ChatGPT

SCULL 커널 모듈 소스코드 (scull-master)

1.1 scull.h

```
/*
 * scull.h -- char 장치 모듈을 위한 정의
 *
 * Copyright (C) 2001 Alessandro Rubini and Jonathan Corbet
 * Copyright (C) 2001 O'Reilly & Associates
 *
 * 이 파일의 소스 코드는 "Linux Device Drivers" (Alessandro Rubini and Jonathan Corbet, O'Reilly &
Associates)에서 가져온 코드로 자유롭게 사용, 수정, 재배포할 수 있습니다. 파생 소스 파일에는 그 출처에 대한
명시가 필요합니다. 이 코드는 어떠한 보증도 제공하지 않으며, 사용상의 오류나 적합성에 대한 책임을 지지 않습
니다.
 *
 * $Id: scull.h,v 1.15 2004/11/04 17:51:18 rubini Exp $
 */

#ifndef _SCULL_H_
#define _SCULL_H_

#include <linux/ioctl.h> /* 이후 사용되는 _IOW 등의 매크로 정의 위해 필요 */

/*
 * 디버깅을 돕는 매크로들
 */

#undef PDEBUG          /* 필요하면 재정의 */
#ifdef SCULL_DEBUG
#  ifdef __KERNEL__
    /* 디버깅이 활성화되고 커널 공간에서 실행될 때 */
#    define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull: " fmt, ## args)
#  else
    /* 사용자 공간에서의 디버깅 */
#    define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
#  endif
#else
#  define PDEBUG(fmt, args...) /* 디버깅하지 않을 때는 아무 것도 하지 않음 */
#endif

#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* placeholder */

#ifndef SCULL_MAJOR
#define SCULL_MAJOR 0   /* 기본 값: dynamic major 할당 */
#endif
```

```
#ifndef SCULL_NR_DEVS
#define SCULL_NR_DEVS 4    /* 디바이스: scull0 ~ scull3 */
#endif

#ifndef SCULL_P_NR_DEVS
#define SCULL_P_NR_DEVS 4  /* 파이프 디바이스: scullpipe0 ~ scullpipe3 */
#endif

/*
 * 기본 디바이스는 가변 길이 메모리 영역으로 표현.
 * 간접 블록들의 링크드 리스트로 관리.
 *
 * "scull_dev->data"는 포인터들의 배열을 가리키며, 각 포인터는 SCULL_QUANTUM 바이트 크기의 메모리 영역을 가리킨다.
 *
 * 이 배열(quantum-set)은 길이가 SCULL_QSET이다.
 */
#ifndef SCULL_QUANTUM
#define SCULL_QUANTUM 4000
#endif

#ifndef SCULL_QSET
#define SCULL_QSET    1000
#endif

/*
 * 파이프 디바이스는 단순한 순환 버퍼(circular buffer). 기본 크기 정의:
 */
#ifndef SCULL_P_BUFFER
#define SCULL_P_BUFFER 4000
#endif

/*
 * scull quantum set의 표현 (링크드 리스트 노드 구조)
 */
struct scull_qset {
    void **data;
    struct scull_qset *next;
};

struct scull_dev {
    struct scull_qset *data;  /* 첫 번째 quantum 세트에 대한 포인터 */
    int quantum;          /* 현재 quantum 크기 */
    int qset;             /* 현재 배열 크기 (quantum-set 크기) */
    unsigned long size;     /* 이 디바이스에 저장된 데이터 크기 (바이트) */
    unsigned int access_key;  /* sculluid와 scullpriv 장치에서 사용 */
    struct mutex mutex;     /* 상호 배제를 위한 mutex */
    struct cdev cdev;       /* 문자 디바이스 구조체 */
};

/*
```

```
 * minor 번호를 두 부분으로 분할하기 위한 매크로 (scullaccess 장치에서 사용)
 */
#define TYPE(minor)    (((minor) >> 4) & 0xf)  /* 상위 4비트 */
#define NUM(minor)     ((minor) & 0xf)       /* 하위 4비트 */

/*
 * 다양한 설정 가능한 매개변수들 (주요 장치, 디바이스 수, quantum, qset 등)
 */
extern int scull_major;    /* main.c 에서 정의 */
extern int scull_nr_devs;
extern int scull_quantum;
extern int scull_qset;

extern int scull_p_buffer;     /* pipe.c 에서 정의 */

/*
 * 공유 함수들의 프로토타입 선언
 */
int    scull_p_init(dev_t dev);
void   scull_p_cleanup(void);
int    scull_access_init(dev_t dev);
void   scull_access_cleanup(void);

int    scull_trim(struct scull_dev *dev);

ssize_t scull_read(struct file *filp, char __user *buf, size_t count,
            loff_t *f_pos);
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count,
             loff_t *f_pos);
loff_t  scull_llseek(struct file *filp, loff_t off, int whence);
long   scull_ioctl(struct file *filp, unsigned int cmd, unsigned long arg);

/*
 * ioctl 정의
 */

/* 'k'를 매직넘버로 사용 */
#define SCULL_IOC_MAGIC  'k'
/* ioctl 커맨드 번호 0번부터 사용 */
#define SCULL_IOCRESET    _IO(SCULL_IOC_MAGIC, 0)

/*
 * 코드 의미:
 * S: 포인터를 통해 Set (설정)
 * T: 인자 값을 직접 Tell (설정)
 * G: Get: 포인터를 통해 가져와 설정
 * Q: Query: 반환 값으로 응답
 * X: eXchange: G와 S를 atomic하게 교환
 * H: sHift: T와 Q를 atomic하게 교환
 */
#define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC,  1, int)
```

```
#define SCULL_IOCSQSET   _IOW(SCULL_IOC_MAGIC, 2, int)
#define SCULL_IOCTQUANTUM _IO(SCULL_IOC_MAGIC,  3)
#define SCULL_IOCTQSET   _IO(SCULL_IOC_MAGIC,  4)
#define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC, 5, int)
#define SCULL_IOCGQSET   _IOR(SCULL_IOC_MAGIC, 6, int)
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC,  7)
#define SCULL_IOCQQSET   _IO(SCULL_IOC_MAGIC,  8)
#define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, int)
#define SCULL_IOCXQSET   _IOWR(SCULL_IOC_MAGIC,10, int)
#define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC, 11)
#define SCULL_IOCHQSET   _IO(SCULL_IOC_MAGIC, 12)

/* 파이프 (scullpipe) 디바이스 관련 ioctl */
#define SCULL_P_IOCTSIZE  _IO(SCULL_IOC_MAGIC, 13)
#define SCULL_P_IOCQSIZE  _IO(SCULL_IOC_MAGIC, 14)

#define SCULL_IOC_MAXNR 14
#endif /* _SCULL_H_ */
```

1.2 main.c

```
/*
 * main.c -- 기본 scull 문자 디바이스 모듈
 *
 * Copyright (C) 2001 Alessandro Rubini and Jonathan Corbet
 * Copyright (C) 2001 O'Reilly & Associates
 *
 * (라이선스 및 책 출처에 대한 내용은 scull.h에 기술된 내용과 동일)
 */

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>

#include <linux/kernel.h>     /* printk() */
#include <linux/slab.h>       /* kmalloc() */
#include <linux/fs.h>         /* 대부분의 커널 파일시스템 관련 정의 */
#include <linux/errno.h>      /* 오류 코드 */
#include <linux/types.h>      /* size_t 등 */
#include <linux/proc_fs.h>
#include <linux/fcntl.h>      /* O_ACCMODE 등 상수 */
#include <linux/seq_file.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>      /* copy_*_user 함수 */

#include "scull.h"            /* 로컬 정의 포함 */

/*
 * 모듈 로드 시 설정할 수 있는 파라미터들
 */
```

```
int scull_major =   SCULL_MAJOR;
int scull_minor =   0;
int scull_nr_devs = SCULL_NR_DEVS;     /* scull 기본 디바이스 수 */
int scull_quantum = SCULL_QUANTUM;
int scull_qset =    SCULL_QSET;

module_param(scull_major, int, S_IRUGO);
module_param(scull_minor, int, S_IRUGO);
module_param(scull_nr_devs, int, S_IRUGO);
module_param(scull_quantum, int, S_IRUGO);
module_param(scull_qset, int, S_IRUGO);

MODULE_AUTHOR("Alessandro Rubini, Jonathan Corbet");
MODULE_LICENSE("Dual BSD/GPL");

struct scull_dev *scull_devices;      /* scull_init_module에서 할당됨 */

#ifdef SCULL_DEBUG /* 디버깅을 위해 /proc 사용 */

/*
 * /proc 설정
 *
 * 각 scull 디바이스에 대한 정보 출력을 위해 읽기 전용 /proc 엔트리를 설정하는 데 필요한 함수들.
 *
 * scullmem은 PAGE_SIZE보다 큰 /proc 엔트리를 생성하는 데 적합하지 않은 구 인터페이스 사용.
 * scullseq는 seq_file 인터페이스를 구현하여 더 큰 /proc 엔트리에 적합.
 */

/* scullmem의 /proc 구현 */
int scull_read_procmem(struct seq_file *s, void *v)
{
    int i, j;
    int limit = s->size - 80; /* 너무 많은 문자를 출력하지 않도록 제한 */

    for (i = 0; i < scull_nr_devs && s->count <= limit; i++) {
        struct scull_dev *d = &scull_devices[i];
        struct scull_qset *qs = d->data;
        if (mutex_lock_interruptible(&d->mutex))
            return -ERESTARTSYS;
        seq_printf(s, "\nDevice %i: qset %i, q %i, sz %li\n",
                   i, d->qset, d->quantum, d->size);
        for (; qs && s->count <= limit; qs = qs->next) { /* 리스트 순회 */
            seq_printf(s, "  item at %p, qset at %p\n",
                       qs, qs->data);
            if (qs->data && !qs->next) /* 마지막 아이템만 덤프 */
                for (j = 0; j < d->qset; j++) {
                    if (qs->data[j])
                        seq_printf(s, "    %4i: %8p\n",
                                   j, qs->data[j]);
                }
```

```
            }
            mutex_unlock(&d->mutex);
        }
        return 0;
}

static int scullmem_proc_open(struct inode *inode, struct file *filp)
{
        return single_open(filp, scull_read_procmem, NULL);
}

struct file_operations scullmem_proc_ops = {
        .owner   = THIS_MODULE,
        .open    = scullmem_proc_open,
        .llseek  = seq_lseek,
        .read    = seq_read,
        .release = single_release,
};

/* scullseq의 /proc 구현 */

static void *scull_seq_start(struct seq_file *s, loff_t *pos)
{
        if (*pos >= scull_nr_devs)
                return NULL;   /* 더 이상 출력할 내용 없음 */
        return scull_devices + *pos;
}

static void *scull_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
        (*pos)++;
        if (*pos >= scull_nr_devs)
                return NULL;
        return scull_devices + *pos;
}

static void scull_seq_stop(struct seq_file *s, void *v)
{
        /* 특별히 정리할 작업 없음 */
}

static int scull_seq_show(struct seq_file *s, void *v)
{
        struct scull_dev *dev = (struct scull_dev *) v;
        struct scull_qset *d;
        int i;

        if (mutex_lock_interruptible(&dev->mutex))
                return -ERESTARTSYS;
        seq_printf(s, "\nDevice %i: qset %i, q %i, sz %li\n",
                        (int) (dev - scull_devices), dev->qset,
```

```
                dev->quantum, dev->size);
        for (d = dev->data; d; d = d->next) { /* 리스트 순회 */
            seq_printf(s, "  item at %p, qset at %p\n", d, d->data);
            if (d->data && !d->next) /* 마지막 아이템만 덤프 */
                for (i = 0; i < dev->qset; i++) {
                    if (d->data[i])
                        seq_printf(s, "    %4i: %8p\n",
                                   i, d->data[i]);
                }
        }
        mutex_unlock(&dev->mutex);
        return 0;
}

static struct seq_operations scull_seq_ops = {
        .start = scull_seq_start,
        .next  = scull_seq_next,
        .stop  = scull_seq_stop,
        .show  = scull_seq_show
};

static int scullseq_proc_open(struct inode *inode, struct file *filp)
{
        return seq_open(filp, &scull_seq_ops);
}

static struct file_operations scullseq_proc_ops = {
        .owner   = THIS_MODULE,
        .open    = scullseq_proc_open,
        .llseek  = seq_lseek,
        .read    = seq_read,
        .release = seq_release,
};

static void scull_create_proc(void)
{
        proc_create_data("scullmem", 0 /* 기본 권한 */,
                NULL /* 상위 디렉터리 */, &scullmem_proc_ops,
                NULL /* 클라이언트 데이터 */);
        proc_create_data("scullseq", 0, NULL, &scullseq_proc_ops, NULL);
}

static void scull_remove_proc(void)
{
        /* 등록되지 않은 경우에도 문제 없음 */
        remove_proc_entry("scullmem", NULL);
        remove_proc_entry("scullseq", NULL);
}

#endif /* SCULL_DEBUG */
```

```
/* scull 디바이스 구현 시작 */

/*
 * scull 디바이스 비우기: 호출 시 dev의 mutex를 이미 획득한 상태여야 함
 */
int scull_trim(struct scull_dev *dev)
{
    struct scull_qset *next, *dptr;
    int qset = dev->qset;   /* dev는 NULL이 아님 */
    int i;

    for (dptr = dev->data; dptr; dptr = next) { /* 모든 리스트 아이템 순회 */
        if (dptr->data) {
            for (i = 0; i < qset; i++)
                kfree(dptr->data[i]);
            kfree(dptr->data);
            dptr->data = NULL;
        }
        next = dptr->next;
        kfree(dptr);
    }
    dev->size = 0;
    dev->quantum = scull_quantum;
    dev->qset = scull_qset;
    dev->data = NULL;
    return 0;
}

/*
 * Open 및 Release 함수
 */

int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* 디바이스 정보 포인터 */

    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = dev; /* 다른 메서드에서 사용하기 위해 디바이스 구조체 저장 */

    /* 디바이스를 write 전용으로 열었다면 내용 길이를 0으로 */
    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY) {
        if (mutex_lock_interruptible(&dev->mutex))
            return -ERESTARTSYS;
        scull_trim(dev); /* 오류는 무시 */
        mutex_unlock(&dev->mutex);
    }
    return 0;
}

int scull_release(struct inode *inode, struct file *filp)
{
```

```
    return 0;
}

/*
 * 링크드 리스트를 따라 n번째 qset을 얻음
 */
struct scull_qset *scull_follow(struct scull_dev *dev, int n)
{
    struct scull_qset *qs = dev->data;

    /* 필요한 경우 첫 번째 qset을 명시적으로 할당 */
    if (! qs) {
        qs = dev->data = kmalloc(sizeof(struct scull_qset), GFP_KERNEL);
        if (qs == NULL)
            return NULL;
        memset(qs, 0, sizeof(struct scull_qset));
    }

    /* 리스트를 따라가 n번째 qset에 도달 */
    while (n--) {
        if (!qs->next) {
            qs->next = kmalloc(sizeof(struct scull_qset), GFP_KERNEL);
            if (qs->next == NULL)
                return NULL;
            memset(qs->next, 0, sizeof(struct scull_qset));
        }
        qs = qs->next;
        continue;
    }
    return qs;
}

/*
 * 데이터 관리: read와 write
 */

ssize_t scull_read(struct file *filp, char __user *buf, size_t count,
        loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr; /* 첫 번째 리스트 아이템 */
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset; /* 한 리스트 아이템의 총 바이트 수 */
    int item, s_pos, q_pos, rest;
    ssize_t retval = 0;

    if (mutex_lock_interruptible(&dev->mutex))
        return -ERESTARTSYS;
    if (*f_pos >= dev->size)
        goto out;
    if (*f_pos + count > dev->size)
```

```
            count = dev->size - *f_pos;

    /* 리스트 아이템 번호, qset 인덱스, quantum 내부 위치 계산 */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum;
    q_pos = rest % quantum;

    /* 해당 위치까지 리스트 따라감 */
    dptr = scull_follow(dev, item);

    if (dptr == NULL || !dptr->data || ! dptr->data[s_pos])
        goto out; /* 해당 영역에 데이터 없음 (hole) */

    /* 이 quantum의 끝까지만 읽기 */
    if (count > quantum - q_pos)
        count = quantum - q_pos;

    if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;

 out:
    mutex_unlock(&dev->mutex);
    return retval;
}

ssize_t scull_write(struct file *filp, const char __user *buf, size_t count,
            loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset;
    int item, s_pos, q_pos, rest;
    ssize_t retval = -ENOMEM; /* "goto out"에서 사용할 값 */

    if (mutex_lock_interruptible(&dev->mutex))
        return -ERESTARTSYS;

    /* 리스트 아이템 번호, qset 인덱스, quantum 내부 오프셋 계산 */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum;
    q_pos = rest % quantum;

    /* 해당 위치까지 리스트 따라감 */
    dptr = scull_follow(dev, item);
```

```
    if (dptr == NULL)
        goto out;
    if (!dptr->data) {
        dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
        if (!dptr->data)
            goto out;
        memset(dptr->data, 0, qset * sizeof(char *));
    }
    if (!dptr->data[s_pos]) {
        dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
        if (!dptr->data[s_pos])
            goto out;
    }
    /* 이 quantum의 끝까지만 쓰기 */
    if (count > quantum - q_pos)
        count = quantum - q_pos;

    if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;

    /* 파일의 사이즈를 업데이트 */
    if (dev->size < *f_pos)
        dev->size = *f_pos;

 out:
    mutex_unlock(&dev->mutex);
    return retval;
}

/*
 * ioctl() 구현
 */

long scull_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{

    int err = 0, tmp;
    int retval = 0;

    /*
     * cmd에서 방향과 번호 추출하고, 부적절한 cmd라면 처리하지 않음.
     * access_ok 실행 전에 ENOTTY(Invalid ioctl) 에러 반환.
     */
    if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
    if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;

    /*
```

```
  * 방향 플래그는 비트마스크로, VERIFY_WRITE는 R/W 전송에서 사용됨.
 * 'Type'은 사용자 관점, access_ok는 커널 관점이므로 "read"와 "write" 개념이 반대임에 주의.
 */
if (_IOC_DIR(cmd) & _IOC_READ)
    err = !access_ok((void __user *)arg, _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)
    err = !access_ok((void __user *)arg, _IOC_SIZE(cmd));
if (err) return -EFAULT;

switch(cmd) {
 case SCULL_IOCRESET:
    scull_quantum = SCULL_QUANTUM;
    scull_qset = SCULL_QSET;
    break;

 case SCULL_IOCSQUANTUM: /* Set: arg가 값을 가리킴 */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    retval = __get_user(scull_quantum, (int __user *)arg);
    break;

 case SCULL_IOCTQUANTUM: /* Tell: arg 값 자체 사용 */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    scull_quantum = arg;
    break;

 case SCULL_IOCGQUANTUM: /* Get: arg가 결과 값을 가리킴 */
    retval = __put_user(scull_quantum, (int __user *)arg);
    break;

 case SCULL_IOCQQUANTUM: /* Query: 현재 값을 반환 */
    return scull_quantum;

 case SCULL_IOCXQUANTUM: /* eXchange: arg가 포인터로 가리키는 값과 교환 */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    tmp = scull_quantum;
    retval = __get_user(scull_quantum, (int __user *)arg);
    if (retval == 0)
        retval = __put_user(tmp, (int __user *)arg);
    break;

 case SCULL_IOCHQUANTUM: /* sHift: 이전 값 반환하면서 arg 값으로 설정 */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    tmp = scull_quantum;
    scull_quantum = arg;
    return tmp;

 case SCULL_IOCSQSET:
```

```
            if (! capable (CAP_SYS_ADMIN))
                return -EPERM;
            retval = __get_user(scull_qset, (int __user *)arg);
            break;

        case SCULL_IOCTQSET:
            if (! capable (CAP_SYS_ADMIN))
                return -EPERM;
            scull_qset = arg;
            break;

        case SCULL_IOCGQSET:
            retval = __put_user(scull_qset, (int __user *)arg);
            break;

        case SCULL_IOCQQSET:
            return scull_qset;

        case SCULL_IOCXQSET:
            if (! capable (CAP_SYS_ADMIN))
                return -EPERM;
            tmp = scull_qset;
            retval = __get_user(scull_qset, (int __user *)arg);
            if (retval == 0)
                retval = put_user(tmp, (int __user *)arg);
            break;

        case SCULL_IOCHQSET:
            if (! capable (CAP_SYS_ADMIN))
                return -EPERM;
            tmp = scull_qset;
            scull_qset = arg;
            return tmp;

        /*
         * 다음 두 case는 scullpipe 장치의 버퍼 크기를 변경하는 데 사용.
         * scullpipe 디바이스도 동일 ioctl 메서드를 사용하도록 하여 중복 코드를 줄임.
         */
        case SCULL_P_IOCTSIZE:
            scull_p_buffer = arg;
            break;

        case SCULL_P_IOCQSIZE:
            return scull_p_buffer;

        default:  /* MAXNR을 넘어서는 경우 */
            return -ENOTTY;
    }
    return retval;
}
```

```
/*
 * 추가 기능: llseek 구현 (커널의 기본 noop_llseek를 override)
 */

loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
    struct scull_dev *dev = filp->private_data;
    loff_t newpos;

    switch(whence) {
     case 0: /* SEEK_SET: 파일 시작으로부터 */
        newpos = off;
        break;

     case 1: /* SEEK_CUR: 현재 위치로부터 */
        newpos = filp->f_pos + off;
        break;

     case 2: /* SEEK_END: 파일 끝으로부터 */
        newpos = dev->size + off;
        break;

     default: /* 잘못된 whence */
        return -EINVAL;
    }
    if (newpos < 0) return -EINVAL;
    filp->f_pos = newpos;
    return newpos;
}


struct file_operations scull_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .unlocked_ioctl = scull_ioctl,
    .open =     scull_open,
    .release =  scull_release,
};

/*
 * 모듈 설정 함수.
 */

/*
 * cleanup 함수는 초기화 실패도 처리해야 하므로,
 * 모든 항목이 초기화되지 않았을 수도 있다는 점에 유의하여 동작해야 함.
 */
void scull_cleanup_module(void)
```

```
{
    int i;
    dev_t devno = MKDEV(scull_major, scull_minor);

    /* 문자 디바이스 장치들 정리 */
    if (scull_devices) {
        for (i = 0; i < scull_nr_devs; i++) {
            scull_trim(scull_devices + i);
            cdev_del(&scull_devices[i].cdev);
        }
        kfree(scull_devices);
    }

#ifdef SCULL_DEBUG /* 디버깅 모드인 경우 /proc 항목 제거 */
    scull_remove_proc();
#endif

    /* register_chrdev_region 호출 실패 시에는 cleanup_module이 불리지 않음 */
    unregister_chrdev_region(devno, scull_nr_devs);

    /* 추가 장치들 (파이프, 접근 제어 디바이스 등) 정리 */
    scull_p_cleanup();
    scull_access_cleanup();

}

/*
 * 이 디바이스의 cdev 구조를 설정하는 보조 함수
 */
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
    int err;
    dev_t devno = MKDEV(scull_major, scull_minor + index);

    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    err = cdev_add(&dev->cdev, devno, 1);
    /* 실패 시 메시지 출력 (심각하지 않은 오류) */
    if (err)
        printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}


int scull_init_module(void)
{
    int result, i;
    dev_t dev = 0;

    /*
      * 사용할 minor 번호 범위를 확보. scull_major가 0이면 dynamic major 할당.
```

```
     */
    if (scull_major) {
        dev = MKDEV(scull_major, scull_minor);
        result = register_chrdev_region(dev, scull_nr_devs, "scull");
    } else {
        result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs, "scull");
        scull_major = MAJOR(dev);
    }
    if (result < 0) {
        printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
        return result;
    }

    /*
     * 디바이스들을 할당. major 번호가 동적으로 정해질 수 있으므로 kmalloc 사용.
     */
    scull_devices = kmalloc(scull_nr_devs * sizeof(struct scull_dev), GFP_KERNEL);
    if (!scull_devices) {
        result = -ENOMEM;
        goto fail;
    }
    memset(scull_devices, 0, scull_nr_devs * sizeof(struct scull_dev));

    /* 각 디바이스 초기화 */
    for (i = 0; i < scull_nr_devs; i++) {
        scull_devices[i].quantum = scull_quantum;
        scull_devices[i].qset = scull_qset;
        mutex_init(&scull_devices[i].mutex);
        scull_setup_cdev(&scull_devices[i], i);
    }

    /* friend 장치들 (파이프, 접근제한 장치 등) 초기화 함수 호출 */
    dev = MKDEV(scull_major, scull_minor + scull_nr_devs);
    dev += scull_p_init(dev);
    dev += scull_access_init(dev);

#ifdef SCULL_DEBUG /* 디버깅 모드인 경우 /proc 항목 생성 */
    scull_create_proc();
#endif

    return 0; /* 성공 */

 fail:
    scull_cleanup_module();
    return result;
}

module_init(scull_init_module);
module_exit(scull_cleanup_module);
```

1.3 scull_access.c

```
/* access.c -- open 시 접근 제어가 있는 scull 디바이스들 */

#include <linux/kernel.h> /* printk() */
#include <linux/module.h>
#include <linux/slab.h>   /* kmalloc() */
#include <linux/sched.h>
#include <linux/sched/signal.h>
#include <linux/fs.h>     /* 시스템 호출 관련 모든 것 */
#include <linux/errno.h>  /* 오류 코드 */
#include <linux/types.h>  /* size_t 등 */
#include <linux/fcntl.h>
#include <linux/cdev.h>
#include <linux/tty.h>
#include <asm/atomic.h>
#include <linux/list.h>
#include <linux/cred.h>

#include "scull.h"        /* 로컬 정의 */

static dev_t scull_a_firstdev;  /* 디바이스 번호 시작점 */

/*
 * 첫 번째 장치: single-open 장치 (동시에 한 프로세스만 열 수 있음)
 * scull_s_device: 단일 장치 구조체, scull_s_available: 원자적 카운트 (1이면 사용 가능)
 */
static struct scull_dev scull_s_device;
static atomic_t scull_s_available = ATOMIC_INIT(1);

static int scull_s_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev = &scull_s_device;

    if (! atomic_dec_and_test(&scull_s_available)) {
        atomic_inc(&scull_s_available);
        return -EBUSY; /* 이미 열려 있음 */
    }

    /* 이후 부분은 기본 scull 디바이스와 동일 */
    if ((filp->f_flags & O_ACCMODE) == O_WRONLY)
        scull_trim(dev);
    filp->private_data = dev;
    return 0;       /* 성공 */
}

static int scull_s_release(struct inode *inode, struct file *filp)
{
    atomic_inc(&scull_s_available); /* 장치 사용 가능 상태로 복원 */
    return 0;
}
```

```
/* single-open 장치의 나머지 file_operations (llseek, read, write 등) 기본 scull과 동일 */
struct file_operations scull_sngl_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .unlocked_ioctl =   scull_ioctl,
    .open =     scull_s_open,
    .release =  scull_s_release,
};

/*
 * 두 번째 장치: "uid" 장치 (동일 사용자 여러 번 열 수 있지만 다른 사용자 접근 시 이미 열려 있으면 거부)
 */
static struct scull_dev scull_u_device;
static int scull_u_count;
static kuid_t scull_u_owner;
static DEFINE_SPINLOCK(scull_u_lock);

static int scull_u_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev = &scull_u_device;

    spin_lock(&scull_u_lock);
    if (scull_u_count &&
            (!uid_eq(scull_u_owner, current_uid())) &&  /* 현재 사용자 UID가 아닌 경우 */
            (!uid_eq(scull_u_owner, current_euid())) && /* 현재 유효 사용자 ID가 아닌 경우 */
            !capable(CAP_DAC_OVERRIDE)) { /* root 권한도 아닌 경우 */
        spin_unlock(&scull_u_lock);
        return -EBUSY;   /* -EPERM은 사용자에게 혼란을 줄 수 있음 */
    }

    if (scull_u_count == 0)
        scull_u_owner = current_uid(); /* 첫 오픈 시 소유자 UID 저장 */

    scull_u_count++;
    spin_unlock(&scull_u_lock);

    /* 이후 부분은 기본 scull 디바이스와 동일 */
    if ((filp->f_flags & O_ACCMODE) == O_WRONLY)
        scull_trim(dev);
    filp->private_data = dev;
    return 0;
}

static int scull_u_release(struct inode *inode, struct file *filp)
{
    spin_lock(&scull_u_lock);
    scull_u_count--;
    spin_unlock(&scull_u_lock);
```

```
    return 0;
}

/* "uid" 장치의 file_operations (open은 scull_u_open, release는 scull_u_release, 나머지는 기본 scull)
*/
struct file_operations scull_user_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .unlocked_ioctl =    scull_ioctl,
    .open =     scull_u_open,
    .release =  scull_u_release,
};

/*
 * 세 번째 장치: "uid 기반 blocking-open" 장치 (scull_w)
 * - 이미 한 사용자가 열었을 때 다른 사용자가 열려고 하면 대기 (sleep) 후 열리길 기다림
 */
static struct scull_dev scull_w_device;
static int scull_w_count;
static kuid_t scull_w_owner;
static DECLARE_WAIT_QUEUE_HEAD(scull_w_wait);
static DEFINE_SPINLOCK(scull_w_lock);

static inline int scull_w_available(void)
{
    return scull_w_count == 0 ||
        uid_eq(scull_w_owner, current_uid()) ||
        uid_eq(scull_w_owner, current_euid()) ||
        capable(CAP_DAC_OVERRIDE);
}

static int scull_w_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev = &scull_w_device;

    spin_lock(&scull_w_lock);
    while (! scull_w_available()) {
        spin_unlock(&scull_w_lock);
        if (filp->f_flags & O_NONBLOCK) return -EAGAIN;
        if (wait_event_interruptible(scull_w_wait, scull_w_available()))
            return -ERESTARTSYS; /* 신호 발생 시 재시도 */
        spin_lock(&scull_w_lock);
    }

    if (scull_w_count == 0)
        scull_w_owner = current_uid();
    scull_w_count++;
    spin_unlock(&scull_w_lock);
```

```
    /* 이후 기본 scull과 동일 */
    if ((filp->f_flags & O_ACCMODE) == O_WRONLY)
        scull_trim(dev);
    filp->private_data = dev;
    return 0;
}

static int scull_w_release(struct inode *inode, struct file *filp)
{
    int temp;
    spin_lock(&scull_w_lock);
    temp = --scull_w_count;
    spin_unlock(&scull_w_lock);
    if (temp == 0)
        wake_up_interruptible_sync(&scull_w_wait);
    return 0;
}

/* "uid blocking" 장치의 file_operations */
struct file_operations scull_wusr_fops = {
    .owner =     THIS_MODULE,
    .llseek =    scull_llseek,
    .read =      scull_read,
    .write =     scull_write,
    .unlocked_ioctl =    scull_ioctl,
    .open =      scull_w_open,
    .release =   scull_w_release,
};

/*
 * scull_access_init: 접근 제어 장치들 초기화 (scull_s, scull_u, scull_w)
 * dev 매개변수는 처음 사용할 dev 번호
 */
int scull_access_init(dev_t firstdev)
{
    int result;

    /* single-open 장치 설정 */
    scull_s_device.quantum = scull_quantum;
    scull_s_device.qset = scull_qset;
    cdev_init(&scull_s_device.cdev, &scull_sngl_fops);
    scull_s_device.cdev.owner = THIS_MODULE;
    result = cdev_add(&scull_s_device.cdev, firstdev, 1);
    if (result)
      printk(KERN_NOTICE "Error %d adding scullsingle", result);
    firstdev++;

    /* uid 장치 설정 */
    scull_u_device.quantum = scull_quantum;
    scull_u_device.qset = scull_qset;
    cdev_init(&scull_u_device.cdev, &scull_user_fops);
```

```
    scull_u_device.cdev.owner = THIS_MODULE;
    result = cdev_add(&scull_u_device.cdev, firstdev, 1);
    if (result)
      printk(KERN_NOTICE "Error %d adding sculluid", result);
    firstdev++;

    /* uid blocking-open 장치 설정 */
    scull_w_device.quantum = scull_quantum;
    scull_w_device.qset = scull_qset;
    cdev_init(&scull_w_device.cdev, &scull_wusr_fops);
    scull_w_device.cdev.owner = THIS_MODULE;
    result = cdev_add(&scull_w_device.cdev, firstdev, 1);
    if (result)
      printk(KERN_NOTICE "Error %d adding scullwuid", result);
    firstdev++;

    return firstdev - scull_a_firstdev;
}

/*
 * scull_access_cleanup: 접근 제어 장치들 정리 (cdev 제거)
 */
void scull_access_cleanup(void)
{
    cdev_del(&scull_s_device.cdev);
    cdev_del(&scull_u_device.cdev);
    cdev_del(&scull_w_device.cdev);
    return;
}
```

1.4 scull_pipe.c

```
/* pipe.c -- scull 파이프 (FIFO) 디바이스 구현 */

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/fs.h>
#include <linux/proc_fs.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include <linux/poll.h>
#include <linux/cdev.h>
#include <linux/sched.h>
#include <asm/uaccess.h>

#include "scull.h"
```

```c
struct scull_pipe {
  wait_queue_head_t inq, outq;  /* 읽기 및 쓰기 대기 큐 */
  char *buffer, *end;       /* 버퍼 시작 포인터, 끝 포인터 */
  int buffersize;           /* 버퍼 크기 */
  char *rp, *wp;            /* 읽기 위치, 쓰기 위치 포인터 */
  int nreaders, nwriters;     /* 현재 읽기/쓰기 열기 수 */
  struct fasync_struct *async_queue; /* 비동기 읽기 대기 리스트 */
  struct semaphore sem;       /* 접근 동기화 세마포어 */
  struct cdev cdev;
};

static struct class *scull_pipe_class = NULL;
static int scull_p_nr_devs = SCULL_P_NR_DEVS;
int scull_p_buffer = SCULL_P_BUFFER;
static dev_t scull_p_devno;

module_param(scull_p_nr_devs, int, 0);
module_param(scull_p_buffer, int, 0);

static struct scull_pipe *scull_p_devices;

/* forward declarations for internal helper functions */
static int scull_p_fasync(int fd, struct file *filp, int mode);
static int spacefree(struct scull_pipe *dev);

/* open function for pipe device */
static int scull_p_open(struct inode *inode, struct file *filp)
{
  struct scull_pipe *dev;

  dev = container_of(inode->i_cdev, struct scull_pipe, cdev);
  filp->private_data = dev;

  if (down_interruptible(&dev->sem))
    return -ERESTARTSYS;
  if (!dev->buffer) {
    /* 버퍼 할당 */
    dev->buffer = kmalloc(scull_p_buffer, GFP_KERNEL);
    if (!dev->buffer) {
      up(&dev->sem);
      return -ENOMEM;
    }
  }
  dev->buffersize = scull_p_buffer;
  dev->end = dev->buffer + dev->buffersize;
  dev->rp = dev->wp = dev->buffer; /* head와 tail 초기화 */

  /* f_mode를 사용하여 읽기/쓰기 플래그 확인 (open.c 참고) */
  if (filp->f_mode & FMODE_READ)
    dev->nreaders++;
  if (filp->f_mode & FMODE_WRITE)
```

```
        dev->nwriters++;
    up(&dev->sem);

    return nonseekable_open(inode, filp);
}

/* release function for pipe device */
static int scull_p_release(struct inode *inode, struct file *filp)
{
    struct scull_pipe *dev = filp->private_data;

    scull_p_fasync(-1, filp, 0);
    down(&dev->sem);
    if (filp->f_mode & FMODE_READ)
        dev->nreaders--;
    if (filp->f_mode & FMODE_WRITE)
        dev->nwriters--;
    if (dev->nreaders + dev->nwriters == 0) {
        kfree(dev->buffer);
        dev->buffer = NULL;
    }
    up(&dev->sem);
    return 0;
}

/* poll (select) support: returns mask of events */
static unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
    struct scull_pipe *dev = filp->private_data;
    unsigned int mask = 0;

    down(&dev->sem);
    poll_wait(filp, &dev->inq, wait);
    poll_wait(filp, &dev->outq, wait);

    /* 읽기 가능 여부 확인 */
    if (dev->rp != dev->wp)
        mask |= POLLIN | POLLRDNORM;  /* 읽기 가능 (normal data) */

    /* 쓰기 가능 여부 확인 (버퍼 여유 공간) */
    if (spacefree(dev))
        mask |= POLLOUT | POLLWRNORM; /* 쓰기 가능 (normal) */
    up(&dev->sem);
    return mask;
}

/* read function for pipe device */
ssize_t scull_p_read(struct file *filp, char __user *buf, size_t count,
                     loff_t *f_pos)
{
    struct scull_pipe *dev = filp->private_data;
```

```
  if (down_interruptible(&dev->sem))
    return -ERESTARTSYS;
  while (dev->rp == dev->wp) { /* 버퍼가 비어 있음 */
    up(&dev->sem);
    if (filp->f_flags & O_NONBLOCK)
      return -EAGAIN;
    PDEBUG("\"%s\" reading: going to sleep\n", current->comm);
    if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))
      return -ERESTARTSYS;
    if (down_interruptible(&dev->sem))
      return -ERESTARTSYS;
  }

  /* 읽을 데이터 크기 계산 */
  if (dev->wp > dev->rp)
    count = min(count, (size_t)(dev->wp - dev->rp));
  else
    count = min(count, (size_t)(dev->end - dev->rp));
  if (copy_to_user(buf, dev->rp, count)) {
    up(&dev->sem);
    return -EFAULT;
  }
  dev->rp += count;
  if (dev->rp == dev->end)
    dev->rp = dev->buffer;
  up(&dev->sem);

  /* 읽기 대기 중인 writer 깨우기 */
  wake_up_interruptible(&dev->outq);
  PDEBUG("\"%s\" did read %li bytes\n", current->comm, (long)count);
  return count;
}

/* write function for pipe device */
ssize_t scull_p_write(struct file *filp, const char __user *buf, size_t count,
                      loff_t *f_pos)
{
  struct scull_pipe *dev = filp->private_data;
  int result;

  if (down_interruptible(&dev->sem))
    return -ERESTARTSYS;
  /* 버퍼가 다 찬 경우 */
  result = spacefree(dev);
  if (!result) {
    /* 버퍼에 여유 공간이 생길 때까지 대기 */
    up(&dev->sem);
    if (filp->f_flags & O_NONBLOCK)
      return -EAGAIN;
    PDEBUG("\"%s\" writing: going to sleep\n", current->comm);
```

```
        if (wait_event_interruptible(dev->outq, spacefree(dev)))
            return -ERESTARTSYS;
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    }
    /* 쓰기 가능한 공간 계산 */
    count = min(count, (size_t)result);
    if (dev->wp >= dev->rp) {
        count = min(count, (size_t)(dev->end - dev->wp));
    } else {
        count = min(count, (size_t)(dev->rp - dev->wp - 1));
    }
    PDEBUG("Going to accept %li bytes to %p from %p\n", (long)count, dev->wp, buf);
    if (copy_from_user(dev->wp, buf, count)) {
        up(&dev->sem);
        return -EFAULT;
    }
    dev->wp += count;
    if (dev->wp == dev->end)
        dev->wp = dev->buffer;
    up(&dev->sem);

    /* 쓰기 대기 중인 reader 깨우기 */
    wake_up_interruptible(&dev->inq);

    /* SIGIO (Async I/O) 알림 */
    if (dev->async_queue)
        kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
    PDEBUG("\"%s\" did write %li bytes\n", current->comm, (long)count);
    return count;
}

/* Helper: compute free space in buffer */
static int spacefree(struct scull_pipe *dev)
{
    if (dev->rp == dev->wp)
        return dev->buffersize - 1;
    return (dev->rp + dev->buffersize - dev->wp) % dev->buffersize - 1;
}

/* fasync support for asynchronous I/O (SIGIO) */
static int scull_p_fasync(int fd, struct file *filp, int mode)
{
    struct scull_pipe *dev = filp->private_data;
    return fasync_helper(fd, filp, mode, &dev->async_queue);
}

/* file_operations for scull_pipe devices */
struct file_operations scull_pipe_fops = {
    .owner =    THIS_MODULE,
    .llseek =   no_llseek,
```

```
    .read =     scull_p_read,
    .write =    scull_p_write,
    .poll =     scull_p_poll,
    .unlocked_ioctl = scull_ioctl,
    .open =     scull_p_open,
    .release =  scull_p_release,
    .fasync =   scull_p_fasync
};

/* scull_pipe module initialization */
int scull_p_init(dev_t firstdev)
{
    int i, result;

    result = 0;
    scull_p_devno = firstdev;
    scull_p_devices = kmalloc(scull_p_nr_devs * sizeof(struct scull_pipe), GFP_KERNEL);
    if (scull_p_devices == NULL) {
        return -ENOMEM;
    }
    memset(scull_p_devices, 0, scull_p_nr_devs * sizeof(struct scull_pipe));
    /* 장치 초기화 */
    for (i = 0; i < scull_p_nr_devs; i++) {
        init_waitqueue_head(&(scull_p_devices[i].inq));
        init_waitqueue_head(&(scull_p_devices[i].outq));
        sema_init(&(scull_p_devices[i].sem), 1);
        /* cdev 설정 */
        cdev_init(&scull_p_devices[i].cdev, &scull_pipe_fops);
        scull_p_devices[i].cdev.owner = THIS_MODULE;
        result = cdev_add(&scull_p_devices[i].cdev, firstdev + i, 1);
        if (result) {
            printk(KERN_NOTICE "Error %d adding scullpipe%d", result, i);
        } else {
            /* class_create 및 device_create로 /dev 노드 자동 생성 (예: udev 사용) */
            if (!scull_pipe_class) {
                scull_pipe_class = class_create(THIS_MODULE, "scull_pipe");
            }
            if (!IS_ERR(scull_pipe_class)) {
                device_create(scull_pipe_class, NULL, firstdev + i, NULL, "scullpipe%d", i);
            }
        }
    }
    return scull_p_nr_devs;
}

/* scull_pipe module cleanup */
void scull_p_cleanup(void)
{
    int i;
    if (scull_pipe_class) {
        for (i = 0; i < scull_p_nr_devs; i++) {
```

```
        device_destroy(scull_pipe_class, scull_p_devno + i);
      }
      class_destroy(scull_pipe_class);
      scull_pipe_class = NULL;
    }
    if (scull_p_devices) {
      for (i = 0; i < scull_p_nr_devs; i++) {
        cdev_del(&scull_p_devices[i].cdev);
        kfree(scull_p_devices[i].buffer);
      }
      kfree(scull_p_devices);
      scull_p_devices = NULL;
    }
}
```

시스템프로그램 : Filesystem

Physical Disk Structure (디스크 구조)

- **Parameters to read from disk (디스크에서 데이터 읽기 위한 파라미터):** cylinder (트랙) 번호, platter (면) 번호, sector 번호, transfer size (전송 크기)

File system Units (파일 시스템 단위)

- **Sector (섹터):** 디스크에서 접근 가능한 가장 작은 단위 (일반적으로 512 바이트)
- **Block (or Cluster, 블록/클러스터):** 파일 구성에 할당될 수 있는 가장 작은 단위 (여러 섹터로 구성됨)
- **예:** 1바이트 크기의 파일도 최소 1클러스터를 차지함. 1클러스터가 1~8개 섹터로 이루어질 수 있으므로, 실제로 1바이트 파일이 약 4KB의 디스크 공간을 차지할 수 있음.

Sector~Cluster~File layout (섹터, 클러스터와 파일 구성)

(섹터와 클러스터가 파일의 저장에 어떻게 대응하는지 보여주는 레이아웃 그림)

FCB – File Control Block (파일 제어 블록)

- 파일의 **속성 및 블록 위치 정보**를 담는 구조체 (파일 시스템에서 파일을 관리하기 위한 메타데이터)
- Permissions (파일 권한 정보)
- Dates (생성일, 접근일, 수정일 등 날짜 정보)
- Owner, group, ACL (소유자, 그룹, 접근 제어 목록)
- File size (파일 크기)
- Location of file contents (파일 내용이 저장된 블록들의 위치)
- **UNIX 파일 시스템:** I-node 구조체 형태로 FCB 역할 수행
- **FAT/FAT32:** FAT (File Allocation Table)의 엔트리로 파일 할당 정보를 관리
- **NTFS:** MFT (Master File Table)의 엔트리로 파일 정보를 관리

Partitions (디스크 파티션)

- 디스크는 하나 이상의 **파티션**으로 나뉠 수 있음
- 각 파티션마다 독립적인 파일 시스템 형식을 가질 수 있음 (예: UFS, FAT, NTFS 등)

A Disk Layout for A File System (파일 시스템의 디스크 상 레이아웃)

- **Boot block (부트 블록)**
- **Super block (슈퍼블록)**
- **File descriptors (FCBs) 영역**
- **File data blocks (파일 데이터 블록들)**

(위 4가지 영역이 디스크에 순서대로 배치된 그림)

- **Super block:** 파일 시스템에 대한 메타데이터를 담고 있음. 예를 들어:
- 파일 시스템의 전체 크기
- 파일 디스크립터(FCB) 영역의 크기
- 자유 블록 목록의 시작 위치
- 루트 디렉터리의 FCB 위치

- 그 외 권한, 시간정보 등의 메타데이터

- **Boot block:** 디스크의 부트스트랩 코드가 위치하는 영역으로, 시스템 부팅 시 읽혀짐

- Dual Boot: 한 시스템에 여러 OS가 설치된 경우, 어떤 OS를 부팅할지 결정
- Boot Loader: 다양한 OS와 파일 시스템을 인식하여 부팅을 수행하는 코드. 디스크의 특정 위치(부트 블록)에 위치하며 부트 이미지를 찾아 실행

Block Allocation (디스크 블록 할당 방식)

- **연속 할당 (Contiguous allocation):** 파일을 디스크 상의 연속된 블록들에 할당
- **연결 할당 (Linked allocation):** 파일을 디스크 상의 블록들에 링크드 리스트 형태로 연결하여 할당
- **인덱스 할당 (Indexed allocation):** 파일의 블록 목록을 별도의 인덱스 블록에 배열 형태로 관리하여 할당

Contiguous Block Allocation (연속 블록 할당)

- **장점(Pros):** 연속된 블록에 저장하므로 디스크 **탐색/접근 속도가 매우 빠름** (순차/임의 접근 모두 블록 위치를 바로 계산 가능 → 공간적 지역성 향상)
- **단점(Cons):** 파일 생성 시 **필요 블록 수를 미리 알기 어려움**, **연속 블록이 부족할 경우 문제 발생** (중간에 공간 부족 시 파일 저장 곤란) → 디스크 **단편화(fragmentation)** 초래

Linked Block Allocation (연결 블록 할당)

- **장점: 디스크 단편화(fragmentation)가 적음**, 파일 크기를 유동적으로 확장 가능 (연속 공간 불필요)
- **단점: 순차 읽기 시** 다음 블록으로 이동하기 위해 디스크 탐색(seek)이 필요 (약간의 성능 저하), **임의 접근 (random access)** 성능 매우 낮음 (모든 블록을 순차적으로 찾아야 하므로 **O(n)** 탐색 필요, n = 파일의 블록 수)

Indexed Block Allocation (인덱스 블록 할당)

- 파일이 사용하는 블록들의 포인터를 **배열 형태의 인덱스 블록**에 보관
- **임의 접근(Random access)** 성능이 순차 접근만큼 용이 (인덱스 통해 즉시 블록 위치 확인 가능)
- **UNIX 파일 시스템**에서 사용하는 방식 (i-node가 인덱스 역할 수행)

Linux 디바이스 드라이버 기초

소개

- 디바이스 드라이버는 **캐릭터(char) 드라이버**와 **블록(block) 드라이버**로 구분됨
- 커널은 드라이버를 식별하기 위해 **주 번호(major number)**를 사용
- 하나의 드라이버가 제어하는 여러 디바이스를 구분하기 위해 **부 번호(minor number)**를 사용
- 커널은 또한 각 드라이버에 대한 **드라이버 이름**을 필요로 함
- 사용자 공간에서는 **디바이스 노드(device node)**를 인터페이스로 사용하여 드라이버에 접근

디바이스 노드 생성

- `'mknod'` 명령어를 사용하여 디바이스 노드를 생성할 수 있음:

```
$ mknod /dev/stash c 40 0
```

(위 예시는 이름이 "stash"인 캐릭터 장치(c), major 번호 40, minor 번호 0인 디바이스 노드를 생성하는 명령어)

- 추가로, 생성된 노드의 접근 권한을 변경하려면 `'chmod'` 명령을 사용:

```
$ chmod a+rw /dev/stash
```

(위 명령은 `/dev/stash` 디바이스 노드에 모든 사용자 읽기/쓰기 권한을 부여함)

- 참고: 위 `mknod`, `chmod` 명령들은 **관리자 권한(슈퍼유저)**이 필요함

'open' 함수 (디바이스 열기)

- 헤더 파일 포함: `#include <fcntl.h>`
- 함수 원형: `int open(const char *pathname, int flags);`
- `open()` 시스템 호출은 파일 경로명을 파일 디스크립터로 변환함
- **파일 디스크립터(file descriptor)**란 열린 파일을 가리키는 음수가 아닌 정수 값
- 이후 입출력 함수들에서 이 파일 디스크립터를 파일 식별자로 사용
- `flags` 인자는 파일을 여는 방식에 대한 상수 값을 지정 (읽기/쓰기 등)

주요 flags 예시: - `O_RDONLY` (읽기 전용 열기) - `O_WRONLY` (쓰기 전용 열기) - `O_RDWR` (읽기/쓰기로 열기)

'close' 함수 (디바이스 닫기)

- 헤더 파일 포함: `#include <unistd.h>`
- 함수 원형: `int close(int fd);`
- `close()` 시스템 호출은 파일 디스크립터와 파일의 연결을 해제함
- 성공 시 0을 반환, 오류 시 -1을 반환

'read' 함수 (디바이스에서 데이터 읽기)

- 헤더 파일 포함: `#include <unistd.h>`

- 함수 원형: `ssize_t read(int fd, void *buf, size_t count);`
- `read()` 시스템 호출은 최대 `count` 바이트만큼 데이터를 읽으려고 시도함
- 읽은 바이트들은 메모리 버퍼 `buf`에 채워 넣음
- 반환값은 읽어들인 바이트 수이며, 오류 시 -1을 반환
- 반환값이 0이면 **EOF(End-of-File, 파일 끝)**에 도달했음을 의미

'write' 함수 (디바이스에 데이터 쓰기)

- 헤더 파일 포함: `#include <unistd.h>`
- 함수 원형: `ssize_t write(int fd, const void *buf, size_t count);`
- `write()` 시스템 호출은 최대 `count` 바이트만큼 데이터를 쓰려고 시도함
- 쓸 데이터는 메모리 버퍼 `buf`에서 가져옴
- 반환값은 실제로 기록한 바이트 수이며, 오류 시 -1을 반환
- 반환값이 0이면 **어떠한 데이터도 기록되지 않았음**을 의미

'lseek' 함수 (파일 오프셋 이동)

- 헤더 파일 포함: `#include <unistd.h>`
- 함수 원형: `off_t lseek(int fd, off_t offset, int whence);`
- `lseek()` 시스템 호출은 열린 파일의 **파일 포인터(file pointer)**를 이동시킴
- 세 가지 이동 기준(`whence` 인자 값)이 있음:
- `SEEK_SET` : 파일 **시작 위치**로부터 `offset`만큼 이동
- `SEEK_CUR` : 파일 **현재 위치**로부터 `offset`만큼 이동
- `SEEK_END` : 파일 **끝 위치**로부터 `offset`만큼 이동
- 이동 후의 새로운 파일 오프셋을 반환, 잘못된 요청인 경우 -1 반환

블로킹(Blocking) I/O와 대기 (Wait Queue)

- **기본 동작 모드**: 리눅스의 `read()`와 `write()` 시스템 호출은 기본적으로 **블로킹 모드**로 동작
- `read()`는 읽을 데이터가 없을 때 곧바로 0을 반환하지 않고 (EOF가 아닌 이상), **데이터가 올 때까지 기다림**
- `write()`도 기록 공간이 가득 차 있으면 0을 반환하지 않고 (더 이상 기록 불가한 경우 제외), **공간이 생길 때까지 기다림**
- 이러한 **대기(wait)** 동작을 구현하기 위해 커널은 해당 프로세스를 **sleep 상태**로 전환함 (바쁜 대기 busy-wait로 CPU를 낭비하지 않도록)
- 프로세스를 sleep으로 만들면 해당 프로세스는 다시 스케줄되지 않음 (대기열에서 빠짐)
- 이때 다른 이벤트(시그널 등)가 발생하면 sleep 상태의 프로세스를 깨워줄 수 있음

멀티태스킹과 스케줄링 배경

- 멀티태스킹 방식:
- **협력형(cooperative)**: 각 태스크가 스스로 CPU를 내줄 때 컨텍스트 전환
- **선점형(preemptive)**: 커널 스케줄러가 강제로 컨텍스트 전환
- **인터럽트(interrupted)**는 일시적으로 CPU 실행이 중단된 것일 뿐, **선점(preempted)**과는 구별됨
- **선점됨(preempted)**은 현재 실행 중인 태스크가 CPU에서 물러나고 다른 태스크로 전환되었음을 의미

프로세스 태스크의 상태

- 태스크는 여러 가지 **상태(state)**를 가짐:
- **Running:** 현재 CPU를 사용하여 실행 중인 상태
- **Ready (ready-to-run):** 실행 가능한 상태지만 CPU를 기다리는 상태 (실행 대기)
- **Blocked:** 어떤 이벤트나 자원(예: I/O 완료)을 기다리느라 실행 불가 상태 (sleep 상태)

- 커널은 태스크들을 상태별로 관리하기 위해 **큐(queue)**를 사용
- **Run Queue:** 실행 대기 중(ready)의 태스크들을 담은 큐 (스케줄러는 이 큐에서 태스크를 선택해 실행)
- **Wait Queue:** 특정 이벤트를 기다리며 블록된 태스크들을 담은 큐 (이벤트 발생 시 해당 큐의 태스크를 깨움)

wait queue (대기 큐)의 사용 (디바이스 드라이버에서 sleep/wake 구현)

- **디바이스 드라이버**는 필요 시 프로세스를 **sleep 상태**로 만들고, 조건이 충족되면 **wake up** 시켜 다시 실행되도록 할 수 있음
- 리눅스 커널은 이러한 메커니즘을 위해 **특수 대기 큐(wait queue)**를 제공

리눅스 wait-queue 사용 방법

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

wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);

sleep_on(&my_queue);
wake_up(&my_queue);
```

- 위와 같이 `wait_queue_head_t` 구조체를 초기화하고, 필요 시 `sleep_on()`으로 프로세스를 재우고 `wake_up()`으로 깨움 - **주의:** `sleep_on()`으로 재운 프로세스는 해당 드라이버 모듈이 언로드될 경우 문제가 됨 (sleep 상태로 남아있으면 안전하게 모듈 제거 불가)

인터럽트 가능한 대기 큐 (interruptible waits)

- 따라서 실제 드라이버에서는 `sleep_on()` 대신 **인터럽트 가능 대기 함수**를 사용:

```
interruptible_sleep_on(&my_queue);
wake_up_interruptible(&my_queue);
```

- 이렇게 하면 대기 중인 태스크가 **시그널(signal)**에 의해 깨울 수 있도록 함 (인터럽트 가능 상태로 sleep)

'sleep' 동작 방식 (커널에서 프로세스 sleep 처리)

- 드라이버는 커널의 `wait queue head` 구조체 인스턴스를 정의함 (앞서 `my_queue`와 같은)
- 이 wait queue head는 해당 대기 리스트의 **anchor(앵커)** 역할을 함 (대기 중인 `task_struct`들의 연결 리스트를 관리)
- 처음에는 이 리스트가 **비어 있음**
- 드라이버가 어떤 태스크를 sleep 시키면:
- 해당 프로세스의 `task_struct`가 **실행 큐(runqueue)**에서 제거되어 우리가 정의한 wait queue에 추가됨

'wake up' 동작 방식 (커널에서 프로세스 깨우기 처리)

- 드라이버가 전에 재웠던 태스크가 (예: 즉시 I/O를 처리할 수 없어서 재운 경우) 이제 작업을 진행해도 된다고 판단하면, 해당 wait queue에 대해 `wake_up()`을 호출함
- 그러면 그 wait queue에 들어있던 모든 `task_struct`들이 wait queue에서 제거되어 다시 **CPU의 실행 대기 큐(runqueue)**에 추가됨 (스케줄러에 의해 곧 실행 재개)

링 버퍼(Ring Buffer)에의 적용 (예시)

- 링 버퍼: **선입선출(FIFO)** 자료 구조의 한 형태
- 고정된 크기의 배열을 사용하여 구현
- **head 인덱스**와 **tail 인덱스** 두 개로 관리 (데이터가 순환하는 구조)
- 데이터는 현재 `tail` 위치에 추가되고, `head` 위치에서 제거됨
- 링 버퍼의 특징:
- 항상 배열의 한 위치는 비워 둠 (full과 empty 구분을 위해)
- **head == tail**이면 버퍼가 "비어 있음(empty)"
- **tail == head-1**이면 버퍼가 "가득 참(full)"
- `head`와 `tail`이 배열 끝을 넘어가면 처음으로 돌아감 (**wraparound**)
- 예시 계산: `next = (next + 1) % RINGSIZE;` (배열 인덱스 순환 증가)

Character Device Driver (문자 디바이스 드라이버) - 주요 개념

Driver Identification (드라이버 식별)

- **캐릭터/블록 드라이버**는 major/minor 번호로 구분됨 (앞서 설명된 내용과 동일)

Creating our device node (디바이스 노드 생성)

- `mknod` 및 `chmod` 명령을 사용하여 디바이스 노드를 생성하고 권한을 설정 (앞서 설명됨)

The ‘open’ function ('open' 시스템 호출)

- `open()` 호출 시 파일 경로명이 커널에 전달되어 새로운 파일 디스크립터가 할당됨
- major 번호를 통해 커널이 해당 디바이스 드라이버로 접근
- 드라이버의 `open` 메서드가 호출되어 필요한 초기화를 수행

The ‘close’ function ('close' 시스템 호출)

- `close()` 호출 시 커널이 파일 디스크립터와 디바이스의 연결을 해제
- 드라이버의 `release` 메서드가 호출되어 자원 해제나 정리 작업 수행

The ‘read’ and ‘write’ functions ('read'와 'write' 시스템 호출)

- `read(fd, buf, count)`:
- 디바이스에서 최대 `count` 바이트를 읽어 `buf`에 저장
- 드라이버의 `read` 메서드가 호출되어 실제 장치에서 데이터 가져옴
- 반환값: 읽은 바이트 수 (0이면 EOF, -1이면 에러)
- `write(fd, buf, count)`:
- `buf`로부터 최대 `count` 바이트를 디바이스에 출력
- 드라이버의 `write` 메서드가 호출되어 실제 장치에 데이터 출력
- 반환값: 출력한 바이트 수 (0이면 출력 없음, -1이면 에러)

Non-blocking I/O (논블로킹 I/O)

- `open()` 시 `O_NONBLOCK` 플래그를 지정하면, `read()`나 `write()`가 즉시 수행되지 않을 경우 블록되지 않고 -EAGAIN 등의 오류를 반환
- 드라이버 구현 시 `filp->f_flags & O_NONBLOCK`를 확인하여 대기 여부를 결정

- 예: scull_w 장치의 `open`에서, 이미 다른 사용자에 의해 열려 있는 경우 O_NONBLOCK이면 바로 -EAGAIN 반환, 아니면 wait queue에서 대기

Wait Queues and Sleep/Wakeup (대기 큐와 슬립/웨이크업)

- 드라이버 내부에서 `wait_event_interruptible(queue, condition)` 매크로 등을 활용하여 **조건(condition)**이 만족될 때까지 프로세스를 sleep 시킴
- `wake_up_interruptible(queue)`를 호출하여 대기 중인 프로세스들을 깨움
- 사용 예시는 앞서 **wait queue 사용 방법** 섹션에서 설명됨

Asynchronous Notification (비동기 통보, SIGIO)

- 드라이버는 `fasync_helper`와 `kill_fasync`를 사용하여 SIGIO 신호를 통해 비동기 입출력 이벤트를 통보 가능
- 예: scull_pipe 장치에서 데이터가 쓰이면 `kill_fasync(&async_queue, SIGIO, POLL_IN)`으로 읽기 가능 신호 보냄
- 사용자 프로그램은 `fcntl(fd, F_SETFL, O_ASYNC)`로 해당 FD에 대한 SIGIO 수신 설정 가능

(위 내용은 Character Device Driver.pdf의 주요 슬라이드 내용을 한국어로 정리한 것입니다.)

Docker (오픈소스 소프트웨어 가이드)

1. 개요

소개:
- Docker(도커)는 2013년 3월 Docker, Inc에서 출시한 오픈 소스 컨테이너 프로젝트이다. 현재 전 세계적으로 큰 인기를 끌고 있으며 AWS, Google Cloud Platform, Microsoft Azure 등의 클라우드 서비스에서 공식 지원하고 있다.

- **주요 기능:** Docker는 이미지 생성, 컨테이너 실행, Docker 이미지의 공유 및 배포 등의 주요 기능을 제공한다.

- **대분류:** 미들웨어

- **소분류:** 클라우드 서비스
- **라이선스 형태:** 아파치 라이선스 2.0
- **사전설치 솔루션:** N/A (별도의 선행 설치 필요 없음)
- **실행 운영체제:** Windows, Linux 등 다양한 OS 지원
- **버전:** 최신 버전 기준

2. 기능요약

Docker의 주요 기능은 다음과 같다:

- **운영 표준화:** 컨테이너화된 작은 애플리케이션을 사용하면 손쉽게 배포하고, 문제를 파악하며, 수정이 필요할 경우 롤백할 수 있다.

- **지속적인 통합 및 제공(CI/CD):** 환경을 표준화하고 언어 스택 및 버전 간 충돌을 제거함으로써 애플리케이션을 더욱 빠르게 제공할 수 있다.

- **마이크로서비스 아키텍처:** Docker 컨테이너를 통해 표준화된 코드 배포를 활용하여 분산 애플리케이션 아키텍처를 구축하고 확장할 수 있다.

3. 실행환경

Docker를 실행하기 위한 호스트 OS 및 하드웨어 요구사항은 다음과 같다:

- **Windows:** Windows 10 64-bit Pro, Enterprise 또는 Education (1607 이상 빌드). CPU 아키텍처 x86_64.
- **macOS:** El Capitan (10.11) 이상 버전의 macOS (2010년 이후 출시된 하드웨어). CPU 아키텍처 x86_64.
- **Fedora:** 26, 27, 28 버전 (x86_64 또는 amd64).
- **CentOS:** 7 버전 (x86_64 또는 amd64).
- **Debian:** Stretch(9 안정), Buster(10) 등 (x86_64, armhf 등 지원).
- **Ubuntu:** 14.04(Trusty), 16.04(Xenial), 18.04(Bionic) 등 LTS 버전 (x86_64, armhf, s390x, ppc64le 등 지원).

(이 밖에도 Docker CE가 지원되는 다양한 리눅스 배포판과 해당 아키텍처들이 있음. 위 목록은 대표적인 예입니다.)

4. 설치 및 실행

Docker 설치 및 실행 가이드에는 운영체제별 상세 단계가 포함된다. 아래는 CentOS에서 Docker를 설치하는 절차의 예시이다:

- **CentOS 설치 (예시):**
- **패키지 설치:** `yum` 패키지 매니저를 통해 필요 패키지 설치:

```
sudo yum install -y yum-utils device-mapper-persistent-data lvm2
```

- **Docker CE 저장소 추가:**

```
sudo yum-config-manager --add-repo https://download.docker.com/linux/
centos/docker-ce.repo
```

- **Docker CE 설치:**

```
sudo yum install -y docker-ce
```

- **Docker 데몬 실행 및 부팅 시 자동 시작 설정:**

```
sudo systemctl start docker
sudo systemctl enable docker
```

- **Ubuntu 설치 (유사하게 apt 패키지 사용)**, **Windows 설치 (전용 설치 프로그램 사용)** 등도 가이드에 포함되어 있으며, 각 OS별로 Docker 설치 방법이 제공된다.

5. 기능소개 (Docker 주요 명령어)

Docker의 핵심 기능 사용법을 몇 가지 명령어로 소개한다:

5.1 컨테이너 목록 확인하기 (`docker ps`)

- 실행 중인 컨테이너 목록을 확인하는 명령어이다.
- 사용법: `docker ps [옵션]`

주요 옵션:
- `-a, --all=false` : 중지된 컨테이너까지 **모든 컨테이너**를 출력
- `--before=""` : 지정한 컨테이너가 생성되기 **이전**에 생성된 컨테이너들을 출력 (중지된 컨테이너 포함)
- `-f, --filter=[]` : 출력 결과를 필터링 (예: `"exited=0"` - 정상 종료된 컨테이너만 표시)
- `-l, --latest=false` : 마지막으로 생성된 컨테이너 하나를 출력 (중지된 컨테이너 포함)
- `-n=-1` : 최근 생성된 컨테이너 N개만 출력 (중지된 컨테이너 포함)
- `--no-trunc=false` : 출력 시 길이가 긴 항목도 **생략 없이 모두 표시**
- `-q, --quiet=false` : 컨테이너 ID만 출력 (기타 정보 생략)

5.2 컨테이너 중지하기 (`docker stop`)

- 실행 중인 컨테이너를 중지하는 명령어이다.
- 사용법: `docker stop [옵션] CONTAINER [CONTAINER...]`

주요 옵션:
- `-t, --time=10` : 컨테이너를 중지하기 전에 **대기할 시간(초)**를 설정 (기본 10초 후 SIGKILL)

5.3 컨테이너 제거하기 (`docker rm`)

- 컨테이너를 제거(삭제)하는 명령어이다. (중지된 컨테이너만 삭제 가능)
- 사용법: `docker rm [옵션] <컨테이너 이름 또는 ID>`

주요 옵션:
- `-f, --force=false` : 컨테이너를 강제로 **정지시키고 삭제** (SIGKILL 신호를 사용하여 실행 중 컨테이너도 삭제)
- `-l, --link=false` : `docker run` 명령의 `--link` 옵션으로 연결된 **링크만 삭제**
- `-v, --volumes=false` : 컨테이너에 연결된 **데이터 볼륨도 함께 삭제**

5.4 이미지 목록 확인하기 (`docker images`)

- 도커 이미지 목록을 출력하는 명령어이다.
- 사용법: `docker images [옵션] [이미지 이름]`

주요 옵션:
- `-a, --all=false` : 중간 계층 이미지까지 **모두 표시**
- `-f, --filter=[]` : 출력 결과를 필터링 (예: `"dangling=true"` - 이름이 없는 이미지만 출력)
- `--no-trunc=false` : 생략된 부분 없이 모두 출력
- `-q, --quiet=false` : 이미지 ID만 출력

5.5 이미지 다운로드하기 (`docker pull`)

- Docker 레지스트리(Docker Hub 등)에서 이미지를 다운로드(풀)하는 명령어이다.
- 사용법: `docker pull [옵션] <저장소이름>/<이미지이름>:<태그>`

주요 옵션:
- `-a, --all=false` : 지정한 이미지의 **모든 태그**를 한꺼번에 받음

5.6 이미지 삭제하기 (`docker rmi`)

- 이미지를 삭제하는 명령어이다. 태그를 지정하지 않으면 기본적으로 `latest` 태그가 삭제된다.
- 사용법: `docker rmi [옵션] <이미지이름>`

(이미지가 하나의 태그만 갖고 있으면 이미지 자체가 삭제되고, 여러 태그 중 하나만 삭제한 경우 해당 태그만 제거됨)

6. 활용예제

(Docker를 활용한 실제 사례 및 예제가 가이드에 수록되어 있음. 예: 다중 컨테이너로 웹 애플리케이션 구성, CI 파이프라인에 Docker 적용 등.)

7. FAQ

(Docker 사용 시 자주 묻는 질문과 답변이 정리되어 있음. 예: "이미지와 컨테이너의 차이", "볼륨 정리 방법" 등.)

8. 용어정리

(Docker 및 컨테이너 기술에 관련된 용어들이 정리되어 있음. 예: 이미지(Image), 컨테이너(Container), Docker 데몬(Docker Daemon), Docker 레지스트리(Registry) 등.)
