4. 디바이스 드라이버

디바이스 드라이버란?.

- 장치를 직접적으로 다루는 소프트웨어로 하드웨어적인 장치가 아니더라도 스프트웨어 적인 장치를 만들어 다룰 수도 있다.
- 커널 모드에서 실행되고, 메모리에 상주하며, 스왑되지 않는다.
- 디바이스 드라이버를 통하여 장치를 파일처럼 다룰 수 있다.

4.1. 장치의 종류.

4.1.1 문자 장치(character device)

버퍼를 통하지 않고, 바로 읽고 쓸 수 있는 장치이다.
terminal, serial, parallel, keyboard, mouse, PC speaker 등이 있다.

4.1.2 블록 장치(block device)

버퍼 캐시를 통해 입출력을 하며, 장치의 어디든 접근이 가능하다.
블록단위로 입·출력을 하며 파일시스템을 구축할 수 있다.
floppy disk, hard disk, Ram disk, CD-ROM 등이 잇다.

4.1.3 네트웍 장치(network device)

네트웍 패킷을 송·수신할 수 있는 장치
ehternet, ppp 등이 있다.

4.2 커널 모듈(kernel module)

리눅스 시스템 부팅후에 동적으로 로드, 언로드할 수 있는 커널의 구성 요소이다. 커널을
다시 컴파일하거나 시스템을 리부팅하지 않고도 커널의 일부분을 교체할 수 있다.
디바이스 드라이바, 파일 시스템, 네트웍 프로토콜 등이 모듈로 만들어진다.

4.3 모듈 프로그램과 일반 응용 프로그램의 차이

: main() 함수가 없다
로드와 언로드시에 불리는 startup / cleanup 함수가 존재한다.
- startup : ini init_module(void)
  성공하면 0, 실패하면 그 외의 값을 돌려준다.
- cleanup : void cleanup_module(void)

: 모듈 프로그래밍시에 주의할 사항

- fault handling -모듈은 커널모드에서 동작하므로 메모리 접근에 대한 어떠한 보호도하지 않으며, 모듈에서 발생한 에러는 시스템에 치명적이므로 메모리를 다룰 때 주의 하고, 커널 함수 호출시에는 반드시 에러코드를 검사하고 에러를 처리해야 한다.
- 실수 연산이나 MMX 명령은 사용할 수 없다. 실수 연산은 정수 연산으로 대체하여 처리하도록 한다.
- 커널이 사용하는 스택 크기는 제한되어 있고(2 page), 인터럽트 핸들러도 동일

한

스택을 사용하므로 스택을 많이 사용하면 안된다. 스택에 큰 배열을 잡으면 안되고(동적으로 할당받도록 해야 한다). recursion이 많이 일어나지 않도록 주의한다.

- 다른 플랫폼과 고려해야 한다. 리눅스는 여러 환경에서 사용될 수 있으므로 32비트와 64비트 환경과 byte order 등을 모두 고려해야 하며 CPU에 특화된 코드

는

줄이도록한다.

4.4 모듈 관련 프로그램

- lsmod : 로드된 모듈의 목록을 보여준다.
- indmos : 커널 모듈을 로드한다.
- rmmod : 커널 모듈을 언로드한다.
- depmod : 커널 모듈간의 의존성을 검사한다.
- modprobe : 모듈을 보다 높은 수준에서 다루는 프로그램으로 의존성에 따라서 필요한 경우 다른 모듈을 로드, 언로드를 한다.
- kerneld : 커널 모듈 로더, 응용 프로그램이 아닌 커널 쓰레드의 일종으로 커널에서 모듈을 필요로 할 때 자동으로 모듈을 로그하고, 필요없을 때 언로드한다. modprobe 프로그램을 이용하여 로드, 언로드를 한다.

4.2. 디바이스 드라이버 만들기

4.2.1 간단한 모듈 프로그램밍하기-1

리눅스에 익숙하지 않은 사람은 디바이스 드라이버가 커널함수이기 때문에 디바이스 드라이버를 만들기위해서 는 전체 커널을 다시 컴파일해야 한다고 생각할지 모르나, 리눅스에서는 이러한 디바이스 드라이버를 커널에 쉽게 삽입할 수 있도록 커널모듈방식을 지원하고 있다. 우리가 사용하고 있는 대부분의 리눅스 시스템에는 이미 각 종 하드웨어 카드에 대한 디바이스 드라이버 모듈이 설치되어 있기 때문에 다음과 같은 명령으로 현재 설치되어 있는 모듈을 확인해 볼 수 있다.

/sbin/lsmod

그럼 이제부터 간단한 커널모듈을 만들고 이것을 커널에 설치해 보자.

커널모듈을 만들기 위해서는 반드시 init_module()과 cleanup_module() 함수가 정의되어야 하고, kernel_version 에 관한 정보를 담고있는 character 변수 kernel_version[]이 정의되어야 한다. 아래에 소스를 적어 놓았다.

kernel_version.c 프로그램소스

```
char kernel_version[]="2.0.35";
```

make_module.c 프로그램소스

```
#include <linux/kernel.h>
int init_module(void)
{
```

```
extern char kernel_version[];
return 0;
}
void cleanup_module(void)
{

}
```

위의 소스코드를 보듯 편의상 두개의 화일로 분리했으며, 커널버전은 자신의 리눅스 시스템의 커널버전을 입력하면 된다. 이 코드를 컴파일하기 위해서 다음과 같은 Makefile 을 사용하였다.

Makefile의 예

```
LD= ld -r
CFLAGS=$(INCLUDE) -D__KERNEL__ -DLINUX -O6
SRCS=kernel_version.c make_module.c
OBJS=kernel_version.o make_module.o

my_driver.o: $(OBJS)
$(LD) -o my_driver.o $(OBJS)
clean:
rm -f *.o
```

이상에서 보듯이 커널모듈은 main 함수를 갖고 있지 않으며, 실행화일이 아닌 오브젝트화일로 만들어진다. 이제 make 를 실행시키면 my_driver.o 오브젝트화일이 생성될 것이다.

이제 이 오브젝트화일을 커널에 삽입시켜야 하는데 그 명령은 다음과 같다.

```
/sbin/insmod my_driver.o
```

모듈을 삽입하였으면

```
/sbin/lsmod
```

를 입력하여 삽입된 모듈을 확인할 수 있다.

이제 init_module을 다음과 같이 수정하여, 모듈 삽입시에 메시지를 출력하고록 해보자.

```
int init_module(void)
{
extern char kernel_version[];
printk("module test₩n");
return 0;
}
```

여기서 주의할 점은 일반적인 표준출력함수 printf 대신 printk를 사용해야 한다는 것이다. 이제 커널 모듈을 고쳤기 때문에 make 를 입력하여 재컴파일 한 후 새로운

my_driver.o 를 커널에 삽입시켜야 한다. 그러기 위해서는 기존에 설치된 my_driver.o 를 제거해야 한다. 이 작업은 다음과 같다.
/sbin/rmmod my_driver (my_driver.o 가 아님)
이제 다시
/sbin/insmod my_driver.o
를 입력하여 커널에 모듈을 삽입해보자.이제 모듈 삽입과 동시에 터미널에 메세지가 출력되는 것을 확인할 수 있을 것이다.(커널 메시지를 확인하려면 X server 가 떠 있는 화면에서는 볼 수 없고, 터미널로 화면을 전환해야한다.)

4.2.2. 모듈 프로그램밍하기-2

4.2.1장에서는 단순한 커널모듈을 만들고 설치하는 방법을 다루었는데, 이번에는 실제 디바이스와 이 커널 모듈을 연결시키는 소스코드를 만들어보자.
리눅스에서는 모든 하드웨어 장치는 화일로 취급하고 다루게 된다. /dev/ 디렉토리를 보면 각종하드웨어 장치에 연결된 화일을 볼 수 있다. 예를 들어 시리얼포트의 경우는 /dev/cua0 와 같은 장치화일로 정의되는데, 다음 과 같이 list를 해보면
ls -l /dev/cua0
다음과 같은 출력을 얻을 수 있다.
crw-rw---- 1 root uucp 5, 64 Aug 23
11:57 /dev/cua0
여기에서 일반적인 보통화일이나 디렉토리화일이 - 나 d 로 시작되는데 비해 c 로 시작되는 것을 볼 수 있다. 이는 시리얼포트가 문자형(character) 디바이스 임을 나타낸다. 디바이스는 일반적으로 블록 디바이스와 문자형 디바이스로 지정할 수 있다. 이는 하드웨어자체가 문자 디바이스와 블록디바이스로 나누어져 있다는 것이 아니라 디바이스 화일이나 디바이스 드라이버를 정의할 때, 문자형이나 블록형으로 설정할 수 있다는 말이다.(한 하드웨어에 대하여 동시에 문자형과 블록형의 지정도 가능) 문자형 디바이스는 유저프로그램이 read 나 write 등의 시스템 콜을 호출할 때마다 커널의 읽기, 쓰기 루틴이 호출되는 방식이고, 블록형의 경우 설정된 조건이 만족되면 커널이 자동적으로 특정 루틴을 호출하는 방식이다. 우리는 간단히 문자형 디바이스 드라이버에 대해서만 언급하기로 하겠다.
디바이스를 위한 장치화일을 만들어 보자. 쉘 상에서 다음과 같이 입력하여보자.
mknod /dev/mydev c 120 0
여기서 c 는 문자 디바이스를 만들기 위한 것이고, 120 은 디바이스의 major 번호 0 minor 번호이다.
major 번호와 minor 번호의 두개를 사용하는 이유는 같은 하드웨어카드에 여러개의 장치가 부착될 수 있기 때문인데, 우리의 경우는 한개만을 고려한 0 을 입력하였다. major 번호는 리눅스의 경우 1 부터 255 까지 할당할 수 있는데 다른 디바이스의 major 번호와 충돌되어서는 않되기 때문에 테스트를 위해 할당된 60-63, 120-127, 240-254 중의 한 번호를 사용하는 것이 좋다. 그런 다음 이 장치화일이 잘 만들어졌는지 다음과 같은 명령으로 확인해보자
ls -l /dev/mydev
crw-r--r-- 1 root root 120, 0 Aug 30

11:24 /dev/mydev
다음과 같은 출력을 얻는다면, 장치화일이 성공적으로 만들어진 것이다.
이제 이 디바이스와 우리가 작성한 커널모듈을 연결시키기 위해 make_module.c 를 다음과 같이 수정하겠다.
make_module.c 소스코드

```
#include <linux/kernel.h>
#include <linux/fs.h>

extern int mydevread(struct inode *, struct file *, char *,int);
extern int mydevwrite(struct inode *, struct file *,char *,int);
extern int mydevioctl(struct inode *, struct file *, unsigned int, unsigned long);
extern int mydevopen(struct inode *, struct file *);
extern void mydevclose(struct inode *, struct file *);

struct file_operations mydev_fops = {
NULL,
mydevread,
mydevwrite,
NULL,
NULL,
mydevioctl,
NULL,
mydevopen,
mydevclose
};
int mydevmajor = 120;
int init_module(void)
{
extern char kernel_version[];
if( register_chrdev
(mydevmajor,"mydev",&mydev_fops) ){
printk(" cannot get Major O₩n", mydevmajor);
}
return 0;
}
void cleanup_module(void)
{
unregister_chrdev(mydevmajor,"mydev");
}
```

이 코드에서 보듯 장치화일과 디바이스드라이버를 연결시켜주는 역할을 하는 함수는

바로
register_chrdev(mydevmajor,"mydev",&mydev_fops))
이다. 이 함수의 첫번째 인수는 바로 디바이스 화일의 major 번호(=120) ,이고 두번째 인수는 장치화일 이름이다. 3번째 인수는 file_operations 구조체의 포인터인데 이 포인터는 사용자 프로세스가 시스템콜을 할 때 대응해서 호출되는 함수의 핸들(주소)를 담고있다. 이 구조
체에 모든 함수를 다 정의할 필요는 없고, 디바이스제어에 필요한 함수만을 정의하고 그 핸들을 할당해 주면 되는데, 우리는 기본적인 함수 read, write, open, close 그리고 유연한 시스템 콜을 가능하게 해주는 ioctl 을 포함시켰다.
이제 이러한 시스템 콜에 대응하는 함수에 대한 코드를 작성해야 하는데, 이 코드는 mydev_fops.c 화일로 별도로 분리시켰다.

mydev_fops.c 의 소스

```
#define OK 0
int mydevread(struct inode *inode, struct file *file,char *buffer,int count)
{
printk("read₩n");
return(OK);
}
int mydevwrite(struct inode *inode, struct file *file, char *buffer,int count)
{
printk("write₩n");
return(OK);
}
int mydevioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
{
printk("ioctl₩n");
return(OK);
}
int mydevopen(struct inode *inode, struct file *file)
{
printk("open₩n");
return(OK);
}
void mydevclose(struct inode *inode, struct file *file)
{
printk("close₩n");
}
```

위에서 보듯이 시스템 콜의 대응함수이지만 아직은 아무런 하드웨어와도 연관시키지

않았으며, 단순히 함수가 이상없이 호출되는지를 체크하기 위해 간단한 출력만을 하도록 구성하였다.
새로운 화일을 작성하였으므로 Makefile 에 이 화일을 추가하여 다음과 같이 수정하였다.
수정된 Makefile

```
LD= ld -r
CFLAGS=$(INCLUDE) -D__KERNEL__ -DLINUX -O6
SRCS=kernel_version.c make_module.c mydev_fops.c
OBJS=kernel_version.o make_module.o mydev_fops.o
my_driver.o: $(OBJS)
$(LD) -o my_driver.o $(OBJS)
clean:
rm -f *.o
```

이제 'make clean' 으로 *.o 화일을 삭제한 후'make' 로 새로운 my_driver.o 를 만들어라.
다음으로

```
/sbin/rmmod my_driver
/sbin/insmod my_driver.o
```

를 하여 새로운 드라이버모듈을 커널에 설치하였다.

이제 이 디바이스 드라이버를 호출하게 될 유저 프로그램을 작성해보자. 각 서브루틴이 잘 호출되는지를 보기위해서 open, read, ioctl, write, close 등을 한번씩 호출하도록 해보았다.
user.c 소스코드

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
main()
{
int fd;
unsigned int cmd;
unsigned long arg;
char buf[100];
int count=0;
fd=open("/dev/mydev",O_RDWR);
read(fd,buf,1);
write(fd,buf,1);
ioctl(fd,cmd,arg);
```

```
close(fd);

}
```
이 코드는 간단하므로 별도의 Makefile은 만들지 않고 다음과 같이 컴파일 링크시켜보자
```
gcc user.c -o user.exe
```
이제 ./user.exe 를 실행시키면 콘솔화면(X-server 화면에서는 보이지 않음, 터미널로 전환해서 확인) 화면에
```
open
read
write
ioctl
close
```
등의 메시지가 나오면 지금까지의 과정이 완성된 것이다.
지금까지의 내용을 이해한 눈치빠른 프로그래머라면 사용자 프로그램이 어떻게 하드웨어를 제어할 수 있는지를 짐작할 수 있을 것이다.
1. 사용자 프로그램에서는 하드웨어를 직접제어할 수 없으므로 open, read,.. 등의 시스템콜을 호출한다.
2. 디바이스 드라이버는 각 시스템 콜에 맞는 함수루틴을 갖고 있으므로 시스템 콜에 대응해 함수를 호출한다. 디바이스 드라이버는 커널의 일부이기 때문에 이들 함수에서는 하드웨어에 직접 접근할 수 있다.

3. 모듈 프로그래밍하기 -커널에 포함하여 컴파일한다

하나의 모듈을 만들어 커널과 함께 컴파일 하는 방법에 대해서 알아본다.
우선 가상의 하드웨어를 하나 정하기로 한다. 이 하드웨어의 성격은 다음과 같다.

문자 디바이스(character device)이다.
인터럽트 방식이다.
하드웨어에는 자체적인 CPU가 있다.
PC의 운영체제와는 shared memory 방식으로 교신한다.
인터럽트와 shared memory 는 카드 설정에 따라 바뀔 수 있기 때문에
부팅할 때나 모듈 적재 시에 옵션으로 바꿀 수 있다.
동시에 같은 하드웨어를 다른 인터럽트를 주고 2개 이상 사용할 수 있다.

위와 비슷한 하드웨어 중에 커널에 포함 된 것은 지능형 멀티 시리얼 포트 종류가 있다. 크게 말하면 사운드 드라이버도 포함 될 수 있을 것이다. 이 정도의 하드웨어를 가정한다면 커널 프로그래밍에 필요한 대부분의 테크닉이 모두 동원되어야 할 것이다. 문자 디바이스 보다 블럭 디바이스가 좀더 복잡하지만 버퍼 입출력만 제외한다면 크게 다르지 않다. 이 하드웨어를 my_device 라고 명명하고 리눅스 커널에서 제대로 동작하게 하기 위해서 필요한 작업을 해보도록 한다.

4.3.1 장치 특수파일

프로그래밍을 할 때 어떤 장치를 열기 위해 open 함수를 호출 한다고 하자. 표준 라이브러리에 있는 open 이라는 함수를 사용하면 이 함수는 커널에 시스템 호출을 하고 커널은 이 호출이 요구하는 파일에 대한 요청 작업을 처리한다. 유닉스의 특성상 모든 디바이스의 입출력은 파일 입출력과 차이가 없다. 우리가 만들어야 하는 것은 추상화 되어 있는 파일 입출력이 최종적으로 호출하게 될 각 장치에 고유한 열기 방법에 관한 것이다. 어떤 장치에 접근 하기 위해서 가장 먼저 해야할 작업은 장치마다 다를 것이다. open 이라는 호출이 오면 필요한 일을 하고 write/read 호출에 대비해서 준비를 하는 작업만 하면 된다.

이 작업은 프로그램 언어만 C를 사용할 뿐 C 라이브러리 함수를 하나도 사용할 수 없는 특수 작업이므로 응용 프로그래밍과 완전히 다른 작업이라고 할 수 있다. 파일 입출력의 상위 인터페이스는 리눅스에서도 다 완성되어 있기 때문에 우리가 신경을 쓸 필요는 없다. 장치 특수파일을 정의하고 표준 파일 입출력에 사용되는 시스템 호출이 사용할 적절한 함수를 만들어 내는 작업만으로 충분하다.

장치 특수파일을 위해서 파일유형, 주(major) 번호와 부(minor) 번호가 필요하다. 주번호는 장치유형을, 부번호는 그 유형의 단위기기를 나타낸다.

리눅스에서는 255 번까지의 장치특수파일 주번호가 있다. 수많은 하드웨어 지원이 계속 추가되고 있어서 100번 이하는 거의 다 할당이 되었다. 특정한 하드웨어를 위한 디바이스드라이버를 만들고 공식적으로 리눅스의 주번호를 받고 싶으면 Documentation/devices.txt 를 참고하여 리눅스 주번호 관리를 하는 사람에게 연락을 취하면 된다. 임의로 번호를 부여한다면 커널이 업그레이드되어 다른 디바이스드라이버가 이 번호를 사용할 때 충돌이 있을 수 있다.

리눅스를 테스트하거나 실험적으로 디바이스드라이버를 만드는 사람을 위해서 60-63, 120-127, 240-254 번이 예약되어 있다. 이 번호 중에서 임시로 적당한 번호를 사용하여 테스트 하고 나중에 정식으로 번호를 할당 받으면 된다. 우리가 만드는 장치를 위해서 125번을 선택하여 MY_DEVICE_MAJOR로 정하자.

이 번호를 커널에 등록하기 위해서는 include/linux/major.h 에 이 값을 기록한다. 위치는 상관없다. 만약 계속 커널 업그레이드에 따라 갈 예정이고 아직 정식으로 커널 배포본에 등록이 안된 테스트 드라이버라면 커널 패치를 할 때 문제가 생기지 않도록 되도록 가능한 가장 뒷부분에 배치하는 것이 좋을 것이다. 우리가 삽입한 코드에 인접한 곳에서 커널 변경 사항이 생긴다면 패치할 때 문제가 생길 수 있다.

```
#define MY_DEVICE_MAJOR 125
```

그리고 이 번호로 된 장치 특별파일을 만든다.

```
mknod /dev/my_device0 c 125 0
mknod /dev/my_device1 c 125 1
mknod /dev/my_device2 c 125 2
mknod /dev/my_device3 c 125 3
chown root.root /dev/my_device
chmod 660 /dev/my_device
```

이제 my_device 는 이 파일을 열고 쓰고 읽음으로써 조작할 수 있다. 참고로 c는 문자 특수파일임을 나타내고 125는 주번호, [0-3]은 부번호이다.

4.3.2 MY_DEVICE를 커널 컴파일 옵션에 삽입

make config 시에 MY_DEVICE 항목이 나오게 하기 위해서 필요한 작업을 하자. 디바이스 드라이버는 계층상 가장 하위에 위치 하기 때문에 디렉토리 위치도 살펴야 한다. MY_DEVICE는 문자 특수파일이며 isdn등과 같이 특수한 분류에 들어가지 않기 때문에 drivers/char 디렉토리에서 작업을 하면 된다.

make config 를 했을 때 MY_DEVICE 항목이 나오게 하기 위해서는 drivers/char/Config.in 에 my_driver에 해당하는 조건을 명시해야 한다.

```
tristate 'insert my device driver in kernel' CONFIG_MY_DEVICE
if [ "$CONFIG_MY_DEVICE" = "y" -o "$CONFIG_MY_DEVICE" = "m" ]; then
int ' my device value' CONFIG_MY_DEVICE_VALUE 1
bool ' support my device something' CONFIG_MY_DEVICE_SUPPORT
fi
```

tristate는 커널에 직접삽입되거나(y), 모듈로 만들거나 (m), 컴파일하지 않는(n)다는 것을 정하는 것이며 bool 은 (y,n) 두가지 중에 선택하는 것이고 int 는 필요한 수치값이 있으면 적어 주는 부분이다.

if-fi 는 여러 계층을 둘 수 있다. 다른 if-fi 문장 사이만 아니라면 이 문장을 삽입하는 위치는 상관이 없다.

help 버튼을 눌렀을 때 설명이 나오게 하기 위해서 Documentation/Configure.help에 적절한 설명을 넣어 준다.

```
My device support
CONFIG_MY_DEVICE
This text is help message for my device driver
```

3.3 커널 부팅 옵션의 처리

커널이 부팅할 때 디바이스 드라이버들은 제어할 수 있는 하드웨어에 대한 설정값을 스스로 찾거나 고정된 값을 사용할 수 있다. 만약 사용자가 어떤 디바이스에 대한 인터럽트값이나 베이스 어드레스등을 바꾸었다면 커널에게 알려 주어야 한다. 설정값을 스스로 찾을 수 없는 디바이스드라이버도 마찬가지로 사용자가 하드웨어 설정값을 알려 주어야 한다. 이를 위해 리눅스 커널은 부팅 옵션을 지원한다.

디바이스 드라이버를 위한 파일 drivers/char/my_device.c을 만들고 헤더파일 include/linux/my_device.h를 만든다. 컴파일 할 때 이 프로그램도 컴파일 시키기 위해서 drivers/char/Makefile의 적당한 곳에 이 파일을 적어준다.

```
ifeq ($(CONFIG_MY_DEVICE),y)
L_OBJS += my_device.o
else
ifeq ($(CONFIG_MY_DEVICE),m)
M_OBJS += my_device.o
endif
endif
```

커널에 직접 삽입(y)하라는 옵션과 모듈로 만들(m)라는 옵션일 때 각각에 대해서 다른 처리를 한다. 그외에 Makefile 을 살펴보면 서브디렉토리를 포함하라는 옵션등이 있다. drivers/char/mem.c 에서 문자 디바이스드라이버를 초기화한다. 드라이버 초기화 함수는 대부분 *_init 형식이므로 my_driver_init라고 정하고 적당한 곳에 넣어 주면 된다. 비슷한 드라이버 초기화 루틴이 삽입되어 있는 위치에 다른 ifdef-endif와 상관없는 곳에 두면 된다.

```
#ifdef CONFIG_MY_DEVICE
my_device_init();
#endif
```

그리고 윗 부분에 함수의 원형을 선언해야 한다. 다음과 같이.

```
extern int my_device_init(void);
```

my_device_init 함수는 my_device를 초기화 시킬 때 필요한 각종 작업을 할 때 한 번 실행되는 함수이다. my_device_init 함수에서 my_device_open, my_device_write 함수를 커널에 등록하게 된다.

make dep 과정에서 Config.in 을 참조하여 Makefile 에 정의된 디바이스 드라이버 파일을 컴파일 하여 커널에 삽입할 것인지, 모듈로 만들 것인지 여부를 결정한다.
부팅과정을 진행한 후에 커널은 디바이스 드라이버 초기화 루틴(drivers/char/mem.c)

으로 뛰어 각 디바이스를 실제로 초기화(my_device_init) 한다. 각각의 디바이스 초기화 함수는 하드웨어 디바이스가 컴퓨터에 존재하는지 검사하고, 하드웨어가 필요로 하는 초기화를 한 후에 system call 루틴을 위해서 read, write, release, ioctl 등의 함수가 정의된 file_operation 함수배열을 등록한다.

모듈로 만들었을 때는 커널 부팅과정을 insmod가 해 준다. 인자 파싱도 마찬가지로 insmod 몫이다. 디바이스 드라이버는 인자 배열을 넘겨 받게 되고 init_module(my_device_init)에서 마찬가지로 하드웨어 검색, 초기화, 인자를 사용한 설정값 변경을 한다. 모듈로 했을 때는 드라이버 프로그램에서 약간의 부가 작업이 필요할 뿐 직접 삽입된 드라이버와 다르지 않다.

4.4 디바이스 드라이버의 적재와 삭제

4.4.1 드라이버 초기화 함수

초기화 함수에서는 특정 주소로 값을 보내기, 바이오스 다운로드, 지원 디바이스 중에서 장착된 디바이스를 구별하는 작업 들을 해야 한다. 커널 쪽으로는 인터럽트 등록, 드라이버 등록, 디바이스 갯수와 번지를 지정하는 작업이 필요하다. 개발 초기에는 모듈적재 방법을 사용 해야 커널 재컴파일 시간을 줄일 수 있으므로 모듈 적재 방법을 중심으로 살펴보자.

my_device_init

우선 my_device 드라이버를 등록하자. 등록 함수는 register_chrdev이다. 이 함수는 주 장치 번호, 디바이스 이름, 파일 조작 함수 인터페이스를 등록한다. 커널에서 이 디바이스를 사용하는 인터페이스는 다음과 같다.

```
static struct file_operations my_device_fops = {
my_device_lseek,
my_device_read,
my_device_write,
my_device_readdir,
my_device_select,
my_device_ioctl,
my_device_mmap,
my_device_open,
my_device_release
};
```

커널에서는 어떤 디바이스라도 파일과 같이 접근하므로 file_operations 구조체로 정의하고 read/write 루틴을 my_device에 고유한 방법으로 작성하면 된다. 문자 디바이스와 블럭 디바이스에 따라 지원하는 함수는 차이가 있다. 예를 들어,

my_device_readdir 함수는 문자 디바이스인 my_device에서는 전혀 의미가 없다. 이 함수는 커널 자체에서 전혀 사용하지 않으므로 NULL로 정의해도 아무 상관이 없다. 어떤 함수가 필요하고 어떤 함수가 필요 없는지는 크게 블록디바이스와 문자 디바이스에 따라 차이가 난다. 만들려는 디바이스의 특성을 고려하여 연관이 있는 드라이버를 조사해 파악하기 바란다.

디바이스가 사용할 번지 주소가 유효한지는 검사하는 함수는 check_region 이다. 주소가 유효하다면 request_region 으로 이 주소를 점유한다. 해제는 release_region이다. 초기화 과정에서 드라이버나 디바이스가 문제가 있으면 반드시 이 함수로 영역을 반환하여야 한다. 함수 정의는 소스를 찾아 보기 바란다. 앞에서 커널 코드를 생략한다고 말했듯이 함수 설명도 가능한한 함수명만을 보이고 정확한 정의는 생략한다.
인터럽트를 사용한다면 이것을 사용하겠다고 요청해야 한다. request_irq/free_irq를 사용하면 된다. request_irq 의 세번째 인자는 SA_INTERRUPT를 사용한다. SA_INTERRUPT는 빠른 irq라고 정의되어 있다. 이 플래그는 신속한 인터럽트 서비스를 위해서 다른 인터럽트를 금지하고 가장 우선으로 인터럽트 처리 루틴을 수행한다. 문맥교환, 재인터럽트, 프로세스 blocking이 일어나지 않는다. 여기에 0을 쓰면 느린 irq로 작동된다. 이 인터럽트는 인터럽트를 금지 하지 않으므로 또 다시 인터럽트가 발생할 수 있다. SA_SHIRQ는 pci 디바이스가 인터럽트를 공유할 때 사용한다. 함수 원형과 자세한 옵션 플래그 설명은 include/asm/sched.h 에서 찾아 볼 수 있다.

request_region, request_irq 를 수행했을 때 요청한 값을 사용할 수 있으면 디바이스가 실제로 장착되어 있는지 검사해야 한다. 디바이스의 정해진 주소에 장치 종류를 나타내는 문자열 같은 식별자가 있다면 그 것을 읽어 오면 된다. 인터럽트를 사용한다면 테스트를 위해 카드가 인터럽트를 일으키도록 한 다음에 돌아오는 인터럽트를 검사하면 된다. 이 방법은 register_chardev(또는 register_blkdev,tty_register_driver)를 사용하여 미리 인터럽트 함수를 등록하고 사용해야 한다. 이런 방법을 쓸 수 없다면 지정한 번지에 디바이스가 있다고 가정하고 초기화를 해야한다. lilo 옵션이 틀리지 않았다는 가정을 해야 하기 때문에 다른 장치와 충돌한다면 심각한 문제가 생길 수 있으므로 가능하다면 확실한 방법을 강구해야 한다. 지정한 장치가 실제로 있는지 검사하는 루틴은 모든 드라이버에 있으므로 잘 살펴보기 바란다.

드라이버가 사용할 메모리 영역이 필요하다면 kmalloc 함수로 일정 영역을 확보해 놓는다. kmalloc은 데이타를 저장할 메모리 영역을 확보하는데 사용하고 request_region 함수는 특정 주소의 메모리 영역을 확보하는데 사용한다.

디바이스에 바이오스를 다운로드 하여 활성화 시켜야 한다면 상당한 고민을 해야 한다. 바이오스의 크기가 적다면 바이너리 파일을 바이트 배열로 만들어 정의해 두고 memcpy_toio 함수를 이용해서 다운로드 할 수 있다.(drivers/char/digi_bios.h) 몇 메가씩 되는 데이타를 다운로드 해야 한다면 이 방법을 사용할 수 없다. 바이트 배열은 정적 데이타가 되어 커널을 적재할 수도 없도록 크게 만들게 되기 때문이다. 가능한 방법은 일단 드라이버를 등록하고 ioctl루틴을 이용해서 디바이스를 열어서 다운로드

하고 그 결과에 따라 드라이버를 활성화거나 우회적으로 드라이버를 사용금지 또는 강제 삭제를 해야 한다.

4.4.2 드라이버 삭제 함수

모듈 방식으로 드라이버를 적재 했으면 rmmod 로 모듈을 삭제하는 함수를 작성해야 한다. cleanup_module로 함수 명이 통일되어 있고 모듈일 때만 필요하므로 #ifdef MODULE - #endif 안에서 정의해야 한다. 우선 kmalloc으로 할당받은 메모리를 kfree 함수로 반납한다. request_region 으로 확보한 주소는 release_region으로 해제하고 request_irq로 인터럽트를 사용했다면 free_irq로 반납한다. 마지막으로 장치 특수번호를 해제하여 커널이 디바이스를 사용하지 않도록 한다. 이 함수는 unregister_chrdev/tty_unregister_device 이다.

그외에 ioremap등의 함수를 사용했으면 이들과 짝이 되는 iounmap 등의 함수를 사용하여 모든 자원을 반납해야 한다. 자원을 제대로 반납하지 않고 모듈을 삭제하면 엄청난 양의 에라 메세지가 /var/log/messages에 쌓이게 된다. 좀 더 심각한 상황이 생겨서 파일 시스템 전체를 잃게 될 수도 있기 때문에 할당받은 자원은 반드시 반납하는 함수를 철저히 확인하여 사용해야 한다.

4.4.3 헤더파일의 구조

헤더파일은 반드시 아래와 같이 시작하고 끝내야 한다.

```
#ifndef __LINUX_MY_DEVICE
#define __LINUX_MY_DEVICE
…
…
#endif
```

이렇게 해야 여러 파일에 헤더파일이 인클루드 되었어도 문제가 생기지 않는다. 커널에서 사용하는 변수이지만 사용자 프로그램에서 보여서는 안되는 변수가 있다면 아래와 같이 막아 놓아야 한다.

```
#ifdef __KERNEL__
…
#endif
```

그리고 my_device_init 는 외부에서 참조 할 수 있도록 헤더파일에 꼭 선언해야 한다. init 함수는 커널에서만 사용하므로 __KERNEL__ 내부에 존재해도 상관은 없다.

4.4.4 초기화 할 때 많이 쓰는 함수들

outb/outw, inb/inw 함수는 물리 주소에 쓰기/읽기 함수이다. 이름이 보여 주듯이 바이트,워드를 읽고 쓴다. readb/writeb 함수는 memory mapped I/O 장비에 읽고 쓰는 함수이다. memcpy_toio/memcpy_fromio 함수는 특정 주소에 데이타를 인자로 준 바이트만큼 쓴다. 각 플랫폼에 따라 커널이 보는 주소와 cpu가 보는 주소, 그리고 물리 주소의 차이를 없애는 역활을 한다. 물리 주소와 가상주소 시스템 버스와의 관계가 복잡하고 여러 플랫폼에 따라 주소 지정법이 다르다. x86 아키텍쳐에서는 물리주소와 memory mapped 주소가 동일하지만 다른 플랫폼에서는 x86과 다르기 때문에 호환성을 위해서 상당한 주의를 해야 한다. Documentation/IO-mapping.txt를 살펴보면 라이너스가 메모리 접근함수 사용시 주의할 점에 대해서 설명해 놓았다. 디바이스가 범용 플랫폼에서 동작하기를 바란다면 꼭 읽어 보아야 한다.

cli()함수를 사용하여 인터럽트를 금지 시키고 중요한 작업을 한 다음에 sti() 함수로 인터럽트를 가능하게 만든다. 드라이버 프로그램을 부르는 함수에서 부르기 전후에 플래그 상태가 변화한다면 문제가 발생할 수 있기 때문에 cil/sti 짝으로만 쓰지는 않고 save_flags(flags); cli(); sti(); restore_flags(flags); 형식으로 쓴다. 이렇게 하면 불리기 전의 상태가 보존되므로 드라이버 프로그램 안에서 안심하고 플래그를 조작할 수 있다. sti()는 드라이버 프로그램 안에서 인터럽트를 가능하게 할 필요가 있을 때 사용하면 된다. 인터럽트 가능 불가능에
관계없이 드라이버가 불릴 때의 상태에서 동작해도 상관없다면 save_flags;cli;restore_flags를 사용하면 된다. 원 상태가 인터럽트 가능이라면 restore_flags가 sti 함수 역활도 하기 때문이다. 주의 할 것은 드라이버 함수에서 여러 하위 루틴으로 뛰는 동안에 save_flags;cli;restore_flags 순서가 유지되어야 하는 것이다. 함수가 조건문 등으로 분기 할 때 차칫 restore_flags 함수가 수행되지 않는 등의 오류가 있으면 시스템이 정지하게 된다.

이런 에러는 상당히 발견하기 어려운 것이다. 어디서 시스템이 정지 했는지 정보를 얻기가 힘들다. printk함수를 사용해서 소스의 위치를 추적하더라도 찾기 힘들다. printk 함수는 수행되는 즉시 문자열을 /var/log/messages 나 콘솔 화면에 쓰지 않고 쓸 데이터가 많거나 시스템이 바쁘지 않은 동안에 flash 를 하므로 printk 함수가 아직 완전히 수행되지 않은 상태에서 시스템이 정지하는 경우가 많기 때문이다. 그러므로 코딩시에 철저히 save_flags;cli;restore_flags의 순서와 흐름을 따져서 사용해야 한다.

디바이스에 어떤 조작을 하고 일정 시간을 기다릴 때에 사용할 수 있는 다양한 함수가 존재한다. 디바이스를 조작하고 결과를 체크하는 일을 수 밀리 초의 간격 안에 모두 해야 하는 경우가 있다. MY_DEVICE에 차체 cpu가 들어 있고 이 cpu를 활성화 시키기 위해서는 MY_DEVICE 의 base 주소에 0을 쓰고 400ms 와 500ms 사이에 1을 써야 한다고 하자. 이때에는 절대적인 대기 함수를 사용해야 한다. 여러 문서에 대기에 대한 함수 설명이 있지만 필자의 경험으로는 __delay() 함수 만 이런 기능을 정상적으로 수행했다. 인자는 __delay((loops_per_sec/100) * (원하는ms))를 사용하면 된다. 이 함수는 문맥교환이 일어나지 않기 때문에 실행되는 동안 시스템이 정지해 있게 된

다. 드라이버의 일반 작업에 사용하면 효율이 엄청나게 떨어 지므로 절대적인 시간이 필요한 초기화 작업 같은 경우를 제외하고는 이 함수를 사용해서는 안된다.

일반 작업에서 대기 함수가 필요하면 다음과 같이 사용하면 된다.

```
current->state = TASK_INTERRUPTIBLE;
current->timeout = jiffies + HZ * 3;
schedule();
```

current 란 현재 수행되고 있는 프로세스를 말한다. 이 프로세스는 일정한 시스템 시간을 할당받아 현재 이 라인을 수행하고 있다. 인터럽트가 가능하게 만들고 (TASK_INTERRUPTIBLE) 깨어나는 시간을 앞으로 3초 후로 정하고 (HZ*3) 잠들게 (schedule)한다. jiffies란 시스템의 현재 시간을 의미한다. HZ는 1초에 해당하는 값이고 schedule은 현재의 프로세스를 블럭 시키는 함수이다. 커널은 이 프로세스가 잠들고 있는 동안 다른 작업을 할 수 있게 되기 때문에 효율을 높일 수 있다. timeout이 3초로 되어 있지만 반드시 이 시각에 깨어난다는 보장은 없다. 깨어날 수 있는 보장이 3초 후 부터라는 것일 뿐이다. timeout 값이 너무 작으면 문맥교환이 자주 일어나서 효율이 떨어지고 너무 크면 드라이버 작업 수행 성능이 떨어지므로 대기 시간을 잘 조사해서 적당한 값을 설정해야 한다.

커널은 프로세스에게 각종 signal 을 줄 수 있다. include/asm/signal.h에서 정의된 여러 시그널을 받은 프로세스는 가능한한 신속하게 작업을 끝내기 위해서 블럭되지 않으므로 schedule 이 무시되기 때문에 코딩을 할 때 이 것을 염두에 두고 깨어 났을 때 정상상태에서 깨어난 것인지 signal을 받았는지 구별해서 동작하게 해야 한다. signal을 받았는지는 깨어난 후에 다음과 같이 알아 낼 수 있다.

```
if(current->signal & ~current->blocked)
//signal
else
// no signal
```

시그널을 받은 프로세스는 더 이상 작업을 진행하는 것이 의미가 없기 때문에 디바이스를 연 프로세스의 갯수가 기록된 변수의 처리등 꼭 필요한 일만 하고 신속하게 끝내야 한다. 2.2 버전에서는 signal_pending(current)로 바뀌었다.

앞에서 말한 대기 함수는 대기하는 프로세스가 일정 시간 이후에 깨어나서 바뀐 조건을 검사하는 것들이다. 대기하고 있는 동안에 다른 루틴에서 조건을 변화 시키고 대기 프로세스를 깨워 준다면 조건이 만족하기 전에 깨어났다가 다시 잠드는 오버헤드를 줄일 수 있을 것이다. 이를 위해서 sleep/wake_up 함수가 있다.

sleep_on/interruptible_sleep_on, wake_up/wake_up_interruptible 함수는 잠들고 깨우

는 일을 한다. interruptible_sleep_on 함수는 signal이 오거나 wake_up_interruptible 함수가 깨워 주거나 current->timeout 값에 시스템 시간이 이르면 깨어난다. 이 함수를 실행하기 전에 timeout 값을 조정해 줄 수 있으므로 깨우는 함수가 제대로 실행되지 않았다고 판단할 만큼 충분한 시간을 주고 잠들게 하면 된다. 이 함수는 대기 큐가 필요하므로 사용하기 전에 전역변수로 큐를 정의하고 초기화 시켜 놓아야 한다. 함수명은 init_waitqueue 이다.

interruptible_sleep_on/wake_up_interruptible 함수는 인터럽트를 사용하는 드라이버에서 디바이스에 인터럽트를 걸어 놓고 잠들면 디바이스가 처리를 끝내고 인터럽트를 걸어 깨워 주는 경우에 많이 사용한다. 이 함수도 schedule 과 마찬가지로 signal을 받으면 무조건 깨어 나기 때문에 꼭 상태를 체크해야 한다.

sleep_on/wake_up 짝은 wake_up 함수가 반드시 수행된다는 확신이 있는 경우에 사용된다. sleep_on 으로 잠들면 timeout 값이 지나거나 시그널이 와도 깨어 나지 않고 오로지 wake_up 만이 깨울 수 있다. wake_up 함수가 수행되지 않는다면 시스템이 교착상태에 빠질 수 있기 때문에 100% 확신이 없으면 사용하지 않는 것이 좋다. drivers/char/lp.c 에서 사용되었지만 2.0.33에서는 interruptible로 바뀌었다. drivers/block 의 일부 드라이버에 사용예가 있으므로 참고하기 바란다.

그외에 커널 해커 가이드의 지원함수 부분에 나머지 대기 함수에 대한 설명이 있다. 함수 설명을 읽고 드라이버를 조사해서 사용 방법을 알아 두기 바란다. 드라이브가 shared memory 방식으로 디바이스와 교신하기 위해 메모리를 확보하고 이 것을 사용한다고 하자. 이 메모리 영역을 메일박스라고 부른다. 이 메일박스를 위해서 일정한 양의 메모리가 필요하다. 메일박스를 만들때 마다 메모리를 할당받는다면 상당한 오버헤드가 생기므로 디바이스 초기화 때에 영역을 확보해 놓고 계속 사용하면 좋을 것이다. 이 때 사용할 수 있는 함수는 kmalloc이다. kmalloc이 한 번에 할당할 수 있는 메모리 양은 16KB로 제한되어 있다.

할당받은 메모리 영역을 초기화 할 때는 memset 함수를 쓸 수 있다. 인자로 주는 값으로 메모리 영역을 채운다. 메모리에서 메모리로 데이타를 복사할 때는 memcpy를 사용할 수 있다. 이 함수는 커널 메모리사이에서 복사할 때만 쓰는 함수이다. 절대로 유저 데이타 영역과 커널 메모리사이의 복사에서 사용해서는 안된다.

개발 초기에는 insmod 인자가 정상적으로 전달되었는지 확인한다든지 어떤 디바이스가 인식되었는지 보여 주는 출력루틴을 가능한 많이 삽입하는 것이 좋다. C 라이브러리 루틴의 printf 는 사용할 수 없지만 커널 프로그래밍을 위해서 printk 가 있다. 몇가지 제한이 있지만 printf 와 거의 유사하다. 정확한 형식은 lib/vsprintf.c에서 볼 수 있다.

4.4.5 디바이스의 I/O 컨트롤

디바이스 드라이버가 제대로 적재 되고 나서 설정값을 바꿀 필요가 있을 때 ioctl 함수를 부른다. C 라이브러리의 ioctl 함수를 이용해서 커널 드라이버의 ioctl 함수에 접근할 수 있다. C 라이브러리의 ioctl 함수 원형은 다음과 같다.

int ioctl(int d, int request,...);

d는 파일기술자이며 /dev/my_device0 에 해당한다. 우선 open 함수로 my_device0를 열어 보고 정상적으로 열리면 ioctl을 실행할 수 있다. ioctl의 두번째 인자에 원하는 명령을 넣는다. 만약 my_device0 의 어떤 값을 바꾸게 하고 싶으면 include/linux/my_device.h 에 아래와 같이 정의하고 유저 프로그램에서 사용할 수 있도록 하면 된다.

#define MY_DEVICE_CHANGE_VALUE _IOWR(MY_DEVICE_MAJOR, 1, int[2])

_IOWR/_IOR 매크로는 include/asm/ioctl.h 에 정의되어 있고 여러가지 비슷한 매크로가 있다. 함수명이 보여주는 대로 세번째 인자에서 지정한 영역을 읽기만 가능/읽고 쓰기 가능한 형태로 커널에 전달하게 된다. 이 매크로는 첫번째 인자와 두번째 인자를 결합해서 커널에 전달한다. 두번째 인자에 들어 갈 수 있는 값의 크기가 얼마나 되는가는 ioctl.h를 조사해 스스로 알아보기 바란다.

ioctl의 세번째 인자는 커널에 전달하는 값이나 값의 배열 또는 커널로 부터 받을 데이타 영역이다. 커널에 값(값배열)을 전달하고 같은 곳에 커널로 부터 값을 받을 수도 있다.

커널의 ioctl은 보내온 명령에 따라 완전히 독립된 작업을 해야 하기 때문에 가장 지저분한 부분이다. 대부분의 드라이버들이 switch 문을 사용해서 명령에 따른 작업을 하는 코드를 작성해 놓았다. 해야 하는 작업에 따라 내용이 다르지만 커널 프로그래밍과 크게 차이가 나는 것은 아니다. 가장 중요한 것은 C 라이브러리 함수인 ioctl에서 보내온 데이타를 주고 받는 방법이다. 커널이 보는 메모리 영역과 사용자 프로그램이 보는 메모리 영역은 완전히 다르기 때문에 memcpy등의 함수를 사용해서는 안된다.

커널에서 유저 프로그램에서 데이타를 읽어 오거나 쓰기 위해서는 우선 읽어 올 수 있는지 확인해야 한다. verify_area(VERIFY_READ/VERIFY_WRITE ..) 함수를 사용해서 읽거나 쓰는데 문제가 없으면 memcpy_fromfs/memcpy_tofs 함수를 사용할 수 있다. 이 함수명은 사용자 데이타 세그먼트와 커널 데이타 세그먼트를 구별하는 인텔 CPU의 레지스터 명이 fs 인데서 유래했다. 2.2 이상에서는 여러 플랫폼에서 일반 명칭으로 사용하기 위해서 copy_from_user/copy_to_user로 바뀌었다. 2.2에서는 또한 verify_area를 할 필요가 없다.

ioctl 함수는 사용하기에 따라 수많은 일을 할 수 있다. ioctl을 사용하여 할당받은 인터럽트를 바꿀 수도 있고 점유하고 있는 물리 주소도 변경할 수 있다. linux-ggi 프로

젝트 그룹에서는 모든 VGA드라이버를 단일화 시키고 ioctl을 사용해서 각 VGA카드의 특성을 조정해서 사용하자고 제안하고 있다. 이 방법은 리눅스 VGA드라이버 작성을 위한 노력을 대폭 줄일 수 있는 획기적인 방법이다

4.4.6 디바이스 드라이버 입출력 함수

이제 디바이스 드라이버의 핵심인 입출력 루틴에 대해서 알아보자. 이 부분은 하드웨어와 가장 밀접한 부분으로 하드웨어의 기계적인 작동까지도 고려해야 하는 복잡하고 힘든 작업을 필요로 한다. 게다가 책에서만 보던 세마포, 교착상태 스케쥴링등 운영체제에서 가장 중요한 부분을 직접 구현해야 하는 일이기도 하다. 온갖 개발툴이 난무하고 마우스만으로 프로그래밍을 끝낼 수 있는 최상위 응용프로그램 개발 환경이 지배하는 요즈음, 이렇게 원론적이고 근본적인 작업을 할 수 있다는 것은 즐거움이기도 하다. 심각한 소프트웨어 위기가 닥치면 닥칠 수록 컴퓨터를 배우는 사람들, 컴퓨터를 사용하여 무엇인가를 이루어 볼려는 사람들은 가장 근본적인 부분을 다시 들여다 보아야 할 필요가 있다. 특히 컴퓨터를 배우고 있는 학생이라면 반드시 리눅스 커널에 관심을 가져야 하는 이유가 바로 여기에 있는 것이다.

커널이 발전하면서 필요에 따라 변수명이나 함수명들을 바꿀 필요가 생긴다. 유저프로그래밍에서는 라이브러리 함수 인터페이스가 바뀔 수도 있지만 write같은 가장 기본적인 인터페이스는 거의 바뀌지 않는다.

static ssize_t my_device_write(struct file *, const char *, size_t count, loff_t *);

4.4.6.1 MY_DEVICE_OPEN 함수

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

void main(int argc, char*argv[])
{
            int   w_dev;
            char read_value[4096];
            w_dev = open("dev/my_device0", O_WRONLY | O_APPEND);
            read(r_dev, read_value, 1024);
            write(w_dev, read_value, 1024);
 }
```

디바이스 드라이버를 이용하는 응용프로그램

라이브러리의 open("/dev/my_device0",...) 호출이 오면 커널이 최종적으로 부르는 루틴은 my_device_open 함수이다. 여기서는 디바이스 상태를 점검하고 열 수 있는지 확인한 다음 필요한 메모리등을 준비하고 다른 루틴에서 사용하지 못하도록 busy 세팅을 하면 된다.

모듈 방식으로 적재 했을 때를 생각해보자. 만약 open,read/write 루틴을 수행 중인 상태에서 rmmod 를 사용해서 모듈을 삭제하면 디바이스드라이버 루틴을 수행할 수도, 정상적으로 중단할 수도 없게 되어 커널이 정지하게 된다. 그러므로 디바이스드라이버 루틴을 수행 중인 프로세스가 있으면 이를 알려주는 변수를 증가시켜서 모듈 삭제 함수가 참고하게 하면 된다. 이 변수 값을 증가/감소 시키는 매크로는 MOD_INC_USE_COUNT/MOD_DEC_USE_COUNT 이다. my_device_open함수를 수행하면서 열기에 문제가 없다면 이 값을 증가시킨다. 만약 이 값을 증가시킨 후에 open 루틴을 수행하면서 문제가 생겨 에러 리턴할 때에는 반드시 값을 감소시켜야 한다. 증가/감소 짝이 맞지 않으면 드라이버를 삭제할 수 없다. 정상적으로 작동된다면 release 루틴에서 감소시키면 된다. write/read 수행 도중에 에라로 리턴하면 커널이 release를 부르기 때문에 write 루틴에서는 신경쓸 필요는 없다.

커널이 open을 호출 할 때에는 주장치번호를 보고 호출하므로 내부에서 부장치번호를 구해야 한다. 부장치번호는 MINOR(inode->i_rdev) 매크로로 구할 수 있다. 사용자가 open("/dev/my_device255"..) 처럼 호출할 수도 있으므로 이 값이 유효한 값인지 그리고 실제로 사용가능한 값인지 확인해야 한다. 전역변수로 MY_DEVICE_MAX를 선언하거나 my_device 구조체 배열을 선언하고 exist 필드를 정의한 다음 유효한 부장치번호의 exist필드를 세팅하면 된다. drivers/char/lp.c를 보면 lp[0-3]까지를 이런 방식으로 사용하고 있다.

커널은 부장치가 다른 루틴에서 사용중인지 검사하지 않는다. 열기 요청이 오면 무조건 open루틴으로 제어가 이동하기 때문에 다른 루틴에서 사용하지 못하게 하거나 사용이 끝나는 시점이라는 것을 알려 주는 일은 드라이버의 몫이다. 마찬가지로 my_device 구조체 안에 busy 필드를 정의하고 이를 사용하면 된다. 간단히 my_device에 필요한 정보를 담는 구조체를 만들어보자.

my_device는 카드에 CPU가 있고 이 카드가 2개 이상의 부장치번호를 제어한다. 만일 멀티시리얼포트라면 8-64개까지의 시리얼포트를 한 카드가 제어하게 될 것이다. 그리고 각 포트는 포트별로 필요한 정보와 이 포트가 어느 카드에 있는 것인지 알려주는 정보가 있어야 할 것이다. 우선 카드 정의를 한다.

```
struct MY_DEVICE_CARD {
int number; /* 카드번호 */
u_int flag; /* 상태 플래그 */
int base; /* base 주소 */
u_int irq; /* 인터럽트 번호 */
```

```
u_int address; /* memory mapped I/O 주소 */
u_char *mailbox_kernel; /* 커널에서 보낼 메일박스 */
u_char *mailbox_card; /* 카드에서 보내온 메일박스 */
struct semaphore semaphore; /* 주장치의 세마포 */
struct wait_queue *queue; /* 주장치의 대기큐 */
};
struct MY_DEVICE_CARD my_device_card[MY_DEVICE_CARD_MAX];
```

이에 따른 포트의 정의는 다음과 같이 할 수 있다.

```
struct MY_DEVICE_PORT {
int number; /* 포트 번호 */
int card_number; /* 포트가 속한 카드 번호 */
u_int flag; /* 상태 플래그 */
u_char *buffer; /* 버퍼 포인터 */
};
struct MY_DEVICE_PORT my_device_port[MY_DEVICE_PORT_MAX];
```

위의 내용은 여태까지 설명했던 글에서 필요하다고 생각되는 데이터를 모아본 것이다. 우선 card->irq, card->base는 인자로 받을 수 있으므로 초기값을 0로 두어서 카드가 존재하는지 판단할 수 있게 한다. 이 값에 따라 포트가 사용가능인지 정해지므로 초기화시에 카드에 딸린 포트를 검사해서 port->flag의 exist필드를 세팅할 수 있을 것이다. card->address는 메모리멥 방식 I/O에서 사용할 주소값이고 mailbox_* 는 카드와 드라이버가 교신하면서 사용할 메일박스 (Documentations/IO-mapping.txt참조)이다. 이 것은 데이타를 주고 받는 것이 아니고 주장치의 상태를 점검하고 카드가 상태를 보고하는 용도로 사용되는 것이므로 부장치가 아닌 주장치에 있어야 한다. 대기큐도 한 주장치번호에 할당되어서 각 부장치에서 돌아가는 프로세스들이 공유하게 된다. 멀티 시리얼포트나 같은 장치를 여러개 달 수 있는 하드웨어는 부장치번호를 가지고 주장치 번호를 구해 낼 수 있어야 하므로 port->card_number가 반드시 필요하다. 버퍼는 실제 데이터를 보내는 것이므로 부장치들이 하나씩 가지게 된다.

이런 형태가 멀티 시리얼이나 동일 하드웨어를 여러개 붙일 경우에 일반적으로 사용되는 방법이다. 각필드에 접근하는 방법은 다음처럼 하게 된다.

```
struct MY_DEVICE_CARD *card = my_device_card[my_device_port[minor].card_number];
u_char *mailbox = card->mailbox_kernel;
```

이런 방법이 코드를 복잡하게 만들기 때문에 일부 드라이버에서는 매크로 정의로 처리하고 있다.

```
#define MY_CARD(minor) my_device_card[my_device_port[(minor)].card_number]
...
struct MY_DEVICE_CARD *card = MY_CARD(minor);
...
```

C 프로그래밍의 기초적인 이야기를 여기서 하는 이유는 커널에서 코드의 가독성과 효율을 높이기 위해서 사용하는 매크로들이 오히려 가독성을 떨어뜨리고 마치 소스가 암호와 같은 모양을 띄게 하기 때문에, 이런 매크로들이 나오면 선언부를 찾아 보고 그 내용을 익히라는 말을 하기 위해서이다. 리눅스 커널에서 사용되는 대문자 변수들은 거의가 상수값이 아니라 복잡한 매크로인 경우가 대부분이기 때문이다.

my_device가 취할 수 있는 상태값은 여러가지가 있을 수 있다. 간단하게 몇가지만 생각해보자.

```
#define MY_DEVICE_PORT_IDLE 0x00001
#define MY_DEVICE_PORT_BUSY 0x00002
#define MY_DEVICE_PORT_ERROR 0x00003
#define MY_DEVICE_STATUS_EXIST 0x00010
#define MY_DEVICE_STATUS_BUSY 0x00020
#define MY_DEVICE_STATUS_ERROR 0x00040
```

위에서 사용한 두가지 방법은 약간 다르다. 우선 첫번째로 값을 10진수로 증가시키는 방법은 하드웨어가 상태를 보내올 때 사용할 수 있다. my_device카드가 보내오는 값이 10진수 값이라는 말이다. 이때 상태를 판단하기 위해서는 다음과 같이 해야 할 것이다.

```
if(status==MY_DEVICE_PORT_IDLE || status==MY_DEVICE_PORT_BUSY)
...
```

이렇게 10진수를 사용하는 방법은 드라이버가 받는 정보가 10진수 일 때를 제외하고는 사용하지 않는 것이 좋다. 왜냐하면 각비트를 상태값으로 이용하는 것에 비해서 단점이 많기 때문이다. 우선 상태정보를 분류할 수 없다. 비트이용법은 처음 4비트는 드라이버 자체 에러
다음 4비트는 하드웨어 에러등으로 분류해서 쉽게 에러 종류를 구해낼 수 있다.

```
if((status & 0xf0) != 0) // hardware error
else if((status & 0x0f) != 0) // driver error
```

10진수 사용법은 상태정보의 추가/변경이 힘들다. idle(1)에서 busy(2)로 바꾸는 것은 문제가 없지만 card_error, card_wait등의 드라이버에서 상태를 유지해야 하는 정보를

위해서는 복잡한 방법을 사용하던지 card_status 필드를 또하나 추가해서 사용해야 한다. 리눅스에서 unsigned int는 32비트이므로 동시에 32개의 상태를 설정할 수 있다. 비트 이용법은 32개의 상태까지는 비트열 정의만으로 간단히 해결할 수 있다. 속도와 효율성 그리고 메모리 사용량의 최적화를 요구하는 커널 프로그래밍에서 필요할 때마다 자원을 추가해 나가는 것은 좋은 방법이 아니다.

10진수 사용법은 속도가 느리다. 여러 상태정보를 얻기 위해서 사용해야 할 연산이 많아지게 되므로 비트이용법보다 느리고 코드가 복잡해진다. 한 하드웨어가 이 이상의 상태정보가 필요한 경우가 별로 없으므로 my_device_card에서 처럼 상태정보는 flag 필드의 비트위치로 빠르고 간단하게 판단할 수 있게 하는 것이 좋다.

드라이버에서 리턴하는 에러값은 include/asm/errno.h 에 정의된 값에 (-)부호를 붙여서 돌려 주면 된다. 치명적인 에러에는 EIO, ENOMEM 등이 있을 수 있고 커널에서 처리를 하고 드라이버 열기를 재시도 할 수 있게 하는 EAGAIN 등이 있을 수 있다. 에라 종류와 리턴값은 여러 드라이버 소스를 보고 파악하기 바란다.

my_device_open에서 해야하는 일을 요약하자면 다른 프로세스가 부장치번호를 사용중인지 검사하고 이 번호가 유효한 것인지 체크한 다음 MOD_INC_USE_COUNT를 사용해서 모듈 삭제시에 참고하게 한 후에 필요한 자원을 할당받고 다른 프로세스가 이 장치를 사용할 수 없도록 busy세팅을 하면 된다.

4.4.6.2 MY_DEVICE_READ/WRITE 함수

열기 함수를 수행했다면 커널에서 읽기/쓰기 함수를 호출할 수 있다. 열기/쓰기 함수의 원형은 2.2.x에서 다음과 같다.

```
static ssize_t my_device_read(struct file * file, char * buf, size_t count, loff_t *ppos);
static ssize_t my_device_write(struct file * file, const char * buf, size_t
count, loff_t *ppos);
```

쓰기함수에서는 쓸 데이타가 있는 주소 buf가 전달되어 오고 읽기 함수에서는 읽어갈 데이터 영역인 buf주소가 전달되어 온다. 쓰기함수의 데이타는 커널에서 처리가 되어 커널 메모리 영역으로 생각하고 사용하면 된다. ioclt 함수 사용법에서 말했듯이 memcpy등을 사용할 수 없고 copy_to_user를 사용해야 한다. 참고로 한번에 buf로 넘어오는 데이타의 최대값은 4096바이트이다.

read/write에서 물론 부장치번호를 체크해야 한다. 사용할 수 없는 부장치에 접근 요구가 오면 에라값을 가지고 리턴해야 한다. inode가 넘어오지 않으므로 file 구조체에서 찾아야 한다. 어떻게 찾는지는 여러 소스를 찾아보기 바란다. count는 읽거나 쓸

데이타의 양을 나타내고 ppos는 그 지점의 주소값이다. ppos는 블럭 디바이스에 주로 쓰게 되며 문자 디바이스에서는 거의 사용되지 않는다.

쓸 데이타가 있고 쓸려는 부장치가 유효한 것이라면 이제 하드웨어를 검사한 후에 데이타를 쓰면 된다. 하드웨어 검사는 드라이버에서 쓸 때마다 할 수도 있고 하드웨어가 상태변화를 보고할 수도 있다. 여러가지 방법은 하드웨어 특성에 따라 다르고 각 디바이스가 사용하는 방법이 다르므로 만들려는 드라이버에 맞게 작성해야 한다. 상태가 정상이라면 문자디바이스일 때 한 번에 한 바이트씩을 쓰고 결과를 보면서 그 다음 바이트를 보내거나(lp.c), 하드웨어가 문자디바이스를 다루지만 내부적으로 대량의 데이타를 받을 수 있는 장치라면 일정 단위의 데이타를 보내면 된다. 블럭디바이스라면 자체적으로 데이타를 보낼 수도 있고 block_write라는 블럭디바이스 공용 루틴을 사용할 수도 있다.

쓰거나 읽을 데이타 양을 모두 처리하는 동안 문제가 생기지 않았으면 쓰거나 읽은 데이타 양을 인수로 하여 리턴하면 된다. 시그널을 받았으면 -EINTR 값을 사용하고 그 외에 에라가 발생하였으나 커널 쪽에서 재시도를 하기를 원한다면 그 때까지 읽은/쓴 데이타 양을 인수로 하여 리턴한다. 각종 에라가 발생했고 커널 쪽에서 읽기/쓰기를 재 시도 할 수 없는 에라이면 커널에서 my_device_release를 스스로 호출하게 되어 있으므로 open/read,write/release 전체에 걸친 변수값등을 read/write 루틴에서 신경쓸 필요는 없다. read/write 루틴 안에서 사용한 값과 이 루틴이 정상적으로 끝났을 때 변경되는 전역 변수에 대해서만 신경을 쓰면 된다.

3.4.6.3 MY_DEVICE_RELEASE 함수

release 함수에서는 open에서 했던 일을 역으로 하면 된다. 우선 메모리영역을 할당받아서 사용했다면 반납한다. 부장치에 대해서 busy세팅을 했다면 이를 해제한다. 모듈을 사용했다면 MOD_DEC_USE_COUNT를 실행해서 모듈 사용값을 감소시킨다. 이 매크로는 자체적으로 모듈로 적재되었는지 커널에 같이 컴파일되었는지 구별하기 때문에 #ifdef MODULE 을 사용하지 않아도 된다. 인터럽트를 open에서 할당받았다면 여기서 해제한다. 인터럽트를 공유하는 장치들은 이렇게 사용하고 있다.

2.0.x에서는 리턴값이 없었으나 2.2.x에서는 리턴값이 생겼다. 정상적으로 진행되었다면 0을 리턴하면 되고 그렇지 않다면 에러값을 가져야 한다. 어떤 값을 가질 수 있는지는 다른 드라이버를 참고하기 바란다. 이 루틴에서 가장 중요하게 고려해야 할 것은 모든 작업이 또다른 open이 가능하게 만들어야 한다는 것이다. 새로운 open함수가 실행되어서 사용할 변수나 값들이 최초로 open이 실행되는 경우와 동일하게 되어 있어야 한다. 그러므로 open, read, write 함수에서 사용된 변수들을 철저히 조사해서 반드시 제자리로 돌려 놓는데 신경을 써야 할 것이다. 이 일을 제대로 하지 못하면 사용할 수 있는 디바이스 인데도 불구하고 한 번만 사용하고 나면 더 이상 쓸 수 없는 상태로 된다. 이런 에러는 open을 실행 할 때 나타나기 때문에 release루틴에서 잘못했다고 생각하지 않고 open 쪽을 먼저 생각하여 시간을 낭비하게 되므로 많은 디버깅

노력이 들어간다.

4.4.7 MY_DEVICE_INTERRUPT 함수

인터럽트는 디바이스가 인터럽트를 사용할 수 있는 장치를 제어할 때 사용할 수 있다. 인터럽트를 사용하면 대기시간 동안 커널에서 다른 작업을 할 수 있으니까 효율이 높다. 그리고 디바이스가 작업을 끝내는 즉시 드라이버의 인터럽트 루틴을 수행해서 대기 프로세스를 활성화 해주기 때문에 대기시간을 최소화 할 수 있다. 드라이버에서 디바이스의 상태를 체크하는 폴링(polling)방식은 인터럽트 방식의 장점을 반대로 생각하면 된다. 대기 프로세스가 작업이 끝나지 않았는데도 깨어나서 디바이스를 체크해야 하는 오버헤드가 있으며 대기 중인 프로세스는 디바이스가 작업을 끝냈는데도 지정된 대기시간을 모두 소모해야 다음 작업을 진행할 수 있다. 즉 폴링방식은 효율이 나쁘고 대기시간이 길다. 폴링방식으로 한 바이트씩 데이타를 쓰는 프린트 드라이버 루틴이 실행 중 일 때 x-window에서 마우스 움직임도 느려지는 것을 볼 수 있다.

이런 단점에도 불구하고 폴링방식은 코드가 이해하기 쉽고 간단하기 때문에 드라이버 제작자가 관심을 가진다. 반대로 인터럽트 방식은 절차적 프로그래밍에 익숙한 응용 프로그램을 만들던 개발자들이 가장 어려워하는 방식이다. 제어가 한 군데에서 이루어지지 않고 전역 변수가 동시에 여러 군데에서 참조가 되면서 개발자들을 혼란스럽게 만들기 때문이다. 폴링 방식은 간단히 이해할 수 있기 때문에 언급을 하지 않겠다. 커널해커가이드의 지원함수 부분을 보면 자세하게 나와있으니까 이를 참조하기 바란다.

인터럽트 방식을 사용하기 위해서 먼저 생각해야 할 것은 인터럽트 함수의 실행시간을 가능한한 짧게 만들어야 한다는 것이다. 같은 작업을 가장 신속하게 끝낼 수 있는 방법을 생각해야 한다. "빠른"인터럽트는 자기보다 우선순위가 낮거나 같은 인터럽트를 금지하기 때문에 인터럽트 처리 루틴이 비효율적이고 시간이 많이 걸리면 시스템이 느려지고 추가 인터럽트가 사라지는 등 신뢰성도 나빠지기 때문이다. 여러 소스를 찾아보다 보면 인터럽트 루틴의 실행시간을 줄이기 위해서 고민한 개발자들의 글을 찾아볼 수 있다.

인터럽트 루틴에서는 공통루틴이 있다고 이들을 모아서 실행한 후에 다른 부분만 뒤에 나누어서 실행하게 하는 등의 "코드를 보기 좋게" 하는 작업을 해서는 안된다. 코드가 지저분하게 되더라도 여러 조건에 따라 다른 작업이 있다면 이들 각각의 작업에 가장 효율적인 형태로 코딩을 해야 한다. 인터럽트 루틴은 아름다움 보다는 효율이 우선되기 때문이다. 일부 드라이버는 이를 위해서 goto문장도 아무 거리낌 없이 사용하고 있다.

인터럽트루틴에서 우선 해야 할 것은 넘어온 인터럽트가 이 디바이스에서 처리할 것인가를 조사하는 것이다. my_device_card의 지정된 인터럽트와 이 인터럽트가 일치하는지 검사한다. 물론 이 인터럽트가 아닐 경우는 매우 희박하다. 인터럽트를 비교하는 중요한 목적은 my_device_card가 여러 디바이스를 동시에 지원할 때에 이 카드 중에

서 어느 것으로 부터 인터럽트가 전달된 것인지 확인하기 위해서이다.

인터럽트에 따라 주장치번호를 구했으면 주장치번호를 대상으로 작업을 처리해야 한다. 인터럽트 루틴에서는 부장치번호를 알아내기 어렵다. 주장치에 따른 부장치가 여러개 있다면 이들 중에서 한개 이상의 부장치가 디바이스에 대해서 작업을 하고 디바이스의 응답을 기다리고 있을 것이다. 주장치 구조체에 대기 중인 부장치의 순서를 정해 넣거나 디바이스가 스스로 인터럽트를 걸면서 부장치번호를 알려 줄 수도 있지만 이를 처리하는 것은 많은 시간이 걸린다. 주장치의 wait_queue에 대기 중인 부장치들은 자동적으로 순서를 부여받으므로 모든 처리를 주장치의 리소스에 대해서 하고 wait_queue를 활성화 시켜서 깨어난 부장치의 루틴에서 이 리소스를 처리하게 하는 것이 인터럽트 루틴에서 소요되는 시간을 줄일 수 있는 방법이다.

인터럽트 루틴에서 메모리를 할당받는 kmalloc등을 사용할 수 없다. kmalloc은 할당에 실패했을 때 스스로 필요한 시간 만큼 기다리고 재시도를 하기 때문에 시스템을 정지시킬 가능성이 있기 때문이다. 마찬가지로 인터럽트 중에 schedule 같은 프로세스 블럭 함수도 사용할 수 없다. 이 것은 인터럽트라는 특성상 블럭이 가능하지 않음을 생각하면 당연한 것이다. 또한 save_flags-cli-sti-restore_flags 루틴도 사용할 수 없다. 인터럽트 루틴은 인터럽트 불가능 상태에서 수행되기 때문에 어떤한 경우에도 sti를 실행해서는 안된다. 만인 인터럽트 수행 도중에 sti를 실행해서 인터럽트를 가능하게 만들었다면 같은 인터럽트가 또다시 걸리거나 다른 인터럽트가 걸릴 수 있다. 인터럽트는 한 번에 수행되고 그 동안 방해받아서는 안되는 루틴이기 때문에 이 경우 많은 문제가 발생할 수 있다. 이 것을 생각하고 있다고 하더라도 인터럽트 함수도 부를 수 있고 다른 루틴에서도 사용할 수 있는 작은 함수를 만들어 쓰다가 필요에 따라 이 함수를 고치는 경우가 있다. 이 때 무심코 다른 루틴에서의 필요에 의해서 이 함수에 save_flags-sti-restore_flags를 쓰게 될 수 도 있다. 그러므로 코드 유지보수를 위해서 인터럽트 함수는 가능한한 필요한 작업을 다른 함수를 부르지 않고 내부적으로 모든 일을 처리하는 것이 바람직하다. C의 함수 호출은 오버헤드가 많으므로 시간적으로도 장점이 있다.

인터럽트는 리턴값이 없다. 인터럽트는 필요한 일을 수행하고 리턴할 뿐이다. 그 결과는 인터럽트를 호출한 프로세스에 필요한 것이 아니라 인터럽트를 기다리던 루틴에서 필요한 것이다. 이 값은 인터럽트 수행과정에서 세팅되고 깨어난 프로세스가 그 값을 스스로 찾아서 이용해야 한다.

4.4.8 WRITE,READ/INTERRUPT 함수의 수행 과정

쓰기(읽기)루틴에서 데이타를 디바이스에 쓰거나 디바이스의 상태를 알아보기 위해서 인터럽트를 사용한다. 이 때 주장치에 부속된 부장치가 여러개라면 이들은 경쟁적으로 자원을 할당받으려고 할 것이다. 같은 자원에 대해서 동시에 여러 프로세스가 접근한다면 충돌이 일어난다. 운영체제의 가장 중요한 주제인 상호배제 문제를 리눅스에서는 어떻게 해결했을까. 결론적으로 말하면 멀티프로세서 환경에서도 잘 작동하는 세마포

함수가 준비되어 있다. 세마포 함수는 독립적으로 개발자가 개발하여 커널에 포함시키고 있으며 테스트 프로그램도 구할 수 있다(include/asm/semaphore.h 참조). 여러 부장치가 경쟁적으로 한 자원을 접근하는 루틴이 있으면 이 루틴의 앞뒤로 세마포를 설정한다.

```
down(&card->semaphore);
중요루틴 실행
up(&card->semaphore);
```

물론 이렇게 사용하기 위해서는 my_device_card 구조체에 세마포 필드를 설정해야 한다. 그리고 세마포의 초기값을 반드시 MUTEX로 지정해 놓아야 한다.

```
card->semaphore=MUTEX;
```

인터럽트 함수와 마찬가지로 세마포가 적용되는 루틴은 가능한한 빨리 끝날 수 있도록 설정해야 한다. 내부루틴이 많은 시간을 소요한다면 down부분에 다른 프로세스가 계속 멈추어 있게 된다. 편리한 루틴이 있다고 내부를 들여다 볼 생각을 하지 않고 그냥 사용하는 데 만족하면 안된다. 세마포 내부 처리 과정을 보면 프로세스대기, 타스크에 대한 처리, 시그널과의 관계, 여러 종류의 CPU에 대한 어셈블러 수준의 비교까지 알아볼 수 있다. 리눅스 세마포 처리 루틴은 유닉스 프로세스에 대한 개념과 그 실제 적용을 파악할 수 있는 좋은 교재라고 생각된다. 이글을 보는 독자라면 arch/*/lib/semaphore.S, kernel/sched.c등을 조사해 보기 바란다.

세마포가 보증하므로 이제 충돌없이 인터럽트를 사용할 수 있다. 부장치마다 카드에 쓸 데이타를 위한 버퍼를 할당받고 카드상태등의 드라이버에 필요한 정보를 모아 놓는 것은 자원의 낭비일 뿐이다. 여러 부장치가 한 디바이스를 접근하고 세마포가 동시에 접근하는 것을 막아 주므로 인터럽트를 주고 받는 디바이스는 최소한의 데이타만을 유지하고 나머지 데이타는 주장치에 할당하는 것이 좋다.

```
my_device_card에 할당하는 리소스
struct MY_DEVICE_CARD {
...
u_int flag; /* 상태 플래그 */
u_char *mailbox_kernel; /* 커널에서 보낼 메일박스 */
u_char *mailbox_card; /* 카드에서 보내온 메일박스 */
struct semaphore semaphore; /* 주장치의 세마포 */
struct wait_queue *queue; /* 주장치의 대기큐 */
};
```

주장치인 my_device_card에 디바이스에 주고받을 데이타와 디바이스 상태정보를 보관할 필드를 정의했다. 인터럽트를 사용해서 데이타를 디바이스로 보내기 위해서 write

루틴은 미리 할당된 mailbox_kernel에 데이타를 보내고 디바이스에 인터럽트를 건다. 돌아온 인터럽트와는 달리 디바이스에 인터럽트를 거는 것은 write루틴의 몫이다. my_device에 인터럽트를 거는 것이 다음과 같다고 하자

```
memcpy_toio(card->address + DATA_ADDR, card->mailbox_kernel, length);
outb(0x01, card->base+INTERRUPT_ADDR);
current->timeout = jiffies + HZ * 3;
interruptible_sleep_on(&card->queue);
```

base 주소에서 INTERRUPT_ADDR 바이트를 더한 주소에 1을 쓰는 것이 이 디바이스에 인터럽트를 거는 행위이다. 이 전에 활성화된 디바이스가 사용할 데이타를 전송한다. memcpy_toio가 그 있을 하게 된다. 보낼 주소, 보낼 데이터가 있는 주소, 길이에 따라 데이타가 전송된다. 마지막으로 디바이스가 활성화 되어 보낸 데이타에 대한 처리를 끝내고 커널에 인터럽트를 걸어 이 루틴이 활성화 될 때까지 기다리기 위해서 잠든다(interruptible_sleep_on) 이 루틴은 깨워줄 대상이 누구를 깨워야 하는지 알 수 있도록 card->queue에 자신을 등록한다.

인터럽트가 걸린 디바이스는 일반적으로 수십밀리초에서 수백밀리초 안에 처리를 끝내고 인터럽트를 돌려 주기 때문에 현 프로세스의 타임아웃값을 크게 잡을 필요는 없다. 물론 그 값은 하드웨어 디바이스의 특성을 파악한 후에 적절한 값을 정해주어야 한다. 인터럽트가 정상적으로 돌아오면 바로 이 프로세스가 활성화 되므로 대기값을 많이 잡는 다고 오버헤드가 증가하는 것은 아니다. 여기에서는 my_device에 대해서만 얘기했지만 하드웨어에 따라 다양한 인터럽트 걸기가 존재할 수 있다. 마찬가지로 여러 드라이버를 찾아보고 필요한 방법을 연구하기 바란다.

대기 후에 활성화된 프로세스는 프로세스가 깨어난 상태에 대한 점검이 필요하다. 시그널에 의해서 깨어난 프로세스는 작업을 중단하고 자신을 부른 상위 루틴으로 복귀해야 한다. 물론 세마포에 대한 정리작업 같은 중요한 작업은 반드시 수행 해야 그 다음 프로세스가 영향을 받지 않는다. 만약 타임아웃값이 지나서 활성화 되었고 시그널이 없다면 디바이스에 이상이 있는 것인지 타임아웃값이 충분하지 않았는지 확인해야 한다. 타임아웃값이 적은 것이라면 늘여 주면 되고 디바이스가 인터럽트를 돌려 주지 못한 것이라면 이에 따른 처리를 해야 할 것이다.

필자의 경험으로는 디바이스가 인터럽트를 돌려 주지 못하는 경우가 생긴 적이 있었다. 대상 디바이스가 비정상적인 작동을 할 때 그 문제를 하드웨어적인 문제라고 생각하고 해결 방법이 없다고 생각했지만 나중에 알아낸 결과 인터럽트를 주고 받을 때의 규약을 정확히 지키지 못했기 때문이었다. 다른 운영체제에서 정상적으로 사용하던 하드웨어는 그 규약만 정확히 지킨 다면 리눅스에서 아무 문제 없이 사용할 수 있다. 커널 프로그래밍을 할 때 종종 운영체제가 달라서 어쩔 수 없다는 말을 듣고는 하는데 이 것은 리눅스의 한계를 말하는 것이 아니고 하드웨어 제작사가 정말 필요한 정보를 공개하지 않고 있음을 말하는것이다. 제대로 만들어진 하드웨어 사양을 가지고 프로그

래밍을 한다면 절대로 실패할 수 없으며 리눅스에서 훌륭히 동작시킬 수 있다.

프로세스가 활성화 된 후에 디바이스가 인터럽트를 제대로 돌려 주어서 드라이버의 인터럽트 루틴이 수행된 결과로 활성화 된 것이라면 디바이스에서 보낸 데이타를 처리해야 한다. 앞에서 말했듯이 인터럽트 루틴은 최소한의 작업만을 하기 때문에 모든 처리는 write에서 하게 해야 한다.

여기에 대응되는 인터럽트 루틴 쪽의 코드는 다음과 같다.

```
memcpy_fromio(card->mailbox_card, buf, buf_size);
card->flag |= MY_DEVICE_ANSWERED;
if (waitqueue_active (&card->queue))
wake_up_interruptible(&card->queue);
return;
```

인터럽트 루틴에서 하는 일은 정당한 인터럽트인지 확인한 후에 디바이스에 있는 데이타를 복사하고 디바이스가 정상적으로 응답했음을 알려 준 후에 대기 중인 프로세스가 있으면 wake_up_interruptible 함수가 이 프로세스를 깨운다.

2.0버전에서는 큐에 대기 중인 프로세스가 있는지 확인하는 함수가 없었지만 2.2에서는 waitqueue_active라는 확인 함수가 생겼다. card->flag 를 세팅하는 이유는 대기 프로세스가 제대로 된 응답을 디바이스로 부터 받았음을 확인 시키기 위한 것이다.

에러 코드에 대해서 드라이버를 작성하면서 여러 에러가 생길 수 있다. 각각의 에라에 대해서 리턴값을 가져야 한다. 드라이버 루틴에서 많이 쓰는 에라에 대해서 간단하게 설명을 하겠다. 여기서 보인 에라를 리턴할 때는 (-)값을 취한다.

완전한 에러 값은 include/asm/errno.h 에 있다.

EINTR : 프로세스 수행 도중에 커널로 부터 시그널을 받았을 때 사용자가 프로세스를 중단했거나 커널에서 강제로 중단 시킬 경우에 사용된다. 해당 프로세스는 신속하게 작업을 끝내야 한다.

ENOMEM : 커널 메모리 영역에 여유가 없을 때 kmalloc 함수를 호출한 후에 에러가 났을 때 리턴값으로 쓴다.

ENXIO : 잘못된 부장치를 호출했거나 사용할 수 없는 주소가 참조 되었을 때
ENODEV : 주장치에 딸린 부장치가 범위를 넘었을 때

EAGAIN : 일시적으로 드라이버에서 디바이스를 사용할 수 없을 때 다른 루틴에서 이 디바이스를 사용 중일 때 잠시 기다린 후에 다시 시도할 수 있게 한다.

EBUSY : 이미 다른 루틴에서 디바이스를 사용 중일 때
ENOSYS : ioctl등의 루틴에서 없는 함수를 부를 때

EIO : 장치를 등록할 수 없거나 디바이스가 정상작동을 하지 않고 있을 때

이 에러는 위의 경우에 해당하지 않는 포괄적인 상황에서 사용된다

. 프로그래밍시 주의점과 기타 정보

리눅스 디바이스 드라이버를 만들기 위해서 필요한 최소한의 참고 사항은 모두 설명했다. 물론 이 것으로는 아무 것도 할 수 없을 것이다. 이 글에서 설명한 것을 참고로 반드시 만들고자 하는 디바이스와 관련된 드라이버 파일을 조사해야 한다. 여전히 드라이버 파일의 내용이 어렵게 느껴지겠지만 이 글에서 언급한 것들이 도움이 될 것이다. 이제 드라이버를 작성할 때 전체적으로 관심을 가져야 할 사항에 대한 설명을 하도록 하자. 그 외에 왕성하게 개발되고 있는 리눅스 커널 관련 프로젝트들에 대해서 언급하고 마지막으로 꼭 찾아 보아야 할 정보에 대해서 알아보자.

4.5 절차적 프로그래밍과 커널 프로그래밍

프로그래밍 언어 C는 코드에 따라 순차적으로 진행되는 프로그램을 만든다. 멀티스레드 개념도 동기화등의 작업을 프로그래머가 제어하게 되므로 순차적이라는 개념을 벗어나지는 않는다. X-window또는 win32에서 이벤트 중심의 프로그래밍을 하게 되는데 이 것도 발생할 수 있는 사건에 대해 프로그래머가 모두 파악하고 처리를 하게 된다. 응용프로그램이 실행된 이후에 발생할 수 있는 외부 사건은 시스템에라 정도이다. 즉 프로그래머가 신경쓸 부분은 응용프로그램의 내부 로직이며 외부 요인에 신경쓸 필요없이 시스템 전체를 응용프로그램이 제어하고 있다고 생각하면서 프로그램밍을 하면 된다.

커널 프로그래밍을 할 때는 프로그래머가 신경쓸 부분이 크게 증가한다. 증가하는 부분은 주로 디바이스 드라이버의 외부 요인에 대한 것이다. 커널은 시작도 끝도 없는 제어구조로 되어있다. 커널에서 주로 하는 작업은 자원테이블의 유지와 갱신이다. 예를 들어 할당된 메모리와 자유메모리에 대한 자원테이블을 유지하기 위해 메모리를 회수하거나 할당했을 때 이 변경사항을 갱신하고 메모리 회수에 필요한 추가작업을 한다. 이런 자원에 대한 요구는 언제 어디서든지 일어날 수 있고 자원 해제 요구 또한 마찬가지다. 커널은 끊임없이 이런 요구를 처리하는데 주로 시간을 보낸다. 동시에 일어나는 수많은 요구에 대한 조정은 매우 힘든 일이기 때문에 가능한 최소한의 작업만을 하고 그외 대부분의 작업은 디바이스 드라이버가 처리하도록 했다. 한 디바이스의 부장치를 억세스하는 프로세스가 활동 중일 때에도 커널은 이를 조사하지 않고 같은 부장치에 대한 억세스 요청이 있을 때 제어를 넘겨 준다. 이를 허용하거나 막는 것은 디바이스 드라이버의 몫이다.

그외에도 드라이버를 만들 때 신경을 써야할 많은 부분이 있다. 커널의 멀티프로세스, 시분할 방식이라는 특성과 비절차적인 외부 요인이 겹쳐서 프로그래밍을 복잡하게 한다.

4.6 경쟁의 제거

디바이스를 open한 프로세스가 가장 먼저 해야 할 것은 동일한 부장치에 대해서 또다른 프로세스가 접근하는 것을 막아야 하는 것이다. 이 것은 부장치의 플래그를 busy로 세팅하는 것으로 쉽게 해결할 수 있다. open 함수에서 여러 조건을 검사한 후에 busy세팅을 하면 된다. 이 디바이스를 두번째로 open한 프로세스가 busy세팅이 되었는지 검사한 후에 busy라면 EBUSY 값을 가지고 리턴하게 된다. 플래그 값이 busy일 때 즉시 EBUSY 값으로 리턴하지 않고 앞서 이 디바이스를 연 프로세스의 상태에 따라(앞의 프로세스가 release 루틴을 수행 중이라면) 커널에서 다시 open을 시도하도록 EAGAIN 값으로 리턴하게 할 수도 있다. 시리얼 포트 제어 디바이스 들이 이 방법을 사용하고 있다.

이 것은 한 개의 부장치에 대한 프로세스의 접근을 막기 위해 고려해야할 점이다. 시스템 시간을 할당받은 프로세스가 어떤 루틴을 수행 하는 도중에 커널로부터 간섭을 받아서는 안되는 경우가 있다. 중요한 전역변수를 바꾸고 중단 없이 필요한 작업을 해야 하거나, 중요 루틴 수행 도중에 제어가 넘어가게 되어서 다른 프로세스가 이 프로세스가 변경한 자원을 또 다시 바꾸지 못하게 해야 할 때 등이다.

이렇게 커널 프로세스의 모든 경쟁을 막고 완전히 프로세스 수행을 독점해야 할 때에는 다음과 같이 해야 한다.

```
...
u_long flags;
save_flags(flags);
cli();
change_variable();
do_critical_job();
recover_variable();
restore_flags(flags);
...
```

cli가 수행되는 순간 restore_flags함수의 수행이 끝날 때까지는 커널의 블럭명령이나 인터럽트가 무시되고 오로지 이 프로세스만이 시스템 시간을 사용하게 된다. 커널 프로세스는 제한된 시스템 시간을 받고 작업을 수행하기 때문에 항상 변수의 변경과 같은 부분에서는 연속된 작업이 다른 프로세스의 방해를 받아도 되는 것인지 확인해야 한다.

save_flags-cli-restore_flags 의 사용은 프로세스 자원할당을 방해하여 커널의 성능을 저하시킨다. 위와 같은 불특정 다수 프로세스와의 경쟁이 아니고 일정 자원을 공유하는 프로세스의 간섭을 막기 위해서라면 세마포를 사용하는 것이 좋다. 주장치에 동일한 부장치가 여러개 있을 때 이들 부장치를 접근하는 프로세스들은 주장치에 대해 서로 경쟁을 한다..

```
struct MY_DEVICE_CARD {
...
u_char *mailbox_kernel; /* 주장치의 메일박스*/
...
}
my_device_write(..) {
...
get_mailbox_and_write();
...
}
```

my_device에는 한개의 주장치에 대해서 4개의 부장치가 있다. my_device_write 함수는 각 부장치(my_device[0-3])에 동시에 접근한 4개의 프로세스가 같이 실행하고 있다. get_mailbox_and_write 함수에 대한 접근과 경쟁이 존재하게 되는 것이다. 이 때에는 경쟁하고 있는 프로세스가 명확하고 한 프로세스가 독점적으로 실행되어야 할 코드가 복잡하고 길기 때문에 save_flags-cli-restore_flags를 사용할 수 없다. 세마포를 적용하면 다음과 같게 된다.

```
struct MY_DEVICE_CARD {
...
u_char *mailbox_kernel;     /* 주장치의 메일박스*/
struct semaphore semaphore; /* 주장치의 세마포 */
...
}
my_device_write(..) {
...
down(&card->semaphore);
get_mailbox_and_write();
up(&card->semaphore);
...
}
```

인터럽트 루틴은 가장 효율적으로 실행되어야 한다고 했다. my_device에서는 하드웨어가 전달할 4096바이트의 데이타 영역이 필요하다고 하자. 인터럽트가 실행될 때마

다 이 영역을 할당받기 위해서 다음과 같이 사용했다.

```
my_device_interrupt(...) {
u_char data[4096];
...
}
```

그런데 이렇게 사용한다면 실행시마다 4096바이트의 영역을 할당받기 위해서 오버헤드가 있을 수 있다. 그래서 효율을 높이기 위해서 다음과 같이 변경하였다.

```
my_device_interrupt(...) {
static u_char data[4096];
...
}
```

이렇게 하면 데이타 배열이 컴파일 시에 지정이 되어서 속도가 개선될 수 있을 것이다.

static 을 사용할 수 있는 이유는 인터럽트를 주고 받는 루틴이 세마포에 의해서 보호받기 때문이다. 즉 일정 시점에 인터럽트를 걸 수 있는 프로세스가 반드시 하나임이 보장되고 지금 처리되고 있는 인터럽트가 보내 주는 데이타는 대기 중인 프로세스가 요청한 데이타이다. 그리고 인터럽트를 거는 루틴이 여기저기에 있는 것이 아니고 세마포에 의해 보호 받는 단 한개의 루틴에서만 존재한다. 여기에 전혀 논리적인 문제가 없다. 실제로 많은 디바이스들이 이렇게 사용하고 있다. 커널 프로그랭밍에서 static 변수를 사용하는 것은 대단히 위험한 일이다. 드라이버를 요청하는 프로세스는 동시다발적으로 자원을 요구한다.

```
my_device_interrupt(...) {
static u_char data[4096];
...
memcp_fromio(card->data, data, 4096);
wake_up_interruptible(&card->queue);
...
}
my_device_write(..) {
...
down(&card->semaphore);
...
interrupt_to_my_device();
interruptible_sleep_on(&card->queue);
up(&card->semaphore);
```

...
}

my_device의 인터럽트를 주고 받는 루틴은 세마포에 의해서 보호받고 있다. 일정 시점에 인터럽트를 걸 수 있는 프로세스가 반드시 하나임이 보장되고 지금 처리되고 있는 인터럽트가 보내 주는 데이타는 대기 중인 프로세스가 요청한 데이타이다. 그리고 인터럽트를 거는 루틴이 여기저기에 있는 것이 아니고 세마포에 의해 보호 받는 단 한 개의 루틴에서만 존재한다. 그렇지만 이렇게 사용하면 문제가 발생한다. 왜냐하면 my_device라는 여러 개의 주장치를 붙일 수 있기 때문이다. my_device[0-3]은 첫번째 주장치에 my_device[4-7]은 두번째 주장치에 연결되어 있다. 인터럽트를 거는 루틴이 세마포에 의해서 보호받고 있지만 이 것은 첫번째 주장치에만 해당되는 사항이다. 두번째 주장치에 연결된 부장치들은 자기들끼리 서로 경쟁한다. 때문에 my_device_interrupt함수를 호출하게 되는 두개 이상의 프로세스가 있을 수 있다. 상황을 설명하면 다음과 같다.

a.process 인터럽트를 걸고 대기모드로 감
b.process 인터럽트를 걸고 대기모드로 감
a.interrupt가 활성화. a.process가 원하는 데이타를 첫번째 my_device에서 static data 영역에 복사하고 리턴.
b.interrupt가 활성화. b.process가 원하는 데이타를 두번째 my_device에서 static data 영역에 복사하고 리턴
a.process 활성화 된 후에 잘못된 데이타가 왔음을 알고 에라로 처리하면서 static data 영역의 내용을 제거.
b.process 활성화 된 후에 데이타가 없으므로 에러로 리턴.

커널에서 함수를 호출할 때 지역변수는 새로 생성하지만 static은 정적 데이터 영역에 컴파일 시에 만들어진 것을 이용하므로 문제가 발생한다. 부장치 끼리의 경쟁을 제거한다고 해서 주장치끼리의 경쟁까지 제거해 주지는 않는다. 위와 같이 되었을 때 세마포를 주장치 단위로 설정하지 않고 주장치 전체에 한개만 만들면 해결되겠지만 드라이버의 성능은 엄청나게 떨어질 것이다. 16개의 부장치가 있다고 한다면 성능이 1/4로 떨어진다. 동시에 4개의 인터럽트 요청을 할 수 있는 것을 한번에 1개씩만 해야 하기 때문이다. 커널 디바이스 드라이버는 같은 하드웨어일 때 주장치가 다르더라도 같은 드라이버를 사용하기 때문에 부장치간에 그리고 주장치간에 서로 경쟁에 의해서 문제가 생기지 않도록 신경을 써야 한다. 이런

인터럽트 루틴 뿐만 아니라 입출력 함수에서도 static 배열을 사용하여 수행시간을 줄이려고 할 때 문제가 생기지 않도록 조심해야 한다.

커널에서는 프로세스에게 필요하다면 언제든지 시그널을 보낸다. 사용자가 입출력을 강제로 중단하거나 커널 자체의 문제로 인해 보내온 시그널을 받은 프로세스는 가능한 빨리 작업을 중단하고 복귀해야 한다. 그렇다고 꼭 필요한 작업까지 무시해서는 안된

다. 다음에 같은 드라이버를 사용할 프로세스가 정상적으로 진입하여 자원을 사용할 수 있도록 보장을 해야한다.

플래그세팅, 세마포처리등을 제대로 복원하고 돌아가야 한다. 시그널을 받았는지는 대부분 대기후 활성화되었을 때 수행하기 때문에 이때 어떤 자원을 복원해야 하는지 꼼꼼히 확인해야 한다.

드라이버를 호출하는 프로세스는 아무런 순서없이 생성되며, 커널은 언제라도 시그널을 보내어 프로세스를 간섭하고, 함수들 간에 부장치 끼리 그리고 커널 전체를 통해서 프로세스의 자원 점유를 위해서 경쟁하기 때문에 코드 순서에 따라 동작할 것이라고 가정하고 코딩을 하게 되면 제대로된 드라이버를 만들 수 없다.

드라이버를 작성할 때에는 모든 함수가 동시에 서로를 간섭할 수 있다는 것을 염두에 두고 동시다발적인 상황을 철저하게 따져가면서 코딩을 해야 할 것이다.

커널 드라이버 디버깅

디바이스 드라이버를 디버깅하는 것은 쉬운 일이 아니다. 드라이버 루틴은 커널모드에서 돌아가는 것이기 때문에 드라이버의 에러는 대부분 시스템 전체를 다운시켜 버리는 치명적 결과를 가져오게 된다. 에라가 생기면 리눅스 시스템 전체가 다운되고 최악의 상황에서는 파일시스템이 날아가 버려서 메세지도 확인할 수 없을 뿐 아니라 드라이버 소스까지 잃게 될 수 있다. 그러므로 커널 프로그래밍시에 백업은 기본적으로 해야 되고 여유가 있다면 똑같은 하드이미지를 만들어 놓기를 바란다. 파일시스템이 날아가더라도 똑같은 하드이미지를 만들어 놓았으면 백업본을 뒤져서 다시 리눅스를 인스톨하는 수고를 하지 않아도 새 하드디스크를 바꿔 달고 갱신된 드라이버 소스만 다시 복사하면 되기 때문이다. 똑같은 하드이미지란 같은 모델의 하드디스크를 두개 준비하여 원본 하드디스크를 /dev/hda에 복사본 하드디스크를 /dev/hdc 에 연결하고 플로피로 부팅한 다음 아래에 있는 두가지 명령 중에서 하나를 실행하여 완전히 같은 하드디스크를 만드는 것이다.

```
dd if=/dev/hda of=/dev/hdc
cat /dev/hda >/dev/hdc
```

이때 두 하드디스크는 마운트하지 않는다. 리눅스는 모든 데이타를 바이트열로 보기 때문에 위 두 명령은 물리적인 하드디스크 섹터를 읽어서 타켓 하드디스크에 쓰는 동일한 명령이다. 실행이 끝나면 마스터부트레코드까지 같은 하드이미지가 만들어진다. 물론 두 하드디스크는 베드섹터가 없어야 한다.

드라이버에 에라가 나서 시스템이 정지했을 때 이상없이 시스템을 재부팅할 수 있는 방법이 있다. 2.2.x 버전에서 컴파일 할 때 CONFIG_MAGIC_SYSRQ 옵션을 활성화 하면 된다. 테스트 도중에 에라가 생기고 키보드 입력을 받아 들이지 않고 네트웍으로

로그인도 되지 않는다면 아래와 같은 키를 사용할 수 있다.

alt+sysrq(print screen)+s : 마운트 되어 있는 파일 시스템에 아직 쓰지 않은 데이타가 있으면 모두 쓴다.
alt+sysrq+e : 모든 프로세스에게 SIGTERM 시그널을 보낸다.(init 제외)
alt+sysrq+i : 모든 프로세스에게 SIGKILL 시그널을 보낸다.(init 제외)
alt+sysrq+u : 마운트한 파일 시스템을 모두 읽기 전용으로 바꾸어 마운트한다.
alt+sysrq+b : 시스템을 리부팅한다.

여기에 있는 키를 차례로 누르면 파일 시스템을 잃을 염려 없이 정지한 시스템을 리부팅할 수 있다. 또다른 키들도 정의되어 있는데 메모리 상태를 보거나 타스크들의 정보를 보거나 죽일 수도 있다. 자세한 내용은 Documentation/sysrq.txt에 있다. 위에 적은 키는 순서까지 외워 놓고 비상시에 사용하기 바란다.

에러 추적을 위해서 printk 함수를 문제가 있다고 의심되는 부분에 넣고 테스트 한다. 콘솔에 이 메세지가 뜨게 되고 /var/log/messages에 같은 내용을 쓴다. 초기에는 가능한 많은 상태 메세지를 넣는 것이 좋다. 에라 출력에 등급을 두고 출력되는 정보량을 조절하면서 드라이버가 안정화 되어갈 수록 꼭 필요한 메세지만 나오도록 하면 된다.

커널 소스 최상위 디렉토리의 README와 Documentation/oops-tracing.txt에서 커널 모드의 디버거를 사용하는 법, 에러난 곳을 찾는 법에 대한 설명이 있다. 여기에 주로 설명한 것은 커널 전체에서 에러난 부분을 찾기 위한 방법과 개발자에게 에러 보고를 위해서 취해야 할 일에 대한 것이다. 우리가 하는 것은 직접 만든 드라이버를 디버깅하는 것이기 때문에 여기서 설명한 에러난 곳을 찾는 방법은 별로 도움이 되지 않는다. 잘 동작하는 커널에 직접 만든 드라이버 모듈을 삽입했을 때 에러가 났다면 당연히 우리의 드라이버 모듈에서 원인을 찾아야 하기 때문이다. 물론 이 문서를 읽는 것이 커널이 알려 주는 에러 메시지를 이해하는데는 상당한 도움이 될 것이다. 직접 만든 드라이버의 디버깅은 에러난 지점에서 어떤 논리적 문제가 있는지 스스로 따져 나가는 방법이 거의 유일한 것이다.

에러가 나지만 어느 정도 안정화 되어서 시스템이 정지하지는 않지만 제대로 동작하지 않을 때 에라가 난 시점에 드라이버의 변수 값등을 조사해 보고 싶으면 ioctl함수를 이용하여 그 기능을 구현할 수 있다.

우선 include/linux/my_device.h 에 그 기능에 대한 정의를 한다.

#define VIEW_VARIABLE _IOWR(MY_DEVICE_MAJOR, 2, int[2])

_IOWR의 두번째 인자 2는 커널의 my_device_ioctl에서 view_variable 에 해당하는 일을 하도록 임의로 정한 값이다. int[2]는 정수 두개를 주고 받을 수 있도록 마련한 영

역이다. 변수값을 보기 위해서 만드는 사용자 프로그램의 중요 부분은 다음과 같을 것이다.

```
...
int value[2];
FILE *fd; // my_device를 가르키는 파일 포인터
ret=ioctl(fd, VIEW_VARIABLE, value);
printf("variable 1 = %d, variable 2 = %d₩n",value[0],value[1]);
...
```

VIEW_VARIABLE 의 세번째 인자는 ioctl 함수의 세번째 인자의 크기를 알려주는 역할을 한다. value는 정수 2개를 할당할 수 있는 영역임을 알 수 있다.

파일 포인터 fd를 이용해 정상적으로 열리고 리턴값 ret가 이상이 없으면 그 값을 볼 수 있다. 커널 내부의 ioctl 함수 중에서 view_variable에 해당하는 부분은 다음처럼 만들 수 있다.

```
...
copy_from_user(&which, (void *)arg, sizeof(int));
if(which == 1) {
data[0] = my_device->variable_one;
date[1] = my_device->variable_two;
} else {
data[0] = my_device->variable_three;
date[1] = my_device->variable_four;
}
...
copy_to_user((void *)arg, data, sizeof(int) * 2);
...
```

arg는 ioctl을 부르는 함수의 value 배열을 가르키는 포인터이다. 사용자 영역에서 데이타를 읽어 온다. which라는 변수에 따라 다양한 커널 변수값을 선택할 수 있다. 요청한 데이타를 처리하고 이 데이타를 사용자 영역에 쓸 수 있는지 확인 한 다음 데이타를 보낸다. 지난 연재에서 말했듯이 커널 영역에서 사용자 프로세스의 데이타를 접근하기 위해서는 copy_to[from]_user 함수를 사용해야 한다. 또 ncurses의 텍스트 윈도우 함수와 wgetch 함수를 사용하면 커널 변수의 실시간 모니터링도 가능하다. ncurses 라이브러리의 wgetch 함수는 정한 시간 동안 사용자의 키보드 입력을 기다리고 키 입력이 없으면 다음 코드로 넘어가기 때문에 루프를 돌며 사용자가 끝내기 키를 누르기 전까지 수밀리초 간격으로 커널의 변수 내용을 보게 할 수 있다.