

1. 운영체제와 C 언어의 관계

1.1 운영체제란 무엇인가?

운영체제(Operating System, OS)는 하드웨어와 사용자 프로그램 사이의 중재자이며, 컴퓨터 시스템 자원을 효율적으로 관리하고 사용자에게 편리한 환경을 제공하는 소프트웨어다.

🔧 운영체제의 정의

운영체제란?

사용자와 하드웨어 사이에서 자원(Resource)을 관리하고, 프로세스가 안전하고 효율적으로 실행될 수 있도록 지원하는 소프트웨어 플랫폼이다.

운영체제가 없다면 사용자 프로그램은 직접 하드웨어를 제어해야 하고, 자원 충돌과 안전 문제를 일으킬 수 있다. OS는 이러한 문제를 추상화 계층을 통해 해결한다.

✳ 운영체제가 하는 주요 역할

역할	설명
프로세스 관리	실행 중인 프로그램(프로세스)을 생성, 스케줄링, 종료하며 CPU를 적절히 분배함
메모리 관리	프로그램이 사용하는 메모리를 할당/해제하고 주소 공간을 관리함
파일 시스템 관리	디스크에 파일을 저장하고 디렉터리 구조, 접근 권한 등을 관리함
입출력(I/O) 관리	다양한 장치(디스크, 프린터 등)의 접근을 추상화하여 드라이버를 통해 통제함
보안 및 권한 관리	사용자 간 자원 접근을 통제하고 보안 정책을 강제함
시스템 콜 제공	사용자 프로그램이 커널 기능을 요청할 수 있도록 인터페이스를 제공함

🏗 운영체제의 구성 계층

```
1  [사용자(User)]
2    ↑
3  [응용 프로그램]
4    ↑ ← 시스템 콜
5  [운영체제 커널(Kernel)]
6    ↑
7  [하드웨어(Hardware)]
```

- **커널(Kernel)**: 운영체제의 핵심. 메모리, CPU, 장치 제어 등의 핵심 기능 수행
- **시스템 콜(System Call)**: 사용자 프로그램이 커널의 기능을 호출하는 방법 (C로 `write()`, `fork()` 등)
- **셸(Shell)**: 사용자와 커널 사이의 CLI 인터페이스 (bash, zsh 등)

운영체제 예시

운영체제	설명
Linux	오픈소스 UNIX 계열 OS. 서버, 임베디드, 슈퍼컴퓨터 등 광범위하게 사용
Windows	GUI 중심의 상용 운영체제. 사용자 친화적이지만 폐쇄적
macOS	UNIX 기반의 Apple 운영체제. GUI + POSIX 호환
FreeRTOS	마이크로컨트롤러용 실시간 운영체제 (RTOS)
Android	리눅스 커널 기반의 모바일 운영체제

운영체제와 C 언어의 관계

- 리눅스와 유닉스는 대부분 **C 언어로 구현**
- 시스템 호출도 C 언어로 인터페이스 제공 (`#include <unistd.h>`)
- 커널/드라이버 모듈도 C로 작성 (일부는 어셈블리 포함)
- 운영체제를 깊이 이해하려면 C를 반드시 알아야 함

보충: 운영체제가 꼭 필요한 이유

- 프로세스 간 충돌 방지
- 자원 낭비 방지 및 공정한 분배
- 하드웨어 제어의 통일성 확보 (장치 독립성)
- 시스템 보안 및 안정성 강화
- 프로그래머의 생산성 향상 (고급 API 제공)

1.2 리눅스 운영체제의 구조

리눅스 운영체제는 어떤 구조인가?

리눅스는 계층적이고 모듈화된 구조를 가진 운영체제이며, 모놀리식 커널(monolithic kernel) 구조에 기반하고 있다.

운영체제 구조를 위에서 아래로 정리하면 다음과 같다:



리눅스 커널의 핵심 구성 요소

리눅스 커널은 하나의 큰 덩어리가 아닌, 아래와 같은 **모듈형 핵심 컴포넌트들로** 구성되어 있어.

구성 요소	역할
프로세스 관리자	프로세스 생성, 제거, 스케줄링, 문맥 교환
메모리 관리자	가상 메모리, 페이지 관리, 힙/스택 구조
파일 시스템	파일 및 디렉토리 구조 관리, VFS
장치 드라이버	하드웨어와의 인터페이스를 모듈화
네트워크 스택	TCP/IP 등 네트워크 통신 계층
시스템 호출 인터페이스	유저 공간에서 커널 기능 요청 시 사용
보안/권한 관리	유저/그룹, 퍼미션, capability 등

커널 공간 vs 사용자 공간

항목	사용자 공간 (User Space)	커널 공간 (Kernel Space)
접근 범위	제한적, 보호된 API 사용	전체 하드웨어 제어 가능
언어	C, C++, Python 등	거의 C (일부 어셈블리)
예시	bash, gcc, vim, etc.	스케줄러, VFS, device driver
전이 방식	<code>syscall</code> , <code>trap</code>	—

예: `write()` 는 유저 공간에서 커널 공간으로 시스템 호출을 통해 전이됨.

✖ 시스템 콜 흐름

사용자가 `write(1, "hello", 5)` 를 호출하면 다음과 같은 흐름을 가짐:

1. `glibc` 라이브러리 함수 → `syscall` 번호 확인
2. 어셈블리 `int 0x80` 또는 `syscall` 명령 실행
3. 커널에서 해당 시스템 콜 핸들러로 진입
4. 파일 디스크립터 테이블 참조, `vfs_write()` 수행
5. 하드웨어 드라이버 호출 → 디스크에 기록
6. 결과값 리턴 → 유저 공간으로 복귀

✖ 모놀리식 커널 vs 마이크로커널 (구조 비교)

구조	설명	예시
모놀리식 커널	하나의 커다란 바이너리로 모든 기능이 포함됨	Linux, BSD
마이크로커널	최소한의 커널만 있고, 나머지는 유저 공간에서 실행	Minix, QNX

🔗 리눅스는 모놀리식이지만, 로드 가능한 커널 모듈 (LKM)을 통해 유연한 확장성 확보

🔧 C 언어 기반 시스템 접근 예시

- 시스템 콜 레벨 접근: `open()`, `fork()`, `ioctl()`
- `/proc`, `/sys` 파일시스템 분석: `cat /proc/cpuinfo`, `stat()`
- 직접 커널 컴파일 후, 기능 수정 및 `insmod` 로 커널 모듈 삽입
- 파일 시스템 레이어 접근: `fopen()` vs `open()` 차이 비교

📦 커널 빌드와 모듈 예시

```
1 // hello_module.c
2 #include <linux/module.h>
3 #include <linux/init.h>
4
5 static int __init hello_init(void) {
6     printk(KERN_INFO "Hello from kernel\n");
7     return 0;
8 }
9
10 static void __exit hello_exit(void) {
11     printk(KERN_INFO "Goodbye from kernel\n");
12 }
13
14 module_init(hello_init);
15 module_exit(hello_exit);
```

```
16 | MODULE_LICENSE("GPL");
17 | bash코드 복사make
18 | sudo insmod hello_module.ko
19 | dmesg | tail
```

정리

- 리눅스는 모놀리식 커널 구조를 가짐
- 모든 기능(메모리, 스케줄링, 파일 등)이 커널 내부에 있으며, C 언어로 구현됨
- 사용자 프로그램은 시스템 콜을 통해 커널 기능을 사용
- C 언어는 커널과 유저 공간을 연결하는 가장 중요한 도구

1.3 사용자 공간 vs 커널 공간

✖ 기본 개념 요약

항목	사용자 공간 (User Space)	커널 공간 (Kernel Space)
정의	일반 애플리케이션이 실행되는 메모리 영역	OS의 핵심 기능이 실행되는 보호된 영역
접근 권한	제한적, 간접적 접근	하드웨어 및 시스템 리소스 직접 제어 가능
예시	gcc, vim, bash, 사용자 작성 C 프로그램 등	커널, 모듈, 드라이버, 시스템콜
접근 방법	시스템 콜, 라이브러리 호출 (glibc)	인터럽트, 직접 제어, 하드웨어 접근

📌 왜 나뉘는가? (보호와 안정성 목적)

운영체제가 시스템 자원의 보호와 안정성 확보를 위해, 사용자 프로그램이 커널의 내부에 직접 접근하지 못하도록 설계한 것.

- 프로세스가 오작동하더라도 시스템 전체가 다운되지 않게 함
- 커널은 보호 모드(Privileged Mode)에서 동작
- C 언어로도 커널에 직접 접근하려면 시스템 콜을 거쳐야 함

📌 메모리 구조 관점에서

🔗 32bit 시스템의 일반적 구조

```
1 | [0x00000000 ----- 0xBFFFFFFF] → 사용자 공간 (약 3GB)
2 | [0xC0000000 ----- 0xFFFFFFFF] → 커널 공간 (약 1GB)
```

🔧 64bit 시스템에서는 가상 메모리 공간을 훨씬 넓게 확보하며 비슷한 구조 유지

🔍 `cat /proc/self/maps` 명령으로 현재 프로세스의 메모리 맵 확인 가능

🔧 시스템 호출(System Call)

사용자 공간에서 커널 기능을 사용하려면 시스템 콜을 통해 전이해야 한다.

🔄 흐름

1. 사용자 공간 프로그램: `write(1, "hi", 2)`
2. `glibc`는 해당 `syscall` 번호(예: `__NR_write`)를 사용해 `syscall` 명령어 실행
3. 커널 모드로 전환 (트랩, `int 0x80` or `syscall`)
4. `sys_write()` 커널 함수 실행
5. 작업 완료 후 유저 공간으로 복귀

👤 C 코드 예시

```
1 #include <unistd.h>
2 #include <sys/syscall.h>
3
4 int main() {
5     syscall(SYS_write, 1, "hello\n", 6); // 직접 syscall 호출
6     return 0;
7 }
```

📁 파일 시스템 인터페이스

- 사용자 공간: `fopen()`, `fread()` → C 라이브러리(`glibc`) 함수
- 커널 공간: `sys_open()`, `sys_read()` → 커널 내 시스템 콜 함수

→ 실제 파일 접근은 커널이 담당하고, 유저 프로그램은 단지 요청할 뿐

🔒 권한 및 예외 처리

- 유저 공간에서 잘못된 포인터 참조 → `Segmentation Fault` (SIGSEGV)
- 커널 공간에서 발생하면 → 커널 패닉(Panic), 시스템 전체 다운 가능
- 그래서 커널 공간에서 메모리 접근은 매우 신중하게 설계

커널 모듈로 확인하는 사용자 공간/커널 공간

dmesg 로 확인 가능한 커널 로그

```
1 // hello_kernel.c
2 #include <linux/module.h>
3 #include <linux/init.h>
4
5 static int __init hello_init(void) {
6     printk(KERN_INFO "Hello from kernel space\n");
7     return 0;
8 }
9
10 static void __exit hello_exit(void) {
11     printk(KERN_INFO "Goodbye from kernel space\n");
12     return;
13 }
14
15 module_init(hello_init);
16 module_exit(hello_exit);
17 MODULE_LICENSE("GPL");
```

```
1 make
2 sudo insmod hello_kernel.ko
3 dmesg | tail
```

커널 공간에서 실행되는 코드는 `printk()` 로 로그를 남기고, `dmesg` 명령으로 확인할 수 있어.

사용자 공간 ↔ 커널 공간 간 인터페이스 요약

방식	설명
System Call	표준적인 접근 방법 (<code>write</code> , <code>fork</code> , <code>exec</code> , etc)
<code>/proc</code> , <code>/sys</code> 파일시스템	읽기 전용 정보 또는 제한된 설정
<code>ioctl()</code>	디바이스에 대한 커맨드 기반 제어
Netlink Socket	유저 공간과 커널의 메시지 교환 채널
Shared Memory (<code>mmap</code>)	성능 최적화를 위한 일부 공유 영역

정리

- 사용자 공간은 안전하고 제한된 환경
- 커널 공간은 강력하고 위험한 환경
- 두 공간은 시스템 콜, `/proc`, `ioctl`, `mmap` 등을 통해 연결됨
- 리눅스는 이 두 영역을 철저히 분리하여 안정성과 보안을 확보함

1.4 C 언어가 운영체제에 적합한 이유

🏰 1. 역사적 배경

- 1960~70년대 초, UNIX는 어셈블리 언어로 개발
- 유지보수가 어려워지자 **Dennis Ritchie**가 C 언어를 개발하고, UNIX를 C로 다시 작성
- 이후 C 언어는 운영체제를 구현하기 위한 사실상 표준 언어가 됨

🔥 대부분의 현대 운영체제(Linux, BSD, Windows NT 커널 등)는 C 언어로 구현됨

🔧 2. 하드웨어 접근이 가능한 저수준 언어

- C는 고급 언어이지만, 어셈블리 언어에 매우 가까운 수준에서 메모리 주소, 포인터, 비트 연산 등 하드웨어 자원을 직접 제어 가능

```
1 // 포인터 연산을 통한 메모리 직접 접근
2 int* ptr = (int*)0xB8000;
3 *ptr = 0x12345678;
```

- 커널이나 장치 드라이버 코드는 직접 메모리 맵을 조작해야 하기 때문에 이런 기능이 필수적

⚙️ 3. 시스템 자원에 대한 정밀한 제어

- malloc/free, brk/sbrk 등으로 동적 메모리 할당 직접 관리
- 포인터 연산을 통해 스택, 힙, 커널 구조체 직접 탐색 가능
- `volatile`, `register`, `restrict` 키워드를 통해 하드웨어와의 직접 인터페이스 최적화

🔑 예: 하드웨어 레지스터를 직접 조작할 수 있는 C의 장점

```
1 #define UART0_DR (*(volatile unsigned int *)0x101f1000)
2 UART0_DR = 'A'; // 문자 전송
```

⚡ 4. 빠르고 가벼운 실행 성능


- C 언어는 컴파일 후 생성되는 바이너리가 작고 빠름
- 메모리 접근, 함수 호출, 반복 구조 등이 어셈블리 수준에 가깝게 동작
- 커널은 컨텍스트 스위칭, 인터럽트 핸들링, 스케줄링 등을 실시간으로 처리해야 하므로 느린 언어는 부적합

🏆 5. 포터블(Portable)한 코드 구조

- 운영체제는 다양한 아키텍처(CPU, 메모리, 장치)에서 돌아가야 함
- 어셈블리는 아키텍처 종속이지만, C는 대부분의 CPU에 맞춰 재컴파일만 하면 동작 가능
- 리눅스는 x86, ARM, RISC-V 등 다양한 플랫폼에 이식되어 있음 (모두 C 기반 덕분)

6. 커널과의 자연스러운 연계성

- 커널 함수(`sys_open`, `do_fork`, `vfs_read`)들은 모두 C로 작성됨
- 시스템 호출 구현도 C 함수에 연결됨

 예: 유저 공간에서 `write()` → 커널의 `sys_write()` 함수 호출

7. C는 운영체제 실습 및 연구에 최적화된 언어

- 커널 모듈, 장치 드라이버, syscall 추가, memory allocator 등 운영체제 각 요소를 C로 직접 구현 가능
- 교육용 운영체제(Minix, xv6, LittleOS)도 C로 제작됨
- 시스템 해킹, 보안 분석, 커널 디버깅 도구들도 C 기반으로 설계됨

운영체제에서 C 언어가 쓰이는 실제 구조 예시

OS 컴포넌트	C에서 구현 예시
프로세스 관리	<code>do_fork()</code> , <code>schedule()</code>
메모리 관리	<code>kmalloc()</code> , <code>vm_area_struct</code>
파일 시스템	<code>vfs_read()</code> , <code>vfs_open()</code>
시스템 콜	<code>SYSCALL_DEFINE3(write, ...)</code>
드라이버	<code>platform_driver_register()</code>
모듈 인터페이스	<code>module_init()</code> , <code>module_exit()</code>

반례: 왜 C++이나 Java는 운영체제에 부적합한가?

언어	이유
C++	런타임 오버헤드, 복잡한 컴파일, 가상함수 등으로 예측 불가
Java	VM 기반으로 커널 레벨 접근 불가, GC의 비결정성
Python 등	해석 언어이므로 커널 수준 속도와 제어력 부족

결론 요약

- C 언어는 하드웨어 접근력 + 효율성 + 이식성 + 최소한의 런타임을 동시에 갖춘
- 운영체제 개발에 필요한 정확성, 속도, 제어력, 유지보수성을 이상적으로 만족
- 리눅스, Windows, macOS 등 거의 모든 OS의 커널이 C 기반으로 작성된 이유

1.5 C 언어로 시스템 리소스를 제어하는 구조 이해

✂ 기본 개념: C → 시스템 리소스까지의 흐름



- C 코드는 단순히 `fopen`, `malloc`, `write` 등을 호출하지만
- 이 함수들은 결국 시스템 콜을 통해 **운영체제 커널에 요청**을 보내고,
- 커널이 하드웨어 자원이나 시스템 자원을 **직접 제어**

⚙ 제어 가능한 시스템 리소스 종류

자원 종류	C 언어 접근 방식 (함수/시스템 콜)
CPU	<code>fork()</code> , <code>exec()</code> , <code>sched_*</code>
메모리	<code>malloc()</code> , <code>mmap()</code> , <code>brk()</code>
파일/디렉토리	<code>open()</code> , <code>read()</code> , <code>write()</code> , <code>stat()</code>
디바이스	<code>ioctl()</code> , <code>/dev</code> 파일, <code>mmap()</code>
네트워크	<code>socket()</code> , <code>bind()</code> , <code>send()</code> , <code>recv()</code>
프로세스 정보	<code>/proc</code> 탐색, <code>getpid()</code> , <code>kill()</code>
시간	<code>time()</code> , <code>gettimeofday()</code> , <code>nanosleep()</code>
시그널	<code>signal()</code> , <code>sigaction()</code> , <code>alarm()</code>

🔧 예제 1: 파일 시스템 제어 (open, read, write)

```
1 #include <fcntl.h>
2 #include <unistd.h>
3
4 int main() {
5     int fd = open("hello.txt", O_WRONLY | O_CREAT, 0644);
6     write(fd, "Hello from C!\n", 14);
7     close(fd);
8     return 0;
9 }
```

🔍 설명

- `open()` 은 시스템 콜이며 커널의 `sys_open()` 호출
- 파일 디스크립터 `fd` 는 커널이 관리하는 테이블의 인덱스
- `write()` 호출 시, 커널이 디스크 드라이버를 호출하여 실제 쓰기 수행

🧠 예제 2: 메모리 직접 제어 (brk, sbrk)

```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 int main() {
5     void* cur = sbrk(0); // 현재 브레이크 포인터
6     printf("Heap before: %p\n", cur);
7     sbrk(4096); // 힙 공간 4KB 증가
8     printf("Heap after : %p\n", sbrk(0));
9     return 0;
10 }
```

🔍 설명

- `sbrk()` 는 힙 공간을 직접 확장하는 저수준 메모리 인터페이스
- 커널은 프로세스의 가상 메모리 맵을 수정하고 새로운 페이지를 할당

⚙️ 예제 3: CPU 스케줄 제어 (fork, nice)

```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 int main() {
5     pid_t pid = fork();
6
7     if (pid == 0) {
8         nice(10); // 자식의 우선순위 낮추기
9         for (int i = 0; i < 5; i++) {
```

```

10         printf("Child process\n");
11         sleep(1);
12     }
13     } else {
14         for (int i = 0; i < 5; i++) {
15             printf("Parent process\n");
16             sleep(1);
17         }
18     }
19     return 0;
20 }

```

🔍 설명

- `fork()` 는 새로운 프로세스를 만들어 **CPU 자원 분기**
- `nice()` 는 해당 프로세스의 **우선순위를 조정** (낮을수록 우선)

🧠 예제 4: 디바이스 제어 (`/dev/random` 사용)

```

1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <stdio.h>
4
5  int main() {
6      int fd = open("/dev/random", O_RDONLY);
7      int r;
8      read(fd, &r, sizeof(int));
9      printf("Random value: %d\n", r);
10     close(fd);
11     return 0;
12 }

```

🔍 설명

- `/dev/random` 은 커널이 제공하는 가상 장치 파일
- `read` 호출 시 커널이 랜덤 디바이스 드라이버에서 값을 추출

📌 구조 요약: 시스템 리소스를 C로 제어하는 흐름

단계	설명
C 함수 호출	ex: <code>write(fd, buf, size)</code>
시스템 콜 인터페이스	커널로 전이 (syscall, trap)
커널 syscall 함수	ex: <code>sys_write()</code>
커널 서브시스템	파일 시스템, 메모리, 스케줄러 등

단계	설명
하드웨어/가상 장치	드라이버 또는 가상 인터페이스

고급 접근 방법

방법	목적
<code>mmap()</code>	디바이스, 파일, 메모리 매핑
<code>ioctl()</code>	장치에 직접 명령어 전송
<code>ptrace()</code>	디버깅 및 프로세스 감시
<code>perf_event_open()</code>	성능 측정 인터페이스
<code>/proc</code> 접근	실시간 자원 상태 확인

정리

- C는 리눅스 시스템 리소스를 제어하기 위한 **최적의 저수준 언어**
- 시스템 콜을 통해 **메모리, CPU, 파일, 장치, 네트워크**를 통제 가능
- 커널은 C 호출을 받아 **자원을 직접 관리하거나 하드웨어에 명령 전달**

실습

Hello Kernel: `printf("Hello from Linux\n")`와 `syscall` 분석

1 겉으로 보기엔 단순한 한 줄

```

1 #include <stdio.h>
2
3 int main() {
4     printf("Hello from Linux\n");
5     return 0;
6 }
```

 겉보기엔 그냥 터미널에 문자열을 출력하는 함수지만...

2 실제 내부 흐름은 훨씬 깊음

✓ `printf()` → `write()` → 시스템 콜 → 커널 → 터미널로 출력

```
1  +-----+
2  | C 표준 라이브러리      |
3  | printf("Hello...\n")   |
4  +-----+
5      ↓
6  +-----+
7  | glibc 내부의 printf 구현 |
8  | → write(1, "Hello...", 17) 호출 |
9  +-----+
10     ↓
11  +-----+
12  | 시스템 콜 인터페이스   |
13  | → syscall(SYS_write, ...) |
14  +-----+
15     ↓
16  +-----+
17  | 커널 공간              |
18  | sys_write() → vfs_write() 등 |
19  +-----+
20     ↓
21  +-----+
22  | 파일 디스크립터 #1 → 터미널 |
23  +-----+
```

3 코드 예제: `printf()` 없이 `syscall`로 직접 출력

```
1  #define _GNU_SOURCE
2  #include <unistd.h>
3  #include <sys/syscall.h>
4
5  int main() {
6      const char* msg = "Hello from Linux\n";
7      syscall(SYS_write, 1, msg, 17); // 1 = STDOUT
8      return 0;
9  }
```

🔍 설명

- `syscall(SYS_write, ...)` 는 glibc를 우회하고 리눅스 커널에 직접 write 요청
- 1 은 파일 디스크립터(STDOUT), `msg` 는 출력 문자열, 17 은 바이트 수
- 내부적으로는 x86_64의 경우 `syscall` 어셈블리 명령이 호출됨

4 시스템 콜 번호 확인 방법

```
1 man syscall # 시스템 콜 개요
2 man 2 write # write() 시스템 콜의 매뉴얼
3
4 # 시스템 콜 번호 확인
5 grep __NR_write /usr/include/asm-generic/unistd.h
```

- x86_64 기준: `__NR_write = 1`
- ARM이나 다른 아키텍처는 다름

5 커널 내부에서는 어떤 함수가 실행될까?

유저 공간에서 `write()` → 커널 공간의 `sys_write()` 로 연결됨

```
1 // 커널 소스 일부 예시 (fs/read_write.c)
2
3 SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf, size_t, count)
4 {
5     return ksys_write(fd, buf, count);
6 }
```

그리고 `ksys_write()` 는 다시 `vfs_write()` → 파일 시스템 → 드라이버 레벨로 전달됨

6 커널 레벨 디버깅: syscall 트레이스

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

👁 출력 예시:

```
1 write(1, "Hello from Linux\n", 17) = 17
```

- 유저 공간 프로그램에서 실제 어떤 시스템 콜이 발생했는지 보여줌
- `strace` 는 커널 진입 지점을 추적하는 데 매우 유용

7 커널의 시점에서 보면?

```
1 // 내부 커널 함수 스택 구조
2 printf → write
3 → syscall(SYS_write, ...)
4 → sys_write(fd, buf, count)
5 → vfs_write()
6 → file->f_op->write()
7 → tty_write() (터미널일 경우)
```

🔥 터미널은 문자 디바이스 드라이버이며, `/dev/pts/0` 같은 파일로 매핑됨

✅ 요약: `printf("Hello")`가 커널을 통해 실제 동작하는 흐름

단계	설명
1. 유저 프로그램	<code>printf()</code> 호출
2. glibc	내부에서 <code>write()</code> 호출
3. syscall	커널에 <code>sys_write</code> 호출
4. 커널 진입	<code>sys_write()</code> → <code>vfs_write()</code>
5. 파일 시스템	STDOUT → 터미널 장치
6. 드라이버	터미널 디바이스에 문자 출력

📦 보너스: 직접 syscall 어셈블리로 호출해보기 (x86_64)

```
1 #include <unistd.h>
2
3 int main() {
4     const char* msg = "Hello from Linux\n";
5     asm volatile (
6         "mov $1, %%rax\n"          // syscall 번호: write
7         "mov $1, %%rdi\n"          // STDOUT
8         "mov %0, %%rsi\n"          // 문자열 주소
9         "mov $17, %%rdx\n"         // 바이트 수
10        "syscall\n"
11        :
12        : "r"(msg)
13        : "%rax", "%rdi", "%rsi", "%rdx"
14    );
15    return 0;
16 }
```

🔥 위 코드는 `syscall` 명령어로 직접 `write` syscall을 호출함

man 2를 활용한 리눅스 시스템콜 문서 확인

🔍 man이란?

`man`은 "manual", 즉 리눅스 명령어, 함수, 시스템 콜, 파일 형식 등을 설명하는 도움말 시스템이다.

```
1 | man [섹션 번호] [명령어 또는 함수 이름]
```


1234 섹션 번호 의미

번호	설명
1	사용자 명령어 (ls, cp 등)
2	시스템 콜 (write, fork 등)
3	C 라이브러리 함수 (printf, malloc 등)
4	디바이스 파일 (/dev 등)
5	파일 포맷 (/etc/passwd, crontab 등)
7	Misc (매뉴얼, 네트워크 프로토콜 등)
9	커널 개발자용 문서 (LKM, 드라이버 등)

👉 우리가 보는 건 시스템 콜이므로 항상 `man 2` 야.

✅ 사용법 예제

✏️ `write()` 시스템 콜 문서 확인

```
1 | man 2 write
```

■ 출력 내용 (요약):

```
1 | NAME
2 |     write - write to a file descriptor
3 |
4 | SYNOPSIS
5 |     #include <unistd.h>
6 |     ssize_t write(int fd, const void *buf, size_t count);
7 |
8 | DESCRIPTION
9 |     write() writes up to count bytes from the buffer starting at buf
10 |    to the file referred to by the file descriptor fd.
11 |
12 | RETURN VALUE
13 |     On success, the number of bytes written is returned. On error, -1 is returned, and
14 |    errno is set appropriately.
15 |
16 | ERRORS
17 |     EAGAIN, EBADF, EFAULT, EINTR, EINVAL, EIO, ENOSPC, ...
```

🔴 여기서 확인 가능한 정보:

항목	의미
<code>#include</code>	필요한 헤더 (<code>unistd.h</code>)
<code>ssize_t write(...)</code>	함수 원형 및 타입
<code>fd</code>	파일 디스크립터 (예: 1은 <code>stdout</code>)
<code>buf</code>	출력할 버퍼 주소
<code>count</code>	바이트 수
<code>errno</code>	실패 시 원인을 나타내는 전역 변수

실전 사용 예시

1. `read()` 시스템 콜 사용하려고 할 때

```
1 | man 2 read
```

→ `read()` 가 `ssize_t read(int fd, void *buf, size_t count);` 라는 걸 확인

→ 오류 코드로 `EAGAIN`, `EINTR` 등이 반환될 수 있다는 것도 파악 가능

2. `mmap()` 사용 전 명세 확인

```
1 | man 2 mmap
```

→ 각 매개변수가 어떤 역할이고 어떤 flag를 넣을 수 있는지 파악

→ `MAP_SHARED`, `MAP_PRIVATE`, `PROT_READ`, `PROT_WRITE` 등 중요 플래그 학습

man 2 유용한 시스템 콜 리스트 (자주 쓰이는 것)

시스템 콜	기능
<code>write</code>	파일/표준 출력으로 데이터 쓰기
<code>read</code>	파일/표준 입력에서 데이터 읽기
<code>open</code> / <code>close</code>	파일 열기/닫기
<code>fork</code>	새로운 프로세스 생성
<code>execve</code>	새 프로그램 실행
<code>wait</code> / <code>waitpid</code>	자식 프로세스 종료 대기
<code>kill</code>	시그널 보내기
<code>getpid</code> , <code>getuid</code>	프로세스/사용자 정보 조회

시스템 콜	기능
<code>mmap</code> , <code>munmap</code>	메모리 매핑
<code>ioctl</code>	디바이스 제어
<code>select</code> , <code>poll</code> , <code>epoll_wait</code>	I/O 이벤트 대기
<code>socket</code> , <code>bind</code> , <code>listen</code> , <code>accept</code>	네트워크 소켓

✳ 잘 모를 땐 `man -k` 검색도 가능

```
1 | man -k write
```

👁 출력 예:

```
1 | write (2) - write to a file descriptor
2 | fwrite (3) - binary stream write
```

→ `write` 라는 이름을 가진 함수 중 **섹션 2 (시스템 콜)**을 확인하고 싶다면 `man 2 write`

✅ 요약

명령어	기능
<code>man 2 write</code>	시스템 콜 <code>write()</code> 설명 열기
<code>man 2 mmap</code>	메모리 매핑 시스템 콜 설명
<code>man 3 malloc</code>	라이브러리 함수 <code>malloc()</code> 설명
<code>man -k 키워드</code>	관련된 문서 전체 검색