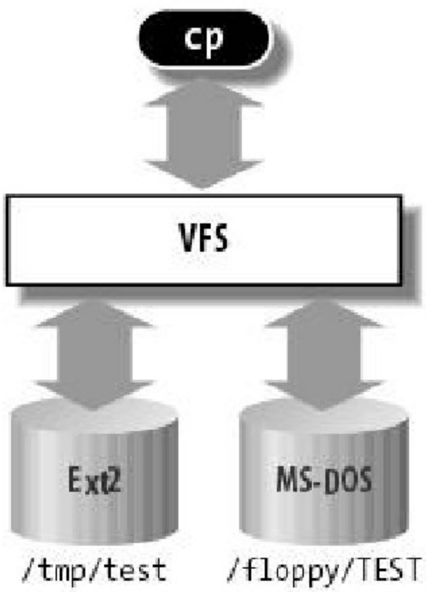
VFS and extended file systems

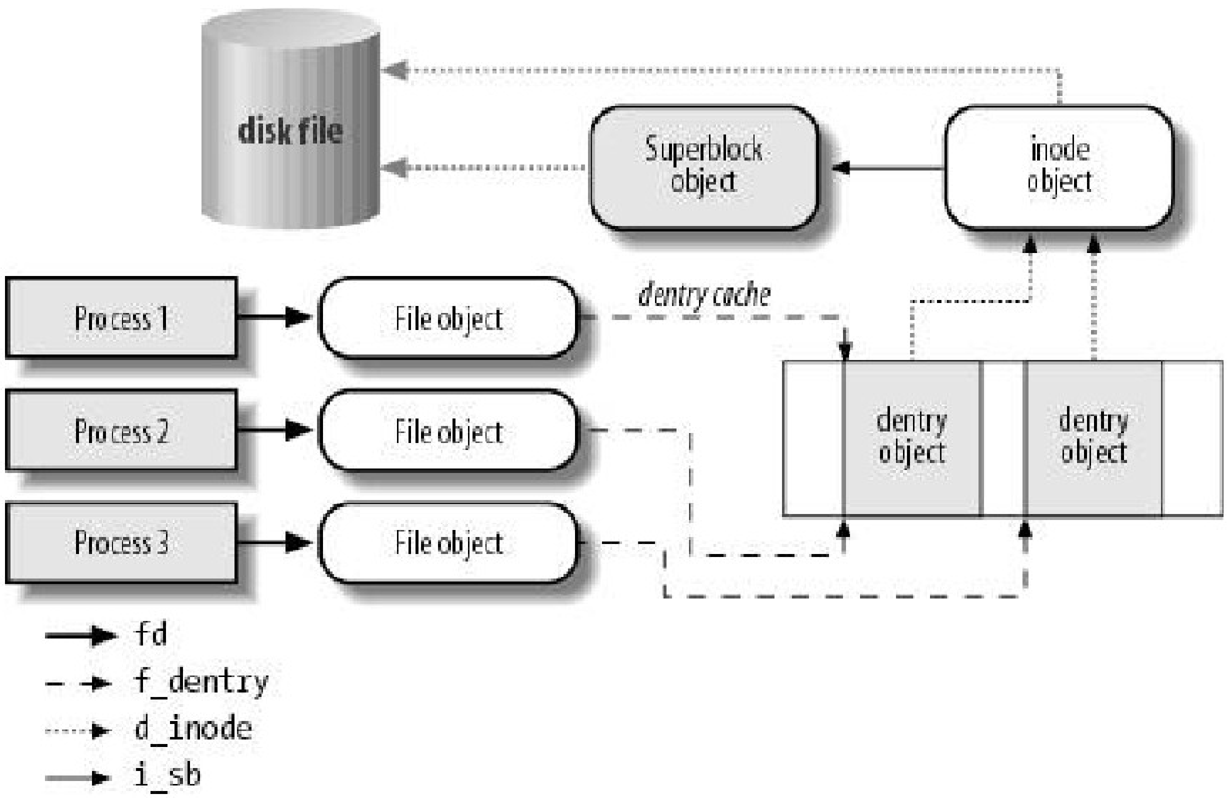
Illustration of interaction between VFS and file-managers




```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

----------------intentionally left blank--------------------

Illustration of interaction between processes and VFS



---------------------intentionally left blank----------------------

important data structures

- when working with files, the central objects differ in kernel space and userspace.

- for user programs, a file is identified by a file descriptor. This is an integer number used as a parameter to identify the file in all file-related operations. The file descriptor is assigned by the kernel when a file is opened and is valid only within a process.

- two different processes may therefore use the same file descriptor, but it does not point to the same file in both cases. shared use of files on the basis of the same descriptor number is not possible.

- the i-node is key to the kernel's work with files. Each file (and each directory) has just one i-node, which contains meta-data such as access rights, date of last change, and so on, and also pointers to the file data.

- however, and this may appear to be slightly strange, the inode does not contain one important item of information — the filename.

- usually, it is assumed that the name of a file is one of its major characteristics and should therefore be included in the object (inode) used to manage it. it is not so

- how can directory hierarchies be represented by data structures?

- as already noted, inodes are central to file implementation, but are also used to implement directories

- in other words, directories are just a special kind of file and must be interpreted correctly

- the elements of an inode can be grouped into two classes:
    - meta-data to describe the file status; for example, access permissions or date of last change
    - a data segments (or a pointers to data) in which the actual file contents are held

----------------intentionally left blank--------------

Directory hierarchies

- to demonstrate how inodes are used to structure the directory hierarchy of the filesystem, let's look at how the kernel goes about finding the inode of /usr/bin/vi

- lookup starts at the root inode, which represents the root directory / and must always be known to the system.

- the directory is represented by an inode whose data segment does not contain normal data but only the directory entries. these entries may stand for files or other directories. each entry consists of two elements.
    - The number of the inode in which the data of the next entry are located
    -. The name of the file or directory

- all inodes of the file-system have a specific number by which they are uniquely identified. The association between filename and inode is established by this number.

Illustration of a pathname look-up using directories

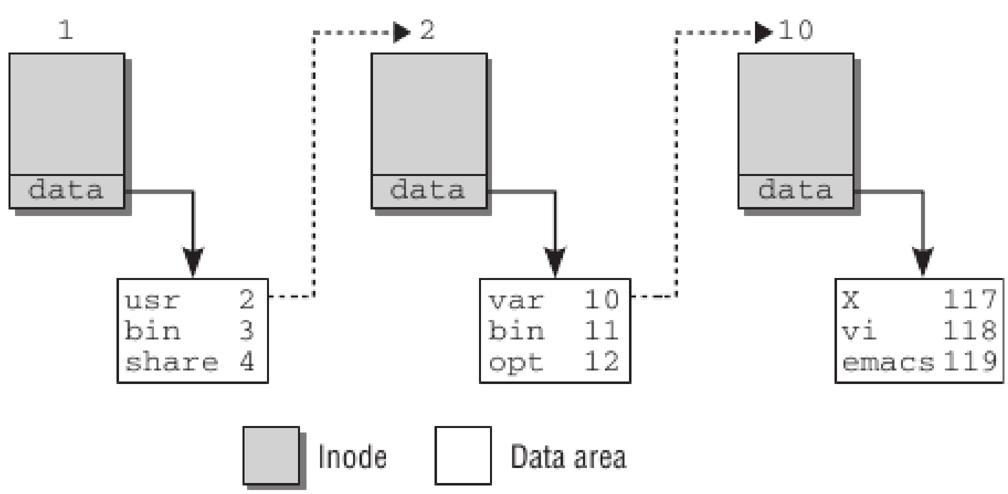
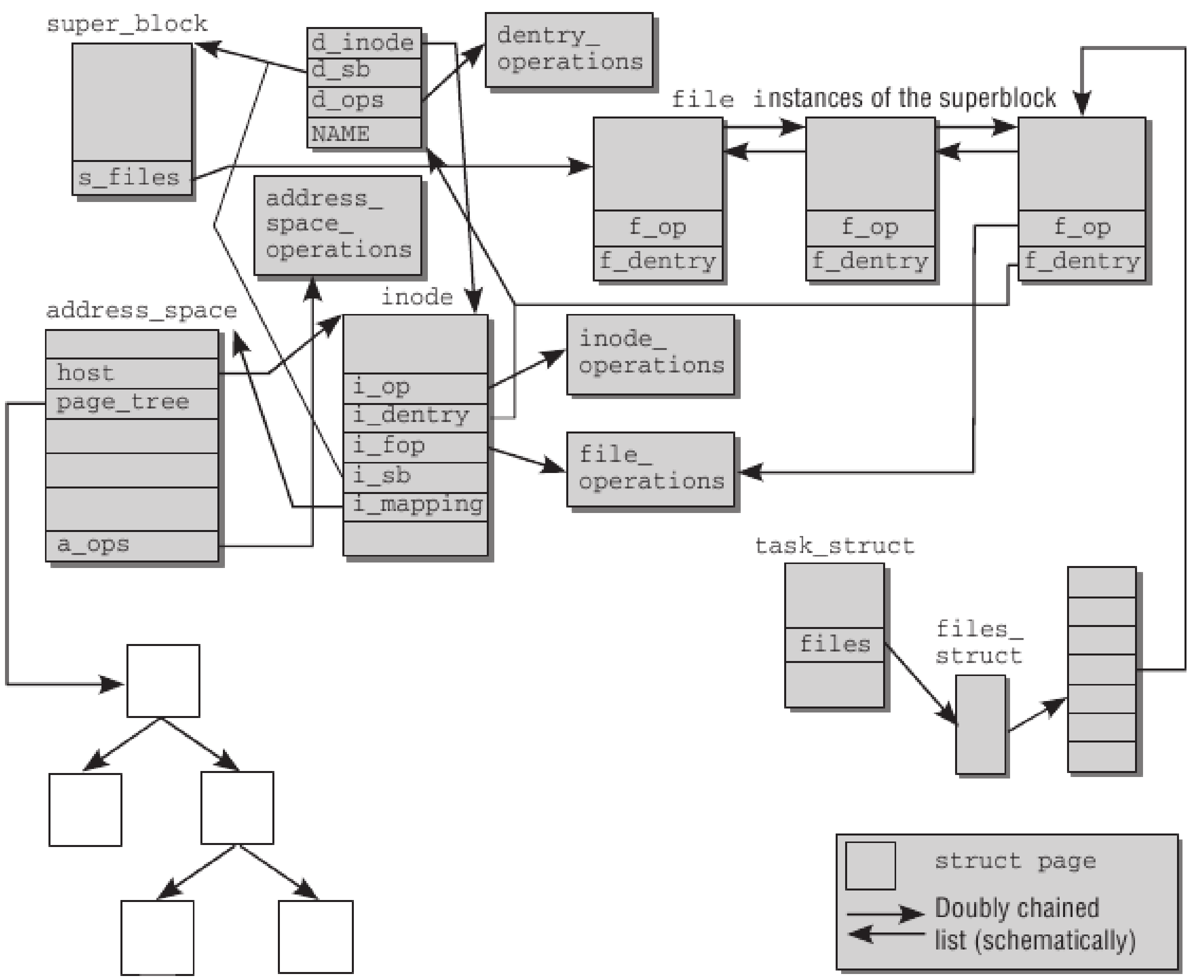


Illustration of the key data-str*u*ctures in VFS



VFS inode data- structure (in-memory)

<linux/fs.h>

```
struct inode {

        struct hlist_node       i_hash;
        struct list_head        i_list;
        struct list_head        i_sb_list;
        struct list_head        i_dentry;
        unsigned long           i_ino;
        atomic_t                i_count;
        unsigned int            i_nlink;
        uid_t                   i_uid;
        gid_t                   i_gid;
        dev_t                   i_rdev;
        unsigned long           i_version;
        loff_t                  i_size;
        struct timespec         i_atime;
        struct timespec         i_mtime;
        struct timespec         i_ctime;
        unsigned int            i_blkbits;
        blkcnt_t                i_blocks;
        umode_t                 i_mode;
        struct inode_operations *i_op;
        const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
        struct super_block      *i_sb;
        struct address_space    *i_mapping;
        struct address_space    i_data;

       ................
        void  * privatedata; //contains on-disk information of the file-system meta-data

       ................
```

-------------intentionally left blank------------------

```
struct dquot          *i_dquot[MAXQUOTAS];
struct list_head       i_devices;
union {
      struct pipe_inode_info *i_pipe;  //used in the case of a pipe-object
      struct block_device *i_bdev;
      struct cdev *i_cdev;              //used in the case of a device-file
};
int                  i_cindex;
__u32                 i_generation;
unsigned long          i_state;
unsigned long          dirtied_when;   /* jiffies of first dirtying */
unsigned int          i_flags;
atomic_t              i_writecount;
void                 *i_security;

....................

}
```

- the in-memory inode is maintained in several lists/hash-lists for efficient access

Inode operations

```
<fs.h>
struct inode_operations {
        int (*create) (struct inode *,struct dentry *,int, struct nameid
        struct dentry * (*lookup) (struct inode *,struct dentry *, struc
        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 *);

        ..................


}
```

- not all inode-operations defined for all the inodes – it depends on the type of file it represents

Important fields related to VFS in a process descriptor

```
< linux / sched.h>

struct task_struct {
...
/* file system info */
        int link_count, total_link_count;
        ...
/* filesystem information */
        struct fs_struct *fs;
/* open file information */
        struct files_struct *files;
/* namespaces */
        struct nsproxy *nsproxy;
...
}
```

```
<linux/sched.h>
struct files_struct {
        atomic_t count;
        struct fdtable *fdt;

        struct fdtable fdtab;
        int next_fd;
        struct embedded_fd_set close_on_exec_init;
        struct embedded_fd_set open_fds_init;
        struct file * fd_array[NR_OPEN_DEFAULT];  //open file-table
};
```

------------------intentionally left blank----------------------

- system-wide open file table (used in many places in the system architecture)

<linux/fs.h>

```
struct file
	struct list_head	fu_list;
	struct path f_path;
#define f_dentry f_path.dentry

#define f_vfsmnt f_path.mnt
	const struct file_operations	*f_op;
	atomic_t		f_count;
	unsigned int		f_flags;
	mode_t			f_mode;
	loff_t			f_pos;
	struct fown_struct	f_owner;
	unsigned int		f_uid, f_gid;
	struct file_ra_state	f_ra;
	unsigned long		f_version;
...
	struct address_space	*f_mapping;
...
};
```

----------------------intentionally left blank----------------------

- file operations of a given open file (will differ based on the type of file/file-system)

```
<linux/fs.h>
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 (*dir_notify)(struct file *filp, unsigned long arg);
	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);
};
```

------------------intentionally left blank------------------

- per task directory-related structure

<fs_struct.h>

```
struct fs_struct {
        atomic_t count;
        int umask;
        struct dentry * root, * pwd, * altroot;
        struct vfsmount * rootmnt, * pwdmnt, * altrootmnt;
};
```

- dentry cache – used to speed-up the path-name translation of recently accessed pathnames

<linux / dcache.h>

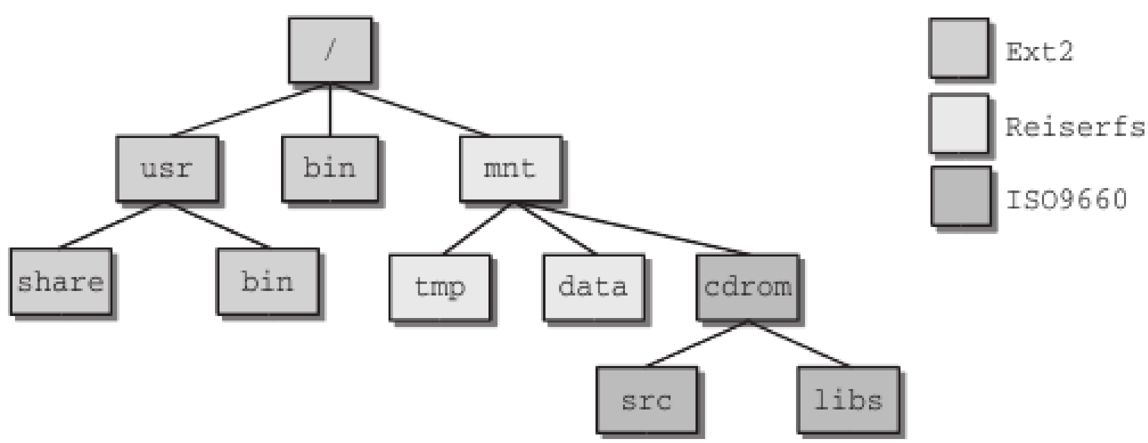
```
struct dentry {
       atomic_t d_count;
       unsigned int d_flags;          /* protected by d_lock */
       spinlock_t d_lock;             /* per dentry lock */
       struct inode *d_inode;         /* Where the name belongs to - NULL is
                                       * negative */
       /*
         * The next three fields are touched by __d_lookup. Place them here
         * so they all fit in a cache line.
         */
       struct hlist_node d_hash;      /* lookup hash list */
       struct dentry *d_parent;       /* parent directory */
       struct qstr d_name;
       struct list_head d_lru;        /* LRU list */
       union {
               struct list_head d_child;      /* child of parent list */
               struct rcu_head d_rcu;
       } d_u;
       struct list_head d_subdirs;    /* our children */
       struct list_head d_alias;      /* inode alias list */
       unsigned long d_time;          /* used by d_revalidate */
       struct dentry_operations *d_op;
       struct super_block *d_sb;      /* The root of the dentry tree */
       void *d_fsdata;                /* fs-specific data */
       unsigned char d_iname[ ]; // file name
       int d_mounted ;  //set to 1, if this directory is a mount-point
}
```

- per file-system type object – may be used mounting the file-system

```
<fs.h>
struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block *(*get_sb) (struct file_system_type *, int,
                                const char *, void *, struct vfsmount *);
        void (*kill_sb) (struct super_block *);
        struct module *owner;
        struct file_system_type * next;
        struct list_head fs_supers;
};
```

Mounting / unmounting

Illustration of a single file-system hierarchy



- in each mounted file-system in the system is represented by a struct vfsmount{}

```
<linux / mount.h>
struct vfsmount {
        struct list_head mnt_hash;
        struct vfsmount *mnt_parent; /* fs we are mounted on */
        struct dentry *mnt_mountpoint; /* dentry of mountpoint */
        struct dentry *mnt_root; /* root of the mounted tree */
        struct super_block *mnt_sb; /* pointer to superblock */
        struct list_head mnt_mounts; /* list of children, anchored here */
        struct list_head mnt_child; /* and going through their mnt_child */
        int mnt_flags;
        /* 4 bytes hole on 64bits arches */
        char *mnt_devname; /* Name of device e.g. /dev/dsk/hda1 */

        struct list_head mnt_list;  //all mounted file-systems are maintained in a master list


        struct list_head mnt_expire; /* link in fs-specific expiry list */
        struct list_head mnt_share; /* circular list of shared mounts */
        struct list_head mnt_slave_list;/* list of slave mounts */
        struct list_head mnt_slave; /* slave list entry */
        struct vfsmount *mnt_master; /* slave is on master->mnt_slave_list */
        struct mnt_namespace *mnt_ns; /* containing namespace */
        /*
         * We put mnt_count & mnt_expiry_mark at the end of struct vfsmount
         * to let these frequently modified fields in a separate cache line
         * (so that reads of mnt_flags wont ping-pong on SMP machines)
         */

  atomic_t mnt_count;
  int mnt_expiry_mark; /* true if marked for expiry */
};
```

- contains file-system specific information

```
<linux/fs.h>

struct super_block {
        struct list_head        s_list;         /* Keep this first */
        dev_t                   s_dev;          /* search index; _not_ kdev_t */
        unsigned long           s_blocksize;
        unsigned char           s_blocksize_bits;
        unsigned char           s_dirt;
        unsigned long long      s_maxbytes;     /* Max file size */
        struct file_system_type *s_type;
        struct super_operations *s_op;
        unsigned long           s_flags;
        unsigned long           s_magic;
        struct dentry           *s_root;
        struct xattr_handler    **s_xattr;
        struct list_head        s_inodes;       /* all inodes */
        struct list_head        s_dirty;        /* dirty inodes */
        struct list_head        s_io;           /* parked for writeback */
        struct list_head        s_more_io;      /* parked for more writeback */
        struct list_head        s_files;


  struct block_device     *s_bdev;
  struct list_head        s_instances;
  char s_id[32];                          /* Informational name */
  void                    *s_fs_info;   /* Filesystem private info */
  /* Granularity of c/m/atime in ns.
     Cannot be worse than a second */
  u32                     s_time_gran;
};
```

--------------intentionally left blank----------------

- file-system specific super-block related operations

```
<linux/fs.h>
struct super_operations {
	struct inode *(*alloc_inode)(struct super_block *sb);
	void (*destroy_inode)(struct inode *);
	void (*read_inode) (struct inode *);
	void (*dirty_inode) (struct inode *);
	int (*write_inode) (struct inode *, int);
	void (*put_inode) (struct inode *);
	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);
	void (*write_super_lockfs) (struct super_block *);
	void (*unlockfs) (struct super_block *);
	int (*statfs) (struct super_block *, 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 *);
};
```

Illustration of mount system call

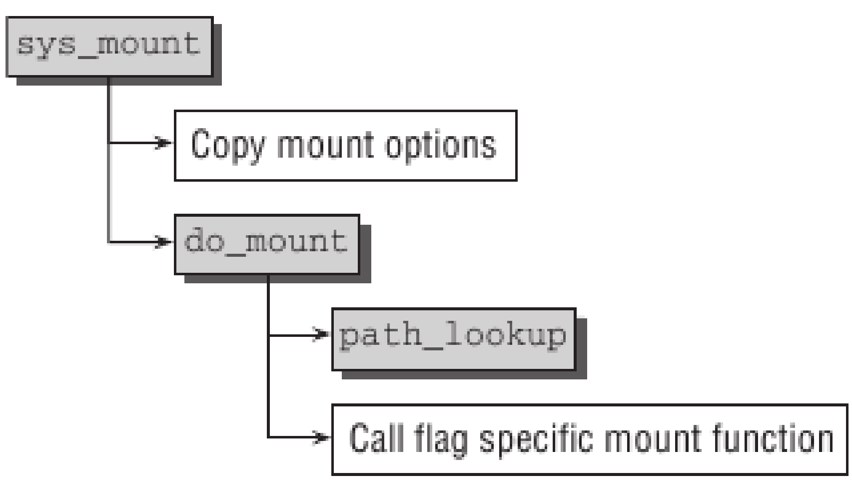
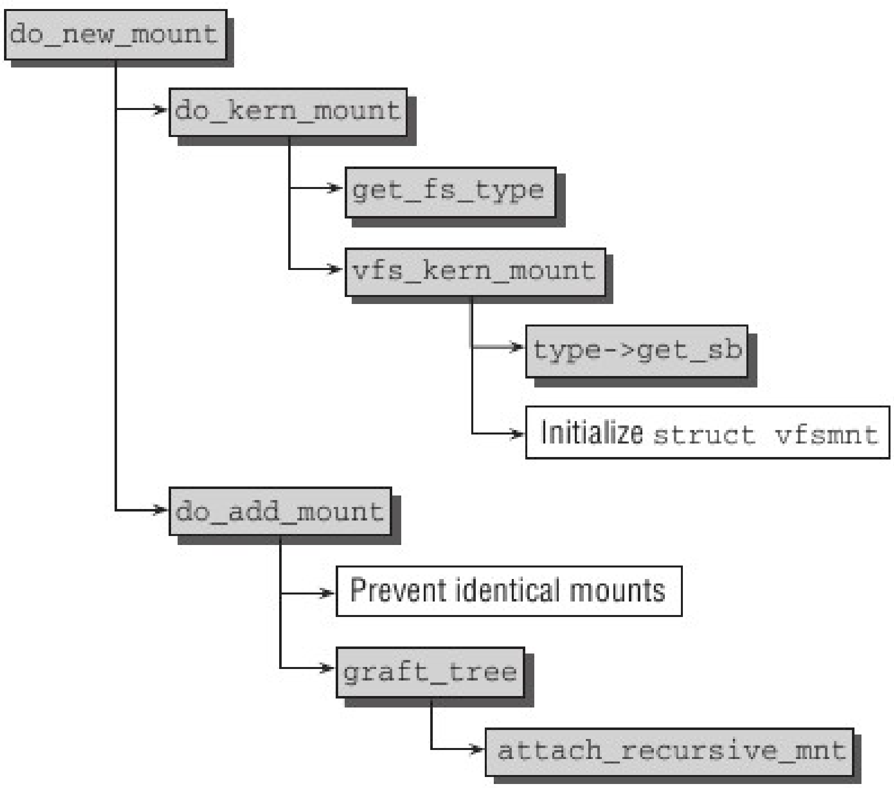


Illustration of the core mounting actions



------------------intentionally left free-----------------------

Illustration of unmount operations

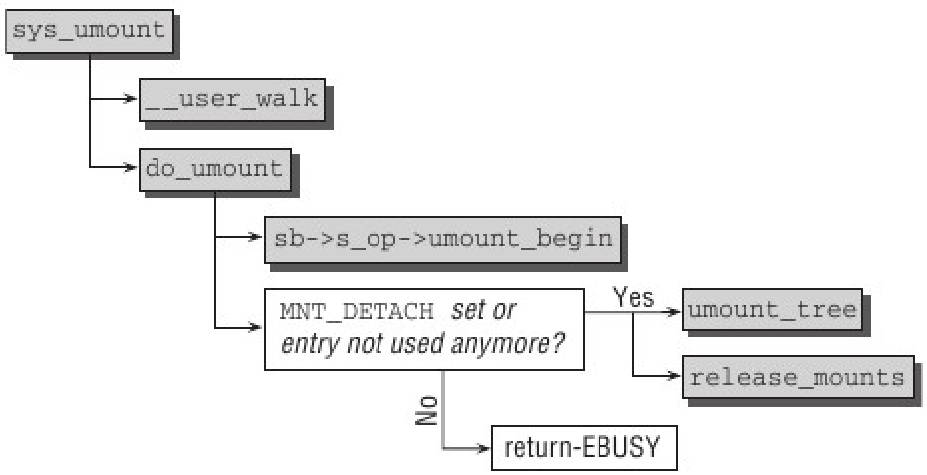
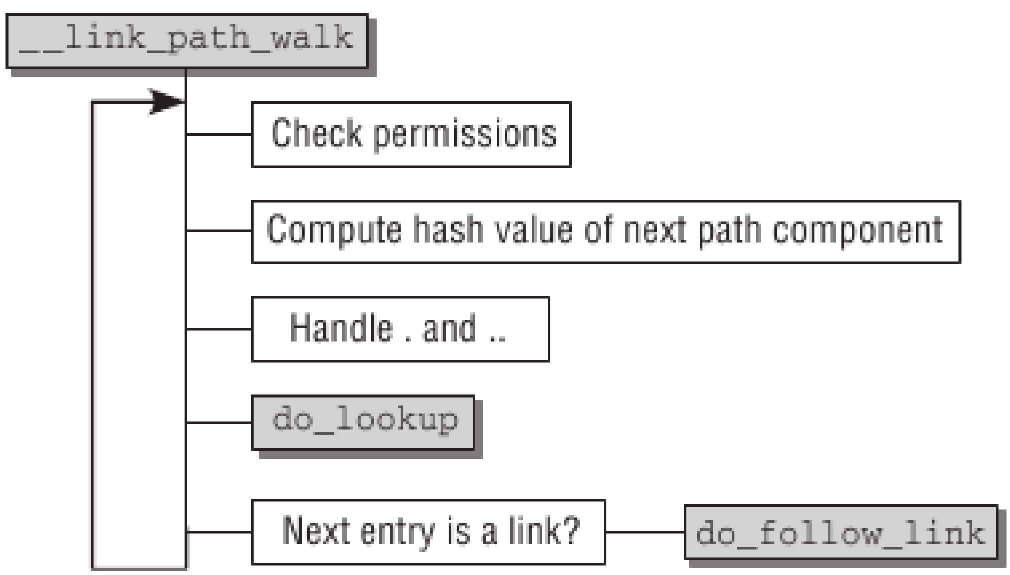
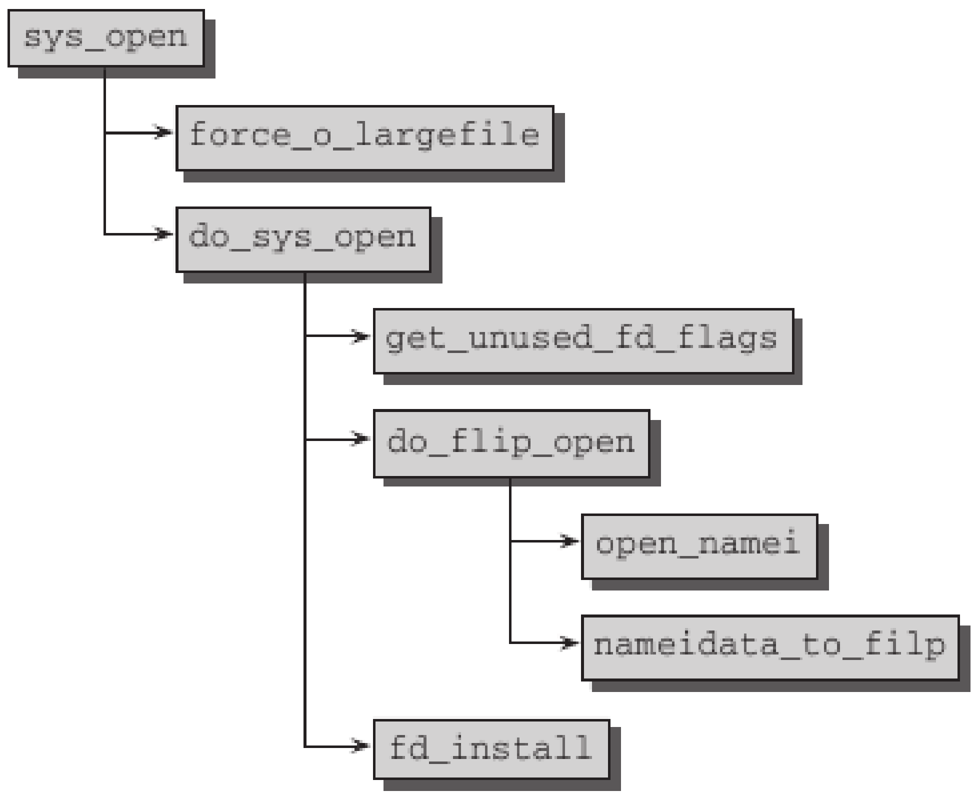


illustration of pathname look-up



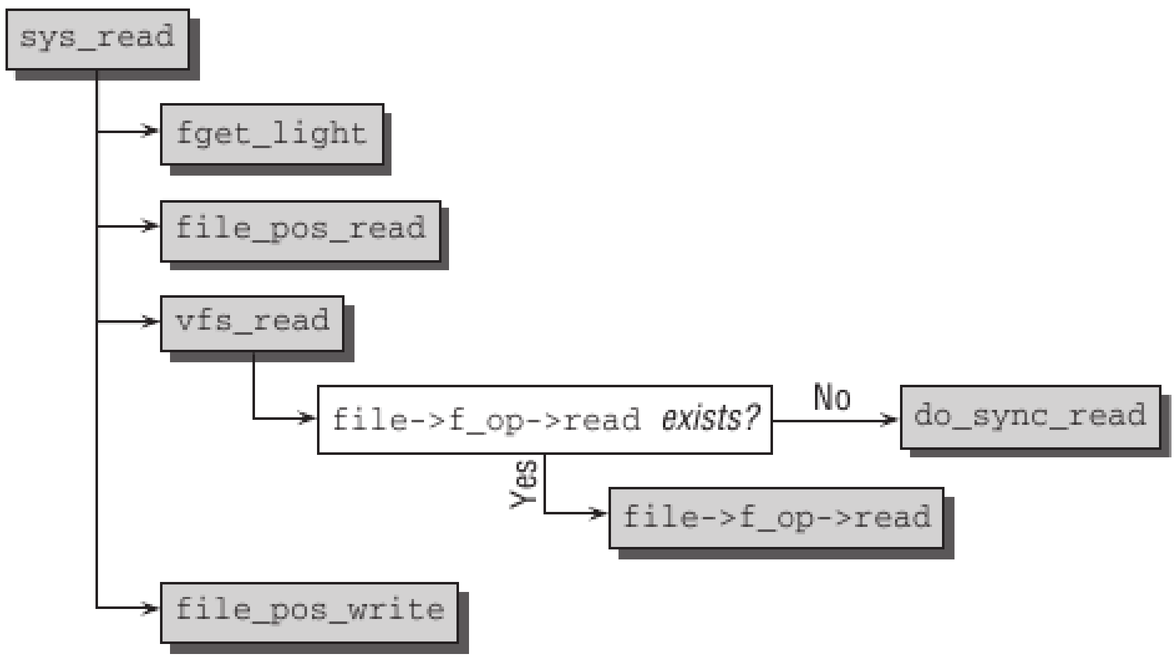
opening(activating a file)

illustration of opening a file(active file)



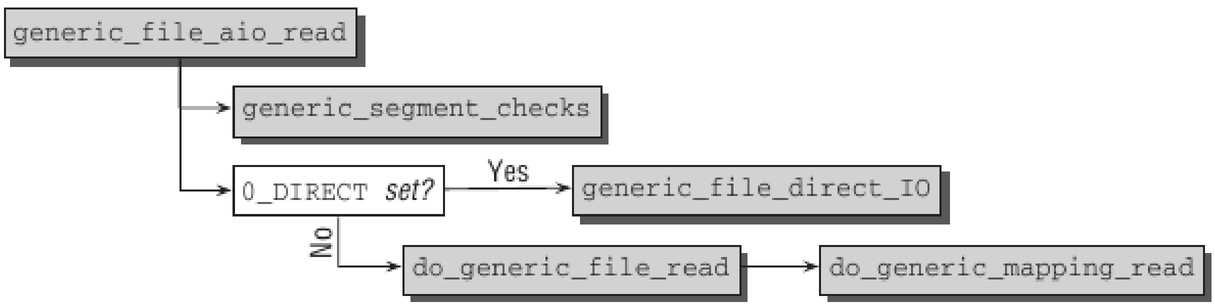
-------------------intentionally left free--------------------

illustration of reading from a file



---------------intentionally left free------------------

illustration of how the read ends up calling a mapping read



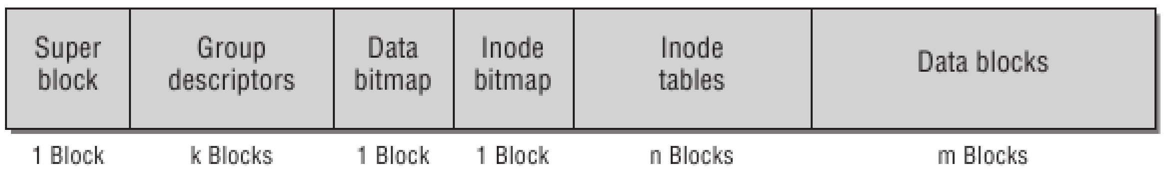
- finally, they end up calling find_get_page() which searches the page-cache and if it is available, give the page – otherwise, get it from the disk using file-system code

- page-cache is checked with a ptr to the address-space object and offset value in the file

------------------------intentionally left free--------------------------

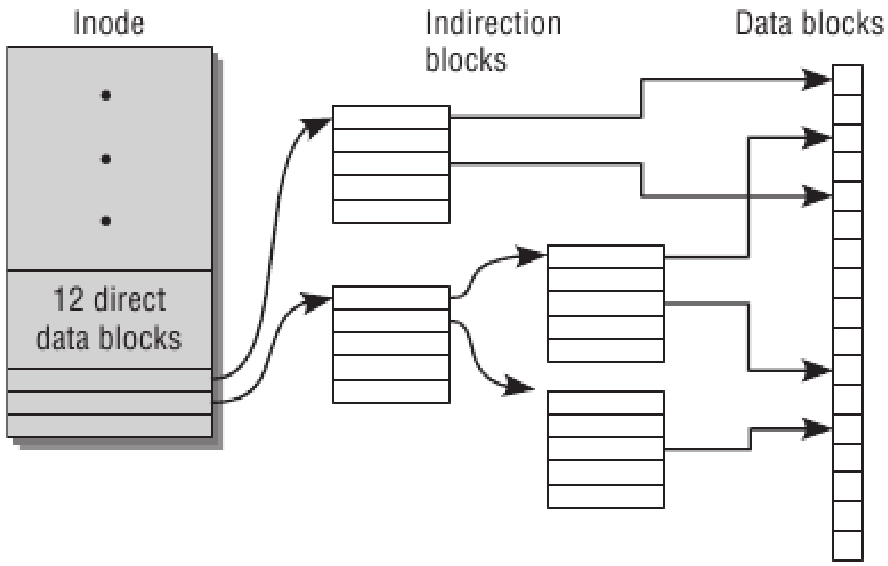
second extended file system

illustration of ext2 layout



-----------------intentionally left blank-----------------------

illustration of i-node((information node) structure on-disk



- 12 direct pointers and 3 indirect pointers(1 single/ 1 double / 1 triple)

illustration of maximum file-sizes with changing logical block size

Block size	Maximum file size
1,024	16 GiB
2,048	256 GiB
4,096	2 TiB

super-block structure on disk

<linux/ext2_fs.h>

```
struct ext2_super_block {
        __le32 s_inodes_count;          /* Inodes count */
        __le32 s_blocks_count;          /* Blocks count */
        __le32 s_r_blocks_count;        /* Reserved blocks count */
        __le32 s_free_blocks_count;     /* Free blocks count */
        __le32 s_free_inodes_count;     /* Free inodes count */
        __le32 s_first_data_block;      /* First Data Block */
        __le32 s_log_block_size;        /* Block size */
        __le32 s_log_frag_size;         /* Fragment size */
        __le32 s_blocks_per_group;      /* # Blocks per group */
        __le32 s_frags_per_group;       /* # Fragments per group */
        __le32 s_inodes_per_group;      /* # Inodes per group */
        __le32 s_mtime;                 /* Mount time */
        __le32 s_wtime;                 /* Write time */
        __le16 s_mnt_count;             /* Mount count */
        __le16 s_max_mnt_count;         /* Maximal mount count */
        __le16 s_magic;                 /* Magic signature */
        __le16 s_state;                 /* File system state */
        __le16 s_errors;                /* Behaviour when detecting errors */
        __le16 s_minor_rev_level;       /* minor revision level */
        __le32 s_lastcheck;             /* time of last check */
        __le32 s_checkinterval;         /* max. time between checks */
        __le32 s_creator_os;            /* OS */
        __le32 s_rev_level;             /* Revision level */
        __le16 s_def_resuid;            /* Default uid for reserved blocks */
        __le16 s_def_resgid;            /* Default gid for reserved blocks */
        /*
         * These fields are for EXT2_DYNAMIC_REV superblocks only.
         *
         * Note: the difference between the compatible feature set and
         * the incompatible feature set is that if there is a bit set
         * in the incompatible feature set that the kernel doesn't
         * know about, it should refuse to mount the filesystem.
         *
         * e2fsck's requirements are more strict; if it doesn't know
```

```
    __le32 s_first_ino;             /* First non-reserved inode */
    __le16   s_inode_size;          /* size of inode structure */
    __le16 s_block_group_nr;        /* block group # of this superblock */
    __le32 s_feature_compat;        /* compatible feature set */
    __le32 s_feature_incompat;      /* incompatible feature set */
    __le32 s_feature_ro_compat;     /* readonly-compatible feature set */
    __u8    s_uuid[16];             /* 128-bit uuid for volume */
    char    s_volume_name[16];      /* volume name */
    char    s_last_mounted[64];     /* directory where last mounted */
    __le32 s_algorithm_usage_bitmap; /* For compression */
    /*
     * Performance hints. Directory preallocation should only
     * happen if the EXT2_COMPAT_PREALLOC flag is on.
     */
    __u8    s_prealloc_blocks;      /* Nr of blocks to try to preallocate*/
    __u8    s_prealloc_dir_blocks; /* Nr to pre-allocate for dirs */
    __u16   s_padding1;
    /*
     * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
     */
...
    __u32   s_reserved[190];        /* Padding to the end of the block */
};
```

group descriptor on-disk

```
<linux/ext2_fs.h>
struct ext2_group_desc
{
        __le32 bg_block_bitmap;             /* Blocks bitmap block */
        __le32 bg_inode_bitmap;             /* Inodes bitmap block */
        __le32 bg_inode_table;      /* Inodes table block */
        __le16 bg_free_blocks_count; /* Free blocks count */
        __le16 bg_free_inodes_count; /* Free inodes count */
        __le16 bg_used_dirs_count;   /* Directories count */
        __le16 bg_pad;
        __le32 bg_reserved[3];
};
```

illustration of max. no of blocks possible per group

Block size	Number of blocks
1,024	8,192
2,048	16,384
4,096	32,768

inode on-disk

```
<ext2_fs.h>
struct ext2_inode {
        __le16 i_mode;        /* File mode */
        __le16 i_uid;         /* Low 16 bits of Owner Uid */
        __le32 i_size;        /* Size in bytes */
        __le32 i_atime;       /* Access time */
        __le32 i_ctime;       /* Creation time */
        __le32 i_mtime;       /* Modification time */
        __le32 i_dtime;       /* Deletion Time */
        __le16 i_gid;         /* Low 16 bits of Group Id */
        __le16 i_links_count; /* Links count */
        __le32 i_blocks;      /* Blocks count */
        __le32 i_flags;       /* File flags */
        union {
              struct {
                    __le32 l_i_reserved1;
              } linux1;
              struct {
              ...
              } hurd1;
              struct {
              ...
              } masix1;
        } osd1;                      /* OS dependent 1 */
        __le32 i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        __le32 i_generation;  /* File version (for NFS) */
        __le32 i_file_acl;    /* File ACL */
        __le32 i_dir_acl;     /* Directory ACL */
        __le32 i_faddr;       /* Fragment address */
```

```
        union {
            struct {

                __u8   l_i_frag;       /* Fragment number */
                __u8   l_i_fsize;      /* Fragment size */
                __u16  i_pad1;
                __le16 l_i_uid_high;   /* these 2 fields    */
                __le16 l_i_gid_high;   /* were reserved2[0] */
                __u32  l_i_reserved2;
            } linux2;
            struct {
            ...
            } hurd2;
            struct {
            ...
            } masix2;
        } osd2;                        /* OS dependent 2 */
};
```

directory entry in the inode

```
<linux/ext2_fs.h>
struct ext2_dir_entry_2 {
        __le32 inode;                  /* Inode number */
        __le16 rec_len;                /* Directory entry length */
        __u8   name_len;               /* Name length */
        __u8   file_type;
        char   name[EXT2_NAME_LEN];    /* File name */
};
typedef struct ext2_dir_entry_2 ext2_dirent;
```

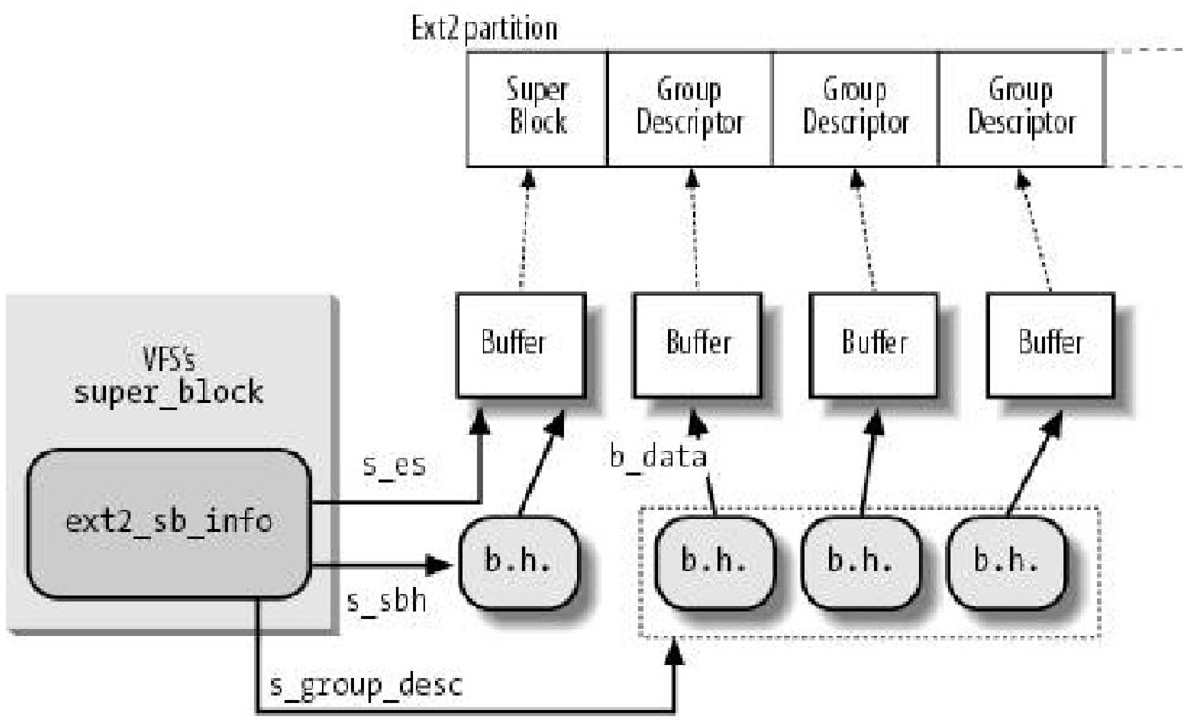
file-type enumeration

```
<linux/ext2_fs.h>
enum {
        EXT2_FT_UNKNOWN,
        EXT2_FT_REG_FILE,
        EXT2_FT_DIR,
        EXT2_FT_CHRDEV,
        EXT2_FT_BLKDEV,
        EXT2_FT_FIFO,
        EXT2_FT_SOCK,
        EXT2_FT_SYMLINK,
        EXT2_FT_MAX    };
```

illustration of a directory data-block entries(directory-entries)

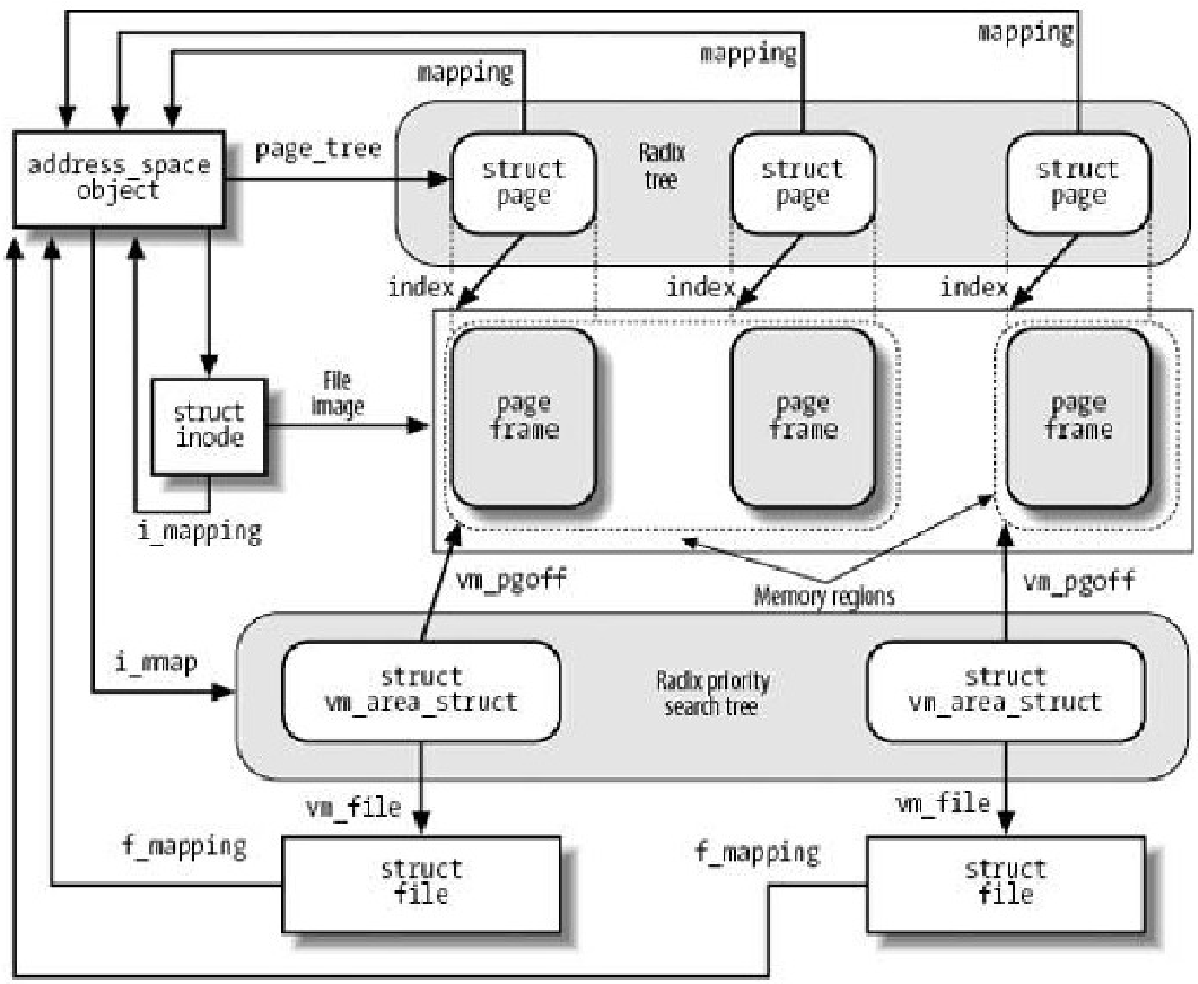
inode	rec_len	name_len	file_type	name							
	12	1	2	.	\0	\0	\0				
	12	2	2	.	.h	\0	\0				
	16	8	4	h	a	r	d	d	i	s	k
	32	5	7	l	i	n	u	x	\0	\0	\0
	16	6	2	d	e	l	d	i	r	\0	\0
	16	6	1	s	a	m	p	l	e	\0	\0
	16	7	2	s	o	u	r	c	e	\0	\0

illustration of in-memory data-*structures*



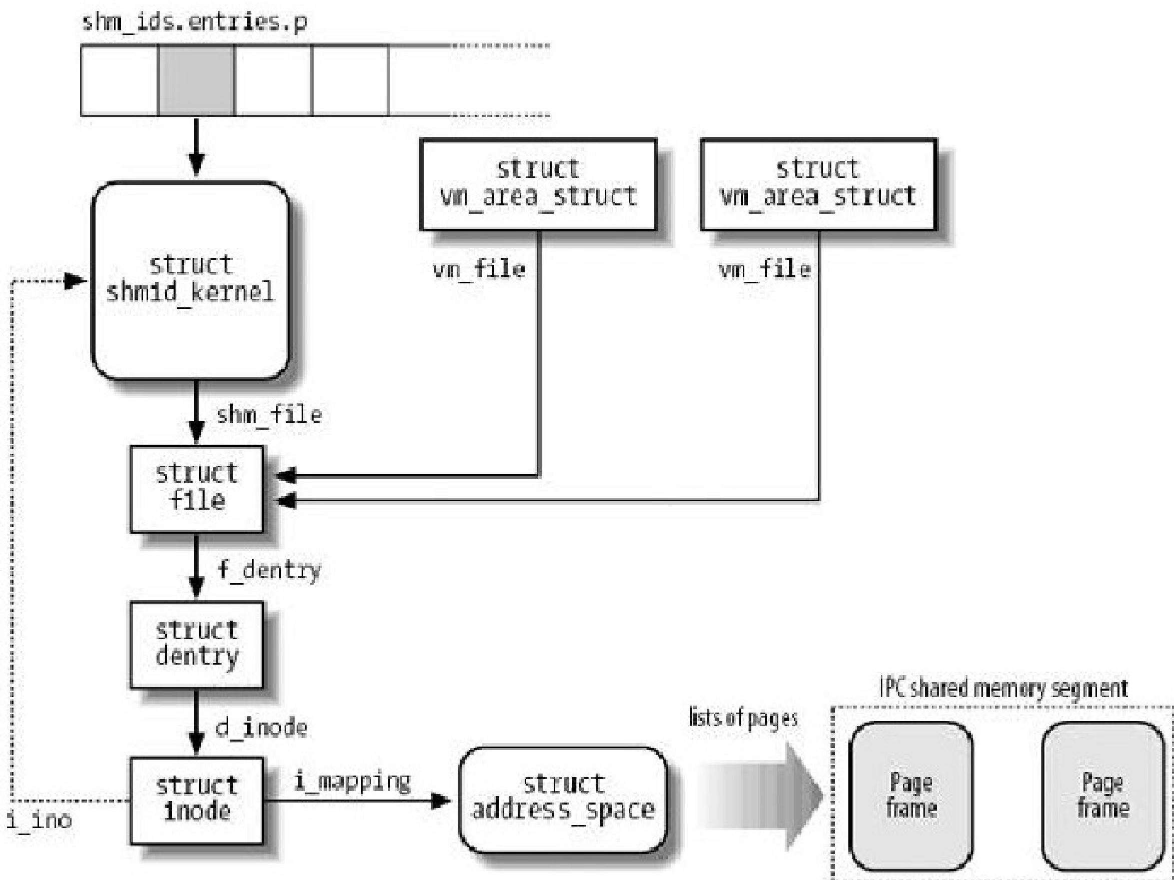
Type	Disk data structure	Memory data structure	Caching mode
Superblock	ext2_super_block	ext2_sb_info	Always cached
Group descriptor	ext2_group_desc	ext2_group_desc	Always cached
Block bitmap	Bit array in block	Bit array in buffer	Dynamic
inode bitmap	Bit array in block	Bit array in buffer	Dynamic
inode	ext2_inode	ext2_inode_info	Dynamic
Data block	Array of bytes	VFS buffer	Dynamic
Free inode	ext2_inode	None	Never
Free block	Array of bytes	None	Never

illustration of memory mapping of a file



- the specific page(page-frame) searched in the page-cache using the address-space object address
  and the offset(in page-size units) of the data needed in the

illustration of the shared-memory being merged with
the memory mapping architecture using a pseudo inode



Journaling

- ext3(third extended) file system is ext2 + journaling capability
- is designed to be backward compatible with ext2

- in particular, it is largely based on Ext2, so its data structures on disk are essentially identical to those of an Ext2 file-system. as a matter of fact, if an Ext3 file-system has been cleanly unmounted, it can be remounted as an Ext2 file-system

- clearly, the time spent checking the consistency of a file-system depends mainly on the number of files and directories to be examined; therefore, it also depends on the disk size. nowadays, with file-systems reaching hundreds of gigabytes, a single consistency check may take hours. the involved downtime is unacceptable for every production environment or high-availability server

- the goal of a journaling filesystem is to avoid running time-consuming consistency checks on the whole filesystem by looking instead in a special disk area that contains the most recent disk write operations named journal. Remounting a journaling filesystem after a system failure is a matter of a few seconds

- there is a special file(reserved) known as journal file

- data is first, quickly written to this file and periodically updated to the disk

- the idea behind Ext3 journaling is to perform each high-level change to the file-system in two steps.
  - first, a copy of the blocks written is stored in the journal
  - when the I/O data transfer to the file-system terminates (data is committed to the file-system) . copies of the blocks in the journal are discarded.

- the ext3 filesystem can be configured to log the operations affecting both the filesystem metadata and the data blocks of the files.

- because logging every kind of write operation leads to a significant performance penalty, Ext3 lets the system administrator decide what has to be logged; in particular, it offers three different journaling modes :
  - journal:

    All filesystem data and metadata changes are logged into the journal. This mode minimizes the chance of losing the updates made to each file, but it requires many additional disk accesses. for example, when a new file is created, all its data blocks must be duplicated as log records. this is the safest and slowest Ext3 journaling mode.

ordered :

only changes to filesystem metadata are logged into the journal. However, the Ext3 filesystem groups metadata and relative data blocks so that data blocks are written to disk before the meta-data. this way, the chance to have data corruption inside the files is reduced; for instance, each write access that enlarges a file is guaranteed to be fully protected by the journal. This is the default Ext3 journaling mode.

write-back:

only changes to filesystem metadata are logged; this is the method found on the other journaling filesystems and is the fastest mode.

e.g.   # mount   -t   ext3   -o data=writeback   /dev/sda2   /mnt

---