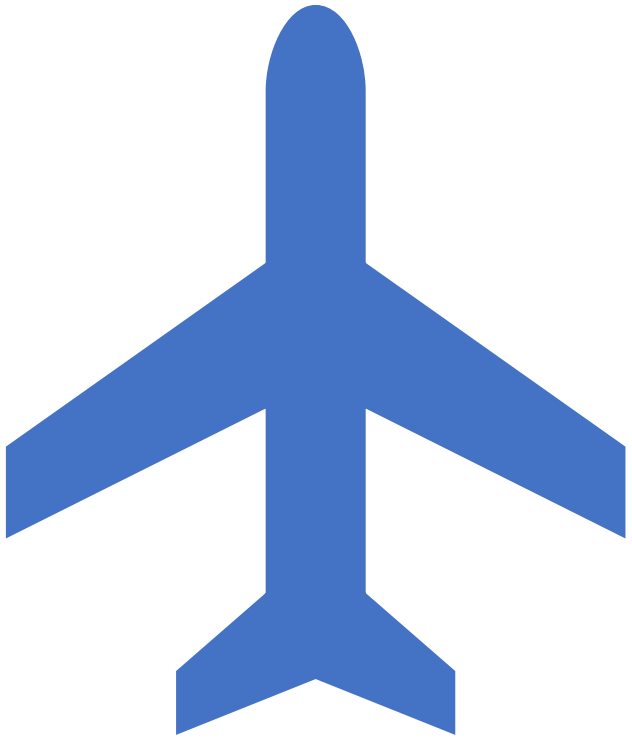


Travel in VFS and Ext4

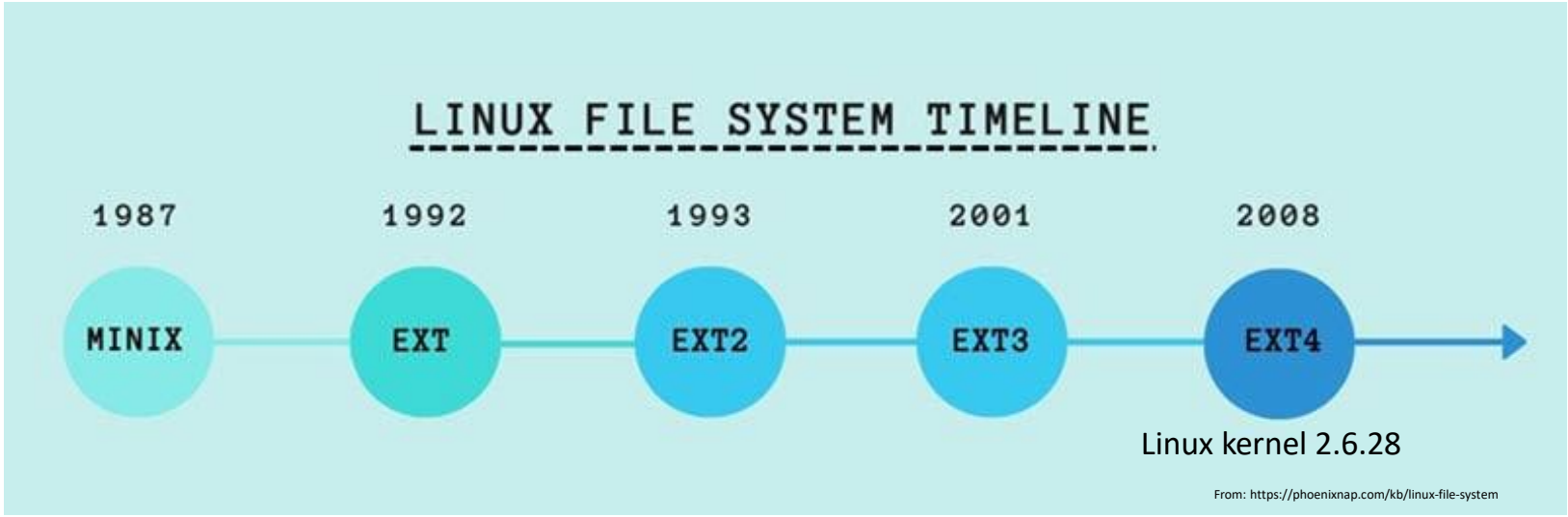
Title and content

- What is Ext4
- VFS
- Superblock
- Disk Layout
- Inode
- Directory Entry
- File Object
- Debugfs – Prowl around a FS
- Syscalls

What is Ext4

The **E**xtended Filesystem 4 (ext4) or fourth extended filesystem is a journaling file system for Linux, developed as the successor to ext3.

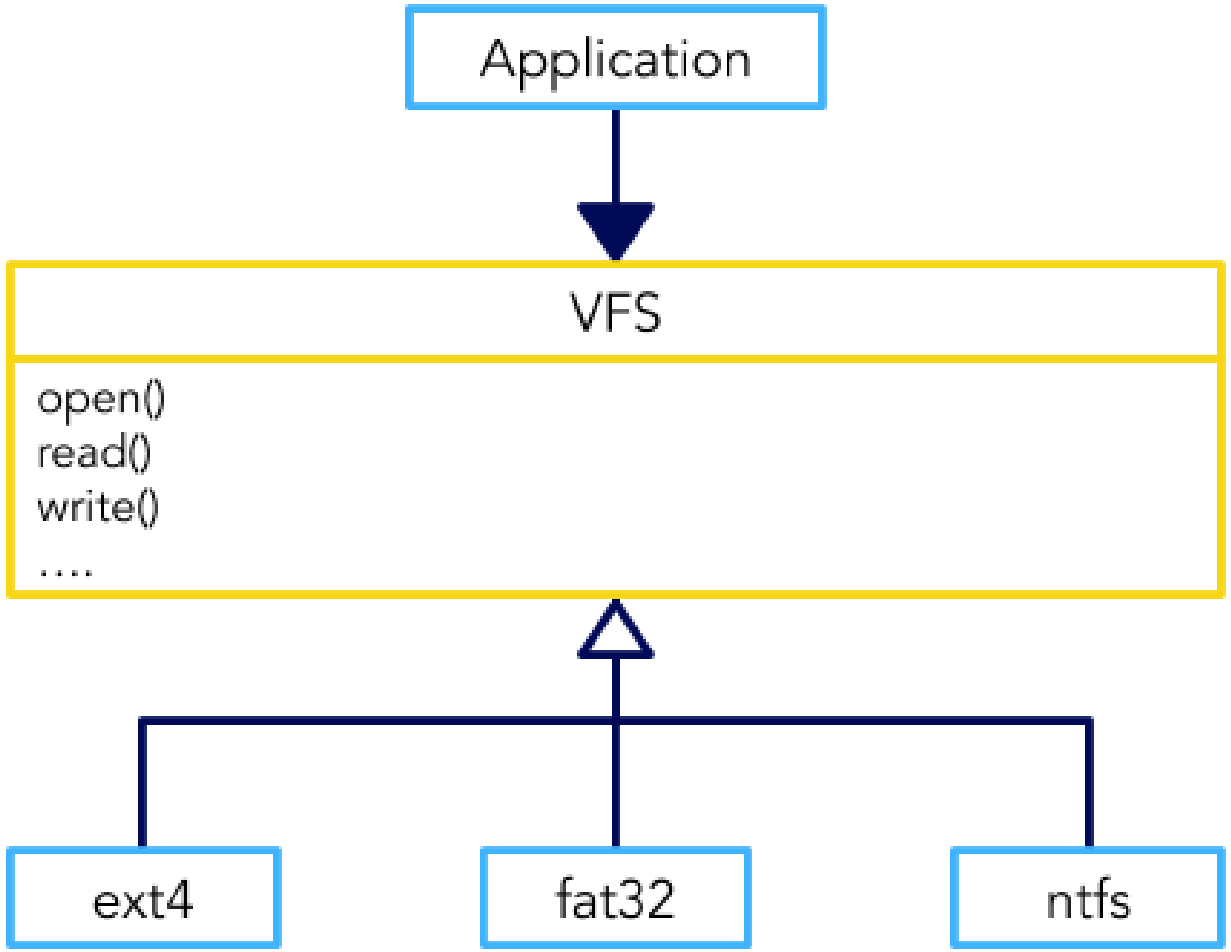
From https://en.wikipedia.org/wiki/Ext4



Virtual File System - VFS

The Virtual File System (also known as the Virtual Filesystem Switch) is the software layer in the kernel that provides the filesystem interface to userspace programs.

It also provides an abstraction within the kernel which allows different filesystem implementations to coexist.



From https://www.starlab.io/blog/introduction-to-the-linux-virtual-filesystem-vfs-part-i-a-high-level-tour

Virtual File System - VFS

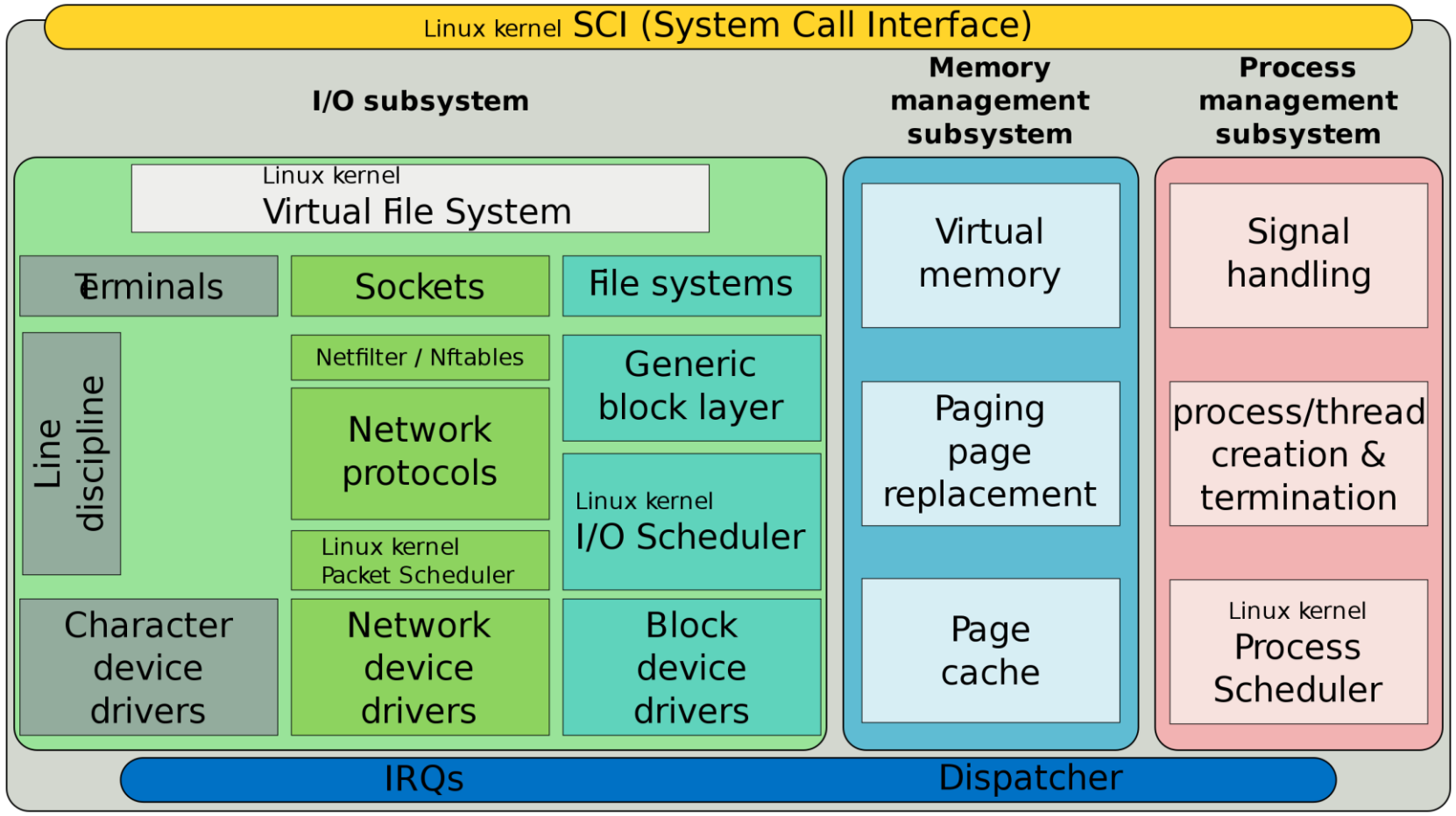
4 Main Abstracted Objects in VFS

- Superblock Object: representing a mounted filesystem.
- Inode Object: representing a file (device/FIFO/socket) in filesystem.
- Directory Entry Object: representing a directory entry (dentry).
- File Object: representing an opened file in process.

Operations/Structs of VFS Object

- *super_operations*: how the VFS can manipulate the superblock of your filesystem
- *inode_operations*: how the VFS can manipulate an inode in your filesystem
- *dentry_operations*: how a filesystem can overload the standard dentry operations
- *file_operations*: how the VFS can manipulate an open file

VFS & Ext4



From https://en.wikipedia.org/wiki/Ext4

Simplified structure of the Linux kernel: ext4 is implemented between the Linux kernel Virtual File System and the generic block layer.

VFS

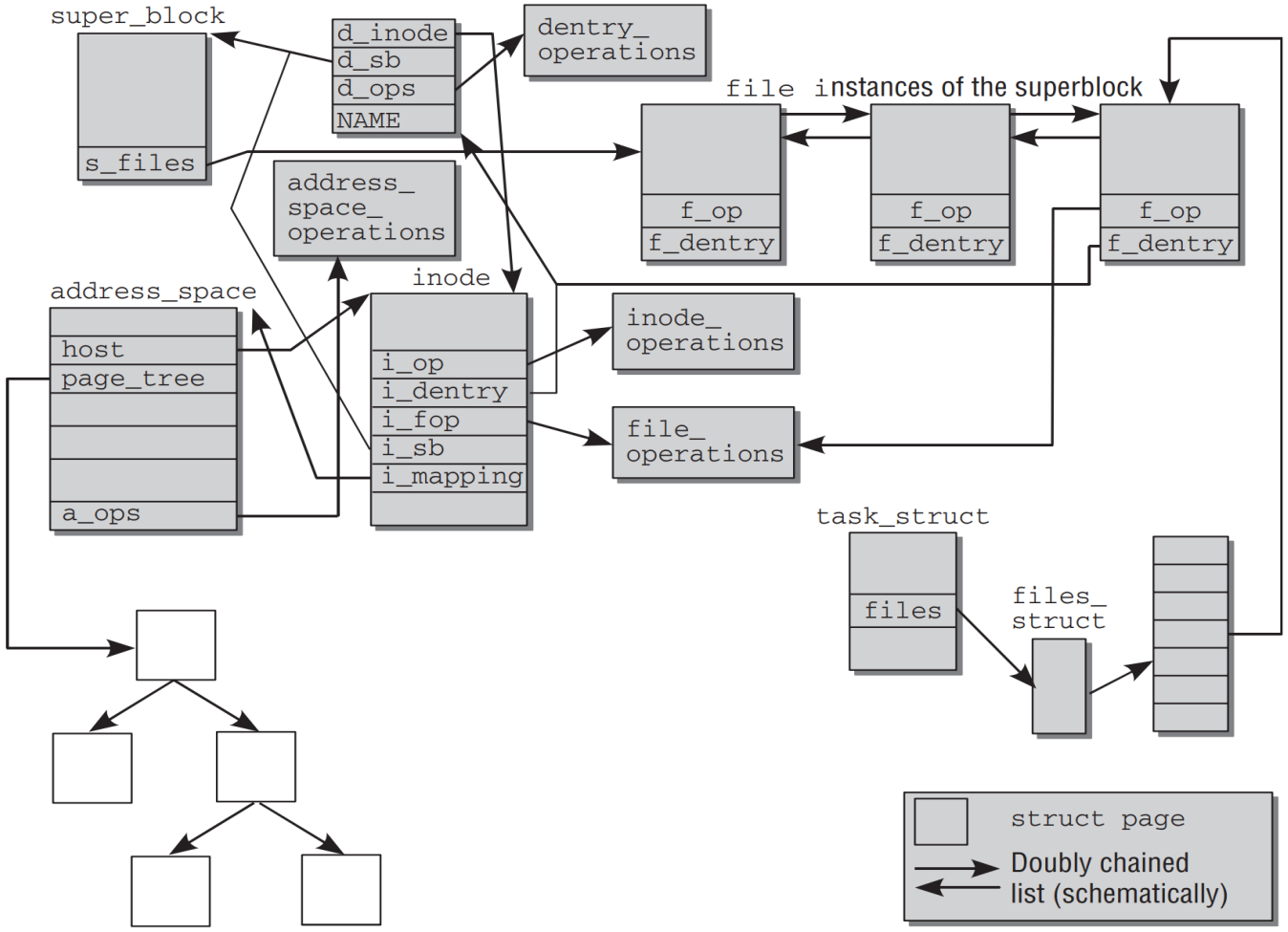


Figure 8-3: Interplay of the VFS components.

Book: Professional Linux Kernel Architecture, by Wolfgang Mauerer

Superblock

```
zegang@zegang-VirtualBox:~/1300-simple-fs$ dumpe2fs -h mydisk
dumpe2fs 1.45.5 (07-Jan-2020)
Filesystem volume name:   <none>
Last mounted on:          /home/zegang/1300-simple-fs/mydisk_mp
Filesystem UUID:          8115e317-4965-4fec-9291-10d2497c8691
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype needs_recovery extent 64bit flex_bg sparse_super large_file huge_file dir_nlink extra_isize metadata_csum
Filesystem flags:         signed_directory_hash
Default mount options:    user_xattr acl
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              32768
Block count:              32768
Reserved block count:     1638
Free blocks:              27620
Free inodes:              32754
First block:              0
Block size:               4096
Fragment size:            4096
Group descriptor size:    64
Reserved GDT blocks:      15
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         32768
Inode blocks per group:   1024
Flex block group size:    16
Filesystem created:       Thu Feb 24 17:36:56 2022
Last mount time:          Fri Feb 25 14:44:24 2022
Last write time:          Fri Feb 25 14:44:24 2022
Mount count:              2
Maximum mount count:      -1
Last checked:             Thu Feb 24 17:36:56 2022
Check interval:           0 (<none>)
Lifetime writes:          289 kB
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
Inode size:               128
Journal inode:            8
Default directory hash:   half_md4
Directory Hash Seed:      0ec78436-d75b-4d60-a4d4-e0b05cdcbfbb
Journal backup:           inode blocks
Checksum type:            crc32c
Checksum:                 0xaf25dd46
Journal features:         journal_64bit journal_checksum_v3
Journal size:             16M
Journal length:           4096
Journal sequence:         0x0000000b
Journal start:            1
Journal checksum type:    crc32c
Journal checksum:         0x4a5386bd
```

Superblock

A superblock is a record of the characteristics of a filesystem, including its size, the block size, the empty and the filled blocks and their respective counts, the size and location of the inode tables, the disk block map and usage information, and the size of the block groups.

$ dumpe2fs mydisk

```
Group 0: (Blocks 0-32767) csum 0x1894 [ITABLE_ZEROED]
  Primary superblock at 0, Group descriptors at 1-1
  Reserved GDT blocks at 2-16
  Block bitmap at 17 (+17), csum 0x48de92ce
  Inode bitmap at 33 (+33), csum 0x08c1e875
  Inode table at 49-1072 (+49)
  27108 free blocks, 32753 free inodes, 4 directories, 32753 unused inodes
  Free blocks: 5660-32767
  Free inodes: 16-32768
```

The superblock acquired its name from the fact that the first data block of a disk or of a partition was used to hold the meta-data (i.e., data about data) about the partition itself.

Superblock: Ext4 to VFS

```c
struct super_block {
        struct list_head        s_list;         /* Keep this first */
        dev_t                   s_dev;          /* search index; _not_ kdev_t */
        unsigned char           s_blocksize_bits;
        unsigned long           s_blocksize;
        loff_t                  s_maxbytes;     /* Max file size */
        struct file_system_type *s_type;
        const struct super_operations   *s_op;
        const struct dquot_operations   *dq_op;
        const struct quotactl_ops       *s_qcop;
        const struct export_operations *s_export_op;
        unsigned long           s_flags;
        unsigned long           s_iflags;       /* internal SB_I_* flags */
        unsigned long           s_magic;
        struct dentry           *s_root;
        struct rw_semaphore     s_umount;
        int                     s_count;
        atomic_t                s_active;
#ifdef CONFIG_SECURITY
        void                    *s_security;
#endif
        const struct xattr_handler **s_xattr;
#ifdef CONFIG_FS_ENCRYPTION
        const struct fscrypt_operations *s_cop;
        struct key              *s_master_keys; /* master crypto keys in use */
#endif
#ifdef CONFIG_FS_VERITY
        const struct fsverity_operations *s_vop;
#endif
```

https://elixir.bootlin.com/linux/v5.13/source/include/linux/fs.h#L1419

Routine: ext4_fill_super

```c
/*
 * Structure of the super block
 */
struct ext4_super_block {
/*00*/  __le32  s_inodes_count;         /* Inodes count */
        __le32  s_blocks_count_lo;      /* Blocks count */
        __le32  s_r_blocks_count_lo;    /* Reserved blocks count */
        __le32  s_free_blocks_count_lo; /* Free blocks count */
/*10*/  __le32  s_free_inodes_count;    /* Free inodes count */
        __le32  s_first_data_block;     /* First Data Block */
        __le32  s_log_block_size;       /* Block size */
```

```c
static struct dentry *ext4_mount(struct file_system_type *fs_type, int flags,
                        const char *dev_name, void *data)
{
        return mount_bdev(fs_type, flags, dev_name, data, ext4_fill_super);
}
```

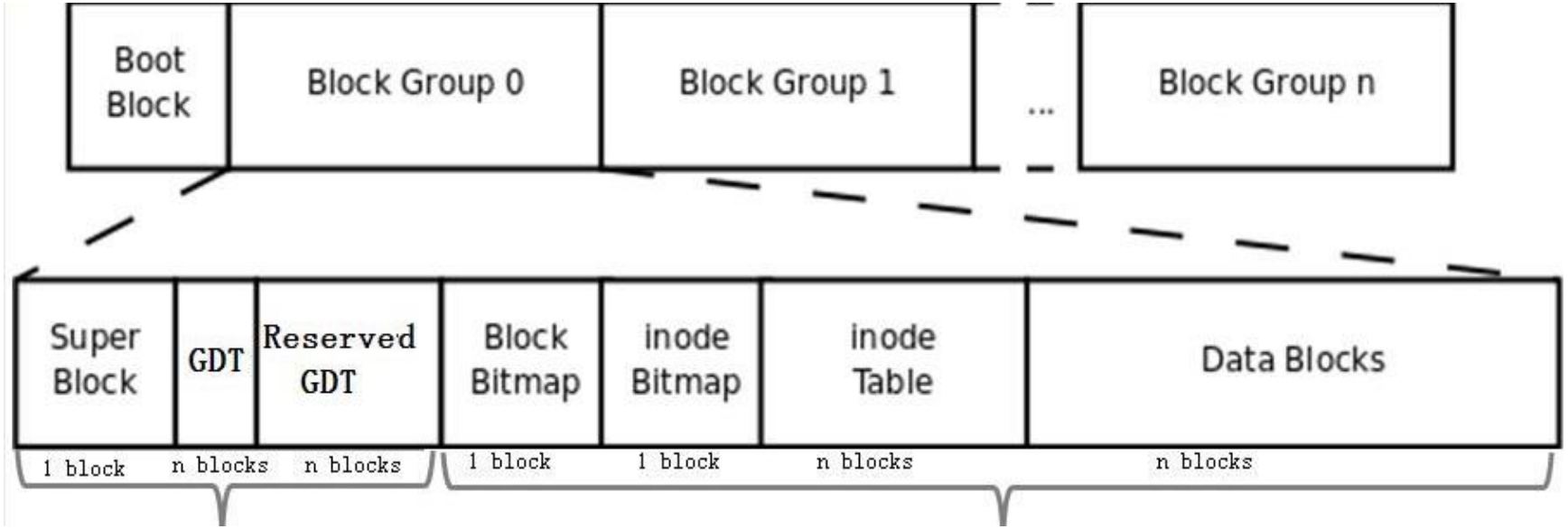
Routine: ext4_fill_super

```c
/*
 * fourth extended-fs super-block data in memory
 */
struct ext4_sb_info {
        unsigned long s_desc_size;      /* Size of a group descriptor in bytes */
        unsigned long s_inodes_per_block;/* Number of inodes per block */
        unsigned long s_blocks_per_group;/* Number of blocks in a group */
        unsigned long s_clusters_per_group; /* Number of clusters in a group */
        unsigned long s_inodes_per_group;/* Number of inodes in a group */
```

https://elixir.bootlin.com/linux/v5.13/source/fs/ext4/ext4.h#L1444

```c
static struct dentry *ext4_mount(struct file_system_type *fs_type, int flags,
                        const char *dev_name, void *data)
{
        return mount_bdev(fs_type, flags, dev_name, data, ext4_fill_super);
}
```

Disk Layout



Not in All Block Group

In All Block Group

The layout of a standard block group is approximately as follows:

Group 0 Padding	ext4 Super Block	Group Descriptors	Reserved GDT Blocks	Data Block Bitmap	inode Bitmap	inode Table	Data Blocks
1024 bytes	1 block	many blocks	many blocks	1 block	1 block	many blocks	many more blocks

Superblock: Ext4

```
1293  /*
1294   * Structure of the super block
1295   */
1296  struct ext4_super_block {
1297  /*00*/  __le32  s_inodes_count;         /* Inodes count */
1298          __le32  s_blocks_count_lo;      /* Blocks count */
1299          __le32  s_r_blocks_count_lo;    /* Reserved blocks count */
1300          __le32  s_free_blocks_count_lo; /* Free blocks count */
1301  /*10*/  __le32  s_free_inodes_count;    /* Free inodes count */
1302          __le32  s_first_data_block;     /* First Data Block */
1303          __le32  s_log_block_size;       /* Block size */
1304          __le32  s_log_cluster_size;     /* Allocation cluster size */
1305  /*20*/  __le32  s_blocks_per_group;     /* # Blocks per group */
1306          __le32  s_clusters_per_group;   /* # Clusters per group */
1307          __le32  s_inodes_per_group;     /* # Inodes per group */
1308          __le32  s_mtime;                /* Mount time */
1309  /*30*/  __le32  s_wtime;                /* Write time */
1310          __le16  s_mnt_count;            /* Mount count */
1311          __le16  s_max_mnt_count;        /* Maximal mount count */
1312          __le16  s_magic;                /* Magic signature */
1313          __le16  s_state;                /* File system state */
1314          __le16  s_errors;               /* Behaviour when detecting errors */
1315          __le16  s_minor_rev_level;      /* minor revision level */
```

```
zegang@zegang-VirtualBox:~/1300-simple-fs$ dumpe2fs mydisk
dumpe2fs 1.45.5 (07-Jan-2020)
Filesystem volume name:   <none>
Last mounted on:          /home/zegang/1300-simple-fs/mydisk_mp
Filesystem UUID:          8115e317-4965-4fec-9291-10d2497c8691
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir
Filesystem flags:         signed_directory_hash
Default mount options:    user_xattr acl
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              32768
Block count:              32768
Reserved block count:     1638
Free blocks:              27108
Free inodes:              32753
First block:              0
Block size:               4096
Fragment size:            4096
```

/ include / uapi / linux / magic.h

```
26  #define EXT4_SUPER_MAGIC        0xEF53
27  #define BTRFS_SUPER_MAGIC       0x9123683E
```

```
zegang@zegang-VirtualBox:~/1300-simple-fs$ hexdump -s 1024 -n 64 mydisk
0000400 8000 0000 8000 0000 0666 0000 69e4 0000
0000410 7ff1 0000 0000 0000 0002 0000 0002 0000
0000420 8000 0000 8000 0000 8000 0000 6217 621c
0000430 64ea 6223 0003 ffff ef53 0001 0001 0000
0000440
```

https://elixir.bootlin.com/linux/v5.13/source/fs/ext4/ext4.h#L1444

Superblock: Operations

/ include / linux / fs.h

```
2144
2145
2146  struct super_operations {
2147          struct inode *(*alloc_inode)(struct super_block *sb);
2148          void (*destroy_inode)(struct inode *);
2149          void (*free_inode)(struct inode *);
2150
2151          void (*dirty_inode) (struct inode *, int flags);
2152          int (*write_inode) (struct inode *, struct writeback_control *wbc);
2153          int (*drop_inode) (struct inode *);
2154          void (*evict_inode) (struct inode *);
2155          void (*put_super) (struct super_block *);
2156          int (*sync_fs)(struct super_block *sb, int wait);
2157          int (*freeze_super) (struct super_block *);
2158          int (*freeze_fs) (struct super_block *);
2159          int (*thaw_super) (struct super_block *);
2160          int (*unfreeze_fs) (struct super_block *);
2161          int (*statfs) (struct dentry *, struct kstatfs *);
2162          int (*remount_fs) (struct super_block *, int *, char *);
2163          void (*umount_begin) (struct super_block *);
2164
2165          int (*show_options)(struct seq_file *, struct dentry *);
2166          int (*show_devname)(struct seq_file *, struct dentry *);
2167          int (*show_path)(struct seq_file *, struct dentry *);
2168          int (*show_stats)(struct seq_file *, struct dentry *);
2169  #ifdef CONFIG_QUOTA
2170          ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
2171          ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
2172          struct dquot **(*get_dquots)(struct inode *);
2173  #endif
2174          int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
2175          long (*nr_cached_objects)(struct super_block *,
2176                                    struct shrink_control *);
2177          long (*free_cached_objects)(struct super_block *,
2178                                      struct shrink_control *);
2179  };
```

/ fs / ext4 / super.c

```
1638  static const struct super_operations ext4_sops = {
1639          .alloc_inode    = ext4_alloc_inode,
1640          .free_inode     = ext4_free_in_core_inode,
1641          .destroy_inode  = ext4_destroy_inode,
1642          .write_inode    = ext4_write_inode,
1643          .dirty_inode    = ext4_dirty_inode,
1644          .drop_inode     = ext4_drop_inode,
1645          .evict_inode    = ext4_evict_inode,
1646          .put_super      = ext4_put_super,
1647          .sync_fs        = ext4_sync_fs,
1648          .freeze_fs      = ext4_freeze,
1649          .unfreeze_fs    = ext4_unfreeze,
1650          .statfs         = ext4_statfs,
1651          .remount_fs     = ext4_remount,
1652          .show_options   = ext4_show_options,
1653  #ifdef CONFIG_QUOTA
1654          .quota_read     = ext4_quota_read,
1655          .quota_write    = ext4_quota_write,
1656          .get_dquots     = ext4_get_dquots,
1657  #endif
1658          .bdev_try_to_free_page = bdev_try_to_free_page,
1659  };
```

Inode

An inode is a data structure on a filesystem on a Unix-like operating system that stores all the information about a file except its name and its actual data.

```
zegang@zegang-VirtualBox:~/1300-simple-fs$ stat mydisk_mp/home/zegang/1.txt
  File: mydisk_mp/home/zegang/1.txt
  Size: 2               Blocks: 8          IO Block: 4096   regular file
Device: 70eh/1806d      Inode: 14          Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/  zegang)   Gid: ( 1000/  zegang)
Access: 2022-02-24 17:56:53.000000000 +0800
Modify: 2022-02-24 17:56:44.000000000 +0800
Change: 2022-02-24 17:56:44.000000000 +0800
 Birth: -
zegang@zegang-VirtualBox:~/1300-simple-fs$ ls -il mydisk_mp/home/zegang/1.txt
14 -rw-rw-r-- 1 zegang zegang 2 2月  24 17:56 mydisk_mp/home/zegang/1.txt
```

Metadata includes:

- the size of the file (in bytes) and its physical location,
- the file's owner and group,
- the file's access permissions,
- timestamps telling when the inode was created, last modified and last accessed,
- a reference count telling how many hard links point to the inode.

Inode

Space for inodes must be set aside when an operating system (or a new filesystem) is installed, and that system does its initial structuring of the filesystem. Within any filesystem, the maximum number of inodes, and hence the maximum number of files, is set when the filesystem is created.

```
zegang@zegang-VirtualBox:~/1300-simple-fs$ mkfs.ext4 mydisk
mke2fs 1.45.5 (07-Jan-2020)
Discarding device blocks: done
Creating filesystem with 32768 4k blocks and 32768 inodes

Allocating group tables: done
Writing inode tables: done
Creating journal (4096 blocks): done
Writing superblocks and filesystem accounting information: done
```

-i bytes-per-inode

References

Specify the bytes/inode ratio. **mke2fs** creates an inode for every bytes-per-inode bytes of space on the disk. The larger the bytes-per-inode ratio, the fewer inodes will be created. This value generally shouldn't be smaller than the blocksize of the filesystem, since in that case more inodes would be made than can ever be used. Be warned that it is not possible to change this ratio on a filesystem after it is created, so be careful deciding the correct value for this parameter. Note that resizing a filesystem changes the number of inodes to maintain this ratio.

-I inode-size

Specify the size of each inode in bytes. The inode-size value must be a power of 2 larger or equal to 128. The larger the inode-size the more space the inode table will consume, and this reduces the usable space in the filesystem and can also negatively impact performance. It is not possible to change this value after the filesystem is created.

In kernels after 2.6.10 and some earlier vendor kernels it is possible to utilize inodes larger than 128 bytes to store extended attributes for improved performance. Extended attributes stored in large inodes are not visible with older kernels, and such filesystems will not be mountable with 2.4 kernels at all.

The default inode size is controlled by the **mke2fs.conf**(5) file. In the **mke2fs.conf** file shipped with e2fsprogs, the default inode size is 256 bytes for most file systems, except for small file systems where the inode size will be 128 bytes.

Inode

```
606  /*
607   * Keep mostly read-only and often accessed (especially for
608   * the RCU path lookup and 'stat' data) fields at the beginning
609   * of the 'struct inode'
610   */
611  struct inode {
612          umode_t                 i_mode;
613          unsigned short          i_opflags;
614          kuid_t                  i_uid;
615          kgid_t                  i_gid;
616          unsigned int            i_flags;
617
618  #ifdef CONFIG_FS_POSIX_ACL
619          struct posix_acl        *i_acl;
620          struct posix_acl        *i_default_acl;
621  #endif
622
623          const struct inode_operations   *i_op;
624          struct super_block      *i_sb;
625          struct address_space    *i_mapping;
626
627  #ifdef CONFIG_SECURITY
628          void                    *i_security;
629  #endif
630
631          /* Stat data, not accessed from path walking */
632          unsigned long           i_ino;
633          /*
634           * Filesystems may only read i_nlink directly.  They shall use the
635           * following functions for modification:
636           *
637           *    (set|clear|inc|drop)_nlink
638           *    inode_(inc|dec)_link_count
639           */
640          union {
641                  const unsigned int i_nlink;
642                  unsigned int __i_nlink;
643          };
644          dev_t                   i_rdev;
645          loff_t                  i_size;
646          struct timespec64       i_atime;
647          struct timespec64       i_mtime;
648          struct timespec64       i_ctime;
```

https://elixir.bootlin.com/linux/v5.13/source/include/linux/fs.h#L611

```
996
997   /*
998    * fourth extended file system inode data in memory
999    */
1000  struct ext4_inode_info {
1001          __le32  i_data[15];     /* unconverted */
1002          __u32   i_dtime;
1003          ext4_fsblk_t    i_file_acl;
1004
1005          /*
1006           * i_block_group is the number of the block group which contains
1087           */
1088          struct rw_semaphore i_mmap_sem;
1089          struct inode vfs_inode;
1090          struct jbd2_inode *jinode;
```

ps://elixir.bootlin.com/linux/v5.13/source/fs/ext4/ext4.h#L1000

/ include / trace / events / ext4.h

```
16  struct mpage_da_data;
17  struct ext4_map_blocks;
18  struct extent_status;
19  struct ext4_fsmap;
20  struct partial_cluster;
21
22  #define EXT4_I(inode) (container_of(inode, struct ext4_inode_info, vfs_inode))
```

```
770  /*
771   * Structure of an inode on the disk
772   */
773  struct ext4_inode {
774          __le16  i_mode;         /* File mode */
775          __le16  i_uid;          /* Low 16 bits of Owner Uid */
776          __le32  i_size_lo;      /* Size in bytes */
777          __le32  i_atime;        /* Access time */
778          __le32  i_ctime;        /* Inode Change time */
```

Inode - VFS APIs & Ext4

```
2064 struct inode_operations {
2065         struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
2066         const char * (*get_link) (struct dentry *, struct inode *, struct delayed_call *);
2067         int (*permission) (struct user_namespace *, struct inode *, int);
2068         struct posix_acl * (*get_acl)(struct inode *, int);
2069
2070         int (*readlink) (struct dentry *, char __user *,int);
2071
2072         int (*create) (struct user_namespace *, struct inode *,struct dentry *,
2073                        umode_t, bool);
2074         int (*link) (struct dentry *,struct inode *,struct dentry *);
2075         int (*unlink) (struct inode *,struct dentry *);
2076         int (*symlink) (struct user_namespace *, struct inode *,struct dentry *,
2077                         const char *);
2078         int (*mkdir) (struct user_namespace *, struct inode *,struct dentry *,
2079                       umode_t);
2080         int (*rmdir) (struct inode *,struct dentry *);
2081         int (*mknod) (struct user_namespace *, struct inode *,struct dentry *,
2082                       umode_t,dev_t);
2083         int (*rename) (struct user_namespace *, struct inode *, struct dentry *,
2084                        struct inode *, struct dentry *, unsigned int);
2085         int (*setattr) (struct user_namespace *, struct dentry *,
2086                         struct iattr *);
2087         int (*getattr) (struct user_namespace *, const struct path *,
2088                         struct kstat *, u32, unsigned int);
2089         ssize_t (*listxattr) (struct dentry *, char *, size_t);
2090         int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
2091                       u64 len);
2092         int (*update_time)(struct inode *, struct timespec64 *, int);
2093         int (*atomic_open)(struct inode *, struct dentry *,
2094                            struct file *, unsigned open_flag,
2095                            umode_t create_mode);
2096         int (*tmpfile) (struct user_namespace *, struct inode *,
2097                         struct dentry *, umode_t);
2098         int (*set_acl)(struct user_namespace *, struct inode *,
2099                        struct posix_acl *, int);
2100         int (*fileattr_set)(struct user_namespace *mnt_userns,
2101                             struct dentry *dentry, struct fileattr *fa);
2102         int (*fileattr_get)(struct dentry *dentry, struct fileattr *fa);
2103 } ____cacheline_aligned;
```

https://elixir.bootlin.com/linux/v5.13/source/include/linux/fs.h#L2064

```
const struct inode_operations ext4_file_inode_operations = {
        .setattr        = ext4_setattr,
        .getattr        = ext4_file_getattr,
        .listxattr      = ext4_listxattr,
        .get_acl        = ext4_get_acl,
        .set_acl        = ext4_set_acl,
        .fiemap         = ext4_fiemap,
        .fileattr_get   = ext4_fileattr_get,
        .fileattr_set   = ext4_fileattr_set,
};
```

From: https://elixir.bootlin.com/linux/latest/source/fs/ext4/file.c

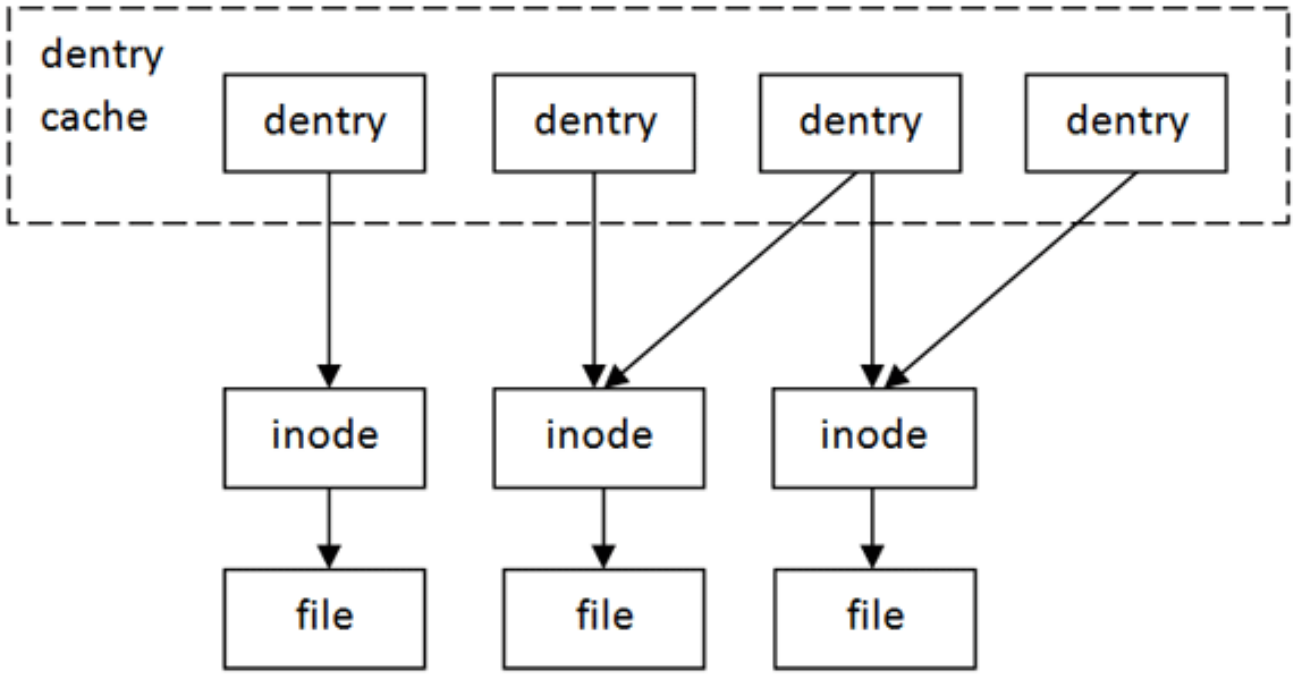
Directory Entry - Dentry

As discussed, the VFS treats directories as files. An inode object represents both these components.

Despite this useful unification, the VFS often needs to perform directory-specific operations, such as path name lookup. To facilitate this, the VFS employs the concept of a directory entry (dentry).

Example: /bin/vi

- /
- bin
- vi



From: https://www.apriorit.com/dev-blog/279-driver-to-hide-files-linux

Directory Entry - Dentry

The VFS constructs dentry objects on the fly, as needed, when performing directory operations.

Mainly target is mapping **File Name** to **Inode object**. Only the **partial** path name/component is stored in *d_name*. If the name is short, it is stored in *d_iname* to fast access.

```
struct dentry {
        /* RCU lookup touched fields */
        unsigned int d_flags;           /* protected by d_lock */
        seqcount_spinlock_t d_seq;      /* per dentry seqlock */
        struct hlist_bl_node d_hash;    /* lookup hash list */
        struct dentry *d_parent;        /* parent directory */
        struct qstr d_name;
        struct inode *d_inode;          /* Where the name belongs to - NULL is
                                         * negative */
        unsigned char d_iname[DNAME_INLINE_LEN];        /* small names */

        /* Ref lookup also touches following */
        struct lockref d_lockref;       /* per-dentry lock and refcount */
        const struct dentry_operations *d_op;
        struct super_block *d_sb;       /* The root of the dentry tree */
        unsigned long d_time;           /* used by d_revalidate */
        void *d_fsdata;                 /* fs-specific data */

        union {
                struct list_head d_lru;         /* LRU list */
                wait_queue_head_t *d_wait;      /* in-lookup ones only */
        };
        struct list_head d_child;       /* child of parent list */
        struct list_head d_subdirs;     /* our children */
        /*
         * d_alias and d_rcu can share memory
         */
        union {
                struct hlist_node d_alias;      /* inode alias list */
                struct hlist_bl_node d_in_lookup_hash;  /* only for in-lookup ones */
                struct rcu_head d_rcu;
        } d_u;
} __randomize_layout;
```

Dentry objects are represented by struct dentry and defined in <linux/dcache.h>.

```
/*
 * Try to keep struct dentry aligned on 64 byte cachelines (this will
 * give reasonable cacheline footprint with larger lines without the
 * large memory footprint increase).
 */
#ifdef CONFIG_64BIT
# define DNAME_INLINE_LEN 32 /* 192 bytes */
#else
# ifdef CONFIG_SMP
#  define DNAME_INLINE_LEN 36 /* 128 bytes */
# else
#  define DNAME_INLINE_LEN 40 /* 128 bytes */
# endif
#endif
```

Directory Entry - Dentry

Kernel caches *dentry* objects in the *dentry* cache or, simply, the ***dcache***. A hash table and hashing function used to quickly resolve a given path into the associated *dentry* object.

All in memory active dentry objects are saved in a global hash list: *dentry_hashtable*.

```
90  /*
91   * This is the single most critical data structure when it comes
92   * to the dcache: the hashtable for lookups. Somebody should try
93   * to make this good - I've just made it work.
94   *
95   * This hash-function tries to avoid losing too many bits of hash
96   * information, yet avoid using a prime hash-size or similar.
97   */
98
99  static unsigned int d_hash_shift __read_mostly;
100
101 static struct hlist_bl_head *dentry_hashtable __read_mostly;
102
103 static inline struct hlist_bl_head *d_hash(unsigned int hash)
104 {
105         return dentry_hashtable + (hash >> d_hash_shift);
106 }
```

https://elixir.bootlin.com/linux/v5.13/source/fs/dcache.c#L101

All in memory unused (d_count == 0) dentry objects are saved in a filesystem instance global LRU list :s_dentry_lru.

```
1419 struct super_block {
1420         struct list_head        s_list;         /* Keep this first */
1421         dev_t                   s_dev;          /* search index; _not_ kdev_t */
1422         unsigned char           s_blocksize_bits;
1423         unsigned long           s_blocksize;
1424         loff_t                  s_maxbytes;     /* Max file size */
1425         struct file_system_type *s_type;
1426         const struct super_operations   *s_op;

1530         /*
1531          * The list_lru structure is essentially just a pointer to a table
1532          * of per-node lru lists, each of which has its own spinlock.
1533          * There is no need to put them into separate cachelines.
1534          */
1535         struct list_lru         s_dentry_lru;
1536         struct list_lru         s_inode_lru;
1537         struct rcu_head         rcu;
1538         struct work_struct      destroy_work;
```

https://elixir.bootlin.com/linux/v5.13/source/include/linux/fs.h#L1535

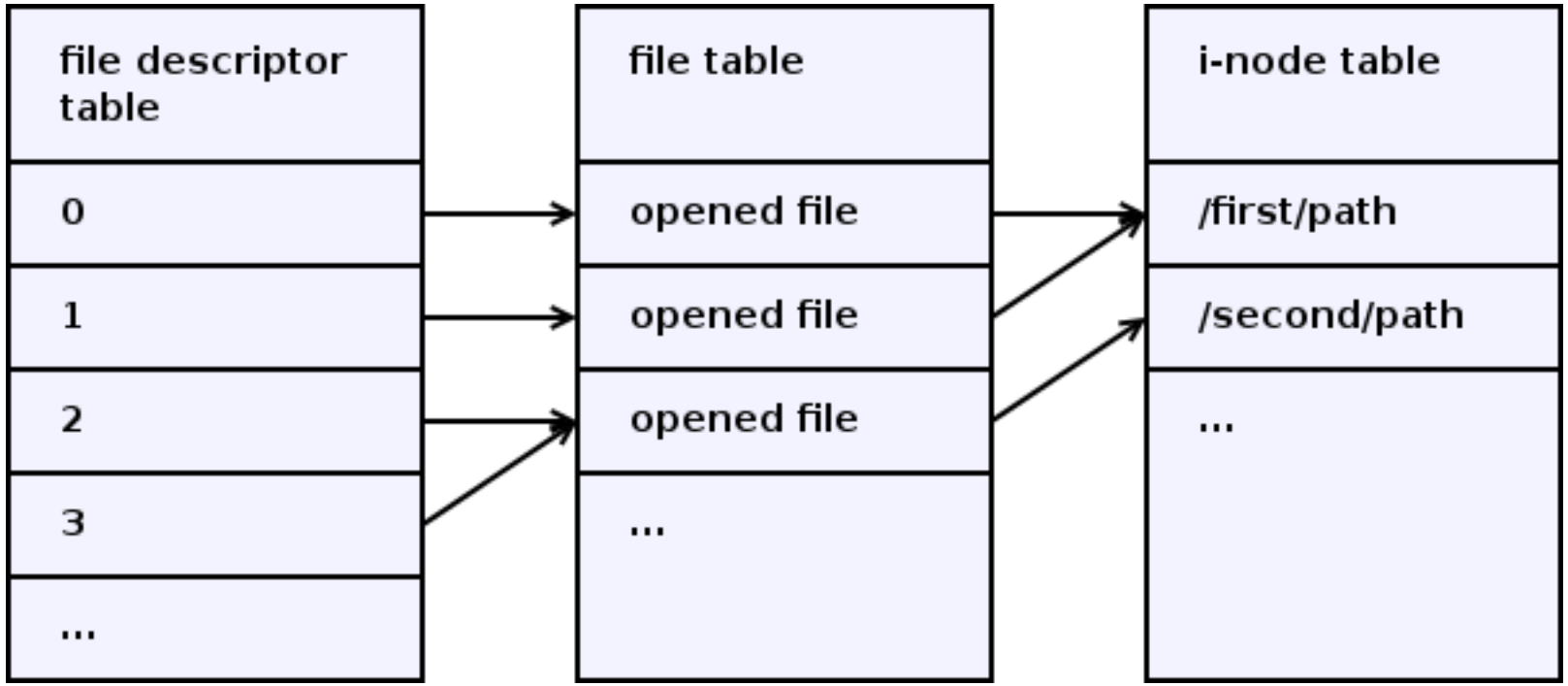
Directory Entry - Dentry

FS Specified dentry operations

```
137  struct dentry_operations {
138          int (*d_revalidate)(struct dentry *, unsigned int);
139          int (*d_weak_revalidate)(struct dentry *, unsigned int);
140          int (*d_hash)(const struct dentry *, struct qstr *);
141          int (*d_compare)(const struct dentry *,
142                           unsigned int, const char *, const struct qstr *);
143          int (*d_delete)(const struct dentry *);
144          int (*d_init)(struct dentry *);
145          void (*d_release)(struct dentry *);
146          void (*d_prune)(struct dentry *);
147          void (*d_iput)(struct dentry *, struct inode *);
148          char *(*d_dname)(struct dentry *, char *, int);
149          struct vfsmount *(*d_automount)(struct path *);
150          int (*d_manage)(const struct path *, bool);
151          struct dentry *(*d_real)(struct dentry *, const struct inode *);
152  } ____cacheline_aligned;
```

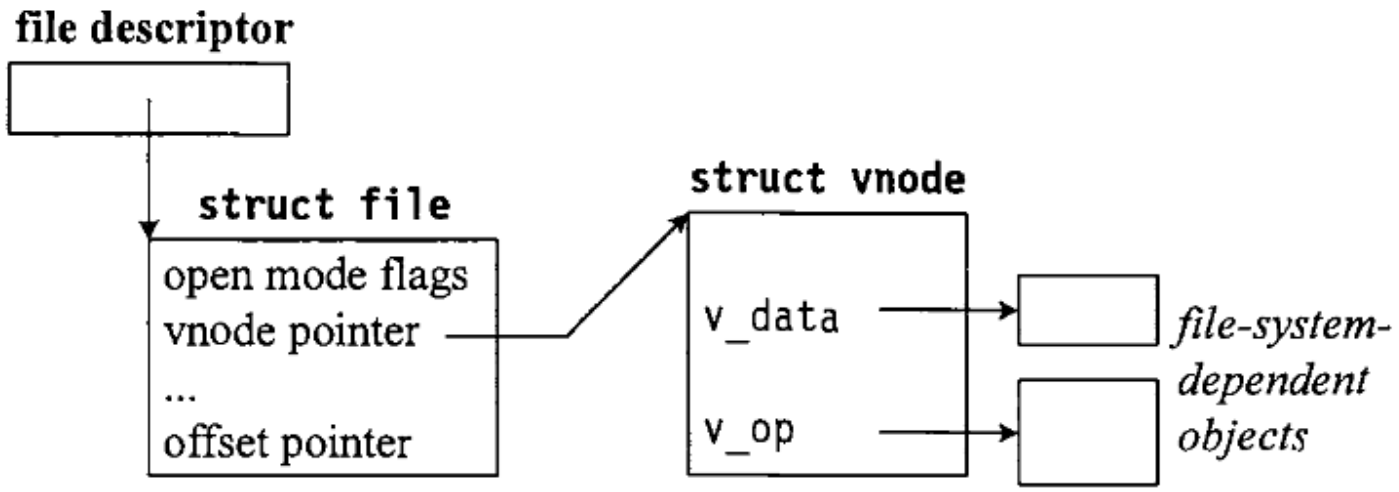
https://elixir.bootlin.com/linux/v5.13/source/include/linux/dcache.h#L137

File Object



From: https://gavv.github.io/articles/file-locks/

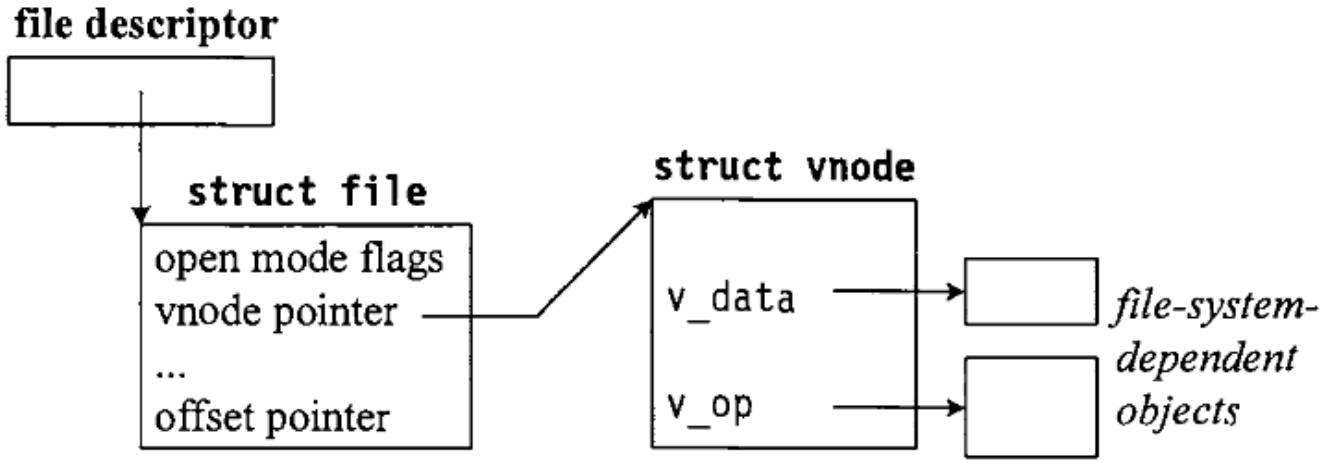
File Object




```
657  struct task_struct {
658  #ifdef CONFIG_THREAD_INFO_IN_TASK
659          /*
660           * For reasons of header soup (see current_thread_info()), this
661           * must be the first element of task_struct.
662           */
663          struct thread_info              thread_info;
664  #endif
665          /* -1 unrunnable, 0 runnable, >0 stopped: */
666          volatile long                   state;
667
```

```
984          /* Filesystem information: */
985          struct fs_struct                *fs;
986
987          /* Open file information: */
988          struct files_struct             *files;
```

File Object



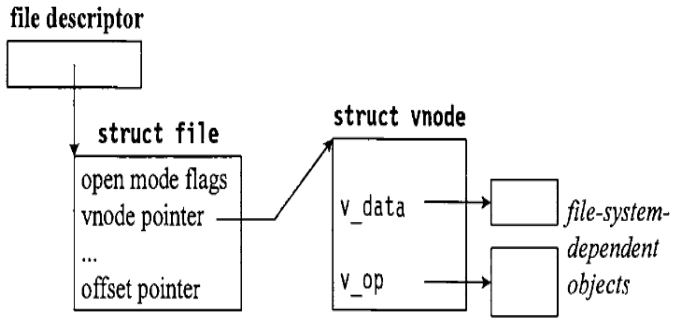
/ include / linux / fdtable.h

```
49  struct files_struct {
50    /*
51     * read mostly part
52     */
53          atomic_t count;
54          bool resize_in_progress;
55          wait_queue_head_t resize_wait;
56
57          struct fdtable __rcu *fdt;
58          struct fdtable fdtab;
59    /*
60     * written part on a separate cache line in SMP
61     */
62          spinlock_t file_lock ____cacheline_aligned_in_smp;
63          unsigned int next_fd;
64          unsigned long close_on_exec_init[1];
65          unsigned long open_fds_init[1];
66          unsigned long full_fds_bits_init[1];
67          struct file __rcu * fd_array[NR_OPEN_DEFAULT];
68  };
```

/ include / linux / fdtable.h

```
27  struct fdtable {
28          unsigned int max_fds;
29          struct file __rcu **fd;      /* current fd array */
30          unsigned long *close_on_exec;
31          unsigned long *open_fds;
32          unsigned long *full_fds_bits;
33          struct rcu_head rcu;
34  };
```

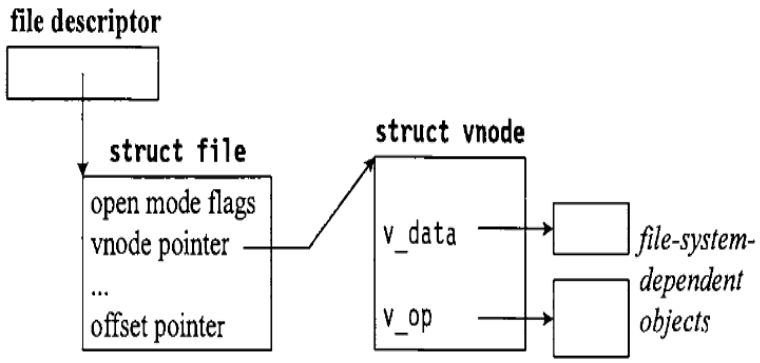
File Object



/ include / linux / fs.h

```
920  struct file {
921          union {
922                  struct llist_node       fu_llist;
923                  struct rcu_head         fu_rcuhead;
924          } f_u;
925          struct path             f_path;
926          struct inode            *f_inode;       /* cached value */
927          const struct file_operations    *f_op;
928
929          /*
930           * Protects f_ep, f_flags.
931           * Must not be taken from IRQ context.
932           */
933          spinlock_t              f_lock;
934          enum rw_hint            f_write_hint;
935          atomic_long_t           f_count;
936          unsigned int            f_flags;
937          fmode_t                 f_mode;
938          struct mutex            f_pos_lock;
939          loff_t                  f_pos;
940          struct fown_struct      f_owner;
941          const struct cred       *f_cred;
942          struct file_ra_state    f_ra;
943
944          u64                     f_version;
945  #ifdef CONFIG_SECURITY
946          void                    *f_security;
947  #endif
948          /* needed for tty driver, and maybe others */
949          void                    *private_data;
950
951  #ifdef CONFIG_EPOLL
952          /* Used by fs/eventpoll.c to link all the hooks to this file */
953          struct hlist_head       *f_ep;
954  #endif /* #ifdef CONFIG_EPOLL */
955          struct address_space    *f_mapping;
956          errseq_t                f_wb_err;
957          errseq_t                f_sb_err; /* for syncfs */
958  } __randomize_layout
959    __attribute__((aligned(4)));  /* lest something weird decides that 2 is OK */
```

File Object

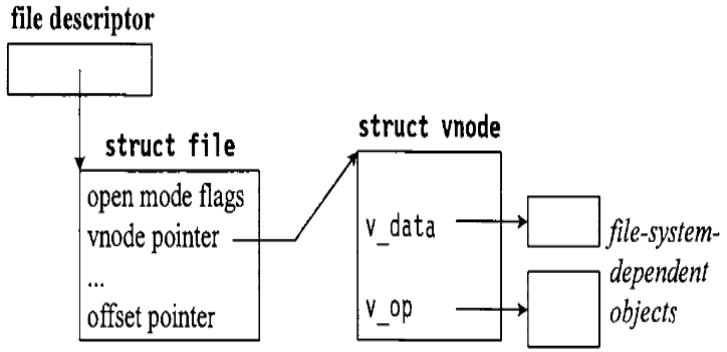


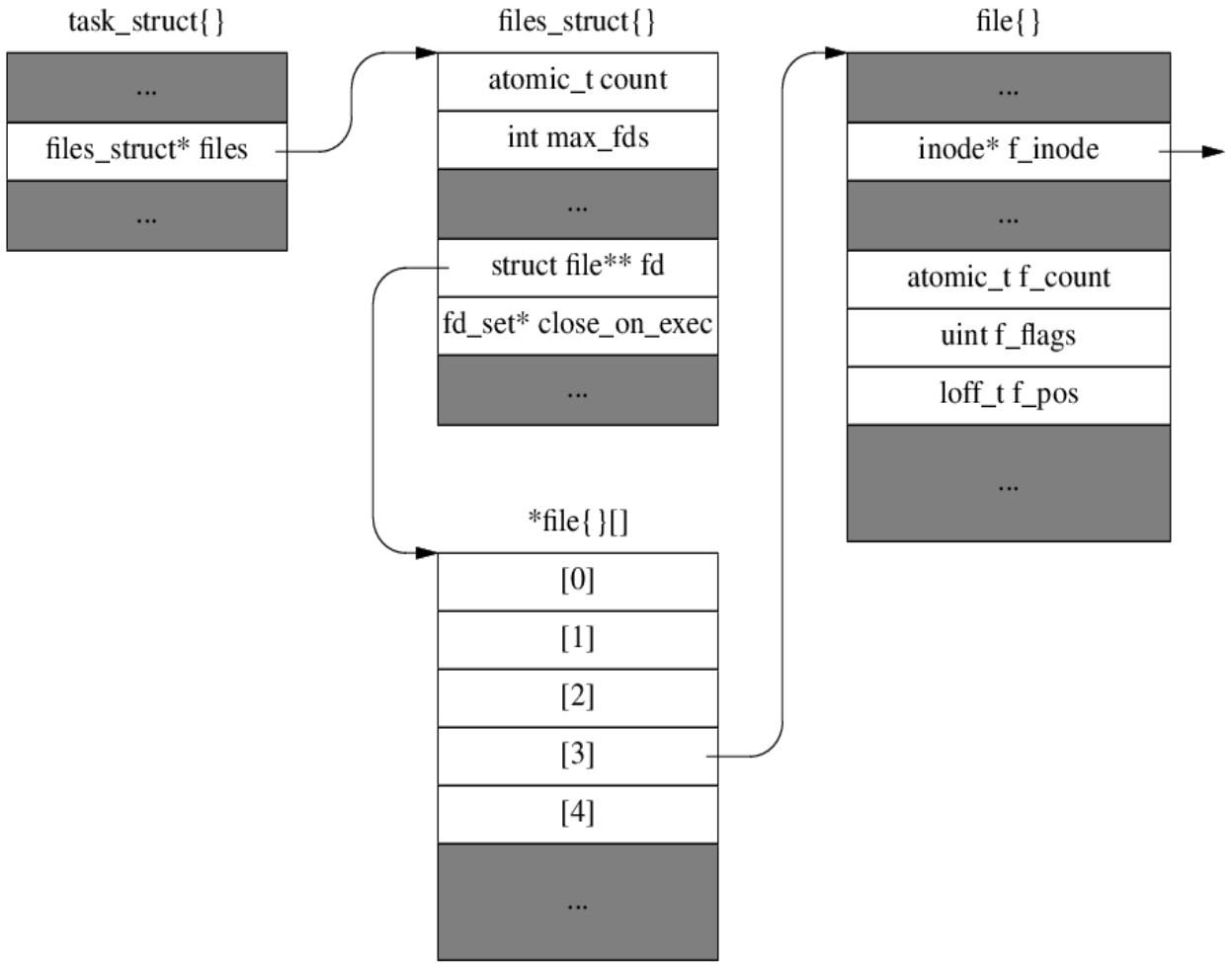
/ include / linux / fs.h

```
605
606  /*
607   * Keep mostly read-only and often accessed (especially for
608   * the RCU path lookup and 'stat' data) fields at the beginning
609   * of the 'struct inode'
610   */
611  struct inode {
612          umode_t                 i_mode;
613          unsigned short          i_opflags;
614          kuid_t                  i_uid;
615          kgid_t                  i_gid;
616          unsigned int            i_flags;
617
618  #ifdef CONFIG_FS_POSIX_ACL
619          struct posix_acl        *i_acl;
620          struct posix_acl        *i_default_acl;
621  #endif
622
623          const struct inode_operations   *i_op;
624          struct super_block      *i_sb;
625          struct address_space    *i_mapping;
```

https://elixir.bootlin.com/linux/v5.13/source/include/linux/fs.h#L611

File Object





From: https://chenshuo.com/notes/kernel/fdt.png

File Object - VFS APIs & Ext4

```
2069 struct file_operations {
2070         struct module *owner;
2071         loff_t (*llseek) (struct file *, loff_t, int);
2072         ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
2073         ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
2074         ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
2075         ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
2076         int (*iopoll)(struct kiocb *kiocb, struct io_comp_batch *,
2077                       unsigned int flags);
2078         int (*iterate) (struct file *, struct dir_context *);
2079         int (*iterate_shared) (struct file *, struct dir_context *);
2080         __poll_t (*poll) (struct file *, struct poll_table_struct *);
2081         long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
2082         long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
2083         int (*mmap) (struct file *, struct vm_area_struct *);
2084         unsigned long mmap_supported_flags;
2085         int (*open) (struct inode *, struct file *);
2086         int (*flush) (struct file *, fl_owner_t id);
2087         int (*release) (struct inode *, struct file *);
2088         int (*fsync) (struct file *, loff_t, loff_t, int datasync);
2089         int (*fasync) (int, struct file *, int);
2090         int (*lock) (struct file *, int, struct file_lock *);
2091         ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
2092         unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
2093         int (*check_flags)(int);
2094         int (*flock) (struct file *, int, struct file_lock *);
2095         ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
2096         ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
2097         int (*setlease)(struct file *, long, struct file_lock **, void **);
2098         long (*fallocate)(struct file *file, int mode, loff_t offset,
2099                           loff_t len);
2100         void (*show_fdinfo)(struct seq_file *m, struct file *f);
2101 #ifndef CONFIG_MMU
2102         unsigned (*mmap_capabilities)(struct file *);
2103 #endif
2104         ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
2105                         loff_t, size_t, unsigned int);
2106         loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
2107                                    struct file *file_out, loff_t pos_out,
2108                                    loff_t len, unsigned int remap_flags);
2109         int (*fadvise)(struct file *, loff_t, loff_t, int);
2110 } __randomize_layout;
2111
```

From: https://elixir.bootlin.com/linux/latest/source/include/linux/fs.h#L2069

```
915 const struct file_operations ext4_file_operations = {
916         .llseek         = ext4_llseek,
917         .read_iter      = ext4_file_read_iter,
918         .write_iter     = ext4_file_write_iter,
919         .iopoll         = iocb_bio_iopoll,
920         .unlocked_ioctl = ext4_ioctl,
921 #ifdef CONFIG_COMPAT
922         .compat_ioctl   = ext4_compat_ioctl,
923 #endif
924         .mmap           = ext4_file_mmap,
925         .mmap_supported_flags = MAP_SYNC,
926         .open           = ext4_file_open,
927         .release        = ext4_release_file,
928         .fsync          = ext4_sync_file,
929         .get_unmapped_area = thp_get_unmapped_area,
930         .splice_read    = generic_file_splice_read,
931         .splice_write   = iter_file_splice_write,
932         .fallocate      = ext4_fallocate,
933 };
```

From: https://elixir.bootlin.com/linux/latest/source/fs/ext4/file.c

Debugfs – Prowl around a FS

DEBUGFS(8) System Manager's Manual DEBUGFS(8)

NAME

debugfs - ext2/ext3/ext4 file system debugger

SYNOPSIS

debugfs [**-DVwcin**] [**-b** blocksize] [**-s** superblock] [**-f** cmd_file] [**-R** request] [**-d** data_source_device] [**-z** undo_file] [device]

DESCRIPTION

The **debugfs** program is an interactive file system debugger. It can be used to examine and change the state of an ext2, ext3, or ext4 file system.

device is a block device (e.g., /dev/sdXX) or a file containing the file system.

OPTIONS

-w Specifies that the file system should be opened in read-write mode. Without this option, the file system is opened in read-only mode.

-n Disables metadata checksum verification. This should only be used if you believe the metadata to be correct despite the complaints of e2fsprogs.

-c Specifies that the file system should be opened in catastrophic mode, in which the inode and group bitmaps are not read initially. This can be useful for filesystems with significant corruption, but because of this, catastrophic mode forces the filesystem to be opened read-only.

-i Specifies that device represents an ext2 image file created by the **e2image** program. Since the ext2 image file only contains the superblock, block group descriptor, block and inode allocation bitmaps, and the inode table, many **debugfs** commands will not function properly. **Warning:** no safety checks are in place, and **debugfs** may fail in interesting ways if commands such as ls, dump, etc. are tried without specifying the data_source_device using the -d option. **debugfs** is a debugging tool. It has rough edges!

-d data_source_device
Used with the -i option, specifies that data_source_device should be used when reading blocks not found in the ext2 image file. This includes data, directory, and indirect blocks.

-b blocksize
Forces the use of the given block size (in bytes) for the file system, rather than detecting the correct block size automatically. (This option is rarely needed; it is used primarily when the file system is extremely badly damaged/corrupted.)

-s superblock
Causes the file system superblock to be read from the given block number, instead of using the primary superblock (located at an offset of 1024 bytes from the beginning of the filesystem). If you specify the -s option, you must also provide the blocksize of the filesystem via the -b option. (This option is rarely needed; it is used primarily when the file system is extremely badly damaged/corrupted.)

Debugfs – Prowl around a FS

```
zegang@zegang-VirtualBox:~/1300-simple-fs$ debugfs mydisk
debugfs 1.45.5 (07-Jan-2020)
debugfs:  stat /home/zegang/1.txt
```

```
Inode: 14   Type: regular    Mode:  0664   Flags: 0x80000
Generation: 982170681    Version: 0x00000001
User:  1000   Group:  1000   Size: 2
File ACL: 0
Links: 1   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x6217565c -- Thu Feb 24 17:56:44 2022
atime: 0x62175665 -- Thu Feb 24 17:56:53 2022
mtime: 0x6217565c -- Thu Feb 24 17:56:44 2022
Inode checksum: 0x0000c357
EXTENTS:
(0):5147
(END)
```

Debugfs – Inode

```
780  /*
781   * Structure of an inode on the disk
782   */
783  struct ext4_inode {
784          __le16  i_mode;         /* File mode */
785          __le16  i_uid;          /* Low 16 bits of Owner Uid */
786          __le32  i_size_lo;      /* Size in bytes */
787          __le32  i_atime;        /* Access time */
788          __le32  i_ctime;        /* Inode Change time */
789          __le32  i_mtime;        /* Modification time */
790          __le32  i_dtime;        /* Deletion Time */
791          __le16  i_gid;          /* Low 16 bits of Group Id */
792          __le16  i_links_count;  /* Links count */
793          __le32  i_blocks_lo;    /* Blocks count */
794          __le32  i_flags;        /* File flags */
795          union {
```

```
debugfs:  id /home/zegang/1.txt
0000  b481 e803 0200 0000 6556 1762 5c56 1762  ........eV.b\V.b
0020  5c56 1762 0000 0000 e803 0100 0800 0000  \V.b............
0040  0000 0800 0100 0000 0af3 0100 0400 0000  ................
0060  0000 0000 0000 0000 0100 0000 1b14 0000  ................
0100  0000 0000 0000 0000 0000 0000 0000 0000  ................
*
0140  0000 0000 39bc 8a3a 0000 0000 0000 0000  ....9..:........
0160  0000 0000 0000 0000 0000 0000 57c3 0000  ............W...
```

```
Inode: 14   Type: regular    Mode:  0664   Flags: 0x80000
Generation: 982170681    Version: 0x00000001
User:  1000   Group:  1000   Size: 2
File ACL: 0
Links: 1   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x6217565c -- Thu Feb 24 17:56:44 2022
atime: 0x62175665 -- Thu Feb 24 17:56:53 2022
mtime: 0x6217565c -- Thu Feb 24 17:56:44 2022
Inode checksum: 0x0000c357
EXTENTS:
(0):5147
(END)
```

```
zegang@zegang-VirtualBox:~/1300-simple-fs$ python3 -c "print(hex(1000))"
0x3e8
zegang@zegang-VirtualBox:~/1300-simple-fs$ python3 -c "print(hex(0o664))"
0x1b4
```

Syscalls

```
zegang@zegang-VirtualBox:~/1300-simple-fs/mydisk_mp/home/zegang$ strace -e trace=file ls -lh 1.txt
execve("/usr/bin/ls", ["ls", "-lh", "1.txt"], 0x7ffc60509530 /* 56 vars */) = 0
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpcre2-8.so.0", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
statfs("/sys/fs/selinux", 0x7ffe2bd3cb00) = -1 ENOENT (No such file or directory)
statfs("/selinux", 0x7ffe2bd3cb00)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/proc/filesystems", O_RDONLY|O_CLOEXEC) = 3
access("/etc/selinux/config", F_OK)     = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/share/locale/locale.alias", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/share/locale/zh_CN.UTF-8/LC_TIME/coreutils.mo", O_RDONLY) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/usr/share/locale/zh_CN.utf8/LC_TIME/coreutils.mo", O_RDONLY) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/usr/share/locale/zh_CN/LC_TIME/coreutils.mo", O_RDONLY) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/usr/share/locale/zh.UTF-8/LC_TIME/coreutils.mo", O_RDONLY) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/usr/share/locale/zh.utf8/LC_TIME/coreutils.mo", O_RDONLY) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/usr/share/locale/zh/LC_TIME/coreutils.mo", O_RDONLY) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache", O_RDONLY) = 3
lstat("1.txt", {st_mode=S_IFREG|0664, st_size=2, ...}) = 0
lgetxattr("1.txt", "security.selinux", 0x55af7a7be390, 255) = -1 ENODATA (No data available)
getxattr("1.txt", "system.posix_acl_access", NULL, 0) = -1 ENODATA (No data available)
openat(AT_FDCWD, "/etc/nsswitch.conf", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libnss_files.so.2", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/etc/passwd", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/etc/group", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/etc/localtime", O_RDONLY|O_CLOEXEC) = 3
-rw-rw-r-- 1 zegang zegang 2 2月  24 17:56 1.txt
+++ exited with 0 +++
```

Syscalls: POSIX

INDEX

lstat
Search..

[Alphabetic | Topic | Word Search]

Select a Volume:
[*Base Definitions* | *System Interfaces* | *Shell & Utilities* | *Rationale*]

[Frontmatter]

[Main Index]

Base Definitions

1. Introduction
2. Conformance
3. Definitions
4. General Concepts
5. File Format Notation
6. Character Set
7. Locale
8. Environment Variables
9. Regular Expressions
10. Directory Structure and Devices
11. General Terminal Interface
12. Utility Conventions
13. Headers

NAME

fstatat, lstat, stat - get file status

SYNOPSIS

```
[OH] #include <fcntl.h>
#include <sys/stat.h>

int fstatat(int fd, const char *restrict path,
        struct stat *restrict buf, int flag);
int lstat(const char *restrict path, struct stat *restrict buf);
int stat(const char *restrict path, struct stat *restrict buf);
```

DESCRIPTION

The *stat*() function shall obtain information about the named file and write it to the area pointed to by the *buf* argument. The *path* argument points to a pathname naming a file. Read, write, or execute permission of the named file is not required. An implementation that provides additional or alternate file access control mechanisms may, under implementation-defined conditions, cause *stat*() to fail. In particular, the system may deny the existence of the file specified by *path*.

If the named file is a symbolic link, the *stat*() function shall continue pathname resolution using the contents of the symbolic link, and shall return information pertaining to the resulting file if the file exists.

The *buf* argument is a pointer to a **stat** structure, as defined in the *<sys/stat.h>* header, into which information is placed concerning the file.

The *stat*() function shall update any time-related fields (as described in XBD *File Times Update*), before writing into the **stat** structure.

[SHM] If the named file is a shared memory object, the implementation shall update in the **stat** structure pointed to by the *buf* argument the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid. The implementation may update other fields and flags. [/SHM]

[TYM] If the named file is a typed memory object, the implementation shall update in the **stat** structure pointed to by the *buf* argument the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid. The implementation may update other fields and flags. [/TYM]

For all other file types defined in this volume of POSIX.1-2017, the structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atim*, *st_ctim*, and *st_mtim* shall have meaningful values and the value of the member *st_nlink* shall be set to the number of links to the file.

The *lstat*() function shall be equivalent to *stat*(), except when *path* refers to a symbolic link. In that case *lstat*() shall return information about the link, while *stat*() shall return information about the file the link references.

For symbolic links, the *st_mode* member shall contain meaningful information when used with the file type macros. The file mode bits in *st_mode* are unspecified. The structure members *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atim*, *st_ctim*, and *st_mtim* shall have meaningful values and the value of the *st_nlink* member shall be set to the number of (hard) links to the symbolic link. The value of the *st_size* member shall be set to the length of the pathname contained in the symbolic link not including any terminating null byte.

The *fstatat*() function shall be equivalent to the *stat*() or *lstat*() function, depending on the value of *flag* (see below), except in the case where *path* specifies a relative path. In this case the status shall be retrieved from a file relative to the directory associated with the file descriptor *fd* instead of the current working directory. If the access mode of the open file description associated with the file descriptor is not O_SEARCH, the function shall check whether directory searches are permitted using the current permissions of the directory underlying the file descriptor. If the access mode is O_SEARCH, the function shall not perform the check.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in *<fcntl.h>*:

AT_SYMLINK_NOFOLLOW
    If *path* names a symbolic link, the status of the symbolic link is returned.

If *fstatat*() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory shall be used and the behavior shall be identical to a call to *stat*() or *lstat*() respectively, depending on whether or not the AT_SYMLINK_NOFOLLOW bit is set in *flag*.

RETURN VALUE

Upon successful completion, these functions shall return 0. Otherwise, these functions shall return -1 and set *errno* to indicate the error.

ERRORS

These functions shall fail if:

[EACCES]
    Search permission is denied for a component of the path prefix.
[EIO]
    An error occurred while reading from the file system.
[ELOOP]
    A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]
    The length of a component of a pathname is longer than {NAME_MAX}.
[ENOENT]
    A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]
    A component of the path prefix names an existing file that is neither a directory nor a symbolic link to a directory, or the *path* argument contains at least one non- <slash> character and ends with one or more trailing <slash> characters and the last pathname component names an existing file that is neither a directory nor a symbolic link to a directory.

https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/

Syscalls

```
zegang@zegang-VirtualBox:~/1300-simple-fs/mydisk_mp/home/zegang$ man 2 lstat | head -15
STAT(2)                                                    Linux Programmer's Manual                                                    STAT(2)

NAME
       stat, fstat, lstat, fstatat - get file status

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <unistd.h>

       int stat(const char *pathname, struct stat *statbuf);
       int fstat(int fd, struct stat *statbuf);
       int lstat(const char *pathname, struct stat *statbuf);

       #include <fcntl.h>           /* Definition of AT * constants */
```

```
1239  /* obsolete: fs/stat.c */
1240  asmlinkage long sys_stat(const char __user *filename,
1241                           struct __old_kernel_stat __user *statbuf);
1242  asmlinkage long sys_lstat(const char __user *filename,
1243                           struct __old_kernel_stat __user *statbuf);
1244  asmlinkage long sys_fstat(unsigned int fd,
1245                           struct __old_kernel_stat __user *statbuf);
1246  asmlinkage long sys_readlink(const char __user *path,
1247                               char __user *buf, int bufsiz);
```

https://elixir.bootlin.com/linux/latest/source/include/linux/syscalls.h#L1242

```
303  SYSCALL_DEFINE2(lstat, const char __user *, filename,
304                  struct __old_kernel_stat __user *, statbuf)
305  {
306          struct kstat stat;
307          int error;
308
309          error = vfs_lstat(filename, &stat);
310          if (error)
311                  return error;
312
313          return cp_old_stat(&stat, statbuf);
314  }
```

https://elixir.bootlin.com/linux/latest/source/fs/stat.c#L303

Syscalls

```
3354  }
3355  static inline int vfs_lstat(const char __user *name, struct kstat *stat)
3356  {
3357          return vfs_fstatat(AT_FDCWD, name, stat, AT_SYMLINK_NOFOLLOW);
3358  }
```

https://elixir.bootlin.com/linux/latest/source/include/linux/fs.h#L3355

```
239
240  int vfs_fstatat(int dfd, const char __user *filename,
241                  struct kstat *stat, int flags)
242  {
243          return vfs_statx(dfd, filename, flags | AT_NO_AUTOMOUNT,
244                           stat, STATX_BASIC_STATS);
245  }
```

https://elixir.bootlin.com/linux/latest/source/fs/stat.c#L240

```
92   */
93   int vfs_getattr_nosec(const struct path *path, struct kstat *stat,
94                         u32 request_mask, unsigned int query_flags)
95   {
96           struct user_namespace *mnt_userns;
97           struct inode *inode = d_backing_inode(path->dentry);
98
99           memset(stat, 0, sizeof(*stat));
100          stat->result_mask |= STATX_BASIC_STATS;
101          query_flags &= AT_STATX_SYNC_TYPE;
102
103          /* allow the fs to override these if it really wants to */
104          /* SB_NOATIME means filesystem supplies dummy atime value */
105          if (inode->i_sb->s_flags & SB_NOATIME)
106                  stat->result_mask &= ~STATX_ATIME;
107
108          /*
109           * Note: If you add another clause to set an attribute flag, please
110           * update attributes_mask below.
111           */
112          if (IS_AUTOMOUNT(inode))
113                  stat->attributes |= STATX_ATTR_AUTOMOUNT;
114
115          if (IS_DAX(inode))
116                  stat->attributes |= STATX_ATTR_DAX;
117
118          stat->attributes_mask |= (STATX_ATTR_AUTOMOUNT |
119                                    STATX_ATTR_DAX);
120
121          mnt_userns = mnt_user_ns(path->mnt);
122          if (inode->i_op->getattr)
123                  return inode->i_op->getattr(mnt_userns, path, stat,
124                                              request_mask, query_flags);
125
126          generic_fillattr(mnt_userns, inode, stat);
127          return 0;
128  }
```

https://elixir.bootlin.com/linux/latest/source/fs/stat.c#L123

References

- https://en.wikipedia.org/wiki/Ext4
- https://phoenixnap.com/kb/linux-file-system
- http://www.linfo.org/superblock
- https://elixir.bootlin.com/linux/latest/source/fs/ext4/file.c
- https://elixir.bootlin.com/linux/latest/source/fs/stat.c#L123

Homework

1. Why is there only one Block Group in the mydisk ext4 format ?
2. Could you make a format disk which has more than 1 block groups ?
3. Why are block numbers of block/inode bitmap 16 instead of 1 ?
4. Starting in ext4, there is a new feature called flexible block groups. How ext4 disk format if this feature enabled ?
5. Write a file with content “Hello 1300-simple-fs” and hexdump the file data on disk.
6. Show the data on disk layout of a big file.

```
zegang@zegang-VirtualBox:~$ mkdir 1300-simple-fs
zegang@zegang-VirtualBox:~$ cd 1300-simple-fs/
zegang@zegang-VirtualBox:~/1300-simple-fs$ touch mydisk
zegang@zegang-VirtualBox:~/1300-simple-fs$ truncate -s 128M mydisk
zegang@zegang-VirtualBox:~/1300-simple-fs$ ll -h mydisk
-rw-rw-r-- 1 zegang zegang 128M 2月  24 17:33 mydisk
```

Reference Paper:
1. Kevin D. Fairbanks, An analysis of Ext4 for digital forensics, Digital Investigation, Volume 9, Supplement, 2012, Pages S118-S130, ISSN 1742-2876, https://doi.org/10.1016/j.diin.2012.05.010.(https://www.sciencedirect.com/science/article/pii/S1742287612000357)

Homework

Background

Ten Inodes FS (TIF)

This demo FUSE filesystem is just a easy start point to felling some basic VFS objects.

This FS is designed for only limited number of files/inodes scenario, 10.

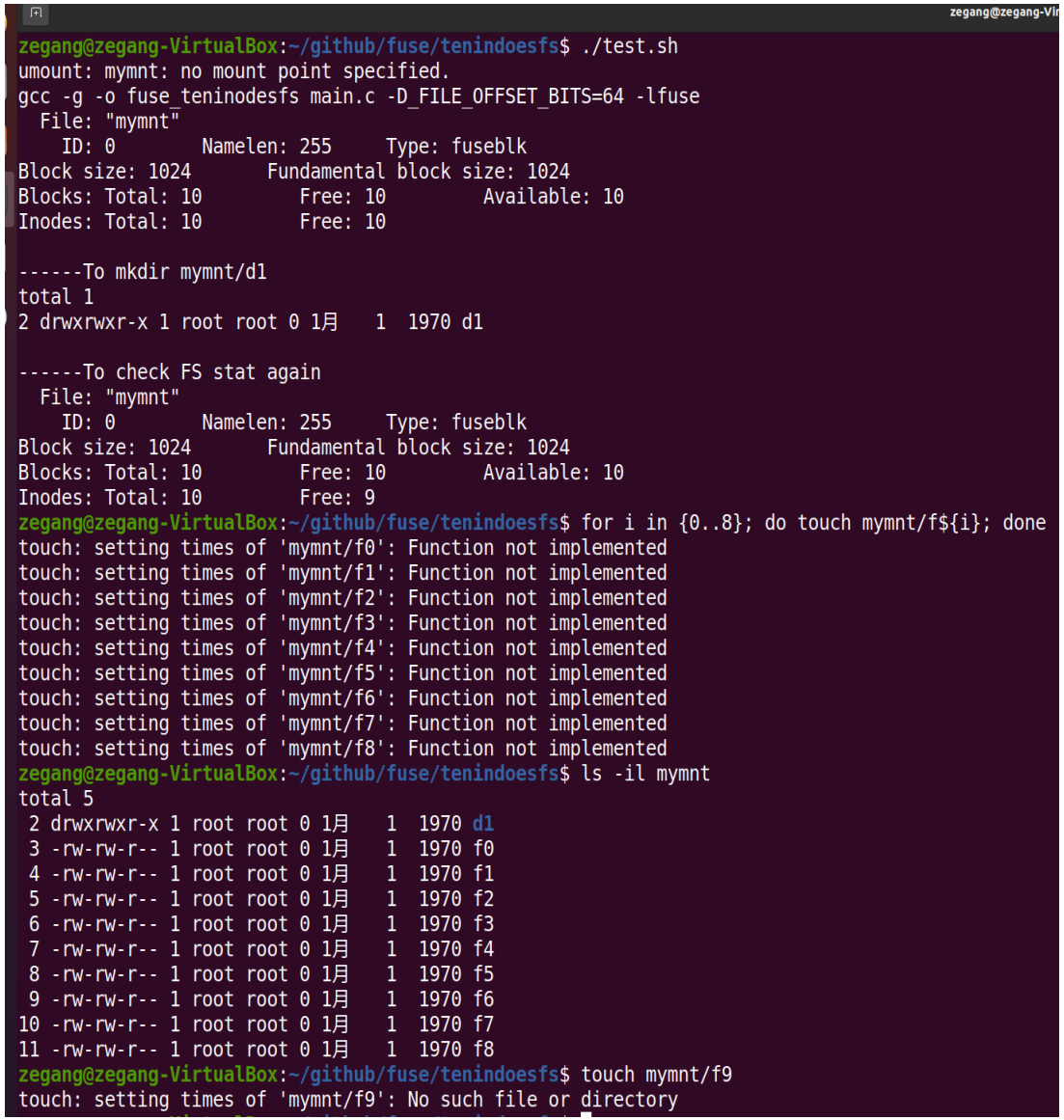
HOW TO RUN

After install libfuse librarys, run "./test.sh".

TAKE IT AWAY

Ops, mkdir/ls, on the mounted FS and check logs to watch VFS objects create/filling...

This demo FUSE filesystem is in-memory only and some implementation differes from VFS standars. Such as, dentry, dcache. You may change it for yourself.



Homework

1. Allocated 1 data block for 1 file.
2. Write data into the file data block.
3. Bind the file data block with file inode.
4. Any what your want…