup2(out, STDOUT_FILENO);
8     dup2(out, STDERR_FILENO); // 에러 출력까지 같은 파일로
9
10    printf("This is stdout\n");
```

```

11     fprintf(stderr, "This is stderr\n");
12     close(out);
13     return 0;
14 }

```

## ⚠ stdout vs stderr 차이: 버퍼링

스트림	버퍼링 방식
stdout	줄 단위 (터미널), 전부 (파일)
stderr	무버퍼 / 즉시 출력

→ 디버깅용 로그는 `stderr` 로 출력하는 이유가 여기에 있음

## ✅ 요약

개념	설명
표준 스트림	FD 0, 1, 2: stdin, stdout, stderr
리디렉션	<code>dup2()</code> 로 FD를 다른 파일에 복제
셸 명령	내부적으로 <code>open</code> → <code>dup2</code> → <code>exec</code> 실행
stderr	비버퍼링되어 즉시 출력됨
시스템콜	<code>open</code> , <code>write</code> , <code>dup</code> , <code>execve</code> 등이 핵심

## 2.4 파일 모드 및 플래그

### 🔧 개요

`open()` 은 파일을 열기 위한 시스템 콜이며, 사용 시 `flags` 와 (선택적으로) `mode` 인자를 전달한다.

```

1 #include <fcntl.h>
2
3 int open(const char *pathname, int flags, mode_t mode);

```

### ◆ 1. flags: 어떻게 열지를 지정 (필수 인자)

#### 📌 접근 모드 (Access Mode) — 반드시 1개 필요

플래그	의미
<code>O_RDONLY</code>	읽기 전용으로 열기
<code>O_WRONLY</code>	쓰기 전용으로 열기

플래그	의미
<code>O_RDWR</code>	읽기 + 쓰기 모드로 열기

→ 위 세 가지 중 하나는 필수이며, 아래 추가 플래그와 **비트 OR ( | ) 연산**으로 결합 가능

## 🔴 부가 플래그 (옵션)

플래그	설명
<code>O_CREAT</code>	파일이 없으면 생성 ( <code>mode</code> 인자 필요)
<code>O_EXCL</code>	<code>O_CREAT</code> 와 함께 사용 → 이미 존재하면 실패
<code>O_TRUNC</code>	기존 파일 내용 모두 잘라냄 (길이 0으로)
<code>O_APPEND</code>	항상 파일 끝에 추가
<code>O_NONBLOCK</code>	블로킹되지 않도록 열기 (파이프, 소켓 등에서 사용)
<code>O_SYNC</code>	동기식 쓰기 (fsync처럼 디스크까지 쓰기 보장)
<code>O_DIRECTORY</code>	디렉터리만 열기 (디렉터리 아니면 오류)
<code>O_NOFOLLOW</code>	심볼릭 링크는 열지 않음
<code>O_CLOEXEC</code>	<code>exec()</code> 호출 시 자동으로 FD 닫기

## 🧪 예제: 플래그 조합

```
1 | int fd = open("data.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

📖 의미:

- 쓰기 전용
- 없으면 생성
- 있으면 내용을 비움 (truncate)

## ◆ 2. mode: 퍼미션을 어떻게 줄지 지정 (`O_CREAT` 시 필수)

`mode_t mode` 는 파일 생성 시의 접근 권한을 지정하며, 8진수로 표현하는 것이 일반적이다.

## 🔴 퍼미션 비트 구조

사용자	읽기	쓰기	실행	비트
사용자(owner)	r	w	x	6xx

사용자	읽기	쓰기	실행	비트
그룹(group)	r	w	x	x6x
기타(other)	r	w	x	xx6

## 📌 예시 모드값

모드	의미
0644	사용자: 읽기/쓰기, 그룹/기타: 읽기
0600	사용자만 읽기/쓰기 가능
0666	누구나 읽기/쓰기 (실행은 없음)
0755	사용자 rwx, 나머지 rx (실행 파일에 적합)

🔗 C 코드에서 쓸 때는 **0**을 접두로 붙여 8진수임을 표시해야 함

## 🧪 예제: 퍼미션 지정

```
1 | int fd = open("log.txt", O_WRONLY | O_CREAT, 0640);
```

→ 쓰기 전용으로 열되, 없으면 만들고

→ 퍼미션은 `rw-r-----` (owner에게 쓰기 가능, 그룹에게만 읽기 가능)

## 📦 고급 예제: 플래그 + 모드

```
1 | #include <fcntl.h>
2 | #include <unistd.h>
3 | #include <stdio.h>
4 |
5 | int main() {
6 |     int fd = open("example.txt",
7 |                  O_RDWR | O_CREAT | O_APPEND | O_CLOEXEC,
8 |                  0644);
9 |     if (fd == -1) {
10 |         perror("open");
11 |         return 1;
12 |     }
13 |
14 |     write(fd, "Log entry\n", 10);
15 |     close(fd);
16 |     return 0;
17 | }
```

- 읽기/쓰기

- 파일 끝에 항상 추가
- `exec()` 호출 시 자동으로 FD 닫기
- `rw-r--r--` 퍼미션

## 실습: `strace`로 플래그 확인

```
1 | strace ./a.out
```

출력 예:

```
1 | open("log.txt", O_WRONLY|O_CREAT|O_TRUNC, 0644) = 3
```

→ 실제로 어떤 플래그/모드가 커널로 전달됐는지 확인 가능

## 요약

구분	인자	역할
접근 방식	<code>O_RDONLY</code> , <code>O_WRONLY</code> , <code>O_RDWR</code>	읽기/쓰기 모드 지정 (필수)
추가 동작	<code>O_CREAT</code> , <code>O_APPEND</code> , <code>O_TRUNC</code> , <code>O_EXCL</code> 등	파일 생성/추가/삭제 동작 제어
퍼미션 설정	<code>mode_t</code> (8진수)	<code>0644</code> , <code>0600</code> 등 파일 권한 지정

## 2.5 `fcntl()`, `dup()`, `lseek()` 실습

### `fcntl()`, `dup()`, `lseek()` 개요

함수	주요 기능
<code>fcntl()</code>	파일 디스크립터의 속성/플래그 조작
<code>dup()</code> / <code>dup2()</code>	FD 복제 (리디렉션 구현 등에서 필수)
<code>lseek()</code>	파일 오프셋(읽기/쓰기 위치) 이동

## 1 `fcntl()`

### 헤더

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

## 📌 기본 원형

```
1 | int fcntl(int fd, int cmd, ... /* optional arg */);
```

## 📌 주요 cmd 종류

cmd	의미
<code>F_GETFL</code>	현재 FD의 플래그 조회
<code>F_SETFL</code>	FD 플래그 설정 (예: <code>O_NONBLOCK</code> 추가)
<code>F_GETFD</code>	FD 속성 조회 ( <code>FD_CLOEXEC</code> 여부 등)
<code>F_SETFD</code>	FD 속성 설정

## 🔧 예제: `O_APPEND` 적용

```
1 | int flags = fcntl(fd, F_GETFL);  
2 | flags |= O_APPEND;  
3 | fcntl(fd, F_SETFL, flags);
```

→ 이후 모든 `write(fd, ...)` 는 파일 끝에 자동으로 추가됨

## 📌 2 `dup()`, `dup2()`

### 📌 헤더

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

## 📌 기본 원형

```
1 | int dup(int oldfd);  
2 | int dup2(int oldfd, int newfd);
```

함수	기능
<code>dup()</code>	<code>oldfd</code> 를 복제하여 사용 가능한 가장 작은 FD 반환
<code>dup2()</code>	<code>newfd</code> 로 강제로 복제 (기존 FD 닫힘 후 재사용)

## 🔧 예제: 리디렉션 구현

```
1 | int fd = open("log.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);  
2 | dup2(fd, STDOUT_FILENO); // STDOUT → log.txt로 연결
```

→ 이후 `printf()` 호출은 모두 log.txt로 출력됨

### ◆ 3 lseek()

#### 📌 헤더

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

#### 📌 기본 원형

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

whence	의미
SEEK_SET	파일 처음 기준으로 이동
SEEK_CUR	현재 위치 기준으로 이동
SEEK_END	파일 끝 기준으로 이동

#### 🔔 주의

- lseek() 는 단순히 파일 오프셋(읽기/쓰기 위치)만 변경 → 데이터 이동 아님
- 파일 크기 초과 위치로 이동하면 sparse file(희소 파일) 생성 가능

#### 🧪 예제: 파일 끝으로 이동 후 쓰기

```
1 | lseek(fd, 0, SEEK_END);
2 | write(fd, "Extra data\n", 11);
```

### 🔧 종합 실습 예제

```
1 | #include <fcntl.h>
2 | #include <unistd.h>
3 | #include <stdio.h>
4 |
5 | int main() {
6 |     // 1 파일 열기
7 |     int fd = open("example.txt", O_RDWR | O_CREAT, 0644);
8 |
9 |     // 2 현재 플래그 확인 후 O_APPEND 추가
10 |    int flags = fcntl(fd, F_GETFL);
11 |    printf("Flags before: %x\n", flags);
12 |    flags |= O_APPEND;
13 |    fcntl(fd, F_SETFL, flags);
14 |    printf("Flags after O_APPEND added.\n");
15 |
16 |    // 3 파일 복제 (리디렉션용)
17 |    int fd_copy = dup(fd);
18 |    write(fd_copy, "Written via dup FD.\n", 20);
19 |}
```

```

20 // 4 파일 끝으로 이동 후 쓰기
21 lseek(fd, 0, SEEK_END);
22 write(fd, "Written at EOF.\n", 16);
23
24 // 5 마무리
25 close(fd);
26 close(fd_copy);
27
28 return 0;
29 }

```

실행 결과:

- `example.txt`에 2개의 다른 방식으로 내용이 기록됨
- 플래그 변경 확인 가능
- `dup()`로 복제된 FD도 동일 파일에 쓰기 가능

## ✓ 요약

함수	주요 기능	사용 예시
<code>fcntl()</code>	FD 플래그 설정/조회	<code>O_APPEND</code> 설정, <code>O_NONBLOCK</code> 적용
<code>dup()</code> , <code>dup2()</code>	FD 복제/리디렉션	<code>dup2()</code> 로 <code>stdout</code> 변경
<code>lseek()</code>	파일 오프셋 이동	<code>SEEK_SET/SEEK_CUR/SEEK_END</code> 사용

## 실습

### 파일 복사 프로그램 (`read` / `write`)

#### 목표

- 입력 파일 → `open()` → `fd_in`
- 출력 파일 → `open()` → `fd_out`
- `read(fd_in)` → 버퍼 → `write(fd_out)`
- `close(fd_in)`, `close(fd_out)`

#### 코드 예제: 파일 복사 프로그램

```

1 #include <fcntl.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 #define BUFFER_SIZE 4096 // 4KB 버퍼 사용

```



```

7
8 int main(int argc, char *argv[]) {
9     if (argc != 3) {
10         fprintf(stderr, "Usage: %s source_file dest_file\n", argv[0]);
11         return 1;
12     }
13
14     int fd_in = open(argv[1], O_RDONLY);
15     if (fd_in == -1) {
16         perror("open source");
17         return 1;
18     }
19
20     int fd_out = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
21     if (fd_out == -1) {
22         perror("open destination");
23         close(fd_in);
24         return 1;
25     }
26
27     char buffer[BUFFER_SIZE];
28     ssize_t bytes_read;
29
30     while ((bytes_read = read(fd_in, buffer, BUFFER_SIZE)) > 0) {
31         ssize_t bytes_written = write(fd_out, buffer, bytes_read);
32         if (bytes_written != bytes_read) {
33             perror("write");
34             close(fd_in);
35             close(fd_out);
36             return 1;
37         }
38     }
39
40     if (bytes_read == -1) {
41         perror("read");
42     }
43
44     close(fd_in);
45     close(fd_out);
46
47     return 0;
48 }

```

## 설명

단계	설명
<code>open(argv[1], O_RDONLY)</code>	원본 파일을 읽기 전용으로 열
<code>open(argv[2], O_WRONLY</code>	<code>O_CREAT</code>

단계	설명
<code>read(fd_in, buffer, BUFFER_SIZE)</code>	원본 파일에서 4KB씩 읽음
<code>write(fd_out, buffer, bytes_read)</code>	대상 파일에 같은 크기만큼 씀
<code>close(fd_in), close(fd_out)</code>	자원 정리

## ⚠ 에러 처리

- `open()` 실패 시 `perror()` 로 오류 출력
- `read()` 실패 시 오류 출력
- `write()` 시 썼는 바이트 수가 읽은 바이트 수와 달라지면 오류

→ 이 부분이 매우 중요: 시스템 콜은 항상 **partial read/write**가 가능하므로 체크 필요

## 🏃 실행 예

```
1 gcc copy.c -o copy
2 ./copy input.txt output.txt
```

→ `output.txt` 에 `input.txt` 내용이 그대로 복사됨.

## ✅ 핵심 개념 정리

시스템 콜	역할
<code>open()</code>	파일 열기 / 생성
<code>read()</code>	파일에서 데이터 읽기
<code>write()</code>	파일에 데이터 쓰기
<code>close()</code>	FD 정리 및 반환

## lseek으로 파일 오프셋 이동 구현

### 1. 개요

`lseek` 시스템 호출은 **열려 있는 파일 디스크립터의 현재 읽기/쓰기 오프셋을 변경**하는 기능을 제공한다.

파일 오프셋은 다음 읽기(`read`) 또는 쓰기(`write`) 작업이 수행될 **파일 내 위치**를 나타낸다.

`lseek` 를 사용하면 파일 내 임의의 위치로 이동하여 데이터를 읽거나 쓸 수 있으며, **희소 파일(sparse file)** 생성이나 파일 크기 확인 등 다양한 활용이 가능하다. 🚀

## 2. 함수 원형

```
1 #include <unistd.h>
2
3 off_t lseek(int fd, off_t offset, int whence);
```

- `fd` : 파일 디스크립터
- `offset` : 이동할 바이트 수
- `whence` : 기준 위치
  - `SEEK_SET` : 파일 시작
  - `SEEK_CUR` : 현재 위치
  - `SEEK_END` : 파일 끝

### 반환값

- 새 오프셋 위치(바이트 단위)
- 오류 시 `-1` 반환

## 3. 사용 목적

- 파일 내 특정 위치로 이동 후 `read` 또는 `write` 수행
- 파일 크기 확인
- 희소 파일 생성
- 파일 헤더 영역을 건너뛰고 데이터에 접근

## 4. 예제 프로그램

아래 예제는 `example.txt` 파일을 열어 **파일 시작에서 10바이트 앞으로 이동**한 후, 그 위치부터 데이터를 읽어 출력한다. 

```
1 #include <fcntl.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 #define BUF_SIZE 100
6
7 int main() {
8     int fd = open("example.txt", O_RDONLY);
9     if (fd == -1) {
10         perror("open");
11         return 1;
12     }
13
14     // 파일 시작에서 10바이트 앞으로 이동
15     off_t pos = lseek(fd, 10, SEEK_SET);
16     if (pos == -1) {
```

```

17     perror("lseek");
18     close(fd);
19     return 1;
20 }
21
22 // 데이터 읽기 및 출력
23 char buffer[BUF_SIZE];
24 ssize_t bytes_read = read(fd, buffer, BUF_SIZE - 1);
25 if (bytes_read == -1) {
26     perror("read");
27     close(fd);
28     return 1;
29 }
30
31 buffer[bytes_read] = '\0'; // 문자열 종료
32 printf("📄 Data after 10 bytes: %s\n", buffer);
33
34 close(fd);
35 return 0;
36 }

```

## 5. 실행 예시

```

1 $ ./lseek_demo
2 📄 Data after 10 bytes: ... (파일 내용 출력)

```

※ 출력 내용은 `example.txt` 의 실제 내용에 따라 달라진다.

## 6. 주의 사항 ⚠

- `lseek` 는 파이프, 소켓, 터미널에서는 사용 불가 → `ESPIPE` 오류 발생
- `lseek` 는 단순히 오프셋만 변경하며, 파일 내 데이터를 이동시키지 않는다
- `lseek` 이후 `read` 또는 `write` 호출 시 새 오프셋부터 동작한다

## 7. 희소 파일 생성 예제

```

1 #include <fcntl.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 int main() {
6     int fd = open("sparse.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
7     if (fd == -1) {
8         perror("open");
9         return 1;
10    }
11

```

```

12     write(fd, "START\n", 6); // 파일 처음에 쓰기
13
14     // 파일 끝에서 1MB 떨어진 곳으로 이동
15     off_t pos = lseek(fd, 1024 * 1024, SEEK_END);
16     if (pos == -1) {
17         perror("lseek");
18         close(fd);
19         return 1;
20     }
21
22     write(fd, "END\n", 4); // 1MB 떨어진 위치에 쓰기
23
24     close(fd);
25     return 0;
26 }

```

## 결과 확인

```

1 $ ls -lh sparse.txt
2 -rw-r--r-- 1 user user 1.0M+4B sparse.txt

```

→ 중간 영역은 디스크 공간을 차지하지 않는 희소 파일로 처리된다.

## 8. 결론

- `lseek`는 파일 내 임의 위치로 이동하여 효율적인 I/O 작업을 가능하게 한다.
- 다양한 파일 처리 기능 (파일 크기 확인, 희소 파일 생성 등)에 필수적으로 사용된다.
- C 언어로 시스템 수준의 파일 I/O를 구현할 때 매우 유용한 도구이다. 🚀

## dup2를 사용한 리디렉션 시뮬레이터

### 1. 개요

`dup2` 시스템 호출은 파일 디스크립터 복제 기능을 제공하며, 이를 통해 표준 입출력(0, 1, 2)을 다른 파일 디스크립터로 리디렉션할 수 있다.

셸에서 사용하는 다음과 같은 리디렉션 기능은 내부적으로 `dup2`로 구현된다.

```

1 command > out.txt 2> err.txt

```

`dup2(oldfd, newfd)`는 `newfd`가 가리키는 FD를 닫고, `oldfd`의 복제본을 그 위치에 할당한다.

결과적으로 프로그램의 **STDOUT/STDERR**가 원하는 파일로 바뀐다. 🚀

## 2. 함수 원형

```
1 #include <unistd.h>
2
3 int dup2(int oldfd, int newfd);
```

### 반환값

- 성공 시 **newfd** 반환
- 실패 시 **-1** 반환

## 3. 사용 목적

- **STDOUT** 리디렉션 → 표준 출력 결과를 파일에 저장
- **STDERR** 리디렉션 → 표준 에러 결과를 파일에 저장
- **STDIN** 리디렉션 → 파일로부터 표준 입력 받기

## 4. 예제 프로그램: 리디렉션 시뮬레이터

아래 예제는 `dup2` 를 사용하여 프로그램 실행 중 **STDOUT**을 **파일로 리디렉션**하는 시뮬레이터이다.

이를 통해 셸에서 > 연산자가 내부적으로 어떻게 동작하는지 이해할 수 있다. 📄

```
1 #include <fcntl.h>
2 #include <unistd.h>
3 #include <stdio.h>
4
5 int main() {
6     // 출력 파일 열기
7     int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
8     if (fd == -1) {
9         perror("open");
10        return 1;
11    }
12
13    // STDOUT(1)을 fd로 리디렉션
14    if (dup2(fd, STDOUT_FILENO) == -1) {
15        perror("dup2");
16        close(fd);
17        return 1;
18    }
19
20    // 이제부터 printf, puts, write(1, ...)는 모두 output.txt로 출력됨
21    printf("📄 This message is redirected to output.txt!\n");
22
23    // 명시적 close 필요 (fd는 사용 후 반드시 정리)
24    close(fd);
25
26    return 0;
```

## 5. 실행 결과 확인

```
1 $ ./dup2_redirect
2 $ cat output.txt
3 📄 This message is redirected to output.txt!
```

→ 리디렉션 성공 🎉

→ 터미널에는 출력이 나타나지 않고, 파일에 저장됨.

## 6. 원리

```
1 open("output.txt") → fd = 3 (예시)
2 dup2(3, 1) → STDOUT_FILENO(1)이 fd 3과 동일한 리소스를 가리키게 됨
3 printf() → write(1, ...) 호출 → 실제로는 output.txt에 기록됨
```

**핵심** → 표준 출력이 파일로 투명하게 연결된다. 이후의 모든 표준 출력 함수는 파일로 기록된다.

## 7. 고급 확장 예시

### STDOUT + STDERR 동시 리디렉션

```
1 int fd = open("all_output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
2 dup2(fd, STDOUT_FILENO);
3 dup2(fd, STDERR_FILENO);
4
5 printf("This goes to STDOUT\n");
6 fprintf(stderr, "This goes to STDERR\n");
7
8 close(fd);
```

- 1(STDOUT)과 2(STDERR)를 같은 파일로 연결 → 쉘의 `command > out.txt 2>&1` 와 동일한 동작.

## 8. 주의사항 ⚠

- `dup2` 는 리디렉션 후 프로그램 실행 중에도 지속적으로 영향을 미친다.
- `dup2` 는 내부적으로 `close(newfd) → dup(oldfd) → newfd = oldfd` 복제 순서로 작동한다.
- 복제한 후 원래 파일 디스크립터(`fd`)는 반드시 `close()` 호출로 정리해야 한다.

## 9. 결론

`dup2`는 파일 디스크립터 리디렉션의 핵심 도구이며, C 언어에서 시스템 수준으로 STDIN, STDOUT, STDERR 흐름을 자유롭게 제어할 수 있게 한다.

셸의 리디렉션 기능은 본질적으로 `dup2`를 기반으로 작동한다.

리디렉션, 로깅, 입출력 제어, 데몬 프로세스 설계 등에서 매우 중요하게 활용된다. 🛠️