TI DSP,MCU 및 Xilinux Zynq FPGA

프로그래밍 전문가 과정

이름	문지희
학생 이메일	mjh8127@naver.com
날짜	2018/4/11
수업일수	35 일차
담당강사	Innova Lee(이상훈)
강사 이메일	gcccompil3r@gmail.com

목차

(2) 물리 메모리 관리 자료구조

2-1 Node

: 뱅크(bank) - 접근속도가 같은 메모리의 집합. UMA 구조는 1개의 뱅크가 존재하고 NUMA 는 2개 이상의 복수개의 뱅크가 존재한다.
: 뱅크를 표현하는 구조가 노드(node)이다.
: UMA 는 contig_page_data 를 통해 접근 가능하고, NUMA 는 pglist_data 배열을 통해 접근 가능하다. UMA 든 NUMA 든 하나의 노드는 pg_data_t 구조체를 통해 표현된다.

node_present_pages : 해당 노드에 속해 있는 물리 메모리의 실제 양
node_start_pfn : 해당 물리 메모리가 메모리 맵의 몇 번지에 위치하고 있는지를 나타내기 위한 변수
node_zones : zone 구조체를 담기 위한 배열
nr_zones : zone 의 개수를 담는 변수

: 만약 리눅스가 물리 메모리의 할당 요청을 받게 되면 되도록 할당을 요청한 태스크가 수행되고 있는 CPU 와 가까운 노드에서 메모리를 할당하려 한다.
→ 캐시를 활용한다

2-2 Zone

: 노드에 존재하는 물리 메모리 중 16MB 이하 부분을 특별하게 관리하는데, node 의 일부분을 따로 관리할 수 있도록 자료구조를 만든 것이 Zone 이다.
메모리의 특정한 영역을 분할할 때 사용한다.

0 ~ 16M	ZONE_DMA, ZONE_DMA32
16 ~ 896M	ZONE_NORMAL
896 ~endM	ZONE_HIGHMEM

: ZONE_DMA(Direct Memory Access) - CPU 를 통하지 않고 메모리에 접근. 이를 통해 데이터 양이 방대한 비디오 데이터, 음성정보, 네트워크 정보들을 활용하는 영역이 DMA zone 영역이다.
: ZONE_NOMAL - 16MB 이상의 메모리 영역
: ZONE_HIGHMEM - 가상의 주소공간과 물리메모리를 1:1로 매칭하게 되면 공간낭비가 심해서 이를 보완하기 위해 사용하는 구조. 물리 메모리가 1GB 이상일 때 896MB 까지를 커널의 가상 주소공간과 1:1로 연결하고 나머지 부분은 필요할 때 동적으로 연결하여 사용하는 구조이다.
페이징 할 때 2^{10}씩(10bit 씩) 2개 해서 2048개로 4GB 의 가상 주소를 다 표현했다. 이처럼 간접 참조를 하여 ZONE_HIGHMEM 을 구현한다.

: 모든 시스템에서 언제나 DMA, NORMAL, HIGHMEME 의 zone 이 존재하지 않는다. ARM CPU 에서는 node 한 개(UMA 1개) , zone 한 개(NOMAL 1개)가 존재한다.

: zone 구조체에는 은 버디할당자가 사용 할 free_area 구조체를 담는 변수 등이 존재한다. watermark 와 vm_stat 를 통해 남아있는 빈 공간이 부족한 경우 적절한 메모리 해제 정책을 결정한다. 프로세스가 zone 에 메모리 할당 요청을 하였으나 free 페이지가 부족하여 할당하지 못한 경우 이러한 프로세스들을 wait queue 에 넣고 이를 해싱(hashing)하여 wait_table 변수가 가리키게 한다. 페이지를 할당하면 run queue 에 올려준다.

2-3 Page frame

: 물리 메모리의 최소 단위
: page 라는 구조체에서 관리. 리눅스 시스템 내의 모든 물리 메모리에 접근 할 수 있도록 한다.
: 시스템이 부팅되는 순간에 구축되어 물리 메모리 특정 위치에 저장된다. node_mem_map 이라는 전역 배열을 통해 접근할 수 있다. 이는 시스템이 부팅되는 순간에 구축되어 물리 메모리 특정 위치에 저장된다. page 구조체는 4KB 보다 작아야 가상메모리를 사용할 수 있다.

(3) Buddy 와 Slab

: 리눅스는 물리 메모리의 최소 단위인 페이지 프레임 단위(4KB)로 메모리를 할당해 준다.
: 사용자가 4KB 보다 작은 메모리 크기를 요청할 때 내부 단편화 문제가 생기는데 이를 슬랩 할당자(Slab Fragmentation)를 도입하고, 4KB 보다 큰 메모리 크기를 요청할 때 외부 단편화를 줄일 수 있도록 16KB 를 할당해 주는 버디 할당자(Buddy Allocator)를 사용한다.

3-1 버디할당자(Buddy Allocator)

: 버디 할당자는 zone 구조체에 존재하는 free_area 배열을 통해 구축된다. zone 당 하나씩 버디가 존재한다.

: free_area 는 10개의 엔트리를 가진다. 0~9까지의 숫자는 free_area 가 관리하는 할당의 크기를 나타낸다. 4MB($2^{0\sim9}$ * 4KB 의 크기를 가진다.
: nr_free 라는 free 한 페이지 개수를 나타내는 변수와 list_head 라는 연결리스트가 존재한다. map 이라는 변수의 비트맵에 페이지 프레임 단위로 봤을 때의 상태를 저장한다.

free_area[order].map

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	pages
0		0		1		0		0		1		0		0		order(0) = 4K * 2^0
0				0				0				0				order(1) = 4K * 2^1
0								1								order(2) = 4K * 2^2
0																order(3) = 4K * 2^3

양 쪽 둘다 alloc, free 일 때 0, 한쪽은 alloc 이고 한쪽은 free 일 때는 1로 적용한다.
order(0)의 한 페이지 프레임 단위로 메모리를 관리하고 order(1)은 두 페이지 프레임 단위로 메모리를 관리한다. 다른 order 들도 마찬가지다.
order(2)에서 8~15사이의 물리메모리에 16KB 만큼의 메모리를 할당할 수 있어 1이 된다. 다른 것 들도 마찬가지이다.
: free_area.free_list 라는 것이 존재하는데 이는 빈 공간을 나타내는 리스트이다.

…	
3	
2	12
1	
0	5, 10

만약 위의 물리메모리에 2page 에 해당하는 메모리를 요청이 오게 된다면 free_list 에 2page 크기의 메모리가 존재하지 않으니 상위 order 에서 페이지 프레임을 할당 받아 두 부분으로 나뉘어 한 부분은 할당하고 나머지 한 부분은 하위 order 에서 가용 페이지 프레임으로 관리한다. 이를 **버디 할당자**라 한다.

3-2 Lazy Buddy

: 한 프레임을 할당/해제를 반복하는 경우에 페이지 프레임을 할당해주기 위해서 큰 페이지를 쪼개 할당해줘야 한다. 이럴 경우에는 slab 이 개입하여 시간이 오래 걸리게 되는데 여기서 해제하게 된다면 다시 큰 페이지로 합쳐지게 되고 이를 반복하면 많은 오버헤드가 동반되어 성능이 떨어지게 된다. 따라서 할당되었던 페이지 프레임을 합치지 않고 작업을 뒤로 미루는 것이 Lazy 버디이다.

ex)index[0]

: zone 구조체에 존재하는 메모리가 어느정도 차 있는지 확인할 수 있는 watermark(high, low, min)값과 현재 사용가능한 페이지 수를 비교해 가용 메모리가 충분한 경우 해제된 페이지의 병합 작업을 최대한 미룬다.

3-3 슬랩할당자(Slab Allocator)

: 페이지 프레임 크기가 클수록 내부 단편화로 인한 낭비되는 공간이 큰데, 버디를 이용하여 공간을 쪼개게 되면 공간을 쪼개는 작업이 속도를 느리게 하는 문제점이 있다.

: 위의 문제를 해결하기 위한 것이 슬랩할당자이다. 페이지 프레임의 크기가 4KB 일 때, 64byte 크기의 메모리를 공간을 요청하면 미리 4KB 페이지 프레임을 할당 받은 뒤 이 공간을 64byte 분할해두어 이 공간을 떼어주도록 한다. 일종의 캐시로 사용하는 것인데 이런 메모리 관리하는 정책을 슬랩 할당자라 부른다.

: 자주 할당하고 해제되는 크기의 cache 를 가지고 있어야 내부 단편화를 최소화 시킬 수 있다. 2의 승수크기의 cache 를 128KB 유지한다. 다양한 크기의 캐시를 관리하는 kmem_cache 라는 자료구조가 정의되어 있다.

: 슬랩 할당자는 위의 함수 외에도 외부 인터페이스 함수로 kmalloc() / kfree()를 제공한다. kmalloc()함수를 이용ㅇ해 한번에 할당 받을 수 있는 최대 크기는 128K 혹은 4MB 이며 할당된 공간은 물리적으로 연속이라는 특징을 가진다.

(4) 가상 메모리 관리 기법

(5) 가상 메모리와 물리 메모리의 연결 및 변환

(4) 랑 (5)는 내일하겠습니다…..ㅜㅜㅜㅜㅜ

Chapter 5 파일 시스템과 가상 파일 시스템

(1) 파일 시스템 일반

메모리 관리 기법과 파일 시스템의 차이점
: 파일 시스템 관리 기법은 메모리 관리 기법과 동일한데 여기서 '이름'이라는 특성이 있다. 사용자가 파일을 생성하고 파일을 찾게 될 때 디스크 블록 번호로 찾기에는 무리가 있어 이름이라는 특성으로 사용자가 편하게 찾을 수 있도록 한다.

하드디스크에 저장하는 내용

1. 메타 데이터 : inode, 슈퍼블록
2. 사용자 데이터 : 실제 기입하는 내용

(2) 디스크 구조와 블록 관리 기법

디스크의 구조
원판(plotter), 팔(arm), 헤드(head)로 구성되어있다. 원판에는 원모양의 트랙(track)들이 존재하며, 모든 원판에서 같은 위치를 갖는 트랙들의 집합을 실린더(cylinder)라 한다. 트랙은 다시 몇 개의 섹터(sector)로 구분된다.
헤드는 각 원판의 읽기 쓰기가 가능한 면 마다 하나씩 존재.
섹터의 크기는 일반적으로 512byte 여서 디스크 블록이 4KB 이면 하나의 디스크 블록에 8개의 섹터가 대응하게 된다.

탐색 시간
:헤드를 요청한 데이터가 존재하는 트랙 위치까지 이동하는데 걸리는 시간.

회전 지연시간
: 요청한 섹터가 헤드 아래로 위치될 때까지 디스크 원판을 회전시키는데 걸리는 시간
전송 시간
: 헤드가 섹터의 내용을 읽거나 기록하는데 걸리는 시간.

탐색시간이 가장 느리고 전송시간이 제일 빠르다.

디스크의 논리적인 구조

: 파일 시스템은 블록단위로 나눠서 관리하려한다. 메모리에 올려야하므로 단편화가 현상이 나타날 수 있으니 블록단위로 나눈다. 그래서 페이지 프레임의크기(4KB)에 따라 디스크 블록의 크기도 4KB 로 따른다.

: 시스템 성능의 병목 요소는 디스크 I/O 여서 최근에는 디스크 블록의 크기를 더 크게 (4K * 2의 n 승)설정하는 경향이 있다.

:파일 시스템이 디스크 블록을 읽도록 요청하면 디스크 블록의 번호를 이에 대응되는 섹터들로 매핑 시키는 일을 디바이스 드라이버 또는 디스크 컨트롤러가 담당한다.

디스크 블록 할당방법

1. 연속할당

탐색 시간이 오래 걸리지 않아 속도가 빠르다.

2. 불연속 할당

불연속으로 할당되면 탐색 시간이 오래 걸리게 되어 속도면에서는 좋지 않지만, 공간 활용이 좋아 용량 면에서는 좋다.

ex)디스크모으기

:파일에 새로운 내용을 추가 할 때 연속 할당을 하지 못하면 불연속 할당을 하게 되고, 파일 크기가 커질 때 기존에 있던 블록들을 다른 곳으로 복사하는 것은 성능 상 매우 큰 문제이므로 free 한 디스크 블록에 할당한다.

불연속 할당 방법

1.블록체인 기법

: 같은 파일에 속한 디스크 블록들을 체인으로 연결해 놓는 방법. Filename 과 start, size 정보를 저장한다.

첫번째 디스크 블록에 가면 포인터를 이용해 다음 블록의 위치를 찾아 갈 수 있다.

lseek()같은 시스템 콜을 사용할 때 파일의 끝 부분을 읽으려는 경우에 앞 부분의 데이터 블록을 읽어야 하고 중간 블록이 유실되면 나머지 데이터까지 모두 잃게되는 단점이 있다.

2.인덱스 블록 기법

: 블록들에 대한 위치 정보들을 기록한 인덱스 블록을 따로 사용하는 방법. File name 과 index, size 를 index block 에 저장하여 각 인덱스들을 디스크 블록을 가리키고 있다.

lseek()를 사용하여 파일의 끝 부분을 접근할 때 데이터 블록을 일일이 읽을 필요가 없지만 인덱스 블록이 유실되면 데이터 전체가 소실되는 단점이 있다.

3.FAT 기법

: 같은 파일에 속해 있는 블록들의 위치를 FAT 라는 자료구조에 기록해 놓는 방법이다. 블록체인 기법과 인덱스 블록 기법의 개념을 합친 기법으로, 블록체인의 형식인 file name, start, size 인덱스가 존재하고 인덱스 블록 기법에서의 index block 처럼 FAT 가 존재한다.

FAT 구조에서는 FF 는 파일의 끝을 의미하고 0은 free 상태를 의미한다.
start 인덱스가 있어 데이터가 중간에 유실되어도 복원이 가능한 장점이 있지만 FAT 도 케이블 자체가 깨지게 되면 복원하지 못한다.
리눅스에서 사용되는 파일시스템인 ext2나 ext4(ext3에서 바뀜)는 블록 기법과 유사한 기법을 사용한다. 그것이 바로 inode 이다.

(3) FAT 파일 시스템

: 파일 시스템이 관리하는 데이터는 메타데이터와 유저데이터로 구분하는데 메타데이터는 FAT 테이블, 디렉토리 엔트리, 수퍼블록으로 구성된다.
: 파일시스템들은 저마다 자신만의 디렉터리 구조체를 선언해 놓은 뒤, 파일이 생성될 때 이 구조체의 각 내용들을 채워서 디스크에 저장한다. 사용자가 특정 이름을 가지는 파일의 내용을 읽어오도록 파일 시스템에 요청한다면 파일시스템은 사용자가 요청한 파일의 이름을 가지고 디렉터리 엔트리를 찾아 낸 뒤 디렉터리 엔트리가 가지고 있는 정보를 통해 데이터 블록을 찾아 사용자에게 제공한다. 다른 파일 시스템 디렉터리 엔트리도 마찬가지이다.
: 리눅스 파일 시스템은 매우 많아서 함수 포인터를 사용하여 FAT 일 때, ext 일 때 각각 옵션별로 지정한다.

(FAT 라는 가정하에)FAT 파일 시스템인 msdos 파일 시스템에서 디렉터리 엔트리의 구조를 보여준다.

```
struct msdos_dir_entry {
        __u8    name[MSDOS_NAME];/* name and extension */
        __u8    attr;           /* attribute bits */
        __u8    lcase;          /* Case for base and extension */
        __u8    ctime_cs;       /* Creation time, centiseconds (0-199) */
        __le16  ctime;          /* Creation time */
        __le16  cdate;          /* Creation date */
        __le16  adate;          /* Last access date */
        __le16  starthi;        /* High 16 bits of cluster in FAT32 */
        __le16  time,date,start;/* time, date and first cluster */
        __le32  size;           /* file size (in bytes) */
};
```

파일시스템이 디렉터리 엔트리를 찾는 방법
: 사용자는 파일 이름을 통해 데이터에 접근하려 한다. 파일의 이름은 현재 디렉터리 위치를 기준으로 시작되는 상대 경로와 '/(root)'에서 부터 시작되는 절대 경로 두 가지로 나뉜다.
- 현재 디렉터리 위치 기준
사용자 태스크의 현재 작업 디렉토리(CWD : Current Working Directory)는 항상 task_struct 내에 저장되어 있어 언제든 절대 경로로 변환 가능하다.
- 절대 경로 기준
1.'/'의 디렉터리 엔트리를 읽기. '/'의 데이터 블록을 찾을 수 있다.
슈퍼 블록(super block)이 최상위 디렉터리가 위치하고 있는 곳 같은 정보를 적어 자신이 관리하는 공간의 맨 앞부분에 적어둔다.
2. 슈퍼블록의 첫 번째 데이터 블록을 찾아, 블록 내에서 다음 파일이나 디렉토리 이름을 찾아 해당 파일의 데이터 블록 번호를 확인한다.

3. 확인한 데이터 블록번호를 통해 접근하여 다음 파일이나 디렉토리 이름을 찾아 해당 파일을 찾을 때 까지 반복한다.

(4) inode 구조

: 리눅스의 디폴트 파일 시스템인 ext4, ext 계열 파일 시스템이 채택하고 있는 구조이다.
: inode 를 접근하면 파일의 속성정보와 파일에 속한 디스크 블록들의 위치를 파악할 수 있다.

```
struct inode {
        umode_t                 i_mode;
        unsigned short          i_opflags;
        kuid_t                  i_uid;
        kgid_t                  i_gid;
        unsigned int            i_flags;

#ifdef CONFIG_FS_POSIX_ACL
        struct posix_acl        *i_acl;
        struct posix_acl        *i_default_acl;
#endif

        const struct inode_operations   *i_op;
        struct super_block      *i_sb;
        struct address_space    *i_mapping;

#ifdef CONFIG_SECURITY
        void                    *i_security;
#endif

        /* Stat data, not accessed from path walking */
        unsigned long           i_ino;
        /*
         * Filesystems may only read i_nlink directly.  They shall use the
         * following functions for modification:
         *
         *    (set|clear|inc|drop)_nlink
         *    inode_(inc|dec)_link_count
         */
        union {
                const unsigned int i_nlink;
                unsigned int __i_nlink;
        };
        dev_t                   i_rdev;
        loff_t                  i_size;
        struct timespec         i_atime;
        struct timespec         i_mtime;
        struct timespec         i_ctime;
        spinlock_t              i_lock; /* i_blocks, i_bytes, maybe i_size */
        unsigned short          i_bytes;
        unsigned int            i_blkbits;
        blkcnt_t                i_blocks;

#ifdef __NEED_I_SIZE_ORDERED
        seqcount_t              i_size_seqcount;
#endif

        /* Misc */
        unsigned long           i_state;
        struct mutex            i_mutex;

        unsigned long           dirtied_when;   /* jiffies of first dirtying */
        unsigned long           dirtied_time_when;

        struct hlist_node       i_hash;
        struct list_head        i_io_list;      /* backing dev IO list */
#ifdef CONFIG_CGROUP_WRITEBACK
        struct bdi_writeback    *i_wb;          /* the associated cgroup wb */

        /* foreign inode detection, see wbc_detach_inode() */
        int                     i_wb_frn_winner;
        u16                     i_wb_frn_avg_time;
        u16                     i_wb_frn_history;
#endif
        struct list_head        i_lru;          /* inode LRU list */
        struct list_head        i_sb_list;
        union {
                struct hlist_head       i_dentry;
                struct rcu_head         i_rcu;
        };
        u64                     i_version;
        atomic_t                i_count;
        atomic_t                i_dio_count;
        atomic_t                i_writecount;
#ifdef CONFIG_IMA
        atomic_t                i_readcount; /* struct files open RO */
#endif
        const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
        struct file_lock_context        *i_flctx;
        struct address_space    i_data;
        struct list_head        i_devices;
        union {
```

i_blocks : 몇 개의 데이터 블록을 가지고 있는지를 나타냄
i_mode : inode 가 관리하는 파일의 속성 및 접근 제어 정보를 유지
i_links_count : 이 inode 를 가리키고 있는 파일 또는 링크 수를 의미
i_uid ,i_gid : 각각 user ID, group ID
i_atime, i_ctime, i_mtime : 각각 접근시간, 생성시간, 수정시간을 의미

i_mode

typed(4bit)	u	g	s	r	w	x	r	w	x	r	w	x

상위 4bit
파일의 유형을 의미. 정규파일(S_IFREG), 디렉터리(S_IFDIR), 문자 장치 파일(S_IFCHR), 블록 장치 파일(S_IFBLK), 링크 파일(S_IFLNK), 파이프(S_IFFIFO), 소켓(S_IFSOCK)

다음 3bit
u - setuid(set user id)
g - setgid(set group id) :
s - sticky 비트 : 태스크가 메모리에서 쫓겨날 때, swap 공간에 유지되도록 할 때, 디렉터리에 대한 접근제어에 사용된다.

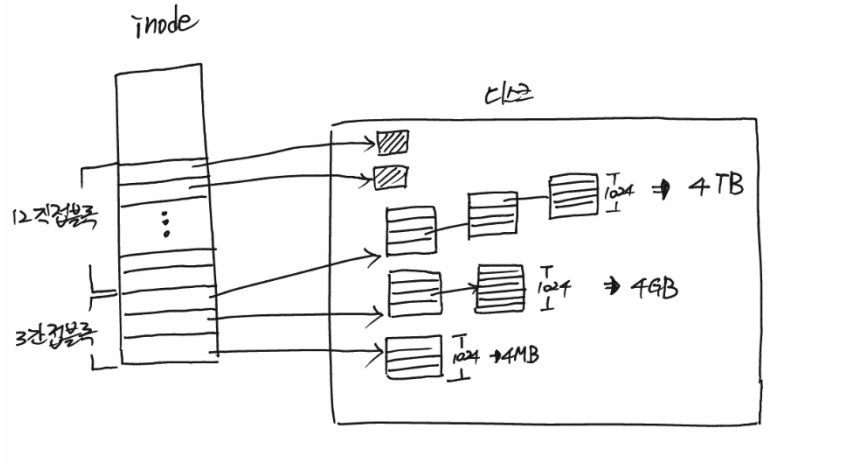
다음 9bit
파일의 접근제어(읽기/쓰기/수행). 처음 3개는 사용자, 다음 3개는 그룹, 마지막 3개는 다른 사용자에 대한 접근제어에 사용

inode 하단부에는 i_block[15]필드 존재
: 파일에 속한 디스크 블록들의 위치를 관리하기 위해 사용
12개는 직접블록(direct block) - 실제 파일의 내용을 담고있는 디스크의 데이터 블록을 가리키는 포인터.
3개는 간접블록(indirect block) - 3개의 간접블록으로 나뉘는데 테이블 형태의 인덱스 블록을 각각 1단계, 2단계, 3단계 인덱스 블록을 가진다.

inode 구조에서 지원할 수 있는 파일의 최대 크기

디스크 블록이 4KB 이므로 인덱스 블록의 크기는 4KB 이고, direct block 이 12개 있으므로 48KB, 3개의 indirect block 이 존재하는데 1단계의 인덱스 블록은 1024개여서 1024 * 4KB = 4MB, 2단계인 인덱스 블록은 1024 * 1024 * 4KB = 4GB, 3단계의 인덱스 블록은 1024 * 1024 * 1024 * 4MB = 4TB 이다.

따라서 총 4TB+4GB+4MB+48KB 가 inode 구조에서 지원할 수 있는 최대 디스크 사이즈이다.



하지만 리눅스가 지원하는 파일의 크기는 4GB 정도 밖에 되지 않는데 커널 내부의 파일과 관련된 함수들이 사용하는 변수나 인자들이 32비트로 구현되었기 때문이다.

ex) task_struct.files_struct.file 의 f_pos 라는 파일옵셋이라는 변수는 파일의 현재 쓰거나 읽을 위치를 나타내는데 32비트 크기를 가진다.

Ext4는 이러한 제한을 해결하여 더 큰 크기의 파일을 지원하고 구글 서버에서 이를 활용한다.

파일 시스템의 디렉터리 엔트리

: 파일 이름과 inode 연결시켜주는 것이 디렉터리 엔트리이다.

```
struct ext2_dir_entry {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __le16  name_len;               /* Name length */
        char    name[];                 /* File name, up to EXT2_NAME_LEN */
};
```

파일의 이름을 담는 name, 파일 시스템이 관리하는 공간의 앞부분에 위치하고 있는 inode 의 테이블에서의 번호인 inode.

(5)Ext2 파일 시스템

: 파일 시스템은 디스크를 관리한다. IDE 방식과 SCSI 방식(스카시방식)이 있는데 IDE 방식은 'hd'라는 이름으로 접근되고 SCSI 는 'sd'라는 이름으로 접근된다.
: df -f 나 mount 로 이를 확인할 수 있다.

```
xeno@xeno-NH:~/kernel/linux-4.4$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            3.8G     0  3.8G   0% /dev
tmpfs           778M  9.3M  769M   2% /run
/dev/sda5       220G   11G  199G   6% /
tmpfs           3.8G   12M  3.8G   1% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
tmpfs           3.8G     0  3.8G   0% /sys/fs/cgroup
none            3.8G  1.9M  3.8G   1% /tmp/guest-y13rtu
tmpfs           778M   76K  778M   1% /run/user/998
tmpfs           778M  112K  778M   1% /run/user/1000
xeno@xeno-NH:~/kernel/linux-4.4$ mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
udev on /dev type devtmpfs (rw,nosuid,relatime,size=3955040k,nr_inodes=988760,mode=755)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000)
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size=796276k,mode=755)
/dev/sda5 on / type ext4 (rw,relatime,errors=remount-ro,data=ordered)
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
tmpfs on /run/lock type tmpfs (rw,nosuid,nodev,noexec,relatime,size=5120k)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,mode=755)
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,release_agent=/lib/system
d/systemd-cgroups-agent,name=systemd)
pstore on /sys/fs/pstore type pstore (rw,nosuid,nodev,noexec,relatime)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,net_cls,net_prio)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpu,cpuacct)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,pids)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
cgroup on /sys/fs/cgroup/rdma type cgroup (rw,nosuid,nodev,noexec,relatime,rdma)
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
systemd-1 on /proc/sys/fs/binfmt_misc type autofs (rw,relatime,fd=27,pgrp=1,timeout=0,minproto=5,maxproto=5,d
irect,pipe_ino=1305)
debugfs on /sys/kernel/debug type debugfs (rw,relatime)
mqueue on /dev/mqueue type mqueue (rw,relatime)
hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime,pagesize=2M)
fusectl on /sys/fs/fuse/connections type fusectl (rw,relatime)
configfs on /sys/kernel/config type configfs (rw,relatime)
none on /tmp/guest-y13rtu type tmpfs (rw,relatime,mode=700,uid=998)
tmpfs on /run/user/998 type tmpfs (rw,nosuid,nodev,relatime,size=796276k,mode=700,uid=998,gid=998)
tmpfs on /run/user/1000 type tmpfs (rw,nosuid,nodev,relatime,size=796276k,mode=700,uid=1000,gid=1000)
gvfsd-fuse on /run/user/1000/gvfs type fuse.gvfsd-fuse (rw,nosuid,nodev,relatime,user_id=1000,group_id=1000)
```

: 디스크가 시스템에 장착되면 디스크는 사용자가 원하는 개수만큼 논리적인 영역(partition)으로 분할될 수 있다. 각 디스크마다 최대 64개까지 파티션 분할이 가능하다. 'mke2fs'같은 명령어를 이용해 파티션에 파일 시스템을 만들 수 있다.(ext2)
: 파티션에 파일 시스템을 만들면 복수개의 블록그룹(Block Group)으로 나뉜다. ext2파일 시스템은 디스크의 성능을 높이기 위해 서로 관련된 inode 와 디스크 블록들을 인접한 실린더에 유지하려고 한다. → 탐색시간이 짧아짐.
: ext2로 파일 시스템을 구축하면 부트스트랩 코드가 존재하는 부트 블록과 여러 개의 블록 그룹들로 구성된다.
: '/' 디렉터리는 inode 번호 2번을 가지고 있다.

(7)가상 파일시스템(Virtual File System)

: 파일시스템 자체는 특정 운영체제에 구애받지 않고 운영체제가 없어도 동작 가능한 독립적인 구조로 설계하고 구현되어야 한다. 대부분 POSIX 시스템 콜을 이용하면 파일 시스템을 이용할 수 있을 것이라 가정하기 때문에 파일을 생성, 쓰기, 읽기 등의 연산을 할 수 있어야한다.

: 파일시스템마다 포맷이 달라 구동 시키는 방법이 다른데 VFS 는 여러 파일 시스템들이 사용하는 모든 포맷들을 구조체 형태로 만들어 함수 포인터로 랩핑(대체)하여 사용한다.

: 처음 디스크를 읽게 될 때 슈퍼블록을 읽어 VFS 에 메타데이터가 채워지면 어떤 파일 시스템을 이용하는지 알 수 있다. inode_operations 와 file_operations 의 함수 포인터들을 해당 파일시스템 전용으로 바꾸어 처리한 뒤 리턴한다.

VFS 의 디렉터리 엔트리 구조

: VFS 가 다양한 파일시스템과 데이터를 주고받기 위해 5개의 객체를 도입하여 사용자 태스크에게 제공할 일관된 인터페이스를 정의.

1.슈퍼블록(super block)

현재 사용중인(마운트 된) 파일시스템 당 주어진다. 각 파일시스템은 자신이 관리하고 있는 파티션에 파일시스템 마다 고유한 정보를 슈퍼 블록에 저장한다.

2.아이노드 객체

특정파일과 관련된 정보를 담기 위한 구조체이다. (좀 더 디렉토리 관점에 가까움). 아이노드 객체를 생성하고 파일 시스템에 특정 파일에 대한 정보를 요청하면 파일시스템은 자신이 관리하고 있는 영역에서 파일의 메타데이터를 읽어 아이노드 객체에 채워준다.

3.파일 객체

태스크가 open 한 파일과 연관 되어있는 정보를 관리한다. 파일 디스크립터가 파일 구조체 포인터 배열의 인덱스에 해당하는데 이를 관리한다.(0은 표준입력, 1은 표준출력, 2는 표준에러, 파일 디스크립터는 3 이상)

두 개의 태스크가 한 개의 파일을 동시에 접근할 때 물리적으로 하나의 파일이기 때문에 VFS 도 하나의 아이노드 객체를 만들어 유지하고 두 태스크가 접근하는 위치(offset) 정보는 태스크마다 다르게 유지하는 용도로 사용된다. - system programming 에서 같은 파일 동시에 읽은 예제

4.디엔트리객체

태스크가 파일에 접근하려면 해당 파일의 아이노드 객체를 자신의 태스크와 연관된 객체인 파일 객체에 연결시켜야 하는데 이를 더 빠르게 연결하기 위한 일종의 캐시 역할을 하는 것이 디엔트리 객체이다.

5.패스(path)

```
struct path {
        struct vfsmount *mnt;
        struct dentry *dentry;
};
```

dentry : 캐시 역할을 하는 것
vfsmount : 현재 가상 파일 시스템이 어떤 파일 시스템을 물었는지

(8)태스크 구조와 VFS 객체

: 이전에 system programming 을 배울 때 같은 파일을 동시에 read 하여 출력하는 예제에 해당하는 내용. 결과는 10 바이트씩 처음부터 읽어 같은 결과가 나타나게 된다.

task_struct	files_struct	file	path	dentry	inode	super_block
task_struct	files_struct	file	path	dentry		

파일을 읽기 위해 접근할 때 다음에 읽을 바이트를 결정하는 변수가 file 내에 f_pos 라는 변수가 file 마다 존재한다. 두개가 분리되어 있으니 파일을 처음부터 읽은 같은 결과 값이 출력되게 된다. 실제로 공유되고 있는 것은 inode 부터 이므로 서로 파일 객체는 공유하지 않는다.

: 파일 객체에서 중요한 변수 f_of, file_operations(open, read, write 함수 포인터 구조체)라는 자료구조를 가리키는 포인터이다.

: 아이노드 객체에서 i_dev, inode 가 실제 존재하고 있는 파일시스템의 위치를 나타냄.

i_rev, 파일이 장치파일인 경우 관련 되어있는 디바이스 드라이버의 주번호를 나타낸다.