Linux Kernel Programming

v1.01

이호 (i@flyduck.com)

http://linuxkernel.to/

LinuxOnc, Inc

Linux Kernel Programming
by flyduck

v1.0 : 2000/10/17
v1.01 : 2000/11/06

이 문서는 리눅스 커널 프로그래밍 강의를 위한 자료입니다. 따라서 길고 자세한 설명보다는 간결하게 요약을 하고 있으며, 예제를 중심으로 하고 있습니다. 보다 자세하게 살펴보려면 참고문헌에 나오는 책과 자료를 보시고, 이 문서는 커널 프로그래밍의 개념과 기본적인 사항들을 이해하고, 실전 프로그래밍을 할 때 참고하는 용도로 사용하면 도움이 될 것입니다. 이 문서는 간단한 문자 디바이스 드라이버에서 블럭 디바이스 드라이버까지 다루고 있으며, 네트웍 장치나, SCSI 장치, USB 장치같이 복잡한 장치는 다루지 않습니다. 이런 내용은 개별 문서를 직접 보시면서 공부하시기 바랍니다. 이 문서의 내용은 2.2.x 커널을 기반으로 하고 있으며, 예제는 2.2.17 커널에서 테스트를 하였습니다. 내용중 틀린 부분이 있거나 추가했으면 좋겠다고 생각하시는 부분이 있다면 저자(i@flyduck.com)에게 메일을 보내주시면 고맙겠습니다. 질문 사항은 메일로 보내지 마시고 아래 나오는 홈페이지에 있는 Q&A 게시판을 이용해주시기 바랍니다. 이 문서에 있는 예제는 http://linuxkernel.to/의 커널 프로그래밍 관련 페이지에서 구할 수 있습니다.

리눅스 커널 홈페이지 : http://linuxkernel.to/
리눅스 커널 메일링 리스트 : http://flyduck.com/mailman/listinfo/linux-kernel

Reference

서적 :

Linux Device Drivers, Alessandro Rubini, O'Reilly

ISBN 1-56592-292-1

`ftp://ftp.oreilly.com`

(번역판) 리눅스 디바이스 드라이버, 한빛미디어

ISBN 89-7914-055-X

문서 :

Kernel Module Programming Guide, Ori Pomerantz

`http://linuxdoc.org/LDP/lkmpg/mpg.html`

(번역판) Kernel Module Programming Guide

`http://kldp.org/Translations/Kernel_Module_Programming_Guide`

Unreliable Guide to Hacking the Linux Kernel, Paul Rusty Russell

`http://netfilter.kernelnotes.org/unreliable-guides/`

차례

1. 커널 프로그래밍의 개요

- 리눅스 커널의 구조 :
  - 응용프로그램 : 사용자 모드(user mode)에서 동작하는 프로그램
  - 시스템 콜 인터페이스 : 응용프로그램에게 커널 서비스를 제공하는 인터페이스
  - 커널 : 프로세스 관리, 메모리 관리, IPC, 파일시스템, 네트워킹, 디바이스 드라이버 등
  - 하드웨어 : CPU, RAM, 디스크, 네트웍, 여러 주변장치들

- 디바이스 드라이버 :
  - 장치를 직접적으로 다루는 소프트웨어.
  - 꼭 하드웨어적인 장치가 아니라도 소프트웨어적인 장치를 만들어 다룰 수도 있다.
  - 커널 모드에서 실행되고, 메모리에 상주하며, 스왑되지 않는다.
  - 디바이스 드라이버를 통하여 장치를 추상화 : 장치를 파일처럼 다룰 수 있다.
  - 장치를 몇가지 종류로 나누고 종류에 따라 정의된 인터페이스를 통하여 장치를 제어한다.

- 장치의 종류 :
  - 문자 장치(character device) : 버퍼를 통하지 않고, 바로 읽고 쓸 수 있는 장치.
    terminal, serial, parallel, keyboard, mouse, PC speaker, joystick, scanner, ...
  - 블럭 장치(block device) : 버퍼 캐시를 통해 입출력을 하며, 장치의 어디든 접근이 가능하다 (random access). 블럭단위로 입출력을 하며, 파일시스템을 구축할 수 있다.
    floppy disk, hard disk, RAM disk, tape, CD-ROM, ...
  - 네트웍 장치(network device) : 네트웍 패킷을 송수신할 수 있는 장치.
    ethernet, slip, ppp

- 커널 모듈(kernel module) :
  - 리눅스 시스템 부팅후에 동적으로 로드, 언로드 할 수 있는 커널의 구성요소.
  - 커널을 다시 컴파일하거나 시스템을 리부팅하지 않고도 커널의 일부분을 교체할 수 있다.
  - 디바이스 드라이버, 파일시스템, 네트웍 프로토콜 등이 모듈로 만들어진다.

- 모듈 프로그램과 일반 응용프로그램의 차이 :
  - `main()` 함수가 없고 로드와 언로드시에 불리는 startup / cleanup 함수가 존재한다.
    startup : `int init_module(void)`
    성공하면 0, 실패하면 그 외의 값을 돌려준다.
    cleanup : `void cleanup_module(void)`
  - dynamic linking :
    symbol : 변수나 함수 이름
    symbol import : 다른 곳에 존재하는 symbol 을 사용하는 것
    symbol export : 다른 곳에서 사용할 수 있도록 symbol 을 제공하는 것
    symbol table : symbol 들의 목록을 가지고 있는 table
    static linking : 컴파일을 하면서 link 작업을 같이 한다. 모든 심볼들이 resolve 되어 있기 때문에 그 자체로 실행 가능하다.
    dynamic linking : symbol 들의 link 는 모듈을 로드할 때 한다. import 되는 symbol 들을 다른 곳에서 제공을 해주어야 한다.
    모듈은 컴파일만을 하고 링크를 하지 않은 object 파일 형태이다.
  - 커널과 현재 로드된 모듈에서 export 하고 있는 symbol 만을 사용할 수 있다.

● 모듈 프로그래밍에서의 주의사항 :

- fault handling : 모듈은 커널모드에서 동작하므로 메모리 접근에 대한 어떠한 보호도 하지 않으며, 모듈에서 발생한 에러는 시스템에 치명적이므로 메모리를 다룰 때 주의하고, 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 동일한 스택을 사용하므로 스택을 많이 사용하면 안된다. 스택에 큰 배열을 잡으면 안되고 (동적으로 할당받도록 한다), recursion 이 많이 일어나지 않도록 주의한다.
- 다른 플랫폼을 고려해야 한다. 리눅스는 여러 환경에서 사용될 수 있으므로 32 비트와 64 비트 환경과 byte order 등을 모두 고려를 해야 하며, CPU 에 특화된 코드는 되도록 줄이도록 한다.

● 모듈 관련 프로그램 :

- lsmod : 로드된 모듈의 목록을 보여준다. (`/proc/modules`)
- insmod : 커널 모듈을 로드한다.
- rmmod : 커널 모듈을 언로드한다.
- depmod : 커널 모듈간의 의존성을 검사한다. (`/lib/modules/*/modules.dep`)
- modprobe : 모듈을 보다 높은 수준에서 다루는 프로그램.
  의존성에 따라서 필요한 경우 다른 모듈을 로드, 언로드를 한다.
- kerneld : 커널 모듈 로더. 응용프로그램이 아닌 커널 쓰레드의 일종
  커널에서 모듈을 필요로 할 때 자동으로 모듈을 로드하고, 필요없을 때 언로드한다.
  modprobe 프로그램을 이용하여 로드, 언로드를 한다.

- 모듈 프로그래밍 관련 프로그램 :
  - ksyms : export 되어 있는 현재 커널의 symbol 목록을 보여준다. (/proc/ksyms)
  - gcc : GNU C compiler
  - make : 프로젝트 관리에 유용한 GNU make utility
  - ld : GNU linker (ld -r : relocatable object file)
  - nm : object file 에 있는 symbol 들의 목록을 보여준다.
  - mknod : device special file 을 만든다.

2. 첫번째 프로그램 : Hello, World

- 목적 : 모듈을 로드하고 언로드할 때 간단한 메시지를 출력하는 모듈 프로그램을 제작한다.

- 구현 방법 : 모듈을 로드할 때 불리는 함수인 `init_module()`에서 커널 메시지를 출력하는 함수인 `printk()`를 이용 "Hello, World" 문자열을 출력한다. 모듈을 언로드할 때 불리는 함수인 `cleanup_module()`에서는 같은 방법으로 "Goodbye" 문자열을 출력한다.

- 테스트 방법 : 모듈을 로드하고 언로드할 때 메시지가 제대로 출력되는지 확인한다.

- 과정 :
  - 소스 입력
  - 컴파일 (gcc)
  - 모듈을 로드하고 결과를 본 후 언로드하기 (insmod, lsmod, rmmod)
  - Makefile 사용하기 (make)
  - 소스를 두개로 나누어 컴파일하기 (gcc, ld, make)
  - export 할 심볼 목록을 제어하기

- **Filename : hello.c**

```
/*
     Hello, World Module Program
*/

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

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

int init_module(void)
{
     printk("<1>Hello, World\n");

     return 0;
}

void cleanup_module(void)
{
     printk("<1>Goodbye\n");
}
```

- **Filename : hello_init.c**

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

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

int init_module(void)
{
     printk("<1>Hello, World\n");

     return 0;
}
```

- **Filename : hello_cleanup.c**

```
#define __NO_VERSION__
#include <linux/kernel.h>
#include <linux/module.h>

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

void cleanup_module(void)
{
     printk("<1>Goodbye\n");
}
```

- **Filename : Makefile (hello.c)**

```
CC = gcc
CFLAGS = -D__KERNEL__ -DMODULE -O -Wall

OBJS = hello.o

all : $(OBJS)

clean :
     rm -f *.o *~
```

- **Filename : Makefile2 (hello_init.c, hello_cleanup.c)**

```
CC = gcc
LD = ld
CFLAGS = -D__KERNEL__ -DMODULE -O -Wall

TARGET = hello.o
OBJS = hello_init.o hello_cleanup.o

all : $(TARGET)

$(TARGET) : $(OBJS)
     $(LD) -r $^ -o $@

clean :
     rm -f *.o *~
```

● 모듈 컴파일하기 :

```
gcc -c -D__KERNEL__ -DMODULE -O hello.c
```

● make 프로그램을 이용한 컴파일 :

```
make                    => 기본 makefile 인 Makefile 을 이용하여 컴파일
make clean              => 컴파일한 결과물을 제거하기
make -f Makefile2       => 다른 makefile 을 사용할 때 사용할 makefile 이름 지정
```

● 모듈 설치 :

```
insmod hello.o
```

● 로드된 모듈 목록 보기 :

```
lsmod
```

● 모듈 제거 :

```
rmmod hello
```

● 모듈에서 출력한 메시지 보기 :

```
tail /var/log/messages          => 최근 메시지 보기
tail -f /var/log/messages       => 최근 메시지와 이후 계속해서 추가되는 메시지 보기
```

● 모듈 버전 문제 :

- 모든 모듈에는 컴파일한 커널 버전 정보가 들어가야 하며, 이는 현재 실행되고 있는 커널 버전과 일치해야 한다.
- 모듈에 버전 정보 포함시키기 :

  `#include <linux/module.h>`에서 자동으로 모듈 버전 정보를 추가한다.

  모듈 버전은 `char kernel_version[]` 변수로 정의된다.

  모듈 버전 정보는 전체 모듈에서 하나만 있어야 한다. 소스가 여러개인 경우 하나의 소스에서만 모듈 버전을 정의하고 나머지 소스에서는 <linux/module.h> 파일을 포함하기 전 `#define __NO_VERSION__`을 통해 모듈 버전을 정의하지 않도록 한다.
- 모듈 버전이 틀리거나 모듈 버전 정보가 없다고 할 때 문제 해결 :

  1) 현재 커널의 모듈 버전을 포함시켜 컴파일을 다시 한다 : 가장 좋은 방법

  2) 버전이 틀리더라도 강제로 모듈 로드하기 : insmod -f

     대체로 minor version 이 같은 경우(2.2.x) 버전이 틀리더라도 호환되는 경우가 많지만, minor versioin 이 틀린 경우에는 호환되지 않을 가능성이 높다.

  * 부팅할 때 사용한 커널 버전과 모듈을 컴파일할 때 사용하는 버전이 틀린 경우 두개의 버전을 맞추어주어야 한다.

● unresolved symbol 에러 문제 :

1) 사용할 수 없는 심볼을 사용한 경우 :

모듈에서는 커널에 정의되어 있는 모든 심볼을 사용할 수 없으며, 커널에서 export 해주는 심볼만 사용할 수 있다.

심볼이 반드시 필요한 경우 커널을 직접 수정하여 해당 심볼을 export 시켜주어야 한다. (`kernel/ksyms.c`, `net/netsyms.c` 또는 각 소스에서)

2) symbol 들에 버전 정보가 포함되어 있는 경우 :

같은 심볼이라도 버전 정보를 포함시켰는지 여부에 따라서(CONFIG_MODVERSION) export 를 할 때 다르게 된다. `printk()` 함수를 예로 들면 커널이 버전 검사를 하지 않으면 그냥 `printk` 로 export 되지만, 버전 검사를 하면 `printk` 뒤에 모듈 버전 정보가 추가되어 `printk_xxxxxx` 형태로 export 된다.

코드에서 다음 부분이 이를 맞추어주는 역할을 한다.

```
#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif
```

3) inline function optimizing :

inline function 으로 정의되어 있는 몇가지 함수들(예 : `get_user`, `put_user`, `strcpy`, `strcmp`)은 optimizing 을 하지 않거나 지나친 optimizing 을 하면 컴파일시에 inline function 이 아닌 일반 function 으로 여길 수도 있다. 이경우 inline function 을 external symbol 로 여기고 link 를 시도하기 때문에 에러가 발생한다.

컴파일 옵션에서 -O 를 반드시 지정해야 한다.

- 최대 optimizing 은 -O2
- -O3 optimizing 은 inline function 을 function 으로 바꿀 수 있기 때문에 사용할 수 없다. (-fkeep-inline-functions)

- 모듈에서 사용할 수 있는 변수와 함수 :

커널에서 export symbol table 을 통하여 export 해주는 심볼

현재 로드된 모듈에서 export 해주는 심볼

inline function 으로 정의된 함수

- symbol export 문제 :
  - 별도로 심볼의 export 여부를 지정하지 않으면 모듈에서 export 가능한 모든 심볼들(static 으로 지정하지 않은 모든 global 심볼들)이 자동으로 커널의 symbol table 에 추가가 된다 => namespace pollusion 을 가져온다.
  - 모듈에서 아무런 심볼도 export 하지 않으려면 소스에 다음을 추가한다.

    ```
    EXPORT_NO_SYMBOLS;
    ```

  - 원하는 심볼만 export 를 하려면 별도로 심볼들의 목록을 지정한다.

    ```
    #define EXPORT_SYMTAB (#include <linux/module.h> 이전에 정의)

    EXPORT_SYMBOL(symbol1); (export 할 심볼들을 차례로 정의)
    EXPORT_SYMBOL(symbol2);
    ```

  - insmod -x : EXPORT_SYMBOL()로 별도로 지정하지 않은 경우 모든 external symbol 들은 자동으로 symbol table 에 추가 되는데 이를 symbol table 에 추가하지 않게 한다.
  - insmod -y : external symbol 이 아니더라도 ksymoops 를 위해 모듈에 대한 정보를 포함하고 있는 심볼을 자동으로 등록하는데 이를 등록하지 않는다. 이 심볼들은 __insmod_[모듈이름]_xxx 의 형태로 되어 있다.

3. 두번째 프로그램 : MiniBuf - 간단한 문자 디바이스 드라이버

- 목적 : 먼저 읽기가 가능한 간단한 문자 디바이스 드라이버를 제작한다. 이 장치는 읽기 기능만을 가지고 있으며, 장치에서 읽기를 하면 내부 버퍼에 들어 있는 초기값을 돌려준다. 두번째로 읽기 쓰기가 가능한 디바이스 드라이버를 제작한다. 여기서는 장치에 쓴 내용을 내부 버퍼에 넣어 두었다가 읽기를 하면 이를 돌려준다. 세번째로 쓰기를 할 때 내부 버퍼의 용량이 부족하면 blocking 이 발생하는 디바이스 드라이버를 제작한다.

- 구현 방법 : 문자 장치 구현에 필요한 file operation 들을 구현한다. 여러가지 operation 중에서 여기서 필요한 것은 open, release, read, write 이다. open, release 에서는 모듈의 사용 횟수를 관리한다. 입력받은 데이터를 저장하기 위해 고정된 크기의 원형버퍼(circular buffer)를 하나 둔다. 버퍼는 전역 변수로 두며 동시에 읽거나 동시에 쓰기를 할 수는 없다. read 에서는 내부 버퍼에 있는 내용을 돌려주고, write 에서는 입력받은 내용을 내부 버퍼에 저장을 한다. write 를 할 때 버퍼의 용량이 부족하면 이를 적절히 처리한다.

- 테스트 방법 : 모듈을 제작하고 로드한 후 mknod 명령으로 해당하는 장치 특수 파일(device special file)을 만든다. 여기서는 장치의 major number 를 지정하지 않았으므로 커널에서 할당해 준 major number 에 따라서 장치 파일을 만든다. 첫번째 모듈에서는 cat 명령을 이용하여 장치를 읽었을 때 버퍼에 미리 넣어둔 내용이 나오는지 확인한다. 두번째 모듈에서는 쓰기를 한 후 이후에 읽기를 하였을 때 이전에 쓴 내용이 제대로 나오는지 확인한다. 세번째 모듈에서는 버퍼의 크기를 넘어서 쓰기를 하였을 때 blocking 이 되는지 여부와, 장치를 읽어들이면 wakeup 이 되는지 확인한다.

● 과정 :

- 읽기만 지원하는 문자 디바이스 드라이버 제작
- 읽기/쓰기를 모두 지원하는 문자 디바이스 드라이버 제작
- 쓰기를 할 때 버퍼가 부족하면 blocking 이 되는 디바이스 드라이버 제작

● 테스트 과정 :

1) 읽기만 지원하는 문자 디바이스 드라이버 :

```
insmod minibuf.o
cat /proc/devices                <= 시스템에 설치된 디바이스 드라이버의 목록을 확인
mknod /dev/minibuf c major 0     <= major number 는 메시지 출력 결과에서 확인
cat /dev/minibuf
rmmod minibuf
```

2) 읽기/쓰기를 모두 지원하는 문자 디바이스 드라이버 :

```
insmod minibuf2.o
cat > /dev/minibuf
cat /dev/minibuf
rmmod minibuf2
```

3) 쓰기를 할 때 버퍼가 부족하면 blocking 이 되는 디바이스 드라이버

```
insmod minibuf3.o
cat minibuf3.c > /dev/minibuf        <= blocking
cat /dev/minibuf                     <= wakeup
rmmod minibuf3
```

- Filename : minibuf.c

```
/*
   Mini Buffer Device Driver
*/

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

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/fs.h>
#include <linux/kdev_t.h>
#include <asm/uaccess.h>

/*
 * Device Definitions
 */

#define DEVICE_NAME           "MiniBuf"
#define BUFFER_LEN            1024

/*
 * Global Variables
 */

static int s_nMajor = 0;
```

```
static int s_bDeviceOpen = 0;
static char s_strBuf[BUFFER_LEN];
static int s_nBufPos = 0, s_nBufEnd = 0;

/*
 * Function Prototypes
 */

static int device_open(struct inode *inode, struct file *filp);
static int device_release(struct inode *inode, struct file *filp);
static ssize_t device_read(struct file *filp, char *buffer, size_t length,
     loff_t *offset);
static ssize_t device_write(struct file *filp, const char *buffer, size_t length,
     loff_t *offset);

static int is_buffer_empty(void);
static int is_buffer_full(void);
static int read_buffer_char(char *buffer);
static int write_buffer_char(char *buffer);

/*
 * Device Operations
 */

struct file_operations device_fops = {
   NULL,            /* seek */
   device_read,     /* read */
   device_write,    /* write */
   NULL,            /* readdir */
```

```
    NULL,            /* poll */
    NULL,            /* ioctl */
    NULL,            /* mmap */
    device_open,     /* open */
    NULL,            /* flush */
    device_release   /* release */
};

/*
 * Module startup/cleanup
 */

int init_module(void)
{
    printk("Loading Mini Buffer Module\n");

    if ((s_nMajor = register_chrdev(0, DEVICE_NAME, &device_fops)) < 0) {
        printk(DEVICE_NAME " : Device registration failed (%d)\n", s_nMajor);
        return s_nMajor;
    }

    printk(DEVICE_NAME " : Device registered with Major Number = %d\n", s_nMajor);

    strcpy(s_strBuf, "Hello, World\n");
    s_nBufEnd = strlen(s_strBuf) + 1;

    return 0;
}

void cleanup_module(void)
```

```
{
    int nRetCode;

    printk("Unloading Mini Buffer Module\n");

    if ((nRetCode = unregister_chrdev(s_nMajor, DEVICE_NAME)) < 0)
        printk(DEVICE_NAME " : Device unregistration failed (%d)\n", nRetCode);
}

/*
 * Device Operations
 */

int device_open(struct inode *inode, struct file *filp)
{
    printk(DEVICE_NAME " : Device open (%d, %d)\n", MAJOR(inode->i_rdev),
        MINOR(inode->i_rdev));

    if (s_bDeviceOpen) {
        printk(DEVICE_NAME " : Device already open\n");
        return -EBUSY;
    }

    ++s_bDeviceOpen;

    MOD_INC_USE_COUNT;

    return 0;
}
```

```
int device_release(struct inode *inode, struct file *filp)
{
   printk(DEVICE_NAME " : Device release (%d, %d)\n", MAJOR(inode->i_rdev),
      MINOR(inode->i_rdev));

   if (!s_bDeviceOpen) {
      printk(DEVICE_NAME " : Device has not opened\n");
      return -EINVAL;
   }

   --s_bDeviceOpen;

   MOD_DEC_USE_COUNT;

   return 0;
}

ssize_t device_read(struct file *filp, char *buffer, size_t length, loff_t *offset)
{
   int count = 0;

   if (is_buffer_empty()) {       /* end of file */
      printk(DEVICE_NAME " : Read return EOF\n");
      return 0;
   }

   while (!is_buffer_empty() && length > 1) {
      read_buffer_char(buffer);
      ++buffer;
      --length;
```

```
        ++count;
    }

    put_user(0, buffer);
    ++count;

    printk(DEVICE_NAME " : Read %d bytes\n", count);

    return count;
}

ssize_t device_write(struct file *filp, const char *buffer, size_t length, loff_t *offset)
{
    return -ENOSYS;              /* not implemented */
}

/*
 * Buffer Management
 */

int is_buffer_empty(void)
{
    return (s_nBufPos == s_nBufEnd) ? 1 : 0;
}

int is_buffer_full(void)
{
    int pos = s_nBufEnd + 1;

    if (pos == BUFFER_LEN)
```

```
        pos = 0;

    return (pos == s_nBufPos) ? 1 : 0;
}

int read_buffer_char(char *buffer)
{
    if (is_buffer_empty())
        return -1;

    put_user(s_strBuf[s_nBufPos], buffer);

    if (++s_nBufPos == BUFFER_LEN)
        s_nBufPos = 0;

    return 0;
}

int write_buffer_char(char *buffer)
{
    if (is_buffer_full())
        return -1;

    get_user(s_strBuf[s_nBufEnd], buffer);

    if (++s_nBufEnd == BUFFER_LEN)
        s_nBufEnd = 0;

    return 0;
}
```

- Filename : minibuf2.c

```
ssize_t device_write(struct file *filp, const char *buffer, size_t length,
     loff_t *offset)
{
   int count = 0;

   if (is_buffer_full()) {        /* out of buffer */
      printk(DEVICE_NAME " : Write return Out of Buffer\n");
      return -ENOMEM;
   }

   while (!is_buffer_full() && length > 0) {
      write_buffer_char((char *) buffer);
      ++buffer;
      --length;
      ++count;
   }

   filp->f_pos += count;

   printk(DEVICE_NAME " : Write %d bytes\n", count);

   return count;
}
```

- Filename : minibuf3.c

```
static int s_bDeviceReadOpen = 0;
static int s_bDeviceWriteOpen = 0;
static struct wait_queue *s_wq = NULL;
```

in device_open() :

```
    if ((filp->f_flags & O_ACCMODE) & (O_WRONLY | O_RDWR)) {
        if (s_bDeviceWriteOpen) {
            printk(DEVICE_NAME " : Device already open for writing\n");
            return -EBUSY;
        } else
            ++s_bDeviceWriteOpen;
    } else {
        if (s_bDeviceReadOpen) {
            printk(DEVICE_NAME " : Device already open for reading\n");
            return -EBUSY;
        } else
            ++s_bDeviceReadOpen;
    }
```

in device_release() :

```
    if ((filp->f_flags & O_ACCMODE) & (O_WRONLY | O_RDWR))
        --s_bDeviceWriteOpen;
    else
        --s_bDeviceWriteOpen;
```

in device_read() :

```
    while (!is_buffer_empty() && length > 1) {
        read_buffer_char(buffer);
        ++buffer;
        --length;
        ++count;
        wake_up_interruptible(&s_wq);
    }
```

in device_write() :

```
    if (is_buffer_full()) {        /* out of buffer */
        printk(DEVICE_NAME " : Write go to sleep\n");
        interruptible_sleep_on(&s_wq);
        printk(DEVICE_NAME " : Write wake up");
    }
```

- 문자 디바이스 드라이버의 제작법
  - 외부와 디바이스 드라이버는 파일 인터페이스를 통해서 연결된다.
  - 디바이스 드라이버는 `file_operations` 를 제공함으로써 구현된다.
  - 디바이스 드라이버는 자신을 구별하기 위해 고유의 major number 를 사용한다.

- 장치의 등록과 해제
  - 등록 :

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

    `major` : 등록할 major number. 0 이면 사용하지 않는 번호중에서 동적으로 할당된다.
    `name` : 장치의 이름
    `fops` : 장치에 대한 파일 연산 함수들
  - 해제 :

```
int unregister_chrdev(unsigned int major, const char *name)
```

- Major Number 와 Minor Number
  - 장치를 구분하는 방법으로 둘을 이용하여 특정 장치를 구별한다.
  - major number : 커널에서 디바이스 드라이버를 구분하는데 사용한다.
  - minor number : 디바이스 드라이버 내에서 필요한 경우 장치를 구분하기 위해서 사용한다.
  - 새로운 디바이스 드라이버는 새로운 major number 를 가져야 한다.
  - `Documentation/devices.txt` 에 모든 장치의 major number 가 정의되어 있다.
  - `register_chrdev()`로 장치를 등록할 때 major number 를 지정한다.
    똑같은 major number 가 이미 등록되어 있으면 등록이 실패한다.
    dynamic allocation : `major` argument 에 0 전달

개발 과정에서는 임의의 major number 를 사용할 수 있지만, 배포시에는 major number 를 할당받아야 한다.

- mknod 명령으로 디바이스 드라이버에 접근할 수 있는 장치 파일 생성
  mknod [device file name] [type] [major] [minor]
- kdev_t : 장치의 major, minor number 를 표현하는 자료구조
  MAJOR() : kdev_t 에서 major number 를 얻어내는 매크로
  MINOR() : kdev_t 에서 minor number 를 얻어내는 매크로
  MKDEV(ma, mi) : major number, minor number 를 가지고 kdev_t 를 만든다.
- cat /proc/devices 명령으로 현재 로드된 디바이스 드라이버 목록을 볼 수 있다.

- struct file : (include/linux/fs.h)
  - opened file 을 표현하는 자료구조
  - open() 함수에 의해서 만들어지고, 나머지 함수에 전달된다.

```
struct dentry *f_dentry;          => directory entry
struct file_operations *f_op;     => file operations (function pointers)
mode_t f_mode;                    => FMODE_xxx
loff_t f_pos;                     => 현재 파일에서 읽고 쓰는 위치
unsigned int f_flags;             => access mode, nonblock, create, append 같은 flag
unsigned int f_uid, f_gid;        => file owner uid, gid
void *private_data;               => open file 마다 추가적인 데이터가 필요할 때
```

- struct file_operations : (include/linux/fs.h)

```
loff_t (*llseek) (struct file *, loff_t, int);
```
파일에서 현재 읽고 쓰는 위치를 이동한다.
```
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
```
장치에서 데이터를 읽어들인다.
```
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
```
장치에 데이터를 기록한다.
```
int (*readdir) (struct file *, void *, filldir_t);
```
장치에서는 사용하지 않고 디렉토리에서만 사용하는 함수이다.
```
unsigned int (*poll) (struct file *, struct poll_table_struct *);
```
장치에 읽고 쓸 수 있는지, 또는 어떤 예외상황이 발생했는지 검사한다.
poll(), select() 함수를 구현하는데 사용한다.
```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```
읽기/쓰기가 아닌 장치마다 필요한 명령을 내리는데 사용한다.
```
int (*mmap) (struct file *, struct vm_area_struct *);
```
장치의 메모리를 프로세스의 메모리와 mapping 을 한다.
```
int (*open) (struct inode *, struct file *);
```
장치를 연다.
```
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
```
장치를 닫는다.
```
int (*fsync) (struct file *, struct dentry *);
```
장치를 flush 한다.
```
int (*fasync) (int, struct file *, int);
```

asynchronous notification 을 나타내는 flag 인 FASYNC flag 가 바뀐 것을 알려준다.

```
int (*check_media_change) (kdev_t dev);
```

플로피같이 사용 도중에 media 가 바뀔 수 있는 블럭 디바이스 드라이버에서 사용한다.

```
int (*revalidate) (kdev_t dev);
```

제거 가능한 블럭 장치에서 buffer cache 를 관리하기 위해서 사용한다.

```
int (*lock) (struct file *, int, struct file_lock *);
```

- open 에서 할 일 :
  - 처음으로 장치를 연 경우 장치를 초기화한다.
  - minor 번호를 확인하고 필요한 경우 `f_op` 포인터를 수정한다.
  - 필요한 경우 메모리를 할당받아 `filp->private_data` 에 넣는다
  - 참조 횟수(usage count)를 증가시킨다.

- release 에서 할 일 :
  - 참조 횟수를 감소시킨다.
  - `filp->private_data` 에 할당받은 데이터가 있으면 해제한다.
  - 마지막으로 close 를 하는 경우 장치를 종료한다.

- read 의 return value
  - 요구한만큼 읽은 경우 `count` 와 같은 값을 돌려준다.
  - 요구한 크기보다 적게 읽은 경우 `count` 보다 작은 양수를 돌려준다.
  - EOF(end of file)인 경우 0 을 돌려준다.
  - 에러가 발생하면 음수를 돌려준다.

- write
  - 요구한만큼 기록한 경우 `count` 와 같은 값을 돌려준다.
  - 요구한 크기보다 적게 쓴 경우 `count` 보다 작은 양수를 돌려준다
  - 하나도 쓰지 못한 경우 0 을 돌려준다. 이 경우 `write()` 함수는 재시도를 한다.
  - 에러가 발생한 경우 음수를 돌려준다.

- 커널 메모리와 사용자 메모리의 데이터 교환 : (include/asm/uaccess.h)
  - 커널 메모리와 사용자 메모리는 서로 다른 주소공간에 존재하기 때문에 커널에서 사용자 메모리를 바로 접근할 수 없다. 따라서 read, write 나 system call 처럼 커널과 응용프로그램 사이에 데이터 교환이 필요한 경우 서로의 주소 공간으로 복사를 해야 한다.
  - 사용자 메모리 => 커널 메모리 :
    ```
    void get_user(to, from);
    void copy_from_user(to, from, n);
    ```
  - 커널 메모리 => 사용자 메모리 :
    ```
    void put_user(from, to);
    void copy_to_user(to, from, n);
    ```
  - `get_user()`, `put_user()` : 넘겨준 인자의 크기에 따라 1, 2, 4 byte 데이터 복사
  - `copy_from_user()`, `copy_to_user()` : 크기 제한 없이 지정한 크기만큼 복사

- 모듈의 참조 횟수 : 모듈 사용중 언로드되는 걸 방지 (include/linux/module.h)
  - 모듈이 다른 모듈이나 프로그램에 의해서 몇번이나 사용되고 있는지를 나타낸다.
  - 디바이스 드라이버를 사용하고 있을 때 rmmod 같은 명령으로 모듈이 제거되는 걸을 막기 위해서 open 을 할 때 참조 횟수를 증가시키기고, release 를 하면 참조 횟수를 감소시켜 언로드 될 수 있게 한다.
  - 모듈 참조 횟수 증가 : `MOD_INC_USE_COUNT`

- 모듈 참조 횟수 감소 : `MOD_DEC_USE_COUNT`
- 모듈 참조 여부 : `MOD_IN_USE`

- Blocking I/O
  - 자원을 필요로 하는데, 현재 자원을 사용할 수 없을 경우, 자원을 사용할 수 있을 때까지 기다리며 sleep 을 하고, 자원을 사용할 수 있게되면 wakeup 이 되어 작업을 계속한다.
  - read : 읽으려고 하는데 읽을 데이터가 없을 때 blocking. 데이터가 도착하면 wakeup
  - write : 쓰려고 하는데 쓸 공간이 없을 때 blocking, 공간이 생기면 wakeup
  - sleep 함수 :

```
void interruptible_sleep_on(struct wait_queue **q);
void sleep_on(struct wait_queue **q);
```

  - wakeup 함수 :

```
void wake_up_interruptible(struct wait_queue **q);
void wake_up (struct wait_queue **q);
```

  - interruptible blocking : 일반적인 I/O 에 의해서 발생한 blocking
    파일에서 읽기/쓰기, 파이프, 시스템 V IPC, 소켓 등
  - uninterruptible blocking : 커널의 critical section 에서 발생한 blocking
    page fault 가 발생하여 디스크에서 페이지를 읽어들이는 경우
  - reentrance 문제 :
    blocking 이 발생한 코드는 재진입이 일어날 수 있다.
    reentrace 문제 해결 :
      장치에 접근할 수 있는 횟수 제한하기 (open)
      global 변수가 아닌 local 변수 사용하기 (stack)
      struct file 의 private data 사용 : 열린 파일마다 데이터 할당
      필요한 경우 kmalloc() 등을 사용하여 heap 영역 이용

4. 세번째 프로그램 : MiniBuf- 좀 더 복잡한 문자 디바이스 드라이버

- 목적 : 첫번째로 MiniBuf 프로그램을 개선하여 버퍼를 정적으로 잡지 않고 동적으로 할당을 받는 디바이스 드라이버를 제작한다. 여기서 동적으로 할당받는 버퍼의 크기는 기본적으로 기본값으로 설정된 크기이지만, insmod 프로그램으로 모듈을 로드할 때 버퍼의 크기를 인자로 지정할 수도 있다. 두번째로 minor 번호에 따라서 각기 다른 동작을 하는 디바이스 드라이버를 제작한다. 이전에 제작한 코드를 이용하여 minor 번호가 0 일 때에는 blocking 을 하지 않고, 1 일 때에는 쓰기에서 blocking 을 하며, 2 일 때에는 읽기에서도 blocking 을 한다.

- 구현방법 : 기존의 minibuf2.c 코드를 수정하여 버퍼를 정적으로 잡지 않고, `init_module()`에서 `kmalloc()` 함수를 이용하여 동적으로 메모리를 할당받는다. 이 때 insmod 프로그램에 전달한 인자를 검사하여 버퍼의 크기를 지정하고 있으면, 해당하는 크기의 버퍼를 할당받는다. `cleanup_module()`에서는 처음에 할당받은 버퍼를 `kfree()` 함수를 이용하여 해제한다. 두번째 모듈은 이번에 제작한 minibuf4.c 코드를 수정하여, 각기 다른 동작을 하는 read/write 함수를 구현하고, open 시에 minor 번호에 따라 `file` 구조체의 file operation 을 각기 다른 함수로 지정을 하여 minor 번호에 따라 다른 동작을 하도록 한다.

- 테스트 방법 : 첫번째 모듈은 모듈을 로드할 때 버퍼의 크기를 지정한다. 그리고 해당한 크기 이상의 데이터를 기록하였을 때 더 이상 기록이 되지 않고 메모리 부족 에러가 발생하는지 본다. 두번째 모듈은 minor 번호가 다른 장치 파일을 세개를 만들어 각각의 동작을 확인한다.

● 과정 :

- 버퍼를 동적으로 할당받는 문자 디바이스 드라이버 제작
- minor 번호에 따라 다른 동작을 하는 문자 디바이스 드라이버 제작

● 테스트 과정 :

1) 버퍼를 동적으로 할당받는 문자 디바이스 드라이버 :

```
insmod minibuf4.o minibuf_bufsize=10
cat > /dev/minibuf        <= 버퍼 크기 이상의 데이터 입력
cat > /dev/minibuf        <= 추가로 입력하였을 때 에러가 발생하는지 확인
cat /dev/minibuf          <= 버퍼 크기만큼만 저장되었는지 확인
rmmod minibuf
```

2) minor 번호에 따라 다른 동작을 하는 문자 디바이스 드라이버 :

```
insmod minbuf5.o
mknod /dev/minibuf0 c major 0
mknod /dev/minibuf1 c major 1
mknod /dev/minibuf2 c major 2
cat minibuf5.c > /dev/minibuf0          <= 버퍼 크기만큼만 입력
cat /dev/minibuf0                       <= 버퍼 크기만큼만 출력
cat minibuf5.c > /dev/minibuf1          <= blocking
cat /dev/minibuf1                       <= wakeup
cat /dev/minibuf2                       <= blocking
cat > /dev/minibuf2                     <= wakeup
rmmod minibuf5
```

- Filename : minibuf4.c

```
/*
   Mini Buffer Device Driver : Dynamic allocation
*/

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

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/fs.h>
#include <linux/kdev_t.h>
#include <linux/slab.h>
#include <asm/uaccess.h>

/*
 * Device Definitions
 */

#define DEVICE_NAME           "MiniBuf"
#define DEFAULT_BUFFER_LEN     1024

/*
 * Global Variables
 */
```

```
static int s_nMajor = 0;

static int s_bDeviceOpen = 0;
static char *s_strBuf = NULL;
static int s_nBufPos = 0, s_nBufEnd = 0;
static int s_nBufSize = 0;

static int minibuf_bufsize = DEFAULT_BUFFER_LEN;

MODULE_PARM(minibuf_bufsize, "i");
MODULE_PARM_DESC(minibuf_bufsize, "Size of buffer");

/*
 * Function Prototypes
 */

static int device_open(struct inode *inode, struct file *filp);
static int device_release(struct inode *inode, struct file *filp);
static ssize_t device_read(struct file *filp, char *buffer, size_t length,
     loff_t *offset);
static ssize_t device_write(struct file *filp, const char *buffer, size_t length,
     loff_t *offset);

static int is_buffer_empty(void);
static int is_buffer_full(void);
static int read_buffer_char(char *buffer);
static int write_buffer_char(char *buffer);

/*
 * Device Operations
```

```
 */

struct file_operations device_fops = {
    NULL,           /* seek */
    device_read,     /* read */
    device_write,    /* write */
    NULL,           /* readdir */
    NULL,           /* poll */
    NULL,           /* ioctl */
    NULL,           /* mmap */
    device_open,     /* open */
    NULL,           /* flush */
    device_release   /* release */
};

/*
 * Module startup/cleanup
 */

int init_module(void)
{
    printk("Loading Mini Buffer Module\n");

    if ((s_nMajor = register_chrdev(0, DEVICE_NAME, &device_fops)) < 0) {
        printk(DEVICE_NAME " : Device registration failed (%d)\n", s_nMajor);
        return s_nMajor;
    }

    printk(DEVICE_NAME " : Device registered with Major Number = %d\n", s_nMajor);
```

```
    printk(DEVICE_NAME " : Buffer allocation size = %d\n", minibuf_bufsize);

    if (minibuf_bufsize <= 0) {
        printk(DEVICE_NAME " : invalid minibuf_bufsize argument\n");
        return -EINVAL;
    }

    if ((s_strBuf = (char *) kmalloc(minibuf_bufsize, GFP_KERNEL)) == NULL) {
        printk(DEVICE_NAME " : Memory allocation error\n");
        return -ENOMEM;
    }

    s_nBufSize = minibuf_bufsize;

    return 0;
}

void cleanup_module(void)
{
    int nRetCode;

    printk("Unloading Mini Buffer Module\n");

    if ((nRetCode = unregister_chrdev(s_nMajor, DEVICE_NAME)) < 0)
        printk(DEVICE_NAME " : Device unregistration failed (%d)\n", nRetCode);

    if (s_strBuf) {
        kfree(s_strBuf);
        s_strBuf = NULL;
        s_nBufSize = 0;
```

```
    }
}

/*
 * Device Operations
 */

int device_open(struct inode *inode, struct file *filp)
{
    printk(DEVICE_NAME " : Device open (%d, %d)\n", MAJOR(inode->i_rdev),
        MINOR(inode->i_rdev));

    if (s_bDeviceOpen) {
        printk(DEVICE_NAME " : Device already open\n");
        return -EBUSY;
    }

    ++s_bDeviceOpen;

    MOD_INC_USE_COUNT;

    return 0;
}

int device_release(struct inode *inode, struct file *filp)
{
    printk(DEVICE_NAME " : Device release (%d, %d)\n", MAJOR(inode->i_rdev),
        MINOR(inode->i_rdev));

    if (!s_bDeviceOpen) {
```

```
        printk(DEVICE_NAME " : Device has not opened\n");
        return -EINVAL;
    }

    --s_bDeviceOpen;

    MOD_DEC_USE_COUNT;

    return 0;
}

ssize_t device_read(struct file *filp, char *buffer, size_t length, loff_t *offset)
{
    int count = 0;

    if (is_buffer_empty()) {       /* end of file */
        printk(DEVICE_NAME " : Read return EOF\n");
        return 0;
    }

    while (!is_buffer_empty() && length > 1) {
        read_buffer_char(buffer);
        ++buffer;
        --length;
        ++count;
    }

    put_user(0, buffer);
    ++count;
```

```
    printk(DEVICE_NAME " : Read %d bytes\n", count);

    return count;
}

ssize_t device_write(struct file *filp, const char *buffer, size_t length,
      loff_t *offset)
{
    int count = 0;

    if (is_buffer_full()) {        /* out of buffer */
        printk(DEVICE_NAME " : Write out of buffer\n");
        return -ENOMEM;
    }

    while (!is_buffer_full() && length > 0) {
        write_buffer_char((char *) buffer);
        ++buffer;
        --length;
        ++count;
    }

    filp->f_pos += count;

    printk(DEVICE_NAME " : Write %d bytes\n", count);

    return count;
}

/*
```

```
 * Buffer Management
 */

int is_buffer_empty(void)
{
   return (s_nBufPos == s_nBufEnd) ? 1 : 0;
}

int is_buffer_full(void)
{
   int pos = s_nBufEnd + 1;

   if (pos == s_nBufSize)
      pos = 0;

   return (pos == s_nBufPos) ? 1 : 0;
}

int read_buffer_char(char *buffer)
{
   if (is_buffer_empty())
      return -1;

   put_user(s_strBuf[s_nBufPos], buffer);

   if (++s_nBufPos == s_nBufSize)
      s_nBufPos = 0;

   return 0;
}
```

```
int write_buffer_char(char *buffer)
{
    if (is_buffer_full())
        return -1;

    get_user(s_strBuf[s_nBufEnd], buffer);

    if (++s_nBufEnd == s_nBufSize)
        s_nBufEnd = 0;

    return 0;
}
```

- Filename : minibuf5.c

```
/*
   Mini Buffer Device Driver : Minor devices
*/

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

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/fs.h>
#include <linux/kdev_t.h>
#include <linux/slab.h>
#include <asm/uaccess.h>

/*
 * Device Definitions
 */

#define DEVICE_NAME           "MiniBuf"
#define DEFAULT_BUFFER_LEN     1024

#define DEVICE_NOBLOCK         0
#define DEVICE_WRITEBLOCK      1
#define DEVICE_BLOCK          2
#define DEVICE_NUM            3
```

```
/*
 * Global Variables
 */

static int s_nMajor = 0;

static struct devicedata_t {
    int m_bDeviceReadOpen;
    int m_bDeviceWriteOpen;
    char *m_strBuf;
    int m_nBufPos, m_nBufEnd;
    int m_nBufSize;
    struct wait_queue *m_writequeue, *m_readqueue;
} s_devicedata[DEVICE_NUM];

int minibuf_bufsize = DEFAULT_BUFFER_LEN;

MODULE_PARM(minibuf_bufsize, "i");
MODULE_PARM_DESC(minibuf_bufsize, "Size of buffer");

/*
 * Function Prototypes
 */

static int device_open(struct inode *inode, struct file *filp);
static int device_release(struct inode *inode, struct file *filp);
static ssize_t device_noblock_read(struct file *filp, char *buffer, size_t length,
     loff_t *offset);
static ssize_t device_noblock_write(struct file *filp, const char *buffer,
```

```
     size_t length, loff_t *offset);
static ssize_t device_writeblock_read(struct file *filp, char *buffer, size_t length,
     loff_t *offset);
static ssize_t device_writeblock_write(struct file *filp, const char *buffer,
     size_t length, loff_t *offset);
static ssize_t device_block_read(struct file *filp, char *buffer, size_t length,
     loff_t *offset);
static ssize_t device_block_write(struct file *filp, const char *buffer,
     size_t length, loff_t *offset);

static int is_buffer_empty(struct devicedata_t *pdata);
static int is_buffer_full(struct devicedata_t *pdata);
static int read_buffer_char(struct devicedata_t *pdata, char *buffer);
static int write_buffer_char(struct devicedata_t *pdata, char *buffer);

/*
 * Device Operations
 */

struct file_operations device_noblock_fops = {
   open:      device_open,
   release:    device_release,
   read:      device_noblock_read,
   write:     device_noblock_write,
};

struct file_operations device_writeblock_fops = {
   open:      device_open,
   release:    device_release,
   read:      device_writeblock_read,
```

```
    write:     device_writeblock_write,
};

struct file_operations device_block_fops = {
    open:      device_open,
    release:    device_release,
    read:      device_block_read,
    write:     device_block_write,
};

struct file_operations *device_fops_array[] = {
    &device_noblock_fops,
    &device_writeblock_fops,
    &device_block_fops
};

/*
 * Module startup/cleanup
 */

int init_module(void)
{
    int i;
    struct devicedata_t *pdata;

    printk("Loading Mini Buffer Module\n");

    if ((s_nMajor = register_chrdev(0, DEVICE_NAME, device_fops_array[0])) < 0) {
        printk(DEVICE_NAME " : Device registration failed (%d)\n", s_nMajor);
        return s_nMajor;
```

```
    }

    printk(DEVICE_NAME " : Device registered with Major Number = %d\n", s_nMajor);

    printk(DEVICE_NAME " : Buffer allocation size = %d\n", minibuf_bufsize);

    if (minibuf_bufsize <= 0) {
        printk(DEVICE_NAME " : invalid minibuf_bufsize argument = %d\n",
            minibuf_bufsize);
        return -EINVAL;
    }

    for (i = 0, pdata = s_devicedata; i < DEVICE_NUM; ++i, ++pdata) {
       if ((pdata->m_strBuf = (char *) kmalloc(minibuf_bufsize, GFP_KERNEL)) == NULL) {
            printk(DEVICE_NAME " : Memory allocation error\n");

            while (--i >= 0)
                kfree(pdata->m_strBuf);

            return -ENOMEM;
        }

        pdata->m_nBufSize = minibuf_bufsize;
    }

    return 0;
}

void cleanup_module(void)
{
```

```
    int i, nRetCode;
    struct devicedata_t *pdata;

    printk("Unloading Mini Buffer Module\n");

    if ((nRetCode = unregister_chrdev(s_nMajor, DEVICE_NAME)) < 0)
        printk(DEVICE_NAME " : Device unregistration failed (%d)\n", nRetCode);

    for (i = 0, pdata = s_devicedata; i < DEVICE_NUM; ++i, ++pdata) {
        if (pdata->m_strBuf) {
            kfree(pdata->m_strBuf);
            pdata->m_strBuf = NULL;
            pdata->m_nBufSize = 0;
        }
    }
}

/*
 * Device Operations
 */

int device_open(struct inode *inode, struct file *filp)
{
    int minor = MINOR(inode->i_rdev);
    struct devicedata_t *pdata = &s_devicedata[minor];

    printk(DEVICE_NAME " : Device open (%d, %d)\n", MAJOR(inode->i_rdev),
        MINOR(inode->i_rdev));

    if ((filp->f_flags & O_ACCMODE) & (O_WRONLY | O_RDWR)) {
```

```
        if (pdata->m_bDeviceWriteOpen) {
            printk(DEVICE_NAME " : Device already open for writing\n");
            return -EBUSY;
        } else
            ++pdata->m_bDeviceWriteOpen;
    } else {
        if (pdata->m_bDeviceReadOpen) {
            printk(DEVICE_NAME " : Device already open for reading\n");
            return -EBUSY;
        } else
            ++pdata->m_bDeviceReadOpen;
    }

    filp->f_op = device_fops_array[minor];

    MOD_INC_USE_COUNT;

    return 0;
}

int device_release(struct inode *inode, struct file *filp)
{
    int minor = MINOR(inode->i_rdev);
    struct devicedata_t *pdata = &s_devicedata[minor];

    printk(DEVICE_NAME " : Device release (%d, %d)\n", MAJOR(inode->i_rdev),
        MINOR(inode->i_rdev));

    if ((filp->f_flags & O_ACCMODE) & (O_WRONLY | O_RDWR))
        --pdata->m_bDeviceWriteOpen;
```

```
    else
        --pdata->m_bDeviceReadOpen;

    MOD_DEC_USE_COUNT;

    return 0;
}

/*
    nonblocking read/write
*/

ssize_t device_noblock_read(struct file *filp, char *buffer, size_t length,
     loff_t *offset)
{
    int count = 0;
    struct devicedata_t *pdata = &s_devicedata[DEVICE_NOBLOCK];

    if (is_buffer_empty(pdata)) {       /* end of file */
        printk(DEVICE_NAME " (NOBLOCK) : Read return EOF\n");
        return 0;
    }

    while (!is_buffer_empty(pdata) && length > 1) {
        read_buffer_char(pdata, buffer);
        ++buffer;
        --length;
        ++count;
    }
```

```
    put_user(0, buffer);
    ++count;

    printk(DEVICE_NAME " (NOBLOCK) : Read %d bytes\n", count);

    return count;
}

ssize_t device_noblock_write(struct file *filp, const char *buffer, size_t length,
     loff_t *offset)
{
    int count = 0;
    struct devicedata_t *pdata = &s_devicedata[DEVICE_NOBLOCK];

    if (is_buffer_full(pdata)) {        /* out of buffer */
        printk(DEVICE_NAME " (NOBLOCK) : Write out of buffer\n");
        return -ENOMEM;
    }

    while (!is_buffer_full(pdata) && length > 0) {
        write_buffer_char(pdata, (char *) buffer);
        ++buffer;
        --length;
        ++count;
    }

    filp->f_pos += count;

    printk(DEVICE_NAME " (NOBLOCK) : Write %d bytes\n", count);
```

```
    return count;
}

/*
    blocking on write read/write
*/

ssize_t device_writeblock_read(struct file *filp, char *buffer, size_t length,
     loff_t *offset)
{
    int count = 0;
    struct devicedata_t *pdata = &s_devicedata[DEVICE_WRITEBLOCK];

    if (is_buffer_empty(pdata))        /* end of file */
        return 0;

    while (!is_buffer_empty(pdata) && length > 1) {
        read_buffer_char(pdata, buffer);
        ++buffer;
        --length;
        ++count;
        wake_up_interruptible(&pdata->m_writequeue);
    }

    put_user(0, buffer);
    ++count;

    printk(DEVICE_NAME " (WRITEBLOCK) : Read %d bytes\n", count);

    return count;
```

```
}

ssize_t device_writeblock_write(struct file *filp, const char *buffer, size_t length,
     loff_t *offset)
{
    int count = 0;
    struct devicedata_t *pdata = &s_devicedata[DEVICE_WRITEBLOCK];

    if (is_buffer_full(pdata)) {
        printk(DEVICE_NAME " (WRITEBLOCK) : Write go to sleep\n");
        interruptible_sleep_on(&pdata->m_writequeue);
        printk(DEVICE_NAME " (WRITEBLOCK) : Write wake up\n");
    }

    while (!is_buffer_full(pdata) && length > 0) {
        write_buffer_char(pdata, (char *) buffer);
        ++buffer;
        --length;
        ++count;
    }

    filp->f_pos += count;

    printk(DEVICE_NAME " (WRITEBLOCK) : Write %d bytes\n", count);

    return count;
}

/*
    blocking on read read/write
```

```
*/

ssize_t device_block_read(struct file *filp, char *buffer, size_t length,
     loff_t *offset)
{
    int count = 0;
    struct devicedata_t *pdata = &s_devicedata[DEVICE_BLOCK];

    if (is_buffer_empty(pdata)) {      /* end of file */
        printk(DEVICE_NAME " (BLOCK) : Read go to sleep\n");
        interruptible_sleep_on(&pdata->m_readqueue);
        printk(DEVICE_NAME " (BLOCK) : Read wake up\n");
    }

    while (!is_buffer_empty(pdata) && length > 1) {
        read_buffer_char(pdata, buffer);
        ++buffer;
        --length;
        ++count;
        wake_up_interruptible(&pdata->m_writequeue);
    }

    put_user(0, buffer);
    ++count;

    printk(DEVICE_NAME " : Read %d bytes\n", count);

    return count;
}
```

```
ssize_t device_block_write(struct file *filp, const char *buffer, size_t length,
     loff_t *offset)
{
   int count = 0;
   struct devicedata_t *pdata = &s_devicedata[DEVICE_BLOCK];

   if (is_buffer_full(pdata)) {
      printk(DEVICE_NAME " (BLOCK) : Write go to sleep\n");
      interruptible_sleep_on(&pdata->m_writequeue);
      printk(DEVICE_NAME " (BLOCK) : Write wake up\n");
   }

   while (!is_buffer_full(pdata) && length > 0) {
      write_buffer_char(pdata, (char *) buffer);
      ++buffer;
      --length;
      ++count;
      wake_up_interruptible(&pdata->m_readqueue);
   }

   filp->f_pos += count;

   printk(DEVICE_NAME " (BLOCK) : Write %d bytes\n", count);

   return count;
}

/*
 * Buffer Management
 */
```

```
int is_buffer_empty(struct devicedata_t *pdata)
{
   return (pdata->m_nBufPos == pdata->m_nBufEnd) ? 1 : 0;
}

int is_buffer_full(struct devicedata_t *pdata)
{
   int pos = pdata->m_nBufEnd + 1;

   if (pos == pdata->m_nBufSize)
      pos = 0;

   return (pos == pdata->m_nBufPos) ? 1 : 0;
}

int read_buffer_char(struct devicedata_t *pdata, char *buffer)
{
   if (is_buffer_empty(pdata))
      return -1;

   put_user(pdata->m_strBuf[pdata->m_nBufPos], buffer);

   if (++pdata->m_nBufPos == pdata->m_nBufSize)
      pdata->m_nBufPos = 0;

   return 0;
}

int write_buffer_char(struct devicedata_t *pdata, char *buffer)
```

```
{
    if (is_buffer_full(pdata))
        return -1;

    get_user(pdata->m_strBuf[pdata->m_nBufEnd], buffer);

    if (++pdata->m_nBufEnd == pdata->m_nBufSize)
        pdata->m_nBufEnd = 0;

    return 0;
}
```

- 메모리의 동적인 할당과 해제 : (include/linux/slab.h)

  `void *kmalloc(size_t, int);`

  지정한 크기만큼의 메모리를 할당하여 돌려준다.

  `void kfree(const void *);`

  `kmalloc()`으로 할당받은 메모리를 해제한다.

- 모듈에서 parameter 전달하기 :

  - `MODULE_PARM(var, type)` 매크로로 parameter 로 사용할 변수를 지정

    var : 변수명. global 변수가 아니어도 상관이 없다.

    type : 문자열로 [min[-max]]{b,h,i,l,s} 의 형식을 갖는다.

    b : byte, h : short, i : int, l : long, s : string

  - `MODULE_PARM_DESC(var, description)` 매크로로 parameter 에 대한 설명을 한다

  - parameter 지정하기 :

    insmod 를 할 때 var=value 형식으로 값을 지정한다.

    insmod 프로그램은 모듈내의 지정한 변수를 해당하는 값으로 치환한다.

  - 2.0.x 버전에서는 export 된 모든 변수를 parameter 를 통해 값을 바꿀 수 있었다.

- minor 번호에 따라 다른 동작을 하는 디바이스 드라이버 :

  - open 과 release 시에 전달되는 `inode` 자료구조의 `i_rdev` 항목을 통해서 open 된 장치의 minor 번호를 알 수 있다.

  - open 을 할 때 `file` 자료구조의 file operation 을 minor 번호에 따라서 다르게 지정함으로써, 또는 file operation 내의 일부 함수들을 바꿈으로써 쉽게 다른 동작을 하도록 한다.

5. 네번째 프로그램 : proc 파일시스템 활용

- 목적 : 먼저 proc 파일시스템에 현재 프로세스의 정보를 보여주는 파일을 추가한다. 다음으로 minibuf 에서처럼 파일에 데이터를 기록하고 읽을 수 있는 파일을 proc 파일시스템에 추가한다.

- 구현 방법 : proc 파일시스템 인터페이스를 이용하여 `/proc` 밑에 `curproc` 이라는 파일을 추가하고, 이 파일에서 읽기를 시도하면 이를 실행한 프로세스의 현재 정보를 동적으로 만들어서 보여준다. 여기서는 `proc_dir_entry` 의 `read_proc` 함수 포인터를 이용하여 간단하게 구현한다. 이 방법은 구현하기는 쉽지만 교환할 수 있는 데이터의 크기가 `PAGE_SIZE` 를 넘을 수 없다는 단점이 있다. 이 크기를 넘는 데이터 교환은 완전한 파일 연산을 구현하여야 한다. 두번째는 `/proc` 밑에 `bufproc` 이라는 파일을 추가하고 이 파일에 기록을 하면 기록한 내용을 내부 버퍼에 저장을 하고, 이 파일에서 읽어들이면 마지막에 기록한 내용을 보여준다. 여기서는 inode operation 과 file operation 을 이용하여 구현한다. 여기서는 앞에서와 같은 `PAGE_SIZE` 제한이 없다.

- 테스트 방법 : 첫번째 모듈을 로드한 후 cat 명령을 내려서 현재 프로세스의 정보가 나오는지 살펴본다. 두번째 모듈을 로드한 후 cat 명령을 이용하여 파일에 데이터를 기록한 후 이를 읽어들였을 때 전에 입력한 내용이 출력되는지 살펴본다.

- 과정 :
  - proc 파일시스템에 curproc 을 추가하고 테스트하기
  - proc 파일시스템의 기능을 완전히 활용한 read/write 가능한 bufproc 파일 만들기

- 테스트 과정 :

1) curproc :

```
cat /proc/curproc
```

2) bufproc :

```
cat > /proc/bufproc
cat /proc/bufproc
```

- **Filename : curproc.c**

```
/*
   Current Process Information
*/

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

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/proc_fs.h>

/*
 * Function Prototypes
 */

static int curproc_read_proc(char *buf, char **start, off_t offset, int count,
     int *eof, void *data);

/*
 * Proc Entry
 */

struct proc_dir_entry curproc_proc_entry = {
     0,                  /* low inode */
     7, "curproc",       /* name length and name" */
```

```
     S_IFREG | S_IRUGO,   /* mode */
     1, 0, 0,            /* nlink, uid, gid */
     0,                  /* size */
     NULL,               /* inode operation : NULL = use default */
     NULL,               /* get_info */
     NULL,               /* fill_inode */
     NULL, NULL, NULL,   /* next, parent, subdir */
     NULL,               /* data */
     curproc_read_proc,  /* read_proc */
     NULL,               /* write_proc */
     NULL,               /* readlink_proc */
     /* nothing more */
};

/*
 * Module Startup/Cleanup
 */

int init_module(void)
{
     int retcode;

     printk("Loading CurProc Module\n");

     if ((retcode = proc_register(&proc_root, &curproc_proc_entry)) < 0)
          return retcode;

     return 0;
}
```

```
void cleanup_module(void)
{
     printk("Unloading CurProc Module\n");

     proc_unregister(&proc_root, curproc_proc_entry.low_ino);
}

int curproc_read_proc(char *buf, char **start, off_t offset, int count, int *eof,
     void *data)
{
     static char *s_strState[] = {
          "Running", "Sleep", "Disk Sleep", "Zombie", "Stop", "Swap", "Exclusive" };

     int state, stateid;

     count = 0;

     for (stateid = 0, state = current->state; state > 0; state >>= 1, ++stateid)
          ;

     count += sprintf(buf + count, "pid : %d\n", current->pid);
     count += sprintf(buf + count, "command : %s\n", current->comm);
     count += sprintf(buf + count, "state : %s\n", s_strState[stateid]);
     count += sprintf(buf + count, "priority : %ld\n", current->priority);
     count += sprintf(buf + count, "counter : %ld\n", current->counter);
     count += sprintf(buf + count, "processor : %d\n", current->processor);

     return count;
}
```

- **Filename : bufproc.c**

```
/*
    Read/Write Proc Filesystem
*/

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

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/proc_fs.h>
#include <asm/uaccess.h>

/*
 * Global Variables
 */

#define BUFFER_LEN            1024

static char s_strBuf[BUFFER_LEN];

/*
 * Function Prototypes
 */

static int bufproc_open(struct inode *inodep, struct file *filp);
```

```
static int bufproc_release(struct inode *inodep, struct file *filp);
static ssize_t bufproc_read(struct file *filp, char *buf, size_t count, loff_t *ppos);
static ssize_t bufproc_write(struct file *filp, const char *buf, size_t count, loff_t *ppos);

static int bufproc_permission(struct inode * inodep, int op);

/*
 * Proc Entry
 */

static struct file_operations bufproc_file_operations = {
   open:      bufproc_open,
   release:    bufproc_release,
   read:      bufproc_read,
   write:     bufproc_write,
};

static struct inode_operations bufproc_inode_operations = {
   default_file_ops:   &bufproc_file_operations,
   permission:         bufproc_permission,
};

struct proc_dir_entry bufproc_proc_entry = {
   0,                              /* low inode */
   7, "bufproc",                   /* name length and name" */
   S_IFREG | S_IRUGO | S_IWUSR,  /* mode */
   1, 0, 0,                        /* nlink, uid, gid */
   0,                              /* size */
   &bufproc_inode_operations,    /* inode operation */
   /* nothing more */
};
```

```
/*
 * Module Startup/Cleanup
 */

int init_module(void)
{
    int retcode;

    printk("Loading BufProc Module\n");

    if ((retcode = proc_register(&proc_root, &bufproc_proc_entry)) < 0)
        return retcode;

    return 0;
}

void cleanup_module(void)
{
    printk("Unloading BufProc Module\n");

    proc_unregister(&proc_root, bufproc_proc_entry.low_ino);
}

/*
 * File Operations
 */

int bufproc_open(struct inode *inodep, struct file *filp)
{
    MOD_INC_USE_COUNT;
```

```
    return 0;
}

int bufproc_release(struct inode *inodep, struct file *filp)
{
    MOD_DEC_USE_COUNT;

    return 0;
}

ssize_t bufproc_read(struct file *filp, char *buf, size_t count, loff_t *ppos)
{
    static int s_bRead = 0;

    int len = strlen(s_strBuf);

    if (len == 0 || s_bRead) {
        s_bRead = 0;
        return 0;
    }

    if (len > count - 1)
        len = count - 1;

    copy_to_user(buf, s_strBuf, len);
    put_user(0, &buf[len]);
    ++len;

    s_bRead = 1;

    return len;
```

```
}

ssize_t bufproc_write(struct file *filp, const char *buf, size_t count, loff_t *ppos)
{
   int len = (count > BUFFER_LEN - 1) ? BUFFER_LEN - 1 : count;

   if (count == 0)
      return 0;

   copy_from_user(s_strBuf, buf, len);
   s_strBuf[len] = '\0';

   return len;
}

/*
 * Inode Operations
 */

/*
   op : 0 = execute, 2 = write, 4 = read
   return : 0 = permission ok, -EACCES = permission error
*/

int bufproc_permission(struct inode * inodep, int op)
{
   if (op == 4 || (op == 2 && current->euid == 0))
      return 0;

   return -EACCES;
}
```

- proc 파일시스템이란 :
  - 실제 디스크가 아니라 메모리상에 존재하는 파일시스템
  - 프로세스에게 정보 전달, 사용자에게 시스템의 자세한 상태 전달하기 위해 만들어짐.
  - 데이터는 실제 읽기를 하는 시점에 만들어진다.

- proc 파일시스템의 예 :
  `/proc/cpuinfo` : 시스템에 설치된 CPU 에 대한 정보를 보여준다.
  `/proc/devices` : 로드된 디바이스 드라이버의 목록을 보여준다.
  `/proc/interrupts` : 현재 사용중인 인터럽트의 목록을 보여준다.
  `/proc/meminfo` : 메모리 사용 상황을 보여준다.
  `/proc/sys/kernel/hostname` : hostname 을 보여주고 바꿀 수 있게 한다.
  `/proc/sys/vm/freepages` : minimum, low, high free page 의 값을 읽고 쓸 수 있게 한다.

- proc 파일시스템 등록과 해제

```
int proc_register(struct proc_dir_entry *, struct proc_dir_entry *);
int proc_unregister(struct proc_dir_entry *, int);
```

  /proc 디렉토리의 proc_dir_entry : proc_root
  /proc/sys 디렉토리의 proc_dir_entry : proc_sys

- struct proc_dir_entry

```
unsigned short low_ino;
unsigned short namelen;
const char *name;
mode_t mode;
nlink_t nlink;
uid_t uid;
gid_t gid;
unsigned long size;
struct inode_operations * ops;
int (*get_info)(char *, char **, off_t, int, int);
void (*fill_inode)(struct inode *, int);
struct proc_dir_entry *next, *parent, *subdir;
void *data;
int (*read_proc)(char *page, char **start, off_t off, int count,
     int *eof, void *data);
int (*write_proc)(struct file *file, const char *buffer,
     unsigned long count, void *data);
int (*readlink_proc)(struct proc_dir_entry *de, char *page);
unsigned int count;     /* use count */
int deleted;            /* delete flag */
```

- struct inode_operation

  struct file_operations * default_file_ops;

  이 inode 에서 사용할 기본 file_operations

  ```
  int (*create) (struct inode *,struct dentry *,int);
  int (*readpage) (struct file *, struct page *);
  int (*writepage) (struct file *, struct page *);
  ```

  int (*permission) (struct inode *, int);

  프로세스가 파일에 대해 어떤 일을 하려 할 때 불리며, 접근 허가 여부를 결정

  현재 uid 와 파일 상태, 기타 여러가지를 가지고 결정

- proc_dir_entry 에 등록하는 mode

  - 파일의 속성을 나타낸다.
  - 파일의 종류 :

    S_IFREG : regular file

    S_IFLNK : link

    S_IFBLK : block device special file

    S_IFCHR : character device special file

    S_IFDIR : directory

    S_IFFIFO : named pipe (FIFO)

  - user permission : S_IRUSR, S_IWUSR, S_IXUSR => S_IRWXU
  - group permssion : S_IRGRP, S_IWGRP, S_IXGRP => S_IRWXG
  - other permission : S_IROTH, S_IWOTH, S_IXOTH => S_RWXO

5. 다섯번째 프로그램 : ioctl 활용하기

- 목적 : 디바이스 드라이버에서 read/write 외의 방법으로 장치를 제어할 수 있는 ioctl 을 구현한다. minibuf 에서 버퍼내의 데이터와, 데이터 길이를 얻고, 버퍼를 비우는 ioctl 명령을 추가하고, 디바이스 드라이버를 이용하는 프로그램에서는 ioctl 명령을 내려서 장치를 제어한다.

- 구현방법 : minibuf3.c 소스를 수정하여 작성한다. 우선 ioctl 명령을 정의하여 이를 헤더 파일에 넣고, 디바이스 드라이버의 file operation 에 ioctl 명령을 추가한다. ioctl 명령을 처리하는 함수는 자신에게 주어진 명령에 따라 해당 작업을 처리한다. 이후 디바이스 드라이버를 이용하는 일반 C 프로그램을 작성하여 장치에 데이터를 입력하고, ioctl 명령을 내린다.

- 테스트 방법 : 테스트 프로그램을 통해 장치에 데이터를 입력하고 ioctl 명령으로 데이터를 확인하고, 데이터의 길이를 얻어온 후 버퍼를 비우고, 다시 데이터의 길이를 얻어온다. 여기서 결과가 원하는데로 나오는지 살펴본다.

- 과정 :
  - ioctl 을 처리하는 디바이스 드라이버 제작
  - 디바이스 드라이버의 ioctl 을 이용하는 테스트 프로그램 제작

- 테스트 과정 :

```
insmod minibuf6.o
./minibuf_test
rmmod minibuf6
```

- **Filename : minibuf.h**

```
/*
   MiniBuf.h
*/

#ifndef __MINIBUF_H
#define __MINIBUF_H

#include <linux/ioctl.h>

#define MINIBUF_MAGICNUM          254

struct minibuf_peek_t {
   char *m_buf;
   int m_maxlen;
};

#define IOCTL_MINIBUF_MAKEEMPTY    _IO(MINIBUF_MAGICNUM, 0)
#define IOCTL_MINIBUF_GETLENGTH    _IOR(MINIBUF_MAGICNUM, 1, int *)
#define IOCTL_MINIBUF_PEEK         _IOR(MINIBUF_MAGICNUM, 2, struct minibuf_peek_t *)

#endif
```

- **Filename : minibuf6.c**

```
/*
   Mini Buffer Device Driver : ioctl
*/

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

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/fs.h>
#include <linux/kdev_t.h>
#include <asm/uaccess.h>

#include "minibuf.h"

/*
 * Device Definitions
 */

#define DEVICE_NAME           "MiniBuf"
#define BUFFER_LEN            1024

/*
 * Global Variables
 */
```

```
static int s_nMajor = 0;

static int s_bDeviceReadOpen = 0;
static int s_bDeviceWriteOpen = 0;
static char s_strBuf[BUFFER_LEN];
static int s_nBufPos = 0, s_nBufEnd = 0;

static struct wait_queue *s_wq = NULL;

/*
 * Function Prototypes
 */

static int device_open(struct inode *inode, struct file *filp);
static int device_release(struct inode *inode, struct file *filp);
static ssize_t device_read(struct file *filp, char *buffer, size_t length,
     loff_t *offset);
static ssize_t device_write(struct file *filp, const char *buffer, size_t length,
     loff_t *offset);
static int device_ioctl(struct inode *inodep, struct file *filp, unsigned int cmd,
     unsigned long arg);

static int peek_buffer(struct minibuf_peek_t *pdata);

static int is_buffer_empty(void);
static int is_buffer_full(void);
static int read_buffer_char(char *buffer);
static int write_buffer_char(char *buffer);
static void make_buffer_empty(void);
```

```
static int get_buffer_length(void);

/*
 * Device Operations
 */

struct file_operations device_fops = {
    NULL,          /* seek */
    device_read,    /* read */
    device_write,   /* write */
    NULL,          /* readdir */
    NULL,          /* select */
    device_ioctl,   /* ioctl */
    NULL,          /* mmap */
    device_open,    /* open */
    NULL,          /* flush */
    device_release  /* release */
};

/*
 * Module startup/cleanup
 */

int init_module(void)
{
    printk("Loading Mini Buffer Module\n");

    if ((s_nMajor = register_chrdev(0, DEVICE_NAME, &device_fops)) < 0) {
        printk(DEVICE_NAME " : Device registration failed (%d)\n", s_nMajor);
        return s_nMajor;
```

```
    }

    printk(DEVICE_NAME " : Device registered with Major Number = %d\n", s_nMajor);

    return 0;
}

void cleanup_module(void)
{
    int nRetCode;

    printk("Unloading Mini Buf Module\n");

    if ((nRetCode = unregister_chrdev(s_nMajor, DEVICE_NAME)) < 0)
        printk(DEVICE_NAME " : Device unregistration failed (%d)\n", nRetCode);
}

/*
 * Device Operations
 */

int device_open(struct inode *inode, struct file *filp)
{
    printk(DEVICE_NAME " : Device open (%d, %d)\n", MAJOR(inode->i_rdev),
        MINOR(inode->i_rdev));

    if ((filp->f_flags & O_ACCMODE) & (O_WRONLY | O_RDWR)) {
        if (s_bDeviceWriteOpen) {
            printk(DEVICE_NAME " : Device already open for writing\n");
            return -EBUSY;
```

```
        } else
            ++s_bDeviceWriteOpen;
    } else {
        if (s_bDeviceReadOpen) {
            printk(DEVICE_NAME " : Device already open for reading\n");
            return -EBUSY;
        } else
            ++s_bDeviceReadOpen;
    }

    MOD_INC_USE_COUNT;

    return 0;
}

int device_release(struct inode *inode, struct file *filp)
{
    printk(DEVICE_NAME " : Device release (%d, %d)\n", MAJOR(inode->i_rdev),
        MINOR(inode->i_rdev));

    if ((filp->f_flags & O_ACCMODE) & (O_WRONLY | O_RDWR))
        --s_bDeviceWriteOpen;
    else
        --s_bDeviceReadOpen;

    MOD_DEC_USE_COUNT;

    return 0;
}
```

```
ssize_t device_read(struct file *filp, char *buffer, size_t length, loff_t *offset)
{
   int count = 0;

   if (is_buffer_empty())        /* end of file */
      return 0;

   while (!is_buffer_empty() && length > 1) {
      read_buffer_char(buffer);
      ++buffer;
      --length;
      ++count;
      wake_up_interruptible(&s_wq);
   }

   put_user(0, buffer);
   ++count;

   printk(DEVICE_NAME " : Read %d bytes\n", count);

   return count;
}

ssize_t device_write(struct file *filp, const char *buffer, size_t length,
     loff_t *offset)
{
   int count = 0;

   if (is_buffer_full()) {
      printk(DEVICE_NAME " : Write go to sleep\n");
```

```
        interruptible_sleep_on(&s_wq);
        printk(DEVICE_NAME " : Write wake up\n");
    }

    while (!is_buffer_full() && length > 0) {
        write_buffer_char((char *) buffer);
        ++buffer;
        --length;
        ++count;
    }

    filp->f_pos += count;

    printk(DEVICE_NAME " : Write %d bytes\n", count);

    return count;
}

int device_ioctl(struct inode *inodep, struct file *filp, unsigned int cmd,
      unsigned long arg)
{
    int length;

    printk(DEVICE_NAME " : ioctl, cmd = %x, arg = %x\n", cmd, arg);

    switch (cmd) {
    case IOCTL_MINIBUF_MAKEEMPTY :
        printk(DEVICE_NAME " : ioctl : make empty\n");
        make_buffer_empty();
        break;
```

```
    case IOCTL_MINIBUF_GETLENGTH :
        printk(DEVICE_NAME " : ioctl : get length\n");
        length = get_buffer_length();
        put_user(length, (int *) arg);
        break;
    case IOCTL_MINIBUF_PEEK :
        printk(DEVICE_NAME " : ioctl : peek\n");
        return peek_buffer((struct minibuf_peek_t *) arg);
    default :
        printk(DEVICE_NAME " : ioctl : unknown\n");
        return -EINVAL;
    }

    return 0;
}

/*
 * Higher level buffer management
 */

int peek_buffer(struct minibuf_peek_t *pdata)
{
    int count = 0, nBufPos;
    int maxlen;
    char *buffer;

    if (is_buffer_empty())        /* end of file */
        return 0;

    get_user(buffer, &pdata->m_buf);
```

```
    get_user(maxlen, &pdata->m_maxlen);

    nBufPos = s_nBufPos;

    while (!is_buffer_empty() && maxlen > 1) {
        read_buffer_char(buffer);
        ++buffer;
        --maxlen;
        ++count;
    }

    put_user(0, buffer);
    ++count;

    s_nBufPos = nBufPos;

    printk(DEVICE_NAME " : Peek %d bytes\n", count);

    return count;
}

/*
 * Buffer Management
 */

int is_buffer_empty(void)
{
    return (s_nBufPos == s_nBufEnd) ? 1 : 0;
}
```

```
int is_buffer_full(void)
{
   int pos = s_nBufEnd + 1;

   if (pos == BUFFER_LEN)
      pos = 0;

   return (pos == s_nBufPos) ? 1 : 0;
}

int read_buffer_char(char *buffer)
{
   if (is_buffer_empty())
      return -1;

   put_user(s_strBuf[s_nBufPos], buffer);

   if (++s_nBufPos == BUFFER_LEN)
      s_nBufPos = 0;

   return 0;
}

int write_buffer_char(char *buffer)
{
   if (is_buffer_full())
      return -1;

   get_user(s_strBuf[s_nBufEnd], buffer);
```

```
    if (++s_nBufEnd == BUFFER_LEN)
        s_nBufEnd = 0;

    return 0;
}

void make_buffer_empty(void)
{
    s_nBufPos = s_nBufEnd = 0;
}

int get_buffer_length(void)
{
    int pos = s_nBufEnd;

    if (pos < s_nBufPos)
        pos += BUFFER_LEN;

    return pos - s_nBufPos;
}
```

- **Filename : minibuf_test.c**

```
/*
   Mini Buffer Test : ioctl
*/

#include <stdio.h>
#include <fcntl.h>

#include "minibuf.h"

#define BUFFER_LEN           1024

static void minibuf_read(int fd);
static void minibuf_write(int fd, char *str);
static void minibuf_peek(int fd);
static void minibuf_ioctl_test(int fd);

static int minibuf_ioctl_get_length(int fd);
static int minibuf_ioctl_make_empty(int fd);
static int minibuf_ioctl_peek(int fd, char *buffer, int length);

int main(void)
{
   int fd;

   if ((fd = open("/dev/minibuf", O_RDWR)) < 0) {
      printf("Error opening device file\n");
      return 1;
   }
```

```
    minibuf_read(fd);
    minibuf_write(fd, "Hello, World");
    minibuf_peek(fd);
    minibuf_ioctl_test(fd);
    minibuf_read(fd);

    close(fd);

    return 0;
}

void minibuf_read(int fd)
{
    int len;
    char str[BUFFER_LEN];

    while ((len = read(fd, str, BUFFER_LEN)) > 0)
        printf("READ : %s (%d)\n", str, len);
}

void minibuf_write(int fd, char *str)
{
    int len;

    if ((len = write(fd, str, strlen(str))) > 0)
        printf("WRITE : %s (%d)\n", str, len);
}

void minibuf_peek(int fd)
```

```
{
    int len;
    char str[BUFFER_LEN];

    if ((len = minibuf_ioctl_peek(fd, str, BUFFER_LEN)) >= 0)
        printf("PEEK : %s (%d)\n", str, len);
    else
        printf("PEEK : Error\n");
}

void minibuf_ioctl_test(int fd)
{
    int length;

    length = minibuf_ioctl_get_length(fd);
    printf("Buffer Length = %d\n", length);
    minibuf_ioctl_make_empty(fd);
    printf("Make buffer empty\n");
    length = minibuf_ioctl_get_length(fd);
    printf("Buffer Length = %d\n", length);
}

int minibuf_ioctl_get_length(int fd)
{
    int length;

    return (ioctl(fd, IOCTL_MINIBUF_GETLENGTH, &length) >= 0) ? length : -1;
}

int minibuf_ioctl_make_empty(int fd)
```

```
{
    return ioctl(fd, IOCTL_MINIBUF_MAKEEMPTY, 0) >= 0 ? 0 : -1;
}

int minibuf_ioctl_peek(int fd, char *buffer, int length)
{
    struct minibuf_peek_t peek_data = { buffer, length };

    return ioctl(fd, IOCTL_MINIBUF_PEEK, &peek_data);
}
```

- ioctl (input output control)
  - read/write 외에 장치를 제어할 수 있는 방법
  - 대부분의 장치는 read/write 만으로 제어를 다 할 수는 없다.
  - 장치에게 여러가지 다양한 명령을 내릴 수 있는 장치 고유의 entry point 제공
  - user programming:

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

  - file operation interface:

```
int (*ioctl) (struct inode *inodep, struct file *filp,
    unsigned int cmd, unsigned long arg);
```

  - cmd : (include/asm/ioctl.h)

    ioctl 로 실행할 명령어를 구분하는 값

    실수로 해당 명령을 다른 디바이스 드라이버에게 내려도 문제가 발생하지 않도록 복잡한 방법으로 생성하여, 시스템 전체에서 중복되는 경우를 가능한 피한다.

  - ioctl cmd 숫자 만들기 : `_IOC(direction, type, number, size)`

    `direction` : 데이터의 전송방향

    `_IOC_NONE`, `_IOC_READ`, `_IOC_WRITE`

    `type` : magic number. 가능한 디바이스 드라이버마다 고유한 번호로

    `number` : 각 ioctl 명령을 구분할 수 있는 숫자

    `size` : 전송할 데이터의 크기

```
#define _IO(type,nr)        _IOC(_IOC_NONE,(type),(nr),0)
#define _IOR(type,nr,size)  _IOC(_IOC_READ,(type),(nr),sizeof(size))
#define _IOW(type,nr,size)  _IOC(_IOC_WRITE,(type),(nr),sizeof(size))
#define _IOWR(type,nr,size) _IOC(_IOC_READ|_IOC_WRITE,(type),(nr),sizeof(size))
```

- predefined command :

FIOCLEX : set close-on-exec flag

FIONCLEX : clear close-on-exec flag

FIOASYNC : set/reset asynchronous write for the file

FIONBIO : nonblocking I/O

`filp->f_flags` 에서 O_NONBLOCK flag 를 바꾼다.

```
fcntl(fd, F_SETFTL, fcntl(fd, F_GETTFL) | O_NONBLOCK));
```

- ioctl 처리 : ioctl handler 함수에서 넘어온 cmd 명령에 따라서 switch 문을 통해 처리

```
switch (cmd) {
case MYDRV_IORESET :
   ......
case MYDRV_IOCLEAR :
   ......
default :
   return -EINVAL;
}
```

6. 디버깅 테크닉

- printk()를 이용한 디버깅
  - 가장 손쉬운 방법으로 필요할 때 메시지를 출력하여 동작을 확인한다.
  - loglevel :
    메시지의 등급을 설정하여, 필요한 등급의 메시지만 출력하는 것을 가능하게 한다.
    메시지 시작에 <1> 같은 방법으로 지정.
    loglevel 을 지정하지 않으면 DEFAULT_MESSAGE_LOGLEVEL 에 해당한다.
    console_loglevel 보다 낮은 loglevel 의 문장만 출력된다.
    sys_syslog system call 이나 klogd -c 명령으로 console_loglevel 을 변경.
  - 메시지 출력 과정 :
    printk()는 내부 원형 버퍼에 메시지를 기록한다. (overflow 가능)
    klogd 프로세스는 정기적으로 커널 메시지를 받아와서 syslogd 에게 전달한다.
    syslogd 는 /etc/syslog.conf 파일의 설정에 따라 커널 메시지의 처리 방법 결정한다. 기본 설정은 메시지를 /var/log/messages 에 저장하는 것이다.
    klogd -f <filename> 명령으로 커널 메시지를 별도의 파일에 저장할 수 있다.
  - 디버깅 모드에서만 메시지를 출력하게 하기 :
    DEBUG 정의가 되어 있을 때만 메시지 출력 (-DDEBUG 로 컴파일)

    방법 1)

```
#ifdef DEBUG
printk(.................)
#endif
```

방법 2)

```
#ifdef DEBUG
#define dprintk(fmt, args...) printk(KERN_DEBUG " : " fmt, ##args)
#else
#define dprintk(fmt, args...)
#endif
```

- 필요할 때에만 정보 출력하기
  - printk()는 필요하지 않은 메시지를 많이 출력하게 하며, 많이 사용하면 시스템이 느려진다.
  - proc 파일시스템 활용하기 :
    사용자가 요구한 시점에 정보를 만들어서 사용자에게 보여준다.
    현재 모듈의 상태, 내부 데이터를 보여줄 수 있어 디버깅을 용이하게 한다.
  - ioctl 활용하기 :
    ioctl 에 디버깅을 위한 명령어들을 추가한다.
    디버깅 정보가 해당 ioctl 명령을 아는 사람이 아니면 노출되지 않는다.
    proc 파일시스템보다 속도가 빠르며 사용방법에 제약이 없다.

- 사용자 프로그램을 사용하는 과정에서 문제 확인하기 :
  - 디버거로 사용자 프로그램을 추적한다.
  - 시스템 콜이나 드라이버를 사용하는 부분에 printf()문으로 정보를 출력한다.
  - strace 프로그램을 이용하여 시스템 함수가 불리는 것을 검사한다.
    시스템 콜에 전달된 인자과 그 결과를 보여준다.
    `strace cat /dev/minibuf`

- oops
  - 커널에 문제가 발생하여 커널이 죽을 때 (예: null pointer access) oops 메시지를 출력한다.
  - oops 메시지는 에러가 발생한 주소와 레지스터 상태, stack 상태 등 만을 출력한다.
  - ksymoops 프로그램을 이용하여 oops 메시지를 해석해서 문제가 발생한 곳을 알수있다.

    ```
    ksymoops /boot/System.map < oops
    ```

  - 단지 에러의 위치만 대강 찾을 수 있을 뿐, 정확한 코드의 위치를 알려면 어셈블리 코드를 추적해야 하고, 당시의 메모리 상태 등을 보여주지 않으며, 시스템 이상으로 문제가 발생했을 때 화면에 oops 메시지를 출력하지 못하거나, 너무 많은 oops 메시지를 출력하여 메시지가 지나가버릴 수도 있다.
  - `System.map` :

    export 된 심볼뿐만 아니라 커널에 들어 있는 모든 심볼들에 대한 정보를 가지고 있다. 커널을 컴파일할 때 만들어진다.
  - 커널과 모듈을 포함한 심볼 목록 만들기 :

    `System.map` 에는 로드된 모듈의 심볼이 들어있지 않으며, `/proc/ksyms` 에는 모듈의 심볼이 들어있지만 커널에서 export 되지 안은 심볼들에 대한 정보가 없다.

    cat /proc/ksyms /boot/System.map | sed 's/ . / /' | awk '{print $1, "T", $2}' | sort -u > symbols.map

- Kernel Dumper
  - kmsgdump : 커널이 죽을 때 최근의 커널 메시지와 oops 메시지를 플로피에 저장한다.
  - mcore : 커널이 죽을 때의 이미지(core)를 압축하여 RAM 에 저장을 하고, 이미 메모리에 로드되어 있던 새로운 커널 이미지로 reboot 를 한다. reboot 후에 이전에 RAM 에 저장한 core 를 추가된 system call 을 이용하여 압축을 풀고 디스크에 저장할 수 있다.
    - 장점 :

interrupt context 에서도 사용할 수 있다.
RAM 에 저장을 하기 때문에 빠르다.

- 단점 :
  설치가 어렵다.
  reboot 시에 메모리 내용이 지워져서는 안된다.

- LKCD (Linux Kernel Core Dumper) : SCSI 디스크의 기능을 활용하여 커널이 죽을 때 core 를 swap partition 에 저장을 한다. SCSI driver 의 문제 때문에 interrupt 쪽에서 에러가 발생한 경우 사용할 수 없다.

- kdb
  - 커널에 link 된 kernel debugger
  - CPU 에서 fault 가 발생하거나, PAUSE 키를 누르거나, serial console 을 통해서 호출된다.
  - keyboard interrupt 나 serial interrupt 안에서 처리된다.
  - single step, breakpoint 를 이용한 디버깅이 가능
  - 커널의 symbol 뿐만 아니라 module 의 symbol 도 알고 있어 모듈 디버깅도 가능하다.
  - stack 에서 frame pointer 를 이용하여 stack trace 를 할 수도 있다.

- kgdb
  - remote gdb 프로그램을 통해서 커널의 행동을 제어한다.
  - 두 대의 기계를 serial 로 연결하고 한쪽 기계에 디버깅할 커널을 다른 기계에서 gdb 를 돌린다.
  - 커널을 debugging symbol 을 포함하여 컴파일하고 frame pointer 를 이용하도록 컴파일하여 ELF 이미지로 만든다.

- single step, breakpoint 를 이용한 디버깅 가능.
- 디버깅을 할 때 다른 기계가 필요하고, 설치가 어려우며, 모듈의 주소를 확인하지 못한다.

- Magic SysRq Key
  - 현재 상태를 보여주는 키 :
    Shift+Scroll : show memory
    Ctrl+Scroll : show state
    Alt + Scroll : show registers
  - 특수한 키 조합을 통해서 특수한 동작을 하도록 할 수 있다.
  - 커널 설정을 할 때 Magic SysRq 옵션을 켜서 컴파일해야 한다.
    사용법 : Alt+SysRq+<command>
      command : (Documentation/sysrq.txt)
      's' : sync all mounted filesystem
      'o' : shut off
      'm' : dump current memory

7. 시간과 타이머, 태스크 큐, 메모리

- 커널에서의 시간 :
  - RTC (Real Time Clock) : 시스템에서 현재 시간을 관리하는 장치
  - 커널은 timer interrupt 가 발생할 때마다 내부의 시간을 관리하는 데이터를 증가시켜 현재 시간을 관리한다.
  - `HZ` : 1 초에 timer interrupt 가 발생하는 횟수. (include/linux/param.h)
  - `jiffies` : 운영체제가 부팅한 후 발생한 클럭 tick 의 횟수. 1 jiffies = 1/HZ 초
  - timer interrupt 의 발생 간격 조정 :
    `HZ` 값 증가 : 속도는 느려지지만 반응 속도는 빨라진다.
    `HZ` 값 감소 : 속도는 빨라지지만 반응 속도는 느려진다.

- 현재 시간 :
  - `do_gettimeofday()` 함수를 통해서 second, microsecond 단위의 시간을 얻을 수 있다.
  - 내부적으로 현재 시간을 가지고 있는 `struct timeval xtime` 이란 변수가 있지만, 커널의 내부 자료구조를 직접 접근하는 것은 바람직하지 않다.
  - 현재 시간 얻기 :

```
struct timeval {
    time_t tv_sec;
    suseconds_t tv_usec;
}

#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
```

- Long Delay
  - 방법 1 : busy waiting

    기다리는 동안에 커널 모드에 계속 있어야 하며, 다른 작업을 실행할 수 없음.

```
unsigned long j = jiffies + delay * HZ;

while (jiffies < j)
    ;
```

  - 방법 2 :

    다른 실행할 프로세스가 없으면 계속 자신이 선택되어 실행됨.

```
while (jiffies < j)
    schedule();
```

  - 방법 3 :

    timeout 을 이용하여 원하는 시간동안 blocking 이 되므로 가장 효율적이다.
    여기서 timeout 은 지연할 시간 (jiffies + α 가 아니라)

```
struct wait_queue *wq = NULL;

interruptible_sleep_on_timeout(&wait, timeout);
```

- Short Delay : udelay() (include/asm/delay.h)
  - prototype : `void udelay(unsigned long usecs);`
  - jiffies 보다 더 짧은 시간의 delay 를 위해서 사용한다.
  - 부팅할 때 계산한 BogoMips 를 바탕으로 `loops_per_seconds` 를 계산하고, 이를 바탕으

로 원하는 시간동안의 delay 를 위해 소프트웨어적으로 loop 를 돈다.

- busy waiting
- 1 second = 1000 milisecond, 1 milisecond = 1000 microsecond

● 커널 타이머

- 어떤 특정 작업을 실행할 시간을 지정하여 일정 시간이 경과한 후에 실행하게 한다.
- 타이머 큐에 들어간 작업은 한번만 실행된다.
- 옛날 방식의 타이머 : 고정된 갯수의 타이머를 제공한다.

```
struct timer_struct {
    unsigned long expires;
    void (*fn)(void);
};

unsigned long timer_active;
struct timer_struct timer_table[32];
```

- 새로운 방식의 타이머 : linked list 방식으로 갯수에 제한이 없다.

```
struct timer_list {
    struct timer_list *next; /* MUST be first element */
    struct timer_list *prev;
    unsigned long expires;
    unsigned long data;
    void (*function)(unsigned long);
};

void init_timer(struct timer_list * timer)
void add_timer(struct timer_list * timer);
int  del_timer(struct timer_list * timer);
```

- **Filename : mytimer.c**

```
/*
    Timer Handling
*/

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

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/sched.h>
#include <linux/timer.h>

/*
 * type definitions
 */

struct mytimer_data {
    int m_times;
    int m_endtimes;
};

/*
 * global variables
 */
```

```
static struct timer_list s_mytimer;
static struct mytimer_data s_mytimer_data;

/*
 * function prototypes
 */

static void mytimer_proc(unsigned long ptr);
static void add_mytimer(void);
static void del_mytimer(void);

/*
 * Module startup/cleanup
 */

int init_module(void)
{
   printk("Loading Timer\n");

   s_mytimer_data.m_times = 0;
   s_mytimer_data.m_endtimes = 5;

   add_mytimer();

   return 0;
}

void cleanup_module(void)
{
   del_mytimer();
```

```
    printk("Unloading Timer\n");
}

void mytimer_proc(unsigned long ptr)
{
    struct mytimer_data *pdata = (struct mytimer_data *) ptr;

    ++pdata->m_times;

    printk("Timer called %d/%d\n", pdata->m_times, pdata->m_endtimes);

    if (pdata->m_times < pdata->m_endtimes)
        add_mytimer();
}

void add_mytimer(void)
{
    init_timer(&s_mytimer);

    s_mytimer.function = mytimer_proc;
    s_mytimer.data = (unsigned long) &s_mytimer_data;
    s_mytimer.expires = jiffies + HZ * 5;   /* 5 seconds */

    add_timer(&s_mytimer);
}

void del_mytimer(void)
{
    del_timer(&s_mytimer);
}
```

- Context
  - user context : system call, trap 처럼 현재 커널의 동작이 현재 프로세스하고 관련이 있을 때.
    blocking 이 가능하다.
    `schedule()` 함수를 불러서 다른 프로세스가 실행되게 할 수 있다.
  - interrupt context : 인터럽트 핸들러에서 인터럽트를 처리하고 있는 중, 또는 현재 프로세스와 무관하게 임의의 시점에 실행되는 인터럽트와 유사한 상황.
    - hardware interrupt context : 인터럽트 핸들러가 처리되고 있을 때로 해당 인터럽트가 금지된 상태에서 실행되며, 경우에 따라 다른 인터럽트가 금지되기도 한다.
    - software interrupt context : 인터럽트가 허용된 상태에서 실행되며, bottom half, timer, 특정 task queue 처럼 임의의 시점에 실행된다.
    - interrupt context 에서 할 수 없는 일 :
      current process 하고 관련되는 일 : current process 가 의미가 없다.
      current process 가 없으므로 user space 와 데이터 교환을 할 수 없다.
      scheduling : schedule() 함수를 부르거나, 간접적으로 부를 수 있는 함수들.
      swap 이 발생할 수 있는 일 : 메모리 할당같이 atomic operation 이 아닌 것들.
      user memory 에서 page fault 가 발생할 수 있는 일 : copy_to_user()같이 사용자 메모리를 접근하는 함수들.

- Task Queue
  - 특정 작업을 어느정도 시간이 지난후에 실행하는 방법 중의 하나이다.
  - polling 을 처리하거나, 하드웨어를 정밀하게 제어할 때 사용한다.
  - bottom half 처럼 interrupt 를 처리할 때 interrupt handler 를 처리한 후에 남은 작업을 처리할 때도 사용한다.

- task queue 자료구조 : linked list 형태

```
struct tq_struct {
   struct tq_struct *next;    /* linked list of active bh's */
   unsigned long sync;        /* must be initialized to zero */
   void (*routine)(void *);   /* function to call */
   void *data;                /* argument to function */
};
```

- task queue 함수

```
#define DECLARE_TASK_QUEUE(q)
```

새로운 태스크 큐를 정의한다.

```
void queue_task(struct tq_struct *bh_pointer, task_queue *bh_list);
```

태스크 큐에 태스크를 추가한다.

```
void run_task_queue(task_queue *list);
```

큐에 있는 모든 작업을 실행한다.

- predefined task queues : 시스템 동작을 위해 미리 정의해 놓은 task queue 들.

`tq_timer` : 타이머 인터럽트가 발생하였을 때 처리되는 큐 (interrupt time)

`tq_immediate` : 가능한한 빠른 시간내에 처리되는 큐 (interrupt time)

`tq_scheduler` : 스케쥴러가 실행될 때 처리되는 큐 (not interrupt time)

● 메모리 할당 1 : kmalloc, kfree (include/linux/slab.h)

```
void *kmalloc(size_t size, int priority);
```

`malloc()`처럼 지정한 크기만큼의 메모리를 할당한다.

```
void kfree(const void *ptr);
```

`free()`처럼 `malloc()`으로 할당받은 메모리를 해제한다.

- priority : 메모리 할당 방식을 지정한다.

`GFP_KERNEL : (__GFP_MED | __GFP_WAIT | __GFP_IO)`

일반적으로 사용하는 priority 로 blocking 이 일어날 수 있다.

메모리가 부족하면 swap 등을 통해서 메모리 공간을 만든다.

`GFP_USER : (__GFP_LOW | __GFP_WAIT | __GFP_IO)`

`GFP_NFS : (__GFP_HIGH | __GFP_WAIT | __GFP_IO)`

NFS 파일시스템에서 되도록 빨리 메모리를 할당받을 때 사용한다.

`GFP_ATOMIC : (__GFP_HIGH)`

interrupt time 에서 사용 (scheduling 이 일어나서는 안되는 경우)

`__min_free_pages` 와 무관하게 가능한 모든 메모리 사용한다.

`GFP_DMA : (__GFP_DMA)`

DMA 메모리 할당한다.

물리적으로 16MB 이하의 영역에 연속된 메모리를 할당한다.

- size :

커널에서는 page pool 을 이용하여 페이지를 관리한다.

page oriented allocation : RAM 을 가장 유연하게 사용할 수 있는 방법

linear allocation : 리눅스 커널같이 page oriented 방식에서는 관리하기 어렵다.

`PAGE_SIZE` * 2^n 보다 조금 작은 크기의 메모리 할당이 가장 효율적이다.

크기제한이 있다 : 128KB

● 메모리 할당 : page (include/linux/mm.h)

```
unsigned long get_free_page(int gfp_mask);
```

한 페이지를 할당한 후 page 를 지운다.

```
unsigned long __get_free_page(int gfp_mask);
```

한 페이지를 할당한다.

```
unsigned long __get_dma_pages(int gfp_mask, order)
```
DMA 용 메모리 페이지를 할당한다.
```
unsigned long __get_free_pages(int gfp_mask, unsigned long gfp_order);
```
지정한 크기만큼의 페이지를 할당한다.
```
void free_page(unsigned long addr);
```
한 페이지를 해제한다.
```
void free_pages(unsigned long addr, unsigned long order);
```
여러 페이지를 해제한다.

- 메모리 할당 : vmalloc, vfree (include/linux/vmalloc.h)

```
void * vmalloc(unsigned long size);
```
가상 주소 공간에서 메모리를 할당한다.
```
void vfree(void * addr);
```
vmalloc()으로 할당받은 메모리를 해제한다.

- 가상 주소 공간(virtual address space)에서 연속된 메모리 영역을 할당한다.
  kmalloc()은 물리적 주소 공간에서 메모리를 할당한다.
  물리적으로 연속되지 않은 메모리도 가상 주소 공간에서는 연속적으로 볼 수 있다.
  page table 을 이용하여 물리적 주소로 변환된다.
- 커널 메모리 공간에서만 보이는 영역이다.
- 물리적인 공간에서만 연속되지 않을 뿐, 할당받은 메모리는 똑같이 사용할 수 있으며, 대부분의 경우 할당받은 메모리가 물리적으로 연속되어야 할 필요는 없다.
- 소프트웨어적으로만 사용하는 큰 버퍼를 할당받을 때 사용한다 (크기 제한이 kmalloc() 보다 크다)

9. 여섯번째 프로그램 : 시스템 콜 가로채기

- 목적 : 어떤 특정한 시스템 콜이 불렸을 때 이를 가로채서 다른 일을 하도록 한다. 여기서는 sys_open()과 sys_close() 시스템 콜을 가로채며, 시스템의 안정성을 위해서 시스템 콜이 불렸는지만 확인하고, 원래의 시스템 콜 처리 함수를 그대로 불러준다.

- 구현방법 : 전체 시스템 콜 핸들러의 목록을 가지고 있는 sys_call_table 벡터의 내용을 직접 수정하여, open, close 시스템 콜의 핸들러를 저장하고, 프로그램 내에 새로 제작한 핸들러로 대치한다. 대치된 핸들러에서는 넘어온 인자의 값을 출력하고, 원래의 시스템 콜 핸들러를 불러서 작업을 처리한 후 결과값을 보여준다.

- 테스트 방법 : 모듈을 로드한 후 open, close 시스템 콜이 불릴 때마다 메시지가 출력되는지 확인한다.

- 과정 :
  - open, close 시스템 콜을 가로채기
  - 다른 시스템 콜을 가로채기

- 테스트 과정 :

```
insmod syscall.o
ls
rmmod syscall
```

- Filename : syscall.c

```
/*
   System Call Spy
*/

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

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <sys/syscall.h>
#include <asm/uaccess.h>

/*
 * Global Variable
 */

extern void *sys_call_table[];

static asmlinkage int (*orig_sys_open)(const char *filename, int flags, int mode);
static asmlinkage int (*orig_sys_close)(unsigned int fd);

/*
 * Function Prototypes
 */
```

```
static asmlinkage int my_sys_open(const char *filename, int flags, int mode);
static asmlinkage int my_sys_close(unsigned int fd);

/*
 * Module startup/cleanup
 */

int init_module(void)
{
   printk("Loading System Call Spy Module\n");

   orig_sys_open = sys_call_table[__NR_open];
   sys_call_table[__NR_open] = my_sys_open;

   orig_sys_close = sys_call_table[__NR_close];
   sys_call_table[__NR_close] = my_sys_close;

   return 0;
}

void cleanup_module(void)
{
   printk("Unloading System Call Spy Module\n");

   if (sys_call_table[__NR_open] != my_sys_open ||
       sys_call_table[__NR_close] != my_sys_close) {
       printk("Error : other program changed the system call handler\n");
   }

   sys_call_table[__NR_open] = orig_sys_open;
```

```
    sys_call_table[__NR_close] = orig_sys_close;
}

/*
 * System Call handler
 */

int my_sys_open(const char *filename, int flags, int mode)
{
    int retcode;
    char fname[1024], *cpsrc = (char *) filename, *cpdes = fname;

    for (;;) {
        get_user(*cpdes, cpsrc);
        if (*cpdes == 0)
            break;
        ++cpdes;
        ++cpsrc;
    }

    printk("sys_open : pid = %d, filename = %s, flags = %x, mode = %x\n",
        current->pid, fname, flags, mode);
    retcode = orig_sys_open(filename, flags, mode);
    printk("sys_open : result = %d\n", retcode);

    return retcode;
}

int my_sys_close(unsigned int fd)
{
```

```
    int retcode;

    printk("sys_close : pid = %d, fd = %d\n", current->pid, fd);
    retcode = orig_sys_close(fd);
    printk("sys_close : result = %d\n", retcode);

    return retcode;
}
```

- 시스템 콜 :
  - 사용자 모드에 있는 응용프로그램이 커널의 기능을 사용할 수 있도록 하는 것
  - 시스템 콜을 부르면 사용자 모드에서 커널 모드로 바뀐다.
  - 커널에서 시스템 콜을 처리하면 커널 모드에서 사용자 모드로 바뀌어 작업을 계속한다.
  - 시스템 콜의 종류 :
    `include/asm/unistd.h` 에 정의되어 있다.

- 시스템 콜의 처리과정 :
  - IA-32 에서 system call 은 0x80 인터럽트를 통하여 처리된다.
  - 사용자 프로그램에서 system call 을 부르면 인터럽트가 발생한다.
  - 0x80 인터럽트 핸들러는 `system_call` (arch/i386/kernel/entry.S)이다.
  - `system_call` 함수는 넘겨준 system call 번호(%eax)에 따라 `sys_call_table[]`에 있는 함수를 호출한다.
  - 각 시스템 콜 처리 함수는 해당 명령을 실행하고 결과를 돌려준다.
  - `ret_from_sys_call` 함수를 통해 system call 을 마치고 사용자 모드로 복귀한다.

10. 인터럽트 처리

- 인터럽트
  - 장치가 CPU 에게 event 가 발생했음을 알려주는 방법
  - asynchronous event
  - polling/interrupt
    polling : 장치의 변화를 주기적으로 감시해서 상태의 변화를 알아챈다.
    interrupt : 장치에 신경쓰지 않고 작업을 하다가 상태 변화가 있을 때 장치가 알려준다.
  - DMA 는 인터럽트와 연계해서 사용한다.

- 인터럽트 핸들러의 등록과 해지 : (include/linux/sched.h)

```
int request_irq(unsigned int irq,
     void (*handler)(int, void *, struct pt_regs *),
     unsigned long flags,
     const char *device,
     void *dev_id);
```

`irq` : 인터럽트 번호.
`handler` : 인터럽트를 처리할 핸들러 함수
`flags` : 인터럽트 관리와 관련한 flag
  `SA_INTERRUPT` : fast interrupt/slow interrupt 를 구별하기 위해 사용.
  `SA_SHIRQ` : 인터럽트를 공유할 수 있다는 것을 나타낸다.
  `SA_SAMPLE_RANDOM` : random 값을 만드는데 기여한다. 장치의 인터럽트가 비정기적으로 발생하는 경우 random 값을 만드는데 도움이 된다.
`device` : 인터럽트 소유자를 표시할 때 나타낼 문자열

dev_id : 인터럽트를 공유할 때 자신을 구별하기 위해서 사용하는 유일한 값.
핸들러에게 데이터를 전달하는 목적으로도 사용할 수 있다.

```
void free_irq(unsigned int irq, void *dev_id);
```

인터럽트를 반납한다. 등록할 때 지정한 irq 와 dev_id 를 그대로 지정한다.

- 인터럽트를 등록/해지할 시점 :

init_module : 모듈을 사용하지 않더라도 인터럽트를 점유하는 문제가 발생
open : 인터럽트를 다른 프로그램이 사용하고 있으면 점유하지 못할 수도 있다.
등록 : 처음으로 장치를 open 할 때에 인터럽트를 등록
해지 : 마지막으로 close 를 할 때 인터럽트를 해지

- /proc/interrupts

현재 점유하고 있는 인터럽트의 목록을 보여준다.

● IRQ 번호 알아내기

- 설정가능한 IRQ : PCI 에서는 IRQ 가 시스템에 의해서 할당된다.
장치 정보에 저장된 인터럽트 설정을 읽어와서 사용한다.
- 고정된 IRQ : ISA 에서는 IRQ 가 하드웨어적으로 고정되어 있다.
표준적인 방법으로 IRQ 를 알아낼 수가 없다.
1) 모듈을 로드할 때 IRQ, I/O 주소를 인자로 전달한다.
2) 장치의 I/O 영역의 값 중에 IRQ 정보가 있는 경우 이를 읽어온다.
3) 장치 특성을 이용하여 IRQ 번호를 probing 한다.

- IRQ probing : (include/linux/interrupt.h)

```
unsigned long probe_irq_on(void);
```

현재 할당되지 않은 인터럽트의 bitmask 를 돌려준다.
실패하면 0 을 돌려준다.

```
int probe_irq_off(unsigned long);
```
probe_on 이후에 발생한 인터럽트 번호를 돌려준다.
인터럽트가 발생하지 않으면 0 을 돌려준다.
여러개의 인터럽트가 발생하면 (ambiguous detection) 음수를 돌려준다.

probe 하기전 : `probe_irq_on()`을 부른다.
장치의 특성을 이용하여 하드웨어적으로 인터럽트를 발생시킨다.
probe 를 마치고 : `probe_irq_off()`를 부른다.

- 인터럽트 핸들러 구현 :

```
void handler(int irq, void *dev_id, struct pt_regs *regs);
```

- 인터럽트 핸들러는 hardware interrupt context 에서 실행된다.
- 인터럽트 처리 시간을 최소화해야 한다.

인터럽트 핸들러에서는 해당 인터럽트가 금지된 상태에서 실행된다.
fast interrupt 의 경우 모든 인터럽트가 금지된 상태에서 실행된다.
시스템의 원활한 작업을 위해서는 인터럽트 처리 시간이 짧을 수록 좋다.
인터럽트 처리에 시간이 많이 필요한 경우 인터럽트 처리를 인터럽트 핸들러에서 처리해야 하는 부분과 그렇지 않아도 되는 부분으로 나누며, 뒷 부분은 bottom half, task queue 를 이용하여 처리한다.

- 인터럽트 허용과 금지 : (include/asm/irq.h)

```
void disable_irq(unsigned int);
```
해당 인터럽트를 금지한다.
```
void enable_irq(unsigned int);
```
인터럽트를 허용한다.

- 모든 인터럽트 핸들러는 현재 처리중인 인터럽트가 금지된 상태에서 처리된다.
- 인터럽트 허용/금지는 인터럽트 핸들러에서 사용할 수 없다.
- fast interrupt/slow interrupt

  fast interrupt : 모든 인터럽트가 금지된 상태에서 인터럽트를 처리한다. 인터럽트 처리 도중에 다른 인터럽트는 발생하지 않는다. 빨리 처리해야하는 인터럽트 처리시에 사용한다.

  slow interrupt : 다른 인터럽트가 허용된 상태에서 인터럽트를 처리한다.

- Bottom Half
  - interrupt handler 에서 처리해야 하는 작업이 많은 경우, 반드시 interrupt handler 에서 처리하지 않아도 되는 일을 interrupt handler 를 빠져나온 후 실행할 수 있도록 한다.
  - top half / bottom half

    top half 에서는 장치의 데이터를 별도의 버퍼에 저장하고 bottom half 를 표시한다.

    bottom half 는 도착한 데이터를 프로세스에게 전달한다.
  - bottom half 처리 도중에는 모든 인터럽트가 허용된다.
  - bottom half 구현 : (include/linux/interrupt.h)

    function pointer 의 배열과 사용여부와 처리여부를 나타내는 bitmask 로 만들어진다.

```
volatile unsigned char bh_running;
atomic_t bh_mask_count[32];
unsigned long bh_active;
unsigned long bh_mask;
void (*bh_base[32])(void);
```

  - bottom half 관련 함수 : (include/asm/softirq.h)

```
void init_bh(int nr, void (*routine)(void))
```

    bottom half 를 등록한다.

`void remove_bh(int nr)`
bottom half 를 해지한다.
`void mark_bh(int nr)`
bottom half 를 처리해야 하는 경우 불러준다.
`void disable_bh(int nr)`
특정 bottom half 처리를 금지한다.
`void enable_bh(int nr)`
특정 bottom half 처리를 허용한다.
`void do_bottom_half(void)`
bottom half 처리 함수. return_from_sys_call()에서 불린다.

- bottom half 의 특징 :

bottom half 의 갯수는 한정되어 있다.
대부분 특정 디바이스 드라이버에게 할당되어 있다.

- 커널에서 사용하는 bottom half :

`IMMEDIATE_BH` : tq_immediate 를 처리한다.
`TQUEUE_BH` : tq_timer 를 처리한다.
`NET_BH` : network driver 에서 사용한다.
`CONSOLE_BH` : tty switching
`TIMER_BH` : timer interrupt 처리 후 호출되며, kernel timer 를 처리한다.

● 인터럽트 공유 :

- 인터럽트를 점유할 때 인터럽트 공유 여부를 설정한다.
- ISA : 인터럽트 공유에 대해서 규정된 것은 없지만, 하드웨어적으로 인터럽트 공유가 불가능하지는 않다. 하드웨어적으로 장치에 인터럽트가 발생했는지 여부를 확인할 수 있는

방법이 있다면 소프트웨어적인 방법으로 해결이 가능하다.

- PCI : 규약에서 기본적으로 인터럽트 공유가 가능하도록 설계되어 있다.
- interrupt handler

  인터럽트가 발생하면 인터럽트를 공유하는 모든 핸들러가 불린다.

  인터럽트 핸들러가 불렸을 때 반드시 자신의 인터럽트가 발생한 것은 아니다.

  하드웨어를 검사하여 인터럽트가 자신의 장치에서 발생한 경우에만 인터럽트를 처리

- race condition :
  - 코드가 인식하지 못한 사이에 자신이 다루는 데이터가 interrupt time 에 바뀌고, 해당 코드는 데이터를 변화를 인식하지 못하고 처리를 계속하는 경우.
  - race : nonatomic operation 과 다른 코드 사이에서 누가 먼저 처리하나 경주.
  - race condition 이 발생하는 경우 :

    함수 내에서 scheduling 이 발생 : 직/간접적으로 schedule() 함수가 불린 경우.

    blocking operation

    인터럽트 핸들러와 시스템 콜이 다른 코드와 데이터를 공유하는 경우
  - race condition 해결 :

    공유 데이터를 사용하지 않고 원형 버퍼를 사용한다.

    공유 데이터를 수정할 때 잠깐동안 인터럽트를 금지시킨다.

    데이터를 접근할 때 lock 을 사용해서 atomic operation 으로 만든다.

11. 하드웨어 제어

- 하드웨어 제어에 사용하는 자원 :
  - I/O Port :
  - I/O Memory :
  - interrupt
  - DMA

- I/O Port 제어 : (include/asm/io.h, include/linux/ioport.h)
  - I/O port 영역 할당 :

    I/O 영역을 사용하기 전에 해당 영역을 다른 드라이버가 사용하는지 검사한다.
    실제로 할당을 받지 않았다고 해서 해당 영역을 접근할 수 없는 것은 아니지만, 그래도 장치 사이의 충돌을 막기 위해서 영역을 검사하고, 사용중인 경우 에러를 발생하는 것이 좋다.

    ```
    int check_region(unsigned long from, unsigned long extent);
    ```
    해당 I/O port 영역이 비어있는지 검사한다.
    ```
    void request_region(unsigned long from, unsigned long extent,
        const char *name);
    ```
    I/O port 영역을 할당받는다.
    ```
    void release_region(unsigned long from, unsigned long extent);
    ```
    I/O port 영역의 사용을 마치고 자원을 반납한다.
  - 하나의 데이터 I/O
    ```
    unsigned int inb(unsigned int port);
    void outb(unsigned char data, unsigned int port);
    ```

8 bit 데이터 I/O

```
unsigned int inw(unsigned int port);
void outw(unsigned short data, unsigned int port);
```

16 bit 데이터 I/O

```
unsigned int inl(unsigned int port);
void outl(unsigned int data, unsigned int port);
```

32 bit 데이터 I/O

- 하나의 port 로 연속된 데이터 I/O

```
unsigned int insb(unsigned int port, void *addr, unsigned long count);
void outsb(unsigned int port, void *addr, unsigned long count);
```

8 bit 데이터 I/O

```
unsigned int insw(unsigned int port, void *addr, unsigned long count);
void outsw(unsigned int port, void *addr, unsigned long count);
```

16 bit 데이터 I/O

```
unsigned int insl(unsigned int port, void *addr, unsigned long count);
void outsl(unsigned int port, void *addr, unsigned long count);
```

32 bit 데이터 I/O

- pausing I/O

CPU 의 속도는 I/O bus 의 속도보다 빠르다.

옛날 하드웨어 중에는 속도차이 때문에 port I/O 에서 데이터를 놓치는 경우가 있다.

`inb_p()`, `outb_p()` 같은 함수를 이용하여 I/O 사이에 delay 를 줄수있다.

● I/O Memory 제어 : (include/asm/io.h)

- I/O memory 영역 할당 :

interrupt 나 I/O port 에서와 같은 영역 할당 메커니즘은 없다.

- I/O memory I/O

```
unsigned int readb(void *addr);
void writeb(unsigned int data, void *addr);
```
8 bit 데이터 I/O
```
unsigned int readw(void *addr);
void writew(unsigned int data, void *addr);
```
16 bit 데이터 I/O
```
unsigned int readl(void *addr);
void writel(unsigned int data, void *addr);
```
32 bit 데이터 I/O
```
void memset_io(void *addr, unsigned int data, int count);
```
지정한 크기만큼 지정한 데이터로 설정
```
void memcpy_fromio(void *dest, void *src, int count);
```
I/O 메모리에서 RAM 으로 데이터 전송
```
void memcpy_toio(void *dst, void *src, int count);
```
RAM 에서 I/O 메모리로 데이터 전송

- I/O memory mapping

PCI 장치같은 곳에서는 memory map 을 통해서 I/O 메모리를 접근한다.
```
void * ioremap (unsigned long offset, unsigned long size)
```
I/O 메모리를 가상 메모리 주소로 mapping 을 한다.
```
void iounmap(void *addr);
```
mapping 된 메모리를 해제한다.

- DMA 제어 : (include/asm/dma.h)
  - DMA 영역 할당 :

```
int request_dma(unsigned int dmanr, const char * device_id);
```
DMA channel 을 요구한다. channel 은 0-7 까지 가능하다.

```
    void free_dma(unsigned int dmanr);
```
DMA channel 을 반납한다.

- DMA 버퍼 할당 :

```
    kmalloc(size, GFP_DMA);
    get_dma_pages(mask, pages);
```

- DMA 사용하기 :

```
    void enable_dma(unsigned int dmanr);
```
DMA channel 에 유효한 데이터가 있다고 표시한다.
```
    void disable_dma(unsigned int dmanr);
```
DMA channel 을 disable 시킨다.
```
    void set_dma_mode(unsigned int dmanr, char mode);
```
DMA 를 통하여 데이터를 읽을지, 쓸지를 지정한다.
```
    void set_dma_addr(unsigned int dmanr, unsigned int a);
```
DMA 버퍼의 주소를 지정한다.
```
    void set_dma_page(unsigned int dmanr, char pagenr);
```
전송할 주소의 page register 만을 설정한다.
```
    void set_dma_count(unsigned int dmanr, unsigned int count);
```
전송할 데이터의 크기를 지정한다.
```
    int get_dma_residue(unsigned int dmanr);
```
DMA 를 통해서 전송할 데이터 중에서 남은 숫자를 알려준다.

DMA 명령의 성공 여부를 알려준다.
```
    void clear_dma_ff(unsigned int dmanr);
```

12. 일곱번째 프로그램 : MiniRD - 간단한 블럭 디바이스 드라이버

- 목적 : 간단한 블럭 디바이스를 만든다. 여기서는 실제 블럭 장치를 사용하지 않고, RAM 상에 일부 메모리를 디스크처럼 활용하는 간단한 RAM 디스크를 만든다.

- 구현방법 : 문자 디바이스 드라이버와 마찬가지로 파일 연산을 정의하고 이를 등록한다. 이때 파일 연산 중에서 read/write 는 일반적인 블럭 read/write 함수를 이용한다. 그리고 블럭 디바이스 드라이버 구현에 필요한 request 함수와 블럭 크기, 섹터 크기, 장치 크기 등을 지정한다. 모듈을 로드할 때 RAM 디스크를 위해 메모리를 할당한다. RAM 디스크의 크기는 모듈에 고정되어 있으며, 모듈을 unload 할 때 할당받은 메모리를 해제한다. request 함수에서는 버퍼캐시의 버퍼와 할당받은 메모리 사이에 read/write 를 수행한다.

- 테스트 방법 : 모듈을 로드한 후 블럭 디바이스 드라이버를 가리키는 장치 파일을 만든다. 여기서는 DEVICE_NUM 갯수만큼의 minor number 를 허용하기 때문에 여러개의 장치 파일을 만들 수 있다. mke2fs 명령으로 장치 파일상에 파일시스템을 만든 후 mount 를 해서 파일시스템처럼 사용을 한다.

- 과정 :
  - 간단한 RAM 디스크 블럭 디바이스 드라이버 제작
  - RAM 디스크 상에 파일시스템 구축 및 테스트

- 테스트 과정 :

```
insmod minird.o minird_size=1024
mknod /dev/minird0 b [major] 0
dd if=/dev/zero of=/dev/minird0 bs=1024 count=1024
mke2fs /dev/minird0 1024
mkdir /mnt/other
mount /dev/minird0 /mnt/other
cp /bin/ls /mnt/other
ls /mnt/other
umount /mnt/other
mount /dev/minird0 /mnt/other
ls /mnt/other
umount /mnt/other
rmmod minird
```

- **Filename : minird.c**

```
/*
   Mini RAM Disk Driver
*/

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

#include <linux/vmalloc.h>
#include <asm/uaccess.h>

#define MINIRD_MAJOR          0             /* major number : dynamic allocation */
#define DEVICE_NAME           "MiniRD"       /* device name */
#define DEVICE_NUM            2             /* minor device number */

#define MINIRD_DEF_SIZE       1024          /* default device size in KB */
#define MINIRD_BLOCKSIZE      1024          /* sizeof block */
#define MINIRD_SECTSIZE       512           /* sizeof sector */

extern int s_nMajor;

#define MAJOR_NR              (s_nMajor)
#define DEVICE_REQUEST        minird_request
#define DEVICE_NR(device)     (MINOR(device))
#define DEVICE_ON(device)                   /* do nothing */
#define DEVICE_OFF(device)                  /* do nothing */
#define DEVICE_NO_RANDOM

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

```
/*
 * Globak Variables
 */

static int s_nMajor = 0;
static int minird_size = MINIRD_DEF_SIZE;

static int s_kbsize[DEVICE_NUM];             /* size in blocks of 1024 bytes */
static int s_blocksize[DEVICE_NUM];           /* size of 1024 byte block */
static int s_hardsect[DEVICE_NUM];            /* sizeof real block in bytes */
static int s_length[DEVICE_NUM];             /* size of disks in bytes */
static char *s_data[DEVICE_NUM];

MODULE_PARM(minird_size, "i");
MODULE_PARM_DESC(minird_size, "RAM Disk size in KB");

/*
 * Function Prototypes
 */

static int minird_open(struct inode *inodep, struct file *filp);
static int minird_release(struct inode *inodep, struct file *filp);
static int minird_ioctl(struct inode *inodep, struct file *filp, unsigned int cmd,
     unsigned long arg);

static void minird_request(void);

/*
 * Device Operations
```

```
 */

static struct file_operations minird_fops = {
    open : minird_open,
    release : minird_release,
    read : block_read,
    write : block_write,
    ioctl : minird_ioctl,
    fsync : block_fsync,
};

/*
 * Module startup/cleanup
 */

int init_module(void)
{
    int i, j;

    printk("Loading Mini RAM Disk Module\n");

    for (i = 0; i < DEVICE_NUM; ++i) {
        s_kbsize[i] = minird_size;
        s_blocksize[i] = MINIRD_BLOCKSIZE;
        s_hardsect[i] = MINIRD_SECTSIZE;
        s_length[i] = (minird_size << BLOCK_SIZE_BITS);

        if ((s_data[i] = vmalloc(s_length[i])) == NULL) {
            for (j = 0; j < i; ++j)
                vfree(s_data[j]);
```

```
            return -ENOMEM;
        }
    }

    if ((s_nMajor = register_blkdev(MINIRD_MAJOR, DEVICE_NAME, &minird_fops)) < 0) {
        printk(DEVICE_NAME " : Device registration failed (%d)\n", s_nMajor);
        return s_nMajor;
    }

    printk(DEVICE_NAME " : Device registered with Major Number = %d\n", MAJOR_NR);

    blk_dev[MAJOR_NR].request_fn = &minird_request;

    blk_size[MAJOR_NR] = s_kbsize;
    blksize_size[MAJOR_NR] = s_blocksize;
    hardsect_size[MAJOR_NR] = s_hardsect;

    return 0;
}

void cleanup_module(void)
{
    int i;

    printk("Unloading Mini RAM Disk Module\n");

    for (i = 0; i < DEVICE_NUM; ++i)
        destroy_buffers(MKDEV(MAJOR_NR, i));       /* flush the devices */

    for (i = 0; i < DEVICE_NUM; ++i)
```

```
        vfree(s_data[i]);

    unregister_blkdev(MAJOR_NR, DEVICE_NAME);

    blk_dev[MAJOR_NR].request_fn = NULL;

    blk_size[MAJOR_NR] = NULL;
    blksize_size[MAJOR_NR] = NULL;
    hardsect_size[MAJOR_NR] = NULL;
}

/*
 * Device Operations
 */

int minird_open(struct inode *inodep, struct file *filp)
{
    if (DEVICE_NR(inodep->i_rdev) >= DEVICE_NUM)
        return -ENXIO;

    MOD_INC_USE_COUNT;

    return 0;
}

int minird_release(struct inode *inodep, struct file *filp)
{
    MOD_DEC_USE_COUNT;

    return 0;
```

```
}

int minird_ioctl(struct inode *inodep, struct file *filp, unsigned int cmd,
     unsigned long arg)
{
    int minor = DEVICE_NR(inodep->i_rdev);
    long devsize;

    switch (cmd) {
    case BLKGETSIZE :    /* return device size */
       devsize = s_length[minor] / s_hardsect[minor];
       return put_user(devsize, (long *) arg);

    case BLKSSZGET :    /* block size of media */
       return put_user(s_blocksize[minor], (int *) arg);

    case BLKFLSBUF :    /* flush */
       if (!capable(CAP_SYS_ADMIN))
           return -EACCES;
       destroy_buffers(inodep->i_rdev);
       break;

    RO_IOCTLS(inodep->i_rdev, arg);   /* default ioctl for BLKROSET, BLKROGET */

    default :
       return -EINVAL;
    }

    return 0;
}
```

```
/*
 * Request Processing
 */

void minird_request(void)
{
    char *ptr;
    int size, minor;

    while (1) {
        INIT_REQUEST;

        if ((minor = DEVICE_NR(CURRENT_DEV)) > DEVICE_NUM) {
            printk(DEVICE_NAME " : Unknown Minor Device\n");
            end_request(0);
            continue;
        }

        ptr = s_data[minor] + CURRENT->sector * s_hardsect[minor];
        size = CURRENT->current_nr_sectors * s_hardsect[minor];

        if (ptr + size > s_data[minor] + s_length[minor]) {
            printk(DEVICE_NAME " : Request past end of device\n");
            end_request(0);
            continue;
        }

        switch (CURRENT->cmd) {
        case READ :
```

```
            memcpy(CURRENT->buffer, ptr, size);
            break;
        case WRITE :
            memcpy(ptr, CURRENT->buffer, size);
            break;
        default :
            end_request(0);
            continue;
        }

        end_request(1);
    }
}
```

- 블럭 디바이스 드라이버의 제작법
  - 문자 디바이스 드라이버와 마찬가지로 외부와는 파일 인터페이스로 연결되며 파일 연산들(file_operations)을 제공한다.
  - 디바이스 드라이버는 major number 로 구별되며 문자 디바이스 드라이버와는 별개로 관리되므로 문자 디바이스 드라이버와 major number 가 겹쳐도 상관없다.
  - 파일 시스템은 buffer cache 를 통해서 블럭 디바이스를 접근한다. 버퍼 캐시를 통한 작업은 일반 read/write 와는 틀리기 때문에 파일 연산의 read/write 와 별도로 버퍼 캐시를 통한 read/write 를 하기 위한 별도의 인터페이스가 존재한다 : request
  - 따로 request 함수를 구현해야 하고, 이 함수에서 실질적인 read/write 작업을 한다. 또한 블럭의 크기, 섹터의 크기, 전체 크기 등을 따로 블럭 장치를 관리하는 전역 변수에 설정해야 한다.

- 장치의 등록과 해제
  - 등록 :
    ```
    int register_blkdev(unsigned int major, const char *name,
       struct file_operations *fops);
    ```
    문자장치에서와 마찬가지로 major 번호에 0 을 지정하여 동적으로 할당받을 수 있다.
  - 해제 :
    ```
    int unregister_blkdev(unsigned int major, const char * name);
    ```

- 파일 연산(file_operations)
  - 블럭 디바이스 드라이버의 파일 연산은 문자 디바이스 드라이버와 조금 차이가 있다.
  - read/write/fsync 함수는 따로 구현할 필요가 없으며, 일반적으로 `block_read()`, `block_write()`, `block_fsync()` 함수를 그대로 사용한다.

- open, release : 문자 디바이스 드라이버에서와 마찬가지로 구현한다.
- ioctl : 블럭 디바이스 드라이버에는 공통적으로 사용하는 여러가지 ioctl 명령이 있으며, 이들의 대부분은 반드시 지원을 해주어야 한다.
  BLKGETSIZE : 장치의 크기를 섹터의 갯수로 돌려준다.
  BLKFLSBUF : 버퍼를 flush 한다.
  BLKRAGET, BLKRASET : 얼마나 미리읽기(read-ahead)를 할 것인지 얻어오고 설정한다.
  BLKROSET, BLKROGET : 장치의 read only flag 를 얻어오고 변경한다. 따로 구현하지 않고 일반적으로 RO_IOCTLS(dev, where) 매크로를 통해 구현한다.
  BLKRRPART : 파티션 테이블을 다시 읽어온다. (파티션이 있는 장치에서)
  HDIO_GETGEO : 하드디스크의 구조를 읽어온다. (하드 디스크)
- check_media_change : media 를 바꿀 수 있는 장치의 경우 media 가 바뀌었는지 알아내기 위해서 사용한다. media 가 바뀐 경우 1 을, 바뀌지 않은 경우 0 을 돌려주면 된다.
- revalidate : media 가 바뀐 경우에 불리는 함수이다. 새로운 media 를 처리하기 위해 내부 정보를 수정하는 작업을 하면 된다.

● #include <linux/blk.h> 이전에 정의할 수 있는 것/정의해야 할 것
- **MAJOR_NR** : 장치의 major 번호
- DEVICE_NAME : 장치의 이름
- DEVICE_NR(kdev_t device) : 실제 장치 번호를 얻기 위한 매크로. 보통은 MINOR(device)
- DEVICE_ON(kdev_t device) :
- **DEVICE_OFF**(kdev_t device) : end_request()가 작업을 마치고 부르는 함수
- DEVICE_NO_RANDOM : random number 를 만드는데 기여하지 않음

● 블럭 디바이스 드라이버 관련 자료구조

```
struct blk_dev_struct {
    request_fn_proc        *request_fn;         => request 처리 함수
    queue_proc             *queue;
    void                   *data;
    struct request         *current_request;    => 현재 처리할 request
    struct request   plug;
    struct tq_struct       plug_tq;
};

extern struct blk_dev_struct blk_dev[MAX_BLKDEV];
```

모든 블럭 디바이스의 request 에 관련된 정보를 가지고 있는 배열

```
extern int * blk_size[MAX_BLKDEV];
```

major number, minor number 에 해당하는 장치의 크기를 KB 단위로 나타낸다.

```
extern int * blksize_size[MAX_BLKDEV];
```

major number, minor number 에 해당하는 장치의 블럭의 크기를 byte 단위로 나타낸다.

```
extern int * hardsect_size[MAX_BLKDEV];
```

major number, minor number 에 해당하는 장치의 섹터의 크기를 byte 단위로 나타낸다. (default=512)

```
extern struct sec_size * blk_sec[MAX_BLKDEV];
extern int * max_readahead[MAX_BLKDEV];
extern int * max_sectors[MAX_BLKDEV];
extern int * max_segments[MAX_BLKDEV];
```

- 블럭 디바이스 드라이버는 블럭 디바이스를 등록한 후 blk_dev[major].request_fn 을 비롯하여 장치에 대한 여러가지 정보를 설정해야 한다.
- 실질적인 데이터 전송은 blk_dev[major].request_fn 함수를 통해서 이루어진다.

- 현재 받은 요청은 blk_dev[major].current_request, 간단히 CURRENT

```
struct request {
    volatile int rq_status;
    kdev_t rq_dev;                  => 처리할 장치 (major/minor)
    int cmd;    /* READ or WRITE */ => 처리할 명령 : READ/WRITE
    int errors;
    unsigned long sector;           => 요청한 I/O 의 첫번째 섹터
    unsigned long nr_sectors;  => clustered request 를 사용할 때 인접한 섹터의 갯수
    unsigned long nr_segments;
    unsigned long current_nr_sectors;    => 요청한 I/O 의 섹터의 갯수
    char * buffer;                  => 데이터를 읽거나 쓸 버퍼 (버퍼 캐시에 있는 버퍼)
    struct semaphore * sem;
    struct buffer_head * bh;        => 첫번째 버퍼에 대한 정보
    struct buffer_head * bhtail;    => 마지막 버퍼에 대한 정보
    struct request * next;          => 다음 request 에 대한 연결 리스트
};
```

- request_fn()의 구현 : CURRENT 에 따라 I/O 를 수행한다. cmd 가 READ 이면 버퍼로 저장하고, WRITE 이면 장치로 저장한다. I/O 를 시작할 위치는 sector, 크기는 current_nr_sectors 에 들어 있으며 request 를 처리할 때마다 end_request() 함수를 부른다.

부록 A. 종합예제 : PC 스피커 디바이스 드라이버

- 목적 : 문자 디바이스 드라이버를 실제로 작성한다. 여기서는 PC 스피커 디바이스 드라이버를 작성하여 이를 이용하여 사운드를 출력한다.

- 구현방법 : PC 스피커로 사운드를 출력하는 것은 세개의 I/O 포트를 제어함으로써 가능하다. 0x42, 0x43, 0x61 포트에 적절한 값을 넣어주면 지정한 주파수의 소리가 발생하게 할 수 있으며, 소리가 나지 않게도 할 수 있다. 여기에 덧붙여 커널 타이머를 이용하여 지정한 시간동안 소리가 나도록 한다. 여러개의 음을 입력받은 경우 이를 내부의 큐에 쌓아두고 타이머를 반복적으로 이용하여 소리가 나도록 한다.

- 스펙 :
  - 지정한 음을 지정한 시간동안 PC 스피커로 출력할 수 있다.
  - 장치에 일반적인 `write()` 명령을 내려서 출력할 수 있다. 이 때 전달되는 데이터의 형식은 소리의 높이와 길이를 같이 가지고 있는 `struct playnote_t` 의 배열이다.
  - `ioctl()` 명령을 통해서 사운드를 출력할 수 있다. 이 때 전달되는 데이터의 형식은 전체 음의 갯수와 템포와 `struct playnote_t` 의 배열을 가리키는 포인터를 가지고 있는 `struct playdata_t` 이다.
  - 다양한 `ioctl()` 명령을 통해서 모듈을 제어할 수 있다.
    - SOUND : 지정한 주파수의 음을 출력한다.
    - SOUNDNOTE : 지정한 음을 출력한다.
    - NOSOUND : 소리가 나지 않게 한다.
    - PLAY : 음악을 연주한다. 현재 출력되고 있는 데이터가 있으면 큐의 끝에 추가한다.

STOP : 모든 연주를 중단한다. 큐에 있는 모든 내용은 삭제된다. 여기서 권한이 있는 프로세스만이 삭제가 가능하다.

SETTEMPO : `write()`로 출력하는 데이터의 기본 템포를 지정한다.

CHANGETEMPO : 현재 연주되는 음악의 템포를 바꾼다.

GETCOUNT : 현재 남은 음의 갯수를 알려준다.

- 장치에 대한 출력은 비동기적으로 할 수 있다. 즉 일반적인 경우에는 출력 요청한 데이터를 모두 출력한 후에 원래 프로세스로 돌아가지만, 비동기적인 경우에는 큐에만 넣은 후 바로 돌아간다.
- 장치는 동시에 하나의 프로그램만이 쓸 수 있다.

● 테스트 방법 : 각각의 기능이 제대로 동작하는지 확인한다. ioctl 로 구현한 sound / nosound / play / stop 등의 동작의 작동을 확인하고 write()로 기록한 악보대로 연주되는지 확인한다. asynchronous write 를 설정한 후 제대로 동작하는지 확인한다.

● 과정 :

- 디바이스 드라이버 제작
- 테스트 프로그램 제작

● 테스트 과정 :

```
insmod pcplay.o
mknod /dev/pcplay c [major] 0
./pcplay_test
rmmod pcplay
```

- **Filename : pcplay.h**

```
/*
 * PC Speaker Play
 */

#ifndef __PCPLAY_H
#define __PCPLAY_H

#include <linux/ioctl.h>

#define PCPLAY_MAGICNUM          0x37

#define PCPLAY_DEFAULTTEMPO       120

enum
    OCTAVE_NONE,
    OCTAVE_1,
    OCTAVE_2,
    OCTAVE_3,
    OCTAVE_4,
    OCTAVE_5,
    OCTAVE_6,
    OCTAVE_7,
    OCTAVE_8,
    OCTAVE_NUM
};

enum
    NOTE_C,
```

```
    NOTE_CSHARP, NOTE_DFLAT = NOTE_CSHARP,
    NOTE_D,
    NOTE_DSHARP, NOTE_EFLAT = NOTE_DSHARP,
    NOTE_E,
    NOTE_F,
    NOTE_FSHARP, NOTE_GFLAT = NOTE_FSHARP,
    NOTE_G,
    NOTE_GSHARP, NOTE_AFLAT = NOTE_GSHARP,
    NOTE_A,
    NOTE_ASHARP, NOTE_BFLAT = NOTE_ASHARP,
    NOTE_B,
    NOTE_NUM
};

#define NOTE_NOSOUND                0
#define NOTE_REST                   0

#define PCPLAY_MAKEINOTE(octave, note)    ((octave) * NOTE_NUM + note)
#define PCPLAY_GETOCTAVE(inote)          ((inote) / NOTE_NUM)
#define PCPLAY_GETNOTE(inote)          ((inote) % NOTE_NUM)

enum
    DURATION_64,
    DURATION_32,
    DURATION_16,
    DURATION_8,
    DURATION_4,
    DURATION_2,
    DURATION_1,
    DURATION_2X,
```

```
    DURATION_4X,
    DURATION_8X,
    DURATION_16X,
    DURATION_32X,
    DURATION_64X,
    DURATION_START = DURATION_64,
    DURATION_END = DURATION_64X,
    DURATION_0 = 0xff
};

enum
    DURATION_POINT = 0x10,
    DURATION_POINT2X = 0x20,
    DURATION_POINT3X = 0x30,
};

#define PCPLAY_GETDURATIONLENGTH(duration)      ((duration) & 0x0f)
#define PCPLAY_GETDURATIONPOINT(duration)      ((duration) & 0xf0)
#define PCPLAY_MAKEDURATION(duration, point)   ((duration) | (point))

struct playnote_t {
    unsigned char m_note;
    unsigned char m_duration;
};

struct playdata_t {
    int m_length;
    int m_tempo;
    struct playnote_t *m_pdata;
};
```

```
#define IOCTL_PCPLAY_SOUND           _IOW(PCPLAY_MAGICNUM, 0, int)
#define IOCTL_PCPLAY_SOUNDNOTE       _IOW(PCPLAY_MAGICNUM, 1, int)
#define IOCTL_PCPLAY_NOSOUND         _IO(PCPLAY_MAGICNUM, 2)
#define IOCTL_PCPLAY_PLAY            _IOW(PCPLAY_MAGICNUM, 10, struct playdata_t *)
#define IOCTL_PCPLAY_STOP            _IO(PCPLAY_MAGICNUM, 11)
#define IOCTL_PCPLAY_SETTEMPO        _IOW(PCPLAY_MAGICNUM, 20, int)
#define IOCTL_PCPLAY_CHANGETEMPO     _IOW(PCPLAY_MAGICNUM, 21, int)
#define IOCTL_PCPLAY_GETCOUNT        _IOR(PCPLAY_MAGICNUM, 22, int *)

#endif
```

- **Filename : pcplay.c**

```
/*
   PC Speaker Play Module
*/

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

#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/sched.h>
#include <linux/slab.h>
#include <linux/timer.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#include "pcplay.h"

/*
 * Device Definitions
 */

#define DEVICE_NAME           "PCPlay"

/*
 * type definitions
```

```
 */

struct playqueue_t {
   struct playqueue_t *m_next;
   pid_t m_pid;
   uid_t m_uid, m_euid;
   gid_t m_gid, m_egid;
   int m_curpos;
   struct playdata_t *m_pdata;
   struct wait_queue *m_wq;
};

/*
 * sound data definitions
 */

static int s_notefreq[OCTAVE_NUM][NOTE_NUM] = {
   { 16, 17, 18, 19, 21, 22, 23, 24, 26, 27, 29, 31 },
   { 33, 35, 37, 39, 41, 44, 46, 49, 52, 55, 58, 62 },
   { 65, 69, 73, 78, 82, 87, 92, 98, 104, 110, 116, 123 },
   { 131, 139, 147, 155, 165, 175, 185, 196, 208, 220, 233, 245 },
   { 262, 277, 294, 311, 330, 349, 370, 392, 415, 440, 466, 494 },
   { 523, 554, 587, 622, 659, 698, 740, 784, 831, 880, 932, 988 },
   { 1046, 1109, 1175, 1244, 1328, 1397, 1480, 1568, 1661, 1760, 1865, 1975 },
   { 2093, 2217, 2349, 2489, 2637, 2794, 2960, 3136, 3322, 3520, 3729, 3951 }
};

/*
 * function prototypes
 */
```

```
static int device_open(struct inode *inode, struct file *filp);
static int device_release(struct inode *inode, struct file *filp);
static ssize_t device_read(struct file *filp, char *buffer, size_t length,
     loff_t *offset);
static ssize_t device_write(struct file *filp, const char *buffer, size_t length,
     loff_t *offset);
static int device_ioctl(struct inode *inodep, struct file *filp, unsigned int cmd,
     unsigned long arg);

static void mytimer_proc(unsigned long ptr);
static void add_mytimer(int clocks);
static void del_mytimer(void);

static int play_curnote(void);

static int add_playdata_notes(char *pdata, int length);
static int add_playdata(struct playdata_t *pdata);
static void process_wait(struct file *filp);

static int enqueue_playdata(struct playdata_t *pdata);
static int dequeue_playdata(void);
static void dequeueall_playdata(void);
static int check_delperm(void);
static int settempo(int tempo);
static int changetempo_playdata(int tempo);
static int get_playqueue_count(void);

static void soundnote(int inote);
static void sound(int freq);
```

```
static void nosound(void);

/*
 * Device Operations
 */

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

/*
 * global variables
 */

static int s_nMajor = 0;
static int s_bDeviceReadOpen = 0, s_bDeviceWriteOpen = 0;

static struct playqueue_t *s_queue = NULL;
static int s_bInPlay = 0;
static int s_nTempo = 120;

static struct timer_list s_mytimer;

/*
 * Module startup/cleanup
 */
```

```
int init_module(void)
{
   printk("Loading PC Speaker Play Module\n");

   if ((s_nMajor = register_chrdev(0, DEVICE_NAME, &device_fops)) < 0) {
      printk(DEVICE_NAME " : Device registration failed (%d)\n", s_nMajor);
      return s_nMajor;
   }

   printk(DEVICE_NAME " : Device registered with Major Number = %d\n", s_nMajor);

   return 0;
}

void cleanup_module(void)
{
   int nRetCode;

   printk("Unloading PC Speaker Play Module\n");

   nosound();
   dequeueall_playdata();

   if ((nRetCode = unregister_chrdev(s_nMajor, DEVICE_NAME)) < 0)
      printk(DEVICE_NAME " : Device unregistration failed (%d)\n", nRetCode);
}

/*
 * Device Operations
```

```
 */

int device_open(struct inode *inode, struct file *filp)
{
   printk(DEVICE_NAME " : Device open (%d, %d)\n", MAJOR(inode->i_rdev),
      MINOR(inode->i_rdev));

   if ((filp->f_flags & O_ACCMODE) & (O_WRONLY | O_RDWR)) {
      if (s_bDeviceWriteOpen) {
         printk(DEVICE_NAME " : Device already open for writing\n");
         return -EBUSY;
      } else
         ++s_bDeviceWriteOpen;
   } else
      ++s_bDeviceReadOpen;

   MOD_INC_USE_COUNT;

   return 0;
}

int device_release(struct inode *inode, struct file *filp)
{
   printk(DEVICE_NAME " : Device release (%d, %d)\n", MAJOR(inode->i_rdev),
      MINOR(inode->i_rdev));

   if ((filp->f_flags & O_ACCMODE) & (O_WRONLY | O_RDWR))
      --s_bDeviceWriteOpen;
   else
      --s_bDeviceReadOpen;
```

```
    MOD_DEC_USE_COUNT;

    return 0;
}

ssize_t device_read(struct file *filp, char *buffer, size_t length, loff_t *offset)
{
    return 0;
}

ssize_t device_write(struct file *filp, const char *buffer, size_t length,
      loff_t *offset)
{
    int count = add_playdata_notes((char *) buffer, length);

    if (count > 0)
       filp->f_pos += count;

    process_wait(filp);

    return count;
}

int device_ioctl(struct inode *inodep, struct file *filp, unsigned int cmd,
      unsigned long arg)
{
    int length, retcode = 0;

    printk(DEVICE_NAME " : ioctl, cmd = %x, arg = %x\n", cmd, (int) arg);
```

```
switch (cmd) {
case IOCTL_PCPLAY_SOUND :
    sound((int) arg);
    break;

case IOCTL_PCPLAY_SOUNDNOTE :
    soundnote((int) arg);
    break;

case IOCTL_PCPLAY_NOSOUND :
    nosound();
    break;

case IOCTL_PCPLAY_PLAY :
    retcode = add_playdata((struct playdata_t *) arg);
    process_wait(filp);
    break;

case IOCTL_PCPLAY_STOP :
    nosound();
    dequeueall_playdata();
    break;

case IOCTL_PCPLAY_SETTEMPO :
    settempo((int) arg);
    break;

case IOCTL_PCPLAY_CHANGETEMPO :
    changetempo_playdata((int) arg);
```

```
        break;

    case IOCTL_PCPLAY_GETCOUNT :
        length = get_playqueue_count();
        put_user(length, (int *) arg);
        break;

    default :
        printk(DEVICE_NAME " : ioctl : unknown\n");
        retcode = -EINVAL;
    }

    return retcode;
}

/*
    Timer Management
*/

void mytimer_proc(unsigned long ptr)
{
    play_curnote();
}

void add_mytimer(int clock)
{
    init_timer(&s_mytimer);

    s_mytimer.function = mytimer_proc;
    s_mytimer.expires = jiffies + clock;
```

```
    add_timer(&s_mytimer);
}

void del_mytimer(void)
{
    del_timer(&s_mytimer);
}

#define DURATION_FACTOR          1000

int play_curnote(void)
{
    int duration, point, clocks;
    struct playnote_t *pnote;

    if (s_queue) {
        if (s_queue->m_curpos == s_queue->m_pdata->m_length) {
            nosound();
            dequeue_playdata();
            s_bInPlay = 0;
            play_curnote();
        } else {
            pnote = s_queue->m_pdata->m_pdata + s_queue->m_curpos;
            duration = PCPLAY_GETDURATIONLENGTH(pnote->m_duration);
            point = PCPLAY_GETDURATIONPOINT(pnote->m_duration);

            clocks = HZ * DURATION_FACTOR;

            if (duration >= DURATION_4 && duration <= DURATION_END)
```

```
            clocks <<= (duration - DURATION_4);
         else if (duration >= DURATION_START && duration < DURATION_4)
            clocks >>= (DURATION_4 - duration);
         else
            clocks = 0;

         while (point-- > 0)
            clocks = clocks * 3 / 2;

         clocks = clocks * 60 / s_queue->m_pdata->m_tempo;
         clocks /= DURATION_FACTOR;

         soundnote(pnote->m_note);
         add_mytimer(clocks);
         ++s_queue->m_curpos;

         s_bInPlay = 1;
      }

      return 0;
   } else
      return -ENODATA;
}

/*
   User space / Queue
*/

int add_playdata_notes(char *pdata, int length)
{
```

```
    struct playdata_t *newdata;
    int itemnum, buflen, retcode;

    itemnum = length / sizeof(struct playnote_t);
    buflen = sizeof(struct playdata_t) + (itemnum - 1) * sizeof(struct playnote_t);

    if ((newdata = kmalloc(buflen, GFP_KERNEL)) == NULL)
        return -ENOMEM;

    newdata->m_length = itemnum;
    newdata->m_tempo = s_nTempo;
    copy_from_user(newdata->m_pdata, pdata, itemnum * sizeof(struct playnote_t));

    if ((retcode = enqueue_playdata(newdata)) < 0) {
        kfree(newdata);
        return retcode;
    }

    if (!s_bInPlay)
        play_curnote();

    return length;
}

int add_playdata(struct playdata_t *pdata)
{
    int length, buflen, retcode;
    struct playdata_t *newdata;
    struct playnote_t *pnotes;
```

```
    get_user(length, &pdata->m_length);
    get_user(pnotes, &pdata->m_pdata);

    buflen = sizeof(struct playdata_t) + sizeof(struct playnote_t) * length -
        sizeof pdata->m_pdata;

    if ((newdata = kmalloc(buflen, GFP_KERNEL)) == NULL)
        return -ENOMEM;

    copy_from_user(newdata, pdata, sizeof(struct playdata_t));
    copy_from_user(newdata->m_pdata, pnotes, sizeof(struct playnote_t) * length);

    if ((retcode = enqueue_playdata(newdata)) < 0) {
        kfree(newdata);
        return retcode;
    }

    if (!s_bInPlay)
        play_curnote();

    return 0;
}

void process_wait(struct file *filp)
{
    if ((filp->f_flags & FASYNC) == 0)
        interruptible_sleep_on(&s_queue->m_wq);
}

/*
```

```
   Queue Management
*/

int enqueue_playdata(struct playdata_t *pdata)
{
    struct playqueue_t *endqueue, *newqueue;

    if ((newqueue = (struct playqueue_t *) kmalloc(sizeof(struct playqueue_t),
        GFP_KERNEL)) != NULL) {
        newqueue->m_next = NULL;
        newqueue->m_pid = current->pid;
        newqueue->m_uid = current->uid;
        newqueue->m_euid = current->euid;
        newqueue->m_gid = current->gid;
        newqueue->m_egid = current->egid;
        newqueue->m_curpos = 0;
        newqueue->m_pdata = pdata;
        newqueue->m_wq = NULL;

        if (s_queue == NULL)
            s_queue = newqueue;
        else {
           for (endqueue = s_queue; endqueue->m_next != NULL; endqueue = endqueue->m_next)
               ;
            endqueue->m_next = newqueue;
        }

        return 0;
    } else
        return -ENOMEM;
```

```
}

int dequeue_playdata(void)
{
    int retcode;
    struct playqueue_t *curqueue;

    if ((retcode = check_delperm()) < 0)
        return retcode;

    if (s_queue) {
        curqueue = s_queue;
        s_queue = s_queue->m_next;

        wake_up_interruptible(&curqueue->m_wq);
        kfree(curqueue->m_pdata);
        kfree(curqueue);

        return 0;
    } else
        return -ENODATA;
}

void dequeueall_playdata(void)
{
    while (dequeue_playdata() == 0)
        ;
}

int check_delperm(void)
```

```
{
    if (s_queue) {
        if (current->uid == s_queue->m_uid ||
            current->euid == s_queue->m_uid ||
            current->euid == s_queue->m_euid)
            return 0;
        else
            return -EPERM;
    } else
        return -ENODATA;
}

int settempo(int tempo)
{
    s_nTempo = tempo;

    return 0;
}

int changetempo_playdata(int tempo)
{
    if (s_queue) {
        s_queue->m_pdata->m_tempo = tempo;
        return 0;
    } else
        return -ENODATA;
}

int get_playqueue_count(void)
{
```

```
    int count;
    struct playqueue_t *queue;

    for (count = 0, queue = s_queue; queue != NULL; queue = queue->m_next)
        count += queue->m_pdata->m_length - queue->m_curpos;

    return count;
}

/*
   PC Speaker Sound Primitive
*/

void soundnote(int inote)
{
    if (inote == NOTE_REST)
        nosound();
    else
        sound(s_notefreq[PCPLAY_GETOCTAVE(inote) - 1][PCPLAY_GETNOTE(inote)]);
}

void sound(int freq)
{
    unsigned int value = inb(0x61);

    freq = 1193181 / freq;

    if ((value & 3) == 0) {
        outb(value | 3, 0x61);
        outb(0xb6, 0x43);
```

```
    }

    outb(freq & 0xff, 0x42);
    outb((freq >> 8) & 0xff, 0x42);

    printk("freq = %x\n", freq);
}

void nosound(void)
{
    unsigned int value = inb(0x61);

    value &= 0xfc;
    outb(value, 0x61);
}
```

- **Filename : pcplay_test.c**

```
/*
   PC Speaker Play Module Test
*/

#include <stdio.h>
#include <fcntl.h>

#include "pcplay.h"

/*
 * global data
 */

static struct playdata_t s_playdata;

static struct playnote_t s_notes[] = {
   { PCPLAY_MAKEINOTE(OCTAVE_4, NOTE_C), DURATION_4 },
   { PCPLAY_MAKEINOTE(OCTAVE_4, NOTE_D), DURATION_4 },
   { PCPLAY_MAKEINOTE(OCTAVE_4, NOTE_E), DURATION_4 },
   { PCPLAY_MAKEINOTE(OCTAVE_4, NOTE_F), DURATION_4 },
   { PCPLAY_MAKEINOTE(OCTAVE_4, NOTE_G), DURATION_4 },
   { PCPLAY_MAKEINOTE(OCTAVE_4, NOTE_A), DURATION_4 },
   { PCPLAY_MAKEINOTE(OCTAVE_4, NOTE_B), DURATION_4 },
   { PCPLAY_MAKEINOTE(OCTAVE_5, NOTE_C), DURATION_4 },
};

#define s_notesnum (sizeof s_notes / sizeof s_notes[0])
```

```
/*
 * function prototypes
 */

static int pcplay_ioctl_sound(int fd, int freq);
static int pcplay_ioctl_soundnote(int fd, int octave, int note);
static int pcplay_ioctl_nosound(int fd);
static int pcplay_ioctl_play(int fd, struct playdata_t *pdata);
static int pcplay_ioctl_stop(int fd);
static int pcplay_ioctl_settempo(int fd, int tempo);
static int pcplay_ioctl_changetempo(int fd, int tempo);
static int pcplay_ioctl_getcount(int fd);

/*
 * main of the program
 */

int main(void)
{
   int i, fd;

   if ((fd = open("/dev/pcplay", O_RDWR)) < 0) {
      printf("Error opening device file\n");
      return 1;
   }

   printf("SOUND : octave 4, C for 1 second\n");
   pcplay_ioctl_soundnote(fd, OCTAVE_4, NOTE_C);
   sleep(1);
   printf("NOSOUND : for 1 second\n");
```

```
    pcplay_ioctl_nosound(fd);
    sleep(1);

    s_playdata.m_length = s_notesnum;
    s_playdata.m_tempo = PCPLAY_DEFAULTTEMPO;

    printf("PLAY 1 : using IOCTL\n");
    pcplay_ioctl_play(fd, &s_playdata);
    printf("PLAY 2 : using WRITE\n");
    write(fd, s_notes, sizeof s_notes);
    printf("PLAY 3 : asynchronous play & stop\n");
    fcntl(fd, F_SETFL, fcntl(0, F_GETFL) | FASYNC);
    write(fd, s_notes, sizeof s_notes);
    sleep(3);
    pcplay_ioctl_stop(fd);
    printf("PLAY end\n");

    close(fd);

    return 0;
}

/*
 * ioctls
 */

int pcplay_ioctl_sound(int fd, int freq)
{
    return ioctl(fd, IOCTL_PCPLAY_SOUND, freq);
}
```

```
int pcplay_ioctl_soundnote(int fd, int octave, int note)
{
   return ioctl(fd, IOCTL_PCPLAY_SOUNDNOTE, PCPLAY_MAKEINOTE(octave, note));
}

int pcplay_ioctl_nosound(int fd)
{
   return ioctl(fd, IOCTL_PCPLAY_NOSOUND, 0);
}

int pcplay_ioctl_play(int fd, struct playdata_t *pdata)
{
   return ioctl(fd, IOCTL_PCPLAY_PLAY, pdata);
}

int pcplay_ioctl_stop(int fd)
{
   return ioctl(fd, IOCTL_PCPLAY_STOP, 0);
}

int pcplay_ioctl_settempo(int fd, int tempo)
{
   return ioctl(fd, IOCTL_PCPLAY_SETTEMPO, tempo);
}

int pcplay_ioctl_changetempo(int fd, int tempo)
{
   return ioctl(fd, IOCTL_PCPLAY_CHANGETEMPO, tempo);
}
```

```
int pcplay_ioctl_getcount(int fd)
{
    int count, retcode;

    return ((retcode = ioctl(fd, IOCTL_PCPLAY_GETCOUNT, &count)) < 0) ? retcode : count;
}
```

부록 B. 짧은 FAQ

Q : 커널 메시지를 보려면?
A : 현재 보고 있는 화면이 console 이라면 여기에 바로 커널 메시지를 출력한다. X Window 상이라면 터미널 창을 하나 더 띄우고 `tail -f /var/log/messages` 라는 명령을 실행해서 출력되는 메시지를 계속 확인하면서 프로그래밍 하는 것이 좋다.

Q : insmod, rmmod 등의 프로그램이 실행이 되지 않는데?
A : 모듈을 로드, 언로드를 하는 프로그램은 superuser 만이 할 수 있다. superuser 로서 명령을 실행을 하고, path 가 잡혀있지 않다면 /sbin/insmod 식으로 디렉토리를 지정한다.

Q : 모듈을 로드하려는데 버전이 맞지 않는다는 에러가 발생한다.
A : 현재 사용하는 버전과 모듈을 컴파일하는데 사용한 커널 버전을 일치시킨다. 자세한 방법은 2 장의 <모듈 버전 문제>를 참고한다.

Q : 모듈을 로드하려는데 참조하는 심볼이 없다는 에러가 발생한다.
A : 2 장의 <unresolved symbol 에러 문제>를 참고하여 원인을 찾아내서 해결한다.

Q : 모듈과 사용자 모드의 프로세스와 데이터를 교환할 때 왜 복사를 해야하나?
A : 커널모드에서 사용하는 데이터 세그먼트 공간과 사용자모드에서 사용하는 데이터 세그먼트 공간이 틀리기 때문에 커널에서는 사용자 메모리를 직접 접근할 수 없다. 따라서 데이터를 주고받을 때 복사를 해야 한다.

Q : 모듈에서 sys_로 시작하는 시스템 콜을 처리하는 함수를 부를 수 있는가?
A : 시스템 콜을 처리하는 함수는 인자로 넘어온 포인터는 사용자모드의 주소공간에 있는 데이터라고 가정을 한다. 따라서 인자로 커널모드의 주소공간에 있는 데이터의 포인터를 넘겨주어야 하는 경우 시스템 콜을 처리하는 함수를 직접 부를 수 없다.

Q : 다른 버전의 커널에서 컴파일을 하는데 에러가 발생하면?
A : 모듈 프로그래밍하는 방법은 대체로 비슷하지만, 세세한 코드에서는 버전마다 차이가 있다. 버전에 따라 함수나 자료구조가 바뀔 수도 있고, 몇가지 인터페이스의 변화가 있을 수 있다. 각 버전의 특징을 살펴서 소스를 고쳐야 한다.