10. 리눅스 커널 모듈 입문 (고급)

10.1 커널 빌드 개념 및 `Makefile` 작성

🧠 1. 커널 모듈 빌드와 일반 C 프로그램의 차이

구분	일반 C 프로그램	커널 모듈
실행 환경	사용자 공간 (User Space)	커널 공간 (Kernel Space)
실행 파일	ELF 실행파일 (`a.out` , `*.elf`)	커널 오브젝트 파일 (`.ko`)
링커 및 라이브러리	`glibc` , `libc.so` , `ld-linux.so` 등	사용자 공간 라이브러리 사용 금지
빌드 명령	`gcc main.c -o app`	`make` + 커널 Makefile 사용

✅ 커널 모듈은 커널 내부 함수와 구조체를 사용하므로

> 반드시 **커널 빌드 환경** 또는 **커널 헤더 디렉토리**가 필요해.
> 보통 `/lib/modules/$(uname -r)/build` 경로로 제공됨.

🧱 2. 커널 모듈 디렉토리 구조 예시

```
my_module/
├── Makefile
└── hello.c
```

`hello.c` (예제 커널 모듈)

```c
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void) {
    printk(KERN_INFO "Hello, kernel world!\n");
    return 0;
}

void cleanup_module(void) {
    printk(KERN_INFO "Goodbye, kernel world!\n");
}
```

📜 3. Makefile 구조 (커널 모듈용)

```
# Makefile
obj-m := hello.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
    make -C $(KDIR) M=$(PWD) modules

clean:
    make -C $(KDIR) M=$(PWD) clean
```

각 줄의 의미

명령어	설명
`obj-m := hello.o`	커널 모듈로 빌드할 대상 (확장자 `.o`)
`KDIR := .../build`	현재 시스템의 커널 소스/헤더 디렉토리
`PWD := $(shell pwd)`	현재 모듈이 있는 디렉토리
`make -C $(KDIR) M=$(PWD)`	커널 빌드 시스템에 모듈 디렉토리 위임
`modules`	실제 `.ko` 모듈 빌드 타겟
`clean`	중간 산출물 정리 (모듈 삭제)

⚙️ 4. 빌드 및 로딩 절차

```
$ make        # hello.ko 생성됨
$ sudo insmod hello.ko
$ dmesg       # "Hello, kernel world!" 출력 확인
$ sudo rmmod hello
$ dmesg       # "Goodbye, kernel world!" 출력 확인
```

🛠️ 5. 커널 헤더 설치 확인 및 환경 설정

```
# 커널 헤더 설치 여부 확인
$ ls /lib/modules/$(uname -r)/build

# 설치가 안 되어 있다면
$ sudo apt install linux-headers-$(uname -r)
```

🔍 6. 확장 포인트 (심화 학습 방향)

주제	설명
`MODULE_LICENSE()`	GPL 명시 여부에 따라 심볼 제한 존재
`module_init()` / `module_exit()`	현대 커널에서 권장되는 초기화/종료 함수 등록 방식
커널 심볼 확인	`/proc/kallsyms`에서 모듈 함수 위치 확인 가능
의존성 설정	`obj-m += a.o b.o`로 다중 모듈 설정 가능

✅ 실습 체크리스트

- 커널 헤더 설치
- `hello.c` 작성
- Makefile 작성
- `make` → `.ko` 생성
- `insmod`, `rmmod` 테스트
- `dmesg` 로그 확인

10.2 간단한 커널 모듈 (`printk`)

1️⃣ `printk()`란?

- 운영체제 커널 공간에서 사용하는 **로그 출력 함수**
- `printf()` 와 비슷하지만 **커널 공간 전용**
- 출력된 메시지는 **커널 로그 버퍼**에 저장됨 → `dmesg` 명령으로 확인

기본 사용법

```
printk(KERN_INFO "Hello from kernel!\n");
```

로그 레벨 (우선순위)

레벨 매크로	의미	로그 심각도
`KERN_EMERG`	시스템 긴급 상황	0
`KERN_ALERT`	즉각 조치 필요	1
`KERN_CRIT`	치명적 오류	2
`KERN_ERR`	오류 발생	3
`KERN_WARNING`	경고	4

레벨 매크로	의미	로그 심각도
`KERN_NOTICE`	일반적인 중요 정보	5
`KERN_INFO`	일반 정보 메시지	6
`KERN_DEBUG`	디버그용 상세 메시지	7

- 로그 레벨을 생략하면 기본 `DEFAULT_MESSAGE_LOGLEVEL` 사용됨

2️⃣ 간단한 커널 모듈 예제 (Hello World 모듈)

📄 hello.c

```c
#include <linux/module.h>    // 필수: module_init, module_exit, MODULE_LICENSE
#include <linux/kernel.h>    // 필수: printk

static int __init hello_init(void) {
    printk(KERN_INFO "Hello, Kernel World! Module loaded.\n");
    return 0;
}

static void __exit hello_exit(void) {
    printk(KERN_INFO "Goodbye, Kernel World! Module removed.\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple Hello World Kernel Module");
```

핵심 포인트 설명

코드 요소	설명
`#include <linux/module.h>`	커널 모듈 관련 API 헤더
`#include <linux/kernel.h>`	`printk` 및 커널 관련 정의 포함
`__init`, `__exit`	함수 최적화 매크로 (초기화, 종료용)
`module_init()`	모듈 로드시 실행 함수 등록
`module_exit()`	모듈 제거시 실행 함수 등록
`MODULE_LICENSE()`	모듈의 라이센스 명시 → GPL 필수 (미지정 시 경고)
`printk()`	커널 로그 출력

---

3 Makefile (복습 차원에서 재확인)

```
obj-m := hello.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
    make -C $(KDIR) M=$(PWD) modules

clean:
    make -C $(KDIR) M=$(PWD) clean
```

---

4 빌드 및 테스트 방법

빌드

```
$ make
```

모듈 삽입

```
$ sudo insmod hello.ko
```

커널 로그 확인

```
$ dmesg | tail -n 20
```

모듈 제거

```
$ sudo rmmod hello
```

다시 커널 로그 확인

```
$ dmesg | tail -n 20
```

---

5 결과 예시

```
[12345.678901] Hello, Kernel World! Module loaded.
[12347.890123] Goodbye, Kernel World! Module removed.
```

6️⃣ 심화 포인트

- `KERN_INFO`, `KERN_WARNING` 등 다른 레벨로도 출력해보기
- 여러 줄 메시지 출력 (멀티라인 `printk`)
- 변수 값 출력 실습 → `printk(KERN_INFO "Value: %d\n", my_value);`
- 타이밍에 따른 출력 순서 관찰 → 시간 측정 / 루프 추가

✅ 실습 체크리스트

- `hello.c` 작성
- `Makefile` 작성
- `make` → `hello.ko` 생성
- `insmod` → `dmesg` 확인
- `rmmod` → `dmesg` 확인
- 다른 로그 레벨 테스트

10.3 모듈 삽입/삭제 (`insmod`, `rmmod`)

1️⃣ 커널 모듈의 역할

- **커널 기능을 동적으로 확장**하는 코드 조각
- 전통적 커널 빌드 후 정적 포함 대신 → 필요할 때만 **삽입 / 제거** 가능

주요 사용 예

상황	모듈 예시
네트워크 드라이버 로드	`e1000e.ko`
파일 시스템 지원 추가	`nfs.ko`, `cifs.ko`
커널 디버그용 모듈	`hello.ko` (학습용)

2️⃣ 모듈 관리 명령어

1. `insmod`

- 모듈 파일 (`.ko`)을 커널에 **삽입**하는 명령
- 기본 문법

```
sudo insmod hello.ko
```

- 특징
  - **의존성 자동 처리 불가**

- 심볼 충돌 발생 시 오류 발생 가능
- 단순 **로우 레벨 로드**

2. `rmmod`

- 현재 커널에 **로드된 모듈 제거** 명령
- 기본 문법

```
sudo rmmod hello
```

- 특징
  - `.ko` 확장자 사용 안 함 → 모듈 이름만 지정
  - 모듈이 **busy** 상태면 제거 실패

3️⃣ 로드/언로드 시 내부 흐름

모듈 로드

```
insmod → module_init() 호출 → init_module() or 사용자 정의 초기화 함수 실행
```

모듈 제거

```
rmmod → module_exit() 호출 → cleanup_module() or 사용자 정의 종료 함수 실행
```

→ 우리는 앞서 `hello.c`에서 `hello_init()`과 `hello_exit()`으로 이 흐름을 구성했지.

4️⃣ 커널 모듈 상태 확인

`lsmod`

- 현재 커널에 **로드된 모듈 목록** 확인

```
lsmod
```

출력 예시:

```
Module                  Size  Used by
hello                  16384  0
...
```

`modinfo`

- 특정 모듈의 **정보 확인**

```
modinfo hello.ko
```

출력 예시:

```
filename:       /path/to/hello.ko
license:        GPL
author:         Your Name
description:    A simple Hello World Kernel Module
vermagic:       5.15.0-56-generic SMP mod_unload
```

5️⃣ 의존성 관리 - `modprobe` (미리 소개)

> `modprobe`는 `insmod`보다 **고급 기능** 제공:

- 모듈 간 의존성 (`depmod` 정보) 자동 처리
- 모듈 파라미터 전달 가능

```
sudo modprobe hello
sudo modprobe -r hello
```

※ `modprobe`를 사용하려면 `/lib/modules/$(uname -r)/modules.dep` 구축 필요
→ `depmod` 명령어로 재생성 가능

6️⃣ 모듈 제거 실패 상황 (중요 ⭐)

`rmmod` 오류 예시

```
rmmod: ERROR: Module hello is in use
```

- 보통 `Used by` 필드가 0이 아닌 경우 발생
- 해결 방법:
  - 사용 중인 프로세스 종료
  - 관련 참조 해제
  - `rmmod -f hello` 강제 제거 (권장 X → 실습용만)

```
sudo rmmod -f hello
```

※ 강제 제거는 커널 패닉 위험 있으니 주의!

7️⃣ 실습 절차

1️⃣ 모듈 빌드

```
make
```

2️⃣ 모듈 삽입

```
sudo insmod hello.ko
dmesg | tail -n 20
```

3️⃣ 모듈 상태 확인

```
lsmod | grep hello
modinfo hello.ko
```

4️⃣ 모듈 제거

```
sudo rmmod hello
dmesg | tail -n 20
```

✅ 실습 체크리스트

- `insmod` 로드 후 `dmesg` 확인
- `lsmod`로 모듈 확인
- `modinfo`로 정보 출력 확인
- `rmmod`로 제거 후 `dmesg` 확인
- 제거 오류 상황 체험 (`rmmod -f`는 실습용만!)

10.4 커널 로그 확인 (`dmesg`)

🧠 1. 커널 메시지 버퍼란?

리눅스 커널은 자체 로그 메시지를 **메모리 내부의 링 버퍼(Ring Buffer)** 에 저장해.
이 버퍼는 다음과 같은 정보를 담고 있음:

- `printk()`으로 출력된 메시지
- 커널 부팅 과정 메시지
- 드라이버 로딩/언로딩 메시지
- 시스템 콜 실패나 경고 등

특성

항목	설명
링 버퍼 구조	오래된 로그는 자동으로 삭제됨
사용자 공간 접근	`dmesg`, `journalctl -k` 사용
커널 내부 인터페이스	`/dev/kmsg`, `/proc/kmsg` 등

🧰 2. `dmesg` 명령어의 역할

- 커널 링 버퍼의 내용을 **사용자 공간에서 출력**
- 주로 디버깅, 모듈 로딩 확인, 하드웨어 상태 확인 등에 사용됨

기본 사용법

```
$ dmesg
```

출력 예시:

```
[  123.456789] Hello, Kernel World! Module loaded.
[  125.123456] Goodbye, Kernel World! Module removed.
```

> 시간은 부팅 후 경과 시간 (초) 기준

🔍 3. 자주 사용하는 옵션

명령어	설명
`dmesg | tail`	
`dmesg -T`	**사람이 읽기 쉬운 시간** 표시
`dmesg -k`	커널 메시지만 출력 (Syslog 메시지 제외)
`dmesg -n <level>`	출력할 최대 로그 레벨 지정
`dmesg -w`	**실시간 로그 감시 (watch)**

예제

```
$ dmesg -T | tail -n 10
$ dmesg -w   # 실시간 로그 확인
```

📝 4. 로그 레벨 필터링 (`printk` 로그 우선순위)

```
printk(KERN_INFO "This is info log\n");
printk(KERN_WARNING "Warning!\n");
printk(KERN_ERR "Critical Error!\n");
```

`dmesg -n LEVEL` 의미

```
$ dmesg -n 3
```

→ `KERN_ERR` 이하 (즉, 0~3 수준의 로그만 출력)

숫자	우선순위	매크로
0	긴급	`KERN_EMERG`
1	경고	`KERN_ALERT`
2	중요 오류	`KERN_CRIT`
3	일반 오류	`KERN_ERR`
4	경고	`KERN_WARNING`
5	공지사항	`KERN_NOTICE`
6	정보	`KERN_INFO`
7	디버깅	`KERN_DEBUG`

📁 5. 내부 경로: `/proc/kmsg`, `/dev/kmsg`

파일 경로	설명
`/proc/kmsg`	커널 메시지 스트림 (읽으면 비워짐)
`/dev/kmsg`	커널 로그 장치 파일 (`dmesg` 도 여길 읽음)

```
sudo cat /proc/kmsg      # 커널 로그 직접 읽기
sudo cat /dev/kmsg       # 실시간 스트림 형태로 출력됨
```

※ 보안 상 일반 사용자에겐 접근 제한됨

🧪 6. 실습 예시

1. 모듈 삽입 → 로그 확인

```
$ sudo insmod hello.ko
$ dmesg | tail -n 5
```

1. 로그에 시간 표시 추가

```
$ dmesg -T | tail -n 5
```

1. 실시간 모니터링

```
$ dmesg -w
```

→ `rmmod` 명령을 실행해보면 실시간으로 로그 뜨는 걸 볼 수 있어

✅ 실습 체크리스트

- `dmesg` 기본 사용
- `dmesg -T`로 시간 확인
- `dmesg -w`로 실시간 로그 감시
- `printk` 레벨 변경 후 로그 구분
- `/dev/kmsg`, `/proc/kmsg` 확인 (선택)

📌 참고: 로그가 안 보일 때 점검할 것

원인	해결 방법
`printk` 레벨이 낮음	`dmesg -n 7` 설정
`printk()` 호출 누락	`module_init()` 연결 확인
버퍼 초과로 삭제됨	`dmesg -w`로 실시간 확인
root 권한 필요	`sudo` 붙여서 실행

10.5 시스템콜 인터셉트 예제

🚩 목적

시스템콜 인터셉트란?
👉 **커널이 제공하는 시스템콜의 동작을 가로채서 내가 원하는 동작을 삽입**하는 것.

주로 사용 목적:

- 보안 모듈(예: rootkit)이 시스템콜을 감시

- 특정 시스템콜 사용을 제한하거나 변조
- 학습용으로 커널 내부 동작 이해

🚫 주의사항

- 최신 커널 (>= 5.x)은 `sys_call_table` 기호가 **내보내지 않음(export 금지)** → 바로 접근 불가
- 강제로 접근할 경우 **불안정성 / 커널 패닉** 발생 가능
- 실습은 반드시 **가상머신(VM)**이나 **테스트용 장비**에서 진행할 것

※ 프로덕션 환경에서는 절대 사용 X
※ 실제 보안 모듈은 보통 **LSM(리눅스 시큐리티 모듈)** Hook을 사용함 (정식 지원 API 존재)

🗺️ 전체 흐름

```
[기존 구조]

user-space → sys_call_table[NR_openat] → sys_openat()

[인터셉트 후 구조]

user-space → sys_call_table[NR_openat] → my_sys_openat() → (원래 sys_openat 호출 가능)
```

1️⃣ 기본 설계

1️⃣ `sys_call_table` 주소 알아내기 (커널 심볼 수집)
2️⃣ 시스템콜 핸들러 교체 (`write_cr0()`로 WP(Write Protect) 비활성화 필요)
3️⃣ 새로운 핸들러 작성 (ex: open() 호출 감지)
4️⃣ 모듈 제거 시 원상복구

2️⃣ 예제 코드

1. 사전 준비

- 시스템콜 테이블 주소 알아내기 → `kallsyms_lookup_name()` 사용
- 커널 설정에 따라 `kallsyms_lookup_name` 사용 가능 여부 달라짐 → 필요시 패치 필요

2. 인터셉트할 시스템콜 고르기

여기서는 `__x64_sys_openat` **(open syscall)** 를 예제로 사용.

---

3. 인터셉트 모듈 코드 (강의용 예제)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/syscalls.h>
#include <linux/kallsyms.h>
#include <linux/uaccess.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("System call intercept example");

unsigned long **sys_call_table;

asmlinkage long (*original_sys_openat)(int, const char __user *, int, umode_t);

asmlinkage long my_sys_openat(int dfd, const char __user *filename, int flags, umode_t
mode) {
    char fname[256];
    long ret;

    if (strncpy_from_user(fname, filename, sizeof(fname)) > 0) {
        printk(KERN_INFO "Intercepted openat(): %s\n", fname);
    }

    ret = original_sys_openat(dfd, filename, flags, mode);

    return ret;
}

static void disable_write_protection(void) {
    unsigned long cr0;
    preempt_disable();
    barrier();
    asm volatile("mov %%cr0, %0" : "=r"(cr0));
    cr0 &= ~0x00010000;
    asm volatile("mov %0, %%cr0" : : "r"(cr0));
    barrier();
}

static void enable_write_protection(void) {
    unsigned long cr0;
    barrier();
    asm volatile("mov %%cr0, %0" : "=r"(cr0));
    cr0 |= 0x00010000;
    asm volatile("mov %0, %%cr0" : : "r"(cr0));
    barrier();
    preempt_enable();
}

static int __init interceptor_init(void) {
    sys_call_table = (unsigned long **)kallsyms_lookup_name("sys_call_table");

```

```
    if (!sys_call_table) {
        printk(KERN_ERR "Failed to locate sys_call_table\n");
        return -1;
    }

    original_sys_openat = (void *)sys_call_table[__NR_openat];

    disable_write_protection();
    sys_call_table[__NR_openat] = (unsigned long *)my_sys_openat;
    enable_write_protection();

    printk(KERN_INFO "System call openat() intercepted\n");
    return 0;
}

static void __exit interceptor_exit(void) {
    if (!sys_call_table)
        return;

    disable_write_protection();
    sys_call_table[__NR_openat] = (unsigned long *)original_sys_openat;
    enable_write_protection();

    printk(KERN_INFO "System call openat() restored\n");
}

module_init(interceptor_init);
module_exit(interceptor_exit);
```

3 핵심 포인트 설명

부분	설명
`kallsyms_lookup_name`	`sys_call_table` 주소 얻기
`disable_write_protection`	`cr0.WP` 비트 변경해서 커널 메모리 쓰기 가능화
`original_sys_openat` 저장	원래 핸들러 백업
`my_sys_openat` 구현	사용자 정의 핸들러 (printk로 로깅만)
복원 과정	`rmmod` 시 원상복구 수행

4️⃣ 실습 절차

1️⃣ 모듈 빌드

```
$ make
```

2️⃣ 모듈 삽입

```
$ sudo insmod interceptor.ko
$ dmesg | tail -n 10
```

3️⃣ 테스트 → 임의의 파일 열어보기

```
$ cat /etc/hostname
$ dmesg | tail -n 10
```

→ **Intercepted openat(): /etc/hostname** 출력 확인

4️⃣ 모듈 제거

```
$ sudo rmmod interceptor
$ dmesg | tail -n 10
```

→ **System call openat() restored** 메시지 확인

5️⃣ 실습 시 주의점

- 최신 커널은 `kallsyms_lookup_name` 도 `EXPORT_SYMBOL_GPL` 상태라 **GPL 명시 필수**
- 시스템콜 인터셉트는 보안상 매우 위험한 기법이라 **정식 커널 개발에선 금지**
- 테스트용으로만 실습!

✅ 실습 체크리스트

- `kallsyms_lookup_name` 정상 동작 확인
- `sys_call_table` 주소 획득
- 시스템콜 핸들러 교체 / 복원
- 파일 열기 시 `dmesg` 에 로깅 확인
- 커널 패닉 없이 모듈 제거 성공

실습

Hello Kernel Module 작성

🧱 1. 목표

- 리눅스 커널 모듈 개발의 **기초 구조 파악**
- `printk()` 를 통해 **커널 메시지 출력**
- `insmod`, `rmmod`, `dmesg` 흐름을 통한 모듈 로드/언로드 체험

📁 2. 디렉토리 구성

```
hello_module/
├── hello.c
└── Makefile
```

📄 3. hello.c 소스 코드

```c
// hello.c
#include <linux/module.h>    // 커널 모듈 관련 매크로
#include <linux/kernel.h>    // printk()
#include <linux/init.h>      // __init, __exit 매크로

static int __init hello_init(void)
{
    printk(KERN_INFO "🔧 Hello, Kernel! Module successfully loaded.\n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_INFO "🧹 Goodbye, Kernel! Module successfully unloaded.\n");
}

module_init(hello_init);   // 초기화 함수 등록
module_exit(hello_exit);   // 종료 함수 등록

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A basic Hello World Linux Kernel Module");
```

📜 4. Makefile

```
# Makefile
obj-m := hello.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
    make -C $(KDIR) M=$(PWD) modules

clean:
    make -C $(KDIR) M=$(PWD) clean
```

⚙️ 5. 빌드 과정

```
$ make
```

빌드가 성공하면 다음과 같은 파일이 생성돼:

- `hello.ko` (커널 오브젝트 모듈)
- `hello.mod.c`, `hello.o`, `.mod.o`, `.symvers` 등 중간 파일들

📦 6. 모듈 삽입 및 확인

```
$ sudo insmod hello.ko
$ dmesg | tail -n 10
```

→ 출력 예시:

```
[ 1234.56789] 🔧 Hello, Kernel! Module successfully loaded.
```

🧹 7. 모듈 제거 및 로그 확인

```
$ sudo rmmod hello
$ dmesg | tail -n 10
```

→ 출력 예시:

```
[ 1236.78900] 🧹 Goodbye, Kernel! Module successfully unloaded.
```

🧪 8. 실습 체크리스트

항목	완료 여부
`hello.c` 작성 완료 ✅	✅
`Makefile` 생성 ✅	✅
`make`로 `.ko` 빌드 ✅	✅
`insmod`로 커널 삽입 ✅	✅
`dmesg`로 로그 확인 ✅	✅
`rmmod`로 제거 ✅	✅

📌 9. 권장 후속 실습

- `printk()` 로그 레벨 다양하게 실험
- `module_param()`으로 모듈 파라미터 전달
- `/proc` 노드 생성 (`proc_create()`)
- `init_module()`과 `cleanup_module()` 직접 써보기 (옛 방식)

`/proc/myinfo` 가상파일 시스템 생성

🗺️ 개념 정리

개념	설명
`/proc`	커널 내부 정보를 사용자 공간에 노출하는 특수 가상 파일 시스템
가상 파일 노드	실제로는 디스크에 존재하지 않음 → 커널 메모리를 통해 실시간 생성
주요 사용 예	디버그 정보, 상태 정보 제공

우리가 만들 `/proc/myinfo`는:

```
$ cat /proc/myinfo
→ "Hello from my kernel module!"
```

를 출력하게 만들 거야.

---

1 커널 API 준비

우리는 `proc_create()` 함수와 `struct proc_ops` 구조체를 사용할 거야.
(이전 커널에서는 `struct file_operations` 사용했으나 최신 커널은 `proc_ops`가 권장됨.)

---

2 전체 코드 예제 (`hello_proc.c`)

```
// hello_proc.c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>

#define PROC_NAME "myinfo"

static int myinfo_show(struct seq_file *m, void *v)
{
    seq_printf(m, "Hello from my kernel module!\n");
    seq_printf(m, "Kernel version: %s\n", UTS_RELEASE);
    return 0;
}

static int myinfo_open(struct inode *inode, struct file *file)
{
    return single_open(file, myinfo_show, NULL);
}

static const struct proc_ops myinfo_fops = {
    .proc_open    = myinfo_open,
    .proc_read    = seq_read,
    .proc_lseek   = seq_lseek,
    .proc_release = single_release,
};

static int __init hello_proc_init(void)
{
    proc_create(PROC_NAME, 0, NULL, &myinfo_fops);
    printk(KERN_INFO "/proc/%s created\n", PROC_NAME);
    return 0;
}

static void __exit hello_proc_exit(void)
{
    remove_proc_entry(PROC_NAME, NULL);
    printk(KERN_INFO "/proc/%s removed\n", PROC_NAME);
}

module_init(hello_proc_init);
module_exit(hello_proc_exit);

```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A simple /proc/myinfo kernel module");
```

---

3 핵심 설명

구성 요소	설명
`proc_create`	`/proc`에 새로운 파일 생성
`myinfo_show`	파일 내용 출력 함수 → `seq_file` 인터페이스 사용
`single_open`	파일 열기 시 동작 연결
`proc_ops`	최신 커널에서 사용되는 proc 파일용 ops 구조체
`remove_proc_entry`	모듈 제거 시 `/proc` 노드 삭제

`seq_file` 인터페이스

- `/proc` 파일에 *버퍼 없이 안전하게 문자열 출력* 지원
- `seq_printf()` 사용

---

4 Makefile 예시

```
obj-m := hello_proc.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
    make -C $(KDIR) M=$(PWD) modules

clean:
    make -C $(KDIR) M=$(PWD) clean
```

---

5 빌드 및 테스트

```
$ make
$ sudo insmod hello_proc.ko
$ dmesg | tail -n 10
→ "/proc/myinfo created" 확인

$ cat /proc/myinfo
Hello from my kernel module!
Kernel version: 6.x.x-xxx-generic
```

모듈 제거

```
$ sudo rmmod hello_proc
$ dmesg | tail -n 10
→ "/proc/myinfo removed" 확인

$ cat /proc/myinfo
→ No such file or directory (정상 동작)
```

✅ 실습 체크리스트

- `hello_proc.c` 작성
- `Makefile` 작성
- `make` 성공
- `/proc/myinfo` 생성 확인
- `cat /proc/myinfo` 내용 확인
- 모듈 제거 후 파일 사라짐 확인

📌 확장 학습 방향

주제	설명
`seq_printf()`로 **동적 정보** 출력하기 (ex: CPU usage, counter 등)	
`/proc/myinfo` → **쓰기 가능**하게 만들기 (`proc_write` 추가)	
서브디렉토리 생성 (`proc_mkdir()`)	
`/proc/my_module/status` 와 같이 **다중 파일 구조 만들기**	

시스템 콜 후킹 테스트

1️⃣ 목표

- 시스템 콜 테이블(`sys_call_table`)에서 특정 시스템 콜을 **가로채기**
- 후킹된 함수에서 `printk()` 로 **로깅** 수행
- 원래 시스템 콜 호출은 유지
- 후킹 → 테스트 → 복원 순으로 흐름을 확인

---

2 전제조건

- **커널 빌드 옵션이 kallsyms_lookup_name 허용 상태일 것**
- 커널이 `sys_call_table`을 export 하지 않으므로 → 반드시 `kallsyms_lookup_name()`으로 가져와야 함
- 테스트 환경은 **가상머신 추천** (크래시 발생 가능성 존재)

---

3 후킹 대상 선정

이번 테스트에서는 → `__x64_sys_openat` (즉, 유저가 파일을 열 때 호출되는 시스템 콜)을 후킹해볼게.

> 왜 openat()냐?
> → open()은 최신 glibc에서 openat()으로 내부 변환돼서 호출됨.
> 따라서 openat() 후킹이 효과적임.

---

4 전체 코드 예제 (후킹 테스트용)

`syscall_hook.c`

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/syscalls.h>
#include <linux/kallsyms.h>
#include <linux/uaccess.h>
#include <linux/unistd.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("System Call Hook Test");

unsigned long **sys_call_table;
asmlinkage long (*original_sys_openat)(int dfd, const char __user *filename, int flags, umode_t mode);

asmlinkage long my_sys_openat(int dfd, const char __user *filename, int flags, umode_t mode)
{
    char fname[256];

    if (strncpy_from_user(fname, filename, sizeof(fname)) > 0)
    {
        printk(KERN_INFO "HOOKED openat(): %s\n", fname);
    }

    // 원래 시스템 콜 호출
    return original_sys_openat(dfd, filename, flags, mode);
}

static void disable_wp(void)
```

```
{
    unsigned long cr0;
    preempt_disable();
    barrier();
    asm volatile("mov %%cr0, %0" : "=r"(cr0));
    cr0 &= ~0x00010000;
    asm volatile("mov %0, %%cr0" : : "r"(cr0));
    barrier();
}

static void enable_wp(void)
{
    unsigned long cr0;
    barrier();
    asm volatile("mov %%cr0, %0" : "=r"(cr0));
    cr0 |= 0x00010000;
    asm volatile("mov %0, %%cr0" : : "r"(cr0));
    barrier();
    preempt_enable();
}

static int __init hook_init(void)
{
    sys_call_table = (unsigned long **)kallsyms_lookup_name("sys_call_table");

    if (!sys_call_table)
    {
        printk(KERN_ERR "Failed to find sys_call_table\n");
        return -1;
    }

    original_sys_openat = (void *)sys_call_table[__NR_openat];

    disable_wp();
    sys_call_table[__NR_openat] = (unsigned long *)my_sys_openat;
    enable_wp();

    printk(KERN_INFO "System call openat() hooked\n");
    return 0;
}

static void __exit hook_exit(void)
{
    if (sys_call_table)
    {
        disable_wp();
        sys_call_table[__NR_openat] = (unsigned long *)original_sys_openat;
        enable_wp();

        printk(KERN_INFO "System call openat() restored\n");
    }
}

```

```
module_init(hook_init);
module_exit(hook_exit);
```

---

5 Makefile

```
obj-m := syscall_hook.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
    make -C $(KDIR) M=$(PWD) modules

clean:
    make -C $(KDIR) M=$(PWD) clean
```

---

6 빌드 및 테스트 절차

빌드

```
$ make
```

모듈 삽입

```
$ sudo insmod syscall_hook.ko
$ dmesg | tail -n 10
→ "System call openat() hooked" 확인
```

테스트

```
$ cat /etc/hostname
```

dmesg 확인

```
$ dmesg | tail -n 10
```

→ 예시 출력:

```
[ ... ] HOOKED openat(): /etc/hostname
```

모듈 제거

```
$ sudo rmmod syscall_hook
$ dmesg | tail -n 10
→ "System call openat() restored" 확인
```

✅ 실습 체크리스트

항목	완료 여부
`sys_call_table` 주소 확보	✅
시스템 콜 핸들러 교체 → 정상 동작	✅
`HOOKED openat()` 출력 확인	✅
원래 시스템 콜 정상 복원	✅
커널 패닉 없이 모듈 제거 성공	✅

📌 주의사항

문제 상황	원인
kallsyms_lookup_name() 사용 불가	커널 구성에 따라 제한 가능 (CONFIG_KALLSYMS)
write_cr0 사용 불가	최신 커널에서는 일부 보안 패치 적용됨
커널 패닉 발생	시스템 콜 복원 실패, WP 미복원
rmmod 강제 실패	`sys_call_table` 복원 정확히 확인 필요

🚀 확장 방향

- 여러 시스템 콜 동시 후킹 (ex: read, write)
- 커널 버전 별 차이 대응 (sys_call_table 위치 달라짐)
- syscall 인터셉트 → 동적 정책 적용 (화이트리스트/블랙리스트)
- Shadow syscall table 사용 (일부 LSM 기반 기법 학습)