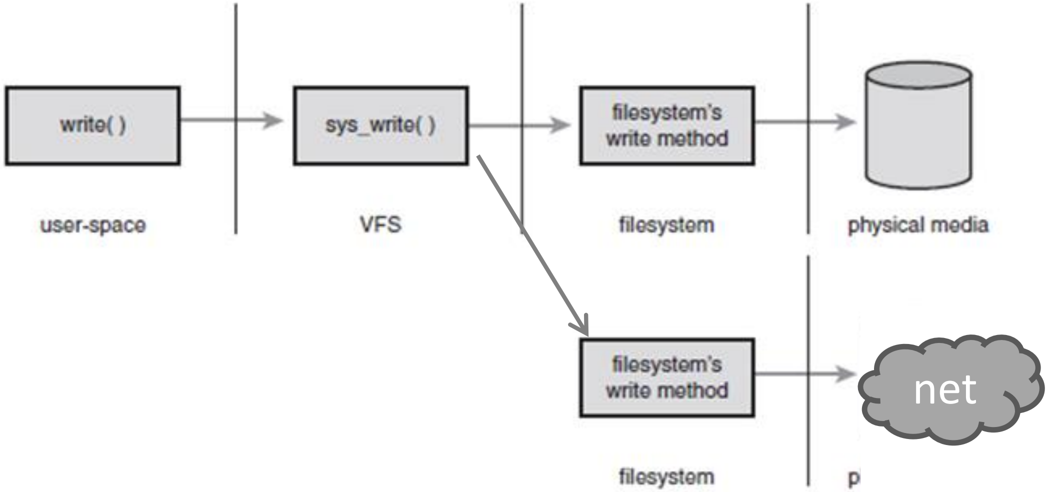
Unit 7: File System Driver

Prof. Li-Pin Chang

ESSLab@NCTU

Linux VFS

- Virtual File-system Switch
  - Provides an abstraction layer between user space and file system drivers
  - A set of common operations for various file system implementations
    - ext234 (disk)
    - nfs (networ)
    - yaffs2 (flash)

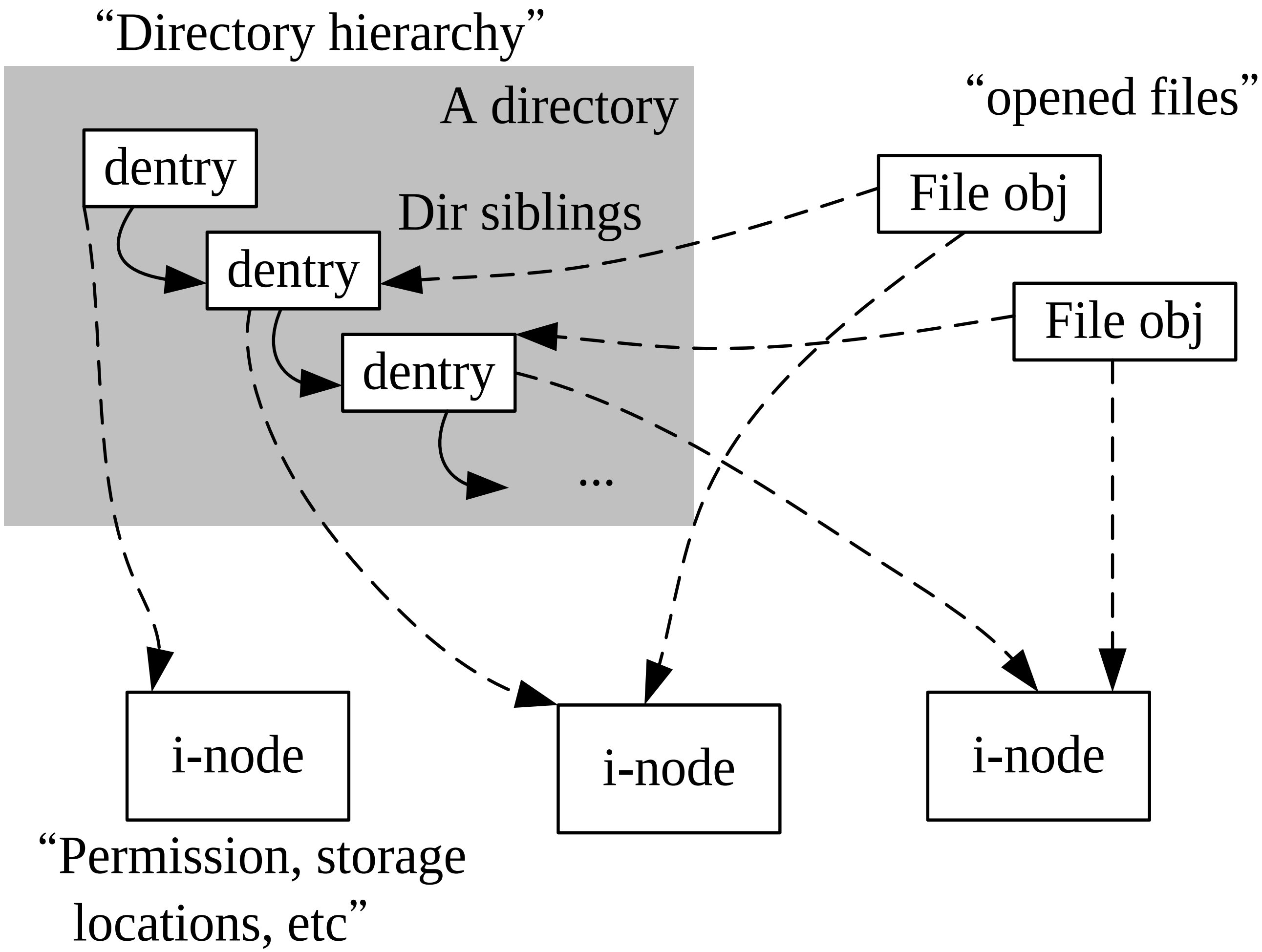


VFS Memory Objects

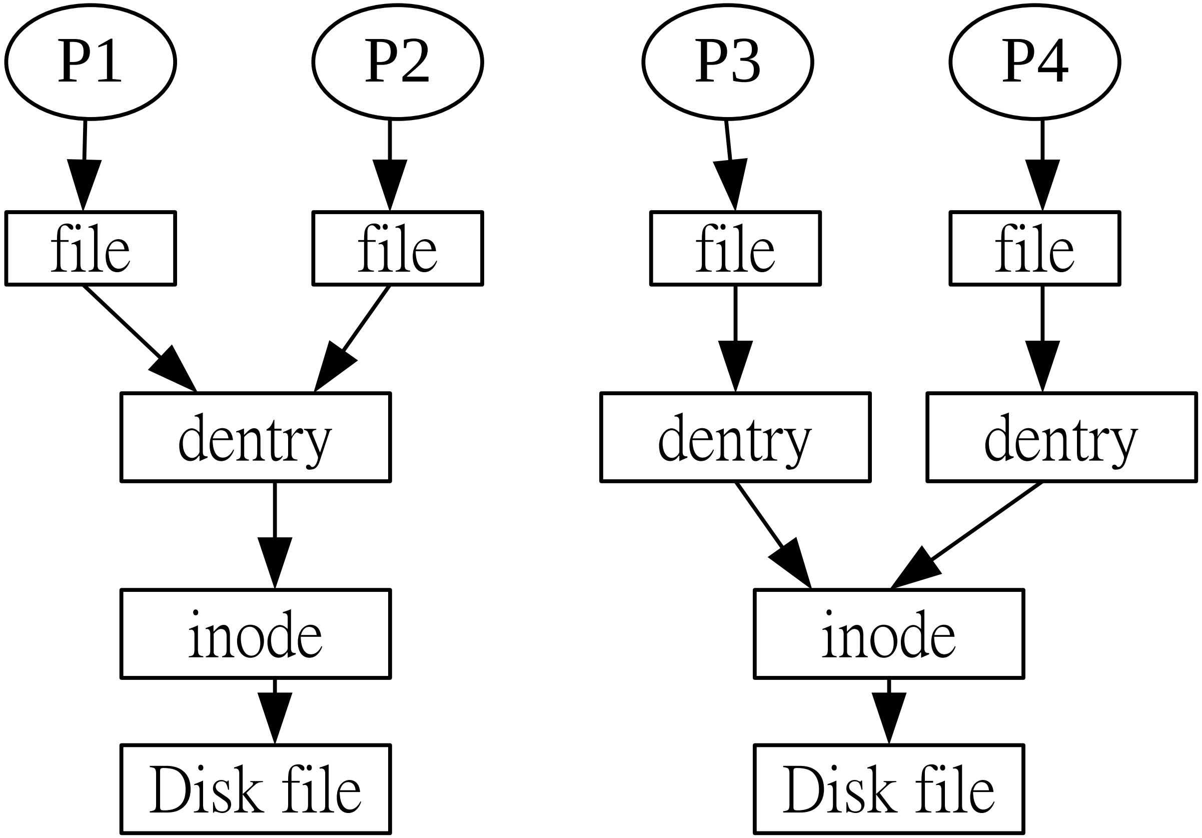
- *Superblock* object
  - represent the entire file system
- *Inode* object
  - represent an individual file
- *file* object
  - represent an opened file
- *dentry* object
  - represent an individual directory entry

- Each type of object is associated with a set of operations

VFS Memory Objects



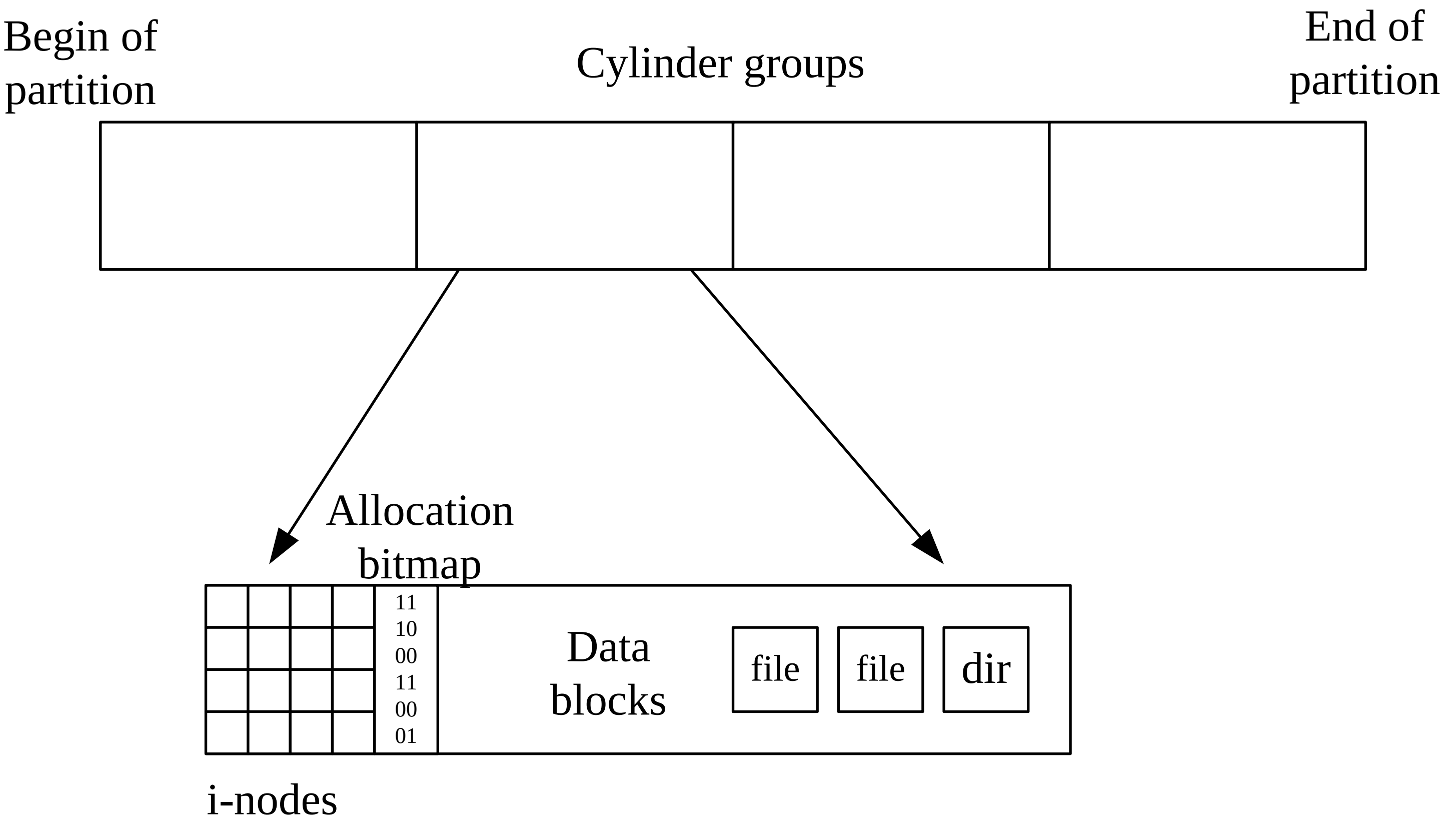
VFS Object Relationship



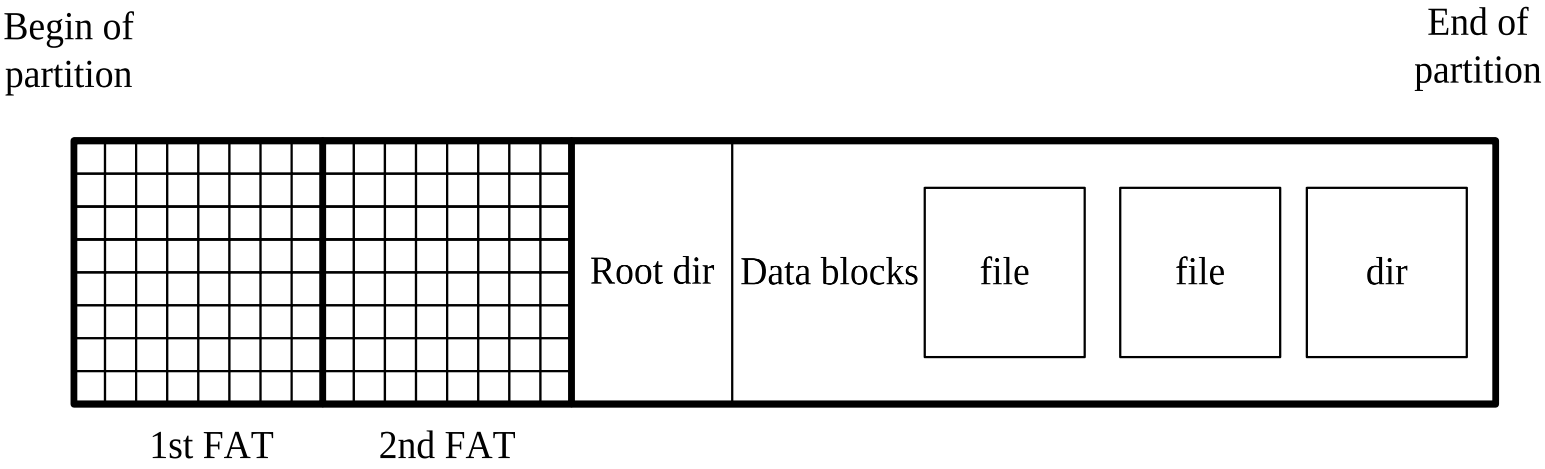
File-System Metadata

- In disk, specific to file system implementation
  - Linux ext234 file system
    - Super block
    - Inode
    - Allocation bitmaps
  - Microsoft FAT file system
    - File allocation tables
    - Directory
- File system drivers must create VFS memory objects based on their metadata
  - Ext234 copies inodes from disk to memory
  - FAT file system does not have disk inodes; it creates inodes on the fly

Layout of the Linux ext 2/3/4 file system



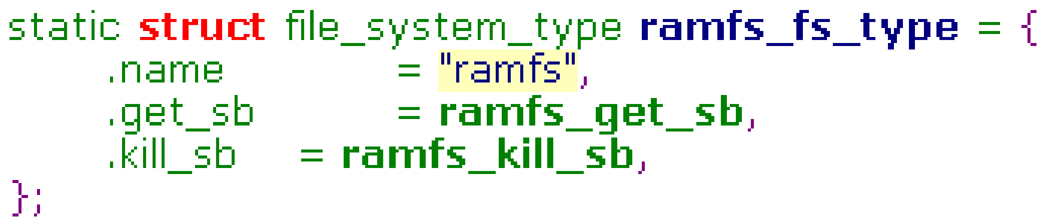
The layout of FAT 12/16/32 file system



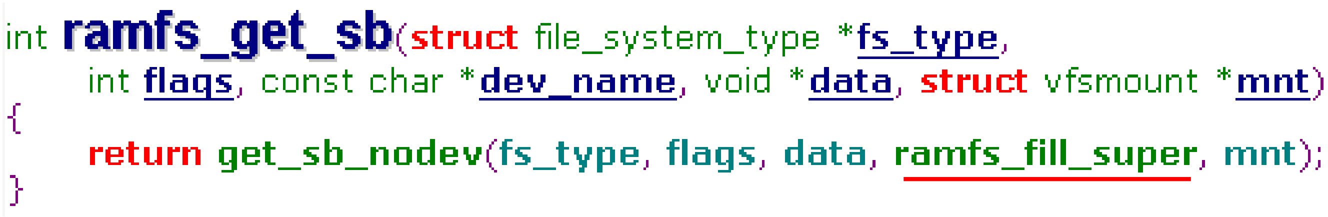
ramfs

- In this lecture, we study ramfs
- ramfs is a file system that allocates memory pages for storing file data
  - Losing all pages after umounting!
  - Pages cannot be swapped to disk
    - Fast, but uses more memory
    - Use *tmpfs* instead if you need to create a very large ram file system
- ramfs does not write to disk
  - mount –t ramfs none /mnt

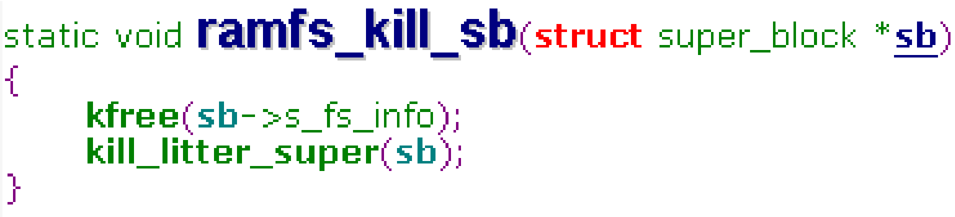
ramfs Initialization



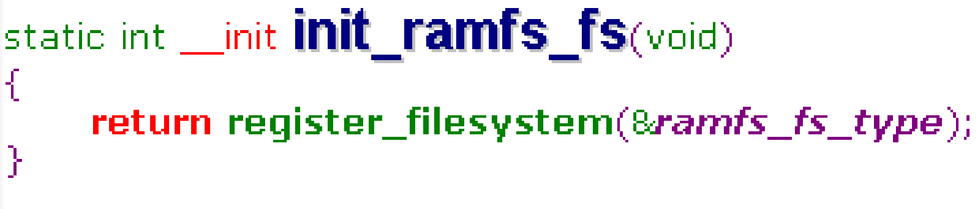
```
static struct file_system_type ramfs_fs_type = {
        .name           = "ramfs",
        .get_sb         = ramfs_get_sb,
        .kill_sb     = ramfs_kill_sb,
};
```



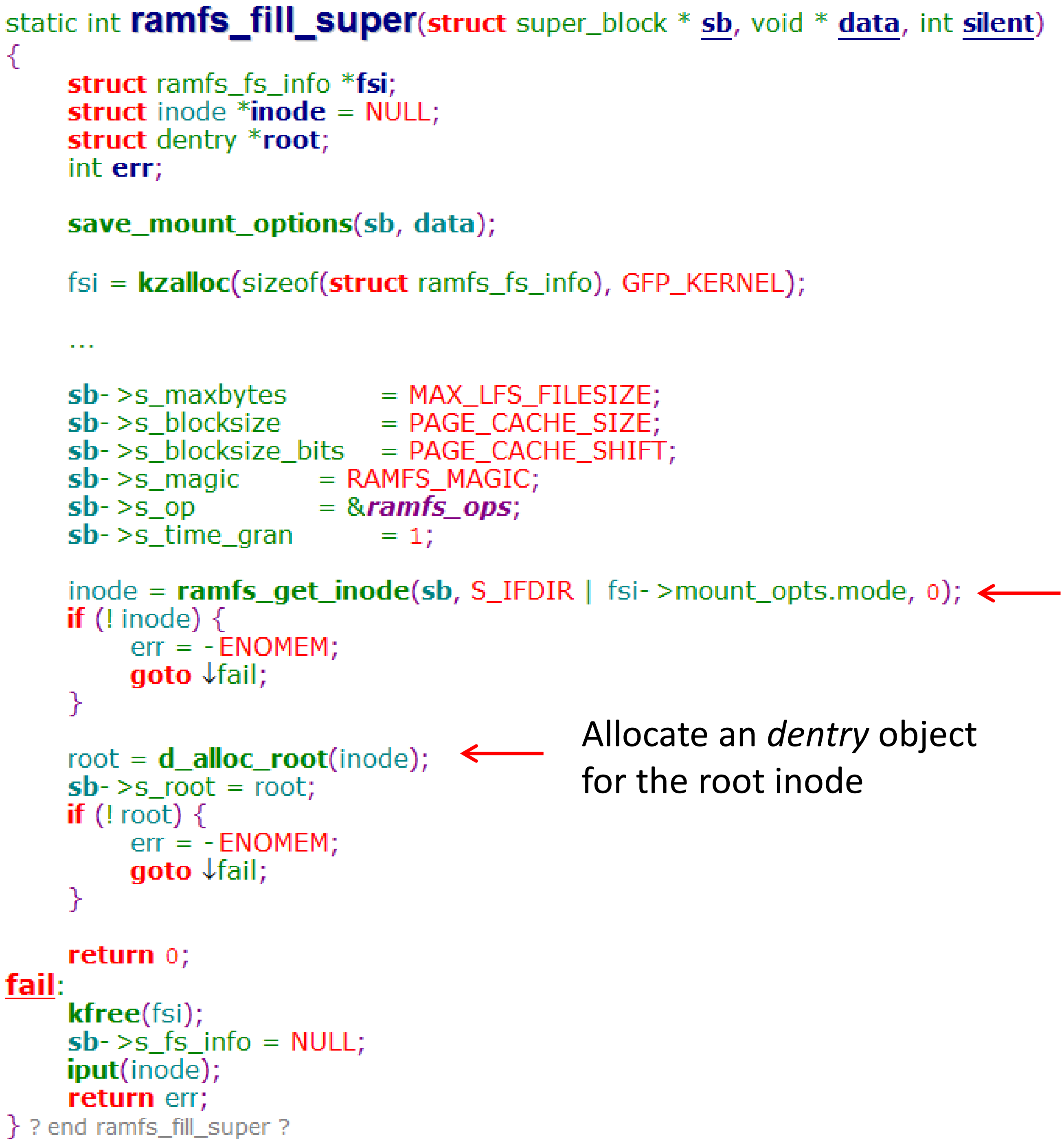
```
int ramfs_get_sb(struct file_system_type *fs_type,
        int flags, const char *dev_name, void *data, struct vfsmount *mnt)
{
        return get_sb_nodev(fs_type, flags, data, ramfs_fill_super, mnt);
}
```



```
static void ramfs_kill_sb(struct super_block *sb)
{
        kfree(sb->s_fs_info);
        kill_litter_super(sb);
}
```



```
static int __init init_ramfs_fs(void)
{
        return register_filesystem(&ramfs_fs_type);
}
```



```
static int ramfs_fill_super(struct super_block * sb, void * data, int silent)
{
        struct ramfs_fs_info *fsi;
        struct inode *inode = NULL;
        struct dentry *root;
        int err;

        save_mount_options(sb, data);

        fsi = kzalloc(sizeof(struct ramfs_fs_info), GFP_KERNEL);

        ...

        sb->s_maxbytes          = MAX_LFS_FILESIZE;
        sb->s_blocksize         = PAGE_CACHE_SIZE;
        sb->s_blocksize_bits    = PAGE_CACHE_SHIFT;
        sb->s_magic         = RAMFS_MAGIC;
        sb->s_op            = &ramfs_ops;
        sb->s_time_gran         = 1;

        inode = ramfs_get_inode(sb, S_IFDIR | fsi->mount_opts.mode, 0);
        if (! inode) {
                err = -ENOMEM;
                goto ↓fail;
        }

        root = d_alloc_root(inode);
        sb->s_root = root;
        if (! root) {
                err = -ENOMEM;
                goto ↓fail;
        }

        return 0;
fail:
        kfree(fsi);
        sb->s_fs_info = NULL;
        iput(inode);
        return err;
} ? end ramfs_fill_super ?
```

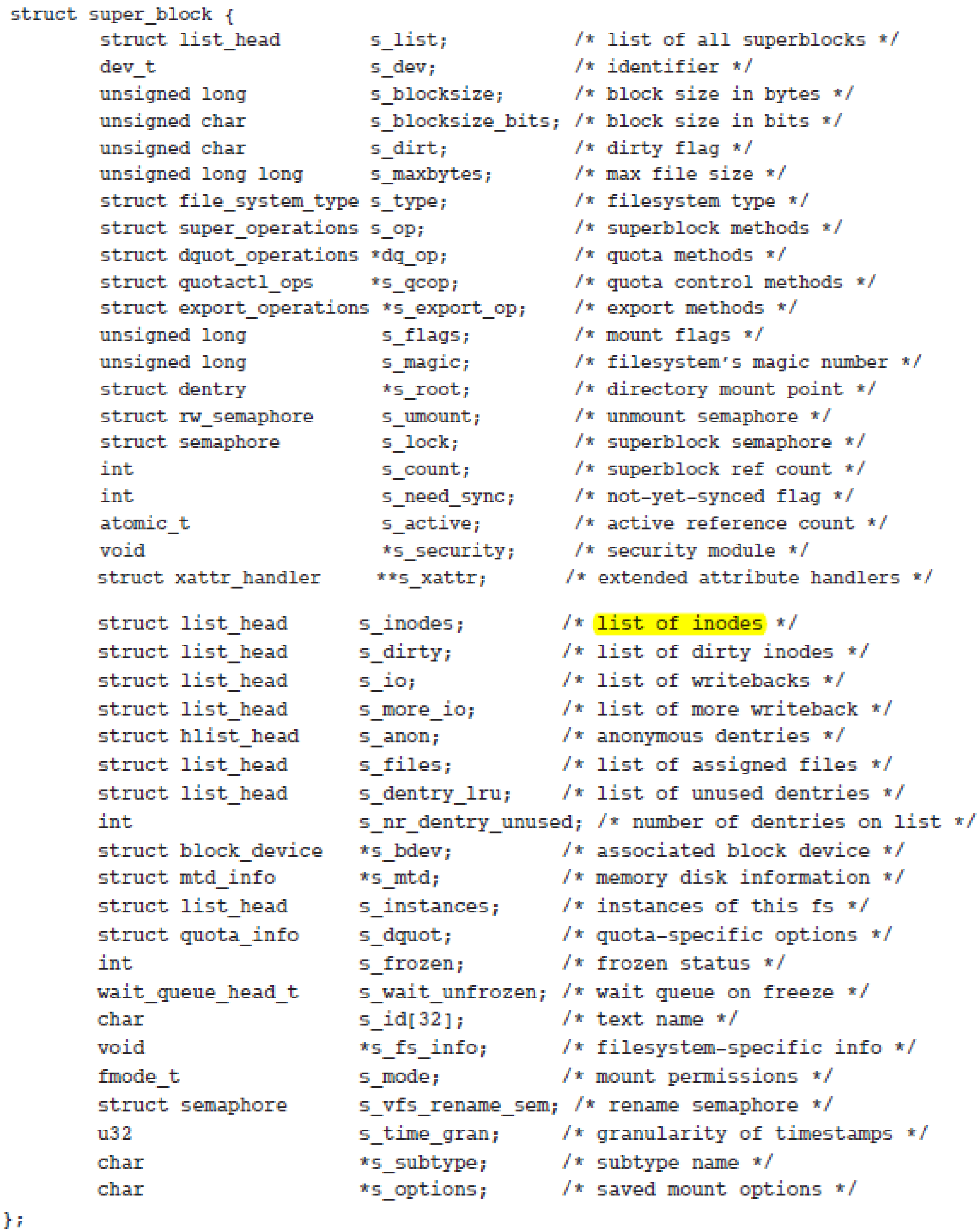
Allocate an *inode* for the root directory

Allocate an *dentry* object for the root inode

Superblock Object

- An object that contains information regarding the whole file system
  - The device that the file system is mounted on
  - Block size (multiple of disk sectors)
  - A dirty flag (needing fsck or not)

- A superblock is created when file system is mounted on a device
  - ramfs_getsb()



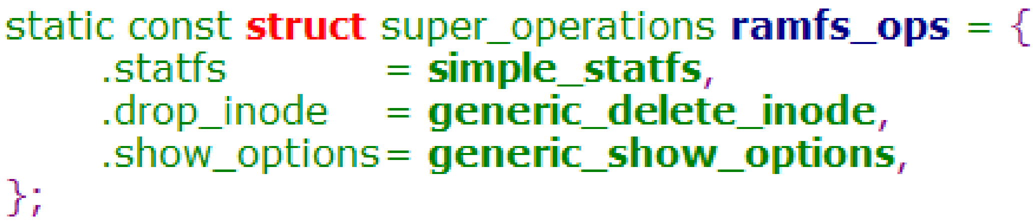
```
struct super_block {
        struct list_head        s_list;           /* list of all superblocks */
        dev_t                   s_dev;            /* identifier */
        unsigned long           s_blocksize;      /* block size in bytes */
        unsigned char           s_blocksize_bits; /* block size in bits */
        unsigned char           s_dirt;           /* dirty flag */
        unsigned long long      s_maxbytes;       /* max file size */
        struct file_system_type s_type;           /* filesystem type */
        struct super_operations s_op;             /* superblock methods */
        struct dquot_operations *dq_op;           /* quota methods */
        struct quotactl_ops     *s_qcop;          /* quota control methods */
        struct export_operations *s_export_op;    /* export methods */
        unsigned long            s_flags;         /* mount flags */
        unsigned long            s_magic;         /* filesystem's magic number */
        struct dentry            *s_root;         /* directory mount point */
        struct rw_semaphore      s_umount;        /* unmount semaphore */
        struct semaphore         s_lock;          /* superblock semaphore */
        int                      s_count;         /* superblock ref count */
        int                      s_need_sync;     /* not-yet-synced flag */
        atomic_t                 s_active;        /* active reference count */
        void                     *s_security;     /* security module */
        struct xattr_handler     **s_xattr;      /* extended attribute handlers */

        struct list_head       s_inodes;         /* list of inodes */
        struct list_head       s_dirty;          /* list of dirty inodes */
        struct list_head       s_io;             /* list of writebacks */
        struct list_head       s_more_io;        /* list of more writeback */
        struct hlist_head      s_anon;           /* anonymous dentries */
        struct list_head       s_files;          /* list of assigned files */
        struct list_head       s_dentry_lru;     /* list of unused dentries */
        int                    s_nr_dentry_unused; /* number of dentries on list */
        struct block_device    *s_bdev;          /* associated block device */
        struct mtd_info        *s_mtd;           /* memory disk information */
        struct list_head       s_instances;      /* instances of this fs */
        struct quota_info      s_dquot;          /* quota-specific options */
        int                    s_frozen;         /* frozen status */
        wait_queue_head_t      s_wait_unfrozen;  /* wait queue on freeze */
        char                   s_id[32];         /* text name */
        void                   *s_fs_info;       /* filesystem-specific info */
        fmode_t                s_mode;           /* mount permissions */
        struct semaphore       s_vfs_rename_sem;  /* rename semaphore */
        u32                    s_time_gran;      /* granularity of timestamps */
        char                   *s_subtype;       /* subtype name */
        char                   *s_options;       /* saved mount options */
};
```

Essential members

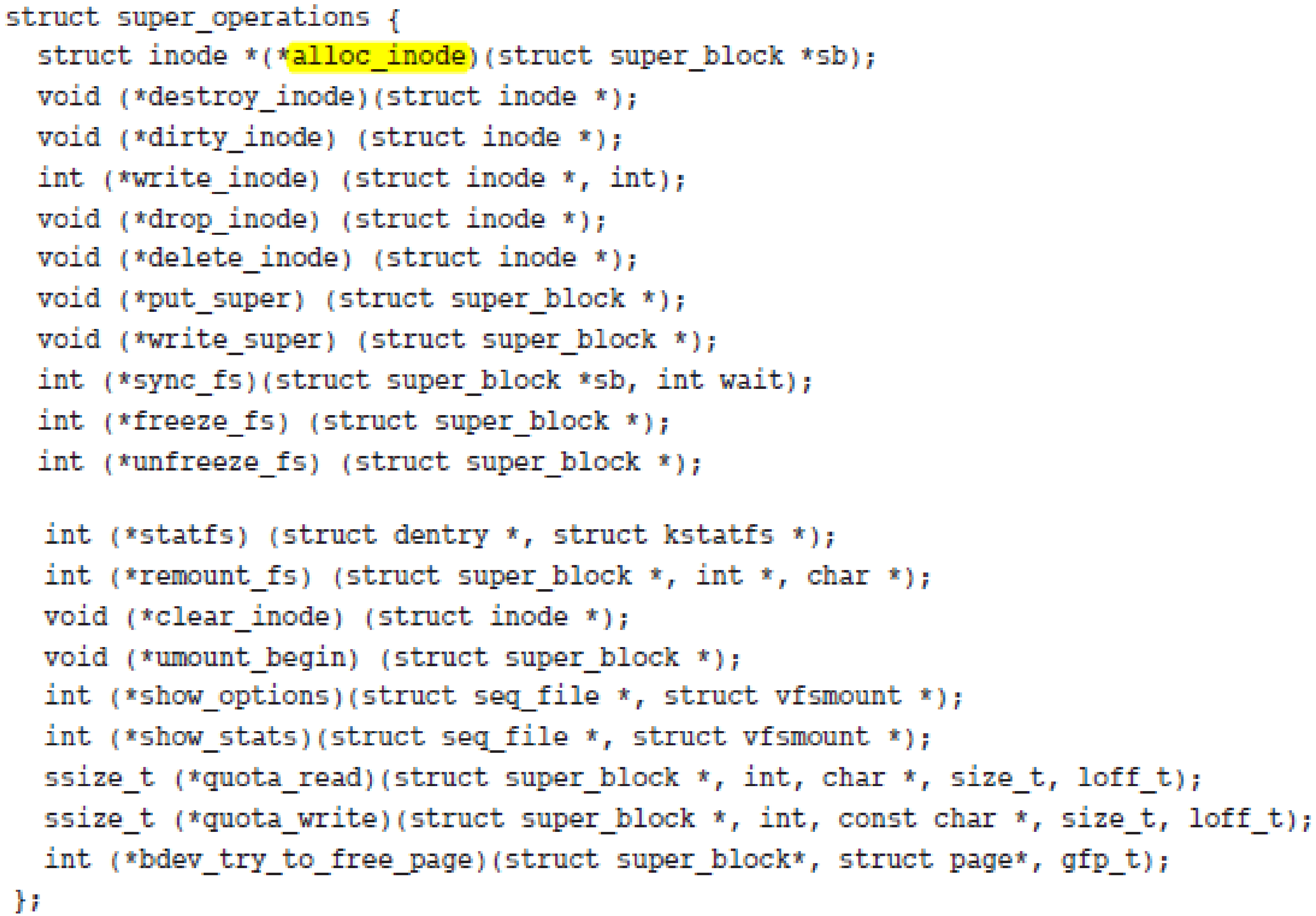
- s_dirt
- s_op
- *s_root
- s_inodes
- s_dirty
- s_files
- *s_bdev

Superblock Operations

- A structure of ~20 callbacks
  - Operations applied to file system
    - Getting file system statistics
    - Synching all dirty data
  - inode allocation/de-allocation (in ram and in disk)
- A file system may or may not register a callback for each superblock operation
  - If a callback is NULL, the VFS simply does nothing or calls the generic handler
- ramfs superblock operations



```
static const struct super_operations ramfs_ops = {
        .statfs         = simple_statfs,
        .drop_inode     = generic_delete_inode,
        .show_options= generic_show_options,
};
```



```
struct super_operations {
  struct inode *(*alloc_inode)(struct super_block *sb);
  void (*destroy_inode)(struct inode *);
  void (*dirty_inode) (struct inode *);
  int (*write_inode) (struct inode *, int);
  void (*drop_inode) (struct inode *);
  void (*delete_inode) (struct inode *);
  void (*put_super) (struct super_block *);
  void (*write_super) (struct super_block *);
  int (*sync_fs)(struct super_block *sb, int wait);
  int (*freeze_fs) (struct super_block *);
  int (*unfreeze_fs) (struct super_block *);

  int (*statfs) (struct dentry *, struct kstatfs *);
  int (*remount_fs) (struct super_block *, int *, char *);
  void (*clear_inode) (struct inode *);
  void (*umount_begin) (struct super_block *);
  int (*show_options)(struct seq_file *, struct vfsmount *);
  int (*show_stats)(struct seq_file *, struct vfsmount *);
  ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
  ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
  int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
};
```

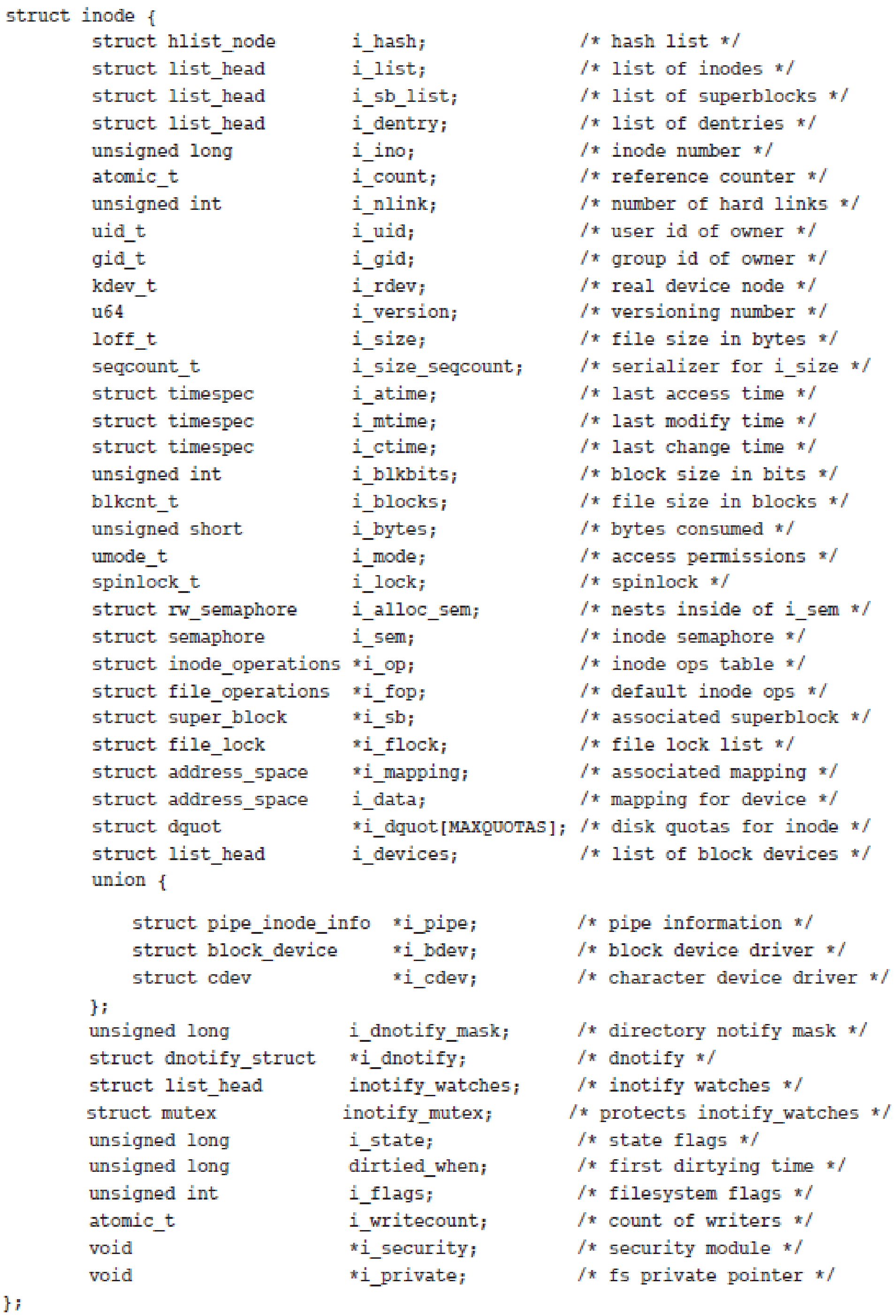
inode

- inode is an object associated with a disk file
  - Contains all the information needed to manipulate a file
    - File size, timstamps, disk locations, permissions, etc
  - Unique to each file, independent of file name

- An inode is constructed in memory when a file is accessed
  - Some file systems have inode in disk (ext234), so just copy it from disk
  - Many file systems have no disk inodes (FAT, NTFS, …), construct on the fly

inode lists

- An inode resides in one of the following lists (via *i_list*)
  - A global list named *inode_unused*
    - Clean inodes but currently no processes use them
  - A global list named *inode_inuse*
    - Clean inodes that some processes refer to them
  - A list *s_dirty* in the corresponding superblock
    - Modified inodes awaiting write back
- A superblock also chains all its inodes in a list *s_inode*
  - Via *i_sb_list*



```
struct inode {
        struct hlist_node       i_hash;              /* hash list */
        struct list_head        i_list;              /* list of inodes */
        struct list_head        i_sb_list;           /* list of superblocks */
        struct list_head        i_dentry;            /* list of dentries */
        unsigned long           i_ino;               /* inode number */
        atomic_t                i_count;             /* reference counter */
        unsigned int            i_nlink;             /* number of hard links */
        uid_t                   i_uid;               /* user id of owner */
        gid_t                   i_gid;               /* group id of owner */
        kdev_t                  i_rdev;              /* real device node */
        u64                     i_version;           /* versioning number */
        loff_t                  i_size;              /* file size in bytes */
        seqcount_t              i_size_seqcount;     /* serializer for i_size */
        struct timespec         i_atime;             /* last access time */
        struct timespec         i_mtime;             /* last modify time */
        struct timespec         i_ctime;             /* last change time */
        unsigned int            i_blkbits;           /* block size in bits */
        blkcnt_t                i_blocks;            /* file size in blocks */
        unsigned short          i_bytes;             /* bytes consumed */
        umode_t                 i_mode;              /* access permissions */
        spinlock_t              i_lock;              /* spinlock */
        struct rw_semaphore     i_alloc_sem;         /* nests inside of i_sem */
        struct semaphore        i_sem;               /* inode semaphore */
        struct inode_operations *i_op;               /* inode ops table */
        struct file_operations  *i_fop;              /* default inode ops */
        struct super_block      *i_sb;               /* associated superblock */
        struct file_lock        *i_flock;            /* file lock list */
        struct address_space    *i_mapping;          /* associated mapping */
        struct address_space    i_data;              /* mapping for device */
        struct dquot            *i_dquot[MAXQUOTAS]; /* disk quotas for inode */
        struct list_head        i_devices;           /* list of block devices */
        union {

            struct pipe_inode_info  *i_pipe;         /* pipe information */
            struct block_device     *i_bdev;         /* block device driver */
            struct cdev             *i_cdev;         /* character device driver */
        };
        unsigned long           i_dnotify_mask;      /* directory notify mask */
        struct dnotify_struct   *i_dnotify;          /* dnotify */
        struct list_head        inotify_watches;     /* inotify watches */
        struct mutex            inotify_mutex;       /* protects inotify_watches */
        unsigned long           i_state;             /* state flags */
        unsigned long           dirtied_when;        /* first dirtying time */
        unsigned int            i_flags;             /* filesystem flags */
        atomic_t                i_writecount;        /* count of writers */
        void                    *i_security;         /* security module */
        void                    *i_private;          /* fs private pointer */
};
```

Essential members

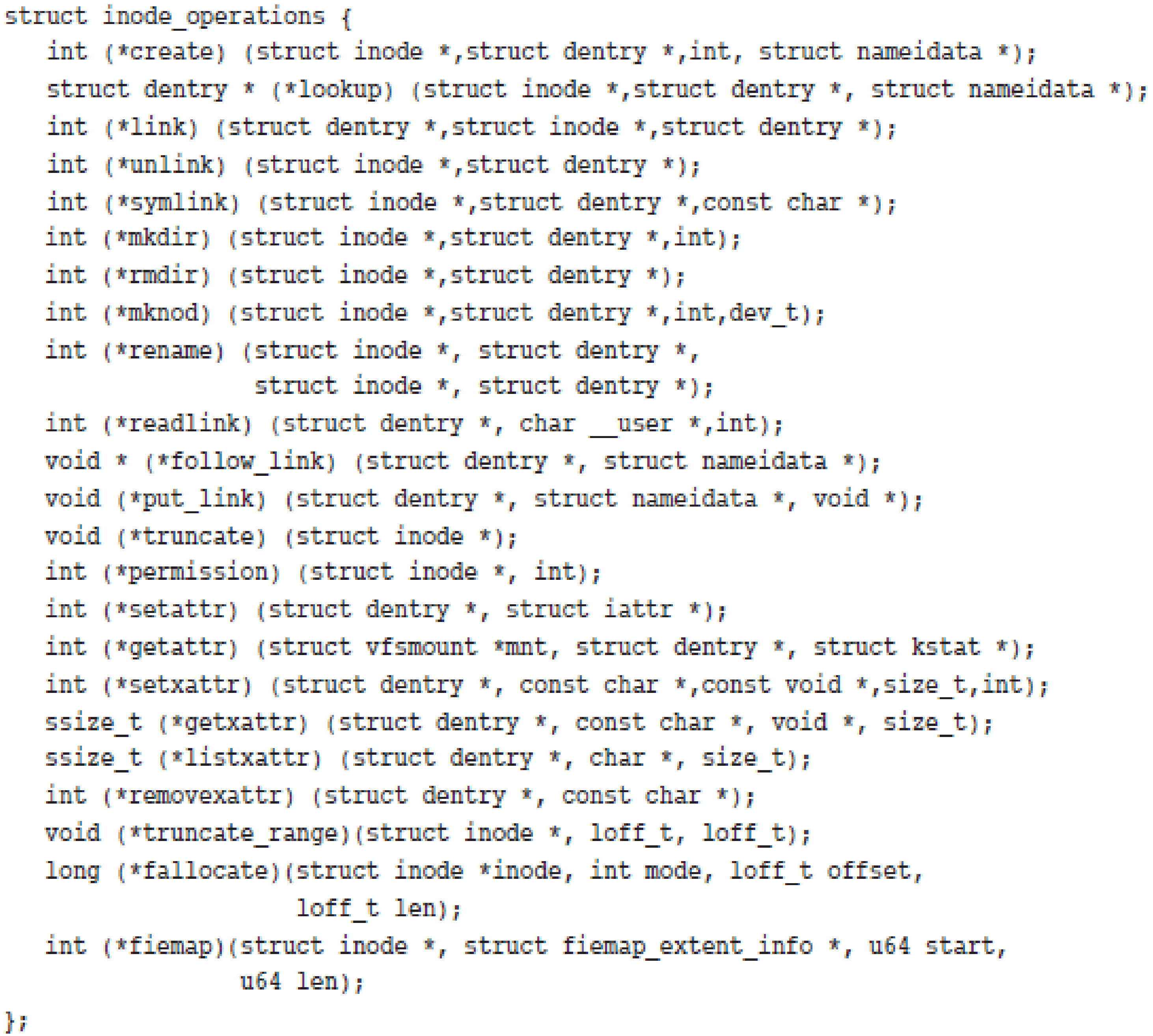
- i_list (the current list)
- i_sb_list (all inodes of the same sb)
- i_dentry
- i_ino
- i_nlink (hard link count)
- i_uid
- i_gid
- i_count (ref count)
- i_size
- i_blocks
- i_?time
- i_mode
- *i_op
- *i_sb
- *i_bdev

inode Structure

- Lists
  - i_list : pointers for the list that describes the inode’s current state
  - i_sb_list: pointers for the list of inodes of the superblock
- Permissions
  - i_uid & i_gid: file user ID and group ID
  - i_mode : file access mode rwxrwxrwx
- Hard-links
  - i_nlink : number of hard links
    - When down to 0, the inode (and the file) can be deleted from disk
  - i_dentry : the head of the list of dentry objects referencing this inode
- Time
  - i_atime, i_mtime, i_ctime: access, modify, create times

inode Structure

- Reference
  - i_count: how many processes refer to this inode
  - When down to 0, the inode can be freed from memory
- inode operations
  - *i_op: a pointer to a structure of callbacks
- Device
  - *i_bdev: a pointer to the underlying block device
- File-system private data
  - *i_private: a pointer to FS-specific extension, e.g., direct/indirect/double indirect/triple indirect pointers of ext2 file system



```
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);
   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 *);
   void (*truncate_range)(struct inode *, loff_t, loff_t);
   long (*fallocate)(struct inode *inode, int mode, loff_t offset,
                     loff_t len);
   int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
                 u64 len);
};
```

inode Operations (1/3)

- inode operations are structural operations, i.e., they may modify disk metadata
- *create*(dir, dentry, mode, nameidata)
  - Creates a new inode for a regular file associated with a dentry object in some directory. (new file)
- *lookup*(dir, dentry, nameidata)
  - Searches a directory for an inode corresponding to the filename included in a dentry object. (existing file)
- *link*(old_dentry, dir, new_dentry)
  - Creates a new hard link that refers to the file specified by old_dentry in the directory dir; the new hard link has the name specified by new_dentry.
- *unlink*(dir, dentry)
  - Removes the hard link of the file specified by a dentry object from a directory. (delete the file when hard link=0)

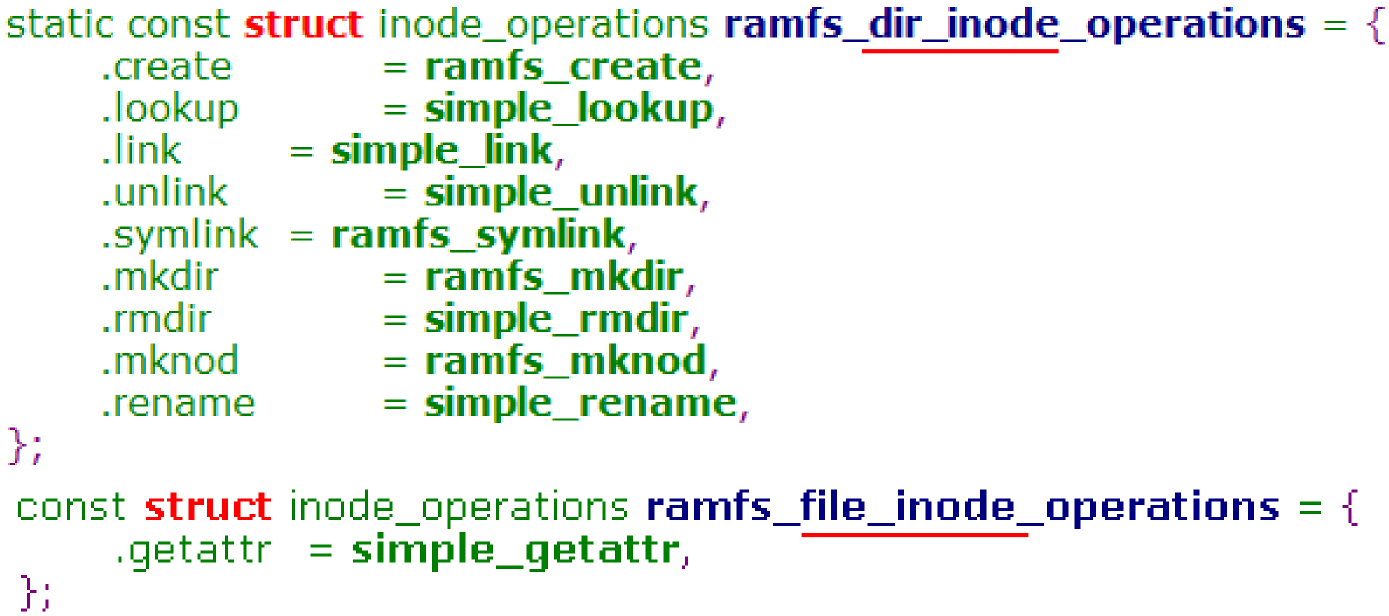
inode Operations (2/3)

- *mkdir*(dir, dentry, mode)
  - Creates a new inode for a directory associated with a dentry object in some directory.
- *rmdir*(dir, dentry)
  - Removes from a directory the subdirectory whose name is included in a dentry object.

inode Operations (3/3)

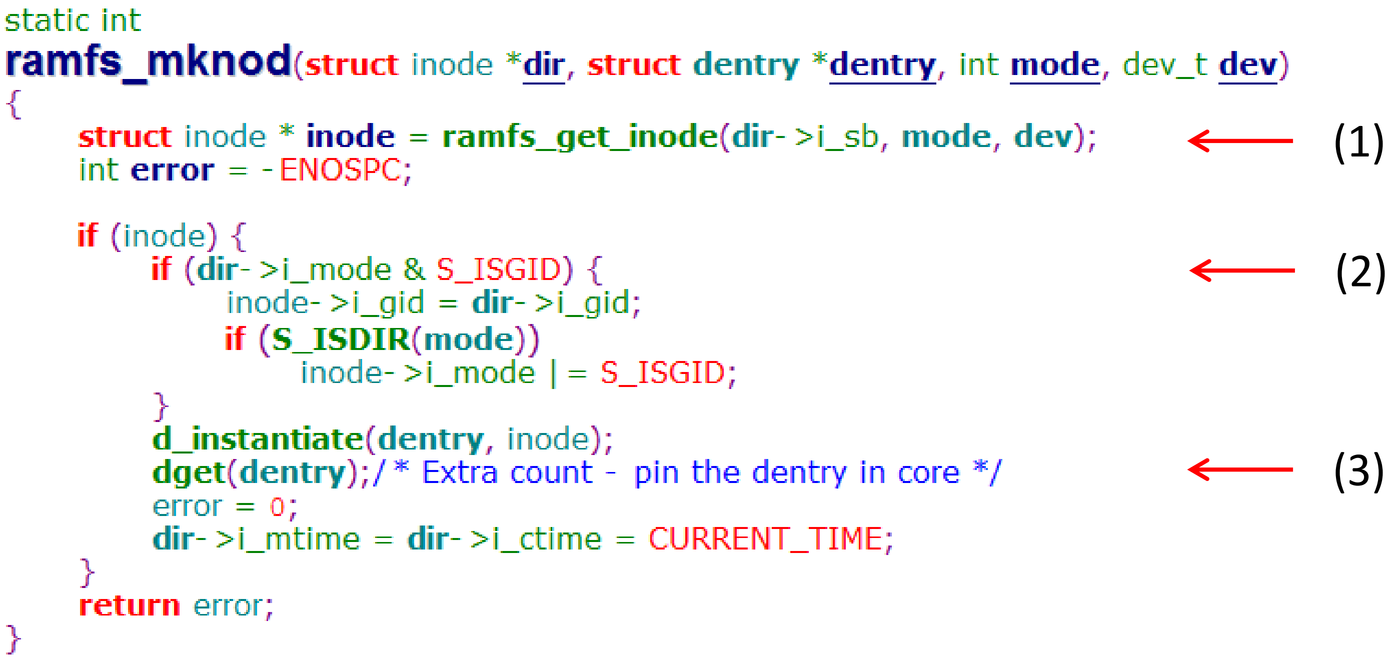
- *mknod*(dir, dentry, mode, rdev)
  - Creates a new disk inode for a special file associated with a dentry object in some directory.
  - The mode and rdev parameters specify, respectively, the file type and the device's major and minor numbers.
- *rename*(old_dir, old_dentry, new_dir, new_dentry)
  - Moves the file identified by old_entry from the old_dir directory to the new_dir one.
- *truncate*(inode)
  - Shrink the size of the file associated with an inode.

ramfs inode Operations



```
static const struct inode_operations ramfs_dir_inode_operations = {
        .create         = ramfs_create,
        .lookup         = simple_lookup,
        .link       = simple_link,
        .unlink         = simple_unlink,
        .symlink  = ramfs_symlink,
        .mkdir          = ramfs_mkdir,
        .rmdir          = simple_rmdir,
        .mknod          = ramfs_mknod,
        .rename         = simple_rename,
};
const struct inode_operations ramfs_file_inode_operations = {
        .getattr  = simple_getattr,
};
```

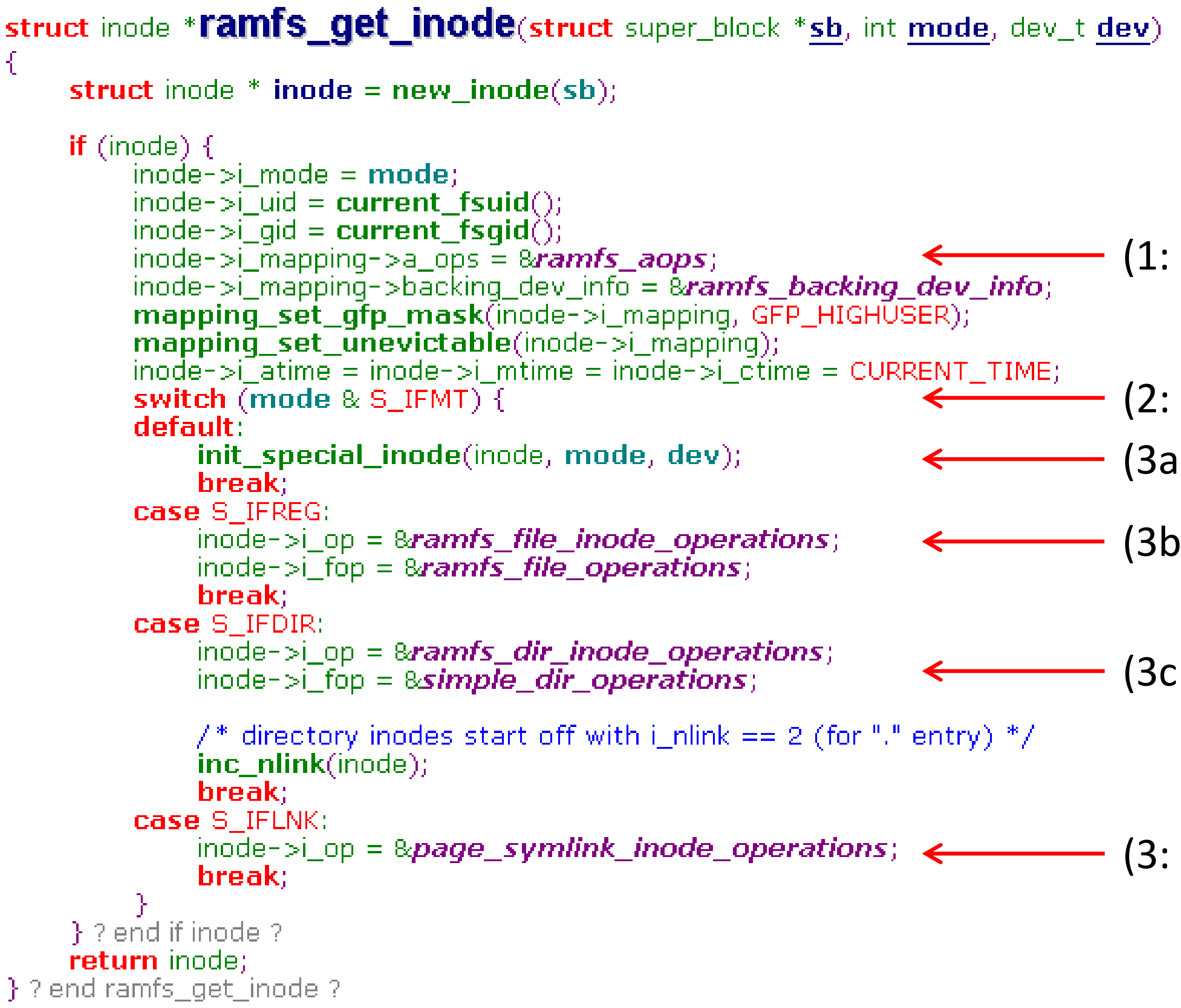
- On the creation of directory, device node, regular file, ramfs just allocates an inode in memory



```
static int
ramfs_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t dev)
{
        struct inode * inode = ramfs_get_inode(dir- >i_sb, mode, dev);
        int error = - ENOSPC;

        if (inode) {
                if (dir- >i_mode & S_ISGID) {
                        inode- >i_gid = dir- >i_gid;
                        if (S_ISDIR(mode))
                                inode- >i_mode | = S_ISGID;
                }
                d_instantiate(dentry, inode);
                dget(dentry);/ * Extra count - pin the dentry in core */
                error = 0;
                dir- >i_mtime = dir- >i_ctime = CURRENT_TIME;
        }
        return error;
}
```

ramfs_mknod() creates an inode for the dentry specified in the 2nd parameter.

(1) Calls *ramfs_get_inode*() to allocate a new inode
(2) If the parent directory has a group id, then the inode inherits the group id from the parent
(3) Increase the reference count of the dentry to prevent it from being de-allocated from memory



```
struct inode *ramfs_get_inode(struct super_block *sb, int mode, dev_t dev)
{
	struct inode * inode = new_inode(sb);

	if (inode) {
		inode->i_mode = mode;
		inode->i_uid = current_fsuid();
		inode->i_gid = current_fsgid();
		inode->i_mapping->a_ops = &ramfs_aops;
		inode->i_mapping->backing_dev_info = &ramfs_backing_dev_info;
		mapping_set_gfp_mask(inode->i_mapping, GFP_HIGHUSER);
		mapping_set_unevictable(inode->i_mapping);
		inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
		switch (mode & S_IFMT) {
		default:
			init_special_inode(inode, mode, dev);
			break;
		case S_IFREG:
			inode->i_op = &ramfs_file_inode_operations;
			inode->i_fop = &ramfs_file_operations;
			break;
		case S_IFDIR:
			inode->i_op = &ramfs_dir_inode_operations;
			inode->i_fop = &simple_dir_operations;

			/* directory inodes start off with i_nlink == 2 (for "." entry) */
			inc_nlink(inode);
			break;
		case S_IFLNK:
			inode->i_op = &page_symlink_inode_operations;
			break;
		}
	} ? end if inode ?
	return inode;
} ? end ramfs_get_inode ?
```

(1: address space operations)

(2: check inode type)

(3a: device node)

(3b: regular file)

(3c: directory file)

(3: symbolic link)

(1) Set address-space operations

(2) S_IFMT is a bitmak to extract the file type code from the mode value

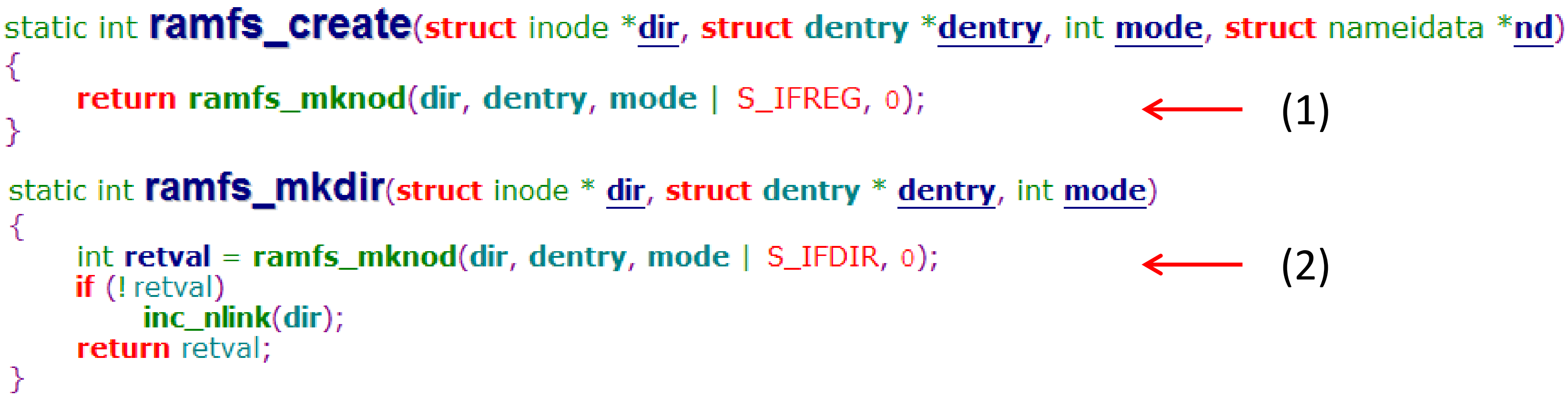
S_IFREG: regular file, S_IFDIR: directory, S_IFLNK: link

S_ISCHR, S_ISBLK, S_ISFIFO, S_ISSOCK: device node

(3) Set inode operations and file operations

ramfs_mknod() is called to create an inode for

- A regular file : ramfs_create()→ramfs_mknod(S_IFREG)
- A directory : ramfs_mkdir() → ramfs_mknod(S_IFDIR)
- A device node : ramfs_mknod() → ramfs_mknod(S_ISCHR, S_ISBLK, S_ISFIFO, or S_ISSOCK)



```
static int ramfs_create(struct inode *dir, struct dentry *dentry, int mode, struct nameidata *nd)
{
        return ramfs_mknod(dir, dentry, mode | S_IFREG, 0);
}

static int ramfs_mkdir(struct inode * dir, struct dentry * dentry, int mode)
{
        int retval = ramfs_mknod(dir, dentry, mode | S_IFDIR, 0);
        if (! retval)
                inc_nlink(dir);
        return retval;
}
```

(1)

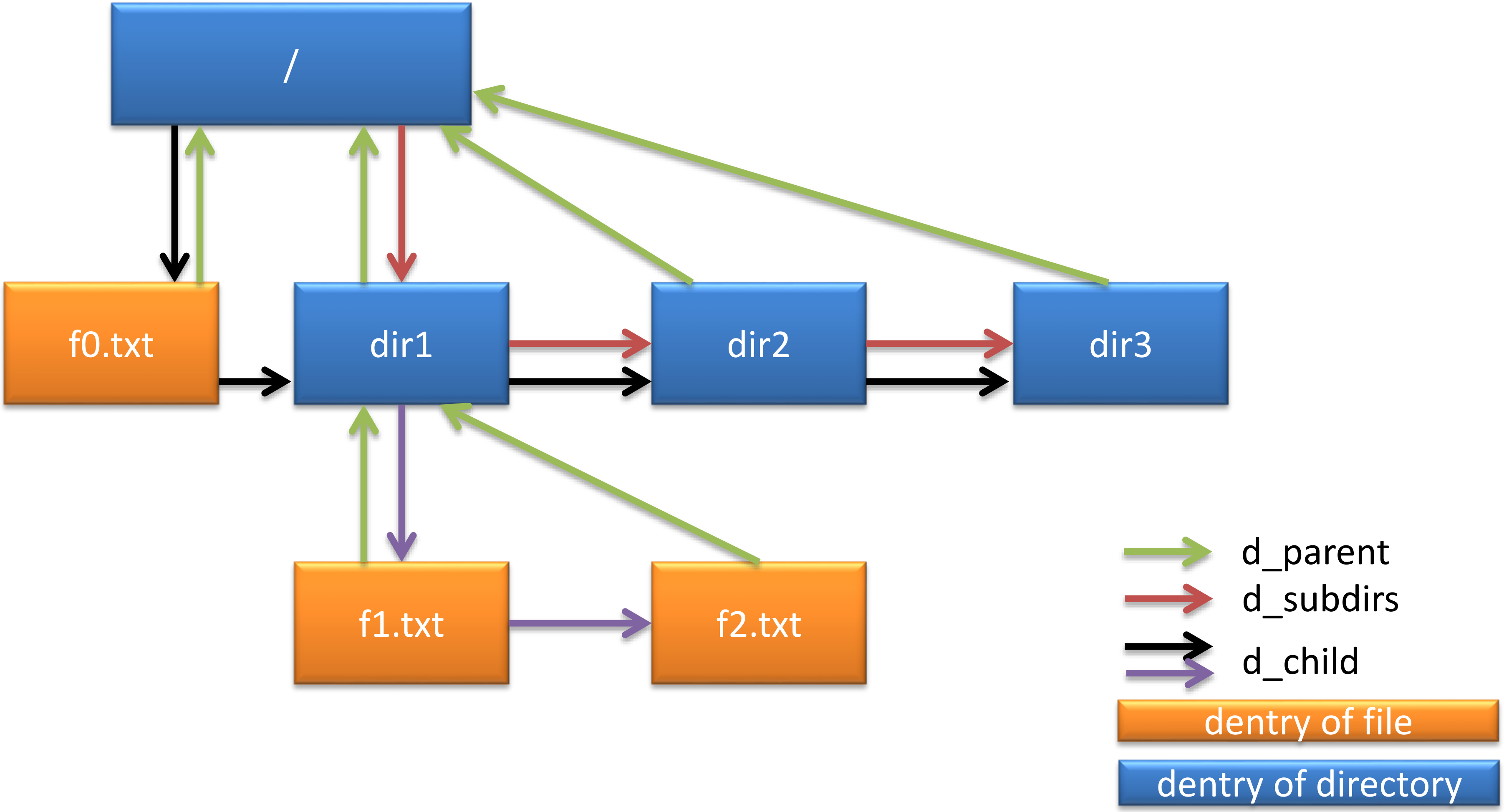
(2)

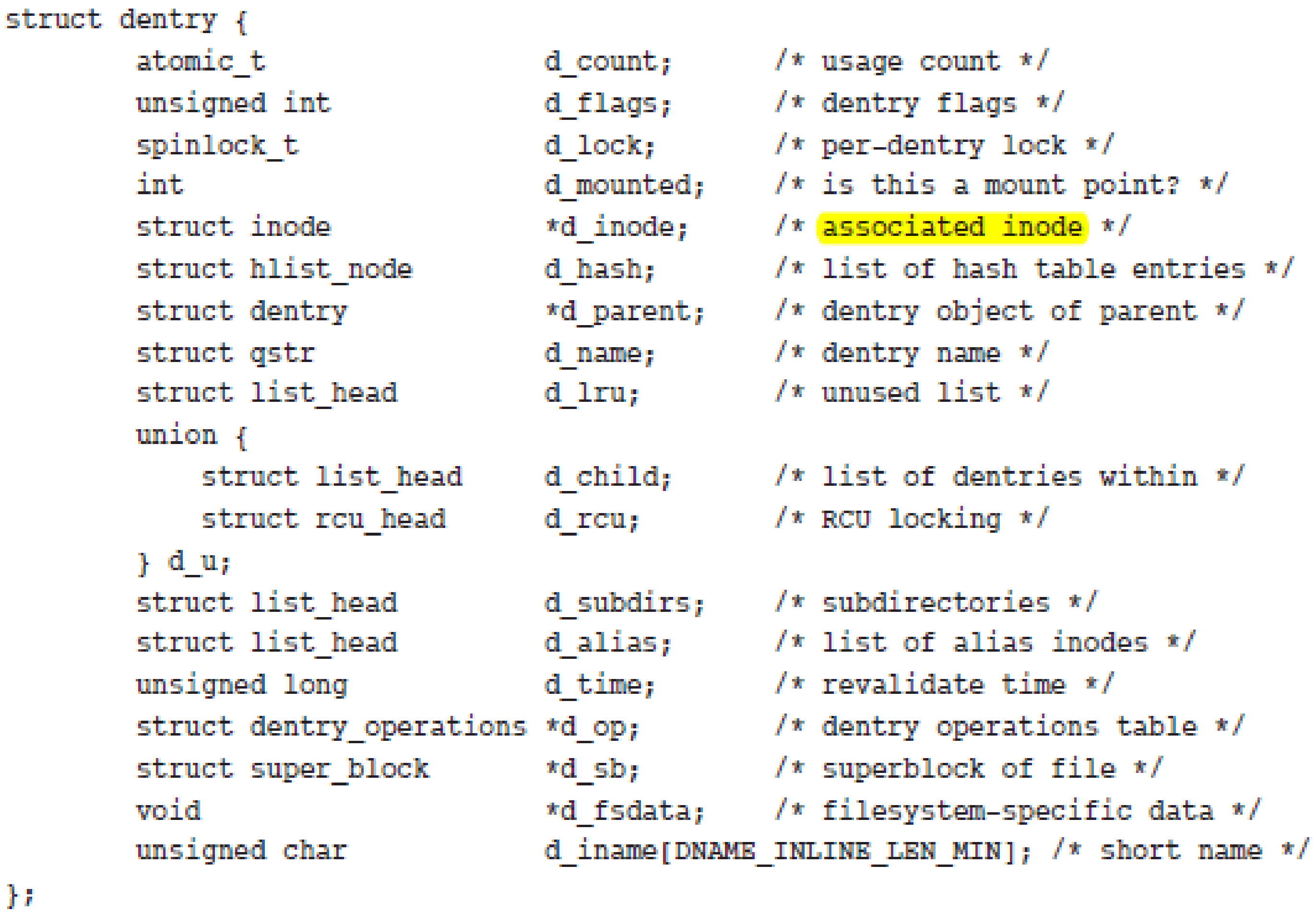
(1) Use S_IFREG to create an inode for a regular file
(2) Use S_IFDIR to create an inode for a directory

Dentry Objects

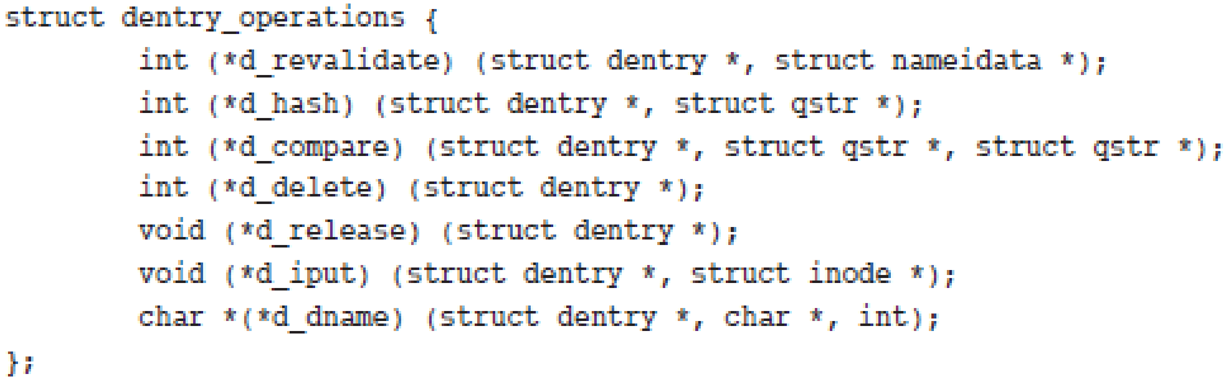
- Describing the relation among pathname components
  - /mnt/dir1/file1.txt
  - mnt, dir1, file1.txt all have an dentry object
- VFS manages dentry creation/deletion/lookup
  - A FS driver is responsible to filling correct values in dentry objects
- An dentry object has one of the following statuses
  - Free: contains no valid information
  - In use: *d_count* >0, currently used by FS (ramfs set it>=1)
  - Unused: *d_count* =0, no processes refer to the corresponding file
  - Negative: *d_count* <0, the corresponding file has been deleted or the specified file does not exist
- Unused dentry objects are cached so that re-opening a previously opened file can be fast
  - Use an LRU list to recycle unused dentry objects
  - Use a hash function to speed up cache searching

Relationship Among dentry Objects





```
struct dentry {
        atomic_t                 d_count;       /* usage count */
        unsigned int             d_flags;       /* dentry flags */
        spinlock_t               d_lock;        /* per-dentry lock */
        int                      d_mounted;     /* is this a mount point? */
        struct inode             *d_inode;      /* associated inode */
        struct hlist_node        d_hash;        /* list of hash table entries */
        struct dentry            *d_parent;     /* dentry object of parent */
        struct qstr              d_name;        /* dentry name */
        struct list_head         d_lru;         /* unused list */
        union {
            struct list_head     d_child;       /* list of dentries within */
            struct rcu_head      d_rcu;         /* RCU locking */
        } d_u;
        struct list_head         d_subdirs;     /* subdirectories */
        struct list_head         d_alias;       /* list of alias inodes */
        unsigned long            d_time;        /* revalidate time */
        struct dentry_operations *d_op;         /* dentry operations table */
        struct super_block       *d_sb;         /* superblock of file */
        void                     *d_fsdata;     /* filesystem-specific data */
        unsigned char            d_iname[DNAME_INLINE_LEN_MIN]; /* short name */
};
```



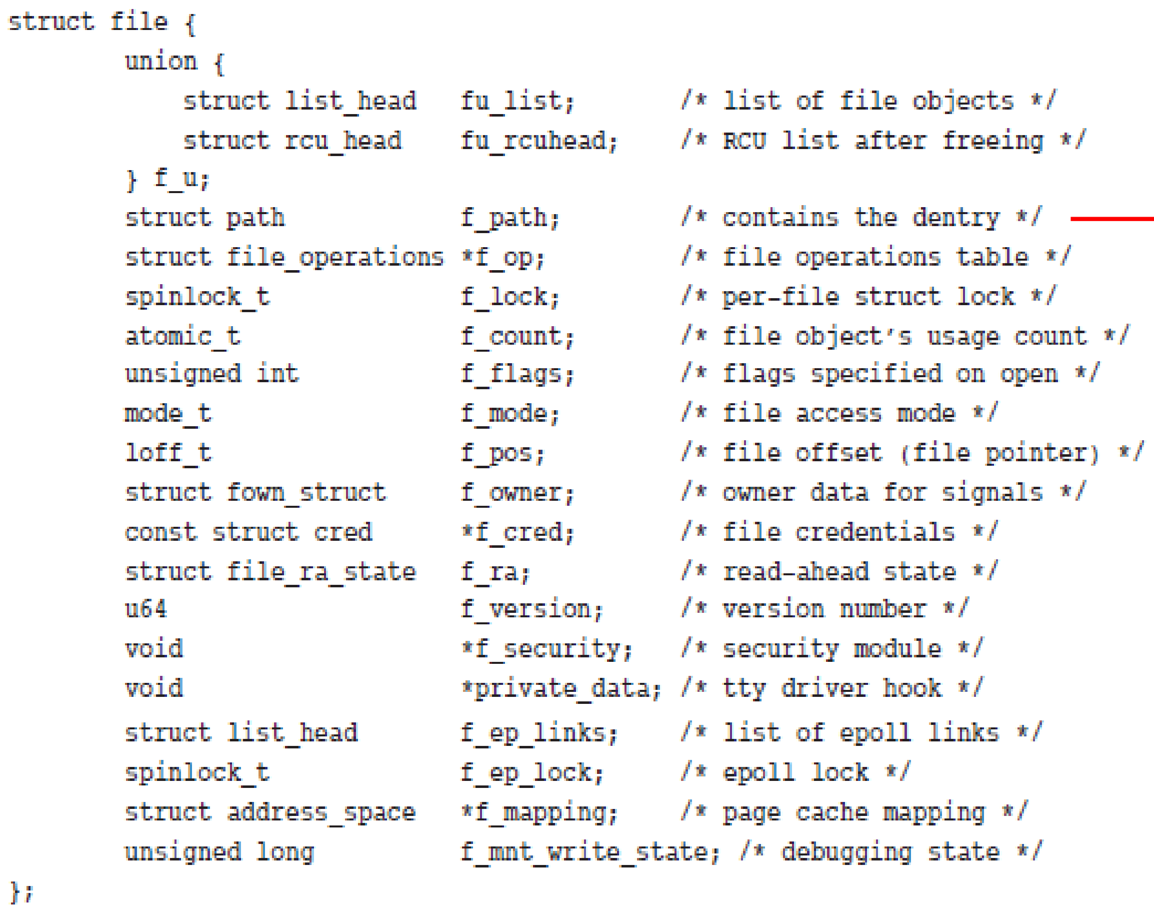
```
 struct dentry_operations {
        int (*d_revalidate) (struct dentry *, struct nameidata *);
        int (*d_hash) (struct dentry *, struct qstr *);
        int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
        int (*d_delete) (struct dentry *);
        void (*d_release) (struct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

Essential members

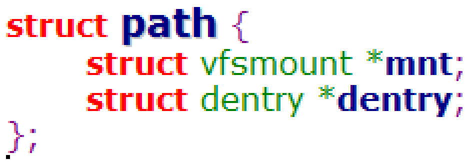
- d_count
- *d_inode
- *d_parent
- d_name (filename)
- d_lru
- d_child
- d_subdirs
- *d_sb

File Objects

- A file object is the representation of an opened file
  - A process's view of an opened file
    - Such as the current file position
    - If many processes open the same file, then there will be many file objects per process and only one dentry and one inode
  - Created upon open() and destroyed upon close()
  - Does not correspond to any disk structures
- Operations on file objects are related to *user data access*, such as read and write
  - Notice that operations on inodes are related *disk data structure (metadata) management*



```
struct file {
        union {
            struct list_head    fu_list;        /* list of file objects */
            struct rcu_head     fu_rcuhead;     /* RCU list after freeing */
        } f_u;
        struct path             f_path;         /* contains the dentry */
        struct file_operations *f_op;           /* file operations table */
        spinlock_t              f_lock;         /* per-file struct lock */
        atomic_t                f_count;        /* file object's usage count */
        unsigned int            f_flags;        /* flags specified on open */
        mode_t                  f_mode;         /* file access mode */
        loff_t                  f_pos;          /* file offset (file pointer) */
        struct fown_struct      f_owner;        /* owner data for signals */
        const struct cred      *f_cred;         /* file credentials */
        struct file_ra_state    f_ra;           /* read-ahead state */
        u64                     f_version;      /* version number */
        void                   *f_security;     /* security module */
        void                   *private_data;   /* tty driver hook */
        struct list_head        f_ep_links;     /* list of epoll links */
        spinlock_t              f_ep_lock;      /* epoll lock */
        struct address_space   *f_mapping;      /* page cache mapping */
        unsigned long           f_mnt_write_state; /* debugging state */
};
```



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

Essential members:

- f_path
  - points to dentry
- f_count
- f_pos
- f_ops
- f_mode
  - How the process opened this file for access, e.g., read, write, or exec (similar to i_mode)
- f_flags
  - Process open flags, such as O_SYNC, O_DIRECT…

```
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 (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
                             unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
                              unsigned long, 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 *, fl_owner_t id);
        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 (*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 (*flock) (struct file *, int, struct file_lock *);
         ssize_t (*splice_write) (struct pipe_inode_info *,
                                  struct file *,
                                  loff_t *,
                                  size_t,
                                  unsigned int);
         ssize_t (*splice_read) (struct file *,
                                 loff_t *,
                                 struct pipe_inode_info *,
                                 size_t,
                                 unsigned int);
         int (*setlease) (struct file *, long, struct file_lock **);
 };
```

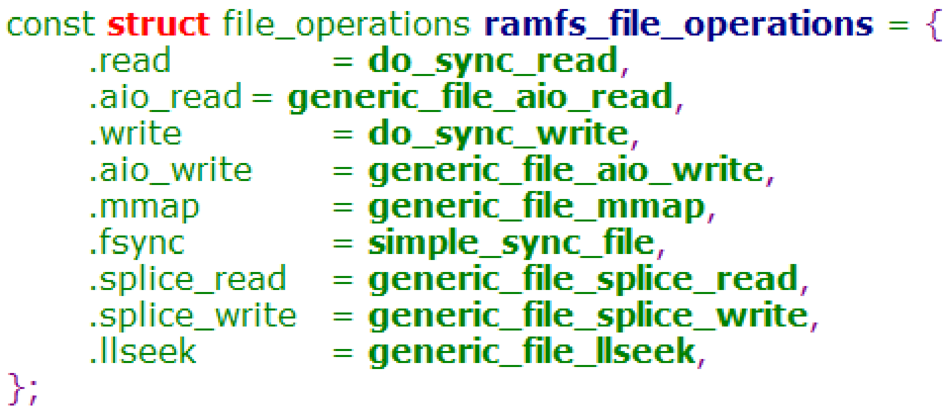
Operations on File Objects

- *llseek*(file, offset, origin)
  - Updates the file pointer
- *read*(file, buf, count, offset)
  - Reads count bytes from a file starting at position*offset; the value *offset is then increased
- *aio_read*(req, buf, len, pos)
  - Starts an asynchronous I/O operation to read *len* bytes into *buf* from file position *pos*
- *write*(file, buf, count, offset)
  - Writes count bytes into a file starting at position*offset; the value *offset is then increased
- *aio_write*(req, buf, len, pos)
  - Starts an asynchronous I/O operation to write *len* bytes from *buf* to file position *pos*

Operations on File Objects

- *ioctl*(inode, file, cmd, arg)
  - Sends a command to an underlying hardware device. This method applies only to device files
- *mmap*(file, vma)
  - Performs a memory mapping of the file into a process address space
- *open*(inode, file)
  - Opens a file by creating a new file object and linking it to the corresponding inode object
- *release*(inode, file)
  - Releases the file object. Called when the last reference to an open file is closed— that is, when the f_count field of the file object becomes 0
- *fsync*(file, dentry, flag)
  - Flushes the file by writing all cached data to disk.

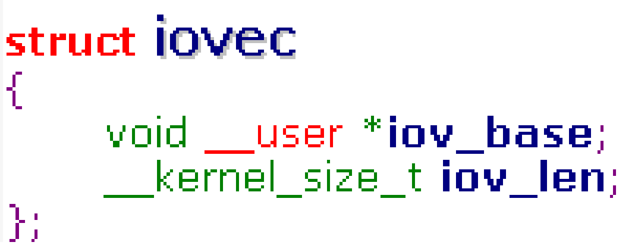
ramfs File Operations



```
const struct file_operations ramfs_file_operations = {
      .read             = do_sync_read,
      .aio_read = generic_file_aio_read,
      .write            = do_sync_write,
      .aio_write        = generic_file_aio_write,
      .mmap             = generic_file_mmap,
      .fsync            = simple_sync_file,
      .splice_read      = generic_file_splice_read,
      .splice_write     = generic_file_splice_write,
      .llseek           = generic_file_llseek,
};
```

- ramfs uses only the default handlers for file operations
  - These default handlers focus on the copy between user pages and kernel pages
  - How data are copied between kernel pages and disk blocks are implemented in *address-space operations*

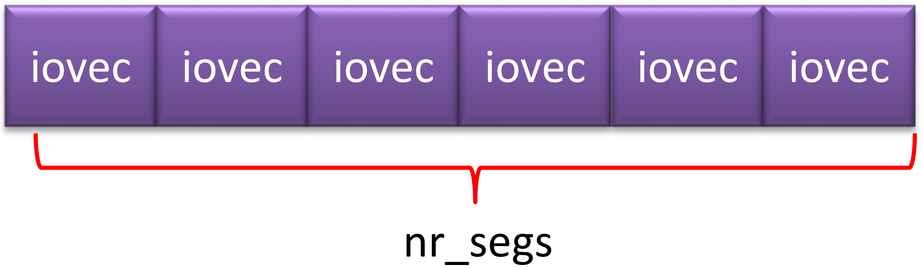
iovec



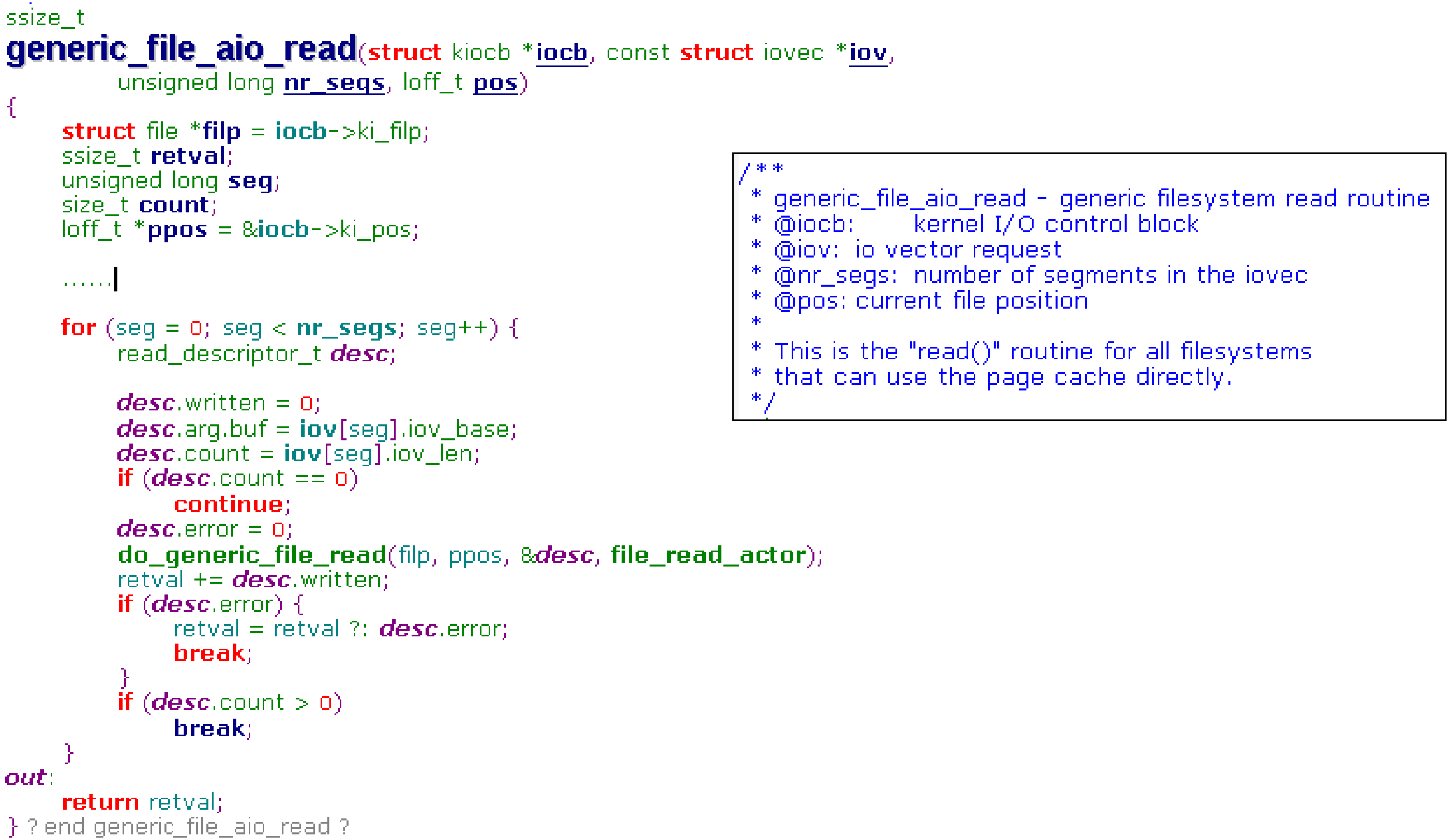
```
struct iovec
{
	void __user *iov_base;
	__kernel_size_t iov_len;
};
```

iov_base: a pointer to user-land buffer

Iov_len: I/O length in bytes



- *iovec, an vector of file system I/O operations
  - A pointer to an array of iovec
  - Array has nr_segs elements (iovec)
  - The buffer pointed by iov_base is accessible in file system drivers
    - No further kmap is required



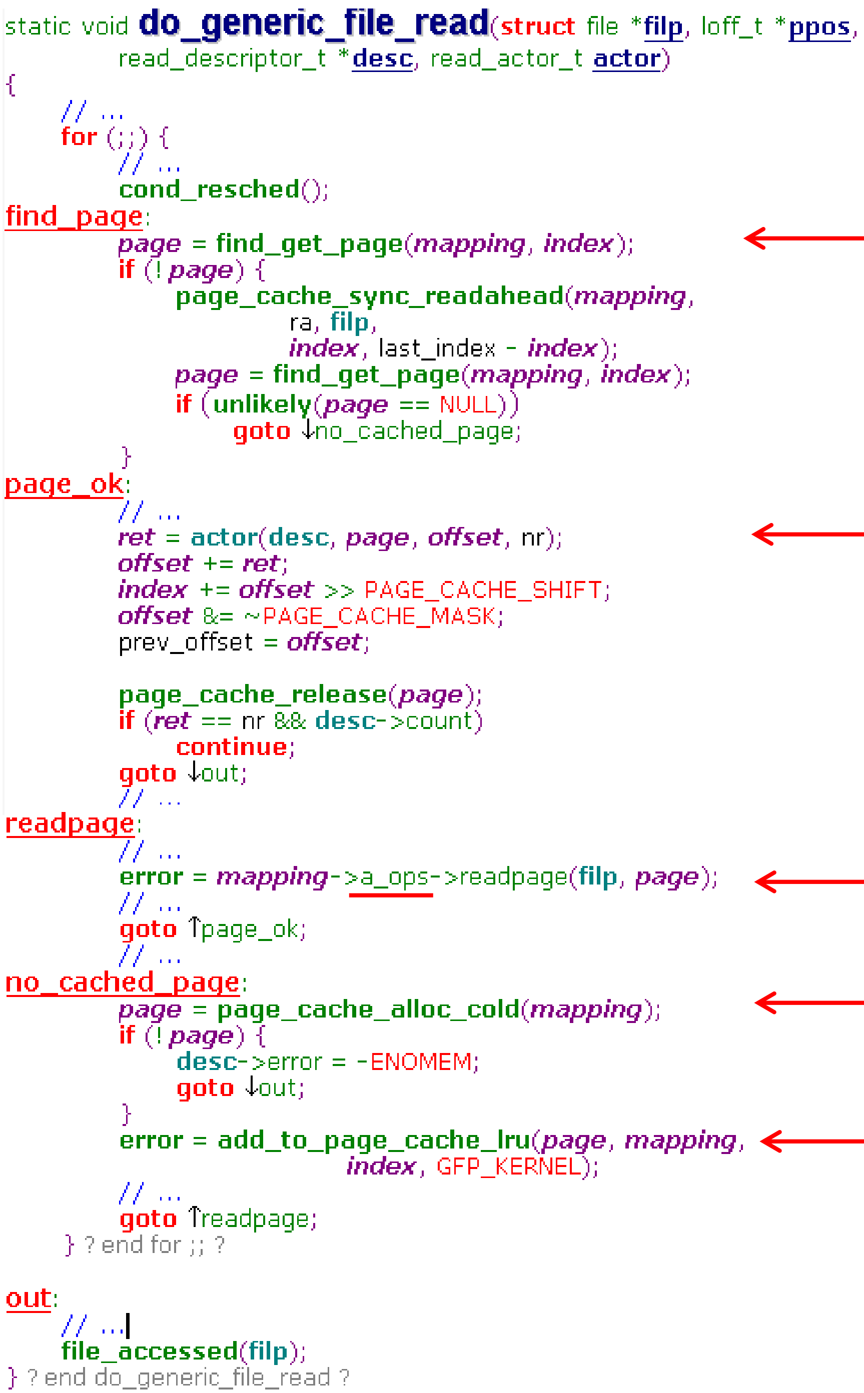
```
ssize_t
generic_file_aio_read(struct kiocb *iocb, const struct iovec *iov,
        unsigned long nr_segs, loff_t pos)
{
    struct file *filp = iocb->ki_filp;
    ssize_t retval;
    unsigned long seg;
    size_t count;
    loff_t *ppos = &iocb->ki_pos;

    ......

    for (seg = 0; seg < nr_segs; seg++) {
        read_descriptor_t desc;

        desc.written = 0;
        desc.arg.buf = iov[seg].iov_base;
        desc.count = iov[seg].iov_len;
        if (desc.count == 0)
            continue;
        desc.error = 0;
        do_generic_file_read(filp, ppos, &desc, file_read_actor);
        retval += desc.written;
        if (desc.error) {
            retval = retval ?: desc.error;
            break;
        }
        if (desc.count > 0)
            break;
    }
out:
    return retval;
} ? end generic_file_aio_read ?
```

```
/**
 * generic_file_aio_read - generic filesystem read routine
 * @iocb:      kernel I/O control block
 * @iov:  io vector request
 * @nr_segs:  number of segments in the iovec
 * @pos: current file position
 *
 * This is the "read()" routine for all filesystems
 * that can use the page cache directly.
 */
```

For each iovec in the iov[], convert the iovec into a “read_descriptor_t” and calls do_generic_file_read()

**notice that do_sync_read() is only a wrapper of generic_file_aio_read()

- Wait on each aio to compelte



```
static void do_generic_file_read(struct file *filp, loff_t *ppos,
        read_descriptor_t *desc, read_actor_t actor)
{
    // ...
    for (;;) {
        // ...
        cond_resched();
find_page:
        page = find_get_page(mapping, index);
        if (! page) {
            page_cache_sync_readahead(mapping,
                    ra, filp,
                    index, last_index - index);
            page = find_get_page(mapping, index);
            if (unlikely(page == NULL))
                goto ↓no_cached_page;
        }
page_ok:
        // ...
        ret = actor(desc, page, offset, nr);
        offset += ret;
        index += offset >> PAGE_CACHE_SHIFT;
        offset &= ~PAGE_CACHE_MASK;
        prev_offset = offset;

        page_cache_release(page);
        if (ret == nr && desc->count)
            continue;
        goto ↓out;
        // ...
readpage:
        // ...
        error = mapping->a_ops->readpage(filp, page);
        // ...
        goto ↑page_ok;
        // ...
no_cached_page:
        page = page_cache_alloc_cold(mapping);
        if (! page) {
            desc->error = -ENOMEM;
            goto ↓out;
        }
        error = add_to_page_cache_lru(page, mapping,
                    index, GFP_KERNEL);
        // ...
        goto ↑readpage;
    } ? end for ;; ?

out:
    // ...
    file_accessed(filp);
} ? end do_generic_file_read ?
```

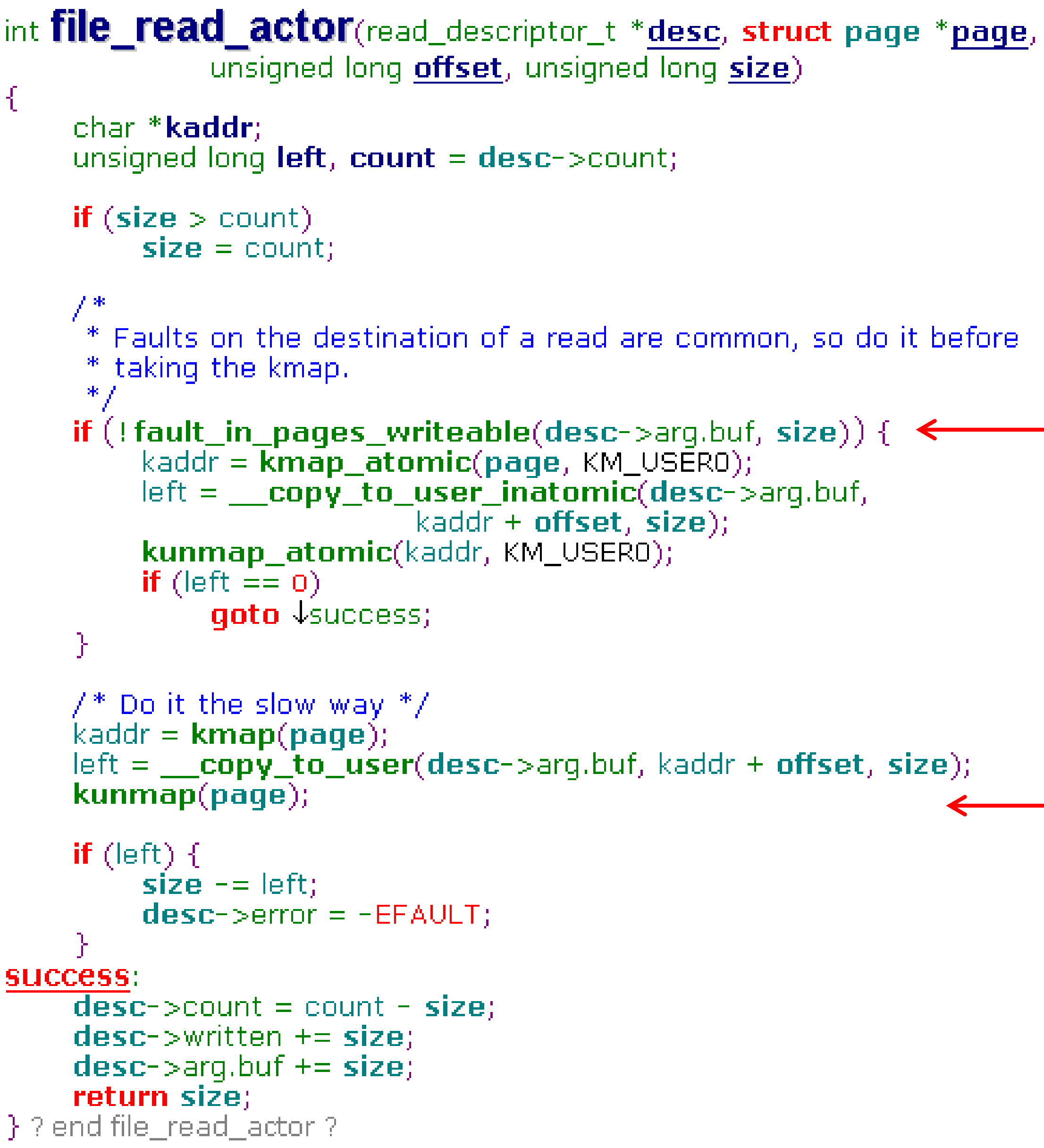
Check whether the desired page is already in page cache?

actor() is a pointer to file_read_actor(), which copies data from kernel pages to user buffer

Copy data to the new page. It will involve disk I/O in conventional file systems

Allocate a free page

Add the page to page cache



```
int file_read_actor(read_descriptor_t *desc, struct page *page,
                    unsigned long offset, unsigned long size)
{
        char *kaddr;
        unsigned long left, count = desc->count;

        if (size > count)
                size = count;

        /*
         * Faults on the destination of a read are common, so do it before
         * taking the kmap.
         */
        if (!fault_in_pages_writeable(desc->arg.buf, size)) {
                kaddr = kmap_atomic(page, KM_USER0);
                left = __copy_to_user_inatomic(desc->arg.buf,
                                                kaddr + offset, size);
                kunmap_atomic(kaddr, KM_USER0);
                if (left == 0)
                        goto ↓success;
        }

        /* Do it the slow way */
        kaddr = kmap(page);
        left = __copy_to_user(desc->arg.buf, kaddr + offset, size);
        kunmap(page);

        if (left) {
                size -= left;
                desc->error = -EFAULT;
        }
success:
        desc->count = count - size;
        desc->written += size;
        desc->arg.buf += size;
        return size;
} ? end file_read_actor ?
```

“fault in” the user buffer. If no page fault, do quick mapping for the kernel page and then copy data (hmm… everything is quick)

Already wasting a lot of time to fault in the user buffer. There is no need to use quick mapping. Do slow mapping for the kernel page and then copy data.

ramfs Write

- do_sync_write() is a wrapper of generic_file_aio_write()
  - Wait on each aio write to complete
  - Descending into generic_perform_write(), which does the following for each page
    - Calls a_op→begin_write() to prepare the next page to write
    - Copy data from user buffers pointed by iovec to the page just prepared
    - Calls a_op→end_write() to mark the page dirty

Address Space

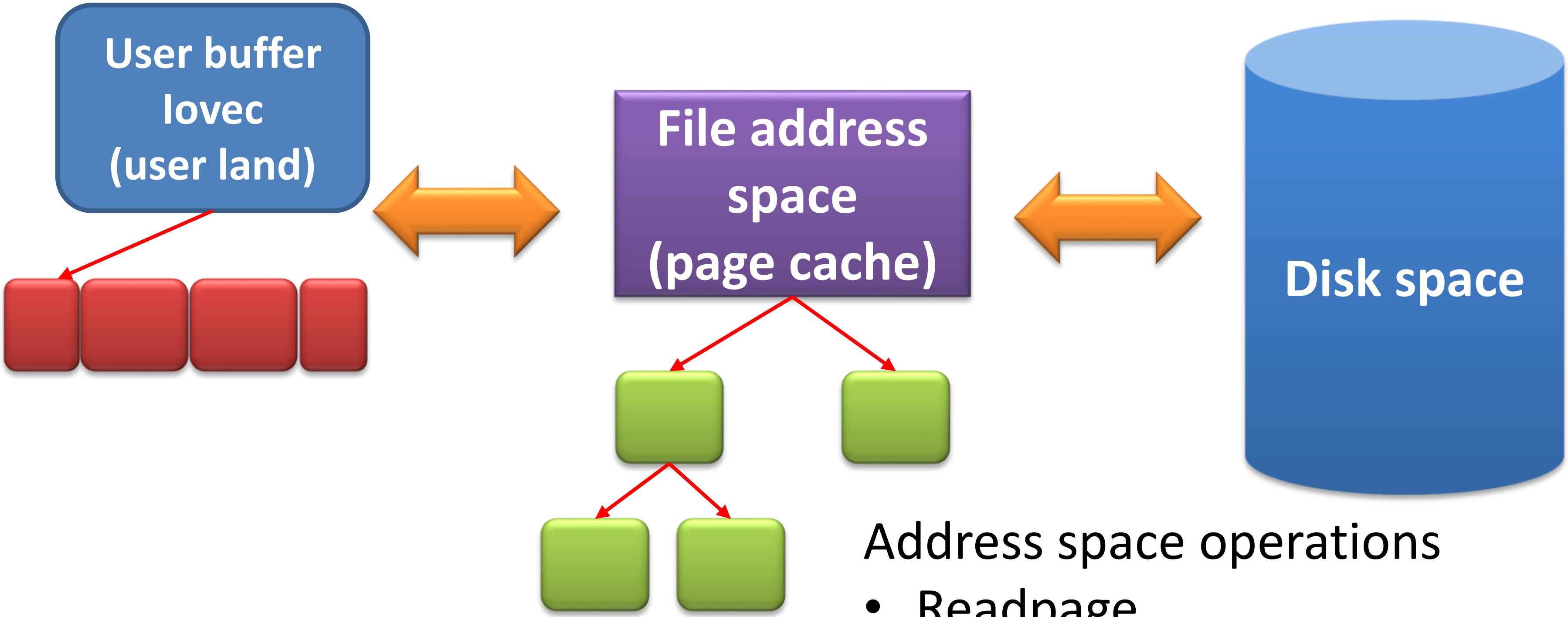
- “file address space” is more appropriate here
  - Not to be confused with process address space
  - A page owned by a process and is not mapped to any disk file is called an anonymous page
    - Pages of heap and stack
  - A page owned by a file is mapped to the file
    - Governed by the address space object “i_data” embedded in the file’s inode
- One of the good reasons to adopt address space is to enable caching for file systems not using block devices
  - Cached pages are indexed by (inode#, page offset), not (dev #, block offset)
  - ramfs, nfs, yaffs2, ubifs, etc.

Address Space

- A generic template of file operation
  - Modify VFS objects for the file operation
  - Calls address space operations
    - map pages to disk blocks, may involve metadata modification
    - Perform data copy
  - Mark modified pages dirty
- Thus, address space operations are to “map” and “copy” between pages and disk blocks

User buffer, address space, and disk space

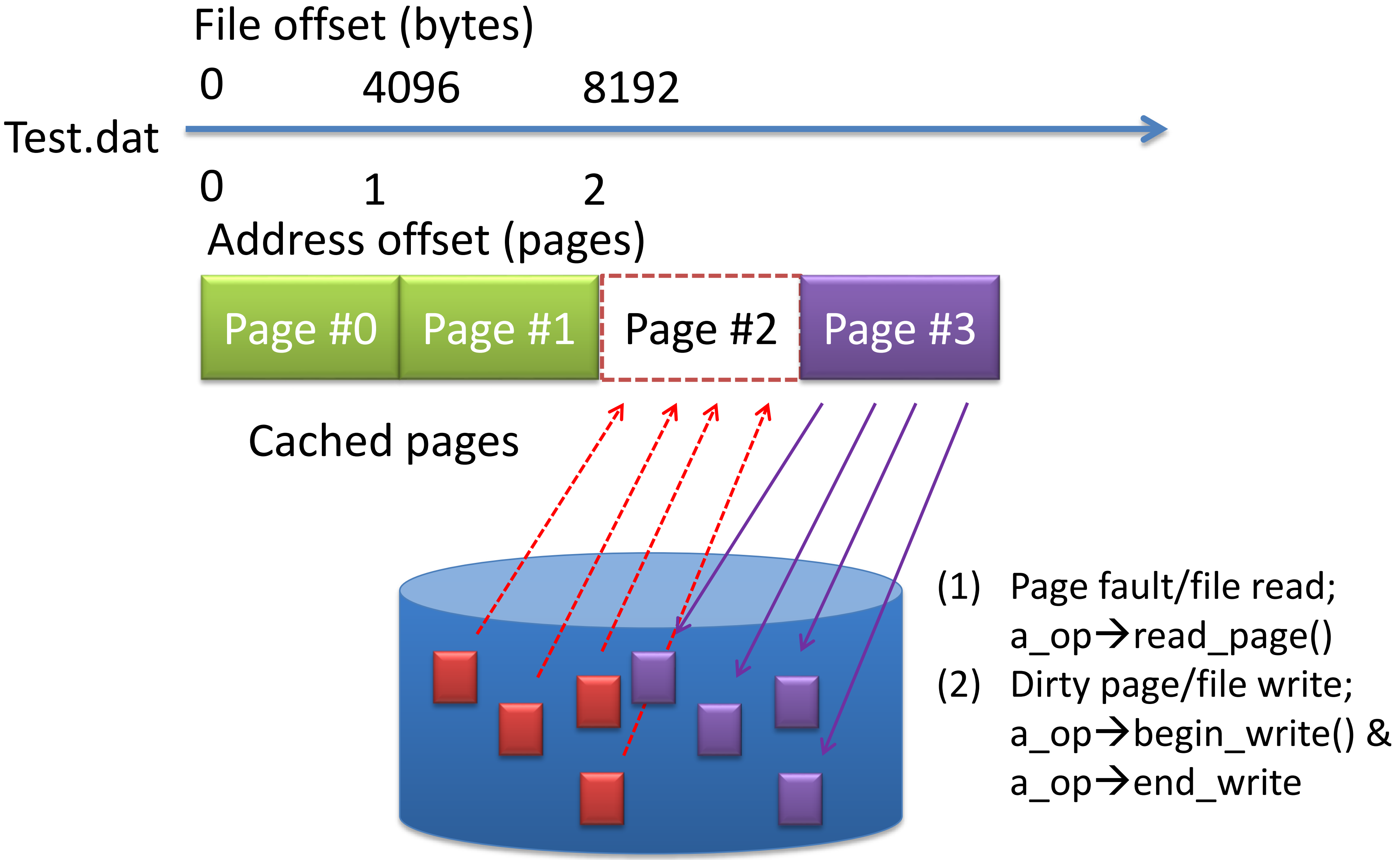
generic_file_aio_read()
generic_file_aio_write()



Address space operations

- Readpage
- Writepage
- begin_write()/end_write()

Address Space



File offset (bytes)

0 4096 8192

Test.dat

0 1 2

Address offset (pages)

Page #0 Page #1 Page #2 Page #3

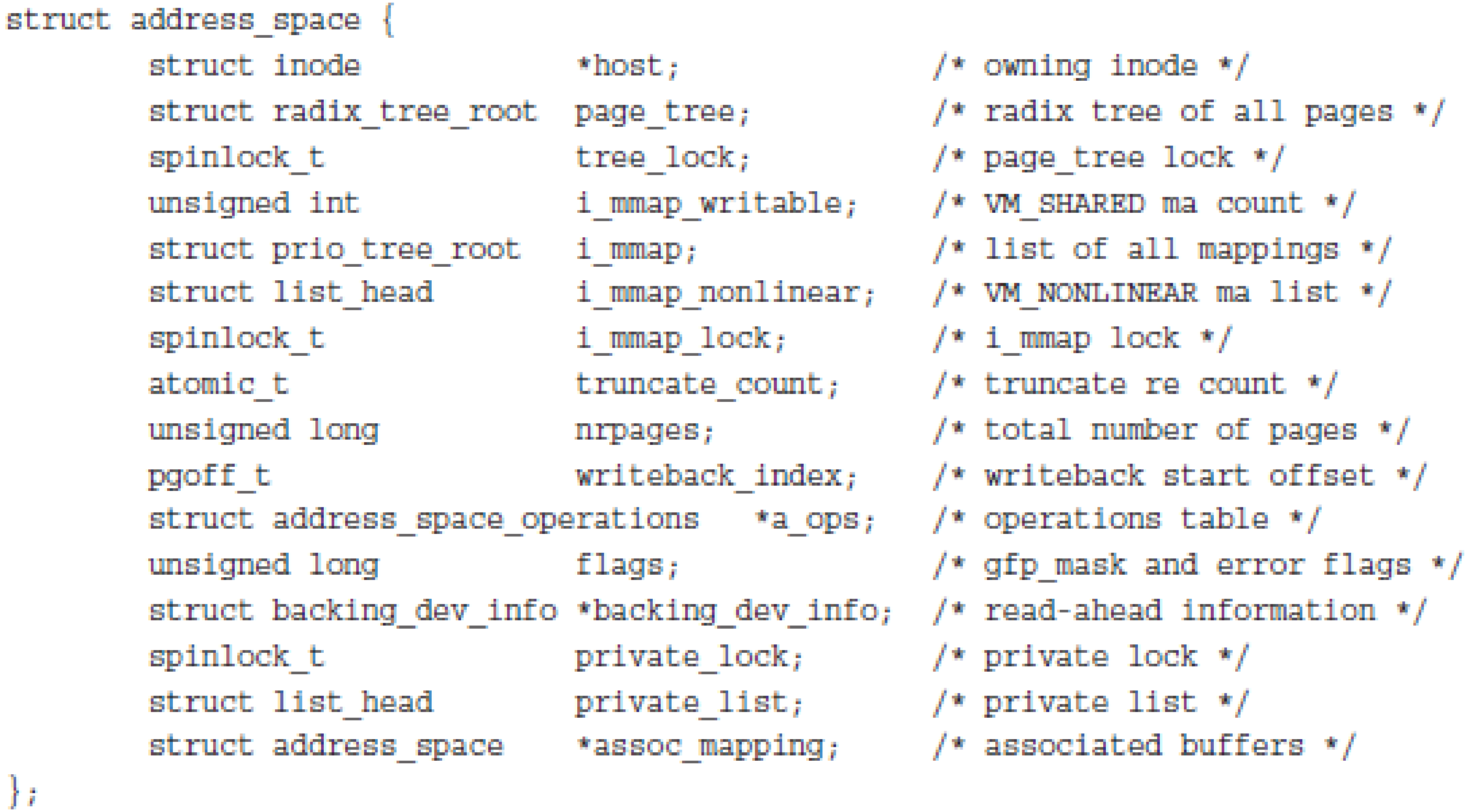
Cached pages

(1) Page fault/file read; a_op→read_page()

(2) Dirty page/file write; a_op→begin_write() & a_op→end_write

begin_write() and end_write()

- Part of file system implementation
- On file write, conventional disk file system will
  - – Calls a_op→begin_write()
    - Modify disk metadata to allocate disk space for new data
    - Prepare VM pages for new data
  - – Copy data from user buffer to VM pages
  - – Calls a_op→end_write() to mark the touched pages dirty



```
struct address_space {
        struct inode            *host;              /* owning inode */
        struct radix_tree_root  page_tree;          /* radix tree of all pages */
        spinlock_t              tree_lock;          /* page_tree lock */
        unsigned int            i_mmap_writable;    /* VM_SHARED ma count */
        struct prio_tree_root   i_mmap;             /* list of all mappings */
        struct list_head        i_mmap_nonlinear;   /* VM_NONLINEAR ma list */
        spinlock_t              i_mmap_lock;        /* i_mmap lock */
        atomic_t                truncate_count;     /* truncate re count */
        unsigned long           nrpages;            /* total number of pages */
        pgoff_t                 writeback_index;    /* writeback start offset */
        struct address_space_operations   *a_ops;   /* operations table */
        unsigned long           flags;              /* gfp_mask and error flags */
        struct backing_dev_info *backing_dev_info;  /* read-ahead information */
        spinlock_t              private_lock;       /* private lock */
        struct list_head        private_list;       /* private list */
        struct address_space    *assoc_mapping;     /* associated buffers */
};
```

Essential members:

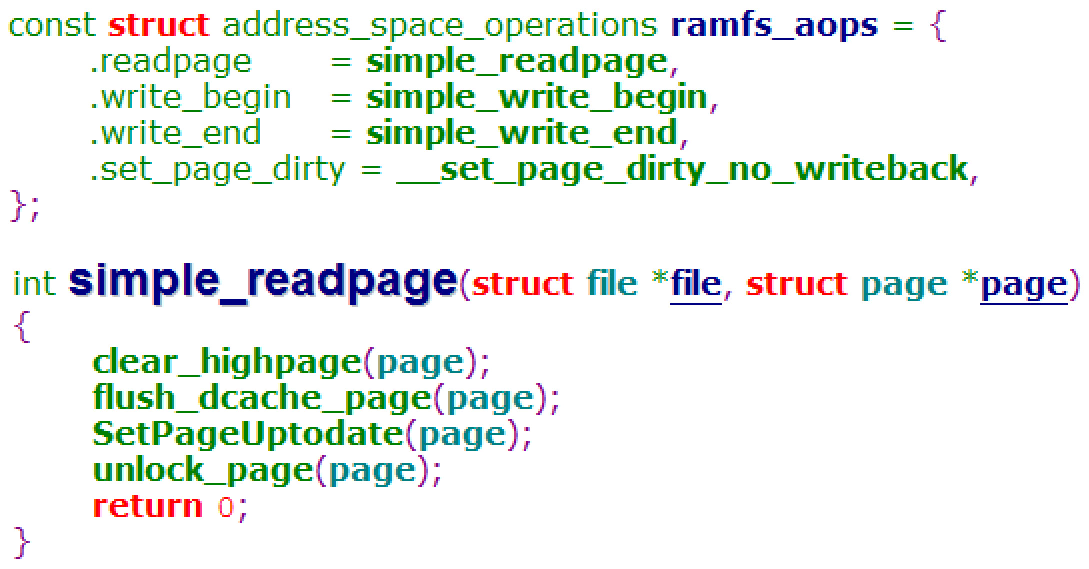
- *host
- page_tree (like the radix tree in ramdisk)
- *a_ops

```
struct address_space_operations {
        int (*writepage)(struct page *, struct writeback_control *);
        int (*readpage) (struct file *, struct page *);
        int (*sync_page) (struct page *);
        int (*writepages) (struct address_space *,
                           struct writeback_control *);
        int (*set_page_dirty) (struct page *);
        int (*readpages) (struct file *, struct address_space *,
                          struct list_head *, unsigned);
        int (*write_begin)(struct file *, struct address_space *mapping,
                           loff_t pos, unsigned len, unsigned flags,
                           struct page **pagep, void **fsdata);
        int (*write_end)(struct file *, struct address_space *mapping,
                         loff_t pos, unsigned len, unsigned copied,
                         struct page *page, void *fsdata);
        sector_t (*bmap) (struct address_space *, sector_t);
        int (*invalidatepage) (struct page *, unsigned long);
        int (*releasepage) (struct page *, int);
        int (*direct_IO) (int, struct kiocb *, const struct iovec *,
                          loff_t, unsigned long);
        int (*get_xip_mem) (struct address_space *, pgoff_t, int,
                            void **, unsigned long *);
        int (*migratepage) (struct address_space *,
                            struct page *, struct page *);
        int (*launder_page) (struct page *);
        int (*is_partially_uptodate) (struct page *,
                                      read_descriptor_t *,
                                      unsigned long);
        int (*error_remove_page) (struct address_space *,
                                  struct page *);
};
```

Address Space Operations

- writepage()
  - Write operation (from the page to the owner’s disk image)
- readpage()
  - Read operation (from the owner’s disk image to the page)
- prepare_write() or write_begin()
  - Prepare a write operation
  - Usually involves metadata modification for disk-based file systems
- commit_write() or write_end()
  - Complete a write operation
  - Mark the modified pages dirty

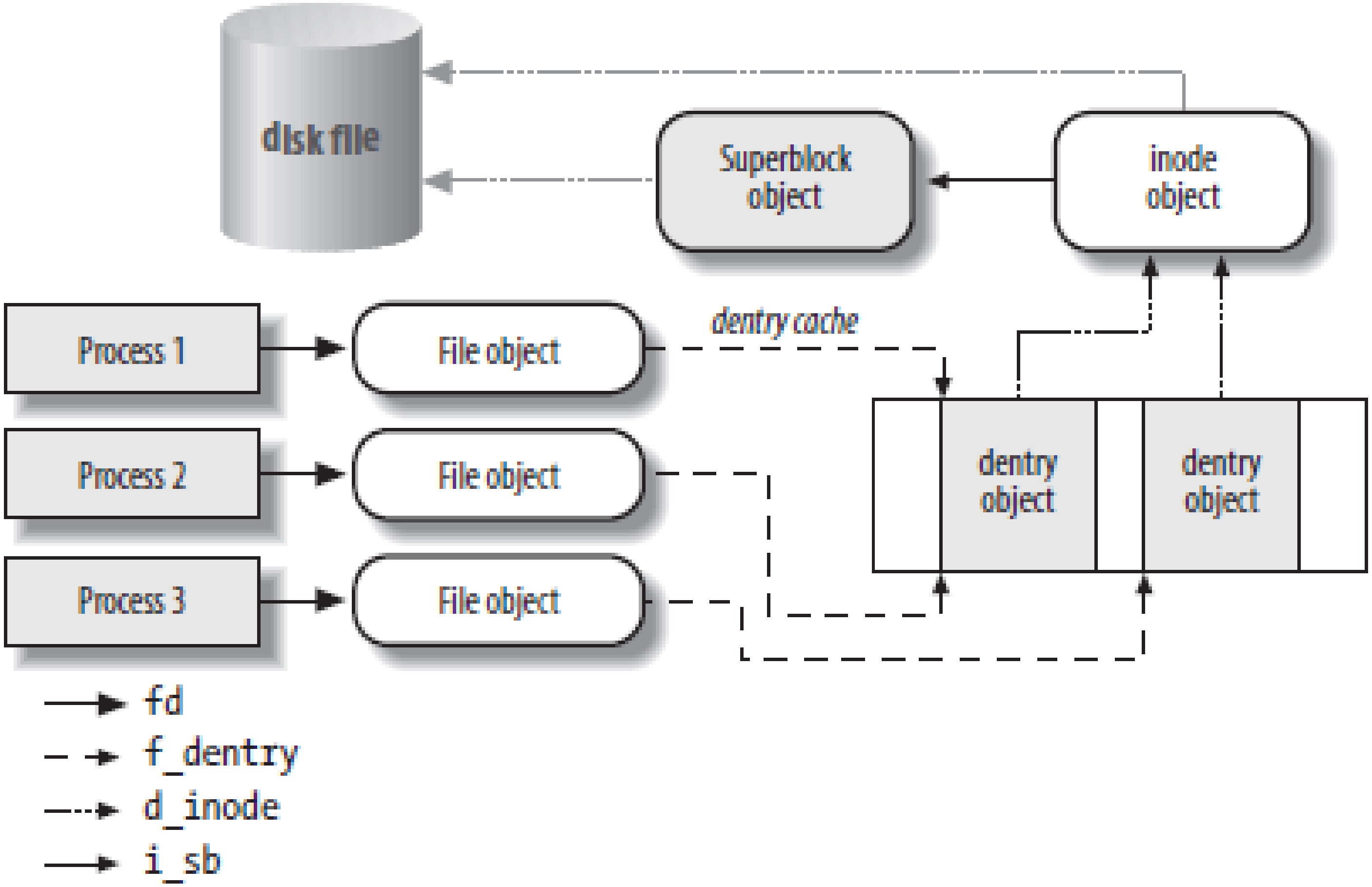
ramfs Address Space Operations



```
const struct address_space_operations ramfs_aops = {
        .readpage       = simple_readpage,
        .write_begin    = simple_write_begin,
        .write_end      = simple_write_end,
        .set_page_dirty = __set_page_dirty_no_writeback,
};

int simple_readpage(struct file *file, struct page *page)
{
        clear_highpage(page);
        flush_dcache_page(page);
        SetPageUptodate(page);
        unlock_page(page);
        return 0;
}
```

- ramfs does nothing useful in address space operations because there is no need to copy/map data between cached pages and disk blocks

Review


disk file
Superblock object
inode object
Process 1
Process 2
Process 3
File object
File object
File object
dentry cache
dentry object
dentry object
fd
f_dentry
d_inode
i_sb

Lab 10: Simple File Encryption

File Encryption

- Implement XOR-based file encryption in ramfs
- Files are encrypted and decrypted by XOR'ing file bytes with a key
- If encryption is on
  - XOR bytes after read
  - XOR byes before write
- By turning off encryption, you get garbage when reading an encrypted file

/proc file system

- An alternative way to communicate with kernel code
  - File system drivers are not “devices”, and they cannot accept ioctl as ramdisks do
- Register an /proc file entry for user-land access, as well as the entry’s read/write handlers
  - The handlers will be called when user programs access the registered /proc entry
- Use this feature to implement turning on/off encryption
  - cat 1 > /proc/ramfs_flag

ramfs Modification

- You need to register a /proc file so that you can turn on/off encryption in user space

- You need to register your own file operation callbacks to process encryption/decryption

References

- Robert love, “Linux Kernel Development 3rd Edition,” 2010
- Daniel B. Bovet et al., “Understanding the Linux Kernel 3rd Edition,” 2005
- Linux kernel source tree 2.6.34