Virtual File System and Devices

Advanced Operating Systems and Virtualization
Alessandro Pellegrini
A.Y. 2019/2020

Virtual File System

- 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
- Roles of the VFS:
  - Keep track of available filesystem types.
  - Associate (and deassociate) 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.

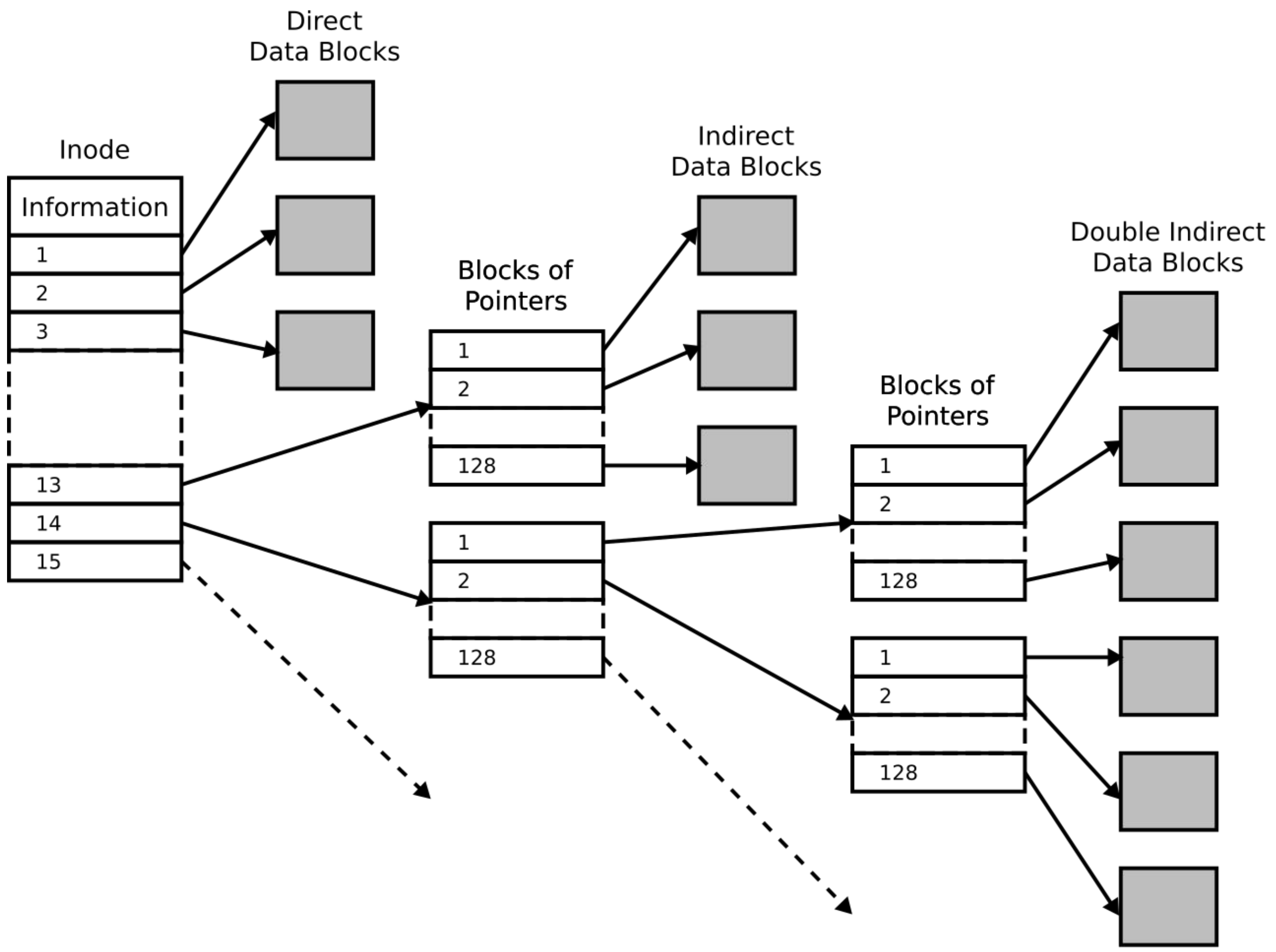
File System: Representations

- In RAM:
  - Partial/full representation of the current structure and content of the File System
- On device:
  - (possibly outdated) representation of the structure and of the content of the File System
- Data access and manipulation:
  - <u>FS-independent part</u>: interface towards other subsystems within the kernel
  - <u>FS-dependent part</u>: data access/manipulation modules targeted at a specific file system type
- In UNIX: "*everything is a file*"

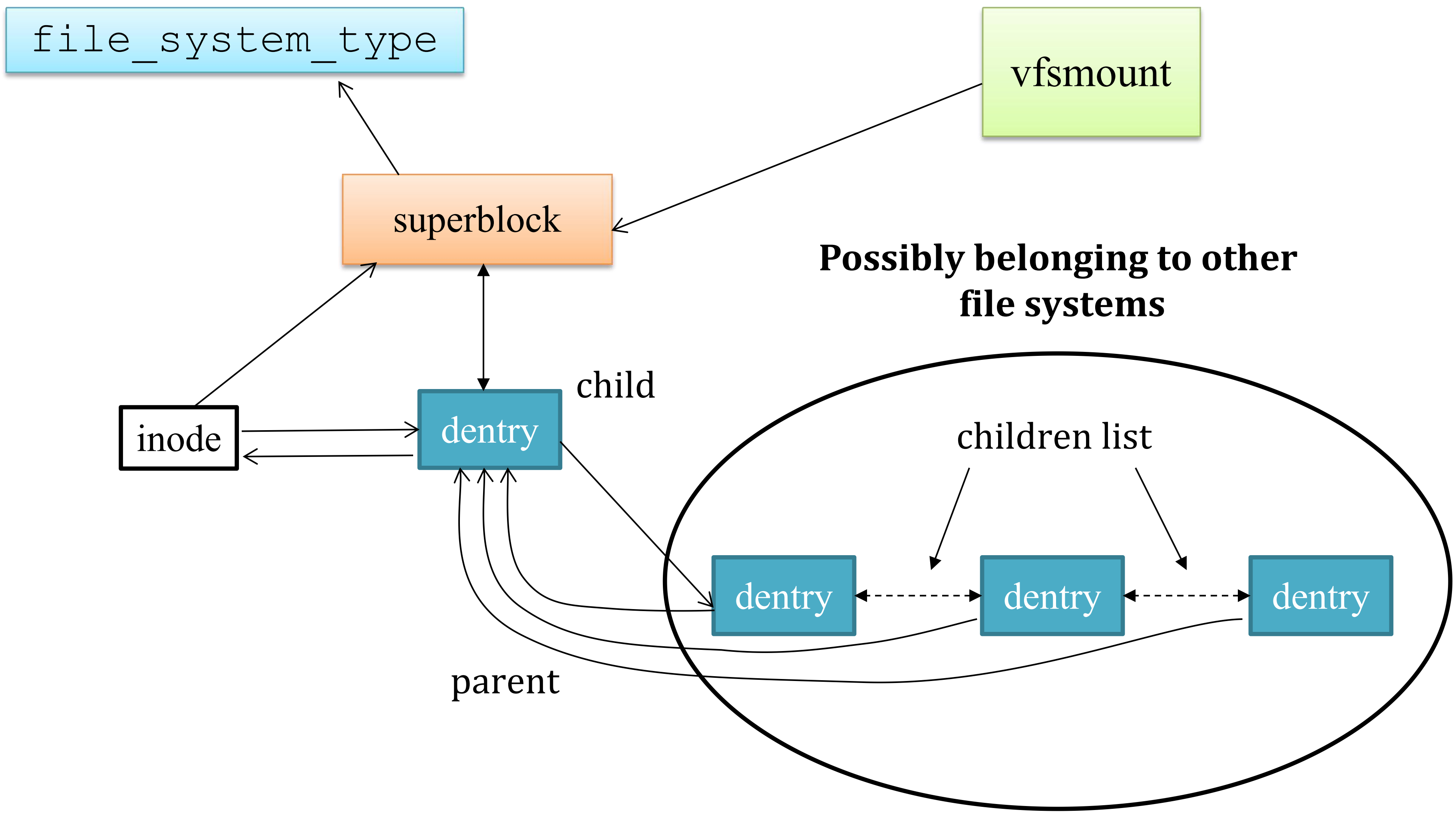
Connecting the two parts

- Any FS object (dir/file/dev) is represented in RAM via specific data structures
- They keep a reference to the code which correctly talks to the actual device, if any
- The reference is accessed using File System independent APIs by other kernel subsystems
- Function pointers are used to reference actual drivers' functions

The EXT2 File System (on device)



VFS Global Organization



File system types

- 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)

```
struct file_system_type {
   const char *name;
   int fs_flags;
   struct super_block *(*read_super)(struct super_block *,
   void *, int);
   struct module *owner;
   struct file_system_type * next;
   struct list_head fs_supers;
};
```

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

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
  - This has the same idea behind the fact that init process cannot be killed
  - Rather than checking for empty lists, we always have at least one placeholder
- During kernel boot, another (actual) filesystem is mounted over rootfs

vfsmount

```
struct vfsmount
{
    struct list_head mnt_hash;
    struct vfsmount *mnt_parent;     /*fs we are mounted on */
    struct dentry *mnt_mountpoint;   /*dentry of mountpoint */
    struct dentry *mnt_root;         /*root of the mounted tree*/
    struct super_block *mnt_sb;      /*pointer to superblock */
    struct list_head mnt_mounts;     /*list of children, anchored
                                                         here */
    struct list_head mnt_child;     /*and going through their
                                                         mnt_child */
    atomic_t mnt_count;
    int mnt_flags;
    char *mnt_devname;               /* Name of device e.g.
                                                    /dev/dsk/hda1 */
    struct list_head mnt_list;
};
```

struct super_block

```
struct super_block {
        struct list_head                s_list;     /* Keep this first */
        ……
        unsigned long                   s_blocksize;
        ……
        unsigned long long              s_maxbytes; /* Max file size */
        struct file_system_type         *s_type;
        struct super_operations         *s_op;
        ……
        struct dentry                   *s_root;
        ……
        struct list_head                s_dirty; /* dirty inodes */
        ……
        union {
                struct minix_sb_info    minix_sb;
                struct ext2_sb_info     ext2_sb;
                struct ext3_sb_info     ext3_sb;
                struct ntfs_sb_info     ntfs_sb;
                struct msdos_sb_info    msdos_sb;
                ……
                void                    *generic_sbp;
        } u;
        ……
};
```

struct dentry

```
struct dentry {
        unsigned int dflags;
        ……
        struct inode  * d_inode;   /* Where the name belongs to */
        struct dentry * d_parent;  /* parent directory */
        struct list_head d_hash;   /* lookup hash list */
        ……
        struct list_head d_child;  /* child of parent list */
        struct list_head d_subdirs;      /* our children */
        ……
        struct qstr d_name;
        ……
        struct lockref d_lockref;  /*per-dentry lock and refcount*/
        struct dentry_operations  *d_op;
        struct super_block * d_sb; /* The root of the dentry tree*/
        ……
        unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
};
```

struct inode

```
struct inode {
       ……
       struct list_head        i_dentry;
       ……
       uid_t                   i_uid;
       gid_t                   i_gid;
       ……
       unsigned long           i_blksize;
       unsigned long           i_blocks;
       ……
       struct inode_operations        *i_op;
       struct file_operations         *i_fop;
       struct super_block             *i_sb;
       wait_queue_head_t              i_wait;
       ……
       union {
              ……
              struct ext2_inode_info        ext2_i;
              struct ext3_inode_info        ext3_i;
              ……
              struct socket                 socket_i;
              ……
              void                          *generic_ip;
       } u;
};
```

VFS and PCBs

- 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;
```

Superblock operations

- Superblock operations must:
  - 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`

struct super_operations

- It is defined in `include/linux/fs.h`

```
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);
    void (*read_inode) (struct inode *);
    void (*read_inode2) (struct inode *, void *) ;
    void (*dirty_inode) (struct inode *);
    void (*write_inode) (struct inode *, int);
    void (*put_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*sync_fs) (struct super_block *);
    void (*write_super_lockfs) (struct super_block *);
    void (*unlockfs) (struct super_block *);
    int (*statfs) (struct super_block *, struct statfs *);
    ...
};
```

Ramfs Example

- Defined in `fs/ramfs/inode.c` and `fs/libfs.c`

```
int simple_statfs(struct dentry *dentry,
                  struct kstatfs *buf)
{
   buf->f_type = dentry->d_sb->s_magic;
   buf->f_bsize = PAGE_SIZE;
   buf->f_namelen = NAME_MAX;
   return 0;
}


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

dentry operations

- 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

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

$\xrightarrow{}$ Removes the pointed i-node (when releasing the dentry)

$\xrightarrow{}$ Removes the dentry, when the reference counter is set to zero

i-node operations

- They specify i-node related operations
- The table maintaining the corresponding function pointers is defined in `include/linux/fs.h`

```
struct inode_operations {
   ...
   int (*create) (struct inode *,struct dentry *,int);
   struct dentry * (*lookup) (struct inode *,struct dentry *);
   int (*link) (struct dentry *,struct inode *,struct dentry *);
   int (*unlink) (struct inode *,struct dentry *);
   int (*symlink) (struct inode *,struct dentry *,const char *);
   int (*mkdir) (struct inode *,struct dentry *,int);
   int (*rmdir) (struct inode *,struct dentry *);
   int (*mknod) (struct inode *,struct dentry *,int,int);
   ...
};
```

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
- 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
- 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)

Pathname Lookup

- Implemented in `fs/namei.c`
- main functions are `vfs_path_lookup()`, `filename_lookup()` and `path_lookup at()`
- Path walking relies on the `namei` data structure (only some members are shown):

```
struct nameidata {
    struct path path;
    struct qstr last;
    struct path root;
    struct inode *inode; /* path.dentry.d_inode */
    unsigned int flags;
    unsigned depth;
} __randomize_layout;
```

$\rightarrow$ Increments the refcount of dentry & inode

$\rightarrow$ Lookup operation flags

$\rightarrow$ current level of symlink navigation

Pathname Lookup

- Lookup operation flags drive the pathname lookup behavior:
- Some flags are:
  - `LOOKUP_FOLLOW`: If the last component is a symlink, follow it
  - `LOOKUP_DIRECTORY`: The last component must be a directory
  - `LOOKUP_AUTOMOUNT`: Ensures that, if the final component is an automount
  - point, then the mount is triggered
  - `LOOKUP_PARENT`: Used to access next-to-last component of the path (e.g., for file creation)
  - `LOOKUP_OPEN`: The intent is to open a file
  - `LOOKUP_CREATE`: The intent is to create a file
  - `LOOKUP_EXCL`: The intent is to access exclusively

$\}$ Not directly used by VFS, but made available to the underlying filesystem

- 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);
```

- `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; filesystem type is ignored.
- `MS_SYNCHRONOUS`: Make writes on this file system synchronous (as though the `O_SYNC` flag to `open(2)` was specified for all file opens to this file system).

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`
- Any path lookup function ignores the content of mount points (namely the name of the dentry) while performing pattern matching

File descriptor table

- The PCB has a member `struct files_struct *files` which points to the descriptor table defined in `include/linux/fdtable.h`:

```
struct files_struct {
        atomic_t count;
        bool resize_in_progress;
        wait_queue_head_t resize_wait;

        struct fdtable __rcu *fdt;
        struct fdtable fdtab;

        spinlock_t file_lock ____cacheline_aligned_in_smp;
        unsigned int next_fd;
        unsigned long close_on_exec_init[1];
        unsigned long open_fds_init[1];
        unsigned long full_fds_bits_init[1];
        struct file __rcu *fd_array[NR_OPEN_DEFAULT];
};
```

struct fdtable

```
struct fdtable {
    unsigned int max_fds;
    struct file __rcu **fd
    unsigned long *close_on_exec;
    unsigned long *open_fds;
    unsigned long *full_fds_bits;
    struct rcu_head rcu;
};
```

bitmaps

struct file

```
struct file {
        struct path                 f_path;
        struct inode                *f_inode;
        const struct file_operations  *f_op;
        spinlock_t                  f_lock;
        atomic_long_t               f_count;
        unsigned int                f_flags;
        fmode_t                     f_mode;
        struct mutex                f_pos_lock;
        loff_t                      f_pos;
        struct fown_struct          f_owner;
        const struct cred           *f_cred;
        ...
        struct address_space        *f_mapping;
        errseq_t                f_wb_err;
}
```

Opening a file

- `do_sys_open()` in `fs/open.c` is logically divided in two parts:
  - First, a file descriptor is allocated (and a suitable `struct file` is allocated)
  - The second relies on an 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

do_sys_open()

```
long do_sys_open(int dfd, const char __user *filename,
int flags, umode_t mode) {
        struct filename *tmp;

        tmp = getname(filename);
        if (IS_ERR(tmp))
                return PTR_ERR(tmp);

        fd = get_unused_fd_flags(flags);
        if (fd >= 0) {
                struct file *f = do_filp_open(dfd, tmp, &op);
                if (IS_ERR(f)) {
                        put_unused_fd(fd);
                        fd = PTR_ERR(f);
                } else {
                        fsnotify_open(f);
                        fd_install(fd, f);
                }
        }
        putname(tmp);
        return fd;
}
```

Kernel Pointers and Errors

- From include/linux/err.h

```
#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 a file

- 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(struct files_struct *files, unsigned fd)
```

- `__close_fd()`:
  - Closes the file descriptor by calling into `__put_unused_fd();`
  - 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_fd()

```
int __close_fd(struct files_struct *files, unsigned fd)
{
        struct file *file;
        struct fdtable *fdt;

        spin_lock(&files->file_lock);
        fdt = files_fdtable(files);
        if (fd >= fdt->max_fds)
                goto out_unlock;
        file = fdt->fd[fd];
        if (!file)
                goto out_unlock;
        rcu_assign_pointer(fdt->fd[fd], NULL);
        __put_unused_fd(files, fd);
        spin_unlock(&files->file_lock);
        return filp_close(file, files);

out_unlock:
        spin_unlock(&files->file_lock);
        return -EBADF;
}
```

__put_unused_fd()

```
static void __put_unused_fd(struct files_struct *files,
unsigned int fd) {
      struct fdtable *fdt = files_fdtable(files);
      __clear_open_fd(fd, fdt);
      if (fd < files->next_fd)
            files->next_fd = fd;
}
```

Traditional Unix FD management is implemented here

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

The `write()` system call

- Defined in `fs/read_write.c`

```
SYSCALL_DEFINE3(write, unsigned int fd, const char __user
*, buf, size_t, count) {
      struct fd f = fdget_pos(fd);
      ssize_t ret = -EBADF;
      if (f.file) {
            loff_t pos = file_pos_read(f.file);
            ret = vfs_write(f.file, buf, count, &pos);
            if (ret >= 0)
                  file_pos_write(f.file, pos);
            fdput_pos(f);
      }
      return ret;
}
```

Calls the file ops

```
file->f_op->write(file, p, count, pos)
```

The `read()` system call

- Defined in `fs/read_write.c`

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *,
buf, size_t, count) {
        struct fd f = fdget_pos(fd);
        ssize_t ret = -EBADF;

        if (f.file) {
                loff_t pos = file_pos_read(f.file);
                ret = vfs_read(f.file, buf, count, &pos);
                if (ret >= 0)
                        file_pos_write(f.file, pos);
                fdput_pos(f);
        }
        return ret;
}
```

proc File System

- 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.

proc File System

- `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.

proc File System

- `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.

- All based on the global array `tgid_base_stuff`

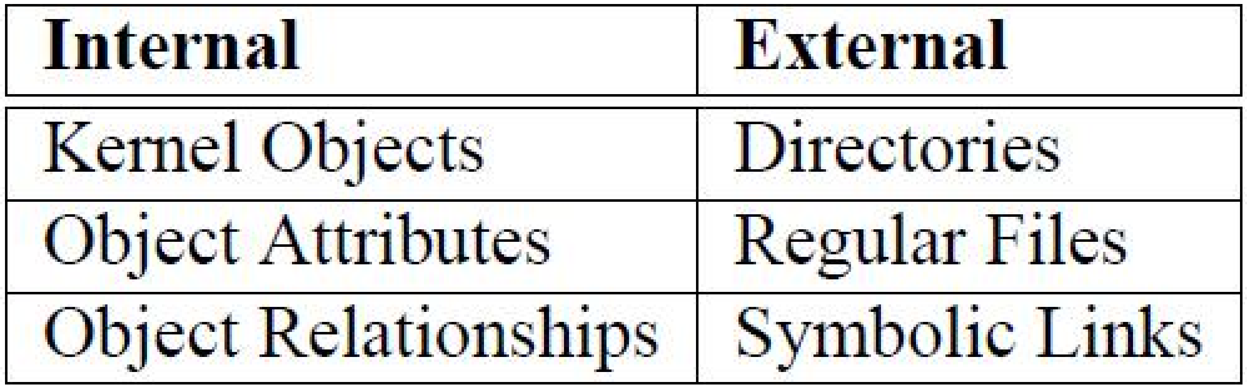
Core data structures for proc

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

```
  struct proc_dir_entry {
      unsigned short low_ino;
      unsigned short namelen;
      const char *name;
      mode_t mode;
      nlink_t nlink;    uid_t uid;    gid_t gid;
      unsigned long size;
      struct inode_operations * proc_iops;
      struct file_operations * proc_fops;
      ...
      read_proc_t *read_proc;
      write_proc_t *write_proc;
      ...
};
```

The Sysfs File System (since 2.6)

- 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



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

Sysfs Core API

```
int sysfs_create_file(struct kobject *, const struct attribute *);

void sysfs_remove_file(struct kobject *, const struct attribute *);

int sysfs_update_file(struct kobject *, const struct attribute *);
```

```
struct attribute {
    char              *name;
    struct module     *owner;
    mode_t            mode;
};
```

The owner field may be set by the caller to point to the module in which the code to manipulate the attribute exists

Kernel Objects (*knobs*)

- Kobjects don't live on their own: they are embedded into objects (think of `struct cdev`)
- They keep a reference counter (`kref`)

```
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);
```

struct kobject

```
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;
};
```

struct kobj_type

```
struct kobj_type {
    void (*release)(struct kobject *);
    struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
```

- A specific object type is defined in terms of the `sysfs_ops` to be executed on it, the defaul attributes (if any), and the `release` function

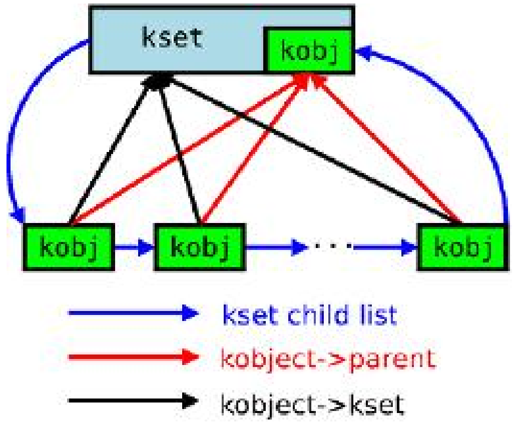
Sysfs Read/Write Operations

- These operations are defined in the kobject thanks to the `struct kobj_type *ktype` member:
  - `struct kobject->ktype->sysfs_ops`

```
struct sysfs_ops {
        /* method invoked on read of a sysfs file */
        ssize_t (*show) (struct kobject *kobj,
                         struct attribute *attr,
                         char *buffer);

        /* method invoked on write of a sysfs file */
        ssize_t (*store) (struct kobject *kobj,
                          struct attribute *attr,
                          const char *buffer,
                          size_t size);
};
```

ksets



```
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

- When a kobject is created it does not immediately appear in Sysfs
- It has to be explicitly added (although the operation can fail):
  - `int kobject_add(struct kobject *kobj);`
- To remove a kobject from Sysfs:
  - `void kobject_del(struct kobject *kobj);`

Device Management

- Any number of devices can be connected to a machine
- The type of devices can also vary significantly
- Everything in Unix is a file:
  - There should be a way to link devices to VFS
- In the end, the management of a device must be carried out by its driver
  - A physical device could eventually generate interrupts

Device 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

Identifying Char and Block Devices

```
$ ls -l /dev/sda /dev/ttyS0
brw-rw---- 1 root disk 8,  0  9 apr 09.31 /dev/sda
crw-rw---- 1 root uucp 4, 64  9 apr 09.31 /dev/ttyS0
```

type

major   minor

Major and Minor Numbers

```
$ ls -l /dev/sd*
brw-rw---- 1 root disk 8, 0  9 apr 09.31 /dev/sda
brw-rw---- 1 root disk 8, 1  9 apr 09.31 /dev/sda1
brw-rw---- 1 root disk 8, 2  9 apr 09.31 /dev/sda2
```

Same driver, different disks or partitions

- 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`

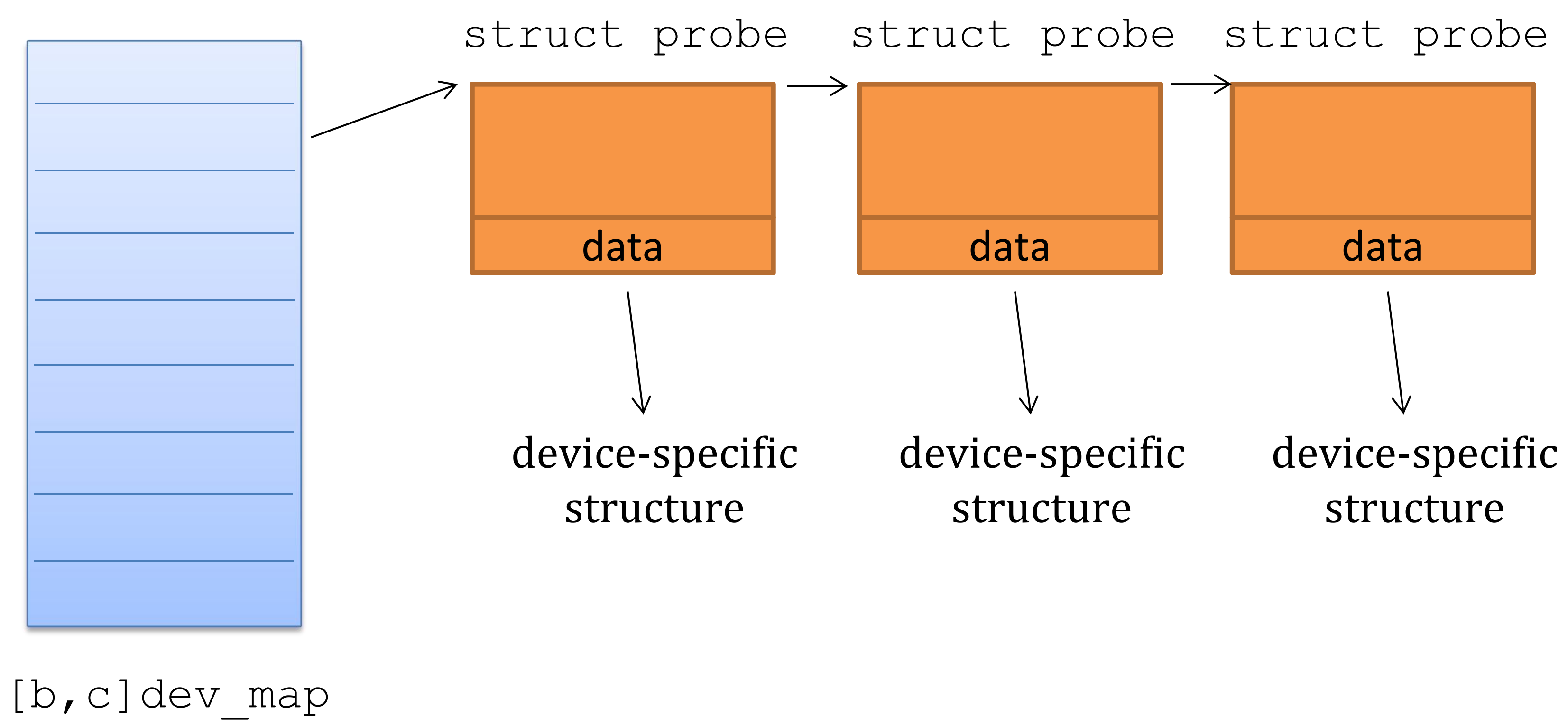
The Device Database

- 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`

```
struct kobj_map {
     struct probe {
          struct probe *next;
          dev_t dev;
          unsigned long range;
          struct module *owner;
          kobj_probe_t *get;
          int (*lock)(dev_t, void *);
          void *data;
     } *probes[255];
     struct mutex *lock;
};
```

← hasing is done as: major % 255

The Device Database


struct probe
struct probe
struct probe
data
data
data
device-specific structure
device-specific structure
device-specific structure
[b,c]dev_map

Device Numbers Representation

- The `dev_t` type keeps both the major and the minor (in `include/linux/types.h`)

```
typedef __u32           __kernel_dev_t;
typedef __kernel_dev_t  dev_t;
```

- In `linux/kdev_t.h` we find facilities to manipulate it:

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

struct cdev

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
} __randomize_layout;
```

Char Devices Range Database

- Defined in `fs/char_dev.c`
- Used to manage device number allocation to drivers

```
#define CHRDEV_MAJOR_HASH_SIZE 255
static struct char_device_struct {
      struct char_device_struct *next;
      unsigned int major;
      unsigned int baseminor;
      int minorct;
      char name[64];
      struct cdev *cdev;
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

Registering Char Devices

- `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 (0 means the choice is up o 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)`

struct file_operations

- It is defined in `include/linux/fs.h`

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t
*);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct
*);
        int (*ioctl) (struct inode*, struct file *, unsigned int,
                        unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        ...
};
```

Registering Device Numbers

- A driver might require to *register* or *allocate* a range of device numbers
- API 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`

Block Devices

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

```
struct gendisk {
      int major;          /* major number of driver */
      int first_minor;
      int minors;         /* maximum number of minors, =1 for
                           * disks that can't be partitioned. */
      char disk_name[DISK_NAME_LEN];/* name of majordriver */
      ...
      const struct block_device_operations *fops;
      struct request_queue *queue;
};
```

- 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)
```

struct block_device_operations

- It is defined in `include/linux/fs.h`

```
struct block_device_operations {
   int (*open) (struct inode *, struct file *);
   int (*release) (struct inode *, struct file *);
   int (*ioctl) (struct inode *, struct file *,
                     unsigned, unsigned long);
   int (*check_media_change) (kdev_t);
   int (*revalidate) (kdev_t);
   struct module *owner;
};
```

- There is nothing here to read and write from the device!

Read/Write on Block Devices

- 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

Linking Devices and the VFS

- The member `umode_t i_mode` in `struct inode` tells the type of the i-node:
  - directory
  - file
  - char device
  - block device
  - (named) pipe
- The kernel function `sys_mknod()` creates a generic i-node
- If the i-inode represents a device, the operations to manage the device are retrieved via the device driver database
- In particular, the i-node has the `dev_t i_rdev` member

The `mknod()` System Call

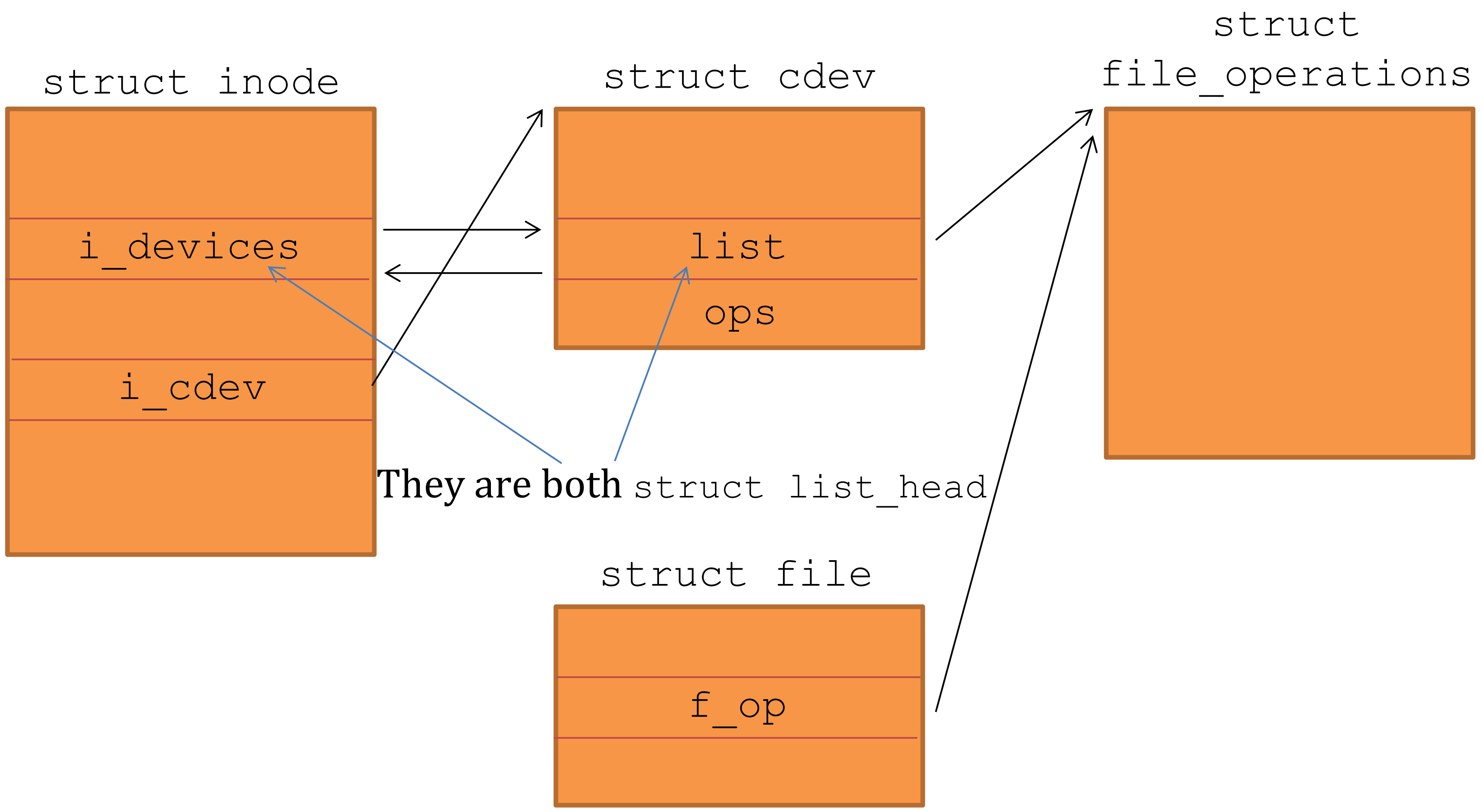
```
int mknod(const char *pathname, mode_t mode, dev_t dev)
```

- `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

i-node to File Operations Mapping



Device Classes

- Devices are organized into "classes"
- A device can belong to multiple classes
- Class membership is shown in `/sys/class/`
  - Block devices are automatically placed under the "block" class
  - This is done automatically whe the gendisk structure is registered in the kernel
- Most devices don't require the creation of new classes

Managing New Classes

- Manage classes, we instantiate and register the struct class declared in `linux/device.h`

```
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)
```

Managing Devices in Classes

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

printf-like way to specify the device node in `/dev`

- `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`

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

Creating Device Attribute Files

- Again placed in `/sys`

- `int device_create_file(struct device *dev,const struct device_attribute *attr)`

- `void device_remove_file(struct device *dev, const struct device_attribute *attr)`

udev

- 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

udev 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"`