Module 10: 디바이스 드라이버 기초

ESP30076 임베디드 시스템 프로그래밍 (Embedded System Programming)

조 윤 석

전산전자공학부

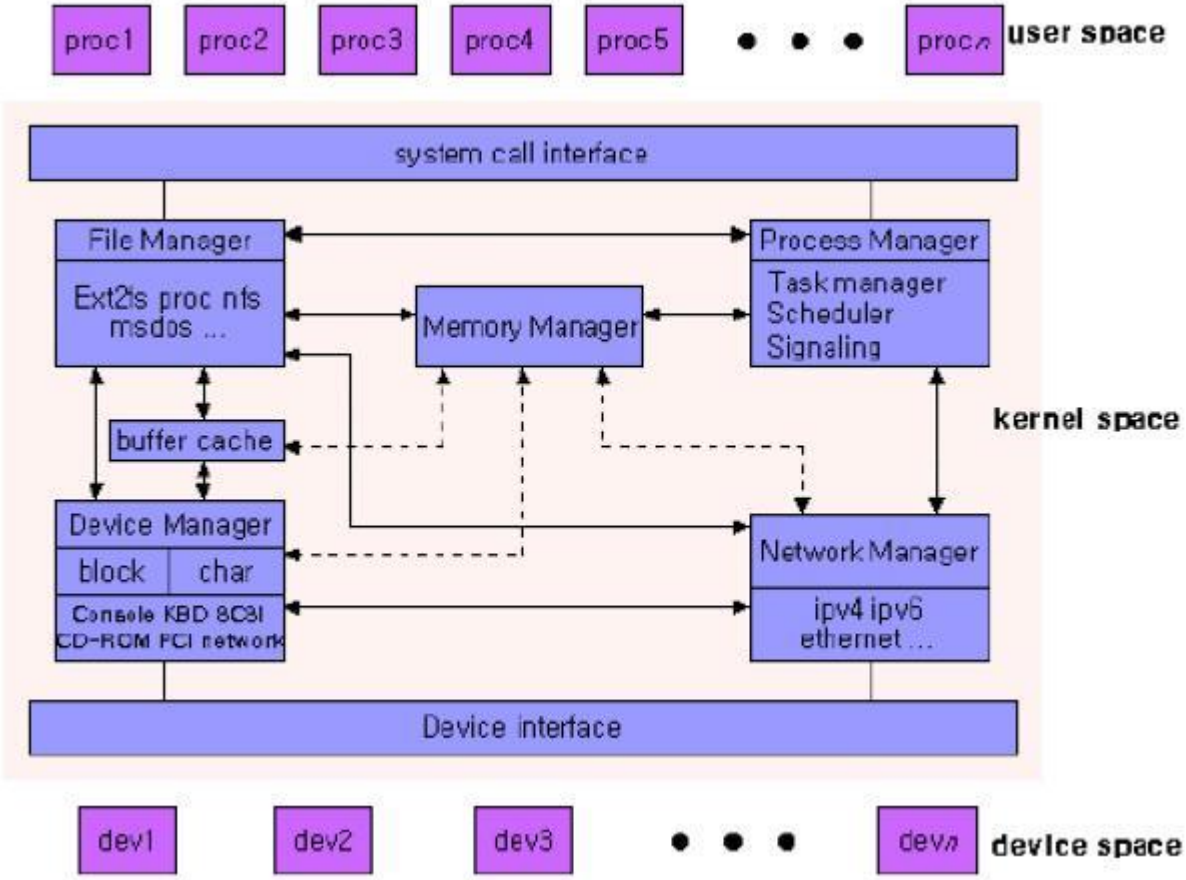
IGU 한동대학교 HANDONG GLOBAL UNIVERSITY

주차별 목표

- 디바이스 드라이버란?
- 디바이스 드라이버 주요 함수 알아보기
- 디바이스 드라이버 기본 골격 구성하기

디바이스 드라이버 프로그래밍

리눅스 커널의 구조



커널 프로그래밍

- 리눅스 커널의 핵심 (core) 기능 추가
- 리눅스 커널 알고리즘 개선
- 리눅스 커널 모듈 프로그래밍 (또는 디바이스 드라이버 프로그래밍)

디바이스 드라이버란?

- 디바이스 (device)
  - 하드 디스크, USB, 프린터, 단말기, 스캐너, 네트워크 어댑터, 터치 스크린, 오디오 등 컴퓨터 시스템 이외의 다른 주변장치들을 말함
- 디바이스 드라이버 (device driver, 이하 DD라고도 함)
  - 위의 디바이스들을 동작시키기 위해서는 구동용 소프트웨어가 필요하며, 이러한 프로그램을 디바이스 드라이버라고 함
  - 응용 프로그램에서 하드웨어 장치를 이용해서 데이터를 직접 읽고 쓰거나 제어해야 하는 경우에 device driver 이용
  - 하드웨어를 구동하기 위한 디바이스 뿐만 아니라, 소프트웨어적인 디바이스를 만들어 디바이스 드라이버를 만들 수도 있음

디바이스 드라이버의 주요 특징

- 주요 특징
  - 디바이스와 시스템 사이에 데이터를 주고 받기 위한 인터페이스를 제공하는 커널 내부 기능 중의 하나임
  - 일반적으로 위쪽으로는 파일 시스템과 인터페이스를 가지며, 아래쪽으로는 실제 디바이스 하드웨어와 인터페이스를 가짐
  - 커널의 일부분으로 내장되어 커널 모드에서 실행
  - 메모리에 상주하면서 스왑되지 않음
  - 디바이스 드라이버는 디바이스를 하나의 파일로 추상화시켜 줌
    - 이를 통해 사용자는 디바이스를 디바이스 파일(/dev/...)을 통해 파일처럼 액세스 가능
    - 따라서 사용자는 파일에 대한 연산 (File Operation)만 하면 됨
    - 디바이스의 고유한 특성을 내포하고 있음

디바이스 드라이버의 주요 특징

- 디바이스마다 고유의 번호를 가지고 있고, 이 번호로 각각의 디바이스를 구분
  - 이 번호는 32 비트로 구성되어 있음 (커널 2.6)
    - 주번호(Major number, 12-bit)
    - 부번호(Minor number, 20-bit)
- 디바이스 드라이버는 커널 함수로 모듈로 커널에 로딩
  - 디바이스 드라이버를 만들기 위해 전체 커널을 컴파일 할 필요는 없음
  - 모듈 방식으로 드라이버를 추가/제거할 수 있음

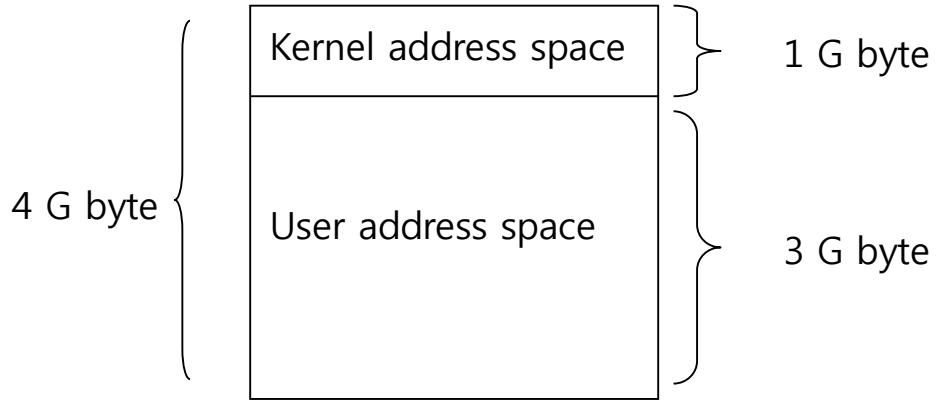
```
% insmod [드라이버명].ko
% rmmod [드라이버명]
```

커널 모듈

- 모듈 (Module)
  - 리눅스에서 디바이스 드라이버는 모듈로 커널에 loading됨
  - 여러 함수와 자료 구조로 이루어진 하나의 독립된 프로그램
  - 설치 과정을 통해 커널에 링크되어 커널에서 실행되는 함수 역할을 함
  - 리눅스 모듈은 처음 커널이 시작될 때 설치되는 정적 로딩 방법과 커널이 실행되는 중간에 설치되는 동적 로딩 방법에 의해 커널에 loading됨
- /proc/modules
  - 현재 시스템에 설치되어져 있는 모듈을 보여줌
- 커널 모듈 빌드 (커널 2.4)
  - gcc를 사용하여 커널 모듈을 생성함. 모듈은 컴파일된 오브젝트 코드(.o)임
- 커널 모듈 빌드 (커널 2.6)
  - 커널 모듈 생성시 kbuild를 사용함. 컴파일된 모듈의 확장자는 .ko임
- 커널 소스 헤더 파일 지정
  - -I/usr/src/linux-`uname –r`/include
  - 또는 makefile내에 다음과 같은 커널 디렉토리를 가르키는 변수명을 지정하여 사용하면 편리함
    - KERNELDIR=/lib/modules/$(shell uname –r)/build
    - 위의 파일은 커널 빌드 과정에서 % make module_install을 실행했을 때 생성되며, 커널 소스를 가르키도록 소프트 링크되어 있음
    - -I$(KERNELDIR)/include

커널과 응용 프로그램과의 차이

- **Address Space**
  - Application Program과 Kernel Program은 서로 다른 메모리 매핑법을 가지고 있으며, 프로그램 코드는 서로 다른 address space를 가지고 있다.



커널과 응용 프로그램과의 차이

- Namespace pollution
  - Application Program: 현재 개발하는 프로그램에서만 각 함수와 변수의 이름을 구별하여 주면 된다.
  - Kernel Program: 현재 개발하는 모듈 외에도 커널 전반적으로 함수와 변수의 이름이 충돌하지 않도록 하여야 한다.

커널 프로그래밍시 주의사항

- Namespace pollution
  - 외부 파일와 link하지 않을 모든 심볼을 static으로 선언 또는 외부 파일과 link할 symbol을 symbol table 등록
    - EXPORT_NO_SYMBOLS;
    - EXPORT_SYMBOL(name);
  - 전역 변수는 잘 정의된 prefix를 붙여 준다.
    - Ex: sys_open()
  - /proc/ksyms
    - Symbol table을 가지고 있는 텍스트 형태의 파일
- Library
  - stdio.h 와 같은 일반 프로그램에서 사용하는 헤더 파일을 include해서는 안된다.
  - 오직 /usr/include/linux 와 /usr/include/asm 아래에 선언된 헤더파일 만을 include 한다.

커널 프로그래밍시 주의사항

- Fault handling
  - Kernel 은 하드웨어 접근에 대해 어떠한 제한도 없기 때문에 커널에서의 에러는 시스템에 치명적인 결과를 발생시킨다.
  - 함수 호출 등의 작업시 모든 에러 코드를 검사하고 처리해야 한다.

- Address space
  - 커널이 사용하는 stack의 크기는 제한되어 있고, 인터럽트 핸들러도 동일한 스택을 사용하므로 큰 배열을 사용하거나, recursion이 많이 일어나지 않도록 주의해야 한다.
  - 응용 프로그램과 데이터를 주고 받기 위해(call by reference) 특별한 함수를 사용 하여야 한다. (뒷 부분에서 이를 위한 몇 가지 함수를 소개한다.)

- 기타
  - 실수연산 이나 MMX 연산을 사용할 수 없다.

디바이스 구분

- 문자 디바이스 (Character device)
  - 자료의 순차성을 지닌 장치로 버퍼를 사용하지 않고 바로 읽고 쓸 수 있는 장치
  - 직렬 포트, 병렬 포트, 마우스, PC 스피커, 터미널 등
- 블록 디바이스 (Block device)
  - 버퍼 캐시(cache)를 통해 블록 단위로 입출력되며, 랜덤 액세스가 가능하고, 파일 시스템을 구축할 수 있음
  - 플로피 디스크, 하드 디스크, CD-ROM, RAM 디스크 등
- 네트워크 디바이스 (Network device)
  - 네트워크 통신을 통해 네트워크 패킷을 주고 받을 수 있는 디바이스
  - Ethernet, PPP, ATM, ISDN, NIC (Network Interface Card) 등

디바이스 드라이버 작성하려면

- 하드웨어에 대한 분명한 이해가 있어야 함
- 소프트웨어 구조에 대한 이해
- 예를 들어 직렬 디바이스 (UART)에 대한 device driver를 작성한다면 다음의 사항들을 분명히 알아야 함 (일부 나열)
  - UART는 세 종류의 레지스터를 가지고 있음
    - Data registers, control registers, status registers
  - 하나의 디바이스 주소에 하나 이상의 디바이스 레지스터들이 있을 수 있음
  - 디바이스는 control register의 bit들을 설정함으로써 초기화 하거나, 설정을 변경할 수 있음
  - 디바이스는 control register의 bit들을 리셋 함으로써 close하거나 리셋을 할 수 있음

디바이스 드라이버 작성하려면

- Control register 비트들은 UART의 모든 동작을 제어할 수 있음
  - 따라서 control register의 각 비트들의 목적 정확히 알아야 함
- Status register 비트들은 디바이스의 현재 상태 (status)에 대한 정보를 가지고 있고, action이 일어날때 마다 해당 플래그의 값들이 변경됨
  - (예)TRH 버퍼 레지스터의 내용이 모든 전송된 후 새로 전송할 비트가 생긴다고 할 경우, 두 상태 사이에 transmitter empty flag의 설정이 변함
  - Status register에서 각 status flag의 목적이 무엇인지 정확히 알아야 함
- 각 레지스터에 대한 주소를 알아야 함
  - 예를 들어 IBM PC의 경우
    - Timer : 0x0040 ~ 0x005F
    - Serial COM1: 0x03F8 ~ 0x03FF
    - Serial COM2: 0x02F8 ~ 0x02FF

디바이스 드라이버 관련 리눅스 명령어

명령어	기능
insmod	커널에 모듈을 올림 (예) % insmod test_dd.ko
rmmod	커널에서 모듈을 삭제함 (예) % rmmod test_dd
lsmod	커널에 적재된 모듈 목록 보여줌
mknod	장치 특수 파일을 만듦 (예) % mknod /dev/test_dd c 253 0
depmod	커널 모듈 간의 의존성을 검사함
modinfo	모듈들을 검사하여 관련된 정보 보여줌
modprobe	depmod 명령으로 생성된 종속성 정보를 이용하여 지정된 디렉토리의 모듈들과 연관된 모듈들을 자동으로 로딩함
ksyms	Export되어 있는 커널의 심볼 목록(/proc/ksyms)을 보여줌
nm	오브젝트 파일에 있는 심볼들의 목록을 보여줌

간단한 모듈 프로그래밍

- 디바이스 드라이버 프로그래밍이 무엇인지를 알아보기 위해 가장 간단한 커널 프로그래밍 해 보기
  - 커널에 모듈이 로딩될 때 “Loading my first device driver…”라는 문자열 출력되게 하기
  - 커널에서 모듈을 삭제할 때 “Unloading my first device driver…” 라는 문자열 출력하기

hello.c

% cd ~/work/dd/hello

% vi hello.c

```
/* filename: hello.c */
#include <linux/module.h>

MODULE_LICENSE("GPL");

int init_module(void) /* module_init() */
{
    printk("Loading my first device driver...₩n");
    return 0; ← 모듈 초기화 성공을 의미함
}

void cleanup_module(void) /* module_exit() */
{
    printk("Unloading my first device driver..₩n");
}
```

컴파일

% vi Makefile

```
obj-m = hello.o
#KDIR = /usr/src/linux-headers-2.6.32-38-generic
#KDIR = /usr/src/linux-headers-`uname -r`
KDIR = /lib/modules/$(shell uname –r)/build
PWD = $(shell pwd)
all: module
module:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
    rm -rf *.ko
    rm -rf *.o
    rm -rf *.symvers *.order *.mod.o
```

% make

실행

- 커널에 로드하기

  % insmod hello.ko

- 커널에서 모듈 제거하기

  % rmmod hello

  - (주의) 모듈을 제거할 때는 hello.o가 아닌 hello를 사용해야 함

- 모듈 load/unload시 메시지가 출력되었는가?

Warning: loading driver will taint the kernel: no license 문제 해결

- 리눅스 커널 2.4 이상 버전부터 라이센스 정의를 위한 방법을 고안하였고, 이러한 내용은 linux/module.h 파일에 정의되어 있고, MODULE_LICENSE() 매크로를 통해 수행됨
- 이러한 문제를 해결하는 간단한 방법은 프로그램의 앞부분에 다음 한 줄을 추가하기
  - MODULE_LICENSE("GPL");

커널에서 출력하는 메시지 보기

- 커널에서 출력하는 메시지는 console 화면(standard out)에 나타나지 않는다
- /var/log/messages
  - 커널에서 출력하는 메시지를 담고 있는 파일

    % tail /var/log/messages
    - 파일의 맨 뒷부분의 내용을 출력해 줌. Foreground로 동작

    % tail –f /var/log/messages
      - 계속 추가되는 모든 메시지 보기
      - 중단하려면 ctrl+C를 누른다

    % tail –f /var/log/messages &
    - Background mode로 동작. Messages 파일이 변경될때마다 그 내용을 화면에 출력해 줌. 백그라운드 수행을 종료하고 싶을 경우 jobs, kill이라는 명령어를 이용하여 프로그램 종료시킴
- dmesg

C 프로그램과 커널 프로그램의 차이

[hello.c]

```
#include   <linux/module.h>
#include   <linux/kernel.h>
MODULE_LICENSE("GPL");
int   init_module(void)
{
        printk("Loading my first device driver...₩n");
        return 0;
}
void   cleanup_module(void)
{
        printk("Unloading my first device driver!!!₩n");
}
```

C 프로그램에서의 실행 시작점은 main() 함수가 없음.
그러나 커널 프로그램에는 main() 함수가 없음.
그렇다면 이 프로그램은 어떻게 실행되는 것일까?

디바이스 드라이버 구성 요소

- 초기화 인터페이스
  - 초기화 함수 <linux/init.h>
    - int init_module(void), void cleanup_module(void)
    - 또는 module_init(), module_exit()
  - 디바이스 등록 및 해제 함수
    - 등록: register_mmmdev(), 해제: unregister_mmmdev()
    - 여기에서 mmm은 chr, blk, net
- 파일 시스템 인터페이스
  - 리눅스에서는 모든 장치(device)들을 파일로 간주하며, 이 파일들은 일반적으로 /dev 디렉토리 밑에 있음. 따라서 어떤 장치를 open한다고 하는 것은 파일 구조를 다루는 것과 동일
  - 커널에서는 디바이스 드라이버에 액세스 하기 위해 file_operations 라는 구조체를 사용함
  - 파일 구조체 "file_operations"에 대한 선언은 <linux/fs.h>에 되어 있으며, 이 구조체의 주요 멤버로는 open(), release(), read(), write(), ioctl(), mmap() 등임
- 하드웨어 인터페이스
  - Memory mapped I/O: 메모리 연산
  - Special in/out instruction: in(), out()

디바이스 등록 및 해제 함수

함수명	설명
int register_chrdev (unsigned int major, const char *name,struct file_operations *fops)	문자(character) 디바이스를 주어진 주번호로 등록
int unregister_chrdev (unsigned int major, const char *name)	주어진 주번호에 등록되어 있는 문자 디바이스 등록을 해제
int register_blkdev (unsigned int major, const char *name,struct file_operations *fops)	블록(block) 디바이스를 주어진 주번호로 등록
int unregister_blkdev (unsigned int major, const char *name)	주어진 주번호에 등록되어 있는 블록 디바이스 등록을 해제
int register_netdev(const char *name)	네트워크 디바이스를 등록
int unregister_netdev(const char *name)	네트워크 디바이스 등록을 해제
MAJOR(kdev_t dev)	장치 번호 dev로부터 주번호 구하기
MINOR(kdev_t dev)	장치 번호 dev로부터 부번호 구하기

file 구조체 (/usr/src/linux/include/linux/fs.h)

```
struct file {
        struct list_head        f_list;
        struct dentry           *f_dentry;
        struct vfsmount         *f_vfsmnt;
        struct file_operations  *f_op;
        atomic_t                f_count;
        unsigned int            f_flags;
        mode_t                  f_mode;
        loff_t                  f_pos;
        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 driver, and maybe others */
        void                    *private_data;

        /* preallocated helper kiobuf to speedup O_DIRECT */
        struct kiobuf           *f_iobuf;
        long                    f_iobuf_lock;
};
```

file_operations 구조체

```
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);
};
```

응용프로그램에서의 시스템콜 함수와 디바이스 드라이버 함수와의 관계

```
struct file_operations test_fops = {
        open:           device_open,
        read:           device_read,
        write:          device_write,
        ioctl:          device_ioctl,
        release:        device_release,
};
```

```
struct file_operations test_fops = {
        .open = device_open,
        .read = device_read,
        .write = device_write,
        .ioctl = device_ioctl,
        .release = device_release,
};
```

ANSI C99에서 표준으로 제공

응용 프로그램에서의 시스템 콜	디바이스 드라이버에서의 실행 함수
open()	device_open()
close()	device_release()
read()	device_read()
write()	device_write()
ioctl()	device_ioctl()

Device Driver에서 사용하는 함수들

- open()
  - 해당 디바이스에 연산을 가하기 위해 해당 디바이스 파일을 열기 위한 함수. 사용수 증가
- read()
  - 해당 디바이스로부터 데이터를 얻은 데이터를 커널 영역에서 사용자 영역으로 복사하기 위한 함수
- write()
  - 사용자 영역의 데이터를 커널 영역으로 복사하기 위한 함수
- release()
  - 해당 드라이버가 응용프로그램에 의해 닫힐 때 호출 하는 함수. 사용수 감소
- ioctl()
  - 읽기 / 쓰기 이외의 부가적인 연산을 위한 인터페이스 - 디바이스 설정 및 하드웨어 제어(향상된 문자드라이버 작성가능)

open()

- 응용 프로그램에서의 함수 선언
  - int open(const char *pathname, int flags);
    int open(const char *pathname, int flags, mode_t mode);
- 디바이스 드라이버에서의 함수 선언
  - int (*open) (struct inode *, struct file *);

  - 응용 프로그램에서 open() 함수는 시스템 콜을 통해 sys_open()을 호출하며, sys_open() 내부에서는 가상 파일 시스템 (VFS)과 관련된 여러 처리 과정을 통해 디바이스 드라이버 함수에 필요한 inode와 파일 구조체 정보로 변환하여 전달된다. 그 다음 단계는 file_operations 구조체의 open() 함수 포인터에 등록된 함수를 사용하여 요청한 파일 시스템에 맞는 연산을 수행한다.

- 응용 프로그램에서의 함수 인자
  - pathname은 생성하고자 하는 파일 이름
  - flags는 파일을 어떤한 모드로 open할 것인지를 결정하기 위해 사용함. "읽기 전용"(O_RDONLY), "쓰기 전용"(O_WRONLY), "읽기/쓰기"(O_RDWR) 모드로 열 수 있음
  - 성공하면 해당 파일을 지시하는 int 형의 파일 지시자 (file descriptor)를 되돌려 줌

read()

- 응용 프로그램에서의 함수 선언
  - ssize_t read(int fd, void *buf, size_t count);
- 디바이스 드라이버에서의 함수 선언
  - ssize_t (*read) (struct file *, char *, size_t, loff_t *);
  - ssize_t (*read) (struct file *filp, char *buf, size_t count, loff_t *f_pos);

- 함수 인자
  - 응용 프로그램에서의 read() 함수는 파일 기술자인 fd로부터 count 바이트 만큼 읽어서 그값을 buf에 저장하는 시스템 콜이다. 성공하면 읽어들인 바이트 크기를 반환하며, 실패하면 -1을 반환하고 errno를 설정한다.
  - 디바이스 드라이버에서의 함수 선언 중 파일 구조체 포인터 filp는 디바이스 파일이 어떤 형식으로 열렸는가에 대한 정보를 가지고 있다. loff_t 타입은 64 비트 길이의 "long offset"으로, (32-bit 플랫폼의 경우에도) 메모리 접근 주소를 지정하며 현재의 읽기와 쓰기 위치를 저장한다. 문자 디바이스 드라이버에서는 특성상 큰 의미가 없다.

write()

- 응용 프로그램에서의 함수 선언
  - ssize_t write (int fd, const char *buf, size_t count);
- 디바이스 드라이버에서의 함수 선언
  - ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
  - ssize_t (*write) (struct file *filp, const char *buf, size_t count, loff_t *f_pos);

  - 응용 프로그램에서 write() 함수는 buf로부터 count 바이트만큼 읽어서 fd가 가리키는 파일에 쓰는 시스템 콜이다. 성공하면 쓴 바이트 크기를 반환하고, 실패하면 -1을 반환한다. f_ops는 메모리 접근 주소를 가리킨다.

- 함수 인자
  - char *buf : 응용프로그램에서 전달한 버퍼의 주소
  - size_t count : 응용프로그램에서 요청한 데이터의 크기
  - struct file *filp : 디바이스 파일이 어떤형식으로 열렸는가에 대한 정보를 저장
  - loff_t *f_pos : 메모리 접근 주소

ioctl()

- 응용 프로그램에서의 함수 선언
  - int ioctl (int fd, int request, ...);
- 디바이스 드라이버에서의 함수 선언
  - int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

  - ioctl() 시스템 콜은 디바이스를 제어하기 위해 특별한 명령을 주고자 할 때 사용한다. 예를 들어 디스크를 포맷 (읽기/쓰기 동작 아님)하거나 직렬 포트의 통신 속도 등을 설정하는 것이다. 이외에도ioctl()은 시스템 콜을 추가하지 않고 커널 영역과 상호 작용하는 다양한 함수들을 개발하는데 사용하기도 한다.
  - 응용 프로그램에서의 파일 기술자 fd는 해당 파일에 대한 inode와 file 구조체로 변환되어 디바이스 드라이버 함수로 전달된다. 응용 프로그램 함수 선언 부분에서 세번째 인자 부분은 "..."으로 표시되어 있는데 이는 인자의 개수가 여러 개일 수 있다는 것을 의미한다.시스템 콜이 성공적으로 실행되면 음수 아닌 값을 반환하며, 반환된 값은 요청했을 때 사용한 값과 동일하다.

release()

- 응용 프로그램에서의 함수 선언
  - int close(int fd);
- 디바이스 드라이버에서의 함수 선언
  - int (*release) (struct inode *, struct file *);

  - 시스템 콜 close() 함수는 파일 기술자를 닫을 때 사용한다. 응용 프로그램에서 사용자에 의해 close() 함수가 호출되면 디바이스 드라이버 내에서는 해당 파일 구조체 내에 선언된 release() 함수가 실행된다.

- 참고:
  - Note that release isn't invoked every time a process calls close. Like open, release can be missing.
  - Whenever a file structure is shared (for example, after a fork or a dup), release won't be invoked until all copies are closed.
  - If you need to flush pending data when any copy is closed, you should implement the flush method.

커널 인터페이스 함수

- 주의 사항
  - Kernel program은 일반적인 library를 사용하지 못하고 kernel에서 export 해 준 함수들 만을 사용할 수 있다.

- Kernel interface 함수 분류
  - Kernel 에서 제공하는 함수 중 kernel programming에 자주 사용되는 함수는 다음과 같이 분류할 수 있다.
    - Port I/O
    - Interrupt
    - Memory
    - Synchronization
    - Kernel message 출력
    - Device Driver register

커널 인터페이스 함수: 입출력 디바이스

- I/O device 와 data를 주고 받기 위한 함수들
  - unsigned inb(unsigned port)
    - Port에서 1byte를 읽는다.
  - unsigned inw(unsigned port)
    - Port에서 2byte를 읽는다.
  - unsigned inl(unsigned port)
    - Port에서 4byte를 읽는다.

  - unsigned outb(char value, unsigned port)
    - Port에 1byte value를 쓴다.
  - unsigned outw(short int value, unsigned port)
    - Port에 2byte value를 쓴다.
  - unsigned outl(long int value, unsigned port)
    - Port에 4byte value를 쓴다

커널 인터페이스 함수: 입출력 디바이스

- I/O device 와 data를 주고 받기 위한 함수들
  - void insb(unsigned port, void *addr, unsigned long count)
    - Port에서 count bytes를 읽어서 메모리의 addr 주소부터 저장
  - void insw(unsigned port, void *addr, unsigned long count)
    - Port에서 16bit * count 만큼 읽어서 메모리의 addr 주소부터 저장
  - void insl(unsigned port, void *addr, unsigned long count)
    - Port에서 32bit * count 만큼 읽어서 메모리의 addr 주소부터 저장

  - void outsb(unsigned port, void *addr, unsigned long count)
    - Memory의 addr번지 에서부터 count bytes를 읽어서 port에 쓴다.
  - void outsw(unsigned port, void *addr, unsigned long count)
    - Memory의 addr번지 에서부터 count * 16bit를 읽어서 port에 쓴다.
  - void outsl(unsigned port, void *addr, unsigned long count)
    - Memory의 addr번지 에서부터 count * 32bit를 읽어서 port에 쓴다.

커널 인터페이스 함수: 입출력 디바이스

- I/O device 와 data를 주고 받기 위한 함수들
  - Pausing I/O
    - 입출력이 너무 빠르면 device에서 처리할 수 없는 경우가 발생할 수 있기 때문에 한번의 입출력 후 잠시 멈추어 줄 수 있다.
    - 앞에 설명한 함수의 이름 뒤에 '_p' 를 붙인 이름의 함수로 구현되어 있다.
      - 예) inb() 함수의 경우 inb_p()

커널 인터페이스 함수: 인터럽트 설정

- 인터럽트의 설정 및 처리에 관한 함수(or 매크로)
  - cli()/sti()
    - 전체 인터럽트를 금지하거나 가능하게 해 주는 매크로 (clear/set interrupt enable)
  - save_flags(unsigned long flag), restore_flags(unsigned long flag)
    - 상태 레지스터 (status register)의 내용을 저장하고 복원
      - Status register는 시스템의 각 상태를 가지고 있는 레지스터로서, 일반적으로 인터럽트가 enable 상태인지 또는 직전 연산에서 올림수(carry)가 발생했는지 등에 대한 정보를 가지고 있다
    - save_flags(), restore_flags() 두 매크로는 같은 함수 안에서 호출되어야 한다. flag를 다른 함수로 pass해서는 안된다.

커널 인터페이스 함수: 인터럽트 설정

- 인터럽트의 설정 및 처리에 관한 함수(or 매크로)
  - int request_irq(unsigned int irq, void (*handler)(int),unsigned long flags, const char *device)
    - 사용자가 인터럽트를 추가하고자 할 때 사용
    - 커널에 현재 사용하고 있지 않은 IRQ를 요청하여 등록하고, 이러한 인터럽트가 발생했을 때 수행될 이 IRQ에 대한 interrupt handler 함수 (또는 interrupt service routine)를 등록
  - void free_irq(unsigned int irq)
    - request_irq()에서 획득한 irq를 반납함

커널 인터페이스 함수: 인터럽트 설정

- **동기화**
  - void sleep_on(struct wait_queue **q)
    - q의 번지를 event로 sleep하며, uninterruptible
    - 여기에서 uninterruptible은 wake_up 함수 호출외의 다른 signal에 의해 process가 깨어날 수 없음을 의미
  - void sleep_in_interruptible(struct wait_queue **q)
    - q의 번지를 event로 sleep하며, interruptible
    - Interruptible 은 wake_up 함수가 호출되지 않아도 임의로 process에서 signal을 주어 깨어나게 할 수 있다는 의미
  - void wake_up(struct wait_queue **q)
    - sleep_on(q)로 sleep된 task를 wakeup
  - void wake_up_interruptible(struct wait_queue **q)
    - sleep_on_interruptible(q)로 sleep된 task를 wakeup

커널 인터페이스 함수: 메모리 할당

- Kernel에서 동적 메모리를 할당할 때 사용하는 함수들
  - void * kmalloc(unsigned int len, int priority)
    - 커널 메모리 할당. 128~131056byte까지 가능
    - priority: GFP_KERNEL, GFP_BUFFER, GFP_ATOMIC, GFP_USER
      - GFP_KERNEL : 일반적인 커널 메모리 할당. 할당 가능한 Memory가 부족할 경우 sleep할 수도 있다.
      - GFP_BUFFER : 버퍼 캐쉬를 관리할때 사용되므로 할당자가 sleep상태로 갈수 있다. I/O 서브 시스템이 스스로 메모리를 필요로 할때 데드락을 피하도록 하기 위해 디스크에 dirty page를 disk에 플러쉬함으로서 메모리를 free한다는 점에서 GFP_KERNEL과 다르다.
      - GFP_ATOMIC : 인터럽트 핸들러 등 프로세스 컨텍스트 외부 코드에서 메모리를 할당할때 사용한다. 결코 sleep상태가 되지 않는다.
      - GFP_USER : 사용자들에게 메모리 할당할때 사용. 낮은 우선순위를 가진다. GFP_HIGHUSER High memory에서 할당할때 사용한다.
    - 물리적으로 연속적인 메모리를 할당한다.
  - void kfree(void *obj)
    - kmalloc()에서 할당 받은 커널 메모리를 반납

커널 인터페이스 함수: 메모리 할당

- Kernel에서 동적 메모리를 할당할 때 사용하는 함수들
  - void * vmalloc(unsigned int len)
    - 커널 메모리 할당
    - 크기 제한 없음
    - 가상 주소 공간에서 연속적인 메모리 영역을 할당
  - void vmfree(void *addr)
    - vmalloc()에서 할당 받은 커널 메모리를 반납

커널 인터페이스 함수: 데이터 공유

- 사용자 공간과 커널공간 사이에 데이터를 공유하기 위한 함수
  - unsigned long copy_from_user(void *to, const void *from, unsigned long n)
    - 사용자 주소공간에서 n byte만큼 data 복사.
  - unsigned long copy_to_user(void *to, const void *from, unsigned long n)
    - 사용자 주소 공간에 n byte만큼 data 복사
  - void * memset(void *s, char c, size_t count)
    - 메모리 s에 c를 count만큼 복사
  - put_user(void *datum, const void *addr)
    - 사용자 공간에 datum을 전달
  - get_user(void *datum, const void *addr)
    - 사용자 공간의 datum을 커널 영역으로 전달

커널 인터페이스 함수: 메시지 출력

- Standard out 으로 메시지를 출력하기 위한 함수
  - printk(const char *fmt,… .)
    - printf의 커널 버전
    - printk(LOG_LEVEL message)
      - LOG_LEVEL
        - » KERN_EMERG, KERN_ALERT, KERN_ERR,KERN_WARNING, KERN_INFO, KERN_DEBUG
        - » console_loglevel 의 값 보다 우선 순위가 낮다면 console에 출력되지 않는다. console_loglevel은 sys_syslog 시스템 콜로 값을 바꿀 수 있다.
        - » LOG_LEVEL 은 <include/linux/kernel.h> 헤더에 정의 되어 있다.
  - 예
    - printk("<1>Hello, World");
    - printk(KERN_WARNING"warning… ₩n");

커널 인터페이스 함수: 디바이스 등록/해제

- 디바이스 드라이버 등록 및 해제 함수
  - int register_xxxdev (unsigned int major, const char *name,struct file_operations *fops)
    - character/block driver를 xxxdev[major]에 등록
    - xxx: chr/blk
  - int unregister_xxxdev (unsigned int major, const char *name)
    - xxxdevs[major]에 등록되 있는 device driver를 제거
  - int register_netdev(const char *name)
  - int unregister_netdev(const char *name)
  - MAJOR(kdev_t dev)/MINOR(kdev_t dev)
    - 장치번호 dev로부터 major/minor 번호를 구함

Device Driver에서 주로 사용하는 헤더 파일

#include <linux/kernel.h>

- printk()에서 사용되는 LOG_LEVEL에 대한 정의(예: KERNEL_ALERT)

#include <linux/module.h>

- MODULE_LICENSE("GPL");

#include <linux/init.h>

- module_init(), module_exit(), __init, __exit

#include <linux/fs.h>

- struct file_operations

#include <linux/fcntl.h>

- 파일 제어 관련 (예: O_RDWR, O_RDONLY, O_WRONLY)

#include <linux/errno.h>

- 에러에 관한 헤더 파일 (에러 번호 관리)

커널에 모듈 올리기/내리기

- 커널에 제작한 디바이스 드라이버 올리기

  % insmod 드라이버명.ko

  예) insmod led_dd.ko → init_module() 또는 module_init() 함수 실행

- 디바이스 드라이버 삭제

  % rmmod 드라이버명

  예)rmmod led_dd → cleanup_module() 또는 module_exit() 함수 실행

- 적재된 디바이스 드라이버 목록 보기

  % lsmod

디바이스 접근용 노드 파일 생성

- 커널에 적재된 디바이스 드라이버에 사용자가 접근할 수 있도록 스페셜 디바이스 노드 생성
  - 일반적으로 /dev 밑에 만듬
  - 사용자는 이 파일을 통해 해당 디바이스에 액세스 함

- 노드 생성하기
  - % mknod /dev/파일이름 드라이버특성 주번호 부번호
    - 예) mknod /dev/iom_led c 245 0
  - 생성 후 속성변경 : chmod ug+w /dev/iom_led

디바이스 드라이버에 연결된 장치 파일

```
% ls -l /dev/hd[ab][12] /dev/tty[01] /dev/fb[01]
brw-rw----  1 root  disk   3,  1  1월 30  2003 /dev/hda1
brw-rw----  1 root  disk   3,  2  1월 30  2003 /dev/hda2
brw-rw----  1 root  disk   3, 65  1월 30  2003 /dev/hdb1
brw-rw----  1 root  disk   3, 66  1월 30  2003 /dev/hdb2
crw--w----  1 root  root   4,  0  1월 30  2003 /dev/tty0
crw-------  1 root  root   4,  1  9월 25 13:44 /dev/tty1
crw-------  1 root  root  29,  0  1월 30  2003 /dev/fb0
crw-------  1 root  root  29,  1  1월 30  2003 /dev/fb1
```

디바이스 종류 | 주번호 (major number) | 부번호 (minor number) | 장치파일명

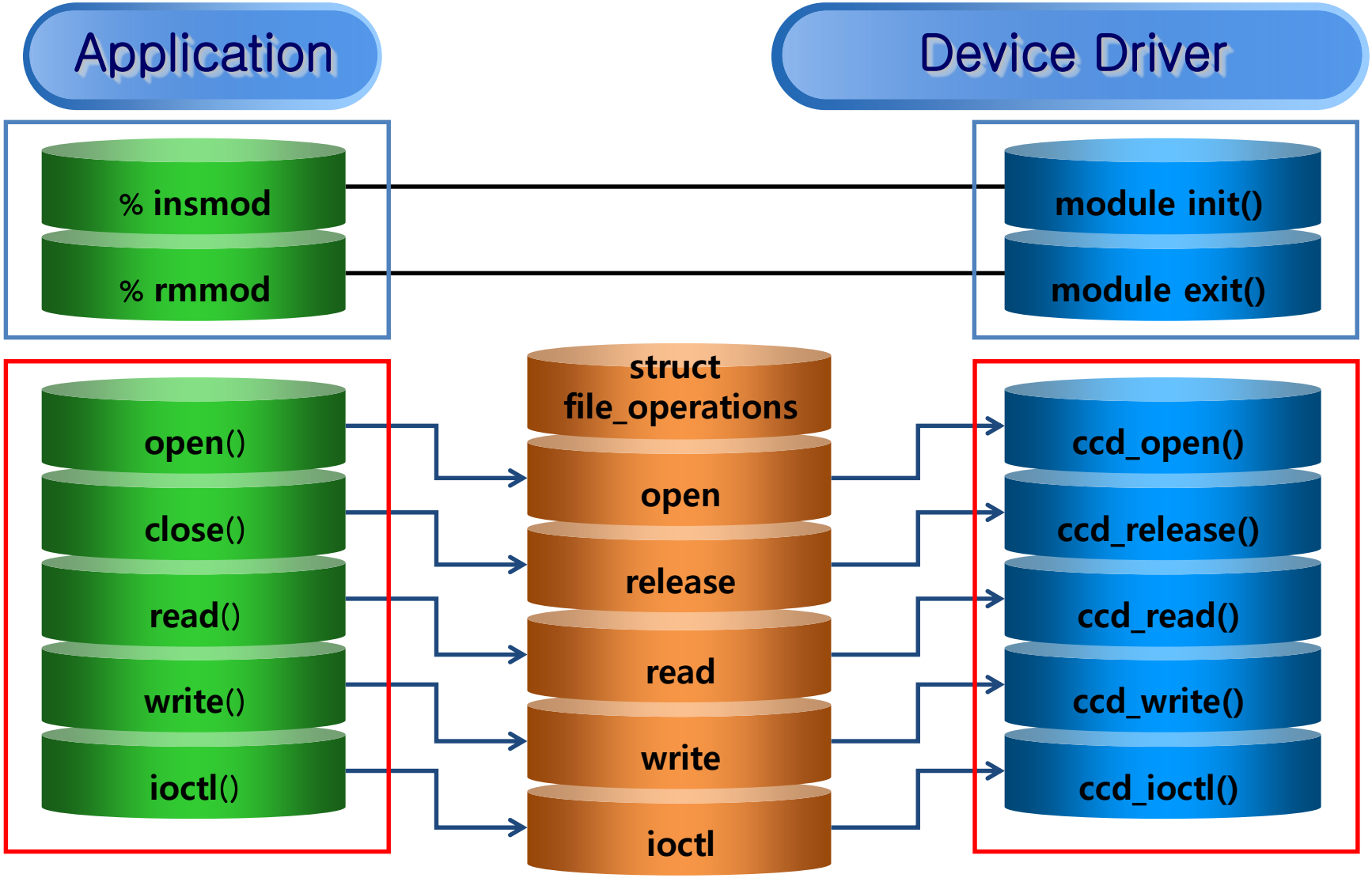
디바이스 번호

- 각 디바이스들은 주번호 (major number)와 부번호 (minor number)로 구성된 고유의 32-bit 번호를 가짐
- Major number (12-bit)
  - 물리적인 디바이스를 구분하는데 사용
  - [KERNEL_DIR]/include/linux/major.h 헤더 파일에 정의되어 있음
- Minor number (20-bit)
  - 부번호는 동일한 디바이스가 여러 개인 경우, 이들 디바이스들을 구분하는 용도로 사용함. 즉 같은 종류의 디바이스의 경우 주번호는 같고, 부번호를 다르게 할당
  - 부번호는 커널에서 사용하지 않고 디바이스 드라이버 내에서 디바이스를 구분하기 위해 사용
- 커널 2.6
  - typedef __u32 __kernel_dev_t;
  - typedef __kernel_dev_t dev_t; /* 32 비트 */
  - 디바이스 번호에 32비트를 할당하며, 주번호는 12 비트로, 부번호는 20비트로 표현함. 따라서 커널 2.6에서는 4096개의 주번호를 사용할 수 있음

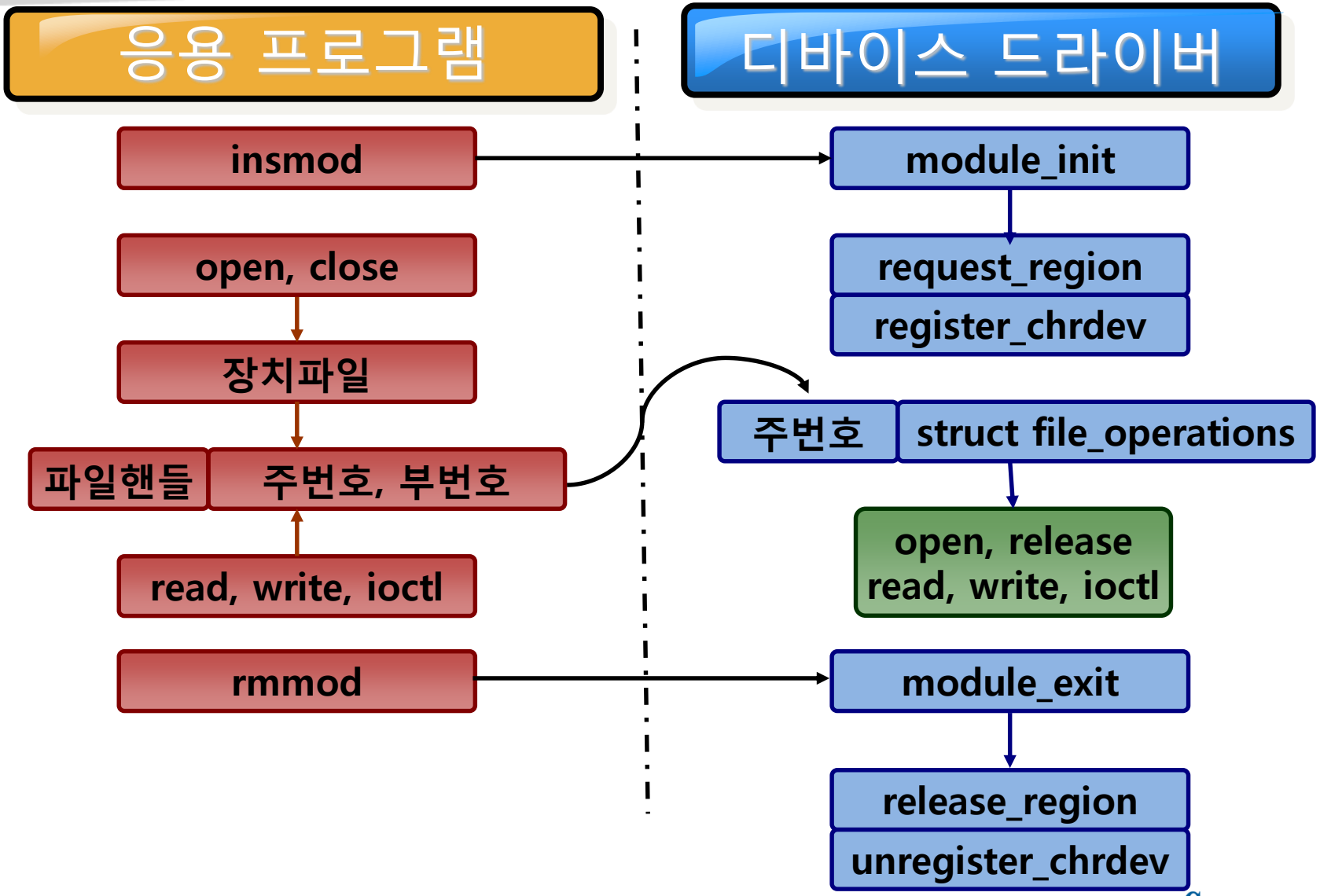
[KERNEL]/include/linux/major.h

```
#define MAX_CHRDEV          255
#define MAX_BLKDEV          255
#define UNNAMED_MAJOR       0
#define MEM_MAJOR           1
#define RAMDISK_MAJOR       1
#define FLOPPY_MAJOR        2
#define PTY_MASTER_MAJOR    2
#define IDE0_MAJOR          3
#define PTY_SLAVE_MAJOR     3
#define HD_MAJOR            IDE0_MAJOR
#define TTY_MAJOR           4
```

Device Driver와 응용프로그램과의 호출관계



디바이스 드라이버 호출



디바이스 드라이버의 기본 골격 (1)

% vi /root/work/dd/basic_device.c

```
/* basic_device.c */
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/module.h>

#define IOM_MYDEVICE_MAJOR_NUM 0
#define DEV_NAME "/dev/mydevice"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("HGU");

int mydevice_init(void);
void mydevice_exit(void);
module_init(mydevice_init);
module_exit(mydevice_exit);

int mydevice_open (struct inode *, struct file *);
int mydevice_release (struct inode *, struct file *);
ssize_t mydevice_read (struct file *, char __user *, size_t, loff_t *);
ssize_t mydevice_write (struct file *, const char __user *, size_t, loff_t *);
int mydevice_ioctl (struct inode *, struct file *, unsigned int, unsigned long);
```

디바이스 드라이버의 기본 골격 (2)

```
struct file_operations mydevice_fops = {
    .owner = THIS_MODULE,
    .open = mydevice_open,
    .release = mydevice_release,
    .read = mydevice_read,
    .write = mydevice_write,
    .ioctl = mydevice_ioctl,
};

int mydevice_open (struct inode *inode, struct file *filp)
{
    return 0;
}

int device_release (struct inode *inode, struct file *filp)
{
    return 0;
}

ssize_t device_read (struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
    return 0;
}
```

디바이스 드라이버의 기본 골격 (3)

```
ssize_t device_write (struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
    return 0;
}
int device_ioctl (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
{
    return 0;
}
int __init mydevice_init(void)
{
    int major_num;
    major_num = register_chrdev(IOM_MYDEVICE_MAJOR_NUM , DEV_NAME, &mydevice_fops);
    if ( major_num < 0 ) {
        printk(KERN_WARNING"%s: can't get or assign major number %d₩n", DEV_NAME,
IOM_MYDEVICE_MAJOR_NUM );
        return major_num;
    }
    printk("Success to load the device %s. Major number is %d₩n", DEV_NAME,
IOM_MYDEVICE_MAJOR_NUM );
    return 0;
}
void __exit mydevice_exit(void)
{
    unregister_chrdev(IOM_MYDEVICE_MAJOR_NUM , DEV_NAME);
    printk("Success to unload the device %s...₩n", DEV_NAME);
}
```

장치 등록과 해제

- Char Device Driver 등록 방법
  - 외부와 device driver는 file interface (node를 의미)를 통해 연결
  - Device driver는 자신을 구별하기 위해 고유의 major number를 사용

- 장치 등록과 해제
  - 등록 : int register_chrdev (unsigned int major,
    const char *name, stuct file_operations *fops)
    - major: 등록할 major number. 0이면 사용하지 않는 번호 중 자동으로 할당
    - name: device의 이름
    - fops: device에 대한 file 연산 함수들
  - 해제 : int unregister_chrdev (unsigned int major,
    const char *name)