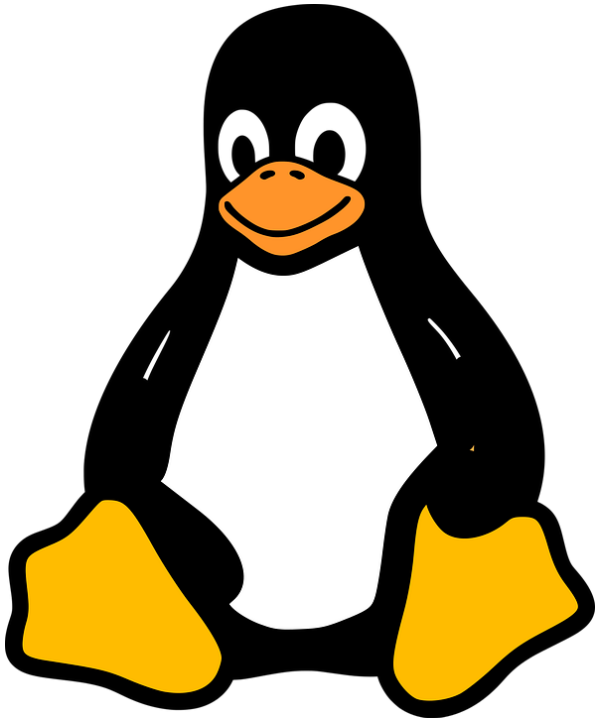
AI TRAINING SERIES

Kernel - 002

Contents



https://www.raspberrypi.com/documentation/computers/linux_kernel.html

I. 라즈베리 파이 커널 빌드 & 크로스 컴파일

라즈베리 파이에서 커널 빌드하기 (44m 25s)

```
# 필수 SW 패키지 설치
sudo apt update
sudo apt install -y git bc bison flex libssl-dev make libc6-dev libncurses5-dev

# Raspberrypi Kernel 소스 다운로드
mkdir kernel && cd kernel
git clone --depth=1 https://github.com/raspberrypi/linux.git
cd linux

KERNEL=kernel_2712
make bcm2712_defconfig

# 커널 빌드
make -j6 Image.gz modules dtbs

# 커널 설치
sudo make -j6 modules_install
sudo cp /boot/firmware/$KERNEL.img /boot/firmware/$KERNEL-backup.img
sudo cp arch/arm64/boot/Image.gz /boot/firmware/$KERNEL.img
sudo cp arch/arm64/boot/dts/broadcom/*.dtb /boot/firmware/
sudo cp arch/arm64/boot/dts/overlays/*.dtb* /boot/firmware/overlays/
sudo cp arch/arm64/boot/dts/overlays/README /boot/firmware/overlays/

sudo reboot
```

https://www.raspberrypi.com/documentation/computers/linux_kernel.html

I. 라즈베리 파이 커널 빌드 & 크로스 컴파일

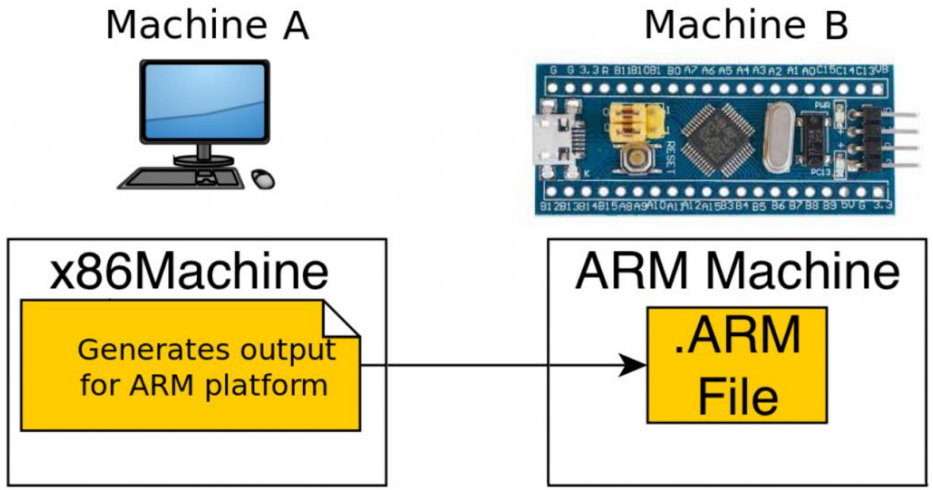
Ubuntu Desktop 에서 라즈베리파이 커널 빌드하기 (9m 35s)

```
sudo apt update
sudo apt install -y git bc bison flex libssl-dev make libc6-dev libncurses5-dev
sudo apt install -y crossbuild-essential-arm64 # Cross Complier 설치

mkdir kernel && cd kernel
git clone --depth=1 https://github.com/raspberrypi/linux.git
cd linux

export KERNEL=kernel_2712
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2712_defconfig

# 커널 빌드
make -j$(nproc) ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image modules dtbs

# 커널 이미지 복사
mkdir mnt
mkdir mnt/boot
mkdir mnt/root
sudo mount /dev/sdb1 mnt/boot
sudo mount /dev/sdb2 mnt/root

sudo env PATH=$PATH make -j12 ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- INSTALL_MOD_PATH=mnt/root modules_install

sudo umount mnt/boot
sudo umount mnt/root
```

Ⅰ. 라즈베리 파이 커널 빌드 & 크로스 컴파일

크로스 컴파일이란?

- 크로스 컴파일(Cross-Compilation)이란 빌드를 수행하는 빌드(Build) 머신과, 최종 실행 대상(Target) 머신의 아키텍처가 다른 경우의 컴파일
  - Build 머신: 실제로 컴파일이 실행되는 머신 (예: Ubuntu PC x86_64)
  - Target 머신: 최종 제품이 될 아키텍처/플랫폼 (예: Raspberry Pi ARM 보드)
  - 필요성: 임베디드나 모바일 장치의 경우 자체에서 컴파일하기에는 성능이 부족하거나 컴파일러 환경 구성이 어려운 경우가 많음. 예를 들어 작은 IoT 보드는 메모리도 수십 MB 수준이라 컴파일을 돌리기 어려움. 또한 개발 편의를 위해 주로 PC에서 코드 편집과 빌드를 하고, 결과 실행파일만 장치로 보내는 워크플로우가 일반적임
  - 필요사항: 크로스 컴파일러(교차 툴체인)이라는, target 용 코드를 생성할 수 있는 컴파일러 필요



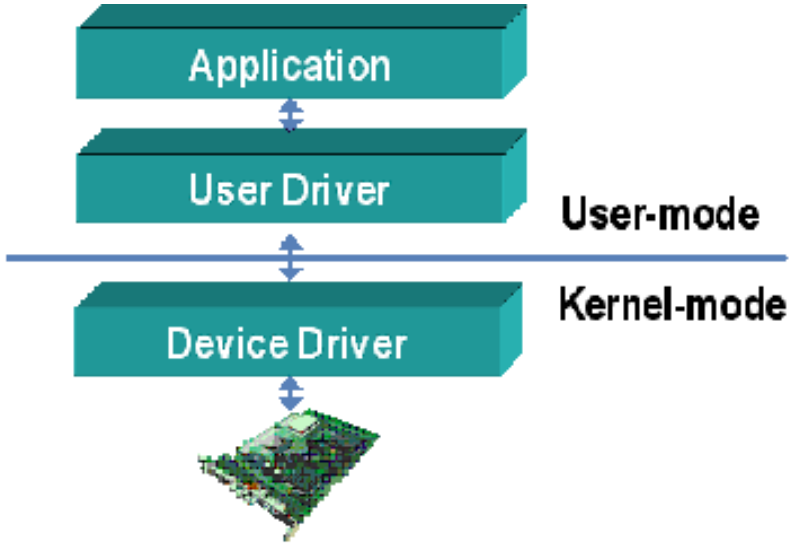
Ⅱ. 디바이스 드라이버

디바이스 드라이버란?

- 정의
  - 장치 드라이버(Device Driver) 는 하드웨어를 OS 커널과 연결하는 소프트웨어

- Device Driver 의 관리 방식 2가지
  - 커널 모듈: 리눅스에서는 Device Driver를 커널 모듈(.ko = kernel object) 형태로 만들고, 이를 동적으로 로드 하여 사용 (CONFIG_EXAMPLE=y)
  - 커널 내장: Kernel 이미지에 포함되어 부팅 시 자동 로드되며 별도 ko 파일 없음 (CONFIG_EXAMPLE=m)

- 역할
  - 하드웨어와 OS 간 데이터 전송 및 제어
  - OS로 하여금 하드웨어를 추상화된 형태로 사용할 수 있도록 함
  - 사용자 애플리케이션이 직접 HW를 접근하지 않고, 표준 인터페이스 제공



프린터 드라이버가 없으면 컴퓨터는 프린터를 사용할 수 없음

R8126 network driver 가 없으면 우리 데스크탑은 Ethernet 연결 안됨

Ⅱ. 디바이스 드라이버

디바이스 드라이버의 종류

종류	설명	쉬운 예시
문자 디바이스	데이터를 순차적이고 작은 단위(byte)로 읽고 씀	키보드 입력, 마우스 클릭
블록 디바이스	데이터를 **블록**(512byte **등**) **단위로 랜덤 접근** 가능	USB 메모리, SSD, HDD
네트워크 디바이스	네트워크 **패킷**으로 데이터를 주고 받음	랜 카드, 와이파이 카드

Ⅱ. 디바이스 드라이버

디바이스 드라이버 module 만들어 보기 - Build 환경 공통

```
max@max-MS-7E01:~/works/kernel/device-driver/00.hello$ tree
.
├── hello.c
└── Makefile
```

Makefile

```
obj-m += hello.o # ← 사용하는 드라이버 파일의 이름에 맞게 수정하기

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

빌드하는 법

```
# 빌드하기: ko 파일 생성됨
make

# 빌드한 내용을 clean 하기: ko 파일 및 기타 obj 파일 제거
make clean
```

```
max@max-MS-7E01:~/works/kernel/device-driver/00.hello$ tree
.
├── hello.c
├── hello.ko
├── hello.mod
├── hello.mod.c
├── hello.mod.o
├── hello.o
├── Makefile
├── modules.order
└── Module.symvers
```

Ⅱ. 디바이스 드라이버

디바이스 드라이버 module 만들어 보기 - Module Skeleton 공통

```
#include <linux/module.h>
int my_init_module(void){
        printk("The module is now loaded\n");
        return 0;
}
void my_cleanup_module(void){
        printk("The module is now unloaded\n");
}
module_init(my_init_module);
module_exit(my_cleanup_module);
MODULE_LICENSE("GPL");
```

Ⅱ. 디바이스 드라이버

```
max@max-MS-7E01:~/works/kernel/device-driver/00.hello$ tree
.
├── hello.c
└── Makefile
```

디바이스 드라이버 module 만들어 보기 - Char 디바이스 드라이버

hello.c

```c
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>

static dev_t dev_num;
static struct cdev cdev;

ssize_t hello_read(struct file *filp, char __user *buf, size_t len, loff_t *offset) {
        char *msg = "Hello World!\n";
        return simple_read_from_buffer(buf, len, offset, msg, strlen(msg));
}

struct file_operations fops = {
        .read = hello_read,
};

static int __init hello_init(void) {
        alloc_chrdev_region(&dev_num, 0, 1, "hello_device");
        cdev_init(&cdev, &fops);
        cdev_add(&cdev, dev_num, 1);
        printk(KERN_INFO "Hello driver loaded\n");
        return 0;
}

static void __exit hello_exit(void) {
        cdev_del(&cdev);
        unregister_chrdev_region(dev_num, 1);
        printk(KERN_INFO "Hello driver unloaded\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```

Build

```
# 빌드하기 - 빌드 성공하면 hello.ko 파일 생성되어야 함
make
```

Module Load

```
# insmod - 디바이스 드라이버 모듈을 load 하는 명령어로 sudo 권한으로 실행
sudo insmod hello.ko
```

확인하기

```
# lsmod - load 된 디바이스 드라이버 list 보는 명령어
lsmod | grep hello
# hello device 가 생성됐는지 확인
ls /dev/hello*
# device node 를 생성해 주는 명령어로 /dev/hello_device 를 생성해줌
sudo mknod /dev/hello_device c $(awk '$2=="hello_device" {print $1}' /proc/devices) 0
# hello device 가 생성됐는지 확인
ls /dev/hello*
# hello device 의 값을 read
cat /dev/hello_device
```

Module Remove

```
# rmmod - 디바이스 드라이버 모듈을 remove 하는 명령어로 sudo 권한으로 실행
sudo rmmod hello
```

Ⅱ. 디바이스 드라이버

Device Driver 관련 주요 명령어

lsmod (list modules)

- **개념**: 현재 로드된 커널 모듈(.ko)을 보여줌
- **목적**: 현재 로드된 드라이버 확인

insmod (insert module)

- **개념**: 커널 모듈(.ko)을 수동으로 로드
- **목적**: 새로운 드라이버를 즉시 사용 가능하게 함

dmesg

- **개념**: 커널에서 발생한 로그를 보는 명령
- **목적**: 디바이스 드라이버 로드 시 상태 메시지 확인

dmesg 주요 사용법

```
# 별도 설정없이 dmesg 실행 – dmesg buffer 에 쌓인 내용 한번에 출력
sudo dmesg


# 별도로 종료하지 않는 한 계속해서 dmesg 를 실시간으로 출력
sudo dmesg -w


# hello 라는 문자열로 filtering 하면서 dmesg 를 실시간으로 출력
sudo dmesg -w | grep hello


# dmesg 의 buffer clear
sudo dmesg -c


# dmesg 내용을 실시간으로 보면서 file 로 내용을 저장
sudo dmesg -w 2>&1 | tee dmesg_dump.txt
```

Ⅱ. 디바이스 드라이버

Kernel Log - printk

- 사용자 공간의 printf()와 유사하게, 커널 공간(kernel space) 에서 텍스트 메시지를 출력하는 함수
- 출력 대상
  - 커널 링 버퍼 (/dev/kmsg 또는 dmesg 로 확인)
  - 콘솔(터미널) (실시간으로 화면에 출력)
- Kernel Config 설정
  - CONFIG_PRINTK: printk() 지원을 켜거나 끔 (왠만하면 =y)
  - CONFIG_LOG_BUF_SHIFT: 로그 버퍼 크기 설정 (예: 18 로 설정됐으면 1<18: 2^18 = 258KB)
  - CONFIG_DYNAMIC_DEBUG:

    /sys/kernel/debug/dynamic_debug/control 을 사용해 runtime 에 log on/off
- 사용 예

```
printk(KERN_INFO "Hello, kernel world! - INFO level\n");
int x = 0; int status = 1;
printk(KERN_DEBUG "x=%d, status=%d\n", x, status);
```

레벨 값	상수	의미
0	KERN_EMERG	긴급: 시스템 동작 불능
1	KERN_ALERT	즉각 조치 필요
2	KERN_CRIT	치명적 오류
3	KERN_ERR	일반 오류
4	KERN_WARNING	경고: 예상치 못한 상황
5	KERN_NOTICE	주목: 일반적인 상태 변화
6	KERN_INFO	정보: 정상 동작 정보
7	KERN_DEBUG	디버그: 개발·디버깅 용

Ⅱ. 디바이스 드라이버

실습하기 - 1

- Hello device driver 를 load / remove 할 때 kernel 의 dmesg 를 사용해서 로그 변화를 살펴보자!

- Hello device driver 를 Ubuntu kernel build 의 소스에 포함시켜 빌드해 보자. 포함시킬 위치는 misc 아래이고 수정할 파일은 Kconfig, Makefile, hello.c 파일이다. CONFIG_MISC_HELLO 를 해당 device 의 config 로 정의하고 모듈 빌드(=m) 를 하자.

```
jammy/
└── drivers/
    └── misc/
        ├── Kconfig
        ├── Makefile
        └── hello.c
```

- hello 디바이스 드라이버를 커널 이미지에 포함시켜 빌드하자. (CONFIG_MISC_HELLO=y)

Ⅱ. 디바이스 드라이버

실습하기 - 2 (Echo Driver)

- Write / Read 가 가능한 driver 를 만들어 적용

- 테스트 방법

```
# 빌드 – echo.ko 파일 생성되어야 함
make

# driver load
sudo insmod echo.ko

# Major 번호 확인 – 결과 예시: 246 echo_device
cat /proc/devices | grep echo_device

# device node 생성
sudo mknod /dev/echo_device c 246 0
sudo chmod 666 /dev/echo_device  # 권한 수정


# 쓰기
echo "Hello, Device Driver!" > /dev/echo_device
# 읽기
cat /dev/echo_device
```

echo.c

```c
// 드라이버에서 마지막 쓴 문자열 기억
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/slab.h>

static char kernel_buf[100] = {0};

ssize_t echo_read(struct file *f, char __user *buf, size_t len, loff_t *off) {
        return simple_read_from_buffer(buf, len, off, kernel_buf, strlen(kernel_buf));
}

ssize_t echo_write(struct file *f, const char __user *buf, size_t len, loff_t *off) {
        memset(kernel_buf, 0, sizeof(kernel_buf));
        if (len > sizeof(kernel_buf) - 1)
                len = sizeof(kernel_buf) - 1;
        if (copy_from_user(kernel_buf, buf, len))
                return -EFAULT;
        return len;
}

struct file_operations fops = {
        .read = echo_read,
        .write = echo_write,
};

static dev_t dev_num;
static struct cdev cdev;

static int __init echo_init(void) {
        alloc_chrdev_region(&dev_num, 0, 1, "echo_device");
        cdev_init(&cdev, &fops);
        cdev_add(&cdev, dev_num, 1);
        printk(KERN_INFO "Echo driver loaded\n");
        return 0;
}

static void __exit echo_exit(void) {
        cdev_del(&cdev);
        unregister_chrdev_region(dev_num, 1);
        printk(KERN_INFO "Echo driver unloaded\n");
}

module_init(echo_init);
module_exit(echo_exit);
MODULE_LICENSE("GPL");
```

Ⅱ. 디바이스 드라이버

Device Tree

- 디바이스 트리란?
  - 리눅스 커널이 하드웨어의 구성을 쉽게 이해할 수 있도록 만든 하드웨어 설명 파일

- 왜 필요한가요?
  - Before: 커널 소스 내에 하드웨어 정보가 직접 포함되어 있어, 장치가 달라질 때마다 커널을 다시 빌드
  - After: 하드웨어 정보만 따로 파일에 저장(*.dts), 커널이 이 파일을 읽어 어떤 장치가 연결됐는지 파악

    → 커널 재빌드 필요없음

    → HW 정보와 코드가 분리되어 유지 관리 용이

    → 같은 커널로 다양한 HW 를 지원 가능

- 사용 예
  - compatible: 이 디바이스 트리가 어떤 장치를 위한 것인지 표시 (라즈베리파이5 등)
  - leds: 이 장치 노드는 LED임을 명시.
  - gpios: 실제 연결된 GPIO 번호를 정의 (17번 GPIO 사용)

```
/dts-v1/;
/ {
    compatible = "raspberrypi,5-model-b";

    leds {
        compatible = "gpio-leds";
        led0 {
            label = "my-led";
            gpios = <&gpio 17 0>;  // GPIO 17 사용
            default-state = "off";
        };
    };
};
```

Ⅱ. 디바이스 드라이버

실습하기 - 3 라즈베리파이에 GPIO 드라이버 구현/적용

- Write / Read 가 가능한 driver 를 만들어 적용

```
# Makefile
obj-m += led_blink.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
```

```c
#include <linux/module.h>
#include <linux/gpio/consumer.h>
#include <linux/timer.h>
#include <linux/of.h>

static struct gpio_desc *led_gpiod;
static struct timer_list blink_timer;
static bool led_on = false;

static void blink_fn(struct timer_list *t)
{
    led_on = !led_on;
    gpiod_set_value(led_gpiod, led_on);
    mod_timer(&blink_timer, jiffies + msecs_to_jiffies(500));
}

static int __init led_blink_init(void)
{
    int ret;
    // label 기반으로 GPIO 디스크립터 얻기
    led_gpiod = gpiod_get(NULL, "my-led", GPIOD_OUT_LOW);
    if (IS_ERR(led_gpiod))
        return PTR_ERR(led_gpiod);

    timer_setup(&blink_timer, blink_fn, 0);
    mod_timer(&blink_timer, jiffies + msecs_to_jiffies(500));
    pr_info("led_blink loaded\n");
    return 0;
}

static void __exit led_blink_exit(void)
{
    del_timer_sync(&blink_timer);
    gpiod_set_value(led_gpiod, 0);
    gpiod_put(led_gpiod);
    pr_info("led_blink unloaded\n");
}

module_init(led_blink_init);
module_exit(led_blink_exit);
MODULE_LICENSE("GPL");
```

Ⅱ. 디바이스 드라이버

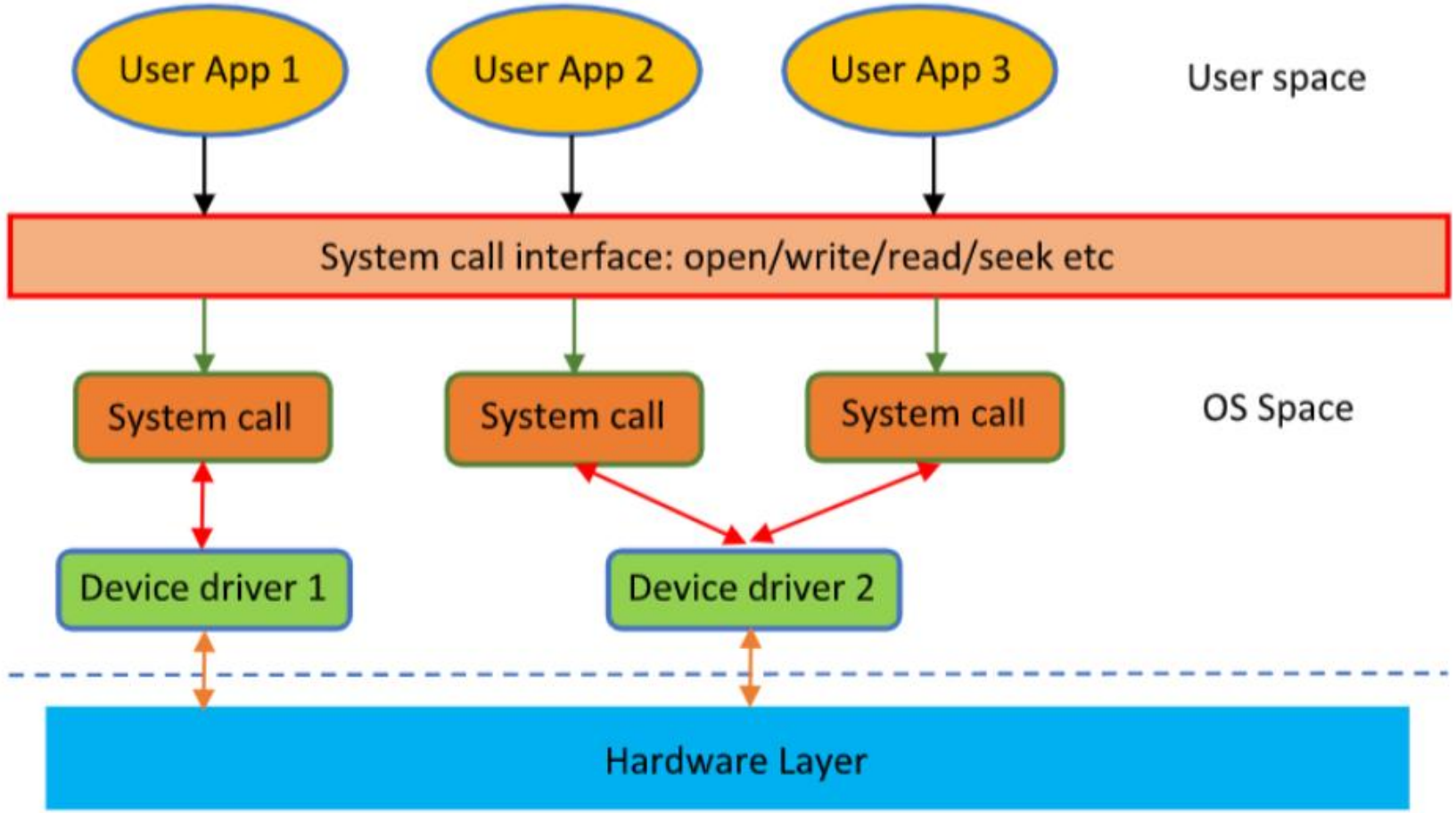
실습하기 - 3 라즈베리파이에 GPIO 드라이버 구현/적용

- Device Tree Update

```
/dts-v1/;
/plugin/;

/ {
    compatible = "brcm,bcm2712";

    fragment@0 {
        target = <&gpio>;
        __overlay__ {
            led_pins: led_pins {
                brcm,pins = <17>;
                brcm,function = <1>; /* GPIO 출력 */
            };
        };
    };

    fragment@1 {
        target-path = "/";
        __overlay__ {
            myled {
                compatible = "gpio-leds";
                led0 {
                    label = "my-led";
                    gpios = <&gpio 17 0>;
                    default-state = "off";
                };
            };
        };
    };
};
```

```
# dts 파일 빌드
dtc -@ -I dts -O dtb -o my_led.dtbo my_led.dts

# dtbo 파일 카피
sudo cp my_led.dtbo /boot/firmware/overlays/

# update 된 dts 가 반영되게 config 파일 수정
sudo vi /boot/firmware/config.txt
dtoverlay=my_led
```

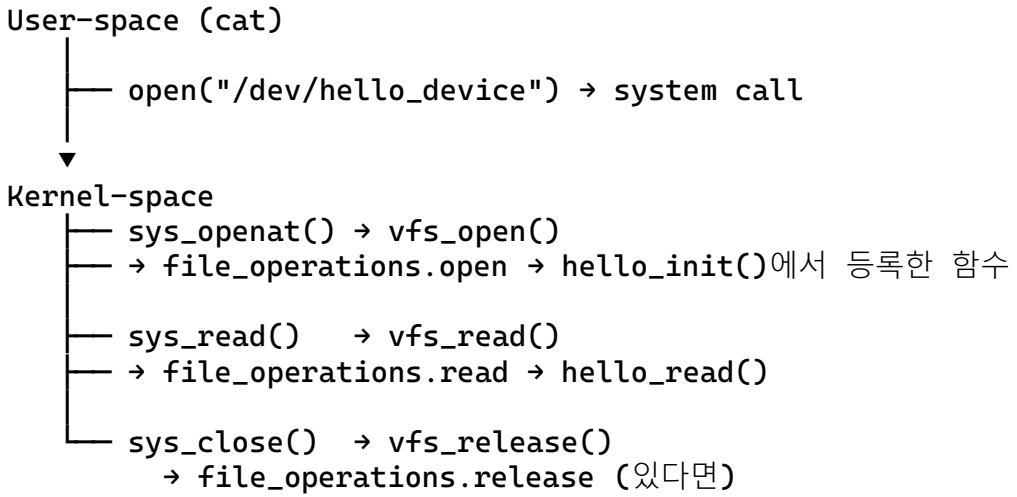
Ⅲ. System Call 과 Device Driver



Ⅲ. System Call 과 Device Driver

Hello device driver 의 System Call Flow

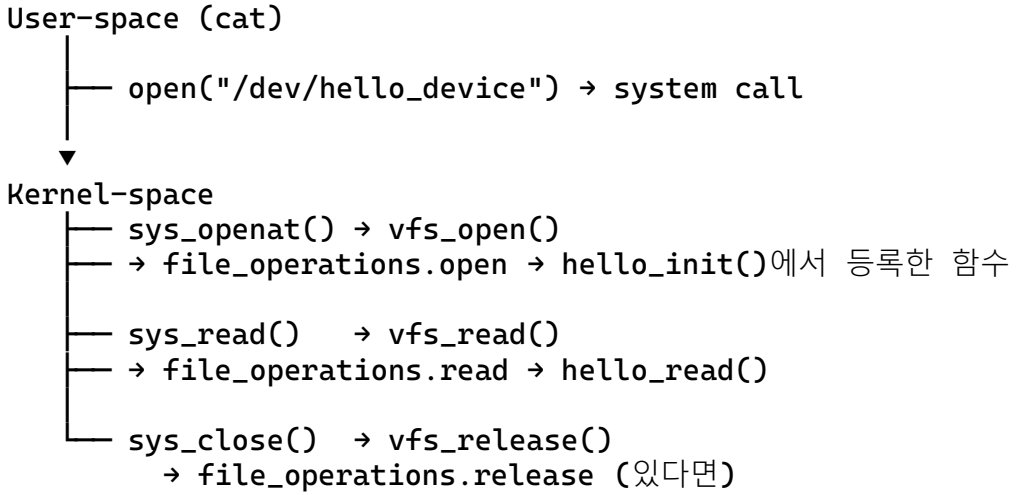
```
# hello device driver init
sudo insmod hello.ko

# hello device node 생성
Sudo mknod /dev/hello_device c $(awk  '$2=="hello_device"  {print $1}'  /proc/devices) 0

# hello device 를 read 할때 발생하는 system call flow
strace -e trace=openat,read,close cat /dev/hello_device
```



Ⅲ. System Call 과 Device Driver

Echo device driver 의 System Call Flow

```
# hello device driver init
sudo insmod echo.ko

# hello device node 생성
Sudo mknod /dev/echo_device c $(awk  '$2=="echo_device"  {print $1}'  /proc/devices) 0

# hello device 를 read 할때 발생하는 system call flow
strace -e trace=openat,write,read,close echo  "Happy Day!"  > /dev/echo_device
strace -e trace=openat,write,read,close cat /dev/echo_device
```



IV. 커널 디버깅

페이지 폴트 상황

NULL 포인터 역참조로 인한 커널 Oops 발생

Makefile

```
obj-m += null_deref.o
EXTRA_CFLAGS += -g

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

```
RIP: 0010:null_deref_init+0x17/0xff0 [null_deref]
```

```
addr2line -e null_deref.ko 0x17
```

null_deref.c

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

static int __init null_deref_init(void)
{
    pr_info("null_deref loaded\n");
    /* NULL 포인터를 강제로 역참조 */
    *(volatile int *)NULL = 0xdeadbeef;
    return 0;
}

static void __exit null_deref_exit(void)
{
    pr_info("null_deref unloaded\n");
}

MODULE_LICENSE("GPL");
module_init(null_deref_init);
module_exit(null_deref_exit);
```

IV. 커널 디버깅

잘못된 명령어 실행

Makefile

```
obj-m += panic_ud2.o
EXTRA_CFLAGS += -g

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

panic_ud2

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

static int __init panic_ud2_init(void)
{
    pr_info("panic_ud2 loaded\n");
    asm volatile ("ud2");  // Invalid opcode
    return 0;
}

static void __exit panic_ud2_exit(void) { }

MODULE_LICENSE("GPL");
module_init(panic_ud2_init);
module_exit(panic_ud2_exit);
```

IV. 커널 디버깅

배열 범위초과 접근

Makefile

```
obj-m += oob_access.o
EXTRA_CFLAGS += -g

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

oob_access.c

```
// oob_access.c
#include <linux/module.h>
#include <linux/slab.h>

static int __init oob_access_init(void)
{
    int i;
    int *arr = kmalloc_array(10, sizeof(int), GFP_KERNEL);
    if (!arr)
        return -ENOMEM;

    pr_info("oob_access loaded, writing out-of-bounds\n");
    /* 정상 범위(0~9)를 벗어난 인덱스에 쓰기 */
    for (i = 0; i < 100; i++)
        arr[i] = i;   /* 여기서 i ≥ 10이면 힙 메모리 오염 발생 */

    kfree(arr);
    return 0;
}

static void __exit oob_access_exit(void)
{
    pr_info("oob_access unloaded\n");
}

MODULE_LICENSE("GPL");
module_init(oob_access_init);
module_exit(oob_access_exit);
```

THANK YOU