Gabriele **Proietti Mattia**

Advanced Operating Systems and Virtualization

[8] Virtual File System

Department of Computer, Control and Management Engineering "A. Ruberti", Sapienza University of Rome

gpm.name · proiettimattia@diag.uniroma1.it

A.Y. 2020/2021 · v2

Outline

1. **Introduction**
2. **The Common File Model**
    1. Operations
3. **Pathname Lookup**
4. **Files**
5. **The /proc filesystem**
6. **The /sys filesystem**
7. **Device Management**
    1. Char Devices
    2. Block Devices
    3. Devices and VFS
    4. Classes
    5. udev

8.1

8. Virtual Filesystem

Introduction

DIAG

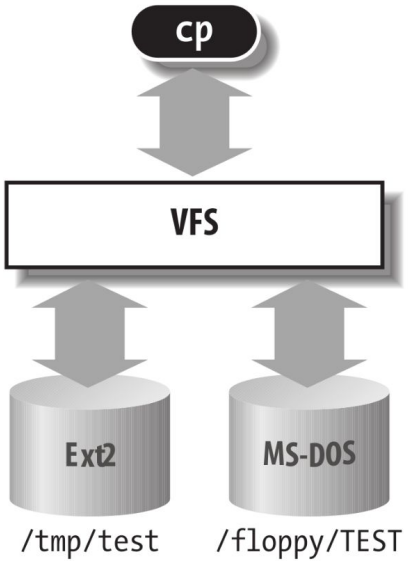
Introduction

The VFS is a software layer which abstracts the actual implementation of the devices and/or the organization of files on a storage system. The VFS exposes a uniform interface to userspace applications.

The main roles of the virtual filesytem are:

- keeping track of available filesystem types;
- associating (and de-associating) devices with instances of the appropriate filesystem.
- do any reasonable generic processing for operations involving files.
- when filesystem-specific operations become necessary, vector them to the filesystem in charge of the file, directory, or inode in question.

Introduction




```
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);
```

(b)

Figure 12-1. VFS role in a simple file copy operation

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

Supported File Systems

The filesystems supported by the VFS can be grouped in:

- **Disk-based Filesystems**

  They manage memory in a disk or in some other device which emulates a disk (e.g. USB disk). Some of the well-known FS are:

  - Linux EXT2/3/4, from Oracle also BTRFS
  - Windows MS-DOS, VFAT, NTFS, ExFAT
  - CD-ROM FS like UDF
  - Other proprietary like Apple HFS, HFS+, APFS, IBM HPFS

- **Network Filesystems**

  They allows easy access to file belonging to other networked PCs (e.g. NFS, CIFS)

- **Special Filesystems**

  They do not manage a disk space (e.g. `/proc` or `/sys`)

File System Representation

The VFS representation has a two fold nature, one in RAM and one on disk. In **RAM** we have a partial or full representation of the current structure and the content of the FS. On the **device** we have the full representation of of the current structure and the content of the FS but possibly outdated.

The data access and manipulation comprehends:

- a **FS-independent** part, that is the interface towards other subsystems within the kernel
- a **FS-dependent** part, that is the code for managing data in that particular filesystem

Connecting the two parts: any filesystem object that can be a directory, a device or a file is represented in RAM via specific data structures. Each data structure keeps a **reference** to the functions that talks directly to the device, if any. That reference is reached by means of a kernel API interface (like `read()`, `write()`, etc.). Function **pointers** are used to reference actual drivers' functions.

Everything is a file.

(*with some exceptions)

8.2

8. Virtual Filesystem

The Common File Model

DIAG

The Common File Model

The key idea behind the VFS is to introduce a common file model capable of representing all the possible filesystems. This means that each physical filesystem implementation **must translate** its physical organization into the VFS's common file model.

For example, in the Common File Model each directory is a file which contains a list of files and other directories, however a FAT (File Allocation Table) filesystem stores the position of each file in a tree and directories are not files. In order to adhere to the VFS model the FAT driver must create on the fly a file object, but this exists only in memory.

The Common File Model

The Common File Model consists of the following "object" types:

- **superblock**

  Stores the information concerning a mounted filesystem, this object corresponds to a filesystem control block stored on disk

- **inode**

  Stores general information about a specific file, this corresponds to to a file control block stored on disk, each inode has a unique number associated to it

- **file**

  Stores the information about the interaction between an open file and a process, this exists only in kernel memory when a process opens a file

- **dentry**

  Stores the information about the linking of a directory entry with the corresponding file, each FS stores this information in its own particular way.

The Common File Model

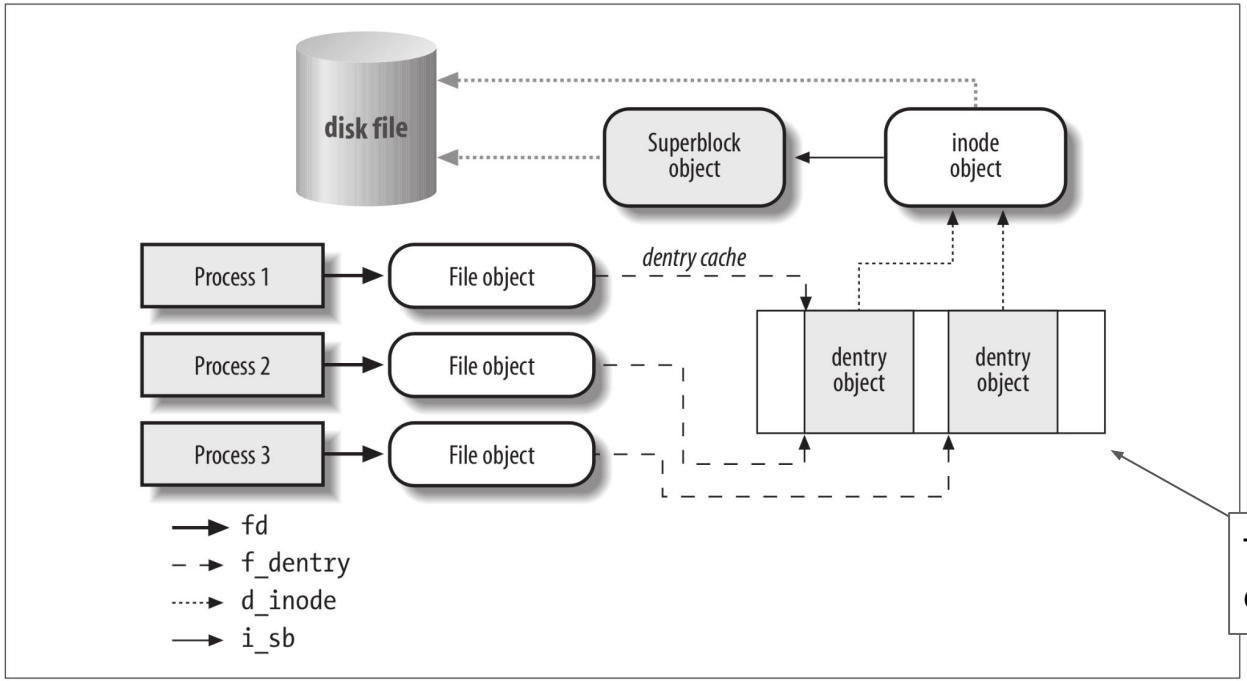


Figure 12-2. Interaction between processes and VFS objects

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

The Common File Model

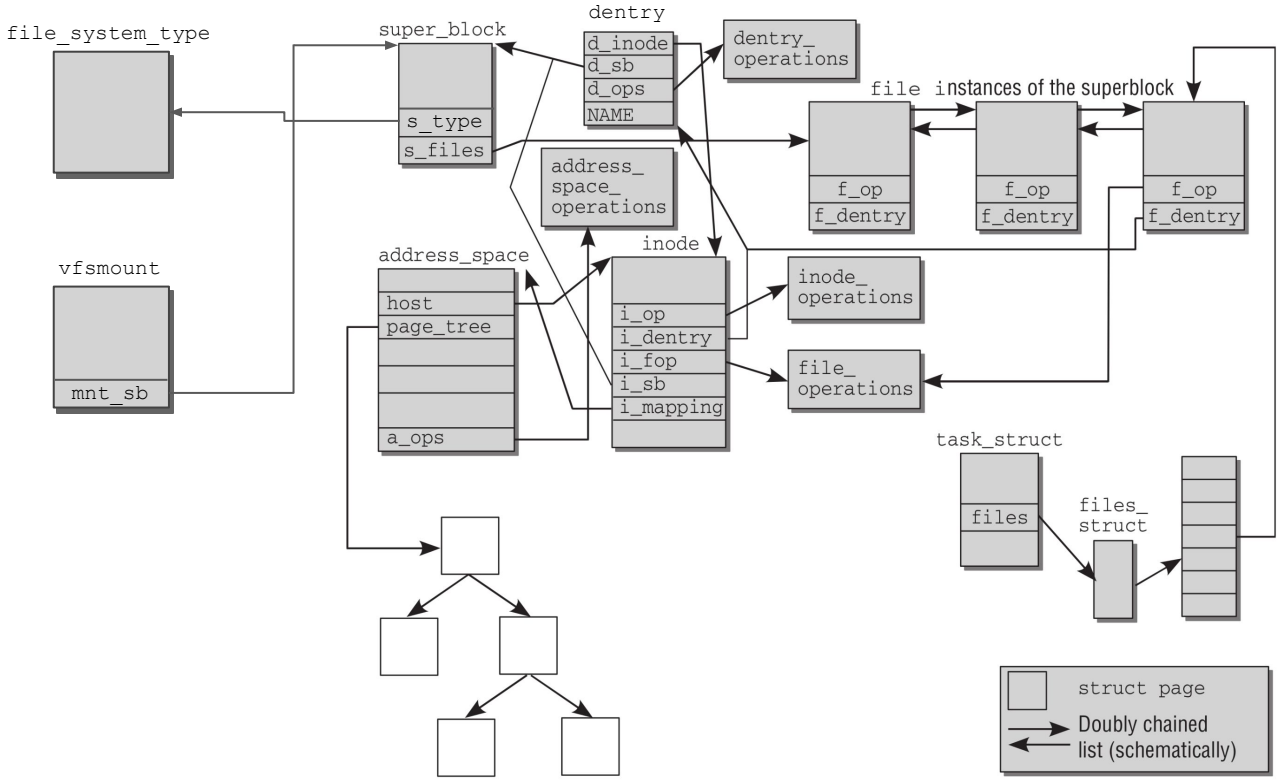


Figure 8-3: Interplay of the VFS components.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

Filesystem Types

V5.11

The file_system_type structure describes a file system (it is defined in `include/linux/fs.h`), it keeps information related to:

- the file system name
- a pointer to a function to be executed upon mounting the file system (superblock-read)

```
2226  struct file_system_type {
2227          const char *name;
2228          int fs_flags;
2236          int (*init_fs_context)(struct fs_context *);
2237          const struct fs_parameter_spec *parameters;
2238          struct dentry *(*mount) (struct file_system_type *, int,
2239                          const char *, void *);
2240          void (*kill_sb) (struct super_block *);
2241          struct module *owner;
2242          struct file_system_type * next;
2243          struct hlist_head fs_supers;
2253  };
```

Filesystem Types

ramfs

Ramfs is a very simple filesystem that exports Linux's disk caching mechanisms (the page cache and dentry cache) as a dynamically resizable RAM-based filesystem.

With ramfs, **there is no backing store**. Files written into ramfs allocate dentries and page cache as usual, but there's nowhere to write them to.

Ramfs can eat up all the available memory:

- tmpfs is a derivative, with size limits
- only root should be given access to ramfs

Filesystem Types

rootfs

Rootfs is a special instance of ramfs (or tmpfs, if that's enabled), which is always present in 2.6 systems.

It provides an empty root directory during kernel boot. Rootfs cannot be unmounted and this has the same idea behind the fact that init process cannot be killed.

During kernel boot, another (actual) filesystem is mounted over rootfs (remember initramfs/initrd).

File System Mounting

In most traditional Unix-like kernel, each filesystem can be mounted once, the command used is for instance

mount -t ext4 /dev/sda1 /mnt

However in Linux it is possible to mount the same filesystem *n* times, this means that its root directory can be accessed through *n* mount points. This means that each mount point (represented by the struct vfsmount) will point to the same superblock.

Mounted filesystems form a **hierarchy**: the mount point of a filesystem might be the directory of a second filesystem, which in turn is already mounted over a third filesystem and so on.

vfsmount

v2.6

```
55  struct vfsmount {
56          struct list_head mnt_hash;
57          struct vfsmount *mnt_parent;    /* fs we are mounted on */
58          struct dentry *mnt_mountpoint;  /* dentry of mountpoint */
59          struct dentry *mnt_root;        /* root of the mounted tree */
60          struct super_block *mnt_sb;     /* pointer to superblock */
61  #ifdef CONFIG_SMP
62          struct mnt_pcp __percpu *mnt_pcp;
63          atomic_t mnt_longterm;          /* how many of the refs are longterm */
64  #else
65          int mnt_count;
66          int mnt_writers;
67  #endif
68          struct list_head mnt_mounts;    /* list of children, anchored here */
69          struct list_head mnt_child;     /* and going through their mnt_child */
70          int mnt_flags;
71          /* 4 bytes hole on 64bits arches without fsnotify */
72  #ifdef CONFIG_FSNOTIFY
73          __u32 mnt_fsnotify_mask;
74          struct hlist_head mnt_fsnotify_marks;
75  #endif
76          const char *mnt_devname;        /* Name of device e.g. /dev/dsk/hda1 */
77          struct list_head mnt_list;
78          struct list_head mnt_expire;    /* link in fs-specific expiry list */
79          struct list_head mnt_share;     /* circular list of shared mounts */
80          struct list_head mnt_slave_list;/* list of slave mounts */
81          struct list_head mnt_slave;     /* slave list entry */
82          struct vfsmount *mnt_master;    /* slave is on master->mnt_slave_list */
83          struct mnt_namespace *mnt_ns;   /* containing namespace */
84          int mnt_id;                     /* mount identifier */
85          int mnt_group_id;               /* peer group identifier */
86          int mnt_expiry_mark;            /* true if marked for expiry */
87          int mnt_pinned;
88          int mnt_ghosts;
89  };
```

https://elixir.bootlin.com/linux/v2.6.39.4/source/include/linux/mount.h#L55

superblock

v2.6

```
1360  struct super_block {
1361          struct list_head        s_list;         /* Keep this first */
1362          dev_t                   s_dev;          /* search index; _not_ kdev_t */
1363          unsigned char           s_dirt;
1364          unsigned char           s_blocksize_bits;
1365          unsigned long           s_blocksize;
1366          loff_t                  s_maxbytes;     /* Max file size */
1367          struct file_system_type *s_type;
1368          const struct super_operations   *s_op;
1369          const struct dquot_operations   *dq_op;
1370          const struct quotactl_ops       *s_qcop;
1371          const struct export_operations *s_export_op;
1372          unsigned long           s_flags;
1373          unsigned long           s_magic;
1374          struct dentry           *s_root;
1375          struct rw_semaphore     s_umount;
1376          struct mutex            s_lock;
1377          int                     s_count;
1378          atomic_t                s_active;
1379  #ifdef CONFIG_SECURITY
1380          void                    *s_security;
1381  #endif
1382          const struct xattr_handler **s_xattr;
1383
1384          struct list_head        s_inodes;       /* all inodes */
1385          struct hlist_bl_head    s_anon;         /* anonymous dentries for (nfs) exporting */
1386  #ifdef CONFIG_SMP
1387          struct list_head __percpu *s_files;
1388  #else
1389          struct list_head        s_files;
1390  #endif
1391          /* s_dentry_lru, s_nr_dentry_unused protected by dcache.c lru locks */
1392          struct list_head        s_dentry_lru;   /* unused dentry lru */
1393          int                     s_nr_dentry_unused;     /* # of dentry on lru */
1394
1395          struct block_device     *s_bdev;
```

https://elixir.bootlin.com/linux/v2.6.39.4/source/include/linux/fs.h#L1360

dentry

v2.6

```
116  struct dentry {
117          /* RCU lookup touched fields */
118          unsigned int d_flags;           /* protected by d_lock */
119          seqcount_t d_seq;               /* per dentry seqlock */
120          struct hlist_bl_node d_hash;    /* lookup hash list */
121          struct dentry *d_parent;        /* parent directory */
122          struct qstr d_name;
123          struct inode *d_inode;          /* Where the name belongs to - NULL is
124                                           * negative */
125          unsigned char d_iname[DNAME_INLINE_LEN];        /* small names */
126
127          /* Ref lookup also touches following */
128          unsigned int d_count;           /* protected by d_lock */
129          spinlock_t d_lock;              /* per dentry lock */
130          const struct dentry_operations *d_op;
131          struct super_block *d_sb;       /* The root of the dentry tree */
132          unsigned long d_time;           /* used by d_revalidate */
133          void *d_fsdata;                 /* fs-specific data */
134
135          struct list_head d_lru;         /* LRU list */
136          /*
137           * d_child and d_rcu can share memory
138           */
139          union {
140                  struct list_head d_child;       /* child of parent list */
141                  struct rcu_head d_rcu;
142          } d_u;
143          struct list_head d_subdirs;     /* our children */
144          struct list_head d_alias;       /* inode alias list */
145  };
```

https://elixir.bootlin.com/linux/v2.6.39.4/source/include/linux/dcache.h#L116

The kernel creates a dentry for every directory. When a path like `/tmp/test` is resolved a dentry is created for "`/`", "`tmp`" and "`test`". dentries have no corresponding image on disk and hence there is no field in the structure which specifies that the object has been modified. The state of each dentry can be:

- **Free,** not used no inode
- **Unused,** not used by inode
- **In Use,** used
- **Negative,** the inode does not exist

inode

v2.6

The state can be:

- I_DIRTY_SYNC
- I_DIRTY_DATASYNC
- I_DIRTY_PAGES
- I_LOCK
- I_FREEING
- I_CLEAR
- I_NEW

```
735  struct inode {
736          /* RCU path lookup touches following: */
737          umode_t                 i_mode;
738          uid_t                   i_uid;
739          gid_t                   i_gid;
740          const struct inode_operations   *i_op;
741          struct super_block      *i_sb;
742
743          spinlock_t              i_lock; /* i_blocks, i_bytes, maybe i_size */
744          unsigned int            i_flags;
745          struct mutex            i_mutex;
746
747          unsigned long           i_state;
748          unsigned long           dirtied_when;   /* jiffies of first dirtying */
749
750          struct hlist_node       i_hash;
751          struct list_head        i_wb_list;      /* backing dev IO list */
752          struct list_head        i_lru;          /* inode LRU list */
753          struct list_head        i_sb_list;
754          union {
755                  struct list_head        i_dentry;
756                  struct rcu_head         i_rcu;
757          };
758          unsigned long           i_ino;
759          atomic_t                i_count;
760          unsigned int            i_nlink;
761          dev_t                   i_rdev;
762          unsigned int            i_blkbits;
763          u64                     i_version;
764          loff_t                  i_size;
765  #ifdef __NEED_I_SIZE_ORDERED
766          seqcount_t              i_size_seqcount;
767  #endif
768          struct timespec         i_atime;
769          struct timespec         i_mtime;
770          struct timespec         i_ctime;
771          blkcnt_t                i_blocks;
772          unsigned short          i_bytes;
773          struct rw_semaphore     i_alloc_sem;
774          const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
775          struct file_lock        *i_flock;
776          struct address_space    *i_mapping;
777          struct address_space    i_data;
```

https://elixir.bootlin.com/linux/v2.6.39.4/source/include/linux/fs.h#L735

inode

Each inode can always appear in one of the following circular **doubly linked lists**:

- list of valid **unused** inodes, they are mirroring on disk but they are not used by any process, they are not dirty and i_count is 0
- list of **in-use** inodes, they are mirroring on disk and used by some process, they are not dirty and i_count > 0
- list of **dirty** inodes

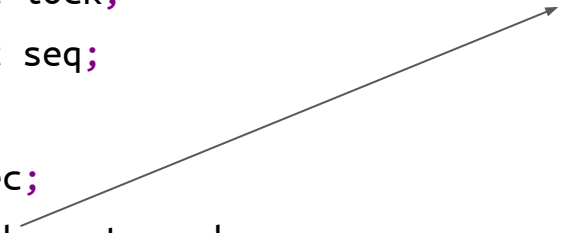
Moreover, inodes objects are also included in a **hash table** that speeds up the search of the inode object when the kernel knows both the inode number and the address of the superblock corresponding to the FS that includes the file.

VFS and PCB

In the PCB, `struct fs_struct *fs` points to information related to the current directory and the root directory for the associated process. `fs_struct` is defined in `include/fs_struct.h`

```
struct fs_struct {
     int users;
     spinlock_t lock;
     seqcount_t seq;
     int umask;
     int in_exec;
     struct path root, pwd;
} __randomize_layout;
```

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



8.2.1

8. Virtual Filesystem
2. The Common File Model

Operations

DIAG

Superblock operations

Super block operations are described by the struct super_operations. They:

- manage statistic of the file system
- create and manage i-nodes
- flush to the device updated information on the state of the file system

Some File Systems might not use some operations (think of File Systems in RAM). Functions to access statistics are invoked by system calls `statfs()` and `fstatfs()`.

super_operations

V5.11

```
1933  struct super_operations {
1934      struct inode *(*alloc_inode)(struct super_block *sb);
1935      void (*destroy_inode)(struct inode *);
1936      void (*free_inode)(struct inode *);
1937
1938      void (*dirty_inode) (struct inode *, int flags);
1939      int (*write_inode) (struct inode *, struct writeback_control *wbc);
1940      int (*drop_inode) (struct inode *);
1941      void (*evict_inode) (struct inode *);
1942      void (*put_super) (struct super_block *);
1943      int (*sync_fs)(struct super_block *sb, int wait);
1944      int (*freeze_super) (struct super_block *);
1945      int (*freeze_fs) (struct super_block *);
1946      int (*thaw_super) (struct super_block *);
1947      int (*unfreeze_fs) (struct super_block *);
1948      int (*statfs) (struct dentry *, struct kstatfs *);
1949      int (*remount_fs) (struct super_block *, int *, char *);
1950      void (*umount_begin) (struct super_block *);
1951
1952      int (*show_options)(struct seq_file *, struct dentry *);
1953      int (*show_devname)(struct seq_file *, struct dentry *);
1954      int (*show_path)(struct seq_file *, struct dentry *);
1955      int (*show_stats)(struct seq_file *, struct dentry *);
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/fs.h#L1933

ramfs example

V5.11

The `ramfs` filesystem is implemented in `fs/libfs.c`.

```
314  static const struct super_operations simple_super_operations = {
315          .statfs         = simple_statfs,
316  };
```

https://elixir.bootlin.com/linux/v5.11/source/fs/libfs.c#L314

```
40  int simple_statfs(struct dentry *dentry, struct kstatfs *buf)
41  {
42          buf->f_type = dentry->d_sb->s_magic;
43          buf->f_bsize = PAGE_SIZE;
44          buf->f_namelen = NAME_MAX;
45          return 0;
46  }
47  EXPORT_SYMBOL(simple_statfs);
```

https://elixir.bootlin.com/linux/v5.11/source/fs/libfs.c#L314

dentry_operations

V5.11

They specify non-default operations for manipulating d-entries. The table maintaining the associated function pointers is defined in `include/linux/dcache.h`. For the file system in RAM this structure is not used.

Removes the pointed inode

Removes the dentry when d_count is o

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

https://elixir.bootlin.com/linux/v5.11/source/include/linux/dcache.h#L136

inode_operations

```
1862  struct inode_operations {
1863          struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
1864          const char * (*get_link) (struct dentry *, struct inode *, struct delayed_call *);
1865          int (*permission) (struct inode *, int);
1866          struct posix_acl * (*get_acl)(struct inode *, int);
1867
1868          int (*readlink) (struct dentry *, char __user *,int);
1869
1870          int (*create) (struct inode *,struct dentry *, umode_t, bool);
1871          int (*link) (struct dentry *,struct inode *,struct dentry *);
1872          int (*unlink) (struct inode *,struct dentry *);
1873          int (*symlink) (struct inode *,struct dentry *,const char *);
1874          int (*mkdir) (struct inode *,struct dentry *,umode_t);
1875          int (*rmdir) (struct inode *,struct dentry *);
1876          int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
1877          int (*rename) (struct inode *, struct dentry *,
1878                         struct inode *, struct dentry *, unsigned int);
1879          int (*setattr) (struct dentry *, struct iattr *);
1880          int (*getattr) (const struct path *, struct kstat *, u32, unsigned int);
1881          ssize_t (*listxattr) (struct dentry *, char *, size_t);
1882          int (*fiemap)(struct inode *, struct fiemap_extent_info *, u64 start,
1883                        u64 len);
1884          int (*update_time)(struct inode *, struct timespec64 *, int);
1885          int (*atomic_open)(struct inode *, struct dentry *,
1886                             struct file *, unsigned open_flag,
1887                             umode_t create_mode);
1888          int (*tmpfile) (struct inode *, struct dentry *, umode_t);
1889          int (*set_acl)(struct inode *, struct posix_acl *, int);
1890  } ____cacheline_aligned;
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/fs.h#L1862

8.3

8. Virtual Filesystem

Pathname Lookup

DIAG

Pathname Lookup

When accessing VFS, the path to a file is used as the "key" to access a resource of interest. Internally, VFS uses inodes to represent a resource of interest. The Pathname lookup is the operation which derives an inode from the corresponding file pathname.

Pathname lookup **tokenizes** the string:

- the passed string is broken into a sequence of filenames
- everything must be a directory, except for the last component

During this procedure there are several aspects to take into account:

- filesystem mount points
- access rights
- symbolic links (and circular references)
- automount
- namespaces (more on this later)
- concurrency (while a process is navigating, other processes might make changes)

Functions

The main function for path name lookup are `vfs_path_lookup()`, `filename_lookup()` and `path_lookupat()`. The path walking is based on the `nameidata` data structure that is filled when the functions return.

```
502 struct nameidata {
503         struct path     path;
504         struct qstr     last;
505         struct path     root;
506         struct inode    *inode; /* path.dentry.d_inode */
507         unsigned int    flags;
508         unsigned        seq, m_seq, r_seq;
509         int             last_type;
510         unsigned        depth;
511         int             total_link_count;
512         struct saved {
513                 struct path link;
514                 struct delayed_call done;
515                 const char *name;
516                 unsigned seq;
517         } *stack, internal[EMBEDDED_LEVELS];
518         struct filename *name;
519         struct nameidata *saved;
520         unsigned        root_seq;
521         int             dfd;
522         kuid_t          dir_uid;
523         umode_t         dir_mode;
524 } __randomize_layout;
```

https://elixir.bootlin.com/linux/v5.11/source/fs/namei.c#L502

The function increments the refcount of dentry and inode

Flags are used for the lookup:

- LOOKUP_FOLLOW
- LOOKUP_DIRECTORY
- LOOKUP_CONTINUE
- LOOKUP_PARENT
- LOOKUP_NOALT
- LOOKUP_OPEN
- LOOKUP_CREATE
- LOOKUP_ACCESS

Current level of symlink navigation

Flags

Lookup flags drive the pathname resolution:

- `LOOKUP_FOLLOW`, if the last component is a symbolic link, interpret (follow) it
- `LOOKUP_DIRECTORY`, the last component must be a directory
- `LOOKUP_CONTINUE`, there are still filenames to be examined in the pathname
- `LOOKUP_PARENT`, look up the directory that includes the last component of the pathname
- `LOOKUP_NOALT`, do not consider the emulated root directory (useless in the 80x86 architecture)
- `LOOKUP_OPEN`, intent is to open a file
- `LOOKUP_CREATE`, intent is to create a file (if it doesn't exist)
- `LOOKUP_ACCESS`, intent is to check user's permission for a file

For further (and more comprehensive) description:

- `Documentation/filesystems/path-lookup.rst`
- `Documentation/filesystems/path-lookup.txt`

The `mount()` system call

```
int mount(const char *source, const char *target, const char *filesystemtype,
          unsigned long mountflags, const void *data);
```

The `mount()` system call is used to mount a generic filesystem, its `sys_mount()` service routine acts on: a **pathname** of a device containing a filesystem (`source` e.g. `/dev/<...>`), a **pathname** of the directory on which the filesystem will be mounted (`target`), the filesystem **type**, a set of **flags** and a pointer to system dependent data (usually NULL). Flags are:

- `MS_NOEXEC`: Do not allow programs to be executed from this file system.
- `MS_NOSUID`: Do not honour set-UID and set-GID bits when executing programs from this file system.
- `MS_RDONLY`: Mount file system read-only.
- `MS_REMOUNT`: Remount an existing mount. This allows you to change the mountflags and data of an existing mount without having to unmount and remount the file system. `source` and `target` should be the same values specified in the initial `mount()` call; fs type is ignored.
- `MS_SYNCHRONOUS`: Make writes on this file system synchronous

Mount points

Directories selected as the target for the mount operation become a “mount point”. This is reflected in struct dentry by setting in `d_flags` the flag `DCACHE_MOUNTED`.

Further information on https://lwn.net/Articles/649115/

8.4

8. Virtual Filesystem

Files

DIAG

File descriptor table

V5.11

The PCB has a member struct files_struct *files which points to the descriptor table defined in `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  };
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/fdtable.h#L49

fdtable

V5.11

```
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  };
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/fdtable.h#L27

file

V5.11

```
915  struct file {
916          union {
917                  struct llist_node       fu_llist;
918                  struct rcu_head         fu_rcuhead;
919          } f_u;
920          struct path             f_path;
921          struct inode            *f_inode;       /* cached value */
922          const struct file_operations    *f_op;
923
924          /*
925           * Protects f_ep, f_flags.
926           * Must not be taken from IRQ context.
927           */
928          spinlock_t              f_lock;
929          enum rw_hint            f_write_hint;
930          atomic_long_t           f_count;
931          unsigned int            f_flags;
932          fmode_t                 f_mode;
933          struct mutex            f_pos_lock;
934          loff_t                  f_pos;
935          struct fown_struct      f_owner;
936          const struct cred       *f_cred;
937          struct file_ra_state    f_ra;
938
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/fs.h#L915

Opening Files

V5.11

A file struct is allocated when a file is opened. The system call that allows a process to open a file is `open()` serviced by `sys_open()` that in the end calls `do_sys_open()`. The function is logically divided into two parts:

1. a file descriptor is allocated, if available
2. invocation of the intermediate function `struct file *do_filp_open(int dfd, struct filename *pathname, const struct open_flags *op)` which returns the address of the struct file associated with the opened file

On kernel 5.11 `do_sys_open()` calls `do_sys_openat2()`.

do_sys_openat2()

V5.11

```
1156  static long do_sys_openat2(int dfd, const char __user *filename,
1157                             struct open_how *how)
1158  {
1159          struct open_flags op;
1160          int fd = build_open_flags(how, &op);
1161          struct filename *tmp;
1162
1163          if (fd)
1164                  return fd;
1165
1166          tmp = getname(filename);
1167          if (IS_ERR(tmp))
1168                  return PTR_ERR(tmp);
1169
1170          fd = get_unused_fd_flags(how->flags);
1171          if (fd >= 0) {
1172                  struct file *f = do_filp_open(dfd, tmp, &op);
1173                  if (IS_ERR(f)) {
1174                          put_unused_fd(fd);
1175                          fd = PTR_ERR(f);
1176                  } else {
1177                          fsnotify_open(f);
1178                          fd_install(fd, f);
1179                  }
1180          }
1181          putname(tmp);
1182          return fd;
1183  }
```

Finds and allocate an empty slot in the fdtable, if available

Deallocate the file descriptor

"Install" the file descriptor assigning the file struct

(Pointers and Errors)

V5.11

```c
#define IS_ERR_VALUE(x) unlikely((unsigned long)(void *)(x) >= (unsigned long)-MAX_ERRNO)

static inline void * __must_check ERR_PTR(long error) {
	return (void *) error;
}

static inline long __must_check PTR_ERR(__force const void *ptr) {
	return (long) ptr;
}

static inline bool __must_check IS_ERR(__force const void *ptr) {
	return IS_ERR_VALUE((unsigned long)ptr);
}
```

Closing Files

V5.11

The close() system call is defined in fs/open.c as:

```
SYSCALL_DEFINE1(close, unsigned int, fd)
```

This function basically calls (in fs/file.c):

```
int close_fd(unsigned fd)
```

that:

- retrieves the file struct associated with the file, and releases the file descriptor
- calls `filp_close(struct file *filp, fl_owner_t id)`, defined in fs/open.c, which flushing the data structures associated with the file (struct file, dentry and i-node)

close_files()

V5.11

```
617  int close_fd(unsigned fd)
618  {
619          struct files_struct *files = current->files;
620          struct file *file;
621
622          file = pick_file(files, fd);
623          if (!file)
624                  return -EBADF;
625
626          return filp_close(file, files);
627  }
```

https://elixir.bootlin.com/linux/v5.11/source/fs/file.c#L617

```
597  static struct file *pick_file(struct files_struct *files, unsigned fd)
598  {
599          struct file *file = NULL;
600          struct fdtable *fdt;
601
602          spin_lock(&files->file_lock);
603          fdt = files_fdtable(files);
604          if (fd >= fdt->max_fds)
605                  goto out_unlock;
606          file = fdt->fd[fd];
607          if (!file)
608                  goto out_unlock;
609          rcu_assign_pointer(fdt->fd[fd], NULL);
610          __put_unused_fd(files, fd);
611
612  out_unlock:
613          spin_unlock(&files->file_lock);
614          return file;
615  }
```

https://elixir.bootlin.com/linux/v5.11/source/fs/file.c#L597

```
537  static void __put_unused_fd(struct files_struct *files, unsigned int fd)
538  {
539          struct fdtable *fdt = files_fdtable(files);
540          __clear_open_fd(fd, fdt);
541          if (fd < files->next_fd)
542                  files->next_fd = fd;
543  }
```

https://elixir.bootlin.com/linux/v5.11/source/fs/file.c#L537

```
250  static inline void __clear_open_fd(unsigned int fd, struct fdtable *fdt)
251  {
252          __clear_bit(fd, fdt->open_fds);
253          __clear_bit(fd / BITS_PER_LONG, fdt->full_fds_bits);
254  }
255
```

https://elixir.bootlin.com/linux/v5.11/source/fs/file.c#L250

The read() system call

V5.11

```
623 ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count)
624 {
625         struct fd f = fdget_pos(fd);
626         ssize_t ret = -EBADF;
627
628         if (f.file) {
629                 loff_t pos, *ppos = file_ppos(f.file);
630                 if (ppos) {
631                         pos = *ppos;
632                         ppos = &pos;
633                 }
634                 ret = vfs_read(f.file, buf, count, ppos);
635                 if (ret >= 0 && ppos)
636                         f.file->f_pos = pos;
637                 fdput_pos(f);
638         }
639         return ret;
640 }
```

https://elixir.bootlin.com/linux/v5.11/source/fs/read_write.c#L623

```
476 ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
477 {
478         ssize_t ret;
479
480         if (!(file->f_mode & FMODE_READ))
481                 return -EBADF;
482         if (!(file->f_mode & FMODE_CAN_READ))
483                 return -EINVAL;
484         if (unlikely(!access_ok(buf, count)))
485                 return -EFAULT;
486
487         ret = rw_verify_area(READ, file, pos, count);
488         if (ret)
489                 return ret;
490         if (count > MAX_RW_COUNT)
491                 count =  MAX_RW_COUNT;
492
493         if (file->f_op->read)
494                 ret = file->f_op->read(file, buf, count, pos);
495         else if (file->f_op->read_iter)
496                 ret = new_sync_read(file, buf, count, pos);
497         else
498                 ret = -EINVAL;
499         if (ret > 0) {
500                 fsnotify_access(file);
501                 add_rchar(current, ret);
502         }
503         inc_syscr(current);
504         return ret;
505 }
```

Release resources

The `write()` system call

V5.11

The read system call is actually the same of the write but uses `vfs_write()` instead of `vfs_read()`.

```
585 ssize_t vfs_write(struct file *file, const char __user *buf, size_t count, loff_t *pos)
586 {
587         ssize_t ret;
588
589         if (!(file->f_mode & FMODE_WRITE))
590                 return -EBADF;
591         if (!(file->f_mode & FMODE_CAN_WRITE))
592                 return -EINVAL;
593         if (unlikely(!access_ok(buf, count)))
594                 return -EFAULT;
595
596         ret = rw_verify_area(WRITE, file, pos, count);
597         if (ret)
598                 return ret;
599         if (count > MAX_RW_COUNT)
600                 count =  MAX_RW_COUNT;
601         file_start_write(file);
602         if (file->f_op->write)
603                 ret = file->f_op->write(file, buf, count, pos);
604         else if (file->f_op->write_iter)
605                 ret = new_sync_write(file, buf, count, pos);
606         else
607                 ret = -EINVAL;
608         if (ret > 0) {
609                 fsnotify_modify(file);
610                 add_wchar(current, ret);
611         }
612         inc_syscw(current);
613         file_end_write(file);
614         return ret;
615 }
```

https://elixir.bootlin.com/linux/v5.11/source/fs/read_write.c#L585

8.5

8. Virtual Filesystem

The /proc filesystem

DIAG

Overview

The /proc filesystem is an in-memory file system which provides information on:

- active programs (processes)
- the whole memory content
- kernel-level settings (e.g. the currently mounted modules)

Common files on proc are:

- `cpuinfo` contains the information established by the kernel about the processor at boot time, e.g., the type of processor, including variant and features.
- `kcore` contains the entire RAM contents as seen by the kernel.
- meminfo contains information about the memory usage, how much of the available RAM and swap space are in use and how the kernel is using them.
- `version` contains the kernel version information that lists the version number, when it was compiled and who compiled it.

https://www.kernel.org/doc/html/latest/filesystems/proc.html

Overview

Then we have:

- **net/** is a directory containing network information.
  - net/dev contains a list of the network devices that are compiled into the kernel. For each device there are statistics on the number of packets that have been transmitted and received.
  - net/route contains the routing table that is used for routing packets on the network.
  - net/snmp contains statistics on the higher levels of the network protocol.
- **self/** contains information about the current process. The contents are the same as those in the per-process information described later.
- **pid/** contains information about process number pid. The kernel maintains a directory containing process information for each process.
  - pid/cmdline contains the command that was used to start the process (using null characters to separate arguments).
  - pid/cwd contains a link to the current working directory of the process.
  - pid/environ contains a list of the environment variables that the process has available.
  - pid/exe contains a link to the program that is running in the process.
  - pid/fd/ is a directory containing a link to each of the files that the process has open.
  - pid/mem contains the memory contents of the process.
  - pid/stat contains process status information.
  - pid/statm contains process memory usage information.

tgid_base_stuff

https://www.kernel.org/doc/html/latest/filesystems/proc.html

Core Data Structures

`proc/pid` is represented using the data structure defined in fs/proc/internal.h

```
30  struct proc_dir_entry {
31          /*
32           * number of callers into module in progress;
33           * negative -> it's going away RSN
34           */
35          atomic_t in_use;
36          refcount_t refcnt;
37          struct list_head pde_openers;   /* who did ->open, but not ->release */
38          /* protects ->pde_openers and all struct pde_opener instances */
39          spinlock_t pde_unload_lock;
40          struct completion *pde_unload_completion;
41          const struct inode_operations *proc_iops;
42          union {
43                  const struct proc_ops *proc_ops;
44                  const struct file_operations *proc_dir_ops;
45          };
46          const struct dentry_operations *proc_dops;
47          union {
48                  const struct seq_operations *seq_ops;
49                  int (*single_show)(struct seq_file *, void *);
50          };
51          proc_write_t write;
52          void *data;
53          unsigned int state_size;
54          unsigned int low_ino;
55          nlink_t nlink;
56          kuid_t uid;
57          kgid_t gid;
58          loff_t size;
59          struct proc_dir_entry *parent;
60          struct rb_root subdir;
61          struct rb_node subdir_node;
62          char *name;
63          umode_t mode;
64          u8 flags;
65          u8 namelen;
66          char inline_name[];
67  } __randomize_layout;
```

APIs

To create a file in `/proc` you can use the function (source):

```
struct proc_dir_entry *proc_create(const char *name, umode_t mode,
                   struct proc_dir_entry *parent,
                   const struct proc_ops *proc_ops)
```

It is essential to define the `proc_ops` in order to use the file.

```
29  struct proc_ops {
30          unsigned int proc_flags;
31          int     (*proc_open)(struct inode *, struct file *);
32          ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
33          ssize_t (*proc_read_iter)(struct kiocb *, struct iov_iter *);
34          ssize_t (*proc_write)(struct file *, const char __user *, size_t, loff_t *);
35          loff_t  (*proc_lseek)(struct file *, loff_t, int);
36          int     (*proc_release)(struct inode *, struct file *);
37          __poll_t (*proc_poll)(struct file *, struct poll_table_struct *);
38          long    (*proc_ioctl)(struct file *, unsigned int, unsigned long);
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/proc_fs.h#L29

8.6

8. Virtual Filesystem

The /sys filesystem

DIAG

Overview

Similar in spirit to proc, mounted to /sys, it is an alternative way to make the kernel export information (or set it) via common I/O operations.

Very simple API, more clear structuring. The VFS objects are mapped using the following scheme:

Internal	**External**
Kernel Objects	Directories
Object Attributes	Regular Files
Object Relationship	Symbolic Links

Core APIs

V5.11

```
static inline int __must_check sysfs_create_file(struct kobject *kobj, const struct attribute *attr)
static inline void sysfs_remove_file(struct kobject *kobj, const struct attribute *attr)
static inline int sysfs_rename_link(struct kobject *kobj, struct kobject *target, const char *old_name, const char *new_name)
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/sysfs.h

The functions uses the **struct attribute** declared as follows.

```
struct attribute {
    const char *name;
    umode_t mode;
}
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/kobject.h

Instead, the **struct kobject** represents the kernel object (next slide). /sysfs is tight inherently with the kobjects architecture.

Kobjects architecture

A **`kobject`** is an object of type struct kobject. Kobjects have a **name** and a **reference count** (`kref`). A kobject also has a **parent** pointer (allowing objects to be arranged into hierarchies), a specific type, and, usually, a representation in the `sysfs` virtual filesystem.

Kobjects are generally not interesting on their own; instead, they are usually **embedded** within some other structure which contains the stuff the code is really interested in (remember `container_of`).

No structure should EVER have more than one kobject embedded within it. If it does, the reference counting for the object is sure to be messed up and incorrect, and your code will be buggy. So do not do this.

https://www.kernel.org/doc/html/latest/core-api/kobject.html

Kobjects architecture

A **`ktype`** is the type of object that embeds a kobject. Every structure that embeds a kobject needs a corresponding ktype. The ktype controls what happens to the kobject when it is created and destroyed.

A **`kset`** is a group of kobjects. These kobjects can be of the same ktype (classic `kset`) or belong to different ktypes (i.e. a `subsystem`). The kset is the basic container type for collections of kobjects. Ksets contain their own kobjects, but you can safely ignore that implementation detail as the kset core code handles this kobject automatically.

When you see a sysfs directory full of other directories, generally each of those directories corresponds to a kobject in the same kset.

https://www.kernel.org/doc/html/latest/core-api/kobject.html

Data Structures

```c
struct kobject {
        const char              *name;
        struct list_head        entry;
        struct kobject          *parent;
        struct kset             *kset;
        struct kobj_type        *ktype;
        struct kernfs_node      *sd; /* sysfs directory entry */
        struct kref             kref;
#ifdef CONFIG_DEBUG_KOBJECT_RELEASE
        struct delayed_work     release;
#endif
        unsigned int state_initialized:1;
        unsigned int state_in_sysfs:1;
        unsigned int state_add_uevent_sent:1;
        unsigned int state_remove_uevent_sent:1;
        unsigned int uevent_suppress:1;
};
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/kobject.h#L64

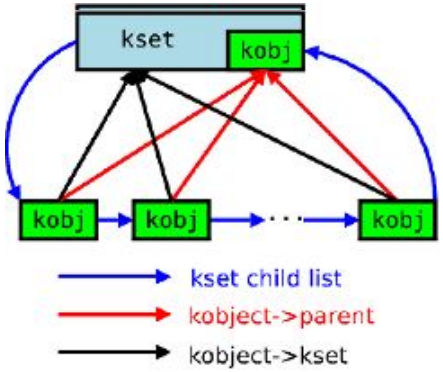
```c
struct kobj_type {
        void (*release)(struct kobject *kobj);
        const struct sysfs_ops *sysfs_ops;
        struct attribute **default_attrs;       /* use default_groups instead */
        const struct attribute_group **default_groups;
        const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject *kobj);
        const void *(*namespace)(struct kobject *kobj);
        void (*get_ownership)(struct kobject *kobj, kuid_t *uid, kgid_t *gid);
};
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/kobject.h#L138

```c
struct sysfs_ops {
        ssize_t (*show)(struct kobject *, struct attribute *, char *);
        ssize_t (*store)(struct kobject *, struct attribute *, const char *, size_t);
};
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/sysfs.h#L230

Kobjects are arranged as in the figure on the right. The kernel offers APIs for initializing objects and for adding/removing them from ksets.



https://lwn.net/Articles/51437/

Example

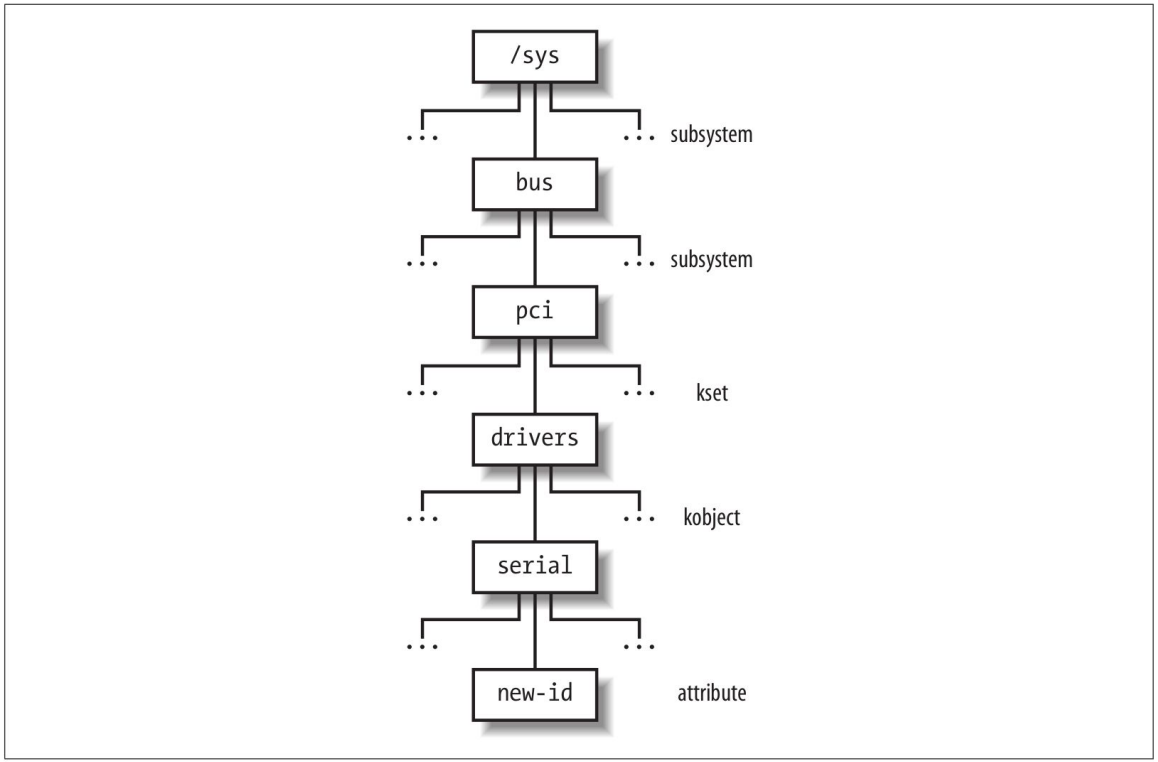


Figure 13-3. An example of device driver model hierarchy

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

APIs

```
void kobject_init(struct kobject *kobj);
int kobject_set_name(struct kobject *kobj, const char *format, ...);
struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);

void kset_init(struct kset *kset);
int kset_add(struct kset *kset);
int kset_register(struct kset *kset);
void kset_unregister(struct kset *kset);
struct kset *kset_get(struct kset *kset);
void kset_put(struct kset *kset);
kobject_set_name(my_set->kobj, "The name");
```

Hooking into sysfs

An initialized kobject will perform reference counting without trouble, but it will not appear in sysfs. To create sysfs entries, kernel code must pass the object to `kobject_add()`:

```
int kobject_add(struct kobject *kobj);
```

As always, this operation can fail. The function:

```
void kobject_del(struct kobject *kobj);
```

will remove the kobject from sysfs.

There is a `kobject_register()` function, which is really just the combination of the calls to `kobject_init()` and `kobject_add()`. Similarly, `kobject_unregister()` will call `kobject_del()`, then call `kobject_put()` to release the initial reference created with `kobject_register()` (or really `kobject_init()`)

8.7

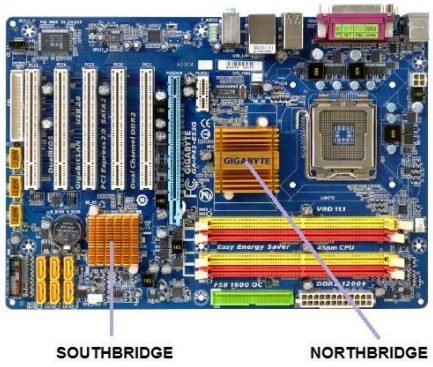
8. Virtual Filesystem

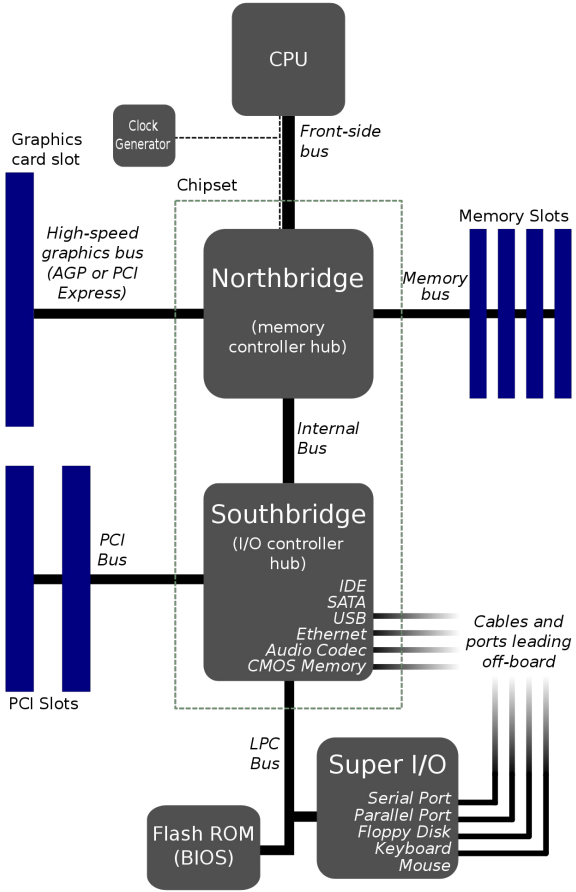
Device Management

DIAG

The I/O Architecture

The essential part of a computer is the internal communication structure which allows all the essential components to communicate. The internal communication is built upon data path which are called buses. Any computer has a **system bus** that connects most of the internal hardware devices (e.g. PCI, SCSI, USB). Since several buses may exists they are linked together by hardware devices called bridges (northbridge and southbridge).





https://en.wikipedia.org/wiki/Southbridge_(computing)

The I/O architecture

Any I/O device is hosted by one and only one bus. The data path that connects a CPU to an I/O device is generally called a I/O bus.

The essential components of the I/O architecture are:

- **I/O Ports** - Each device has its own set of I/O addresses which are called I/O ports accessible through special assembly instructions (e.g. in, out)
- **I/O Interfaces** - That are hardware circuits between a group of I/O ports and the corresponding device controller, they acts as interpreter translating data and also issuing interrupts (examples: keyboard int., graphic int., disk int., network int., serial/parallel port, SCSI and USB)
- **Device Controllers** - They have two important roles:
  - interpreting high level commands from I/O ports to electrical signals to the device
  - converts electrical signals from the device and updates status registers

The Device Driver Model

In the early days devices were very different to each other, and offering a unified view made no sense. With years and standards the need of a unified model of devices arose. Different devices have more or less the same set of functionalities that regards:

- power management
- plug and play
- hot-plugging

To implement these kind of operations Linux offers a set of data structures and functions that unify view of all buses, devices and devices drivers. This framework is called the **Device Driver Model**. Its main components are:

- Devices
- Drivers
- Buses
- Classes

Devices

Device are representation is stored in the `device` object, but in the Linux kernel they are also represented by special files called ***device files*** (in the folder /dev), thus the same system calls used to interact with regular files can be used.

According to the characteristics of the underlying drivers, device files can be of two types:

- **block devices**, they allow data to be accessed randomly, in blocks and in relative small time (e.g. hdd, dvd)
- **character devices**, they cannot allow data to be accessed randomly and character by character (i.e. bit by bit) (e.g. sound card)

Network cards are not associated with device files and some device may be not associated with a real hardware (as `/dev/null`).

Devices

device object

V5.11

```
447  * At the lowest level, every device in a Linux system is represented by an
448  * instance of struct device. The device structure contains the information
449  * that the device model core needs to model the system. Most subsystems,
450  * however, track additional information about the devices they host. As a
451  * result, it is rare for devices to be represented by bare device structures;
452  * instead, that structure, like kobject structures, is usually embedded within
453  * a higher-level representation of the device.
454  */
455 struct device {
456         struct kobject kobj;
457         struct device           *parent;
458
459         struct device_private   *p;
460
461         const char              *init_name; /* initial name of the device */
462         const struct device_type *type;
463
464         struct bus_type *bus;           /* type of bus device is on */
465         struct device_driver *driver;   /* which driver has allocated this
466                                            device */
467         void            *platform_data; /* Platform specific data, device
468                                            core doesn't touch it */
469         void            *driver_data;   /* Driver data, set and get with
470                                            dev_set_drvdata/dev_get_drvdata */
471 #ifdef CONFIG_PROVE_LOCKING
472         struct mutex            lockdep_mutex;
473 #endif
474         struct mutex            mutex;  /* mutex to synchronize calls to
475                                          * its driver.
476                                          */
477
478         struct dev_links_info   links;
479         struct dev_pm_info      power;
480         struct dev_pm_domain    *pm_domain;
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/device.h#L455

Numbers

Each device is associated with a couple of numbers: **MAJOR** and **MINOR**:

- MAJOR is the key to access the device driver as registered within a driver database
- MINOR identifies the actual instance of the device driven by that driver (this can be specified by the driver programmer)

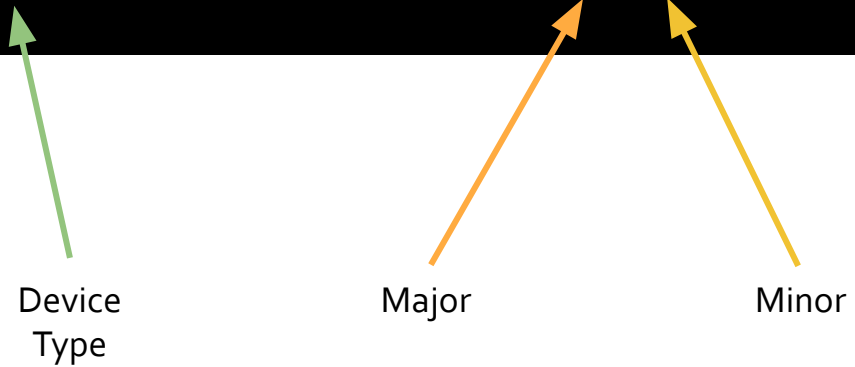
There are different tables to register devices, depending on whether the device is a char device or a block device:

- `fs/char_dev.c` for char devices
- `fs/block_dev.c` for block devices

In the above source files we can also find device-independent functions for accessing the actual driver.

Device numbers

```
[gpm@fedora-works ~]$ ls -l /dev/sda /dev/ttyS0
brw-rw----. 1 root disk    8,  0 Apr 20 08:37 /dev/sda
crw-rw----. 1 root dialout 4, 64 Apr 20 08:37 /dev/ttyS0
```



Device numbers

In general, the same major can be given to both a character and a block device! Numbers are "assigned" by the Linux Assigned Names and Numbers Authority (http://lanana.org/) and kept in Documentation/devices.txt. Defines are in `include/uapi/linux/major.h`

```
[gpm@fedora-works ~]$ ls -l /dev/sd*
brw-rw----. 1 root disk 8,  0 Apr 20 08:37 /dev/sda
brw-rw----. 1 root disk 8,  1 Apr 20 08:37 /dev/sda1
brw-rw----. 1 root disk 8,  2 Apr 20 08:37 /dev/sda2
brw-rw----. 1 root disk 8,  3 Apr 20 08:37 /dev/sda3
brw-rw----. 1 root disk 8, 16 Apr 20 08:37 /dev/sdb
brw-rw----. 1 root disk 8, 17 Apr 20 08:37 /dev/sdb1
```

All of these devices have the same major number, so they are probably linked to the same driver

The Device Database

V5.11

Char and Block devices behave differently, but they are organized in identical databases which are handled as **hashmaps**. They are referenced as `cdev_map` and `bdev_map`.

```
19  struct kobj_map {
20          struct probe {
21                  struct probe *next;
22                  dev_t dev;
23                  unsigned long range;
24                  struct module *owner;
25                  kobj_probe_t *get;
26                  int (*lock)(dev_t, void *);
27                  void *data;
28          } *probes[255];
29          struct mutex *lock;
30  };
```

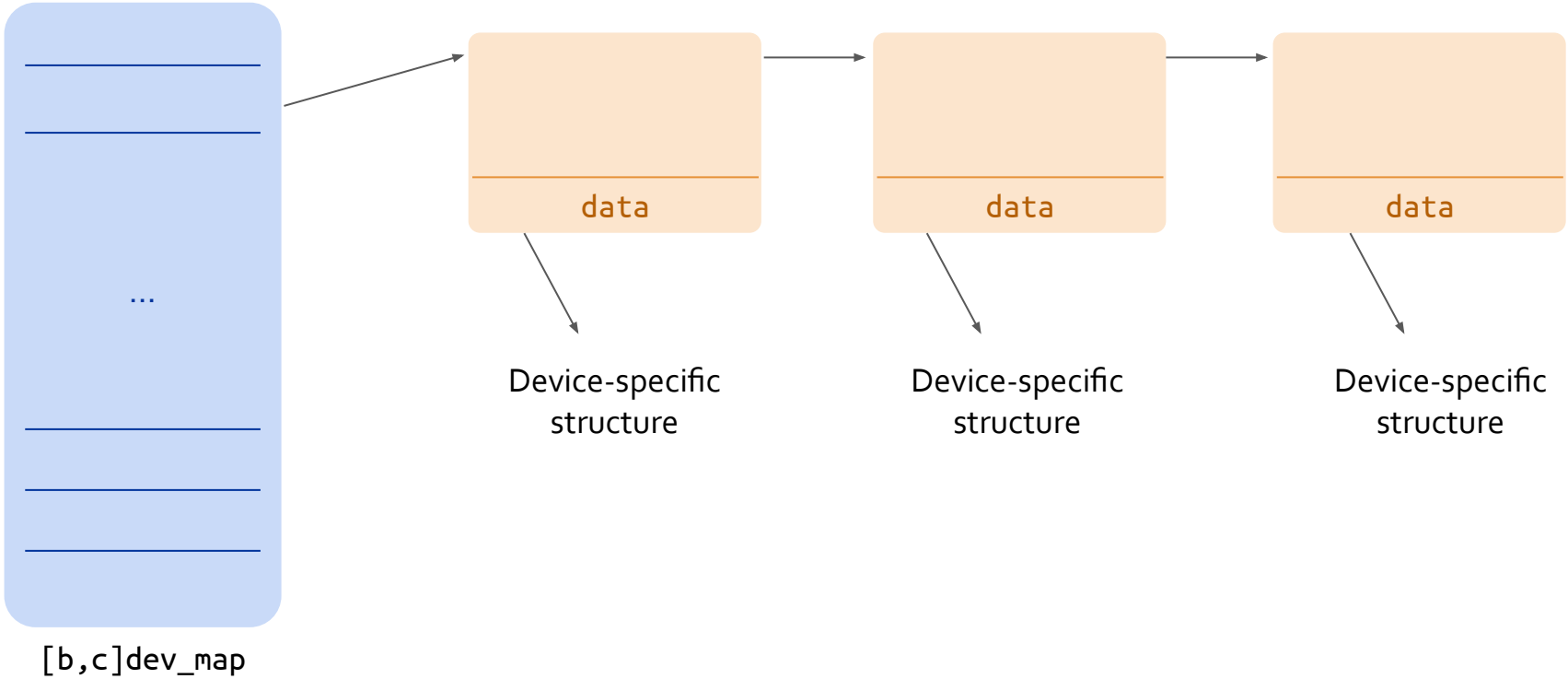
https://elixir.bootlin.com/linux/v5.11/source/drivers/base/map.c#L19

32bit unsigned integer storing major and minor

```
7   #define MINORBITS       20
8   #define MINORMASK       ((1U << MINORBITS) - 1)
9
10  #define MAJOR(dev)      ((unsigned int) ((dev) >> MINORBITS))
11  #define MINOR(dev)      ((unsigned int) ((dev) & MINORMASK))
12  #define MKDEV(ma,mi)    (((ma) << MINORBITS) | (mi))
```

hasing is done as:
major % 255

The Device Database



8.7.1

8. Virtual Filesystem
7. Device Management

Char Devices

DIAG

struct cdev

V5.11

```
14  struct cdev {
15          struct kobject kobj;
16          struct module *owner;
17          const struct file_operations *ops;
18          struct list_head list;
19          dev_t dev;
20          unsigned int count;
21  } __randomize_layout;
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/cdev.h#L14

Range Database

V5.11

The struct char_device_struct is used to manage device number allocation to drivers.

```
32  #define CHRDEV_MAJOR_HASH_SIZE 255
33
34  static struct char_device_struct {
35          struct char_device_struct *next;
36          unsigned int major;
37          unsigned int baseminor;
38          int minorct;
39          char name[64];
40          struct cdev *cdev;              /* will die */
41  } *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

https://elixir.bootlin.com/linux/v5.11/source/fs/char_dev.c#L34

Registering Char Device

V5.11

linux/fs.h provides the following wappers to register/deregister a driver:

- `int register_chrdev(unsigned int major, const char *name, struct file_operations *fops)`: registration takes place onto the entry at displacement MAJOR (o means the choice is up the kernel). The actual MAJOR number is returned.
- `int unregister_chrdev(unsigned int major, const char *name)`: releases the entry at displacement MAJOR

They map to actual operations in fs/char_dev.c:

- `int __register_chrdev(unsigned int major, unsigned int baseminor, unsigned int count, const char *name, const struct file_operations *fops)`
- `void __unregister_chrdev(unsigned int major, unsigned int baseminor, unsigned int count, const char *name)`

File Operations

V5.11

```
1820  struct file_operations {
1821          struct module *owner;
1822          loff_t (*llseek) (struct file *, loff_t, int);
1823  ----->  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
1824  ----->  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
1825          ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
1826          ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
1827          int (*iopoll)(struct kiocb *kiocb, bool spin);
1828          int (*iterate) (struct file *, struct dir_context *);
1829          int (*iterate_shared) (struct file *, struct dir_context *);
1830          __poll_t (*poll) (struct file *, struct poll_table_struct *);
1831  ----->  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
1832          long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
1833          int (*mmap) (struct file *, struct vm_area_struct *);
1834          unsigned long mmap_supported_flags;
1835          int (*open) (struct inode *, struct file *);
1836          int (*flush) (struct file *, fl_owner_t id);
1837          int (*release) (struct inode *, struct file *);
1838          int (*fsync) (struct file *, loff_t, loff_t, int datasync);
1839          int (*fasync) (int, struct file *, int);
1840          int (*lock) (struct file *, int, struct file_lock *);
1841          ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
1842          unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, uns
1843          int (*check_flags)(int);
1844          int (*flock) (struct file *, int, struct file_lock *);
```

Registering Device Numbers

A driver might require to register or allocate a range of device numbers.

APIs are in fs/char_dev.c and exposed in include/linux/fs.h:

- `int register_chrdev_region(dev_t from, unsigned count, const char *name)` - Major is specified in `from`
- `int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name)` - Major and first minor are returned in `dev`

8.7.2

8. Virtual Filesystem
7. Device Management

Block Devices

DIAG

struct gendisk

The structure corresponding to cdev for a block device is struct gendisk in `include/linux/genhd.h`.

```
137 struct gendisk {
138         /* major, first_minor and minors are input parameters only,
139          * don't use directly.  Use disk_devt() and disk_max_parts().
140          */
141         int major;                      /* major number of driver */
142         int first_minor;
143         int minors;                     /* maximum number of minors, =1 for
144                                          * disks that can't be partitioned. */
145
146         char disk_name[DISK_NAME_LEN];  /* name of major driver */
147
148         unsigned short events;          /* supported events */
149         unsigned short event_flags;     /* flags related to event processing */
150
151         /* Array of pointers to partitions indexed by partno.
152          * Protected with matching bdev lock but stat and other
153          * non-critical accesses use RCU.  Always access through
154          * helpers.
155          */
156         struct disk_part_tbl __rcu *part_tbl;
157         struct block_device *part0;
158
159         const struct block_device_operations *fops;
160         struct request_queue *queue;
161         void *private_data;
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/genhd.h#L137

APIs

In `block/genhd.c` we find the following functions to register/deregister the driver:

- `int register_blkdev(unsigned int major, const char * name, struct block_device_operations *bdops)`
- `int unregister_blkdev(unsigned int major, const char * name)`

As far as regard the block device operations we have neither read nor write!

```
1852  struct block_device_operations {
1853          blk_qc_t (*submit_bio) (struct bio *bio);
1854          int (*open) (struct block_device *, fmode_t);
1855          void (*release) (struct gendisk *, fmode_t);
1856          int (*rw_page)(struct block_device *, sector_t, struct page *, unsigned int);
1857          int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
1858          int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
1859          unsigned int (*check_events) (struct gendisk *disk,
1860                                        unsigned int clearing);
1861          void (*unlock_native_capacity) (struct gendisk *);
1862          int (*revalidate_disk) (struct gendisk *);
1863          int (*getgeo)(struct block_device *, struct hd_geometry *);
1864          int (*set_read_only)(struct block_device *bdev, bool ro);
1865          /* this callback is with swap_lock and sometimes page table lock held */
1866          void (*swap_slot_free_notify) (struct block_device *, unsigned long);
1867          int (*report_zones)(struct gendisk *, sector_t sector,
1868                              unsigned int nr_zones, report_zones_cb cb, void *data);
1869          char *(*devnode)(struct gendisk *disk, umode_t *mode);
1870          struct module *owner;
1871          const struct pr_ops *pr_ops;
1872  };
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/blkdev.h#L1852

Block Devices Handling

For char devices the management of read/write operations is in charge of the device driver. This is not the same for block devices read/write operations on block devices are handled via a single API related to buffer cache operations.

The actual implementation of the buffer cache policy will determine the real execution activities for block device read/write operations.

Request Queues

Request queues (strategies in UNIX) are the way to operate on block devices. Requests encapsulate optimizations to manage each specific device (e.g. via the elevator algorithm). The Request Interface is associated with a queue of pending requests towards the block device

Block Devices Handling

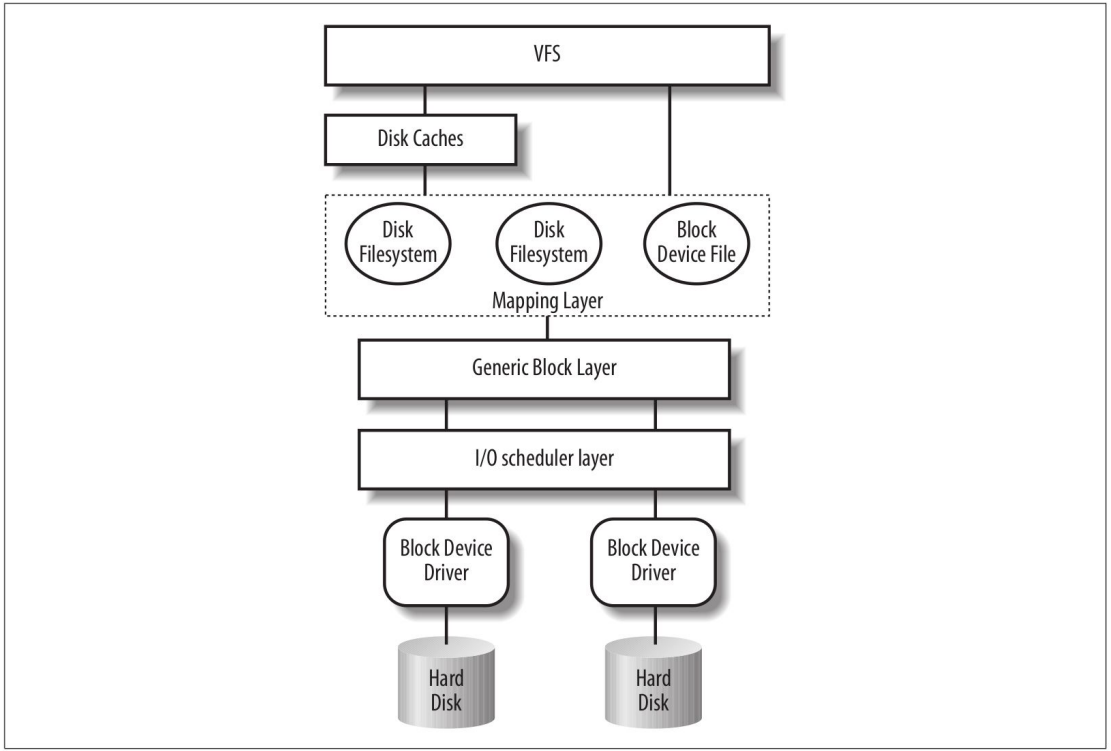


Figure 14-1. Kernel components affected by a block device operation

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

8.7.3

8. Virtual Filesystem
7. Device Management

Devices and VFS

DIAG

Linking Devices and the VFS

The member **`umode_t`** `i_mode` in **`struct`** `inode` tells the type of the inode:

- directory
- file
- char device
- block device
- (named) pipe

The kernel function `sys_mknod()` creates a generic inode. If the iinode represents a device, the operations to manage the device are retrieved via the device driver database.

In particular, the inode has the **`dev_t`** `i_rdev` member

The mknod() system call

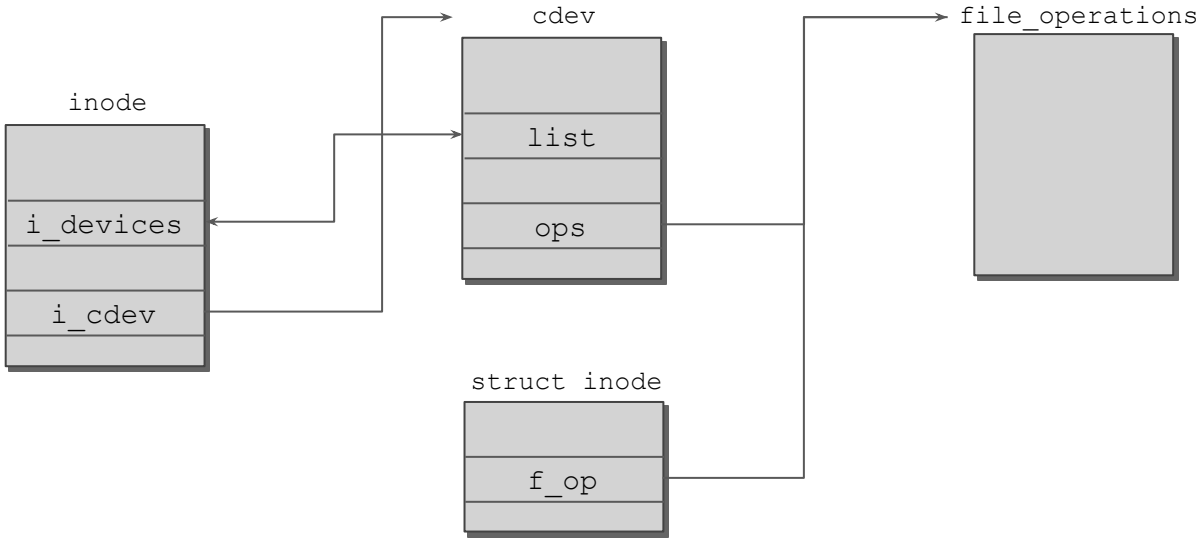
```
int mknod(const char *pathname, mode_t mode, dev_t dev)
```

Where

- **mode** specifies permissions and type of node to be created, permissions are filtered via the umask of the calling process (mode & umask)
- different macros can be used to define the node type: S_IFREG, S_IFCHR, S_IFBLK, S_IFIFO.
- When using S_IFCHR or S_IFBLK, the parameter **dev** specifies Major and Minor numbers of the device file to create, otherwise it is a don't care

Opening Device Files

In fs/devices.c there is the generic **chrdev_open()** function. This function needs to find the dev-specific file operations. Given the device, number, **kobject_lookup()** is called to find a corresponding **kobject**. From the kobject we can navigate to the corresponding **cdev**. The device-dependent file operations are then in **cdev->ops**. This information is then cached in the i-node



8.7.4

8. Virtual Filesystem
7. Device Management

Classes

DIAG

Overview

Devices are organized into "classes", and a device can belong to multiple classes.

The device class membership is shown in /sys/class/. Block devices for example are automatically placed under the "`block`" class, this is done automatically when the gendisk structure is registered in the kernel. To each class is associated a `class` object.

Most devices don't require the creation of new classes.

struct class

```
48   * A class is a higher-level view of a device that abstracts out low-level
49   * implementation details. Drivers may see a SCSI disk or an ATA disk, but,
50   * at the class level, they are all simply disks. Classes allow user space
51   * to work with devices based on what they do, rather than how they are
52   * connected or how they work.
53   */
54  struct class {
55          const char              *name;
56          struct module           *owner;
57
58          const struct attribute_group    **class_groups;
59          const struct attribute_group    **dev_groups;
60          struct kobject                  *dev_kobj;
61
62          int (*dev_uevent)(struct device *dev, struct kobj_uevent_env *env);
63          char *(*devnode)(struct device *dev, umode_t *mode);
64
65          void (*class_release)(struct class *class);
66          void (*dev_release)(struct device *dev);
67
68          int (*shutdown_pre)(struct device *dev);
69
70          const struct kobj_ns_type_operations *ns_type;
71          const void *(*namespace)(struct device *dev);
72
73          void (*get_ownership)(struct device *dev, kuid_t *uid, kgid_t *gid)
74
75          const struct dev_pm_ops *pm;
76
77          struct subsys_private *p;
78  };
```

https://elixir.bootlin.com/linux/v5.11/source/include/linux/device/class.h#L54

APIs

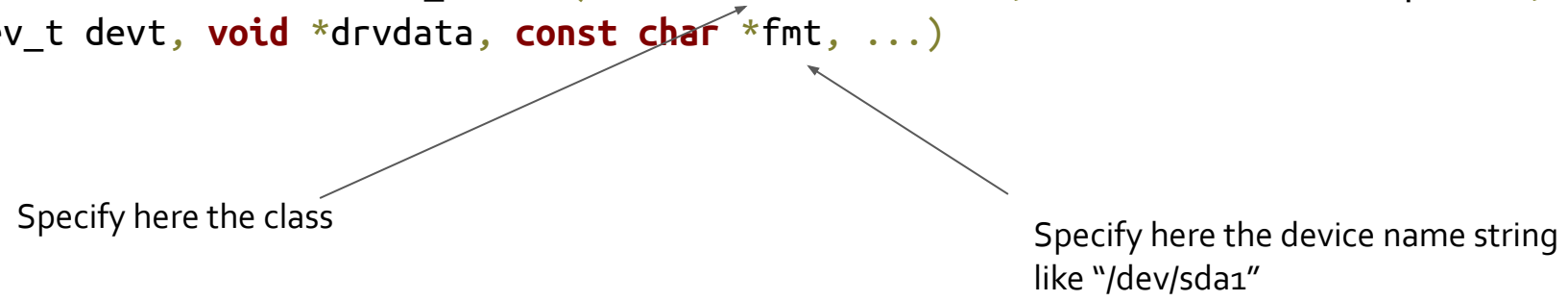
```
static struct class sbd_class = {
     .name = "class_name",
     .class_release = release_fn
};

int class_register(struct class *cls);
void class_destroy(struct class *cls);
struct class *class_create(struct module *owner, const
char *name, struct lock_class_key *key)
```

APIs

Devices can be added to classes with the following function:

```
struct device *device_create(struct class *class, struct device *parent,
dev_t devt, void *drvdata, const char *fmt, ...)
```



And removed with:

```
void device_destroy(struct class *class, dev_t devt)
```

Device Class Attributes

Specify attributes for the classes, and functions to "read" and "write" the specific class attributes.

```
CLASS_DEVICE_ATTR(name, mode, show, store);
```

This is expanded to a structure called `dev_attr_name` where we have (as kobjects):

- `ssize_t (*show)(struct class_device *cd, char *buf);`
- `ssize_t (*store)(struct class_device *, const char *buf, size_t count);`

8.7.5

8. Virtual Filesystem
7. Device Management

udev

DIAG

Overview

udev is the userspace Linux device manager, it manages device nodes in `/dev`. It also handles userspace events raised when devices are added/removed to/from the system. The introduction of udev has been due to the degree of complexity associated with device management. It is highly configurable and rule-based.

Rules

Udev in userspace looks at `/sys` to detect changes and see whether new (virtual) devices are plugged. Special rule files (in `/etc/udev/rules.d`) match changes and create files in /dev accordingly. Syntax tokens in syntax files:

- `KERNEL`: match against the kernel name for the device
- `SUBSYSTEM`: match against the subsystem of the device
- `DRIVER`: match against the name of the driver backing the device
- `NAME`: the name that shall be used for the device node
- `SYMLINK`: a list of symbolic links which act as alternative names for the device node

```
KERNEL=="hdb", DRIVER=="ide-disk", NAME="my_spare_disk", SYMLINK+="sparedisk", MODE="0644"
```

Advanced Operating Systems and Virtualization

[8] Virtual File System

LECTURER

Gabriele **Proietti Mattia**

BASED ON WORK BY

http://www.ce.uniroma2.it/~pellegrini/

gpm.name · proiettimattia@diag.uniroma1.it

DIAG