



리눅스 디바이스 드라이버 기초

디바이스 드라이버의 역할

■ 디바이스 드라이버의 역할

- 디바이스 드라이버는 하드웨어를 사용 가능하게 만들어 줄 뿐 하드웨어를 어떻게 사용할지에 대한 결정은 응용 프로그램에게 넘겨야 한다.

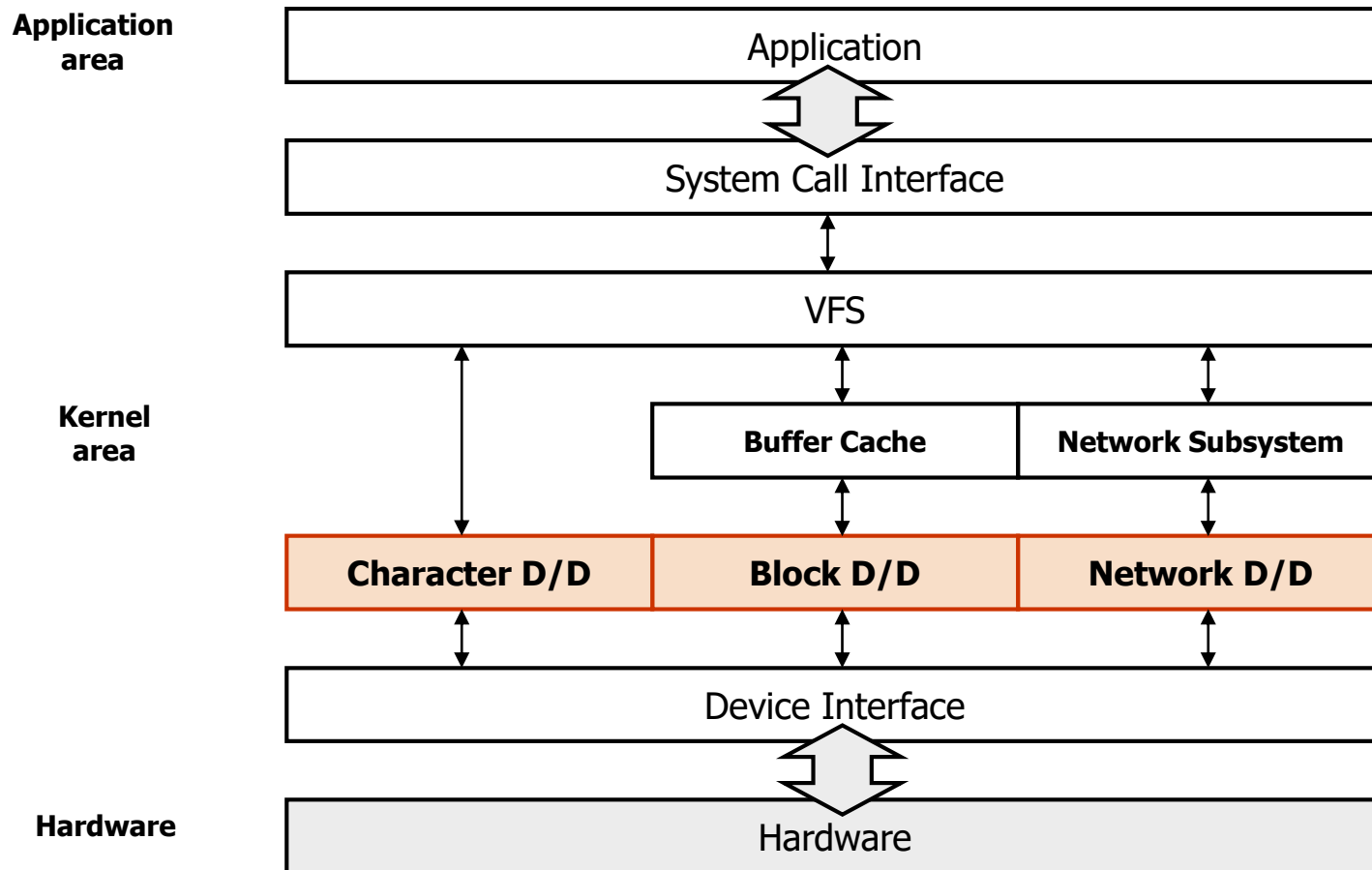
■ 디바이스 드라이버는 온갖 상황을 고려해 균형 있게 작성한다.

- 최대한 유연하고 많은 기능을 사용자에게 제공하려고 할 수록 디바이스 드라이버 제작자는 많은 부분을 구현해야 한다.
- 동기식 비동기식 모두 지원할 것인가?
- 장치를 여러 번 열 것인가?
- 정책 독립성을 제공할 것인가?

■ 제공 되는 디바이스드라이버 유틸리티

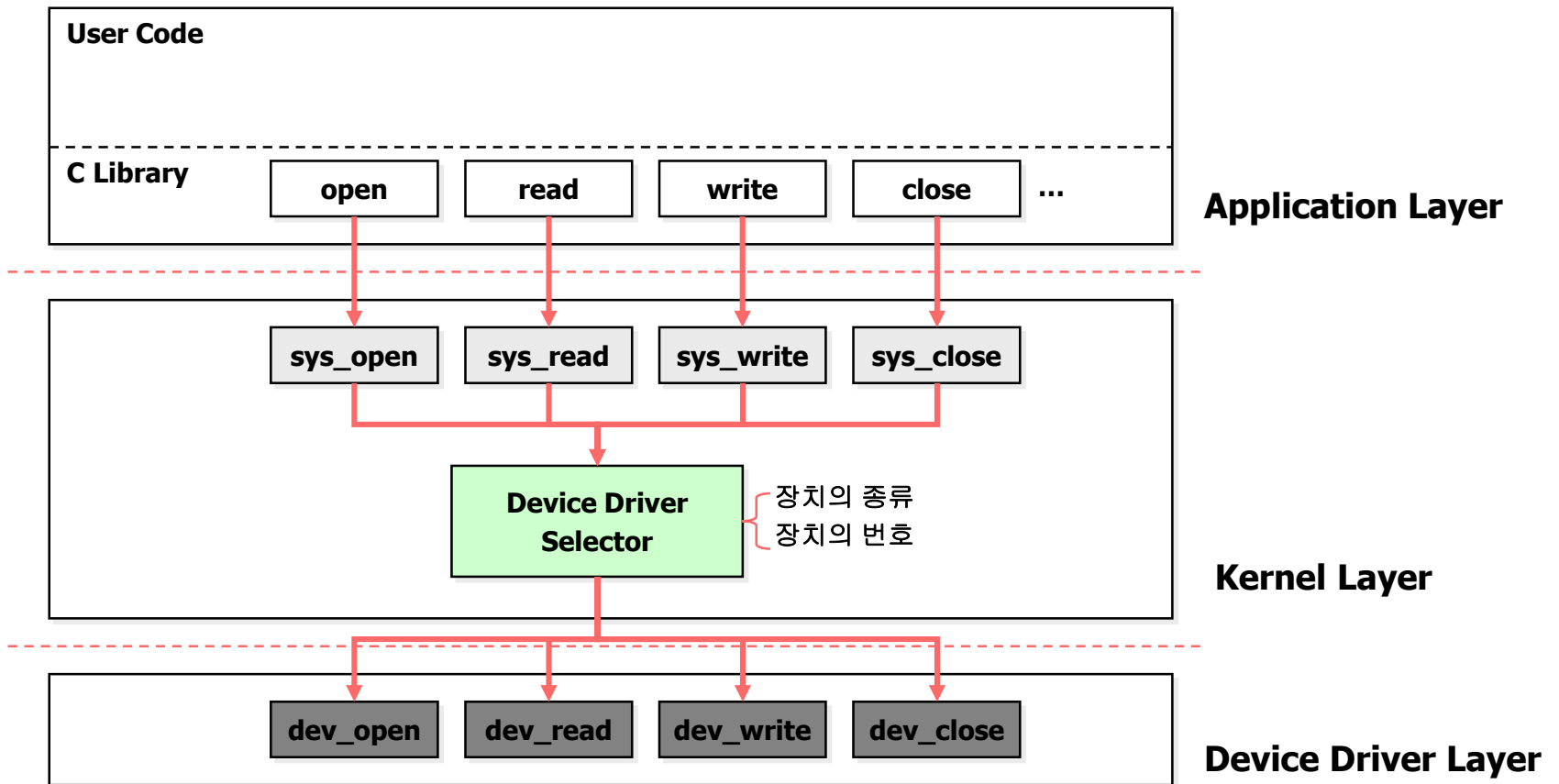
디바이스 제어와 구성을 도울 목적으로 간단한 유틸리티와 디바이스 드라이버를 같이 출시 하는 경우가 있다.

디바이스 드라이버 관점에서 본 리눅스 구조



< 디바이스 드라이버 관점에서 본 리눅스 구조 >

응용프로그램 관점에서 본 리눅스 구조



커널 메시지 출력 (1)

■ printk()

printf()와 유사하지만 커널의 메시지를 출력하고 관리할 수 있는 특성이 있다.

- 메시지 기록 관리를 위한 로그 레벨의 지정
- 원형 큐 구조의 관리
- 출력 디바이스의 다중 지정
- 콘솔에서 확인하거나 dmesg 명령을 이용해서 로그 파일을 확인

■ 로그 레벨 지정

로그 레벨은 printk() 함수에 전달되는 문자열의 선두 문자에 “<1>” 과 같이 숫자로 등급을 표현한다. linux/kernel.h 에 정의된 선언문을 이용이 바람직하다.

상수 선언문	의미
#define KERN_EMERG	<0> 시스템이 동작하지 않는다
#define KERN_ALERT	<1> 항상 출력된다.
#define KERN_CRIT	<2> 치명적인 정보
#define KERN_ERR	<3> 오류 정보
#define KERN_WARNING	<4> 경고 정보
#define KERN_NOTICE	<5> 정상적인 정보
#define KERN_INFO	<6> 시스템 정보
#define KERN_DEBUG	<7> 디버깅 정보

커널 메시지 출력 (3)

- /proc/kmsg
 - 커널 메시지가 발생할 때마다 관찰할 수 있다.
 - `cat /proc/kmsg`
- printk() 사용 시 주의점
 - printk를 과도하게 사용하지 않는다. → 실행시간이 길다
 - 개행 문자가 있어야 출력을 시작한다. → '\n'을 포함 하도록 한다.

실습2: 드라이버 정보 매크로 사용 (1)

- `init_module()` 와 `cleanup_module()` 함수 변경 매크로(`linux/init.h` 에 정의 되어 있음)

`module_init()`

`module_exit()`

```
/* hello-2.c – Demonstrating the module_init() and module_exit() macros.  
This is preferred over using init_module() and cleanup_module(). */
```

```
#include <linux/module.h> /* Needed by all modules */  
#include <linux/kernel.h> /* Needed for KERN_INFO */  
#include <linux/init.h> /* Needed for the macros */
```

```
static int __init hello_2_init(void)  
{
```

```
    printk(KERN_INFO "Hello, world 2\n");  
    return 0;
```

```
}
```

```
static void __exit hello_2_exit(void)
```

```
{
```

```
    printk(KERN_INFO "Goodbye, world 2\n");
```

```
}
```

```
module_init(hello_2_init);
```

```
module_exit(hello_2_exit);
```

```
MODULE_LICENSE("GPL");
```

실습2: 드라이버 정보 매크로 사용 (2)

- MODULE_LICENSE()
GPL, GPL v2, Dual BSD/GPL, Proprietary 등
- MODULE_DESCRIPTION()
모듈의 하는 일을 설명
- MODULE_AUTHOR()
모듈 제작자
- MODULE_SUPPORTED_DEVICE()
모듈이 지원하는 장치의 타입

실습2: 드라이버 정보 매크로 사용 (3)

```
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */
#include <linux/init.h> /* Needed for the macros */

#define DRIVER_AUTHOR "Peter Jay Salzman <p@dirac.org>"
#define DRIVER_DESC "A sample driver"

static int __init init_hello_3(void)
{
    printk(KERN_INFO "Hello, world 4\n");
    return 0;
}

static void __exit cleanup_hello_3(void)
{
    printk(KERN_INFO "Goodbye, world 4\n");
}

module_init(init_hello_4);
module_exit(cleanup_hello_4);
/* Get rid of taint message by declaring code as GPL.*/
MODULE_LICENSE("GPL");
MODULE_AUTHOR(DRIVER_AUTHOR); /* Who wrote this module? */
MODULE_DESCRIPTION(DRIVER_DESC); /* What does this module do */
MODULE_SUPPORTED_DEVICE("testdevice");
```



캐릭터 디바이스 드라이버

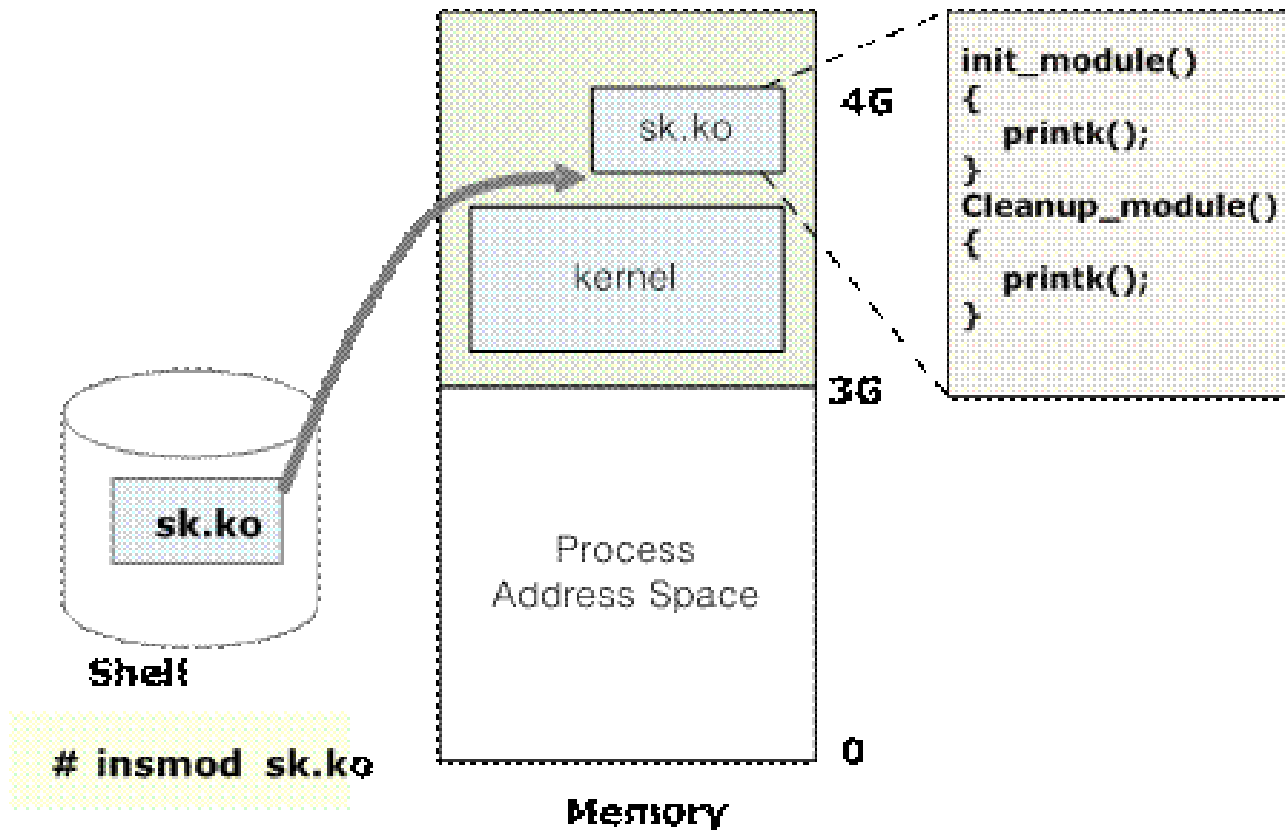


캐릭터 디바이스 드라이버의 구현

■ 캐릭터 디바이스

- 키보드 같은 디바이스는 바이트 단위로 처리되고 써넣은 데이터는 보존될 수도, 보존되지 않을 수도 있다.
- 시간에 따라 연속적으로 발생할 수도 있고, 발생한 데이터의 끝을 알 수 없다.
- fs/char_dev.c 에 정의
`struct char_device_struct chrdevs[CHRDEV_MAJOR_HASH_SIZE];`

캐릭터 디바이스 드라이버의 구현



<Fig. sk.o의 메모리 로드>

실습1: 기본 뼈대 구성 – SK.C

```
/******  
* Filename: sk.c  
* Title: Skeleton Device  
* Desc: module_init, module_exit  
*****/  
#include <linux/module.h>  
#include <linux/init.h>  
  
MODULE_LICENSE("GPL");  
  
static int sk_init(void)  
{  
    printk("SK Module is up... \n");  
    return 0;  
}  
  
static void sk_exit(void)  
{  
    printk("The module is down... \n");  
}  
  
module_init(sk_init);  
module_exit(sk_exit);
```

실습1: 기본 뼈대 구성 – MAKEFILE

```
obj-m := sk.o
KDIR  := (커널 소스 디렉토리 또는 모듈 build 디렉토리)
PWD := $(shell pwd)

all:
    make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -C $(KDIR) \
    M=$(PWD) modules

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

■ 컴파일 및 모듈 적재와 해제

```
# sudo insmod sk.ko
SK Module is up...
# lsmod
# sudo rmmod sk
The module is down...
```

캐릭터 디바이스 등록

■ 주요 구현 순서

- `file_operations`의 레코드를 할당한다.
- `register_chrdev()`함수를 `init_module()`함수에 삽입한다.
- open file operation(함수포인터)인 `sk_open()` 함수를 만든다.

캐릭터 디바이스 드라이버 API

■ 등록과 해제 관련 API

- `register_chrdev` 문자 디바이스의 등록
- `unregister_chrdev` 문자 디바이스를 해제
- `register_chrdev_region` 디바이스 번호 요구 등록
- `unregister_chrdev_region` 사용중인 디바이스 번호 해제

캐릭터 디바이스 등록

■ register_chrdev()

- 유일한 Major 번호 할당.

`int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);`

major : 요청할 major 번호

name : 디바이스 이름. /proc/devices에 나타난다.

fops : file operation function pointer

- Major 번호 : 기존 것과 겹치지 않는 번호 이용
- 문자 디바이스 드라이버 테이블 `chrdevs[]`의 인덱스로 사용됨.

MAJOR 번호의 결정

- Manual allocation

현재 사용 중이지 않은 major 번호를 알아낸 후 직접 하드 코딩 함.

- Dynamic allocation

- major 값에 0을 넣으면 kernel이 빈 번호를 알아서 할당

```
result = register_chrdev(scull_major, "scull", &scull_fops);
if (result < 0) {
    printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
    return result;
}
if (scull_major == 0) scull_major = result; /* dynamic */
```

- 단점 : 미리 디바이스 파일을 만들어 둘 수 없음.
- 해결책 : 장치번호 등록 후 `/proc/devices`를 읽어 major 번호를 얻고, 장치 파일을 만드는 스크립트를 활용한다.

스크립트를 이용한 MAJOR 번호의 동적 할당 예

```
#!/bin/sh
DEV_FILE=gpio
DEV_NAME=GPIO

/sbin/rmmod $DEV_FILE &> /dev/null
/sbin/insmod $DEV_FILE.o

set $(grep -r $DEV_NAME /proc/devices | cut -d ' ' -f1)
devmajor=$1

set $(grep -r $DEV_NAME /proc/devices | cut -d ' ' -f2)
devname=$1

/bin/rm /dev/$devname &> /dev/null
/bin/mknod /dev/$devname c $devmajor 0
```

<스크립트를 이용한 동적 할당의 예 - **inst_device.sh** >

캐릭터 디바이스 해제

- unregister_chrdev()
 - `int unregister_chrdev(unsigned int major, const char *name);`
 - 디바이스 파일도 삭제해야 한다.
 - Major 번호 삭제 없이 장치 드라이버를 삭제하면 문제 발생(Oops 메시지 출현)
- 스크립트를 이용한 드라이버 모듈 제거

```
#!/bin/sh
```

```
DEV_NAME=gpio
```

```
DEV_FILE=GPIO
```

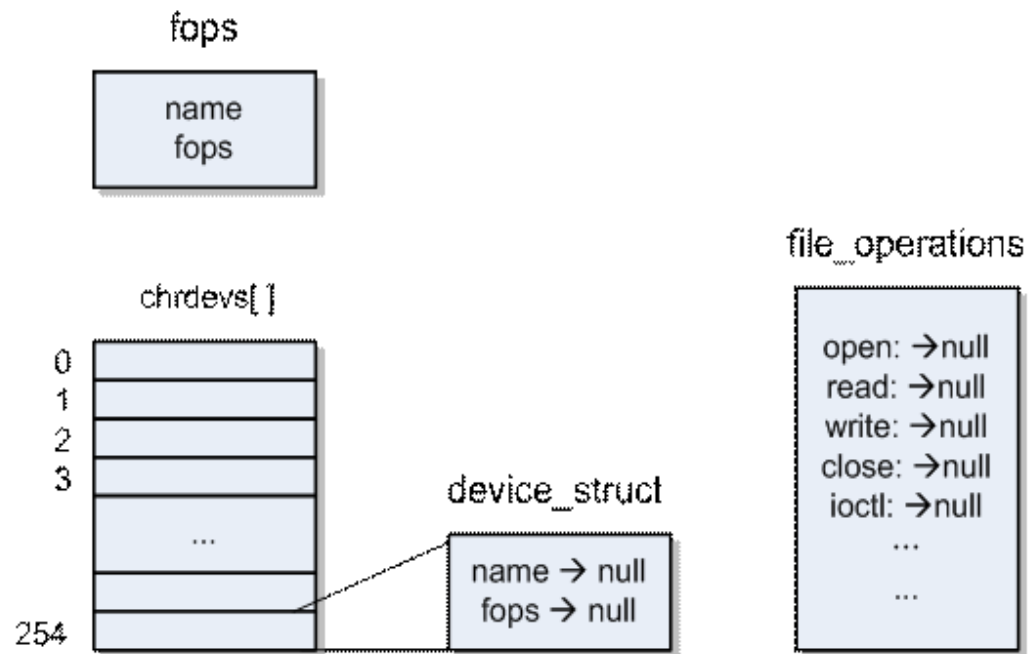
```
/sbin/rmmod $DEV_NAME
```

```
/bin/rm /dev/$DEV_FILE
```

```
/bin/rm /root/*
```

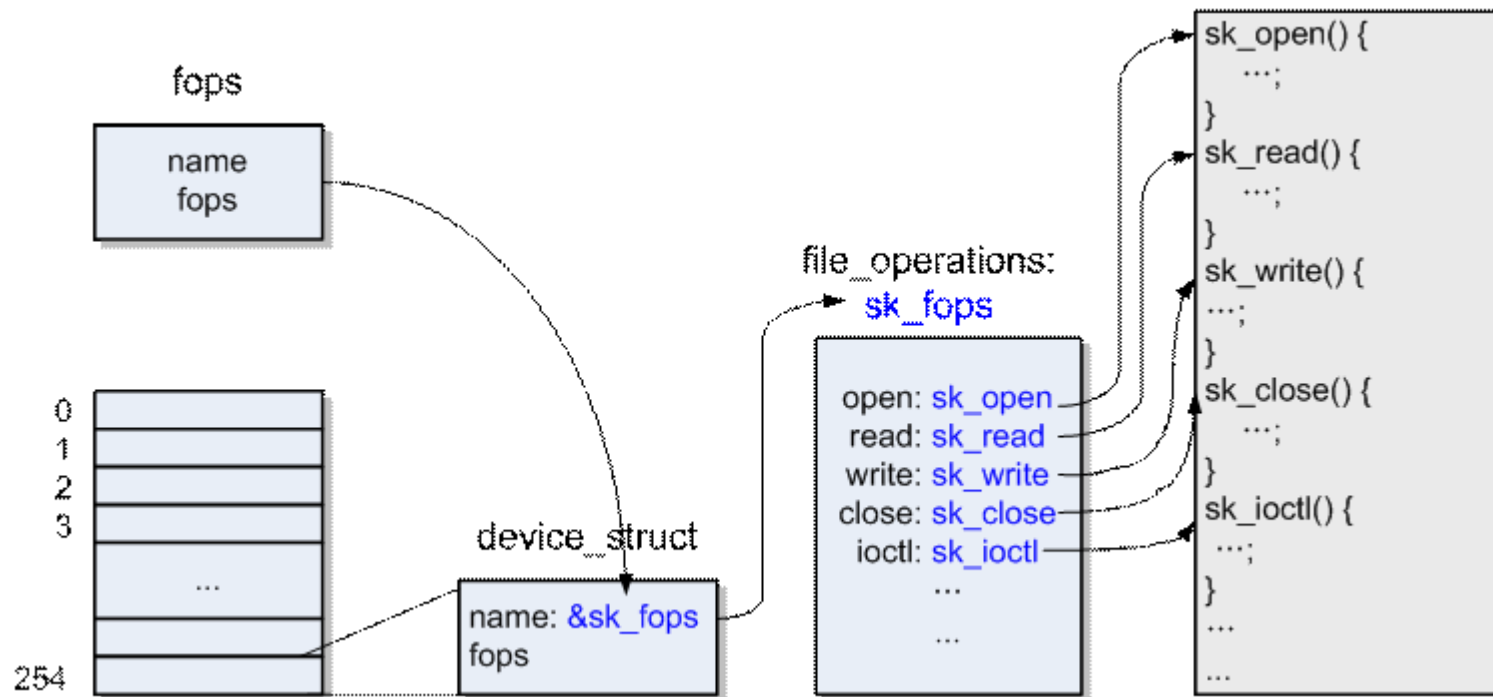
REGISTER_CHRDEV 구조

- register_chrdev() 실행 전 상태



REGISTER_CHRDEV 구조

- register_chrdev() 실행 후 상태



디바이스 타입 – DEV_T와 KDEV_T

- Major/Minor numbers가 조합된 수형
major 번호와 쌍을 이뤄 하나의 장치 번호를 구성.

Major(**12bits**), Minor(**20bits**)

- macros(include/linux/kdev_t.h)

```
MAJOR(kdev_t dev);      /* 주 번호 추출 */  
MINOR(kdev_t dev);      /* 부 번호 추출 */  
MKDEV(int ma, int mi);  /* 번호 설정 */  
kdev_t_to_nr(kdev_t dev); /* kdev_t->dev_t */  
to_kdev_t(int dev);      /* dev_t->kdev_t */
```

실습2: 디바이스 등록 및 해제 – SK.C

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

MODULE_LICENSE("GPL");

int result;
/* file_operations의 레코드를 할당한다. */
struct file_operations sk_fops;

static int sk_init(void)
{
    printk("SK Module is up... \n");
    /* 문자 디바이스의 등록 */
    result = register_chrdev(0, "SK", &sk_fops);
    printk("major number=%d\n", result);
    return 0;
}

static void sk_exit(void)
{
    printk("The module is down... \n");
    unregister_chrdev(result, "SK");
}

module_init(sk_init);
module_exit(sk_exit);
```


실습2: 디바이스 등록 및 해제 – SK.C

- 컴파일 및 적재 실행

```
# make  
# insmod sk.ko  
# lsmod  
# cat /proc/devices  
# rmmod sk
```

초기화 및 해제 작업의 기본 형식

```
static xxx_info *info = NULL;

int xxx_init(void)
{
    xxx_probe( ... // 하드웨어 검출 처리 및 에러처리
    xxx_setup( ... // 하드웨어 초기화
    register_chrdev( ... // 디바이스 드라이버 등록
    info = kmalloc( ... // 디바이스 드라이버에 동작에 필요한 내부 구조체 메모리 할당

    xxx_setupinfo( ... // 여러 프로세스가 디바이스 하나에 접근하는 경우에 필요한 사전 처리
    xxx_setupminor( ... // Major번호에 종속된 Minor번호를 관리하기 위한 사전 처리
}

void xxx_exit(void)
{
    kfree( ... // 디바이스 드라이버에 할당된 모든 메모리 해제
    unregister_chrdev( ... // 디바이스 드라이버의 해제
    xxx_shutdown( ... // 하드웨어 제거에 따른 처리
}

module_init(xxx_init);
module_exit(xxx_exit);
```

장치 파일의 생성

■ 목 적

- Device Driver 로 사용할 장치 파일을 만든다.
- 유저가 Device Driver File을 이용해 장치를 사용할 수 있다.
- 장치 파일을 생성하기 위해서는 mknod를 통해서 장치 파일을 생성한다.

■ 리눅스 커널에서 파일이란?

- 절대경로 혹은 상대경로로 표시되는 path와 file명으로 구성되어 있다.
(PATH: dentry, File: inode가 관리)
- 커널은 항상 절대경로와 상대경로를 알고 있다.
- 커널은 파일을 경로와 파일명으로 인식한다.

디바이스 이름 생성

■ mknod의 사용

- 일반적으로 /dev 경로에 생성 한다.
- 장치파일은 user와 Device Driver를 연결해 주는 매개체이다.
- devfs에서 관리된다.

```
# mknod /dev/name type major minor
```

Minor number

Major number

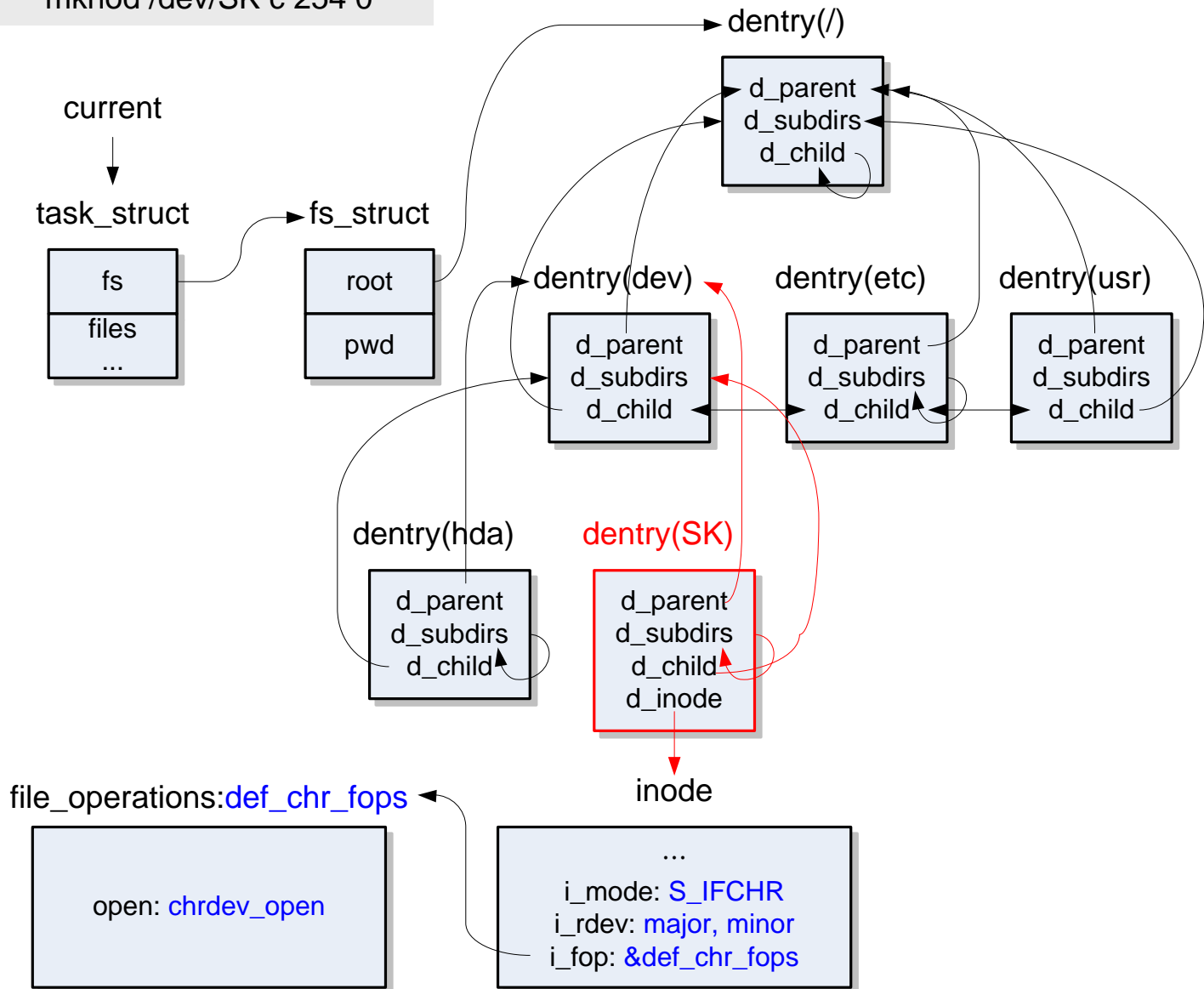
b(block) or c(character)

Device Name

■ 파일의 구성

- dentry 구조체 : 경로를 관리하는 구조체
- inode 구조체 : 파일을 관리하는 구조체
- 부모는 dentry.d_subdirs, 자식들은 dentry.d_child와 double linked list로 구성되어 있다.

mknod /dev/SK c 254 0



실습3: 디바이스 파일 생성 – 실행

- 실습 실행 순서
 - sk.c 파일: 변동사항 없음

```
# insmod sk.ko
# mknod /dev/SK c 253 0
# ls /dev -al | grep SK
    crw-r--r--  1 root   root   253,  0 Jan  1 05:41 SK
# rm /dev/SK
# rmmmod sk
```

FILE_OPERATIONS

■ 개요

- 문자 디바이스 드라이버와 응용 프로그램을 연결하는 고리.
- linux/fs.h에서 정의하는 이 구조체는 함수 포인터 집합이다.
- 특정 동작 함수를 구현하여 가리켜야 한다.
- 지정하지 않으면 NULL로 남겨 두어야 한다.

FILE_OPERATIONS

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
                                     unsigned long, unsigned long);
};
```


FILE_OPERATIONS 구조체 필드

- **struct module *owner;**

파일 오퍼레이션의 소유자를 나타낸다. 보통 <linux/module.h>에 정의되어 있는 THIS_MODULE 매크로를 사용해 초기화 한다.

- **loff_t (*llseek) (struct file *, loff_t, int);**

디바이스 드라이버의 파일 포인터 위치를 강제로 이동 시키는 함수를 지정한다.

- **ssize_t (*read) (struct file *, char *, size_t, loff_t *);**

디바이스에서 자료를 읽는데 사용한다. NULL이면 -EINVAL 반환

- **ssize_t (*write) (struct file *, const char *, size_t, loff_t *);**

자료를 디바이스로 보낸다. NULL이면 -EINVAL 반환

- **unsigned int (*poll) (struct file *, struct poll_table_struct *);**

다중 입출력 처리를 가능하게 해주는 poll, epoll, select의 백엔드이다.

- **int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);**

디바이스 관련 명령들을 제어할 수 있다.

FILE_OPERATIONS 변수선언 스타일

■ 선언 예

```
struct file_operations xxx_fops = {  
    .owner  = THIS_MODULE,  
    .llseek = xxx_llseek,  
    .read   = xxx_read,  
    .ioctl  = xxx_ioctl,  
    .open   = xxx_open,  
    .release = xxx_release,  
};
```

실습4: OPEN과 RELEASE 기능 추가

■ 실습

- 디바이스 드라이버가 구동 가능하도록 시스템 콜을 구현한다.
- 디바이스 드라이버 애플리케이션 작성
- sk_app.c 응용프로그램을 작성해 사용해 본다.

OPEN, RELEASE의 역할

■ open 메소드의 역할

- 디바이스 관련 오류 확인(디바이스가 준비되지 않았거나 이와 유사한 하드웨어 문제)
- 처음으로 디바이스를 열 경우 디바이스 초기화
- 필요에 따라 f_op 포인터 갱신
- 필요에 따라 자료구조를 할당하고 filp->private_data에 들어갈 값을 채움

■ release 메소드의 역할

- device_close로 부르는 경우도 있다
- open이 filp->private_data에 할당한 데이터의 할당 삭제
- 마지막 close 호출 시 디바이스 종료

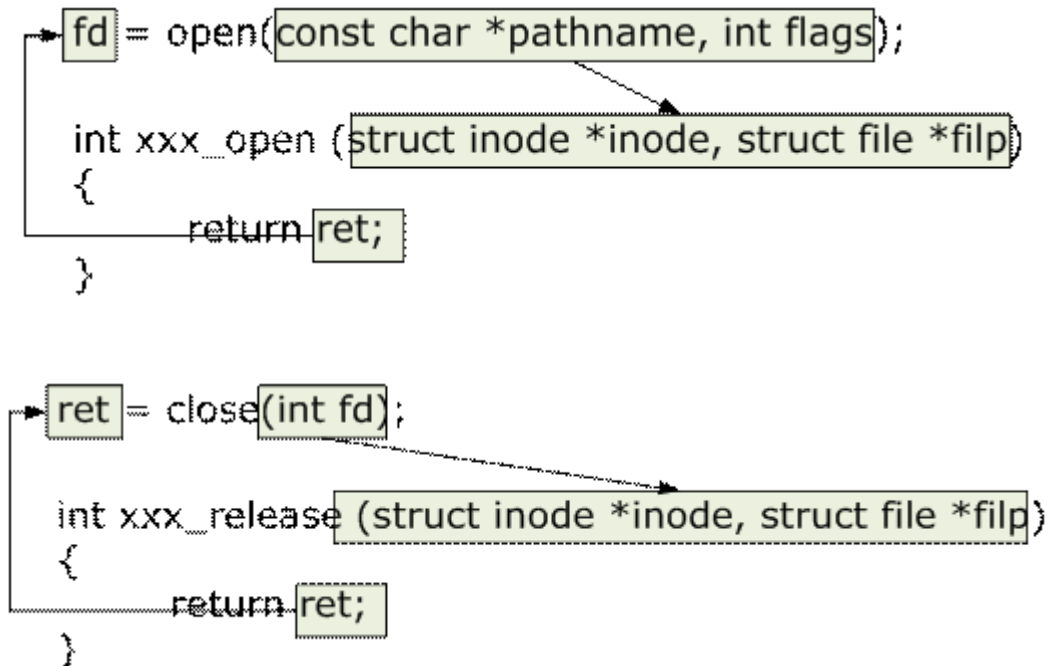
OPEN, RELEASE 호출의 기본 처리

- open은 다음 항목을 처리해야 한다.
 - 디바이스 드라이버가 처음 열렸을 때 하드웨어 초기화
 - 디바이스 드라이버의 동작에 필요한 에러 체크
 - ENODEV 하드웨어가 존재하지 않는다.
 - ENOMEM 커널 메모리가 부족하다.
 - EBUSY 디바이스가 이미 사용 중이다.
 - Minor 번호에 대한 처리가 필요한 경우
file_operation 구조체를 갱신
 - 프로세스 별 메모리 할당과 초기화
보통 file 구조체 filp의 private_data에 등록하여 사용한다.
Ex) filp->private_data = vmalloc(1024)

매개변수 반환 값 상관관계

■ 호출되는 함수의 매개변수 반환 값 상관관계

- open()함수의 pathname과 flags는 xxx_open()함수에 직접적으로 전달되지는 않는다.
→ 사전 처리를 하기 때문
- 적절히 inode와 filp변수에 배분되어 전달된다.



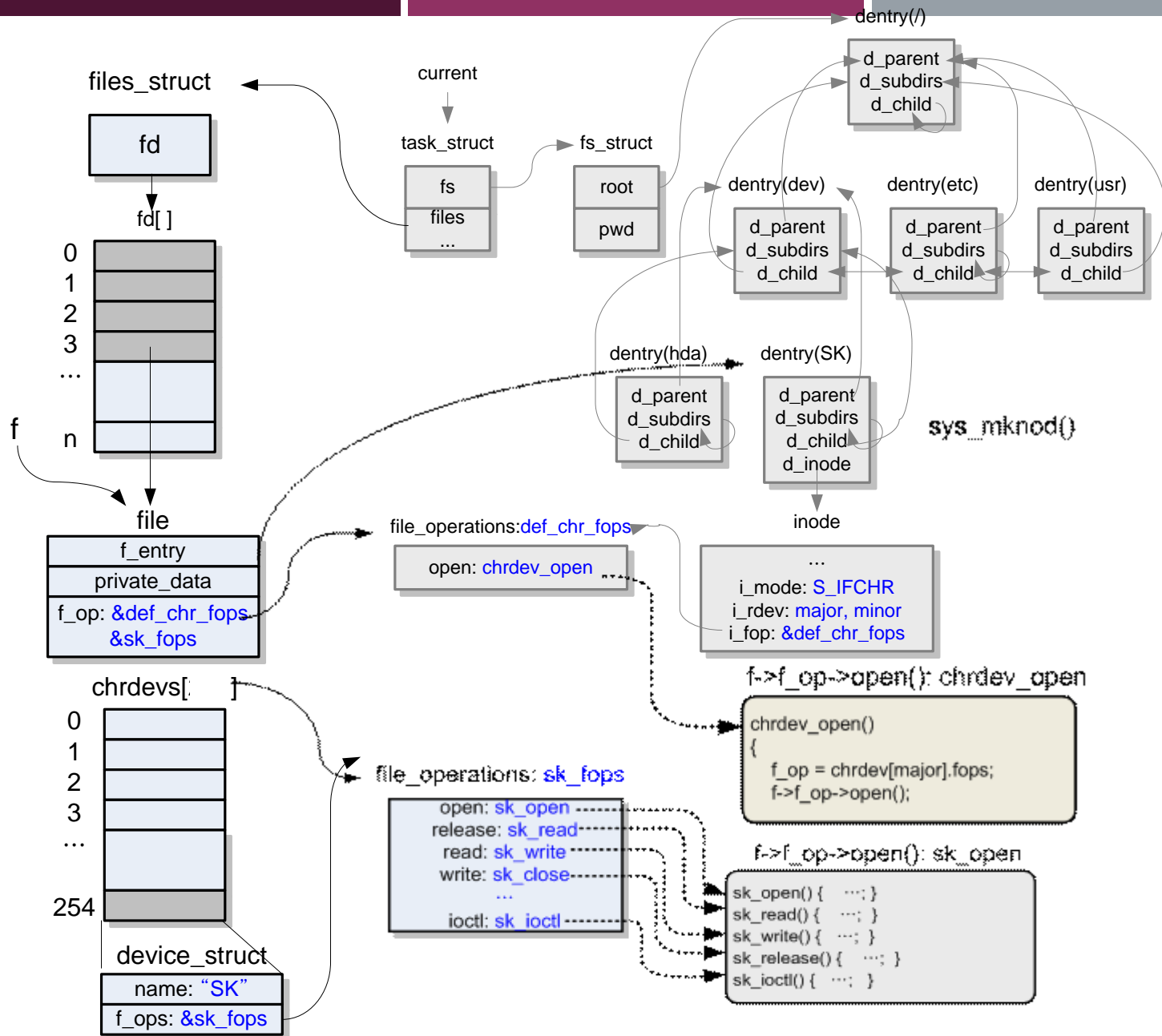
FIE STRUCTURE

- file structure in <linux/fs.h>
 - 디바이스 드라이버가 사용하는 중요 커널 자료구조
 - 오픈 파일에 하나씩 할당됨.

```
struct file {  
    struct file *      f_next, **f_pprev;  
    struct dentry *    f_dentry;  
    struct file_operations *f_op;  
    mode_t             f_mode;  
    loff_t             f_pos;  
    unsigned int       f_count, f_flags;  
    unsigned long      f_reada, f_ramax, f_raend,  
                      f_ralen, f_rawin;  
    struct fown_struct f_owner;  
    unsigned int       f_uid, f_gid;  
    int               f_error;  
    unsigned long      f_version;  
    /* needed for tty drivers, and maybe others */  
    void *             private_data;  
};
```

FIE STRUCTURE

- **mode_t f_mode**
 - FMODE_READ/FMODE_WRITE 비트로 결정
 - 파일 시스템 호출에서 R/W 권한이 미리 검사됨.
- **loff_t f_pos**
 - 현재의 읽기/쓰기 위치.
 - lseek / read / write 메소드에서 갱신해주어야 함.
- **unsigned int f_flags**
 - O_RDONLY, O_NONBLOCK, O_SYNC, ...
 - 드라이버는 자신의 동작 제어를 위해 이 필드를 참조해야 함.
- **struct file_operations *f_op**
 - 드라이버 메소드 테이블을 가리키는 포인터
 - method overriding될 수도 있음
- **void *private_data**
 - 드라이버/모듈들의 내부 데이터 유지.
 - 커널이 file 구조체를 파괴하기 전에 미리 파괴해야 함.



실습4: OPEN과 RELEASE 추가 – SK.C

```
/*
 * Filename: sk.c
 * Title: Skeleton Device
 * Desc: Implementation of system call
 */
#include <linux/module.h>
#include <linux/init.h>
#include <linux/major.h>
#include <linux/fs.h>

MODULE_LICENSE("GPL");

int result;

/* Define Prototype of functions */
int sk_open(struct inode *inode, struct file *filp);
int sk_release(struct inode *inode, struct file *filp);

/* Implementation of functions */
int sk_open(struct inode *inode, struct file *filp)
{
    printk("Device has been opened...\n");

    /* H/W Initialization */

    //MOD_INC_USE_COUNT; /* for kernel 2.4 */

    return 0;
}
```

실습4: OPEN과 RELEASE 추가 – SK.C

```
int sk_release(struct inode *inode, struct file *filp)
{
    printk("Device has been closed...\n");

    return 0;
}
```

```
struct file_operations sk_fops = {
    .open      = sk_open,
    .release   = sk_release,
};
```

```
static int __init sk_init(void)
{
    printk("SK Module is up... \n");

    result = register_chrdev(0, "SK", &sk_fops);
    if (result < 0 ) {
        printk("Couldn't get a major number...\n");
    }
    printk("major number=%d\n", result);
    return 0;
}

static void __exit sk_exit(void)
{
    printk("The module is down...\n");
    unregister_chrdev(result, "SK");
}
```

```
module_init(sk_init);
module_exit(sk_exit);
```

실습4: OPEN과 RELEASE 추가 – SK_APP.C

```
/******  
 * Filename: sk_app.c  
 * Title: Skeleton Device Application  
 * Desc: Implementation of system call  
 *****/  
#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h>  
#include <fcntl.h>  
  
int main(void)  
{  
    int fd;  
  
    fd = open("/dev/SK", O_RDWR);  
    printf("fd = %d\n", fd);  
  
    if (fd < 0) {  
        perror("/dev/SK error");  
        exit(-1);  
    }  
    else  
        printf("SK has been detected...\n");  
  
    getchar();  
    close(fd);  
  
    return 0;  
}
```

실습4: OPEN과 RELEASE 추가 – 실행

■ 실행 순서

- 모듈 적재 (make clean, make, insmod)
- 장치파일 생성 (mknod)
- sk_app 컴파일 (gcc sk_app.c -o sk_app)
- 실습 후 장치 파일 제거 및 모듈 제거

■ 실행결과 예시

```
# ./sk_app
Device has been opened...
fd = 3
SK has been detected...
Device has been opened...
```

■ 작성 포인트

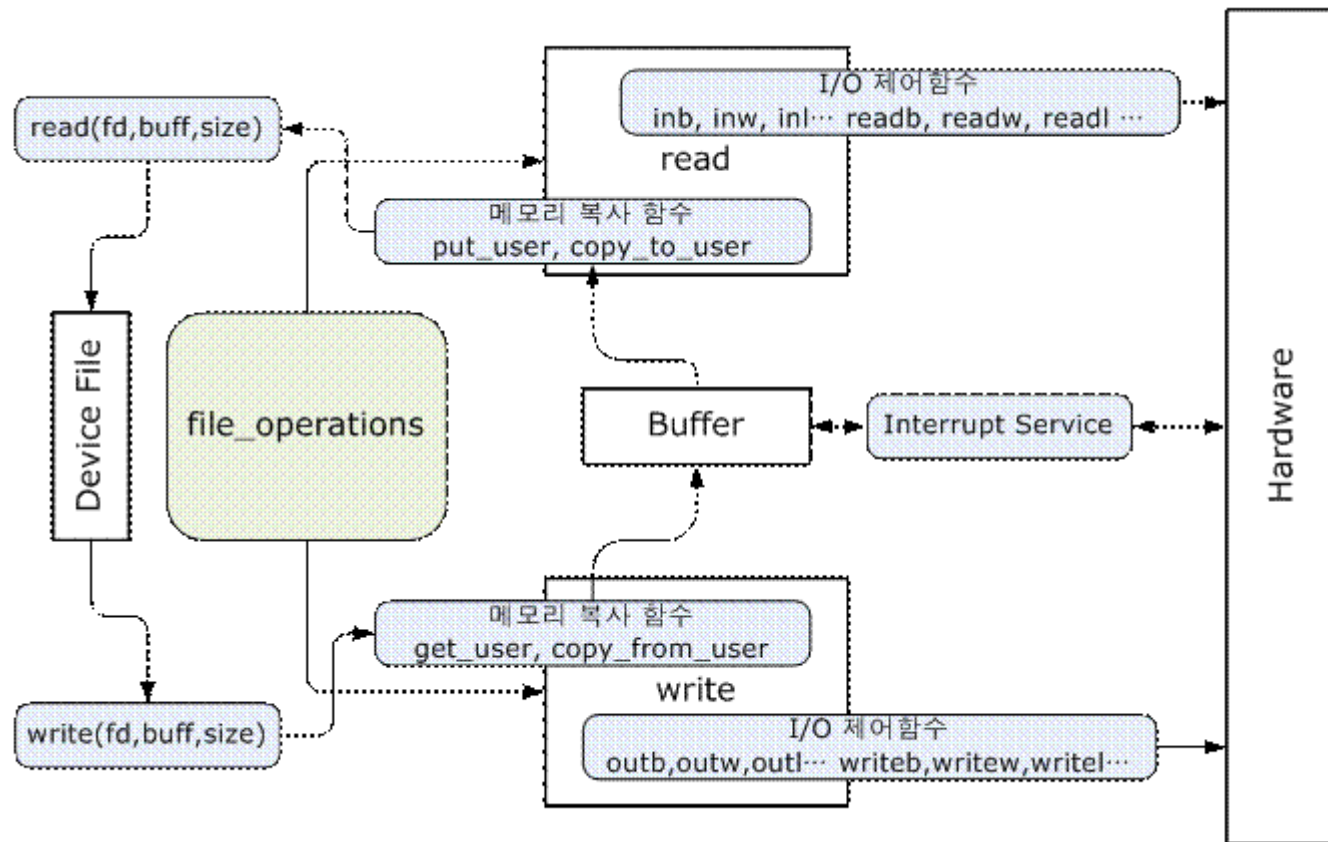
open의 수행 순서는 다음과 같다.

open → library → S/W Interrupt → System call → VFS → CHR. Device
File → Device Driver

읽기와 쓰기의 구현

- read와 write를 구현하기 위해 알아야 할 사항.
 - 사용자 메모리 공간과 커널 메모리 공간 사이의 데이터 이동
 - 처리 조건이 되지 않을 때의 처리
 - 하드웨어 제어함수
 - 여러 프로세스가 동시에 접근했을 때의 경쟁 처리
 - 인터럽트 서비스 함수와의 경쟁 처리

디바이스 드라이버 읽기 쓰기 구조



< 디바이스 드라이버 읽기 쓰기의 구조 >

실습5: WRITE() SYSTEM CALL 추가하기

■ 구현 내용

- write() 함수의 용도를 이해하고, write()를 구현한다.
- 응용 프로그램에서 디바이스에 데이터를 전달하는 것이 목적이다.

하드웨어에서 전달될 데이터일 수도 있고, 디바이스 드라이버 동작에 영향을 미치는 데이터일 수도 있다

■ 기본 구현 개념

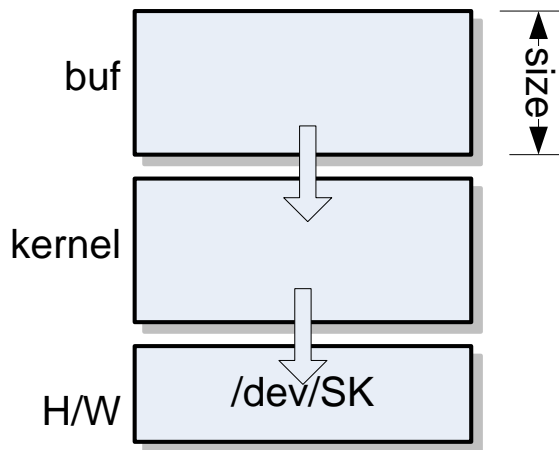
- 응용 프로그램은 디바이스 파일에 write() 함수를 이용해 하드웨어에 데이터 쓰기를 시도하고, 커널은 file_operations 구조체 write 필드에 지정된 xxx_write() 함수가 수행 되도록 호출한다.
- 이때 디바이스 파일은 쓰기를 허가하는 모드로 열려있어야 한다. 그렇지 않으면 응용 프로그램에서 디바이스 파일에 write() 함수를 수행해도 호출되지 않는다.

실습5: WRITE() SYSTEM CALL 구현 개념

■ 매개변수 사용

char *buf: 응용 프로그램에서 전달한 버퍼의 주소

size_t count: 응용 프로그램에서 요청한 데이터의 크기



```
n = write(fd, buf, size);
```

```
sk_write(file, buf, count, f_ops) {  
    copy_from_user(data, buf, count);  
    // H/W access  
}
```

outb(), writeb() (8bit 단위)
outw(), writew() (16bit 단위)
outl(), writel() (32bit 단위)

```
*addr = data;  
    ▼  
ldr r0, =data  
ldr r1, =addr  
str r1, [r0]
```

WRITE 함수의 구현

- write()함수의 매개변수의 의미

응용 프로그램에서 디바이스에 데이터를 전달하는 것이 목적.

- 주요 메서드

- `ssize_t xxx_write(struct file *filp, char *buff, size_t count, loff_t *offp)`
사용자 영역인 buff에서 count 바이트 만큼 읽은 후 디바이스의 offp 위치로 저장
- `unsigned long copy_from_user(void *to, const void *from, unsigned long count);`
사용자 메모리 from을 커널 메모리 to로 count만큼 복사한다.
- `get_user(x, ptr)`
x 변수에 ptr의 사용자 메모리값을 대입한다.

WRITE 함수의 구현

- 주요 에러

EAGAIN: O_NONBLOCK로 열렸지만 write()호출 시 즉시 처리할 수 있는 상황이 아니다.

EIO: I/O 에러가 발생했다.

EFAULT: buf가 접근할 수 없는 주소 공간을 가리키고 있다.

ENOSPC: 데이터를 위한 공간이 없다.

WRITE 함수의 구현

■ struct file *filp

- 읽기와 쓰기에 전달되는 file 구조체 변수의 선두 주소를 담은 filp는 디바이스 파일이 어떤 형식으로 열렸는가에 대한 정보를 담고 있다.
- 자주 사용되는 필드는 다음과 같다.
- unsigned int f_flags;
- 가장 많이 참조되는 것으로 O_RDONLY, O_NONBLOCK 또는 O_NDELAY, O_SYNC가 있다.
- O_RDONLY는 읽기 모드로 열리는 조건이 O_RDWR와 관련이 있기 때문에 참조되며, O_NONBLOCK 또는 O_NDELAY, O_SYNC는 블록모드 처리에 관련되어 참조된다.

■ loff_t f_pos;

f_pos 필드 변수에는 현재의 읽기/쓰기 위치를 담는데, read(), write(), lseek()과 같이 읽기/쓰기의 위치를 변경할 수 있는 함수에 의해 변경된다.

WRITE() 구현 시 일반적인 구조

```
ssize_t xxx_write (struct file *filp, const char *buf, size_t count, loff_t *f_pos)
{
    if(!((데이터가 처리 가능한가?))){
        if(!((filp->f_flags & O_NONBLOCK) )){
            // 블록 모드로 열렸다면 프로세스를 재운다.
        }
    }
    // 사용자 공간에 데이터를 가져온다.
    // copy_from_user, get_user

    // 하드웨어에서 데이터를 쓴다.
    // outb(),..., writeb() 함수 사용
    // 또는 버퍼를 읽는다.

    return 처리된 데이터 개수;
}
```

실습5: WRITE() 추가 – SK.C

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/major.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");

int result;

/* Define Prototype of functions */
int sk_open(struct inode *inode, struct file *filp);
int sk_release(struct inode *inode, struct file *filp);
ssize_t sk_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos);

/* Implementation of functions */
int sk_open(struct inode *inode, struct file *filp)
{
    printk("Device has been opened...\n");

    /* H/W Initialization */

    return 0;
}
```

실습5: WRITE() 추가 – SK.C

```
int sk_release(struct inode *inode, struct file *filp)
{
    printk("Device has been closed...\n");

    return 0;
}

ssize_t sk_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos)
{
    char data[11];

    copy_from_user(data, buf, count);
    printk("data >>>> = %s\n", data);

    return count;
}

struct file_operations sk_fops = {
    .open    = sk_open,
    .release = sk_release,
    .write   = sk_write,
};
```

실습5: WRITE() 추가 – SK.C

```
static int __init sk_init(void)
{
    printk("SK Module is up... \n");
    result = register_chrdev(0, "SK", &sk_fops);
    if (result < 0 ) {
        printk("Couldn't get a major number...\n");
    }
    printk("major number=%d\n", result);
    return 0;
}

static void __exit sk_exit(void)
{
    printk("The module is down...\n");
    unregister_chrdev(result, "SK");
}

module_init(sk_init);
module_exit(sk_exit);
```


실습5: WRITE() 추가 – SK_APP.C

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

int main(void)
{
    int retn;
    int fd;
    /* write에서 사용할 버퍼 */
    char buf[100] = "write...\n";

    fd = open("/dev/SK", O_RDWR);
    printf("fd = %d\n", fd);

    if (fd < 0) {
        perror("/dev/SK error");
        exit(-1);
    }
    else
        printf("SK has been detected...\n");
    /* fd가 가르키는 파일에 buf에 있는 10바이트를 쓰라는 의미 */
    retn = write(fd, buf, 10);
    printf("\nSize of written data : %d\n", retn);

    close(fd);
    return 0;
}
```

실습5: WRITE() 추가 - 실행

■ 실행 절차

- 모듈 적재
- 장치파일 만들기
- 애플리케이션 실행

```
# ./sk_app
```

```
Device has been opened...
```

```
fd = 3
```

```
SK has been detected...
```

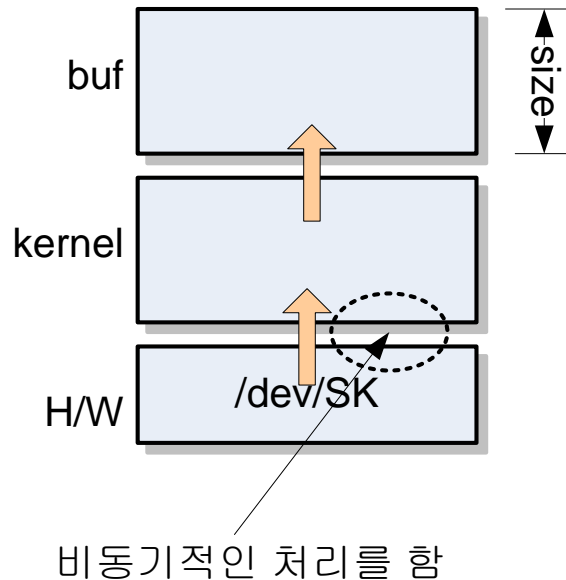
```
data >>>> = write...
```

```
Size of written data : 10
```

```
Device has been opened...
```

실습6: READ() 함수 추가

- read 함수 매개변수
 - char *buf → 응용프로그램에서 전달할 버퍼의 주소
 - size_t size: 응용 프로그램에서 요청한 데이터의 크기



```
n = read(fd, buf, size);
```

```
sk_read(file, buf, count, f_ops) {  
    // H/W access code here!  
    copy_to_user(buf, data_read, count);  
}
```

```
request_irq(service_number, irq_service)  
{  
    irq_service();  
}
```

```
irq_service()  
{  
    H/W access;  
    조건;  
    Wakeup interruptable;  
}
```

READ 함수의 구현

■ 개 요

디바이스 드라이버가 동작하는 메모리 공간은 커널 메모리 공간이고, 응용 프로그램이 동작하는 메모리 공간은 사용자 메모리 공간이기 때문에 다른 공간 사이에 메모리를 전송하기 위해 커널에서는 다음과 같은 함수를 사용해야 한다.

■ 주요 매서드

`ssize_t xxx_read(struct file *filp, char *buff,
size_t count, loff_t *offp)`

- 디바이스의 offp 위치에서 count 바이트 만큼을 읽어서 사용자 영역인 buff로 저장해주는 기능
- `unsigned long copy_to_user(void *to,
const void *from, unsigned long count);`
 - 커널 메모리 from을 사용자 메모리 to로 count만큼 복사한다.
- `put_user(x, ptr)`
 - x변수 값을 ptr의 사용자 메모리값에 대입한다.

READ 함수의 구현

■ 디바이스를 열 때 옵션 사항

- 이 값은 read() 함수의 매개변수 중에서 struct file *filp를 참조하여 판단한다.
- 응용 프로그램이 O_NONBLOCK이나 O_NDELAY를 지정하지 않은 상태로 디바이스 파일을 열었다면 count 값이 만족될 때까지 기다려야 한다.
- 그렇지 않으면 현재 발생된 데이터만 버퍼에 써넣고 함수를 종료해야 한다. 반환 값은 버퍼에 써넣은 데이터 개수다.

```
ssize_t xxx_read (struct file *filp, const char *buf, size_t count, loff_t *f_pos)
{
    ...
    if(filp->f_flags & O_NONBLOCK) {
        // 즉시 처리한다.
    } else {
        // 블록 처리한다.
    }
}
```

READ 함수의 구현

- read함수에서 하드웨어를 다루려면 다음 함수를 이용해야 한다.
 - port-map I/O
inb, inw, inl, insb, insw, insl,
 - memory-mapped I/O
readb, readw, readl,
 - I/O memory block
memset_io, memcpy_fromio, memcpy_toio

READ함수 구현의 일반적인 형태

```
ssize_t xxx_read(struct file *filp, const char *buf,  
                 size_t count, loff_t *f_pos)  
{  
    if(!(준비된 데이터가 있는가?))  
    {  
        if(!(filp->f_flags & O_NONBLOCK))  
        {  
            //블록 모드로 열렸다면 프로세스를 재운다.  
        }  
    }  
    // 하드웨어에서 데이터를 읽는다.  
    // inb(), ... readb(), ... 등 함수 사용  
    // 또는 버퍼를 읽는다.  
  
    // 사용자 공간에 데이터를 전달한다.  
    // copy_to_user, put_user  
  
    return 0;  
}
```

실습6: READ() 추가 – SK.C

```
...
ssize_t sk_read(struct file *filp, char *buf, size_t count, loff_t *f_pos);
...

ssize_t sk_read(struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
    /* App. 에 전달할 문자열을 담은공간 */
    char data[20] = "this is read func...";
    /* App. 으로 전달 받은 주소로부터 count까지의 내용을 buf로 옮긴다 */
    copy_to_user(buf, data, count);

    return count;
}
...
struct file_operations sk_fops = {
    .open    = sk_open,
    .release = sk_release,
    .write   = sk_write,
    .read    = sk_read,
};
```


실습6: READ() 추가 – SK_APP.C

```
...
int main(void)
{
    int retn;
    int fd;

    // char buf[100] = "write...\n";
    char buf[100] = {0};

    fd = open("/dev/SK", O_RDWR);
    printf("fd = %d\n", fd);

    if (fd < 0) {
        perror("/dev/SK error");
        exit(-1);
    }
    else
        printf("SK has been detected...\n");
    //retn = write(fd, buf, 10);
    retn = read(fd, buf, 20); // fd가 가르키는 파일에 buf에서 20byte 읽음
    printf("\ndata : %s\n", buf);

    close(fd);
    return 0;
}
```

실습6: READ() 추가 - 실행

■ 주요 코드 설명

- read 가 호출되면 sk_read가 호출된다. VFS에서 다음과 같이 호출된다.
f->f_op->read(); // 함수포인터 호출
- buf는 user 영역에 있는 buffer의 위치를 의미한다.

■ 실행 절차

- 모듈 적재
- 장치파일 만들기
- 애플리케이션 실행

```
[root@2440REBIS dd]$ ./sk_app
```

```
Device has been opened...
```

```
fd = 3
```

```
SK has been detected...
```

```
Read Data : this is read func...
```

```
Device has been opened...
```

실습7: IOCTL 함수 추가

■ 목적

- 일반적으로 I/O Control에 관련한 작업을 수행하는 함수 이다.
- 대부분의 ioctl 메소드 구현은 cmd 인수 값에 따라 올바른 동작을 선택하는 switch 문으로 구성한다.

■ 사용자 영역에서 ioctl 함수 시스템 콜

```
int ioctl(int fd, int cmd, ...);
```

■ 커널 영역에서의 ioctl 함수

- `int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);`
- inode와 filp 포인터는 응용 프로그램의 파일 디스크립터 fd와 일치하는 인수
- **cmd 인수**는 명령을 나타내는 응용 프로그램의 인수 전달
- arg 인수는 명령 실행의 결과 데이터가 전달되는 unsigned long 형의 정수 또는 포인터

CMD의 구성

- cmd 명령의 해석 매크로 함수

_IOC_NR 구분 번호 필드값을 읽는 매크로

_IOC_TYPE 맵핑 넘버 필드값을 읽는 매크로

_IOC_SIZE 데이터의 크기 필드값을 읽는 매크로

_IOC_DIR 읽기와 쓰기 속성 필드값을 읽는 매크로

Ex) if (_IOC_TYPE(cmd) != MY_MAGIC) return -EINVAL

- cmd 명령의 작성 매크로 함수

_IO 부가적인 데이터가 없는 명령을 만드는 매크로

_IOR 데이터를 읽어오기 위한 명령을 작성

_IOW 데이터를 써 넣기 위한 명령을 작성

_IOWR 디바이스 드라이버에서 읽고 쓰기위한 명령을 작성하는 매크로

Ex) _IOW(매핑넘버, 구분번호, 변수형)

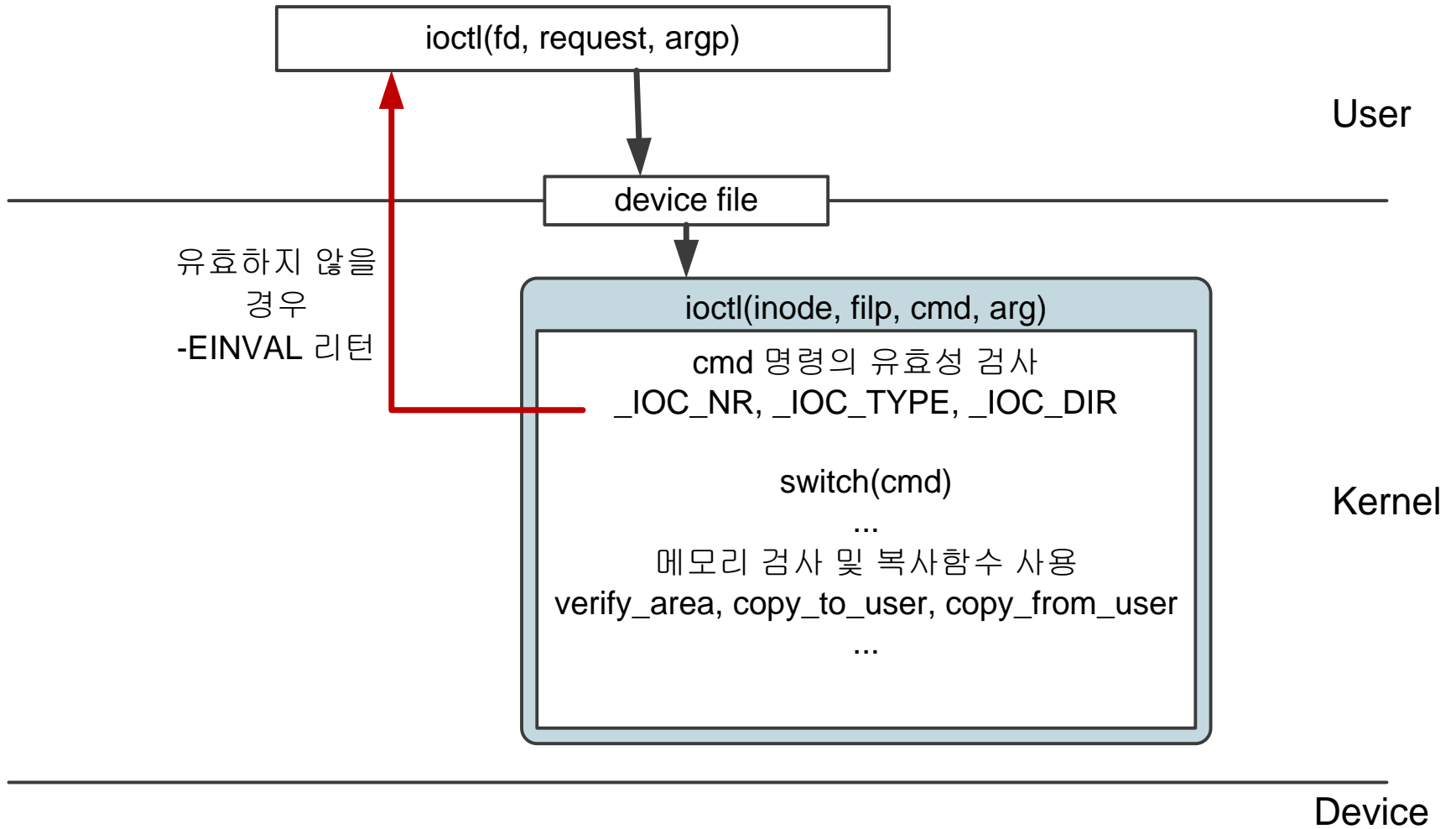


(읽기 쓰기 구분)

명령 CMD 상수 값

- ioctl 구현 시 해당 디바이스의 **명령들에 대한 고유의 상수 값**을 지정
 - 타입, 시퀀스 번호, 전송방향, 인수의 크기 등을 표시하는 비트필드로 표현
 - include/asm/ioctl.h와 Documentation/ioctl-number.txt를 체크해서 중복되지 않도록 드라이버의 ioctl 번호를 선정
- 타입(type)
 - 서로 중복되지 않는 8비트의 고유 번호(매직 넘버)
 - Documentation/ioctl-number.txt를 참조하여 선정 (_IOC_TYPEBITS)
- 시퀀스 번호(sequence number)
 - 8비트 크기의 시퀀스 번호 (_IOC_NRBITS)
- 방향(direction)
 - 명령의 데이터의 전송 방향을 표시
 - _IOC_NONE(데이터 전송이 없음), _IOC_READ, _IOC_WRITE, _IOC_READ | _IOC_WRITE (양방향 전송) 등의 값을 가짐

응용프로그램과의 관계



< **ioctl**의 동작 개념도 >

IOCTL 설계 시 일반적인 형식 (1)

```
1 #ifndef _SK_H_
2 #define _SK_H_
3
4 #define SK_MAGIC    'k'
5 #define SK_MAXNR    6
6
7 typedef struct {
8     unsigned long size;
9     unsigned char buff[128];
10 } __attribute__((packed)) sk_info;
11
12 #define SK_LED_OFF    _IO(SK_MAGIC, 0)
13 #define SK_LED_ON     _IO(SK_MAGIC, 1)
14 #define SK_GETSTATE   _IO(SK_MAGIC, 3)
15
16 #define SK_READ        _IOR(SK_MAGIC, 3, sk_info)
17 #define SK_WRITE       _IOW(SK_MAGIC, 4, sk_info)
18 #define SK_RW         _IOWR(SK_MAGIC, 5, sk_info)
19
20 #endif /* _SK_H_ */
```

<디바이스 드라이버 헤더 설계 시 예 **sk.h** >

IOCTL 설계 시 일반적인 형식 (2)

```
65 int sk_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
66 {
67     sk_info    ctrl_info;
...
71     /* Check on cmd */
72     if(_IOC_TYPE(cmd) != SK_MAGIC) return -EINVAL;
73     if(_IOC_NR(cmd) != SK_MAXNR) return -EINVAL;
74     size = _IOC_SIZE(cmd);
75     if (size) {
76         err = 0;
77         if(_IOC_DIR(cmd) & _IOC_READ)
78             err = verify_area(VERIFY_WRITE, (void *)arg, size);
79         else if(_IOC_DIR(cmd) & _IOC_WRITE)
80             err = verify_area(VERIFY_READ, (void *)arg, size);
81         if(err) return err;
82     }
...
85     switch(cmd) {
86         case SK_LED_OFF: {
87             ...
88         }
```

<디바이스 드라이버 구현 설계 시 예 **sk.c** >

실습7: IOCTL 함수 추가 – SK.C

```
...  
long sk_ioctl(struct file *filp, unsigned int cmd, unsigned long arg);  
...
```

```
long sk_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)  
{  
    switch(cmd) {  
        case SPEED_UP: {  
            printk("\n"); printk("UP\n"); break;  
        }  
        case SPEED_DOWN: {  
            printk("\n"); printk("DOWN\n"); break;  
        }  
        default:  
            return 0;  
    }  
    return 0;  
}
```

```
struct file_operations sk_fops = {  
    .open    = sk_open,  
    .release = sk_release,  
    .read    = sk_read,  
    .write   = sk_write,  
    .unlocked_ioctl = sk_ioctl,  
};
```

실습7: IOCTL 함수 추가 – SK_IOCTL.H

```
#ifndef SK_IOCTL_H
#define SK_IOCTL_H

#define IOCTL_MAGIC    'A'
#define SPEED_UP    _IO( IOCTL_MAGIC, 1 )
#define SPEED_DOWN    _IO( IOCTL_MAGIC, 2 )
#endif
```

실습7: IOCTL 함수 추가 – SK_APP.C

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <termio.h>
#include "sk_ioctl.h"

int main(void)
{
    int retn;
    int fd;

    int flag = 0;

    fd = open("/dev/SK", O_RDWR);
    printf("fd = %d\n", fd);

    if (fd < 0) {
        perror("/dev/SK error");
        exit(-1);
    }
    else
        printf("SK has been detected...\n");

    // getchar();
```

```
int choice;
printf("1. UP  2. DOWN ");
scanf("%d", &choice);
if(choice==1) {
    ioctl(fd, SPEED_UP, 0);
} else {
    ioctl(fd, SPEED_DOWN, 0);
}
close(fd);
return 0;
}
```

INTERRUPT 핸들러 등록

- request_irq()

gpio function in kernel

커널 소스/Documentation/driver-api/gpio/legacy.rst

gpio_to_irq() : GPIO_PIN에 IRQ mapping

SIGNAL TO USER PROCESS

- 커널에서 user 공간의 프로세스로 signal 전송

```
send_sig(SIGUSR1, my_task, 0);  
send_sig(SIGUSR2, my_task, 0);  
관련 링크
```

인터럽트 핸들러에 구현하면 user 프로세스에게 바로 인터럽트 이벤트를 신호로 알려줄 수 있음!!

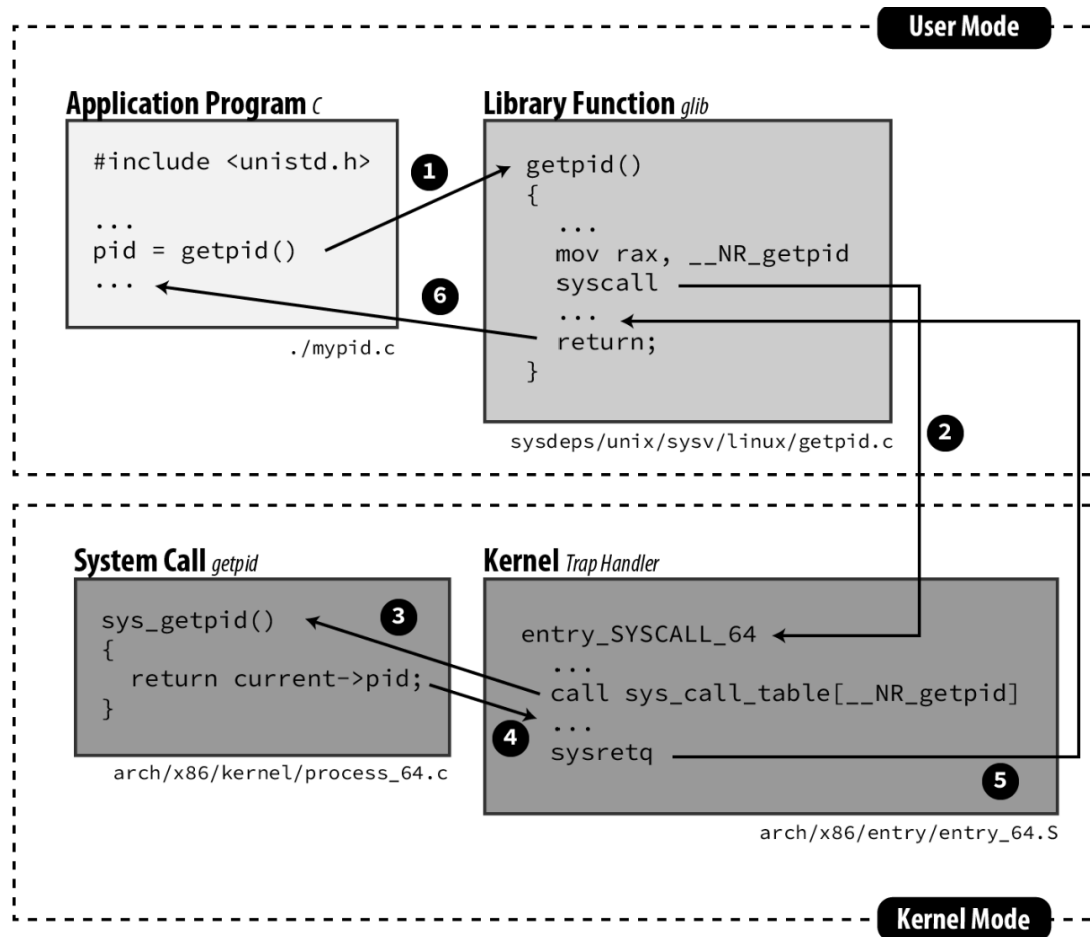
my_task 부분에는 signal을 전달할 task struct를 넣어주면되는데, pid를 알고 있다면

```
struct task_struct *my_task;  
my_task = pid_task(find_vpid(process_pid), PIDTYPE_PID);
```

by ioctl() 로 process_pid 인자로 전달

<https://qna.programmers.co.kr/questions/1786/%EB%A6%AC%EB%88%85%EC%8A%A4-%EC%BB%A4%EB%84%90-%EC%A7%88%EB%AC%B8%EC%9E%85%EB%8B%88%EB%8B%A4>

시스템 콜



시스템 콜

- 리눅스 시스템 콜

<https://www.chromium.org/chromium-os/developer-library/reference/linux-constants/syscalls/>

- ARM64 아키텍처의 리눅스 시스템콜

<https://www.linuxbnb.net/home/adding-a-system-call-to-linux-arm-architecture/>

<https://eastrivervillage.com/Anatomy-of-Linux-system-call-in-ARM64/>

ARM64 리눅스 시스템 콜 추가

- include/uapi/asm-generic/unistd.h

```
// add new system call
#define __NR_##### 451
__SYSCALL(__NR_##### , sys_#####) // 시스템콜 번호 정의

// add a 'plus one' at total number...
#define __NR_syscalls 452
```

- include/linux/syscalls.h

```
asmlinkage long sys_##### (.....); // 시스템 콜 함수 선언
// 매개변수 없는 경우 반드시 void 선언
```

- arch/arm64/kernel/sys.c (또는 별도의 소스 파일을 이용하여 구현)

```
SYSCALL_DEFINE#(#####) // 시스템 콜 구현
{
    . . . .
    return 0;
}
```


시스템 콜 실습

- GPIO 제어 시스템 콜: 라즈베리파이의 GPIO 핀을 제어하는 시스템 콜(출력)
- 시스템 콜 호출 방법(on 응용 프로그램)

```
long res = syscall(SYS_GPIO_CONTROL, pin, value);
```

MINI PROJECT

다음과 같이 센서들을 제어하는 프로그램을 작성해 봅시다.

- 조도 센서에 빛이 감지 되면, LED 켜기/알람 연주 – 디바이스 드라이버 이용
- 스위치를 누르면 LED 끄고, 알람도 끄기 – 디바이스 드라이버 이용
- LED ON/OFF(웹 기반) – 시스템 콜 기반