System Programming

HW : In-memory / On-disk

2013210111 남세현

1. On-disk 자료구조가 어떻게 In-memory Object로 관리되고 있는가?

On-disk 자료구조에는 다음과 같은 요소가 있다.

- Boot Block
- Super Block
- Inode
- Data Block

이 중에서 Boot Block은 제외하고, Super Block부터 In-Memory로서의 Object 정보를 찾아보자.

File System을 담당하고 있는 /linux/fs.h 헤더파일에 보면 super_block에 대한 정의가 나와있다.

```
1289 struct super_block {
1290         struct list_head        s_list;
1291         dev_t                   s_dev;
1292         unsigned char           s_blocksize_b
1293         unsigned long           s_blocksize;
1294         loff_t                  s_maxbytes;
1295         struct file_system_type *s_type;
1296         const struct super_operations   *s_op
1297         const struct dquot_operations   *dq_o
1298         const struct quotactl_ops       *s_qc
1299         const struct export_operations *s_exp
1300         unsigned long           s_flags;
1301         unsigned long           s_iflags;
1302         unsigned long           s_magic;
1303         struct dentry           *s_root;
1304         struct rw_semaphore     s_umount;
1305         int                     s_count;
1306         atomic_t                s_active;
1307 #ifdef CONFIG_SECURITY
1308         void                    *s_security;
1309 #endif
1310         const struct xattr_handler **s_xattr;
```

약 코드라인이 100줄이여서 이 보고서에 다 담기에는 부적절해, 중요한 멤버들만 간추려 보면 다음과 같다.

- struct list_head s_inodes
- const struct super_operations *s_op

다른 멤버들은 특별한 것이 없었다. Super Block에서 가장 중요한 Inode Table이 s_inodes라는 멤버변수로 저장되고 있음을 확인할 수 있었다. 그 다음으로 가장 특별한 것은 바로 super block에 대한 operation들을 저장해 놓는 s_op이다.

```
1689 struct super_operations {
1690         struct inode *(*alloc_inode)(struct super_block *sb);
1691         void (*destroy_inode)(struct inode *);
1692
1693         void (*dirty_inode) (struct inode *, int flags);
1694         int (*write_inode) (struct inode *, struct writeback_
1695         int (*drop_inode) (struct inode *);
1696         void (*evict_inode) (struct inode *);
1697         void (*put_super) (struct super_block *);
1698         int (*sync_fs)(struct super_block *sb, int wait);
1699         int (*freeze_super) (struct super_block *);
1700         int (*freeze_fs) (struct super_block *);
1701         int (*thaw_super) (struct super_block *);
1702         int (*unfreeze_fs) (struct super_block *);
1703         int (*statfs) (struct dentry *, struct kstatfs *);
1704         int (*remount_fs) (struct super_block *, int *, char
1705         void (*umount_begin) (struct super_block *);
```

struct_operations는 역시 linux/fs.h에 정의되어 있는데, destory_inode, write_inode 같은 super block이 가져야 할 operation들이 정의되어 있다. 가장 특이한 점은 이 함수들이 함수포인터로 만들어져 있다는 것인데, 그 말은 자신만의 함수로 대체를 할 수 있다는 것이다.

자신만의 File Structure를 만들었을 경우 그것에 가장 최적화된 operation을 custom으로 사용할 수 있도록 만들어 놓은 것 같다.

```
584 struct inode {
585         umode_t                 i_mode;
586         unsigned short          i_opflags;
587         kuid_t                  i_uid;
588         kgid_t                  i_gid;
589         unsigned int            i_flags;
590
591 #ifdef CONFIG_FS_POSIX_ACL
592         struct posix_acl        *i_acl;
593         struct posix_acl        *i_default_acl;
594 #endif
595
596         const struct inode_operations   *i_op;
597         struct super_block      *i_sb;
598         struct address_space    *i_mapping;
599
600 #ifdef CONFIG_SECURITY
601         void                    *i_security;
602 #endif
603
604         /* Stat data, not accessed from path wa
605         unsigned long           i_ino;
606         /*
```

struct inode도 역시 /linux/fs.h에 저장되어 있다. 구분하기 위한 i_uid, super_operation과 비슷한 inode_operations *i_op, 자신의 super_block을 가리키는 super_block *i_sb, 그리고 sturct list_head i_lru가 있다. Inode LRU List라고 주석이 되어있다.

on-disk structrue를 in-memory object로 바꾸는 이유는 캐싱을 위해서인데, 그 캐싱의 기법이 LRU여서 그렇게 네이밍을 한게 아닌가 싶다. 그 다음으로는 i_dentry가 있다. 이 내용은 다음 주제와 연결된다.

2. In-Memory에만 존재하는 object들은 어떻게 관리되고 있는가?

```
108 struct dentry {
109         /* RCU lookup touched fields */
110         unsigned int d_flags;           /* protected by d_lock
111         seqcount_t d_seq;               /* per dentry seqlock
112         struct hlist_bl_node d_hash;    /* lookup hash list */
113         struct dentry *d_parent;        /* parent directory */
114         struct qstr d_name;
115         struct inode *d_inode;          /* Where the name belo
116                                          * negative */
117         unsigned char d_iname[DNAME_INLINE_LEN];        /* sma
118
119         /* Ref lookup also touches following */
120         struct lockref d_lockref;       /* per-dentry lock and
121         const struct dentry_operations *d_op;
122         struct super_block *d_sb;       /* The root of the den
123         unsigned long d_time;           /* used by d_revalidat
124         void *d_fsdata;                 /* fs-specific data */
```

i_dentry는 dentry의 리스트인데, dentry는 /linux/dcache.h에 정의되어있다. Dentry는 그 자체의 이름인 struct qstr d_name, 소속되어 있는 inode인 *d_inode, 역시 operation들의 집합인 *d_op도 있다. Dentry의 root인 super block을 가리키는 d_sb, 가장 최근에 수정한 시간을 저장하는 d_time 등이 있다.

그리고 d_lru (LRU list), d_child, d_subdirs가 있다. 디렉토리 엔트리 내부에 있는 파일이나 서브디렉토리에 대한 항목을 저장하고 있는 것으로 파악된다.

하지만 dentry는 위에 나열한 On-disk 자료구조 4개에는 포함되지 않는다. 그와 비슷한 only in-memory object에는 file이 있다.

```
839 struct file {
840         union {
841                 struct llist_node       fu_lli
842                 struct rcu_head         fu_rcu
843         } f_u;
844         struct path             f_path;
845         struct inode            *f_inode;
846         const struct file_operations    *f_op;
847
848         /*
849          * Protects f_ep_links, f_flags.
```

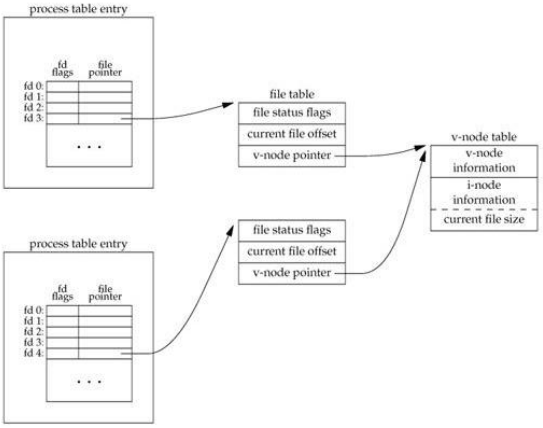
File은 /linux/fs.h에 정의되어 있다. 캐싱해놓은 inode 포인터 *f_inode와 파일의 경로 f_path, operation를 정의해놓은 *f_op가 있다. 추가적인 변수로는 f_owner 등이 있다. 우리가 ls 명령어를 통해 얻을 수 있는 정보들이 저장되어 있는 듯 하다.

File에서 가장 반가운 멤버는 바로 f_pos다. 우리가 흔히 말하는 파일 포인터의

역할을 하는 것으로 보인다. 역시 File의 concurrency를 위해 mutex로 f_pos_lock을 만들어 둠을 확인할 수 있다. 리눅스 개발자들은 바보가 아니었다.

추가적으로, file_operations에는 커널에 기본적으로 정의되어 있는 함수를 사용할 것인지, 아니면 VFS로서 custom된 함수를 사용할 것인지 정의가 다 되어있다.

3. Open된 파일 디스크립터는 어떻게 관리되고 있는가?



1학기때 진행된 운영체제 수업때 언급한 것 처럼, Proc/PID/FD 형태로, 즉 Process당 FD Table이 따로 존재한다. 하지만 그 FD는 바로 파일에 일대일 이 아니라 정확하게는 다대일 대응이 된다.

우리가 파일을 열거나 만들면 위 1, 2에서 언급한 내용들이 메모리에 캐싱되게 된다. 그럼 그 캐싱된 테이블에 대해 Proc/PID/FD에 포인터가 저장되고, 유저 입장에서는 FD로만 그 지점까지 접근이 가능한 것이다.

File의 count 멤버변수를 이용하면 스마트 포인터처럼 현재 reference count를 체크하여 캐싱을 계속 하고 있어야만 하는지를 결정할 수도 있다는 이야기다.

Inode에 대해서는 어차피 File이 inode에 대한 포인터를 가지고 있으므로 크게 문제는 없지만, 확실한 것은 Super Block 단위로 테이블이 만들어져 있으며 새로운 mount에 대해서도 Super Block 1개로써 관리된다고 명시되어 있다.

세번째 주제에 대해서 실습시간에도 정확하게 교수님이 말씀하신 주제가 무엇인지 혼란이 있었는데, 당시 교수님이 숙제를 내기 전에 한 학생이 "FD는 프로세스별로 관리되는지 커널별로 관리되는지"를 질문하였으므로, 그것에 대한 주제라 생각하여 위와 같이 작성하였다.