ISBN 978-89-5884-986-5 98560

기술 분석: Virtual File System

ver.1.0

부　서: 슈퍼컴퓨팅센터
작성자: 조혜영, 차광호, 김성호

한국과학기술정보연구원

Korea Institute of Science and Technology Information

목 차

제 1장 서론 3

1.1 개요 3
1.2 범위 3
1.3 용어 및 약어 4
1.4 참고자료 4

제 2장 중요 자료 구조 5

2.1 SUPERBLOCK 6
2.2 INODE 11
2.3 FILE 16
2.4 DENTRY 19
2.5 FS_STRUCT 21
2.6 FILES_STRUCT 22
2.7 VFSMOUNT 23
2.8 FILE_SYSTEM_TYPE 24

제 3장 중요 자료 구조 사이의 연관 관계 25

3.1 SUPERBLOCK, INODE, FILE_SYSTEM_TYPE 중심 연관 관계 25
3.2 DENTRY 중심 연관 관계 25
3.3 프로세스 관련 파일 연관 관계 26

제 4장 모듈(MODULE) 28

제 5장 리눅스 파일 락킹 30

제 6장 디스크 캐시 33

제 1장 서론

1.1 개요

가상 파일 시스템(VFS, Virtual File System)은 표준 유닉스 파일 시스템과 관련한 모든 시스템 콜을 처리하는 커널 소프트웨어 계층이다. VFS는 실제 파일 시스템과 사용자 프로세스 사이에 존재하는 추상화 계층으로 다른 여러 종류의 파일 시스템에 일반적인 공통 인터페이스를 제공한다. 따라서 VFS는 실제 파일 시스템 설계를 위해서는 반드시 파악해야 하는 소프트웨어 계층이다. 본 문서는 리눅스 가상 파일 시스템의 구조 및 실제 파일 시스템 구현에 필요한 정보를 분석 정리하는 것을 목표로 한다.

1.2 범위

본 문서에는 클러스터 시스템을 위한 파일 시스템 설계 및 개발에 필요한 VFS 구조 및 구현에 관한 분석 내용을 기술하였다. VFS는 그림 1과 같이 supberblock, inode, file, dentry 등의 여러 자료 구조들이 서로 연결되어 유기적으로 연결되어 동작한다. 본 문서에서는 VFS의 구성 자료 구조 중 특히 중요도가 높은 supberblock, inode, file, dentry, file_system_type, vfsmount, fs_struct, file_struct에 대해서 설명한다. 2장에서는 각 자료 구조에 대해 자세히 알아보고, 3장에서는 각 객체들 사이의 연관 관계를 알아본다.

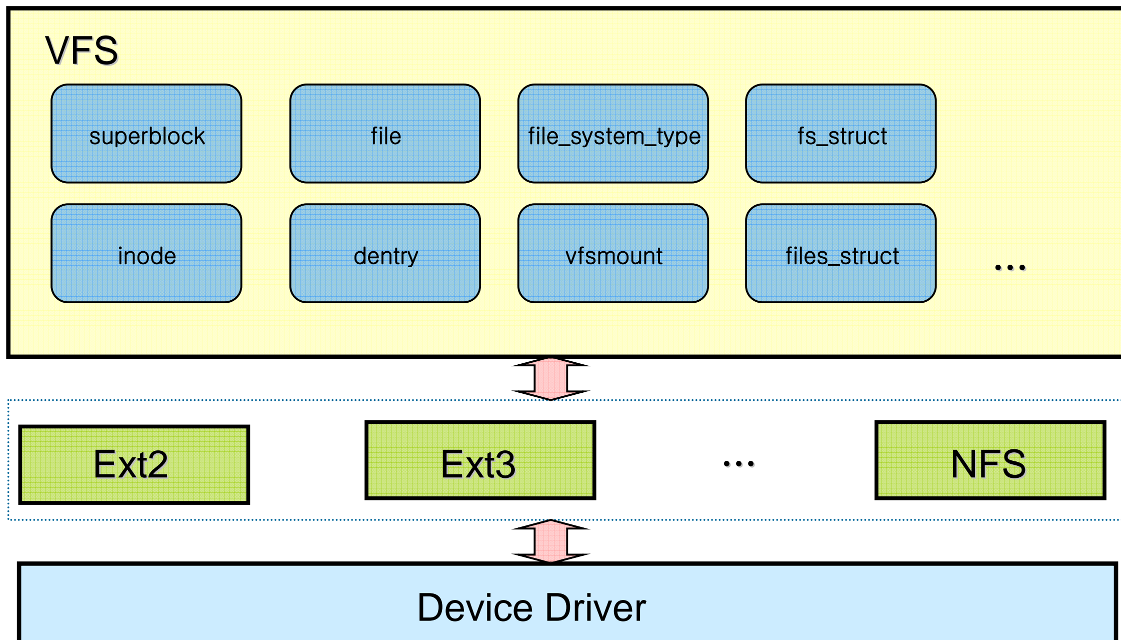

그림 1. VFS 구조

1.3 용어 및 약어

SB: Superblock

2장의 테이블 '비고'란의 표기는 다음의 형식을 따른다.

- 붉은색은 타 자료구조와 관계되는 내용을 표시한다.
- 푸른색은 요구사항정의서와 관련된 내용을 표시한다.
- 녹색은 오퍼레이션에 관련된 내용을 표시한다.
- 주황은 VFS를 통해 동작하는 ext2, ext3와 같은 개별 파일 시스템에 dependent한 부분을 나타낸다.

1.4 참고자료

[1] Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel," 3rd Edition, 2005.

[2] Linux Kernel Source 2.6.15.4

[3] Linux Kernel Source 2.4.32

제 2장 중요 자료 구조

VFS는 실제 파일 시스템의 구현과 사용자 프로세스 사이에 존재하는 추상화 계층으로, supberblock, inode, file, dentry 등과 자료 구조들이 서로 연결되어 유기적으로 연결되어 동작한다. 그림 2와 같이 ext3, gfs, nfs의 3가지 파일 시스템이 마운트되어 사용될 경우를 가정하여, 그림 3에 중요 자료 구조들의 연관 관계를 개략적으로 도시화하였다. 객체들의 연관 관계의 복잡도가 높아서, 더욱 자세한 내용은 아래에 각 자료 구조를 설명한 장에 기술하였다.

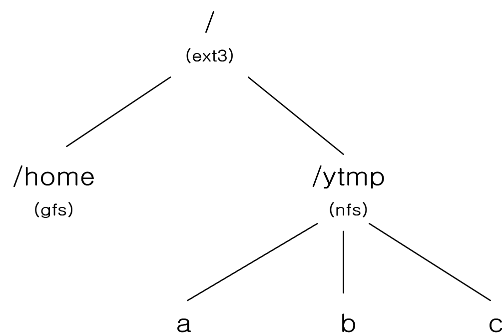

그림 2. mount 된 파일 시스템 상태

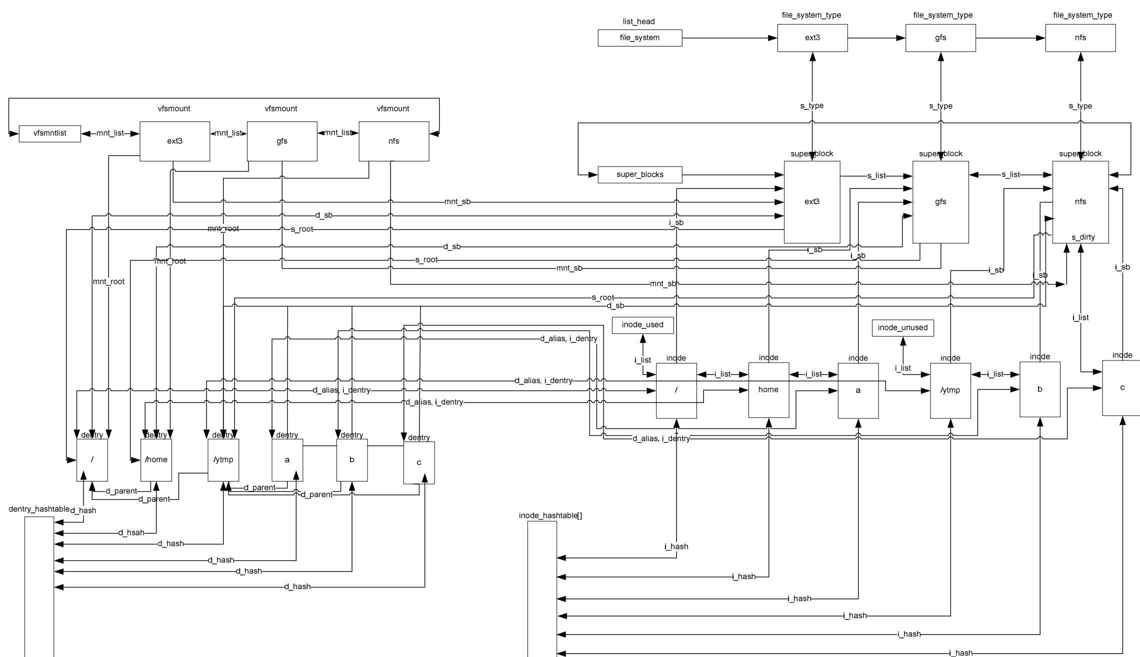

그림 3. 중요 자료 구조 연관 관계

2.1 superblock

□ 개요

슈퍼블록(superblock) object는 마운트 시킨 파일 시스템 정보를 저장하는 object로 보통 disk에 저장된 filesystem의 control block에 해당한다. VFS는 super_blocks라는 이름으로 슈퍼블록 객체를 원형 더블 링크로 관리한다. 슈퍼블록의 자료구조는 include/linux/fs.h에 define되어 있다. 슈퍼블록의 구조를 살펴보면 그림 4와 같이 일반적인 파일 시스템의 특성을 나타내는 부분과 특정 파일시스템에 속한 슈퍼블록 정보를 포함하는 부분을 포함한다. 또한 s_op라는 필드에 슈퍼블록과 관련한 메소드(method)인 슈퍼블록 오퍼레이션의 구조체 포인터를 가지고 있다. 모든 슈퍼블록 객체(마운트된 파일 시스템마다 하나씩)는 그림 5와 같이 원형 이중 연결 리스트(circular doubly linked list)로 연결되어 있다.

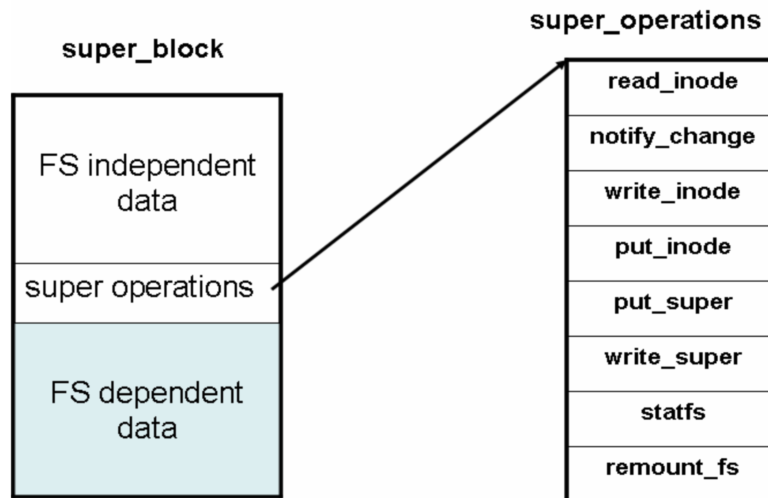

그림 4. 슈퍼블록 객체 내부

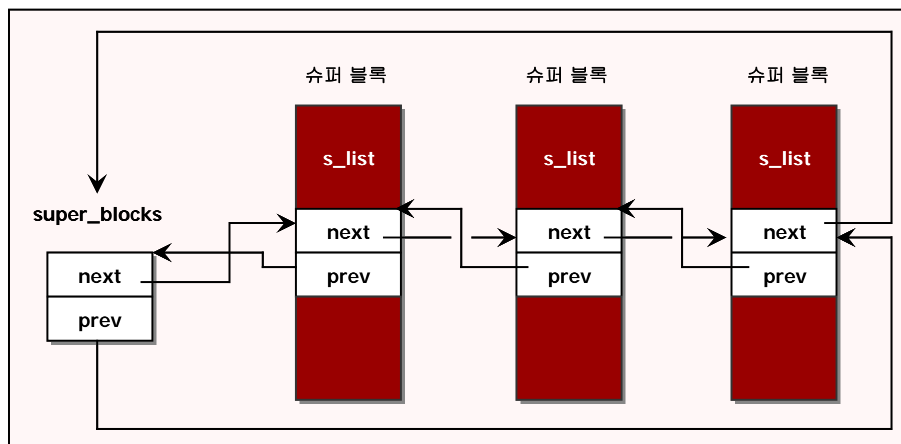

그림 5. 슈퍼블록 객체 연결 구조

□ 세부 자료구조

유형	필드	설명	비고
struct list_head	s_list	슈퍼블록 리스트 포인터	SB 리스트
dev_t	s_dev	장치 식별자	
unsigned long	s_blocksize	블록 크기(바이트 단위)	
unsigned long	s_old_blocksize	블록 크기(바이트 단위) 블록장치 드라이버에 의해서 보고됨	
unsigned long	s_blocksize_bit	블록 크기(비트 단위)	
unsigned char	s_dirt	변경된 플래그	
unsigned char	s_maxbytes	파일의 최대 크기	파일 크기 제한
struct file_system_type *	s_type	파일 시스템 유형	file_system_type
struct super_operations *	s_op	슈퍼 블록에 관련된 오퍼레이션	SB 관련 오퍼레이션
struct dquot_operations *	dq_op	디스크 용량(쿼터) 오퍼레이션	디스크 쿼터 할당 관련 오퍼레이션
struct dqotactl_ops *	s_qcop	디스크 쿼터 관리 오퍼레이션	디스크 쿼터 관리 오퍼레이션
struct export_operations *	s_export_op	네트워크 파일 시스템(nfs) 관련된 export 오퍼레이션	네트워크 파일 시스템 관련 export 오퍼레이션
unsigned long	s_flags	마운트 플래그	
unsigned long	s_magic	파일 시스템 매직 넘버	
struct dentry *	s_root	파일 시스템의 root 디렉토리의 디엔트리 객체	dentry
struct rw_semaphore	s_umount	unmounting을 위해 사용되는 세마포어	
struct semaphore	s_lock	슈퍼블록 세마포어 - 여러 프로세서가 동시에 접근하는 것에 대해서 스핀(spin) 락으로 보호한다.	
int	s_count	참조 카운터	
int	s_syncing	슈퍼블록의 inodes들이 동기화 되었다는 것을 가리키는 플래그	
int	s_need_sync_fs	슈퍼블록에 마운트 된 파일 시스템의 동	

		기화에 사용되는 플래그	
atomic_t	s_active	2차 참조 카운터	
void *	s_security	슈퍼블록의 보안(security) 자료구조 포인터	security structure
struct xattr_handler	**s_xattr	슈퍼블록의 확장된 attribute 자료구조의 포인터	extended attribute structure
struct list_head	s_inodes	모든 inode 리스트	inode 리스트
struct list_head	s_dirty	변경된(dirty) inode 리스트	inode 리스트
struct list_head	s_io	디스크의 write되기를 기다리는 inode 리스트	
struct hlist_head	s_anon	remote network filesytem을 핸들링하기 위한 익명(anonymous) denty들의 리스트	denty 리스트
struct hlist_head	s_files	슈퍼블록에 할당된 파일 객체 리스트	file 리스트
struct block_device *	s_bdev	블록 장치 디바이스 디스크립터의 포인터	
struct list_head	s_instances	주어진 파일시스템 타입의 슈퍼블록 개체의 리스트	superblock VFS 타입의 SB 리스트
struct quota_info	s_dquot	디스크 쿼터 옵션	
int	s_frozen	파일 시스템이 일관성(consistent)을 잃은 상태(freezing 상태)에 사용하는 플래그	
wait_queue_head_t	s_wait_unfrozen	파일 시스템이 freezing상태를 벗으날 때까지 프로세스가 대기(sleep)하는 큐	
char[]	s_id	슈퍼블록을 가지는 블록 장치 이름	
void *	s_fs_info	특정한 파일시스템의 슈퍼블록의 포인터	ext2, ext3, VFS 등의 슈퍼블록 포인터 2.4에서는 union 타입에 generic_spb사용
struct semaphore	s_vfs_rename_sem	디텍토리들 사이에 파일을 renaming할 때 VFS에 의해 사용되는 세마포어	
u32	s_time_gran	타임스탬의 단위 timestamp's granularity (in nonaoseconds)	

☐ 오퍼레이션(s_op)

• struct inode *(*alloc_inode)(struct super_block *sb);

- inode 객체를 위한 스페이스를 할당받는다. 개별 파일 시스템을 위한 스페이스를 포함한다.

• void (*destroy_inode)(struct inode *);

- 개별 파일 시스템을 위한 영역을 포함하여, inode 객체를 destroy한다.

• void (*read_inode) (struct inode *);

- 디스크 정보를 읽어서 inode 객체를 채운다.

• void (*dirty_inode) (struct inode *);

- inode가 편경됨으로 표시될 때 호출된다. Reiser FS, ext3 같은 파일시스템에서 디스크의 파일시스켐 저널을 변경하기 위해 사용된다.

• int (*write_inode) (struct inode *, int);

- parameter로 넘겨준 inode object의 내용을 가지고 filesystem inode를 갱신한다.

- inode object의 i_ino field는 관련된 disk에 있는 inode를 명시한다.

• void (*put_inode) (struct inode *);

- inode 객체 해지, 다른 프로세서가 사용할 수 있으므로 메모리 반환을 의미하지 않음

• void (*drop_inode) (struct inode *);

- inode가 destroy 되었을 때 불림. 일반적으로 generic_drop_inode()함수가 불림.

• void (*delete_inode) (struct inode *);

- 메모리의 VFS inode, 디스크의 inode, 파일 데이터 블록을 삭제한다.

• void (*put_super) (struct super_block *);

- 매개 변수로 넘겨준 주소의 슈퍼블록 객체를 해제한다. (대응하는 파일시스템을 umount 한 경우다.)

• void (*write_super) (struct super_block *);

- 지정한 객체 내용으로 superblock 객체를 갱신한다.

• int (*sync_fs)(struct super_block *sb, int wait);

- 디스크의 filesystem-specific 정보를 업데이트할 때 사용한다. (저널링 파일 시스템에서)사용

• void (*write_super_lockfs) (struct super_block *);

- 파일 시스템에 대한 변경을 차단하고 매개 변수로 넘겨준 값을 사용하여 슈퍼블록을 변경한다. 저널링 파일 시스템이 이 메소드를 구현해야 하며, 논리 볼륨 관리자(LVM, Logical Volume Manager) 드라이버가 호출해야 한다..

• void (*unlockfs) (struct super_block *);

- 위의 write_super_lockfs가 파일시스템 변경을 차단한 것을 해지한다.

• int (*statfs) (struct super_block *, struct kstatfs *);

- 파일 시스템 통계정보를 리턴한다.

• int (*remount_fs) (struct super_block *, int *, char *);
- 새로운 옵션을 가지고 파일시스템을 다시 마운트한다.
• void (*clear_inode) (struct inode *);
- put_inode와 비슷하지만 파일의 자료를 포함한 모든 페이지를 해제한다.
• void (*umount_begin) (struct super_block *);
지정한 inode에 대해 unmount 연산을 시작했으므로 mount 연산을 중단한다. (네트워크 파일시스템에서만 사용)
• int (*show_options)(struct seq_file *, struct vfsmount *);
filesystem-specific 옵션을 보여준다.
• ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
쿼터 시스템(quota system)에서 사용, 쿼터 정도를 읽어옴
* quota system: 각 usr/group 별로 용량 제한을 하는 시스템, 시스템 콜 quotactl() 사용
• ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
쿼터 시스템(quota system)에서 사용, 쿼터 정도를 씀

각 오퍼레이션은 공용으로 사용할 수도 있고, 개별 파일시스템별로 독자적으로 구현할 수도 있다. 다음은 ext2의 super 오퍼레이션 테이블이다(/fs/ext2/super.c). 오퍼레이션이 정의되지 않은 경우(NULL 포인터), VFS는 자신의 범용 함수를 사용한다.

```
static struct super_operations ext2_sops = {
        .alloc_inode    = ext2_alloc_inode,
        .destroy_inode  = ext2_destroy_inode,
        .read_inode     = ext2_read_inode,
        .write_inode    = ext2_write_inode,
        .put_inode      = ext2_put_inode,
        .delete_inode   = ext2_delete_inode,
        .put_super      = ext2_put_super,
        .write_super    = ext2_write_super,
        .statfs         = ext2_statfs,
        .remount_fs     = ext2_remount,
        .clear_inode    = ext2_clear_inode,
        .show_options   = ext2_show_options,
 #ifdef CONFIG_QUOTA
        .quota_read     = ext2_quota_read,
        .quota_write    = ext2_quota_write,
 #endif
```

```
};
struct ext2_sb_info {
        unsigned long s_frag_size;      /* Size of a fragment in bytes */
        unsigned long s_frags_per_block;/* Number of fragments per block */
        unsigned long s_inodes_per_block;/* Number of inodes per block */
        unsigned long s_frags_per_group;/* Number of fragments in a group */
        unsigned long s_blocks_per_group;/* Number of blocks in a group */
        unsigned long s_inodes_per_group;/* Number of inodes in a group */
        unsigned long s_itb_per_group;  /* Number of inode table blocks per group */
        unsigned long s_gdb_count;      /* Number of group descriptor blocks */
        unsigned long s_desc_per_block; /* Number of group descriptors per block */
        unsigned long s_groups_count;   /* Number of groups in the fs */
        struct buffer_head * s_sbh;     /* Buffer containing the super block */
        struct ext2_super_block * s_es; /* Pointer to the super block in the buffer */
        struct buffer_head ** s_group_desc;
        unsigned long  s_mount_opt;
        uid_t s_resuid;
        gid_t s_resgid;
        unsigned short s_mount_state;
        unsigned short s_pad;
        int s_addr_per_block_bits;
        int s_desc_per_block_bits;
        int s_inode_size;
        int s_first_ino;
        spinlock_t s_next_gen_lock;
        u32 s_next_generation;
        unsigned long s_dir_count;
        u8 *s_debts;
        struct percpu_counter s_freeblocks_counter;
        struct percpu_counter s_freeinodes_counter;
        struct percpu_counter s_dirs_counter;
        struct blockgroup_lock s_blockgroup_lock;
};
```

at include/linux/ext2_fs_sb.h

2.2 inode

□ 개요

파일시스템이 파일을 다루는데 필요한 모든 정보는 inode라는 자료구조에 들어 있다. inode는 유일하며 파일이 존재하는 동안 남아 있다. 메모리에 내에 있는 inode 객체는 inode구조체로 구성되며, unused inode, in-use inode, dirty inode의 3가지 원형 이중 링크 리스트(circular doubly linked list)로 관리된다. inode객체는 항상 위의 3가지 리스트 중 하나에 속한다.

□ 세부 자료구조

유형	필드	설명	비고
struct list_head	i_hash	해시 리스트 포인터	inode_hashtable[]과 연결
struct list_head	i_list	아이노드 리스트 포인터	inode_unused, inode_in_use, sb->s_dirty 3개 리스트로 관리됨
struct list_head	i_sb_list	슈퍼블록의 아이노드 리스트 포인터	sb->s_list
struct list_head	i_dentry	dentry 리스트 포인터	dentry
unsigned long	i_ino	inode 번호	
atomic_t	i_count	사용 카운터	
umode_t	i_mode	파일 유형과 접근 권한	
unsigned int	i_nlink	하드 링크 수	
uid_t	i_uid	소유자 식별자	
gid_t	i_gid	그룹 식별자	
dev_t	i_rdev	실제 장치 식별자	
loff_t	i_size	파일 길이(바이트 단위)	
struct timespec	i_atime	최종 파일 접근 시간	
struct timespec	i_mtime	최종 파일 기록 시간	
struct timespec	i_ctime	최종 inode 변경 시간	
unsigned int	i_blkbits	블록 크기(비트 단위)	
unsigned long	i_blksize	블록 크기(바이트 단위)	
unsigned long	i_version	버전 번호, 사용 후 자동으로 증가 디렉토리 파일을 일관성있게 유지하기 위해 f_version필드와 함께 사용	

unsigned long	i_blocks	inode 세마포어	
unsigned short	i_bytes	파일의 마지막 블록의 바이트 수	
spinlock_t	i_lock	inode의 어떤 필드를 보호하기 위한 스핀 락	
struct semaphore	i_sem	inode 세마포어	
struct rw_semaphore	i_alloc_sem	direct I/O 파일 오퍼레이션에서 경쟁 조건(race conditions)을 보호하기 위한 read/write 세마포어	
struct inode_operation *	i_op	아이노드 오퍼레이션	아이노드 오퍼레이션
struct file_operations *	i_fop	기본 파일 오퍼레이션	기본 파일 오퍼레이션
struct super_block *	i_sb	슈퍼블록 객체 포인터	SB
struct file_lock *	i_flock	파일 락 리스트 포인터	file_lock
struct address_space *	i_mapping	address_space 객체 포인터	address_space 포인터
struct address_space	i_data	파일에 대한 address_space 객체	address_space (15장)
struct dquot *[]	i_dqout	inode 디스크 쿼터	inode 디스크 쿼터
struct list_head	i_devices	블록 장치 파일 inode 리스트 포인터	
struct pipe_inode_info *	i_pipe	파일이 파이프일 때 사용	
struct block_device *	i_bdev	블록 장치 드라이버 포인터	
struct cdev *	i_cdev	문자 장치 드라이버 포인터	
int	i_cindex	minor 수 그룹안에서 디바이스 파일 인덱스	
__u32	i_generation	inode 버전 번호(일부 파일 시스템에서 사용_	inode 버전 번호
unsigned long	i_dnotify_mask	디렉토리 알립 이벤트의 비트 마스크	
struct dnotify_struct *	i_dnotify	디렉토리 알림에 사용됨	
unsigned long	i_state	inode 상태 플래그	
unsigned long	dirtied_when	inode의 dirtying 타임(in ticks)	
unsigned int	i_flags	파일시스템 마운트 플래그	

atomic_t	i_writecount	쓰기 중인 프로세서 사용 카운터	
void *	i_security	inode security structure 포인터	security structure
void *	u.generic_ip	특정 파일 시스템 정보	KFS inode 정보
seqcount_t	i_size_seqcount	sequence 카운터(SMP 시스템에서 동기화를 위함)	

☐ 오퍼레이션

```
struct inode_operations {
        int (*create) (struct inode *,struct dentry *,int, struct nameidata *);
        struct dentry * (*lookup) (struct inode *,struct dentry *, struct nameidata *);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char *);
        int (*mkdir) (struct inode *,struct dentry *,int);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,int,dev_t);
        int (*rename) (struct inode *, struct dentry *,
                        struct inode *, struct dentry *);
        int (*readlink) (struct dentry *, char __user *,int);
        void * (*follow_link) (struct dentry *, struct nameidata *);
        void (*put_link) (struct dentry *, struct nameidata *, void *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int, struct nameidata *);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
        int (*setxattr) (struct dentry *, const char *,const void *,size_t,int);
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        int (*removexattr) (struct dentry *, const char *);
};
```

아래는 ext2의 inode 오퍼레이션 정의내용이다.(fs/ext2/file.c)

```
struct inode_operations ext2_file_inode_operations = {
        .truncate       = ext2_truncate,
 #ifdef CONFIG_EXT2_FS_XATTR
```

```
        .setxattr       = generic_setxattr,
        .getxattr       = generic_getxattr,
        .listxattr      = ext2_listxattr,
        .removexattr    = generic_removexattr,
#endif
        .setattr        = ext2_setattr,
        .permission     = ext2_permission,
};
```

```
struct ext2_inode_info {
        __le32  i_data[15];
        __u32   i_flags;
        __u32   i_faddr;
        __u8    i_frag_no;
        __u8    i_frag_size;
        __u16   i_state;
        __u32   i_file_acl;
        __u32   i_dir_acl;
        __u32   i_dtime;

        /*
         * i_block_group is the number of the block group which contains
         * this file's inode.  Constant across the lifetime of the inode,
         * it is ued for making block allocation decisions - we try to
         * place a file's data blocks near its inode block, and new inodes
         * near to their parent directory's inode.
         */
        __u32   i_block_group;

        /*
         * i_next_alloc_block is the logical (file-relative) number of the
         * most-recently-allocated block in this file.  Yes, it is misnamed.
         * We use this for detecting linearly ascending allocation requests.
         */
        __u32   i_next_alloc_block;
```

```
        /*
         * i_next_alloc_goal is the *physical* companion to i_next_alloc_block.
         * it the the physical block number of the block which was most-recently
         * allocated to this file.  This give us the goal (target) for the next
         * allocation when we detect linearly ascending requests.
         */
        __u32   i_next_alloc_goal;
        __u32   i_prealloc_block;
        __u32   i_prealloc_count;
        __u32   i_dir_start_lookup;
#ifdef CONFIG_EXT2_FS_XATTR
        /*
         * Extended attributes can be read independently of the main file
         * data. Taking i_sem even when reading would cause contention
         * between readers of EAs and writers of regular file data, so
         * instead we synchronize on xattr_sem when reading or changing
         * EAs.
         */
        struct rw_semaphore xattr_sem;
#endif
#ifdef CONFIG_EXT2_FS_POSIX_ACL
        struct posix_acl        *i_acl;
        struct posix_acl        *i_default_acl;
#endif
        rwlock_t i_meta_lock;
        struct inode    vfs_inode;
};
```

2.3 file

□ 개요

process가 opened file과 어떻게 상호작용을 하는지를 나타낸다. 파일이 열릴 때 생성되고, file structure로 구성된다. disk의 image와 관련이 없기 때문에 dirty field가 없다. 몇몇 프로세스들이 같은 파일을 동시에 access할 수 있다. 각 file 객체는 다음의 Circular doubly linked list 중 하나에 포함된다.

• "unused" 파일 객체의 리스트
파일 객체를 위한 메모리 캐시와 superuser를 위한 예약에 작용하는 list
object가 unused이면 f_count는 null, 이 list의 처음은 free_filps 변수에 저장
superuser가 시스템에 있는 동적메모리를 모두 썼다 하더라도 파일을 오픈 하도록 해 준다.
• "in use" 파일 객체 리스트
리스트의 각 element는 프로세스에 의해 적어도 한번은 사용
f_count는 null이 아니다. list의 처음은 inuse_filps 변수에 저장
• "anon_list" 파일 객체 리스트
사용 중이며, 슈퍼 블록에 할당되지 않은 파일 객체 리스트, f_count=1
open만 하고 아직 write하지 않은 상태라 예상됨

□ 세부 자료구조

유형	필드	설명	비고
struct list_head	f_list	일반적인 파일 객체 리스트 포인터	file 객체 리스트
struct dentry *	f_dentry	파일과 관련된 디엔트리 포인터	denty
struct vfsmount *	f_vfsmount	파일을 포함하는 마운트된 파일 시스템	vfsmount
struct file_operations *	f_op	파일 연산 테이블의 포인터 디스크에서 메모리로 inode를 읽을 때 i_fop에 파일 연산 테이블의 포인터를 가지고 있다. 파일을 열 때, 그 주소를 f_op에 저장한다.	파일 관련 오퍼레이션
atomic_t	f_count	파일 객체의 사용 카운트	
unsigned int	f_flags	파일을 열 때 지정된 플래그	
mode_t	f_mode	프로세스 접근 모드	
loff_t	f_pos	현재 파일 오프셋(파일 포인터)	
struct file_ra_state	f_ra	파일 read-ahead 상태	
unsigned long	f_version	버전 (매번 사용 후마다 자동 증가) 디렉토리 파일을 일관성있게 유지하기 위해 i_version필드와 함께 사용	
void *	f_security	파일 객체의 security 객체 포인터	security struct
void *	private_data	파일시스템, 디바이스 드라이버를 위한특별한 데이터 포인터 예)tty 드라이버	
struct list_head	f_ep_links	이 파일을 기다리는 이벤트 poll waiter 리스트의 헤더	

spinlock_t	f_ep_lock	f_ep_links 리스트를 위한 스핀락	
struct address_space	f_mapping	address space 객체의 포인터	address space

□ 오퍼레이션

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long,
unsigned long);
        int (*check_flags)(int);
        int (*dir_notify)(struct file *filp, unsigned long arg);
        int (*flock) (struct file *, int, struct file_lock *);
};
```

파일 시스템의 독자적인 파일 연산은 아래에서 보듯이 VFS 메소드 중 공용으로 사용할 수도 있고, 독자적으로 구현할 수도 있다. 아래는 ext2의 file 오퍼레이션 테이블이다.(fs/ext2/file.c) 대응하는 ext2 메소드가 정의되지 않은 경우(NULL 포인터), VFS는 자신의 범용 함수를 사용한다.)

```
struct file_operations ext2_file_operations = {
        .llseek         = generic_file_llseek,
        .read           = generic_file_read,
        .write          = generic_file_write,
        .aio_read       = generic_file_aio_read,
        .aio_write      = generic_file_aio_write,
        .ioctl          = ext2_ioctl,
        .mmap           = generic_file_mmap,
        .open           = generic_file_open,
        .release        = ext2_release_file,
        .fsync          = ext2_sync_file,
        .readv          = generic_file_readv,
        .writev         = generic_file_writev,
        .sendfile       = generic_file_sendfile,
};
```

2.4 dentry

□ 개요

디렉토리 entry를 메모리로 읽으면, VFS에 의해서 dentry structure의 dentry 객체로 변환된다. 커널은 프로세스가 찾은 경로의 각각의 component로 구성된 dentry 객체 만든다. dentry 객체는 각 구성 요소를 대응하는 inode와 연결된다. 예를 들어, /tmp/test 경로명을 찾을 때 커널은 /디렉토리를 위해 dentry 객체를 생성, 두번째 /디렉토리의 tmp entry를 위한 dentry 객체를 생성, 세번째 /tmp 디렉토리의 test entry를 위한 dentry 객체를 생성한다. Dentry 객체는 디스크에 대응하는 이미지가 없고, dentry 구조체에는 변경된 객체를 나타내는 field가 없다. dentry 객체는 dentry_cache라고 불리는 slab allocator cache에 저장된다. dentry 객체는 kmem_cache_alloc()와 kmem_cache_free()에 의해 생성과 소멸된다. dentry 자료구조는 /include/linux/dcache.h에 정의되어 있다.

dentry 객체는 다음의 4가지 상태 중 하나에 해당한다.

- 해제(Free)

유효한 정보가 없고 VFS에 의해 사용되지 않음

해당 메모리 영역은 슬랩 할당자(slab allocator)에 의해 처리

• 사용하지 않음(Unused)

현재 커널에서 사용되지 않음

d_count는 0이고, d_inode는 여전히 관련된 inode를 가리킴

유효한 정보를 유지하지만, 그 내용은 삭제될 수 있다.

• 사용중 (Inuse)

현재 커널에 의해 사용

d_count 사용 카운터는 양수, d_inode 필드는 연관된 아이노드 객체를 가리킴

유효한 정보를 유지하지만 이를 제거할 수 없다.

• 음수 (Negative)

dentry와 관련된 inode가 더 이상 존재하지 않고 d_inode는 null로 설정

같은 file pathname에 대해 탐색 연산을 빨리 해결하기 위해 여전히 dentry cache에 남아있음

□ 세부 자료구조

유형	필드	설명	비고
atomic_t	d_count	dentry 객체 사용 카운터	
unsigned int	d_flags	dentry 캐시 플래그	
spinlock_t	d_lock	dentry 객체 보호를 위한 스핀락	
struct inode *	d_inode	파일명과 관련 된 inode	inode
struct hlist	d_hash	해시 테이블 엔트리 내 리스트 포인터	dentry_hashtable
struct dentry *	d_parent	부모 디렉토리의 dentry 객체	dentry
struct qstr *	d_name	파일 이름	
struct list_head	d_lru	unused dentry 리스트 포인터	dentry
struct list_head	d_child	부모 디렉토리에 포함된 dentry 객체 리스트의 포인터	dentry
struct list_head	d_subdirs	디렉토리의 경우 서브 디렉토리의 dentry 객체 포인터	dentry
struct list_head	d_alias	같은 inode와 관계 있는 아이노드(앨리어스) 리스트	inode
unsigned long	d_time	d_revalidated 매소드에서 사용됨	
struct dentry_operations *	d_op	dentry 오퍼레이션	dentry 오퍼레이션
struct super_block *	d_sb	파일의 슈퍼블록 객체	superblock

void *	d_fsdata	파일시스템 dependant 정보	파일 시스템에 따른 데이터 (for KFS)
struct rcu_head	d_rcu	RCU(Read-Copy Update) 디스크립터 dentry *RCU: 여러 CPU가 동시게 read를 할 때 자료 구조 보호하기 위한 동기화 기법	
struct dcookie_struct *	d_cookie	kernel profile에서 사용하는 자료구조 포인터	dcookie_struct
int	d_mounted	이 dentry에 마운드된 파일 시스템 수를 위한 카운터 denty가 파일시스템을 위한 마운트 포인트일 때만 1로 설정되는 플래그	
unsigned char[]	d_iname	짧은 파일명 공간	

□ 오퍼레이션

```
struct dentry_operations {
        int (*d_revalidate)(struct dentry *, struct nameidata *);
        int (*d_hash) (struct dentry *, struct qstr *);
        int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
        int (*d_delete)(struct dentry *);
        void (*d_release)(struct dentry *);
        void (*d_iput)(struct dentry *, struct inode *);
};
```

2.5 fs_struct

각 프로세서는 자신의 현재 작업 디렉토리와 루트 디렉토리를 가진다. 이렇게 프로세스와 파일시스템의 상호 관계를 표현하기 위해 fs_struct 구조체를 사용한다. 각 프로세스 디스크립터에는 프로세스의 fs_struct 구조체를 가리키는 fs 필드가 있다. 이 객체는 include/linux/fs_struct.h에 정의되어 있다.

유형	필드	설명	비고
atomic_t	count	이 구조체를 공유하는 프로세스 수	
rwlock_t	lock	구조체의 필드를 위한 읽기/쓰기 스핀락	
int	umask	파일을 열 때 접근 권한 설정을 위한 비	

		트 마스크	
struct dentry *	root	root 디렉토리의 dentry	dentry
struct dentry *	pwd	현재 작업 디렉토리의 dentry	dentry
struct dentry *	altroot	에뮬레이션하는 root 디렉토리의 dentry(x86 아키텍처에서는 NULL)	dentry
struct vfsmount *	rootmnt	root 디렉토리에 마운트된 파일시스템 객체	vfsmount
struct vfsmount *	pwdmnt	현재 작업 디렉토리에 마운트된 파일시스템 객체	vfsmount
struct vfsmount *	altrootmnt	에뮬레이션하는 root 디렉토리에 마운트된 파일시스템 객체(x86 아키텍처에서는 NULL)	vfsmount

2.6 files_struct

프로세스 디스크립터의 files 필드에 files_struct 구조체의 주소를 가지고 있는데, 이 구조체는 프로세스가 연 파일을 나타낸다.

유형	필드	설명	비고
atomic_t	count	이 구조체를 공유하는 프로세스 수	fs
rwlock_t	file_lock	구조체의 필드를 위한 읽기/쓰기 스핀 락	
int	max_fds	허용되는 파일 객체 수의 최대 값 처음 단계 32개이고, 그 이상이면 갱신된다.	허용되는 파일 객체 수의 최대 값
int	max_fdset	허용되는 파일 디스크립터 수의 최대 값	fd의 최대 값
int	next_fd	지금까지 할당된 파일 디스크립터의 최대 값 – 1	
struct file **	fd	파일 객체 포인터 배열의 포인터	
fd_set *	close_on_exec	exec()를 통해 닫을 파일 디스크립터의 포인터	fd
fd_set *	open_fds	열린 파일 디스크립터의 포인터	fd
fd_set	close_on_exec_init	exec()를 통해 닫을 파일 디스크립터의 포인터의 초기 집합	
fd_set	open_fds_init	파일 디스크립터의 초기 집합	
struct file **	fd_array	파일 객체 포인터의 초기 배열	file

* 한 프로세스는 파일 디스크립터를 NR_OPEN(일반적으로 1048576)개 이상 사용할 수 없다. 커널은 프로세스 디스크립터의 rlim[RIMIT_NOFILE] 구조에 파일 디스크립터 수의 최대 값에 대한 동적인 제한을 가지고 있는데, 이 값은 대개 1024이다. 프로세스가 루트 권한을 가지고 있으면 이 값을 늘일 수 있다.

2.7 vfsmount

마운트된 파일 시스템들의 관리를 위해 마운트 포인터, 마운트 플래그, 마운트 될 파일 시스템과 다른 파일 시스템들과의 관계 등을 메모리에 저장하기 위해서 vmfsmount(mounted filesystem descriptor)라는 구조체 사용한다.

유형	필드	설명	비고
struct list_head	mnt_hash	해시 테이블 리스트의 포인터	
struct vfsmount *	mnt_parent	이 파일 시스템이 마운트되는 부모 파일 시스템의 포인터	vfsmount
struct dentry *	mnt_mountpoint	이 파일 시스템의 마운트 디렉토리의 dentry 포인터	dentry
struct dentry *	mnt_root	이 파일 시스템 루트 디렉토리의 dentry 포인터	dentry
struct super_block *	mnt_sb	이 파일 시스템의 슈퍼블록 객체의 포인터	super_block
struct list_head	mnt_mounts	디스크립터의 부모 리스트의 헤드(이 파일 시스템)	
struct list_head	mnt_child	디스크립터의 부모 리스트의 헤드(부모 파일시스템)	
atomic_t	mnt_count	사용 카운트	
int	mnt_flags	플래그	
int	mnt_expiry_mark	expired되면 셋팅되는 플래그(이 플래그가 셋팅되고, 아무도 사용하지 않으면 자동적으로 unmount될 수 있음)	
char *	mnt_devname	장치 파일 이름	
struct list_head	mnt_list	파일시스템 디스크립터의 전역 리스트의 포인터	
struct list_head	mnt_fslink	실제 파일시스템(filesystem-specific)의 expire된 리스트 포인터	

struct namespace *	mnt_namespace	마운트된 파일 시스템의 프로세스 namespace 포인터	

2.8 file_system_type

VFS는 커널 안에 코드가 현재 포함되어 있는 모든 파일 시스템 유형을 관리하며, 등록된 파일 시스템은 file_system_type 객체로 표현된다.

유형	필드	설명	비고
const char *	name	파일 시스템 이름	
int	fs_flags	파일 시스템 유형 플래그	
struct super_block *(*)()	get_sb	슈퍼블록을 읽기 위한 오퍼레이션	
void (*)()	kill_sb	슈퍼블록을 삭제하기 위한 오퍼레이션	
struct module *	owner	파일 시스템을 구현하는 모듈의 포인터	module
struct file_system_type *	next	다음 리스트 요소를 가리키는 포인터	file_system_type
struct list_head	fs_supers	슈퍼블록 객체 리스트의 헤드	list_head(SB)

```
static struct file_system_type ext2_fs_type = {
            .owner              = THIS_MODULE,
            .name               = "ext2",
            .get_sb             = ext2_get_sb,
            .kill_sb            = kill_block_super,
            .fs_flags           = FS_REQUIRES_DEV,
 };
```

제 3장 중요 자료 구조 사이의 연관 관계

본 장에서는 VFS 자료 구조들 사이의 연결 관계를 중요 자료 구조를 중심으로 나타내었다.

3.1 superblock, inode, file_system_type 중심 연관 관계

supberblock, inode, file_system_type은 서로 포인터를 참조하면서 연결되어 있다.

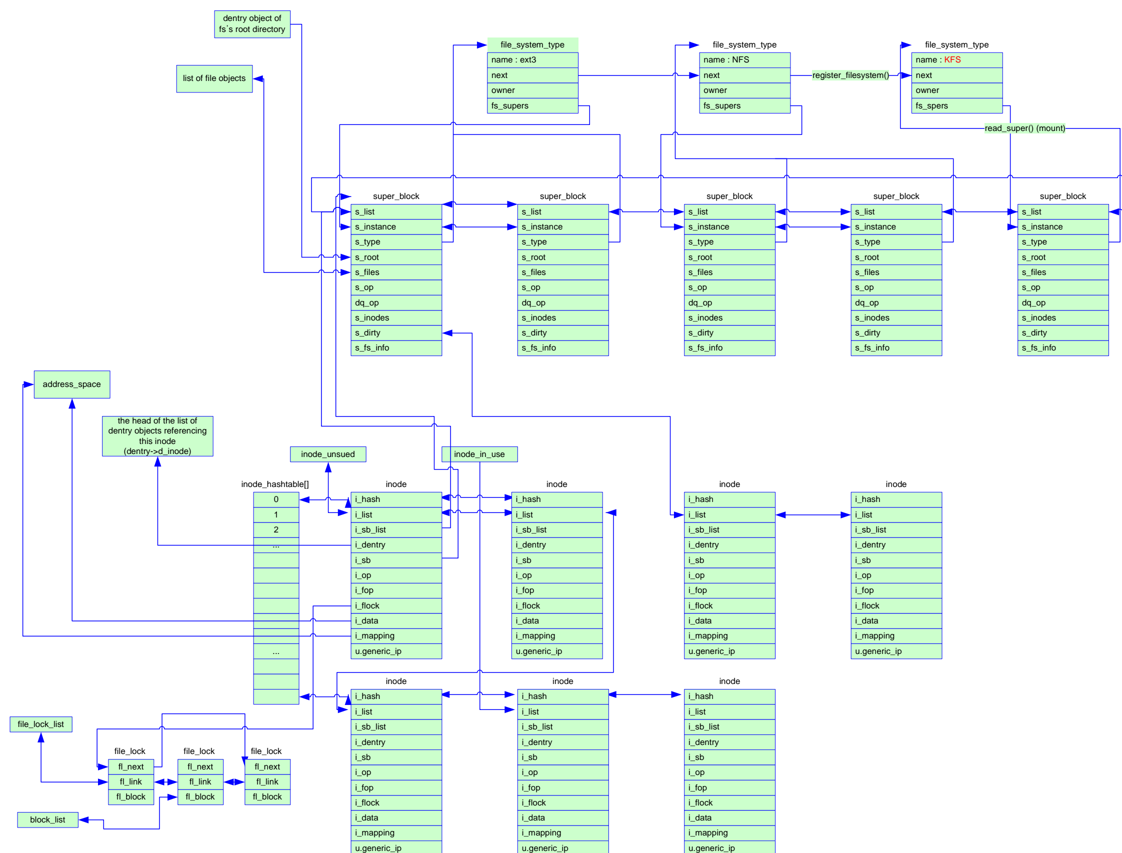

그림 6, superblock, inode, file_system_type 연관 관계

3.2 dentry 중심 연관 관계

dentry 끼리 리스트를 이루고 있으며, dentry구조 안에 inode, superblock을 가리키는 포인터를 가지고 있다.

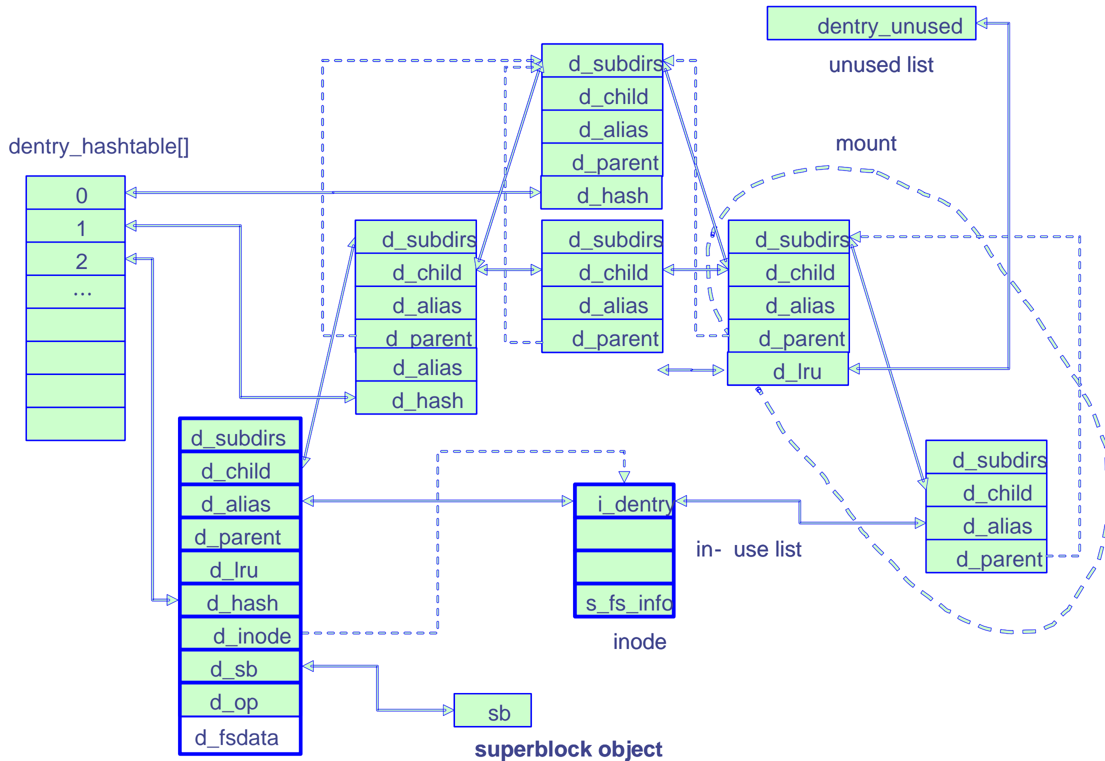

그림 7. dentry 중심 연관 관계

3.3 프로세스 관련 파일 연관 관계

각 프로세스는 자신의 현재 작업 디렉토리와 루트 디렉토리를 가진다. 이처럼 프로세스와 파일 시스템의 상호 관계를 표현하기 위해 커널은 task_struct, fs_struct, files_struct와 같은 자료 구조를 사용한다.

* task_struct: 프로세스 디스크립터

* fs_strcut: 현재 작업 디렉토리, 루트 디렉토리 등 프로세스와 파일 시스템과의 상호 관계

* files_struct: 현재 프로세스가 연 파일에 대한 정보, fd_arrary에 현재 오픈하고 있는 file 구조체의 포인트를 가짐

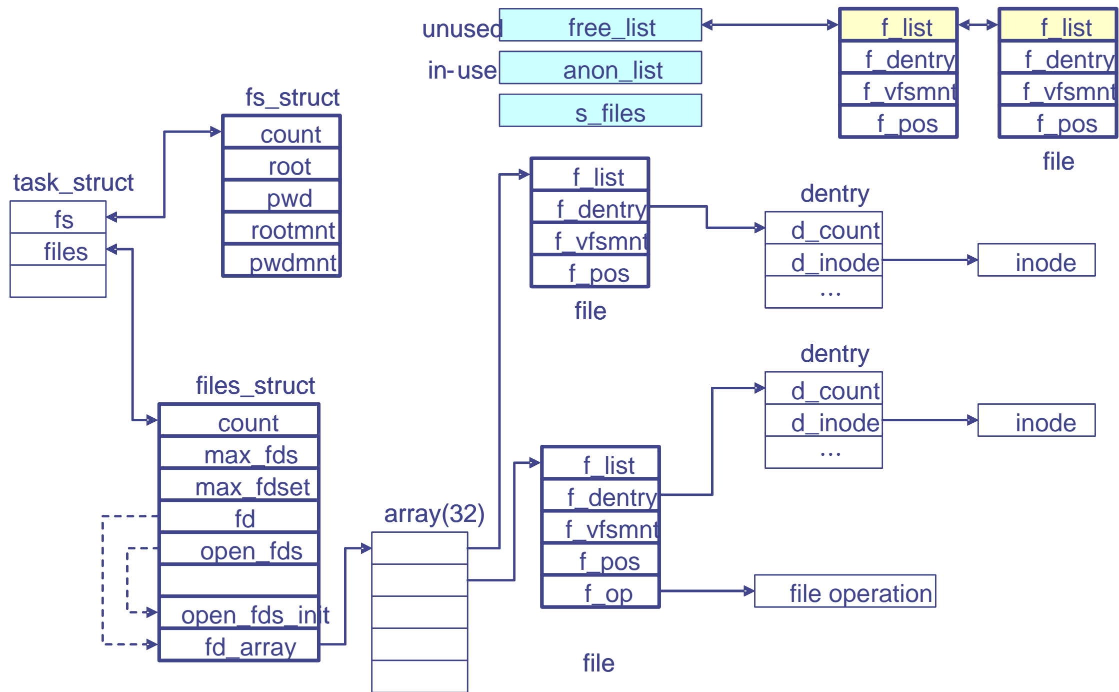

그림 8. 프로세스 관련 파일 연관 관계

제 4장 파일 시스템 유형 등록

실제 파일시스템을 위한 코드를 커널 이미지에 포함할 수도 있고, 동적으로 모듈로 로드할 수도 있다. 모듈을 링크할 때 모듈의 객체 코드에 들어 있는 전역 커널 심볼(변수와 함수)에 대한 참조는 모두 올바른 주소로 교체되어야 한다. 이 임무는 insmod가 담당한다. module_init()에서 불리어지는 함수(ext3의 경우, init_ext2_fs(), fs/ext2/super.c)에서 register_filesystem() 함수를 이용하여 파일시스템 유형 등록 작업을 수행한다.

- file_system_type 객체의 필드에 struct module의 포인터 유형으로 파일시스템을 구현하는 모듈의 포인터를 가리키는 필드가 있다. file_system_type 객체는 파일시스템을 등록할 때 사용된다.

```
static struct file_system_type ext2_fs_type = {
        .owner          = THIS_MODULE,
        .name           = "ext2",
        .get_sb         = ext2_get_sb,
        .kill_sb        = kill_block_super,
        .fs_flags       = FS_REQUIRES_DEV,
};
 static int __init init_ext2_fs(void)
{
        int err = init_ext2_xattr();
        if (err)
                return err;
        err = init_inodecache();
        if (err)
                goto out1;
        err = register_filesystem(&ext2_fs_type);
        if (erar)
                goto out;
        return 0;
```

```
 out:
         destroy_inodecache();
 out1:
         exit_ext2_xattr();
         return err;
}
```

제 5장 리눅스 파일 락킹

□ 개요

리눅스는 권고적(advisory), 강제적(mandatory) 락을 제공하는 fcntl(), flock(), lockf() 시스템 콜을 모두 지원한다. 리눅스는 file_lock 자료 구조로 파일 락을 나타낸다. 모든 file_lock 자료 구조는 이중 연결 링크 리스트로 구성된다.

□ 세부 자료 구조

유형	필드	설명	비고
struct file_lk *	fl_next	아이노드 리스트의 다음 요소	file_lock
struct list_head	fl_link	전역 리스트 포인터	
struct list_head	fl_block	block된 프로세스 리스트 포인터	
struct file_strct *	fl_owner	소유자의 files_struct	files_struct
unsigned int	fl_pid	프로세스 소유자의 PID	
wait_queue_head_t	fl_wait	블록된 프로세스의 대기 큐	
struct file*	fl_file	파일 객체 포인터	file
unsigned char	fl_flags	락 플래그	
unsigned char	fl_type	락 유형	
loff_t	fl_start	락 상태 영역 시작 오프셋	
loff_t	fl_end	락 상태 영역 끝 오프셋	
struct fasync_start *	fl_fasync	리스(lease)[*]중단을 알리는 데 사용	
unsigned long	fl_break_time	리스가 중단되기까지 남은 시간	
struct file_lock_operations *	fl_ops	파일 락 오퍼레이션 포인터	파일 락 오퍼레이션 포인터
struct lock_manager_operation *	fl_mop	락 관리 오퍼레이션 포인터	락 관리 오퍼레이션 포인터

[*] 리스(lease)는 flock() 기반 강제적 락을 가리킨다. 리스로 보호한 파일을 다른 프로세스가 열려고 하면 다른 경우와 마찬가지로 차단되지만, 락을 소유한 프로세스가 시그널을 받는다. 시스널을 받은 프로세스는 먼저 내용의 일관성을 유지하기 위해 디스크의 파일을 갱신하고, 락을 풀어야 한다. 소유한 프로세스가 시스템에 정의된 시간 간격(초 단위 수를 /proc/sys/fs/lease-break-time에 기록하면 되며, 보통 45초다) 동안 락을 풀지 않으면, 커널이 리스를 자동으로 제거하고, 차단된 프로세스의 실행을 허가한다.

union	fl_u	파일 시스템별 정보 (struct nfs_lock_info nfs_fl; struct nfs4_lock_info nfs4_fl; 중에 선택하게 되어 있음)	

□ 오퍼레이션

```
struct file_lock_operations {
        void (*fl_insert)(struct file_lock *);  /* lock insertion callback */
        void (*fl_remove)(struct file_lock *);  /* lock removal callback */
        void (*fl_copy_lock)(struct file_lock *, struct file_lock *);
        void (*fl_release_private)(struct file_lock *);
};
```

```
struct lock_manager_operations {
        int (*fl_compare_owner)(struct file_lock *, struct file_lock *);
        void (*fl_notify)(struct file_lock *);  /* unblock callback */
        void (*fl_copy_lock)(struct file_lock *, struct file_lock *);
        void (*fl_release_private)(struct file_lock *);
        void (*fl_break)(struct file_lock *);
        int (*fl_mylease)(struct file_lock *, struct file_lock *);
        int (*fl_change)(struct file_lock **, int);
};
```

* 파일 락 관련 오퍼레이션은 include/linux/fs.h에 정의되어 있다.

* fs/lock.c 의 flock_make_lock, flock_to_posix_lock와 같은 함수들을 구현하는데 위의 오퍼레이션이 사용된다. 그리고 locks.c의 함수들은 fcntl()(fs/fcntl.c), flock()(fs/locks.c)를 구현하는데 사용된다. 각 오퍼레이션의 접근 방법은 아래와 같이 각 오퍼레이션이 구현되어 있는지 체크하고 사용하는 형태를 취한다.

```
if (fl->fl_ops && fl->fl_ops->fl_insert)
        fl->fl_ops->fl_insert(fl);
```

□ file_lock 중심 연관 관계

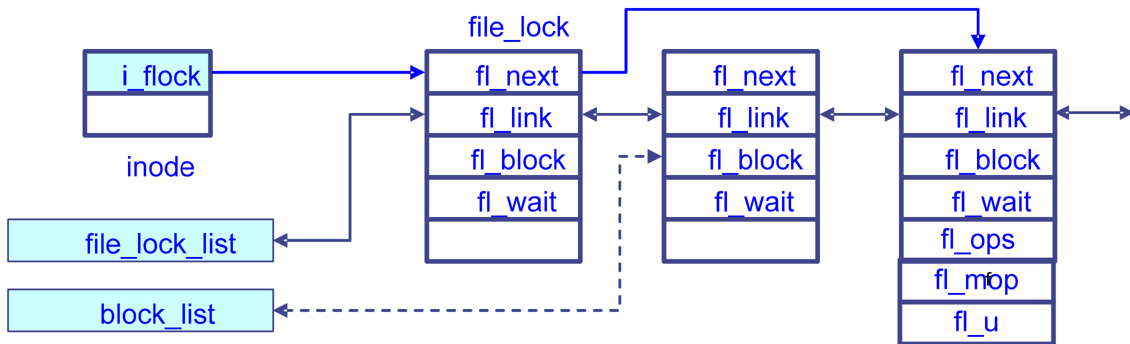

그림 9. file_lock 중심 연관 관계

제 6장 디스크 캐시

캐시에 관련 된 기본 용어를 정리하면 다음과 같다.

• sector : 블록 디바이스 물리적인 최소 단위(512바이트(보통),1024, 2048바이트)

• block [*]: 블록 디바이스 논리 최소 단위(sector의 배수), 파일 시스템 디스크 연산의 기본 I/O 단위, 보통 512, 1024, 2048, 4096바이트(on 80 X 86 아키텍쳐)

• 버퍼 : 커널 메모리에 있는 디스크 블록을 표현하는 단위

• 세그먼트 : DMA(Direct Memory Access) 연산을 위한 단위, 한 페이지가 될 수도 있고, 한 페이지의 일부가 될 수도 있음(연속된 디스크 섹터의 데이터를 가짐)

• 페이지 : 한 파일에 대한 여러 개의 블록으로 이루어져 있음, 페이지 내의 블록들은 논리적으로 연속(물리적으로 연속일 필요 없음)

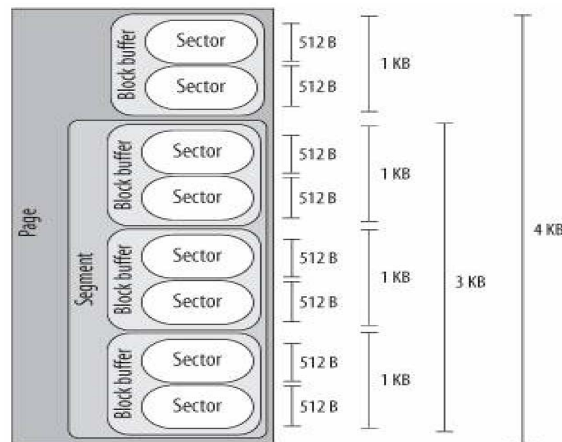

그림 10. 섹터, 버퍼, 세스먼트, 페이지 구성(참조. 리눅스 커널의 이해 3th)

• 버퍼 헤더 : 각 버퍼를 관리하기 위한 자료 구조. 해당 블록이 어떤 디바이스의 어떤 블록과 상관있으며 상태 등의 정보 관리

• address_space : 페이지 캐시에 있는 페이지 식별자 구조체, 물리적 디스크 블록에 대한 정보 대신 이 구조체로 페이지 구분

* 페이지 캐시와 버퍼 캐시가 서로 다른 디스크 캐시이지만, 리눅스 2.4 버전에서는 서로 공유되어 있다.

---

[*] 최근 리눅스 시스템은 성능을 높이기 위해 일반적으로 허용하는 가장 큰 블록 크기인 4096바이트를 주로 사용한다고 함.(참조. 리눅스 커널의 이해, 2th, 역자주, p578