# Project 3: An Introduction to File Systems

COP4610

Florida State University

# Introduction

- The goal of project 3 is to understand
  - basic file system design and implementation
  - file system testing
  - data serialization/de-serialization

- At the end of the project, you will feel like a file system expert!

# Outline

- Background
  - Setting up your environment
  - Mounting file systems
- Project 3
  - Specification
  - Downloading and testing file system image
  - General FAT32 data structures
  - Endian-ness

# Environment Setup

Get ready for Project 3!

# Project 3 Environment

- Must develop inside Linux environment with root access
- Make sure that they compile in the lab machines, which runs the most recent version of Linux Mint
- I will be grading your projects inside a similar environment

# Project Environment

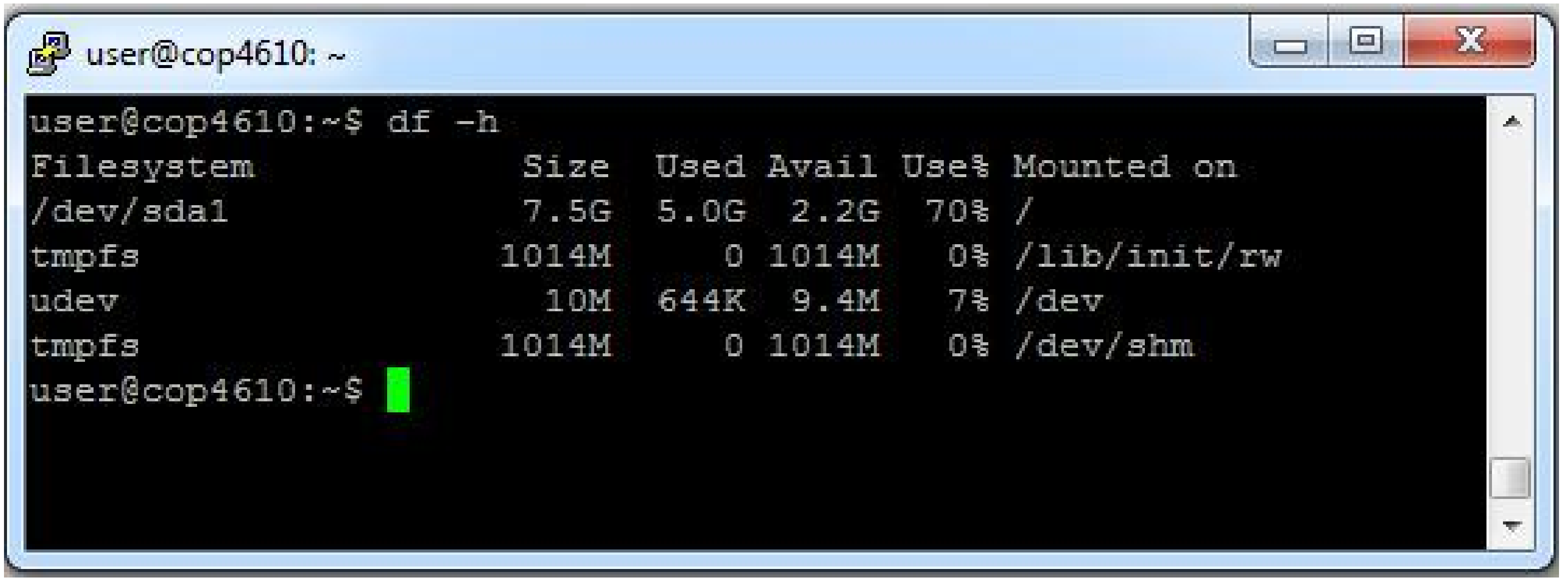
- Kernel version no longer matters
- Entire project will be in userspace, not kernel!
  - Please use debuggers, they will save you time
    - gdb, ddd, others…
- Programming language is still C

# Running out of room?

- You must have at least 64MB free, plus room for your source code
- To see how much room you have left inside your machine, issue the following command:

```
$> df -h
```

# df -h



```
user@cop4610:~$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda1             7.5G  5.0G  2.2G  70% /
tmpfs                1014M     0 1014M   0% /lib/init/rw
udev                   10M  644K  9.4M   7% /dev
tmpfs                1014M     0 1014M   0% /dev/shm
user@cop4610:~$
```

- /dev/sda is root file system mounted on “/”
  - Has 2.2GB currently available

# Mounting File Systems

# Unix File Hierarchy

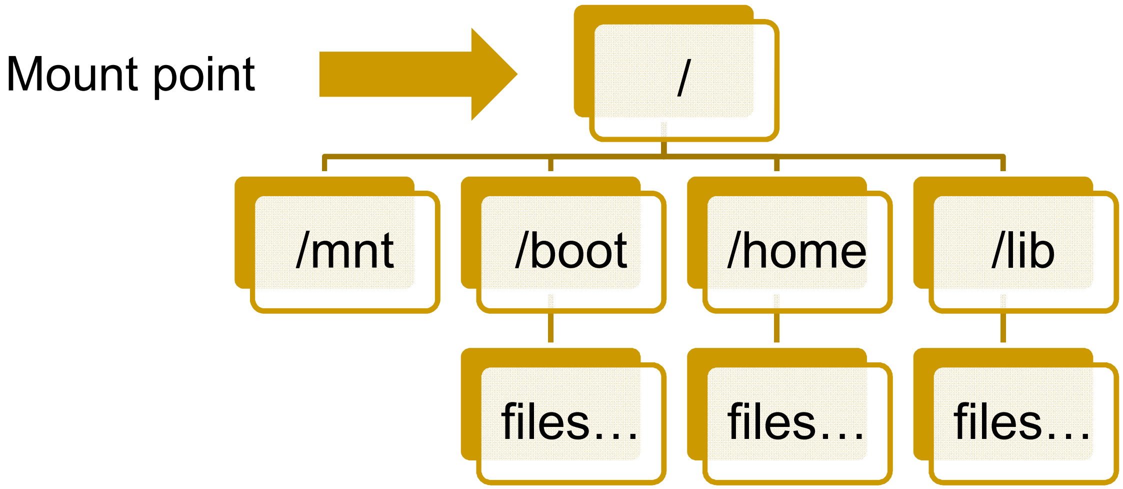
- All files accessible in a Unix system are arranged in one big tree
  - Also called the ***file hierarchy***
  - Tree is rooted (starts) at **/**
- These files can be spread out over several devices
- The **mount** command serves to attach the file system found on some device to the big file tree

# ‘mount’ command

- `mount`
- `mount <device> <mount directory>`

- Typing ‘mount’ without arguments shows you what is mounted and where
- Second example attaches a device or partition to a directory
  - Must have root privileges

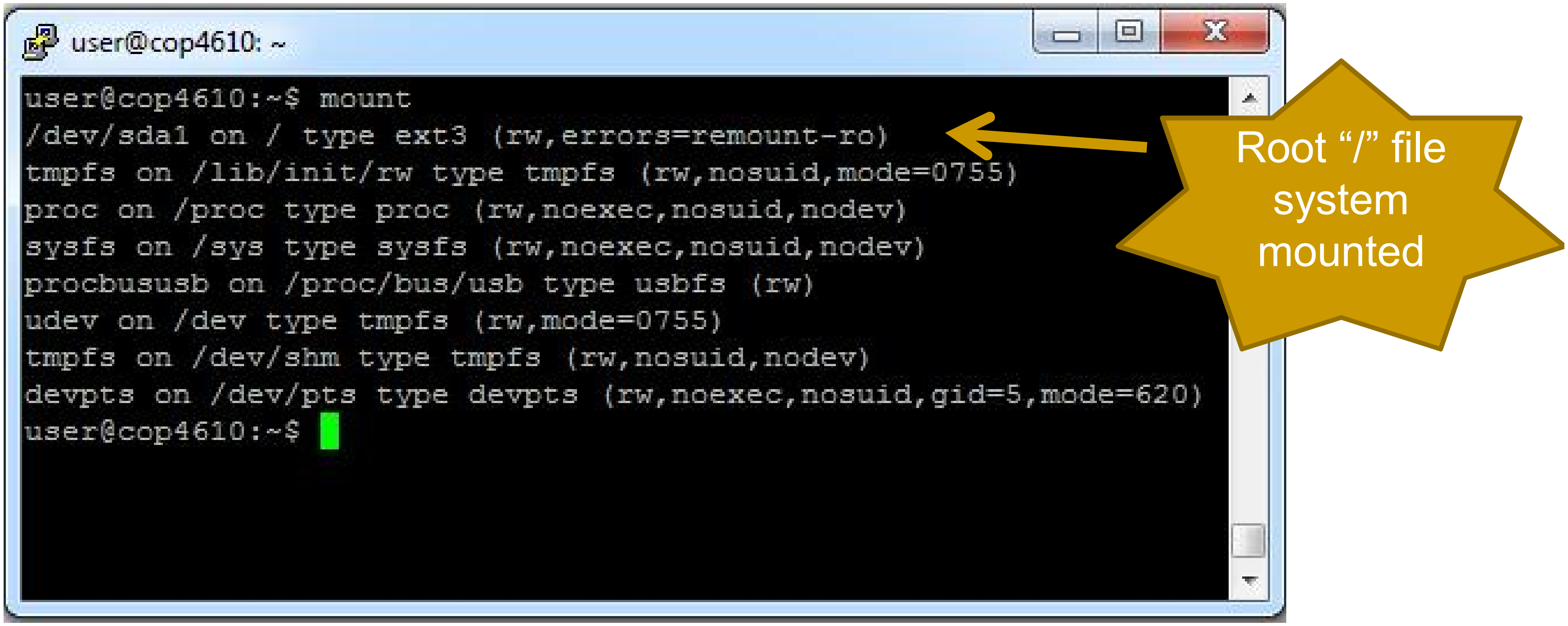
# Mount Example



The device sda partition 1 is mounted at “/”. All files and dirs below “/” come from this device.

# Mount Example

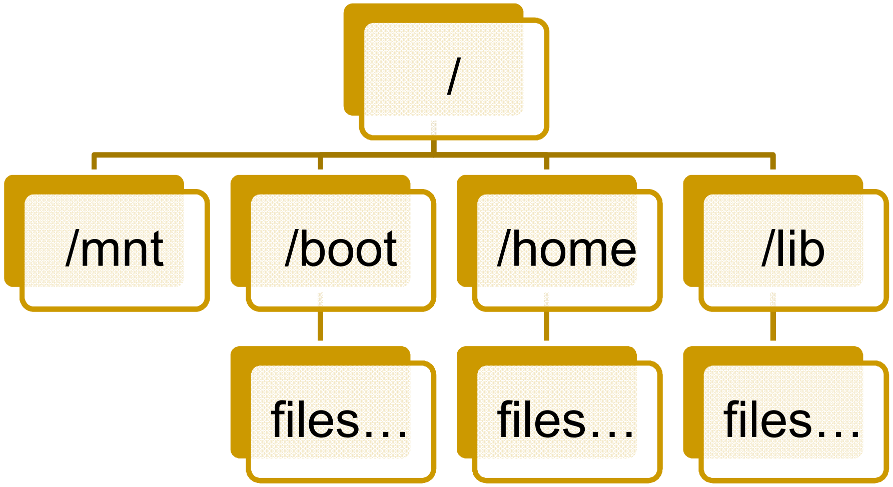
- Type command ‘mount’ without any arguments to see what is mounted and where



```
user@cop4610:~$ mount
/dev/sda1 on / type ext3 (rw,errors=remount-ro)
tmpfs on /lib/init/rw type tmpfs (rw,nosuid,mode=0755)
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
procbususb on /proc/bus/usb type usbfs (rw)
udev on /dev type tmpfs (rw,mode=0755)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=620)
user@cop4610:~$
```

Root “/” file system mounted

# Mount Example



/

/mnt

/boot

/home

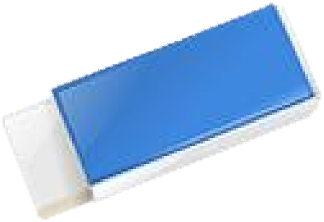
/lib

files…

files…

files…

/dev/sda1

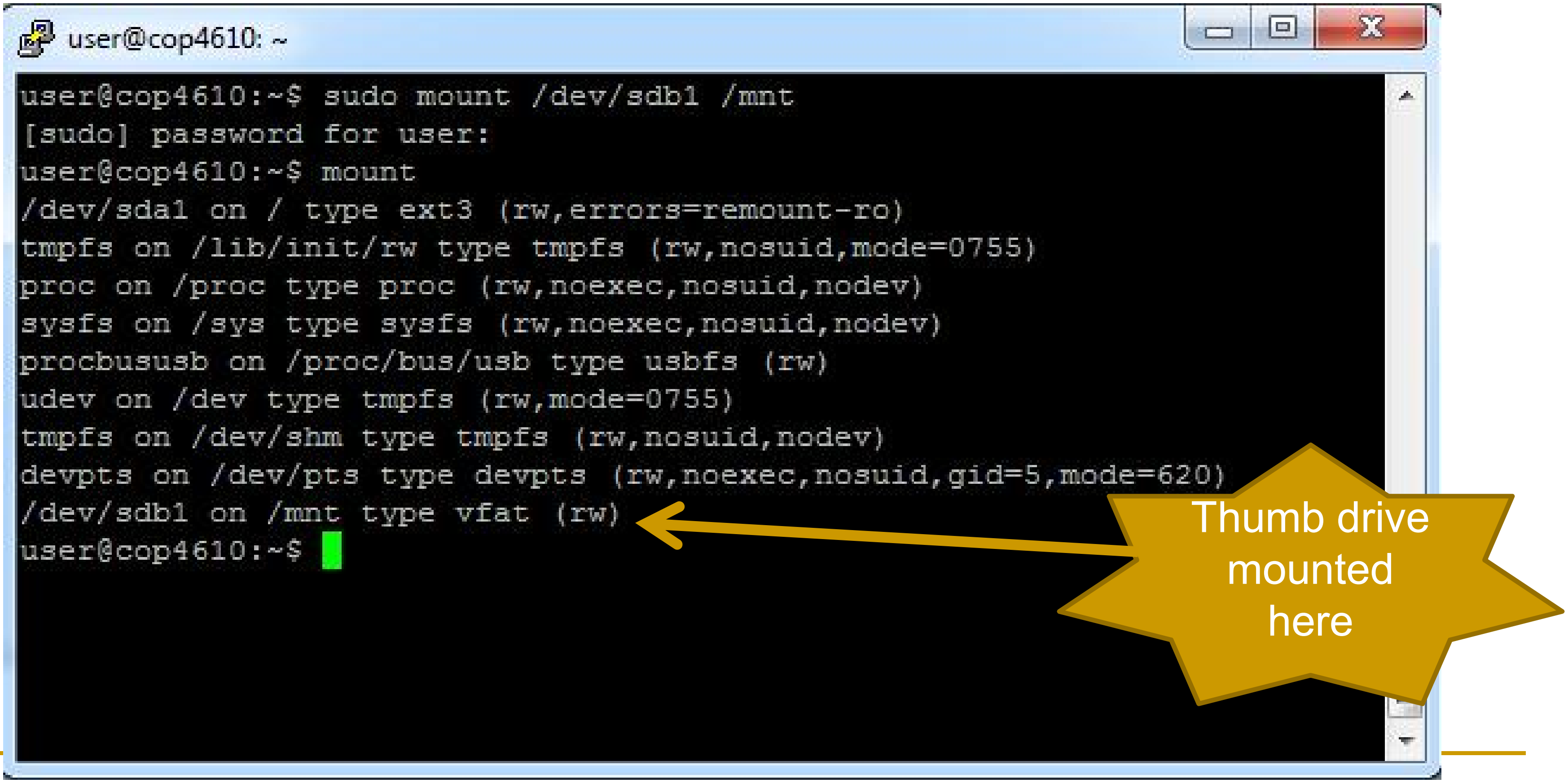


/dev/sdb1

Now suppose we attach a thumb drive and want our thumb drive files accessible under /mnt…

# Mount Example

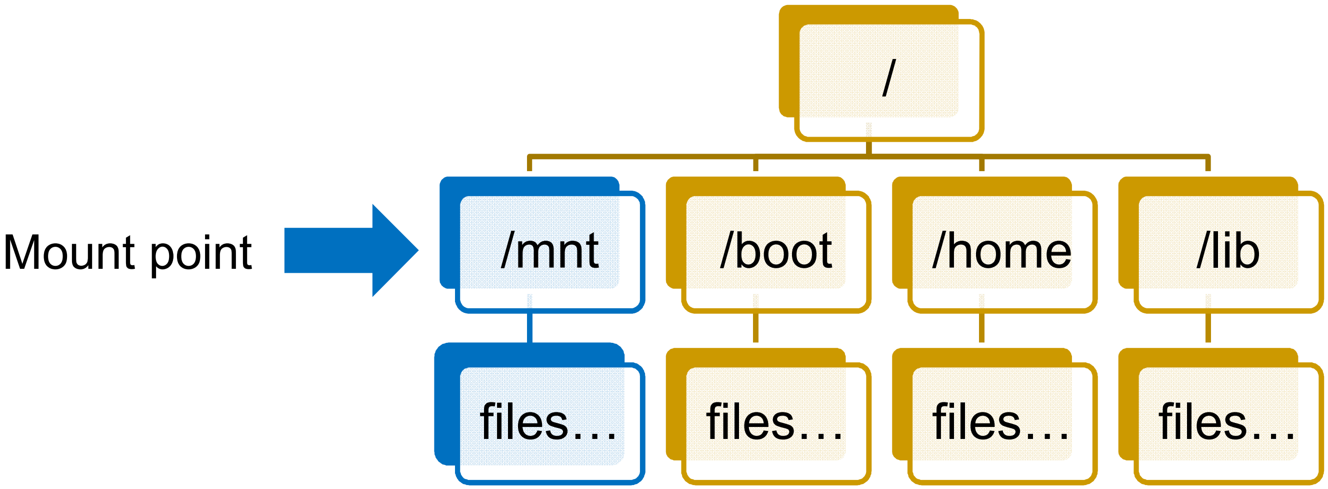
- sudo mount /dev/sdb1 /mnt

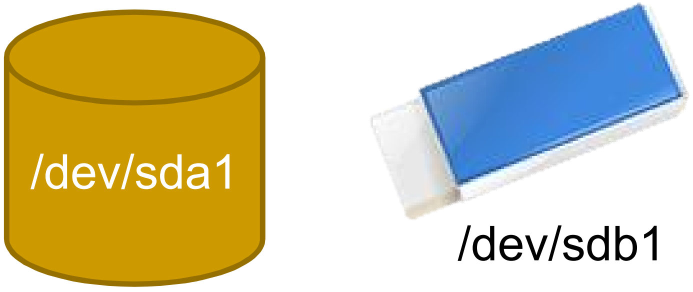


```
user@cop4610:~$ sudo mount /dev/sdb1 /mnt
[sudo] password for user:
user@cop4610:~$ mount
/dev/sda1 on / type ext3 (rw,errors=remount-ro)
tmpfs on /lib/init/rw type tmpfs (rw,nosuid,mode=0755)
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
procbususb on /proc/bus/usb type usbfs (rw)
udev on /dev type tmpfs (rw,mode=0755)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=620)
/dev/sdb1 on /mnt type vfat (rw)
user@cop4610:~$
```

Thumb drive mounted here

# File Hierarchy Example





Files from the thumb drive are now accessible under /mnt

# Un-mount Command

-
  ```
  umount <dir>
  ```

- In our example where the thumb drive was mounted at /mnt, we can issue

  -
    ```
    $> umount /mnt
    ```
  - Must have root privileges

# Figuring out names of devices

- /etc/fstab – Has list of devices and file systems that get auto-mounted on boot

# Project 3

More than you wanted to know about FAT32..

# Project 3

- You will create a user-space utility to manipulate a FAT32 file system image
  - No more kernel programming!
- Utility must understand a few basic commands to allow simple file system manipulation
- Utility must not corrupt the file system and should be robust

# FAT32 Manipulation Utility

Utility will only recogniz the following built-in commands:

- open
- close
- create
- rm
- size
- cd
- ls
- mkdir
- rmdir
- read
- write

# File System Image

- Manipulation utility will work on a pre-configured FAT32 ***file system image***
  - Actually a file

- File system image will have raw FAT32 data structures inside
  - Just like looking at the raw bytes inside of a disk partition

# File System Image

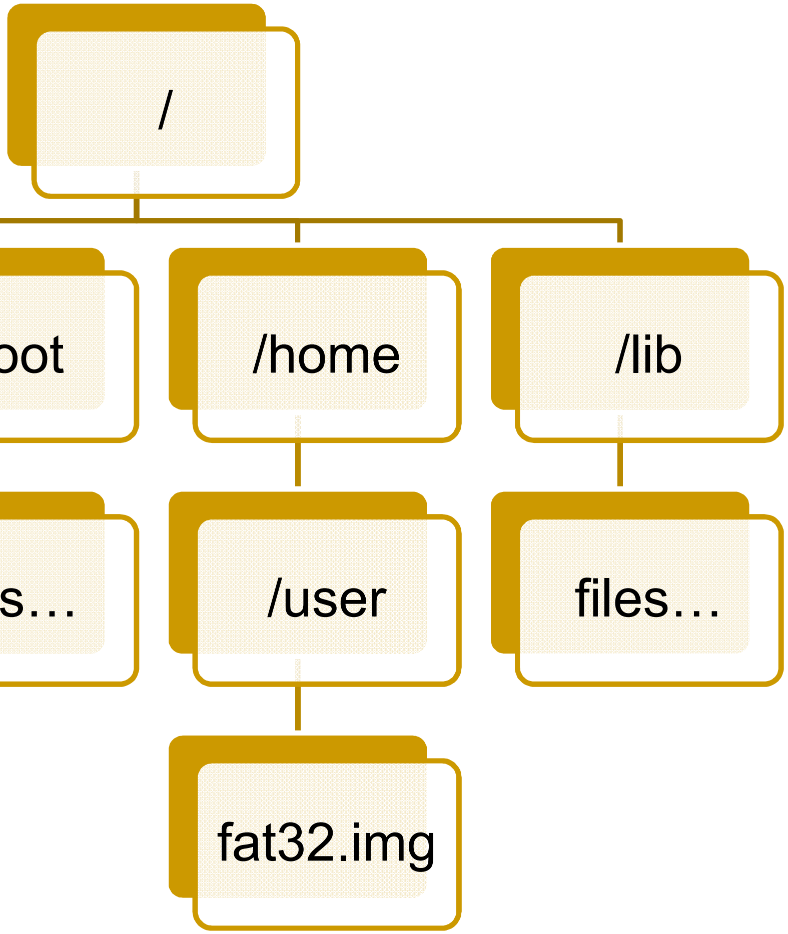
- Your FAT32 manipulation utility will have to
  - Open the FAT32 file system image
  - Read parts of the FAT32 file system image and interpret the raw bytes inside to service your utility’s file system commands…

…just like a file system!

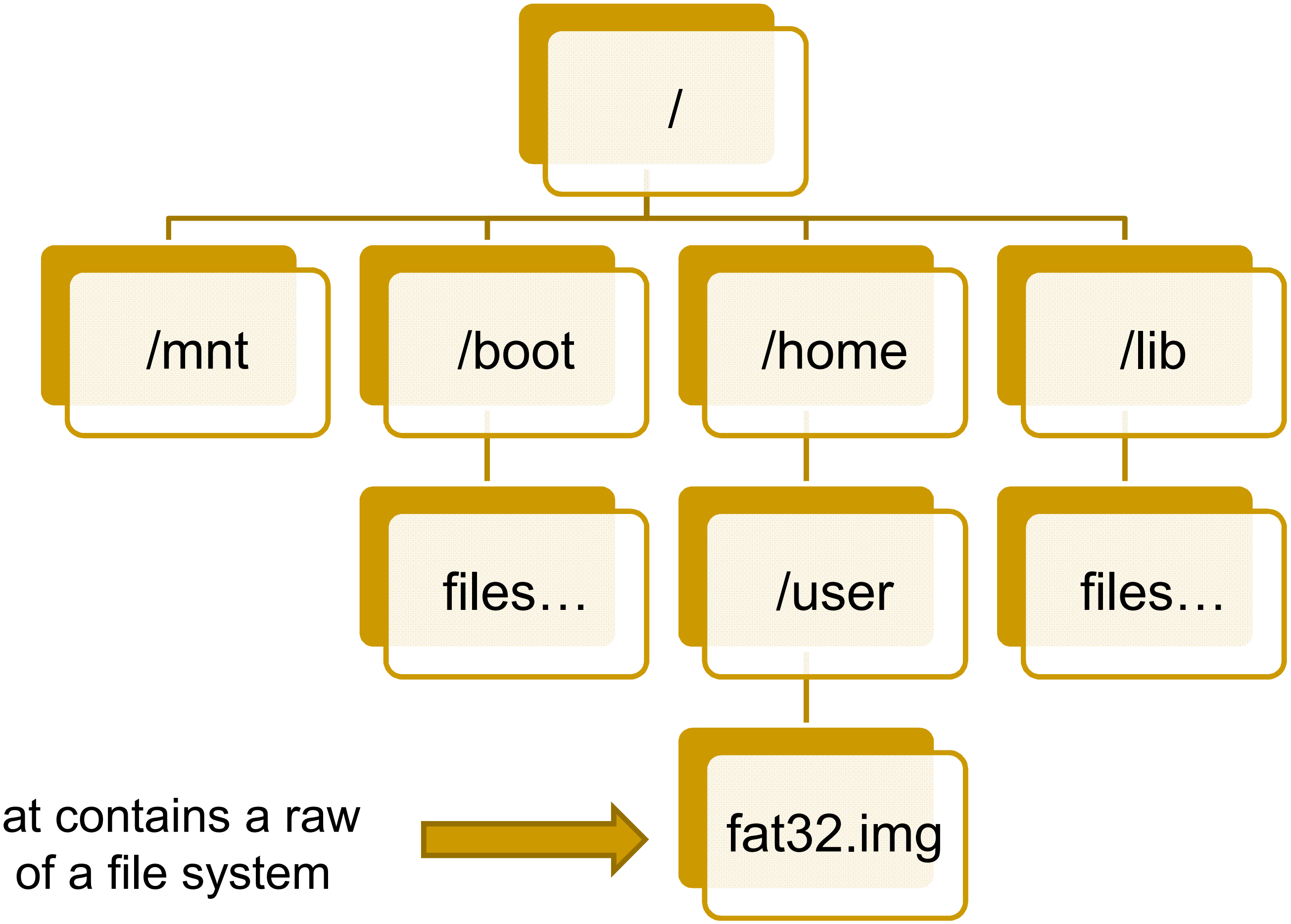
# File System Image

- Sometimes you may want to check that you haven’t corrupted your file system image, or that you can add or write files successfully
  - Mount your file system image with the OS FAT32 driver
  - Just like the file system image is a device

# File System Image Mount Example

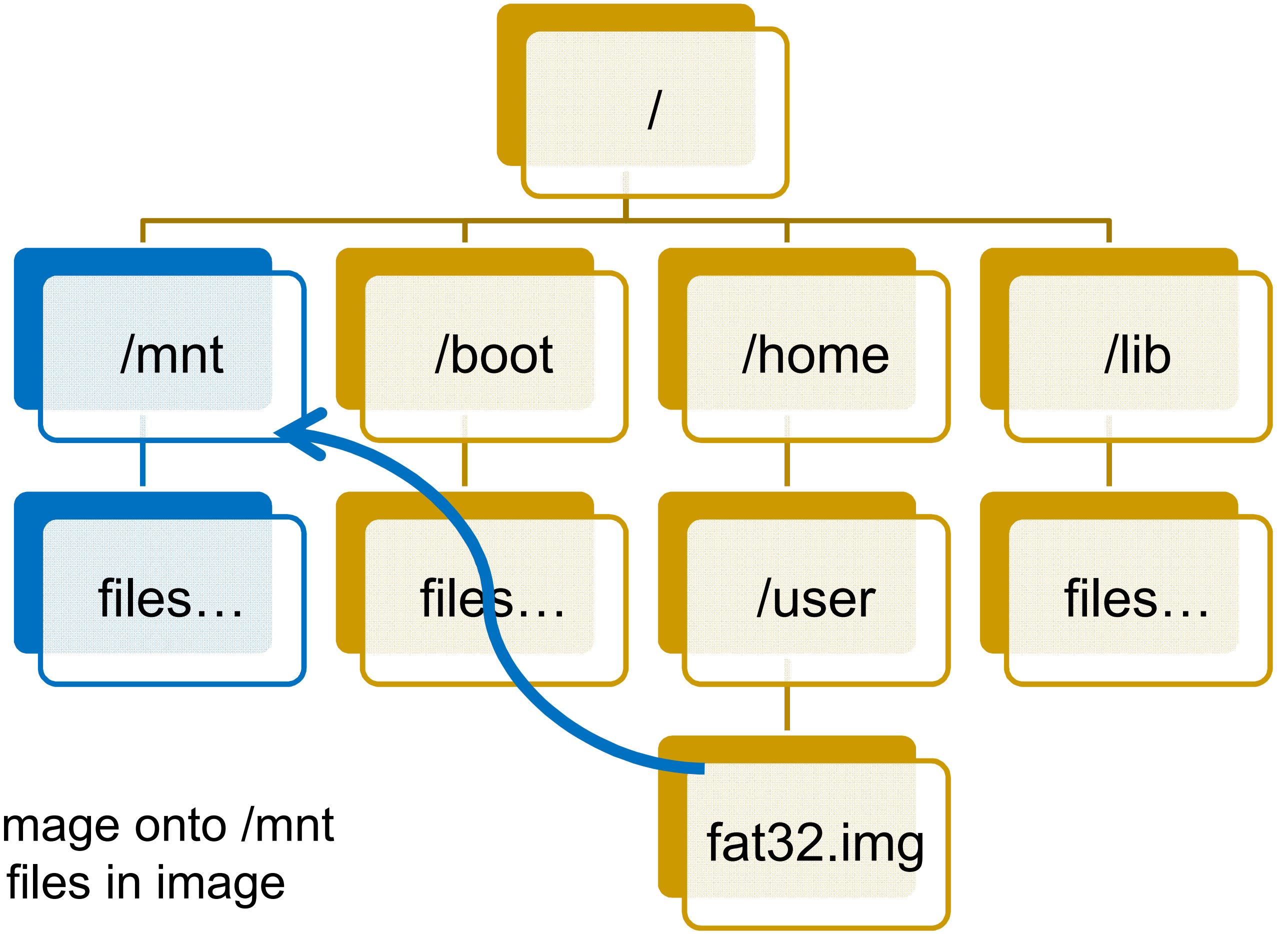


# File System Image Mount Example



File that contains a raw image of a file system

# File System Image Mount Example



Mount image onto /mnt
to read files in image

# File System Image Mount Example

```
$> sudo mount fat32.img /mnt
$> cd /mnt
```

- `fat32.img` is your image file
- `/mnt` is your mounting directory
- Once the file is mounted, you can go into the /mnt directory and issue all your normal file system commands like:
  - ls, cat, cd, …

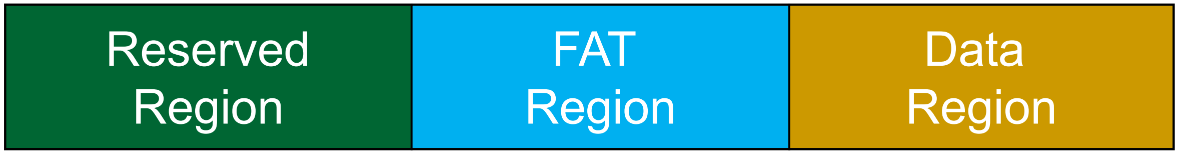
# General FAT32 Data Structures

# Terminology

- ***Byte*** – 8 bits of data, the smallest addressable unit in modern processors
- ***Sector*** – Smallest addressable unit on a storage device. Usually this is 512 bytes
- ***Cluster*** – FAT32-specific term. A group of sectors representing a chunk of data
- ***FAT*** – Stands for *file allocation table* and is a map of files to data

# FAT32 Disk Layout

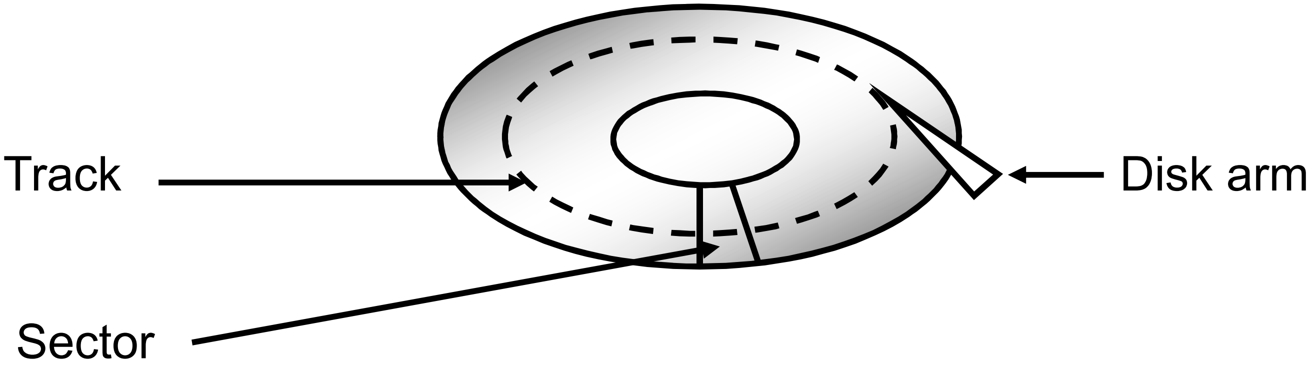
- 3 main regions…



Reserved Region

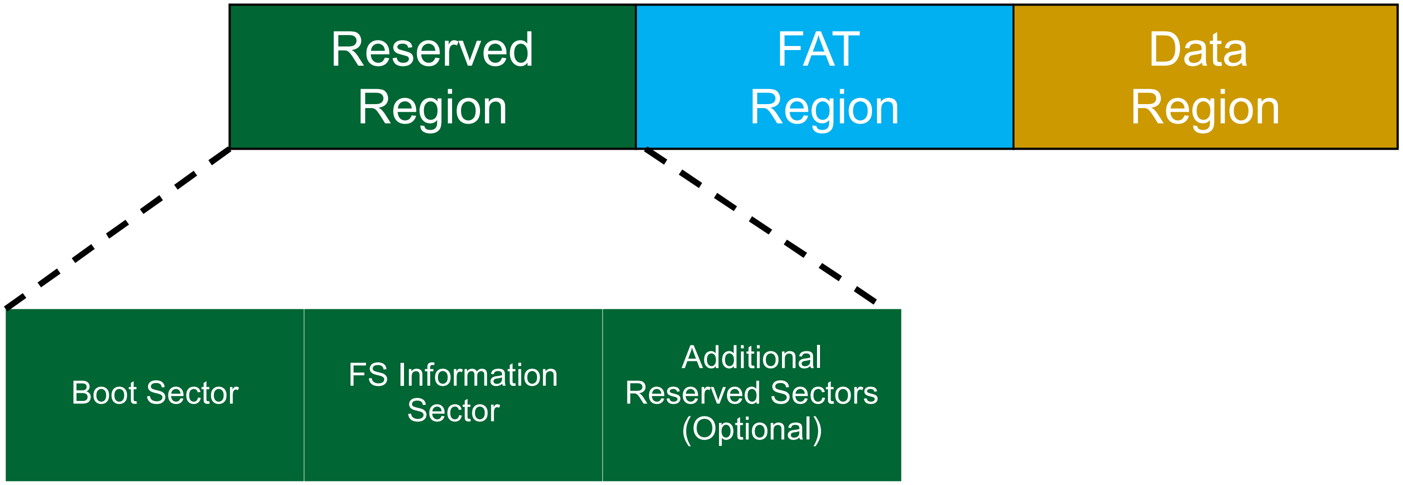
FAT Region

Data Region



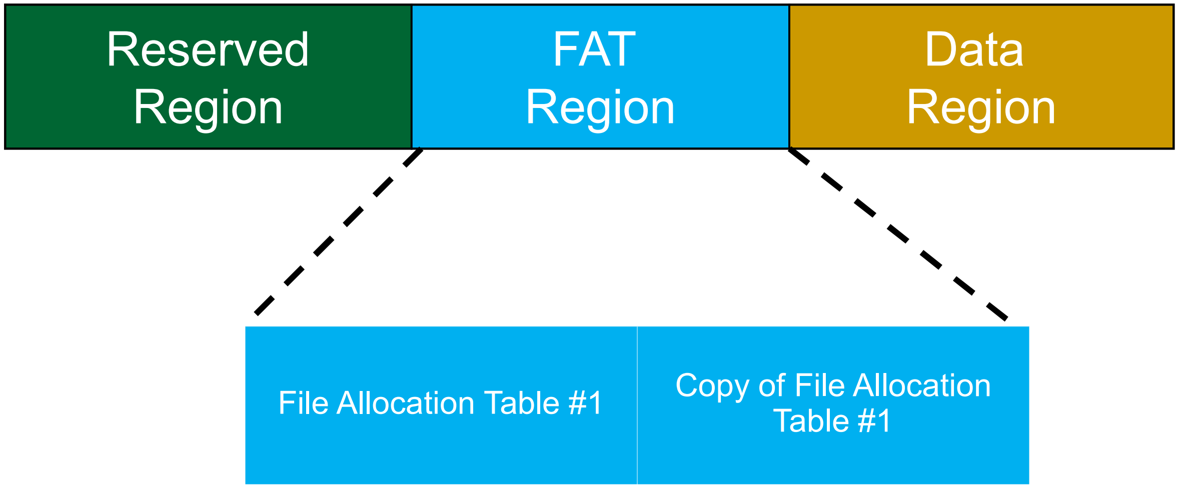
# Reserved Region

- **Reserved Region** – Includes the boot sector, the extended boot sector, the file system information sector, and a few other reserved sectors



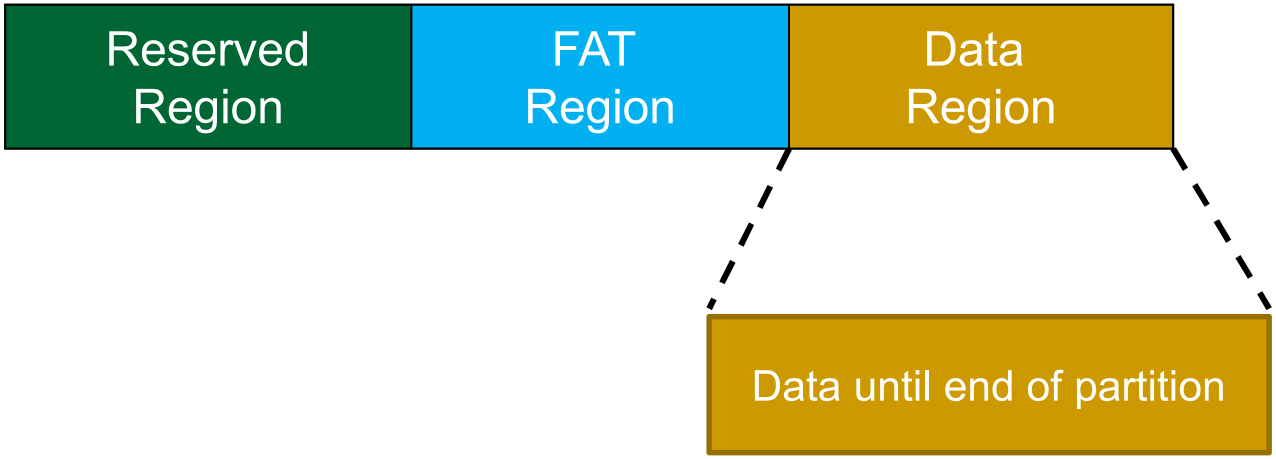
# FAT Region

- **FAT Region** – A map used to traverse the data region. Contains mappings from cluster locations to cluster locations



# Data Region

- **Data Region** – Using the addresses from the FAT region, contains actual file/directory data



# Endian

Big or little?

# Machine Endianness

- The endianness of a given machine determines in what order a group of bytes are handled (ints, shorts, long longs)
  - Big-endian – most significant byte first
  - Little-endian – least significant byte first

- This is important to understand for this project, since FAT32 is always formatted as little-endian

# FAT32 Endianness

- The following are a few cases where endianness matters in your project:
  - Reading in integral values from the FAT32 image
  - Reading in shorts from a FAT32 image
  - Combining multiple shorts to form a single integer from the FAT32 image
  - Interpreting directory entry attributes

# Endian Example (English Version)

- Imagine you can only communicate three letters at a time, and your word is “RAPID”
- Big-endian
  - 1. RAP
  - 2. ID
  - Word = RAPID
- Little-endian
  - 1. PID
  - 2. RA
  - Word = PIDRA (come again?)

# Endian Example (data version)

- short value = 15; /* 0x000F */
- char bytes[2];
- memcpy(bytes, &value, sizeof(short));
- In little-endian:
    - bytes[0] = 0x0F
    - bytes[1] = 0x00
- In big-endian:
    - bytes[0] = 0x00
    - bytes[1] = 0x0F

# Endian Example (data version 2)

- int value = 13371337; /* 0x00CC07C9 */
- char bytes[4];
- memcpy(bytes, &value, sizeof(int));
- In little-endian:
  - bytes[0] = 0xC9
  - bytes[1] = 0x07
  - bytes[2] = 0xCC
  - bytes[3] = 0x00
- In big-endian:
  - bytes[0] = 0x00
  - bytes[1] = 0xCC
  - bytes[2] = 0x07
  - bytes[3] = 0x09

# Visualizing Example 2
# Value = 13371337 (0x00CC07C9)

| index | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| little endian | 0xC9 | 0x07 | 0xCC | 0x00 |
| big endian | 0x00 | 0xCC | 0x07 | 0xC9 |

# Additional Project 3 Information

- **Group project**. 3 People in each group. Each group member will receive same grade.

- **Deadline:** December 1. Late penalties apply. Absolutely no submission after December 6 11:59 pm.

# Until Next Time

- Set up your environment
- Download the image file
- Practice mounting the image file with the OS FAT32 drivers
  - Make sure you can cd into /mnt and read/write to the files
- Read over the FAT32 Specification