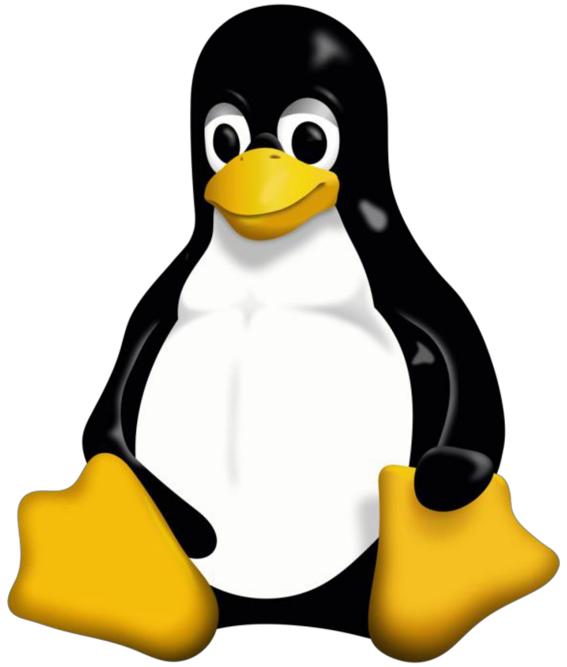


Linux Character Device Driver

Sunbeam Infotech

File

- File is collection of data and information on storage device.
- File = Data (Contents) + Metadata (Information)
- File metadata is stored in inode (FCB).
  - Type, Mode, Size, User & Group, Links.
  - Timestamps, Info about data blocks.
- File data is stored in data block(s).
- File types: regular, directory, link, pipe, socket, character and block.
- Directory file contains directory entries for each sub-directory and file in it. Each dentry contains inode number & ~~data block.~~ file name

mode: $\underset{u}{\underline{rwx}}$ $\underset{g}{\underline{rwx}}$ $\underset{o}{\underline{rwx}}$

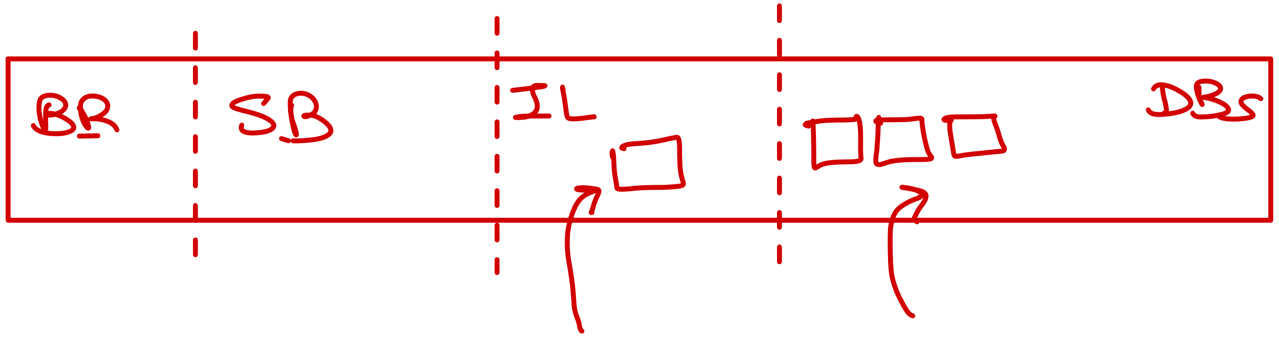
directory - special file.
- contains info about all files & subdirs in that directory (folder).
- directory entry (dentry)
  - ↳ inode num
  - ↳ file name

File system

- File system is way of organizing files on the disk. (any storage).
- File systems have
  - Book block
  - Super block
  - Inode list
  - Data blocks

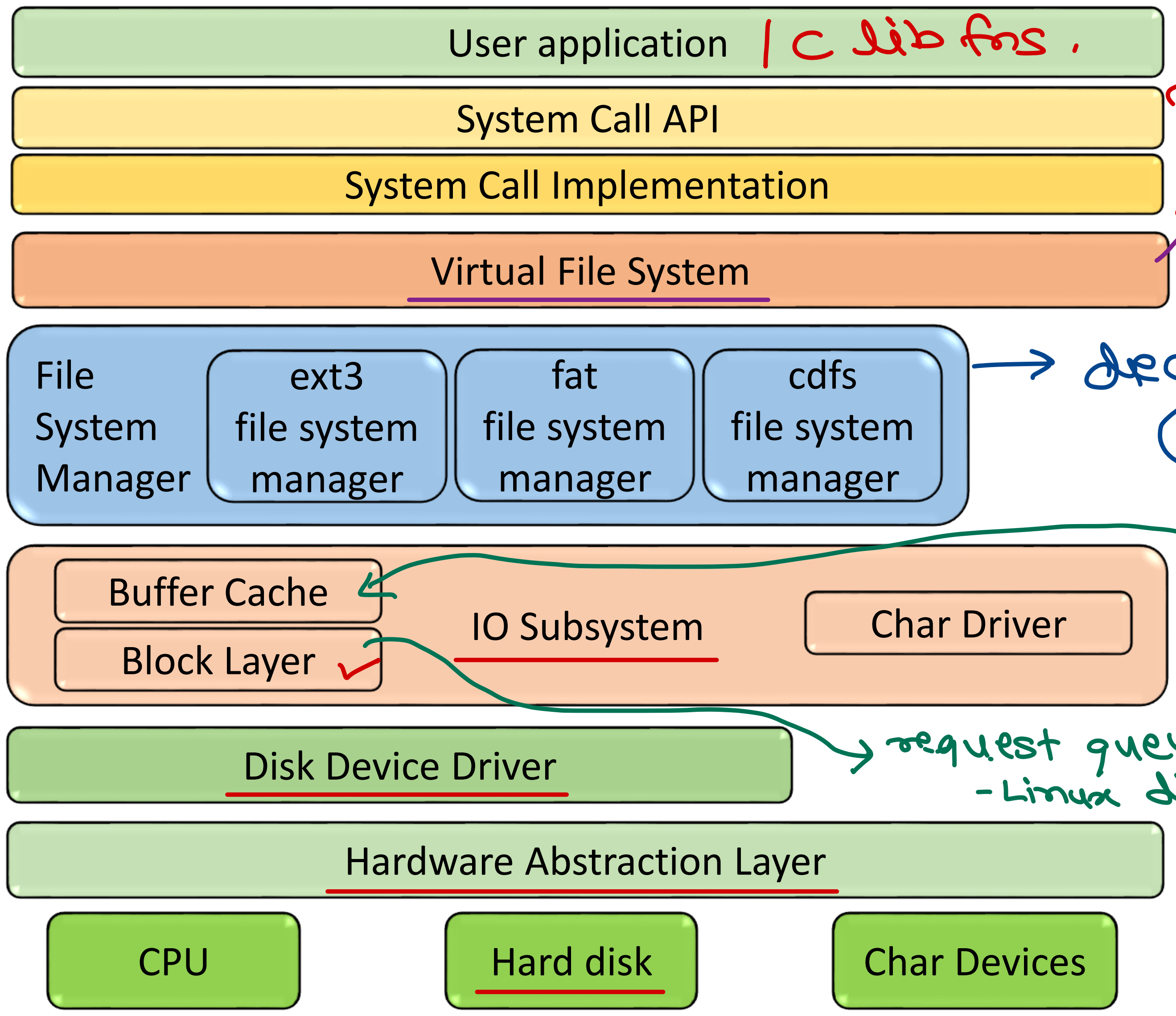
FAT, NTFS, EXT2/3/4, XFS, ReiserFS, HFS, ...

- Content/arrangement of these components will change FS to FS.
- Different FS use different algorithms/ data structures to allocate disk.
- File system layout & operations are handled by file system manager using IO subsystem and device driver.



file = inode + data blocks

File System architecture



} syscalls - open(), close(), read(), write(), lseek(), link(), ..

→ redirect fs op req to appropriate fs mgr.

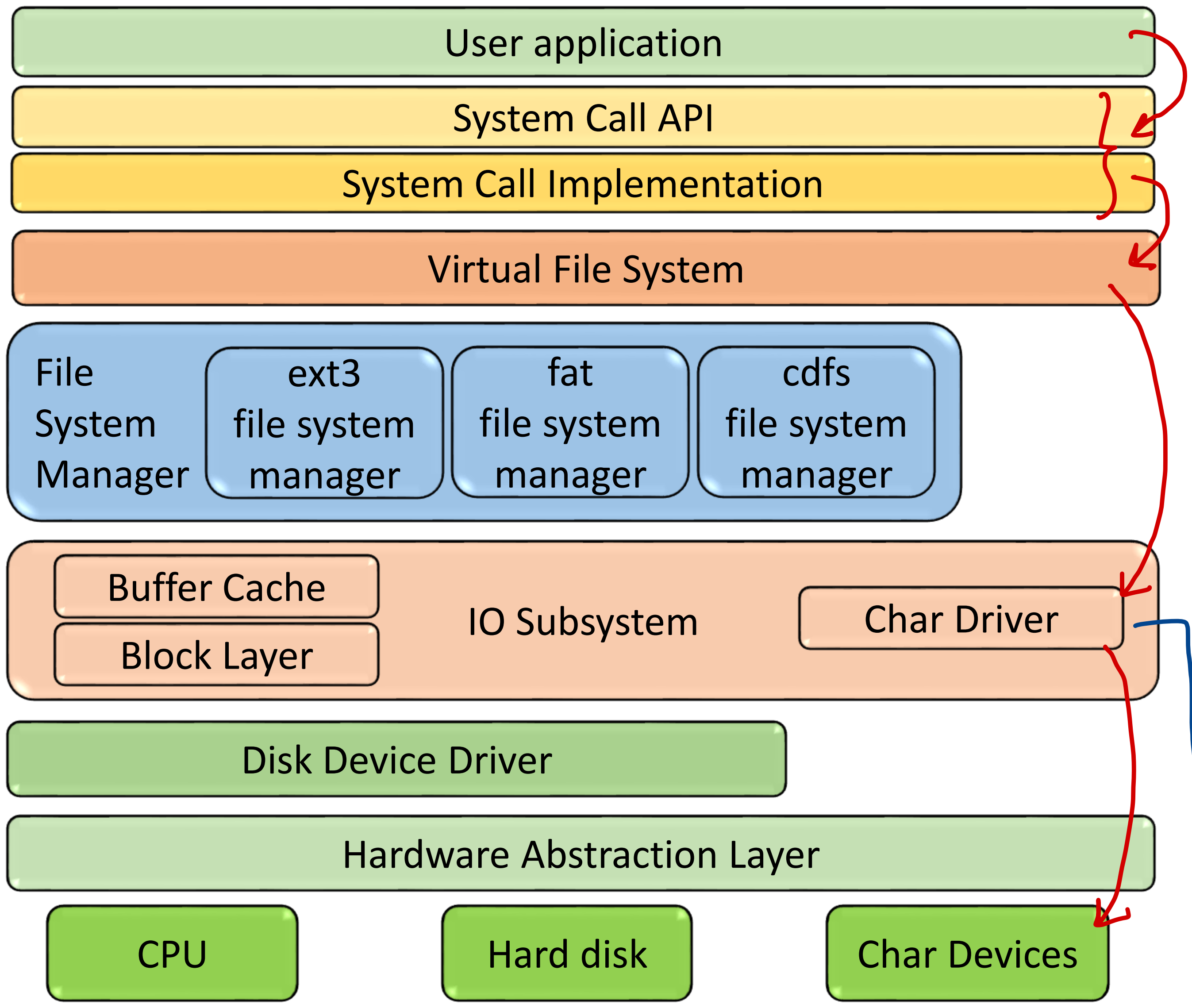
→ decides FS ops to be performed. (read/write to which parts of disk).

holds memory bufs for disk read/write ops.

→ request queue (disk sched algo)
- Linux disk sched: elevator (scan)

File System architecture

Char device driver execution



} syscalls

→ VFS structures (PLKA - VFS).

- ↳ [redacted] (inode cache)
- ↳ [redacted] (open file table)
- ↳ [redacted] (dentry cache)
- ↳ [redacted] (open file desc table)
- ↳ [redacted]
- ↳ [redacted]

→ kernel module + file io ops.

File IO System Calls

- File system operations deals with metadata (inode and dentry).
  - stat(), link(), symlink(), unlink(), mkdir(), …
- File IO operations deals with data/contents.
  - * open(), close(), read(), write(), lseek(), ioctl(), …
- fd = open("filepath", flags, mode); ✓
  - Open or create a file.
- close(fd); ✓
  - Close opened file.
- ret = read(fd, buf, nbytes); ✓
  - Read from file into user buffer.
- ret = write(fd, buf, nbytes); ✓
  - Write from file into user buffer.
- newpos = lseek(fd, offset, whence); ✓
  - Change file current position.

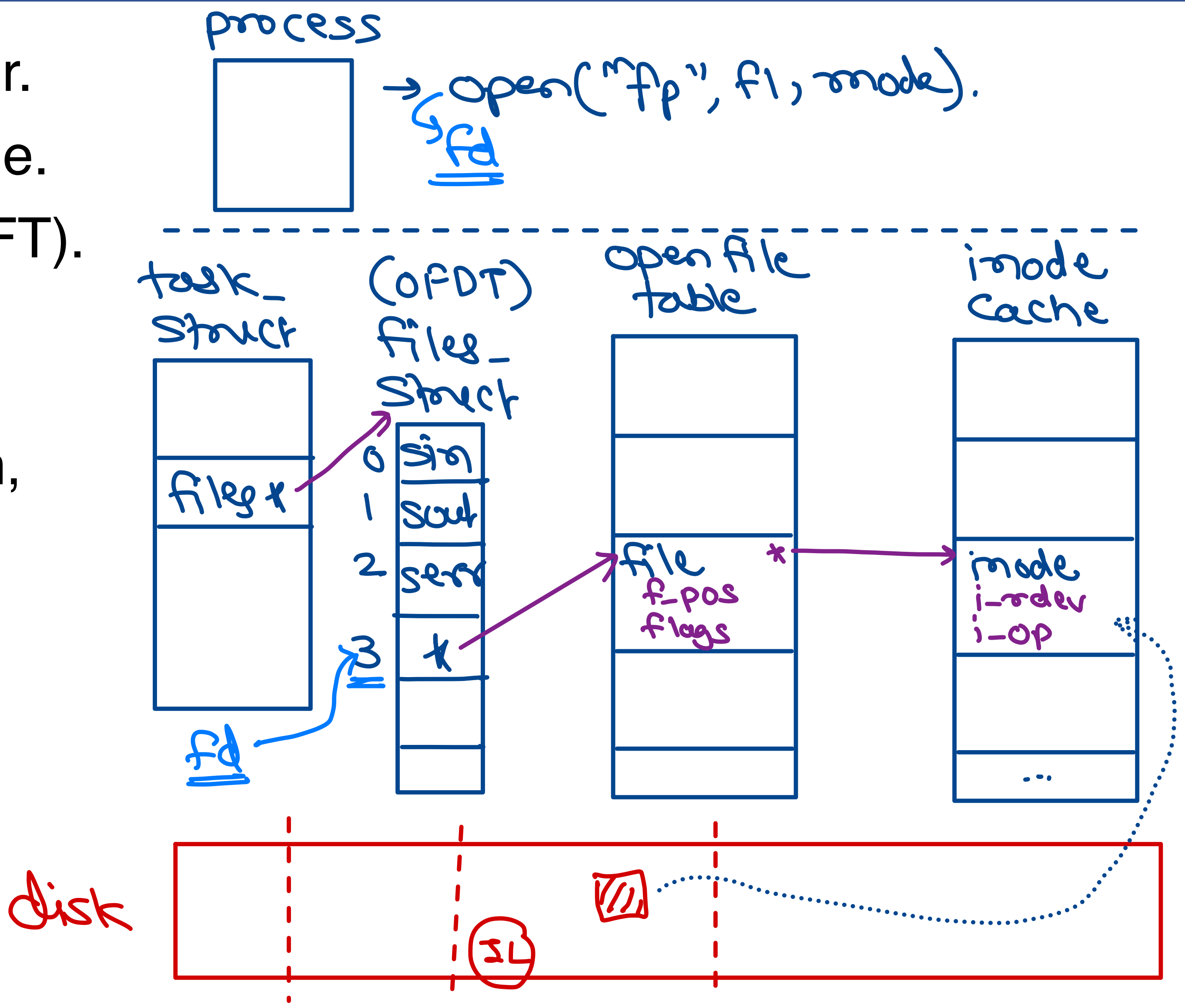
File copy program

- 1. Open source file for reading.
- 2. Get file metadata (mode).
- 3. Open destination for writing.
    - Truncate if exists.
    - Create if doesn’t exists.
- 4. Read n bytes from source file.
- 5. Write them into destination file.
- 6. Repeat 4-5 until file ends.
- 7. Close destination file.
- 8. Close source file.

fd = open("filepath", flags, mode);

namei().

- Convert path name into inode number.
- Load inode into in-memory inode table.
- Make an entry into open file table (OFT).
- Add pointer to OFT into open file descriptor table (OFDT).
- Return index of OFDT to the program, known as file descriptor (FD).
- struct inode
- struct file



ret = read(fd, buf, nbytes);

- Get current file position from OFT entry.
- Calculate file logical block number and block byte offset.
- Convert file logical block to disk block using data blocks information in inode.
- Check if disk block is available in buffer cache.
- In not available, instruct disk driver to load the block from disk.
- Disk driver initiate disk IO and block the current process.
- Read desired part of block from buffer cache into user buffer. Update file position into OFT.
- Repeat for subsequent file blocks, until desired number of bytes are read.
- Return number of bytes allocated.

ret = write(fd, buf, nbytes);

- Get current file position from OFT entry.
- Calculate file logical block number and block byte offset.
- Convert file logical block to disk block using data blocks information in inode.
- Check if disk block is available in buffer cache.
- In not available, instruct disk driver to load the block from disk.
- Disk driver initiate disk IO and block the current process.
- Overwrite on intended part of block in buffer cache (from user buffer). Update file position into OFT. Mark the block in buffer cache as dirty, so that it will be flushed on the disk.
- Repeat for subsequent file blocks, until desired number of bytes are written.
- Return number of bytes allocated.

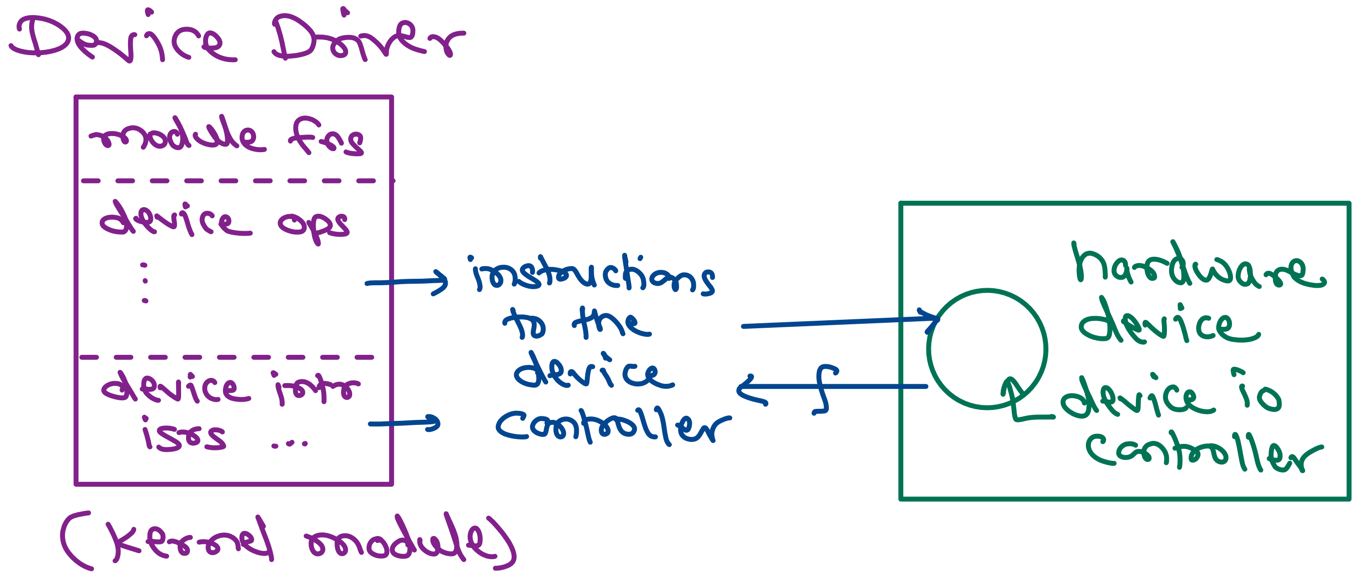
close(fd) and newpos = lseek(fd, offset, whence);

- close(fd);
  - Decrement reference count in OFT entry.
  - If count is zero, release the file/resources.

- newpos = lseek(fd, offset, whence);
  - Get current file position from OFT entry.
  - Calculate new file position.
  - Update file position in OFT entry.

Linux device drivers

- Device driver is a kernel module that instructs device controllers to perform the operations and also handles interrupts generated from it.



Linux device drivers

- Character device drivers ✓
  - Char devices transfer data in byte by byte manner. So device drivers are implemented to read/writer data as stream of bytes. They support four major operations i.e. open(), close(), read() and write(). Example: Serial port, parallel port, keyboard, tty, etc.

gpio, i2c eeprom, ...

- Block device drivers
  - Block devices transfer data as bunch of bytes i.e. block by block. Size of block is typically 512 Bytes. Support major operations open(), close(), read(), write() and lseek(). Example: All mass storage devices.
- Network device drivers
  - Network drivers are responsible for packets transmit and receive, however network protocols are implemented up in network stack. Unlike character and block devices network device entry is not done under /dev.

Overview of character device driver

- Char device driver is a kernel module.
- Driver initialization includes
  - Allocate device number ✓
  - Create device class & file ✓
  - Init cdev object & add it. ✓
- Driver de-initialization includes
  - Release cdev ✓
  - Destroy device file & ~~close~~ class ✓
  - Release device number ✓
- Implement minimal device operations
  - ✓ open() and release() ✓
  - ✓ read() and write() ✓

Pseudo character device driver

char buf[32];

- A memory buffer is treated as device.
- open() operation – do nothing.
- release() operation – do nothing.
- write() operation
  - Take data from user buffer and write into device (memory) buffer.
  - Update current file position.
  - Return number of bytes successfully written.
- read() operation
  - Take data from device (memory) buffer and write to user buffer.
  - Update current file position.
  - Return number of bytes successfully read.

Device numbers

- Each device is uniquely identified by a major number and minor number.
- (Kernel < 2.6): 16 bit device number
  - 8-bit major + 8-bit minor.
- (Kernel >= 2.6): 32 bit device number
  - 12-bit major + 20-bit minor.
  - 32 bit = ~~8 bit + 12 bit + 12 bit~~
    8 + 8 + 12 + 4
- Device number represented as $\boxed{\text{dev\_t.}}$ data type
  - MKDEV(major, minor) } create dev no.
  - MAJOR(devno)
  - MINOR(devno) } extract major & minor nums from given dev no.

8 bit	8 bit
major	minor

8 bit	8 bit	12 bit	4 bit
major	minor	minor	maj

12-bit major = 4096 types

Register character device

- (1) Register/allocate character device number.
  - register_chrdev_region(devno, count, “name”);
  - alloc_chrdev_region(dev, baseminor, count, “name”);

✓ to register in kernel — out param — 0 — 1 (single device instance).

- Makes entry into
  - struct char_device_struct *chrdevs[…];
  - hashed by major number of device.
- Allocated device numbers can be seen under /proc/devices.
- Allocated device number can be released using
  - unregister_chrdev_region (devno, count); — 1
- Char device should also be added into cdev_map.

```
struct char_device_struct {
        struct char_device_struct *next;
        unsigned major;
        unsigned baseminor;
        int minorct;
        char name[...];
        struct cdev *cdev;
}
```

/proc/devices

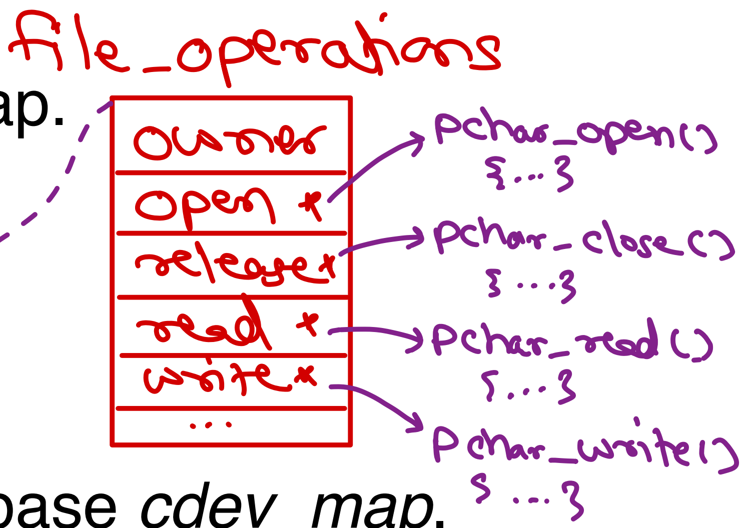
Register character device

- Each char device is represented by struct cdev.
- *cdev_map* is global array (hash table) to keep track of all char devices.
- cdev_map is object of struct kobj_map.
  - key = device major number
  - hash function = major % 255
  - value = struct probe
    - void *data = struct cdev
- (2) Add device into char device database *cdev_map*.
  - cdev_init(&cdev, &fops);
  - cdev_add(&cdev, devno, dev_count);

dev no

1

- Added device can be removed.
  - cdev_del(&cdev);



```
struct kobj_map {
   struct probe {
      dev_t dev;
      unsigned long range;
      ...
      void *data;
   } *probes[255];
   ...
};
```

struct kobj_map cdev_map;

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

Device class and device file

older Linux kernel → device files created using mknod cmd → mknod() syscall.

- Char device added in kernel space should be accessible from user space.
- Traditionally device is created using *mknod*.
  - mknod /dev/devname c major minor

/sys /dev

- In newer kernel device is accessible from *sysfs* and *devfs*.
- Device class is created under /sys/class (category of device)
  - pclass = class_create(module, "class_name"); → THIS_MODULE ↗ Not required in kernel 6.x.
- Device file is create under /sys/devices/ virtual/class_name/devname and /dev/devname
  - pdev = device_create(pclass, NULL /*parent*/, devno, NULL /*drvdata*/, "devname", ...);

maj+min

device file name

pchar %d

0/1

- Device class and device file are destroyed using.
  - device_destroy(pclass, devno);
  - class_destroy(pclass);

Char device operations

- Char device driver provide char device operations and register with the system.
- Driver device operations can be do one of the following:
    - Overwrite default function
    - Provide additional functionality
    - Minimal placeholder
    - NULL – use default/no implementation
- These operations are specified as callback functions in struct file_operations and associated with struct cdev. All functions have pre-defined prototype and purpose.
    - int (*open)(struct inode *, struct file *);
    - int (*release)(struct inode *,struct file *);
    - ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);
    - ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);
    - loff_t (*llseek)(struct file *, loff_t, int);
    - long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    - …

SUNBEAM

Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>