Project 3:
An Introduction to File Systems

COP 4610 / CGS 5765

Principles of Operating Systems

Introduction

- Project 3 learning objectives
  - File system design and implementation
  - File system testing
  - Data serialization/de-serialization

Project 3 Expectations

- Teams of **2**
  - Partner is **required**
- Program executes in user-space, not kernel
- Source code
  - C/C++ only
  - Standard C/C++ libraries only

Project 3 Expectations

- Testing environment for grading
  - MCH 202 lab machines
  - You may develop on another machine just make sure your program works on the machines in the lab
- May require a demo

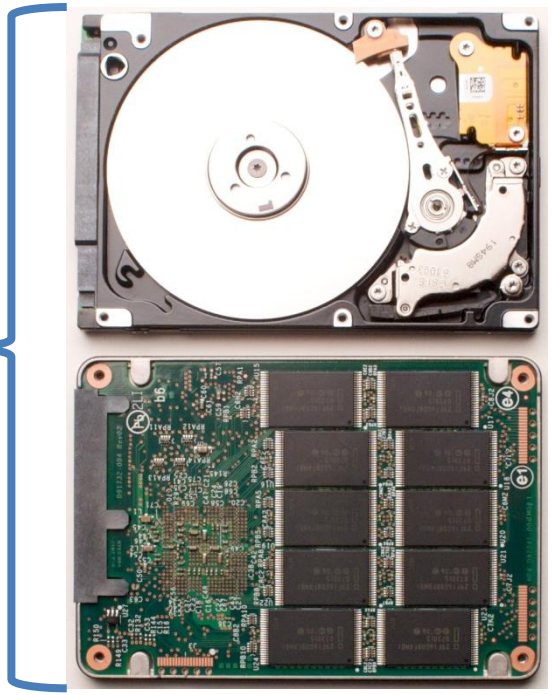
OVERVIEW

Typical Goals of File System Design

- Means for users (e.g., apps) to access data
  - Store data
  - Locate and retrieve data
- Uses persistent media (e.g., SSD, hard disks)
- Consistency (e.g., after power failure)
- Good performance

Persistent Storage

- Typically accessed as 1-dimenstional array of data units



http://download.intel.com/pressroom/images/InsideHDDSSD.jpg

Design Questions

- How best to use persistent storage?
- Where to store data
  - Fragmentation
  - Performance
    - What abstract data structure to use?
    - Timing characteristics of storage device
- Additional information (metadata)
  - User
    - E.g., name of file, date stamps
  - Filesystem
    - E.g., pointers to locate specific data, journal
- Other considerations
  - Guarantees to user
    - E.g., data recoverable when file closed
  - Media corruption/failure
    - E.g., duplication

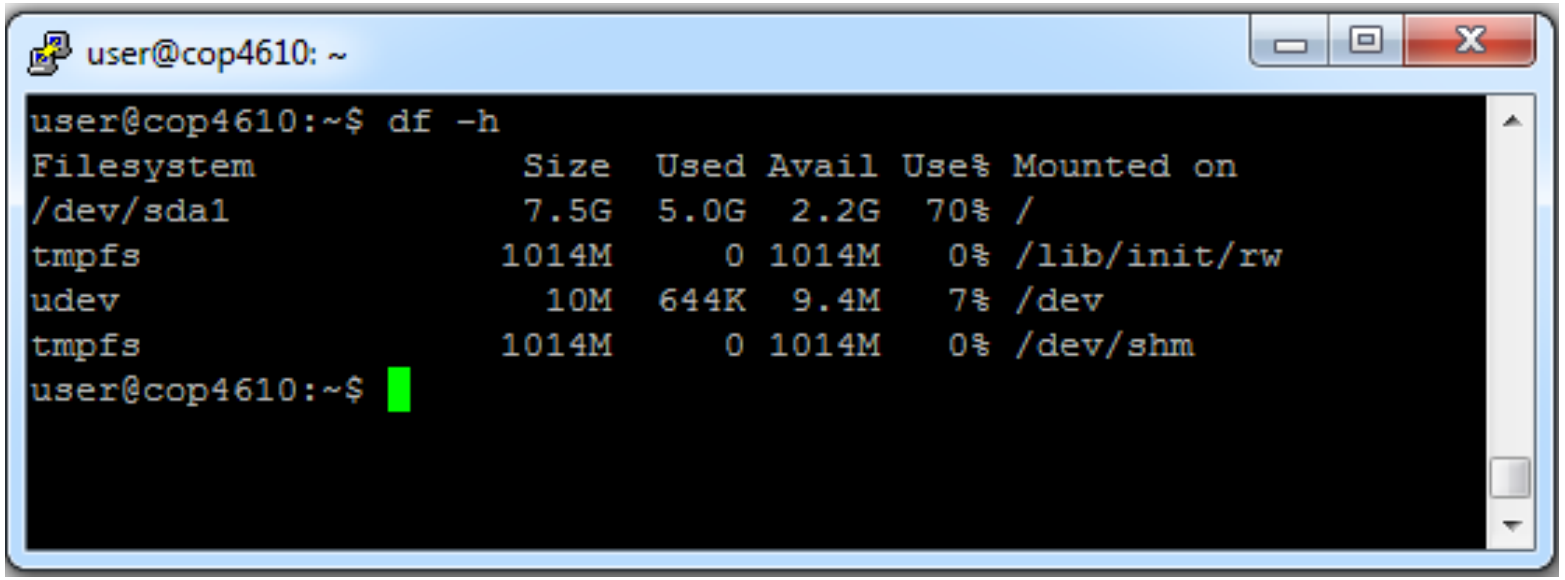
REAL-WORLD EXAMPLE

Running out of storage space?

- At least 64MB free plus space for your source code
- To see how much storage space you have remaining, issue the following command:

```
$> df -h
```

df -h



- **/dev/sda1**
  - root file system (mounted at "/")
  - 2.2GB currently available

MOUNTING FILE SYSTEMS

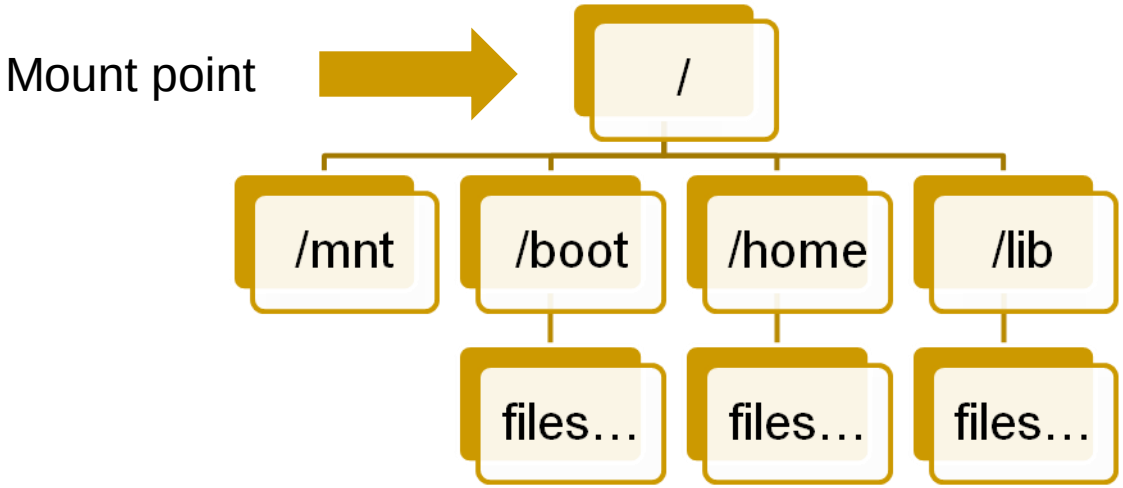
Unix File Hierarchy

- All files accessible in a UNIX system are arranged in one big tree
  - Also called the ***file hierarchy***
  - Tree is rooted (starts) at ***/***
- Files may be spread across several devices
- The **`mount`** utility serves to attach a filesystem to the file hierarchy
  - E.g., mount filesystem on usb device

‘mount’ utility

- mount
- mount <device> <mount directory>
- ‘mount’ without arguments
- What device (e.g., partition) is mounted and where
- Second example attaches a device or partition to a directory
- May require root privileges

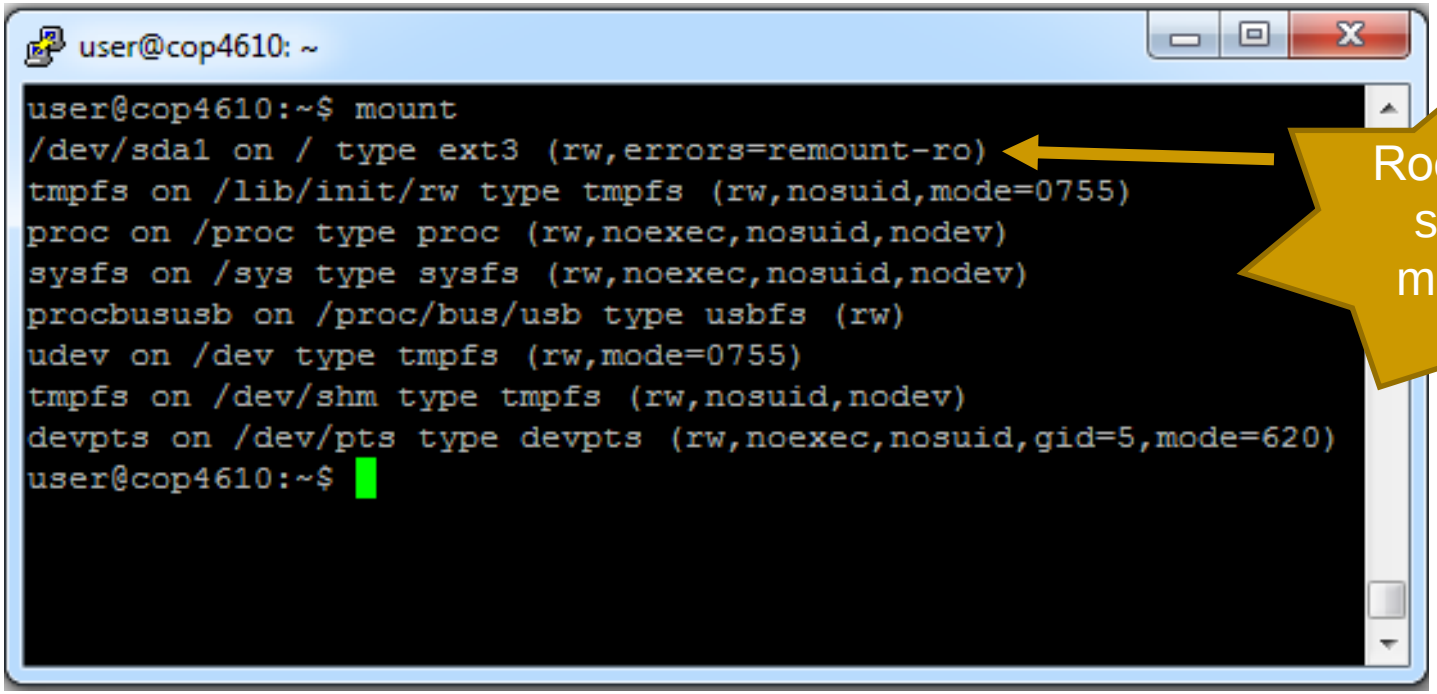
Mount Example



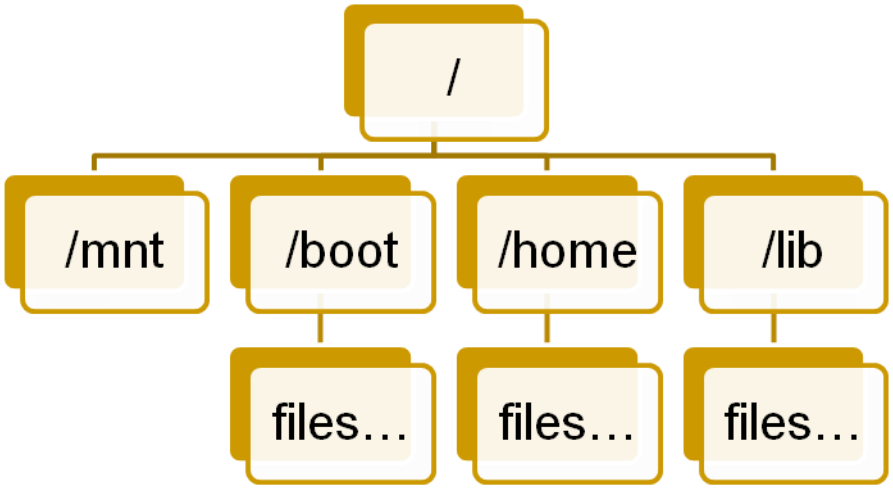
The device sda partition 1 is mounted at “/”. All files and dirs below “/” come from this device.

Mount Example

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



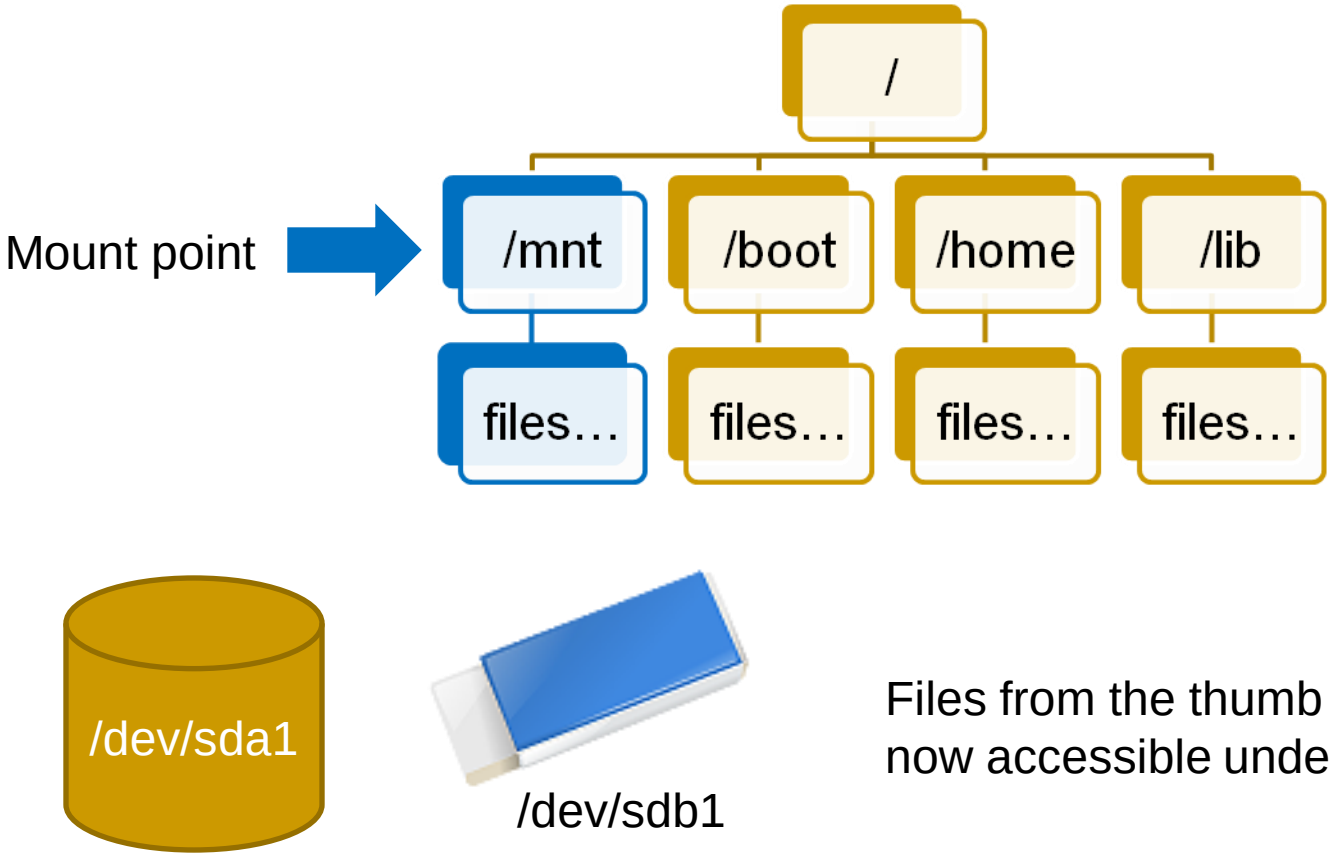
Mount Example



/dev/sdb1

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

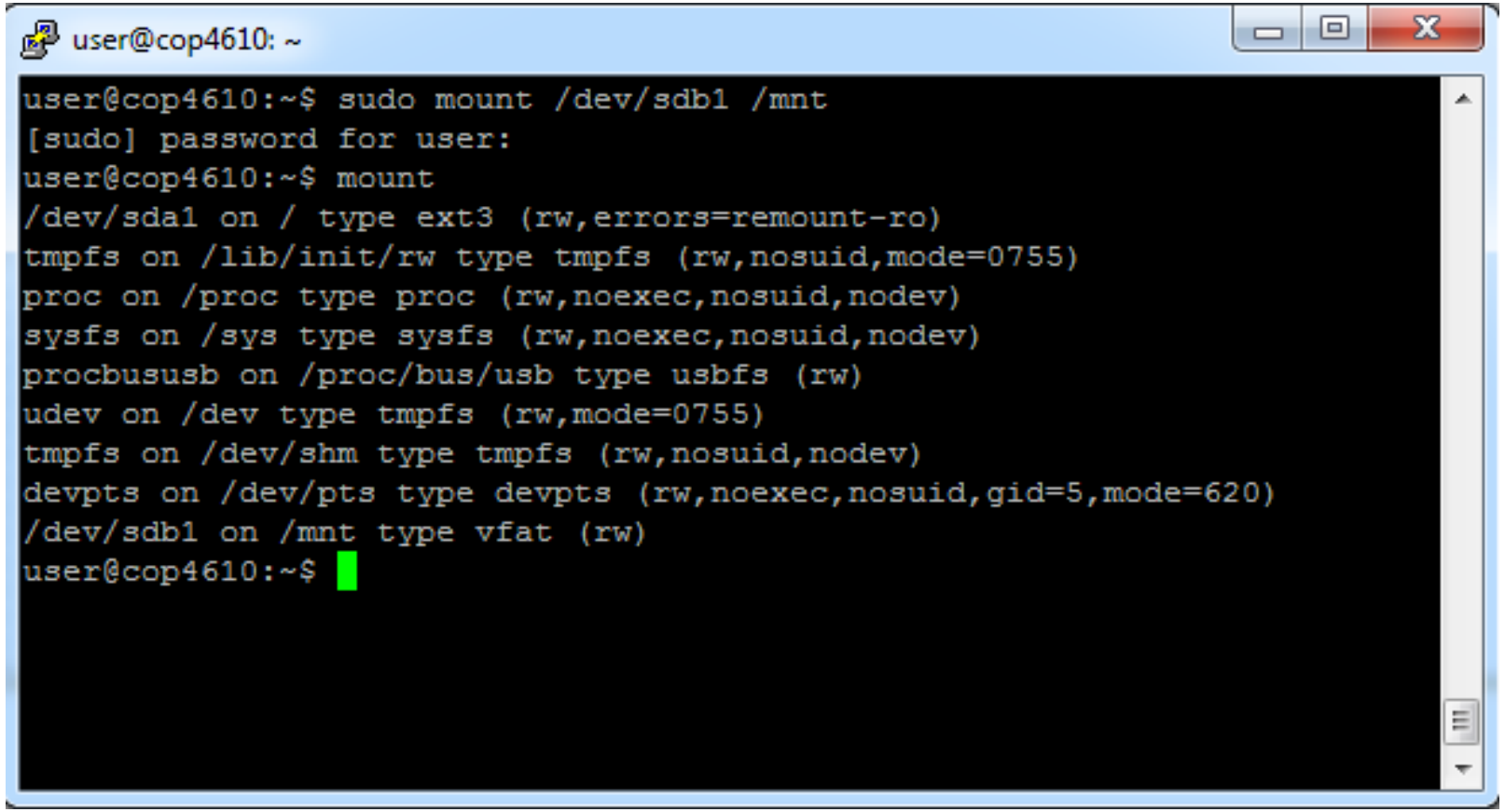
File Hierarchy Example



Files from the thumb drive are now accessible under /mnt

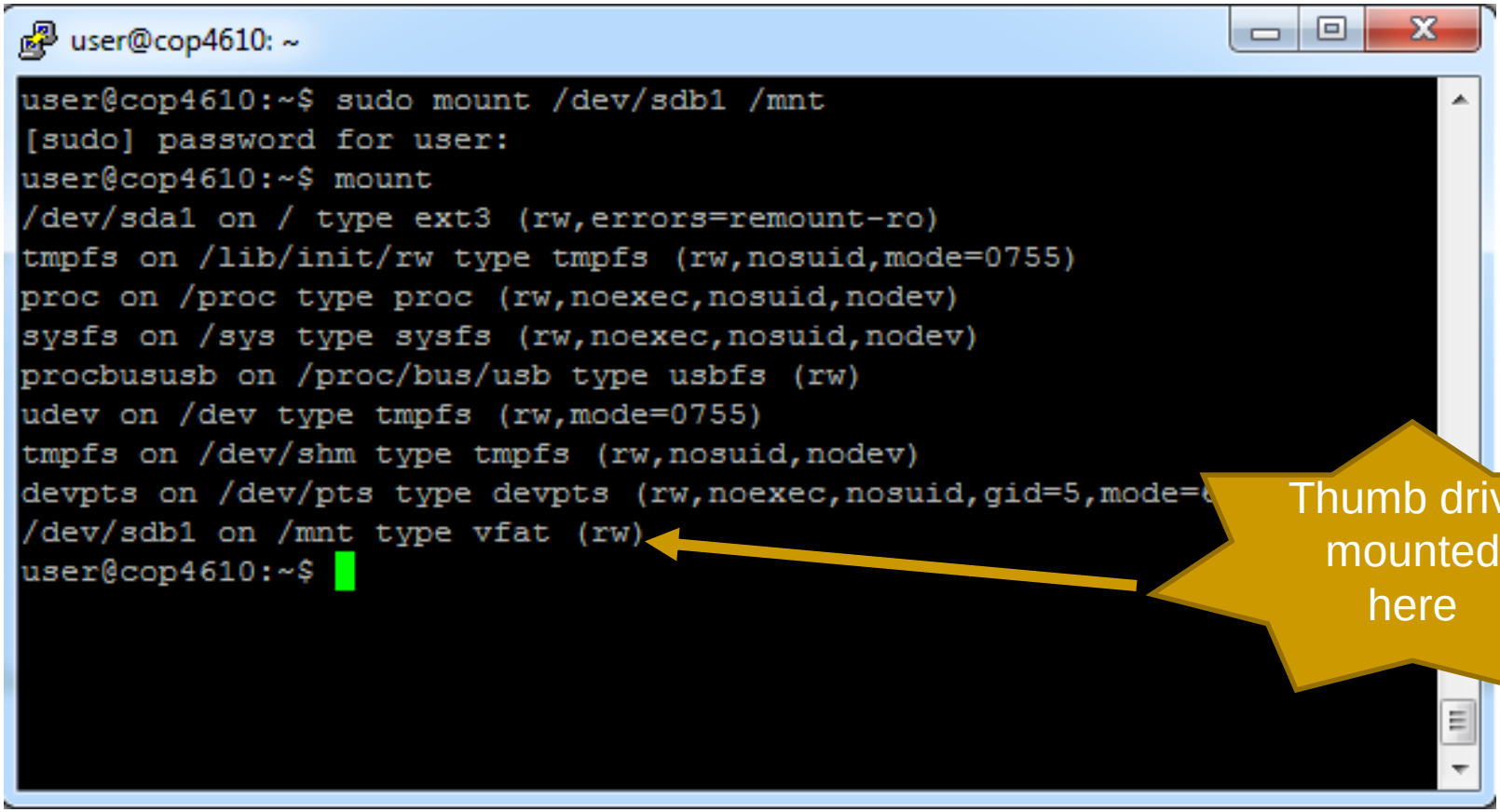
Mount Example

- The ‘mount’ utility dynamically attaches filesystems to the existing hierarchy



Mount Example

- The ‘mount’ command can dynamically attach new devices to new mount points



Unmounting

- **`umount <dir>`**

- In our example, the thumb drive was mounted at /mnt, to unmount
  - **`$> umount /mnt`**
  - May require root privileges

System Configuration

- **`/etc/fstab`**
  - List of filesystems and mount locations
    - Mounting options
      - auto-mount during boot
      - read-only
      - …
- View output of **`dmesg`** when plugging in device

Project 3

- You will create a user-space utility to manipulate a FAT32 file system image
  - No more kernel programming!
- Utility must provide a few basic operations on a given FAT32 filesystem
  - Must not corrupt the filesystem (i.e., deviate from the specifications)

FAT32 Manipulation Utility

Utility only recognizes the following built-in commands:

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

File System Image

- Manipulation utility will work on a pre-configured FAT32 ***file system image***
- Raw FAT32 data structures
  - Just like looking at the raw bytes of a disk partition

File System Image

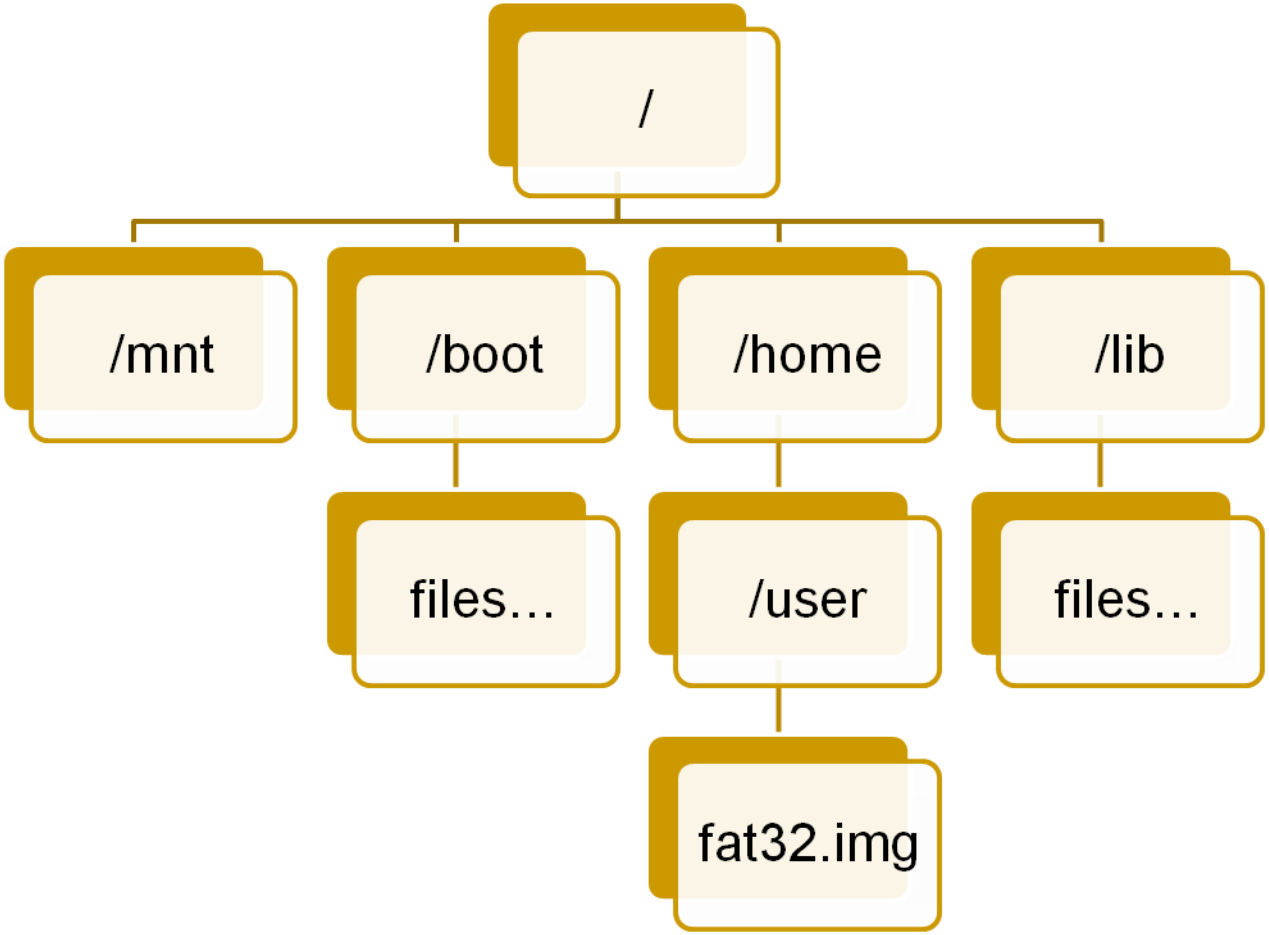
- Your FAT32 manipulation utility will have to
  - Open the FAT32 file system image
  - Read parts of the FAT32 file system image and interpret the raw bytes inside to service your utility's file system commands…

…just like a file system!

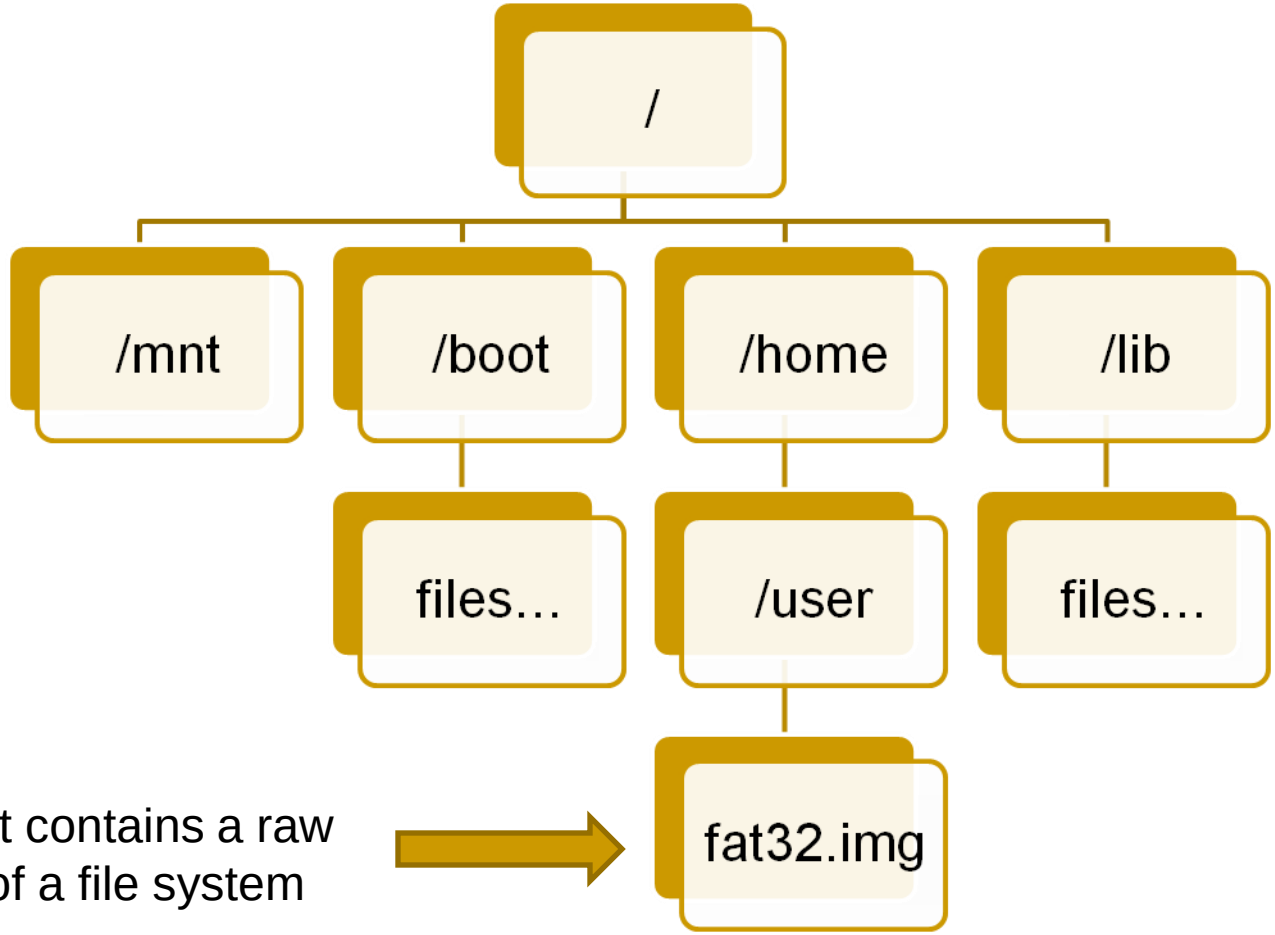
File System Image

- 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

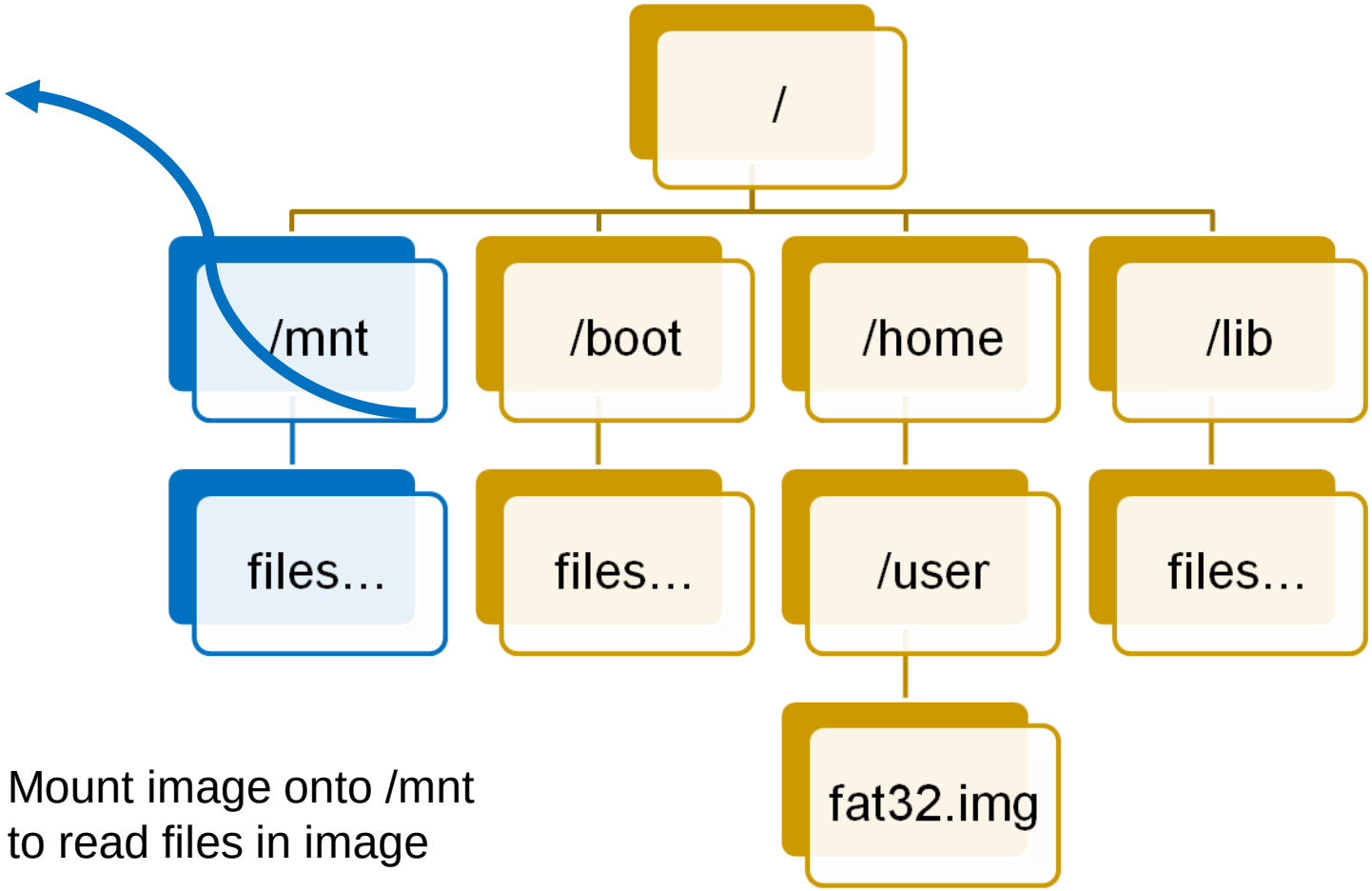


File System Image Mount Example



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 -o loop 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, …

FAT32 DATA STRUCTURES

Terminology

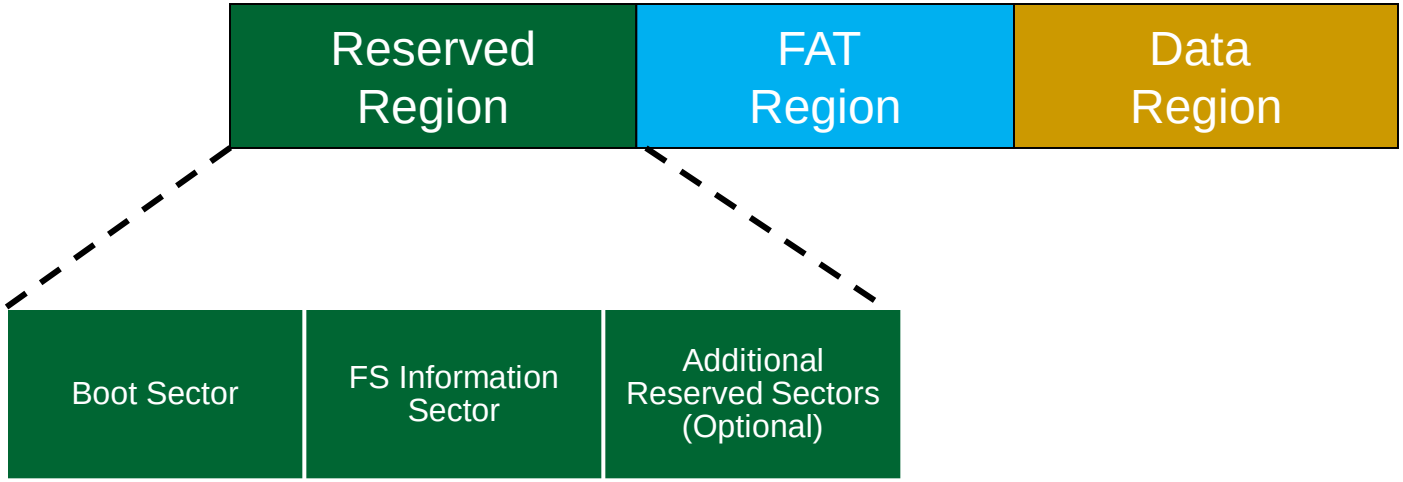
- Byte
  - 8 bits
  - Smallest addressable unit of processor
- Sector
  - Smallest addressable unit of persistent storage device
  - Typically 512 or 4096 bytes
- Cluster
  - Group of sectors representing a chunk of data
  - FAT32-specific
- FAT
  - File Allocation Table
  - Mapping of files to data

FAT32 Disk Layout

- 3 main regions…

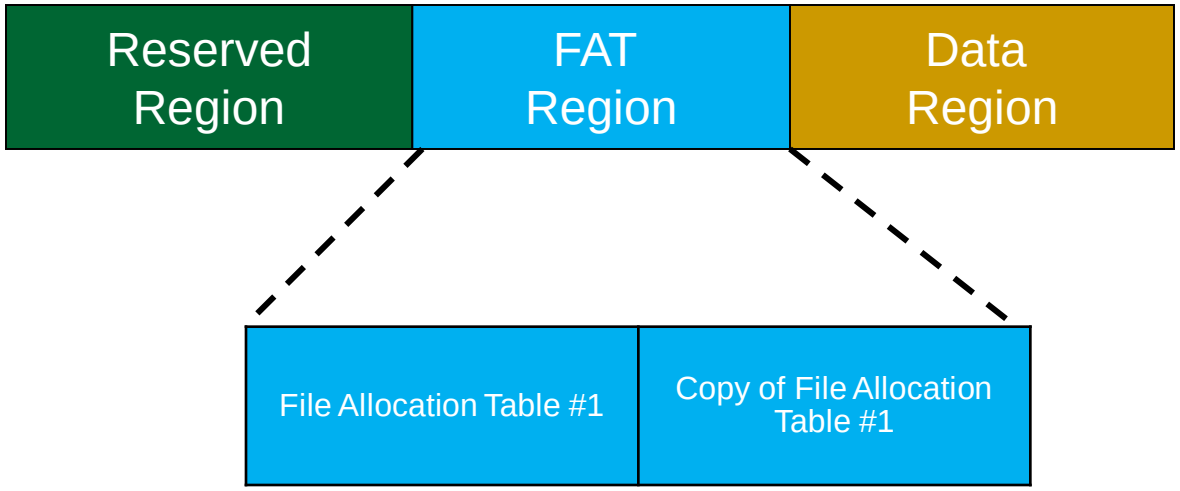
Reserved Region

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



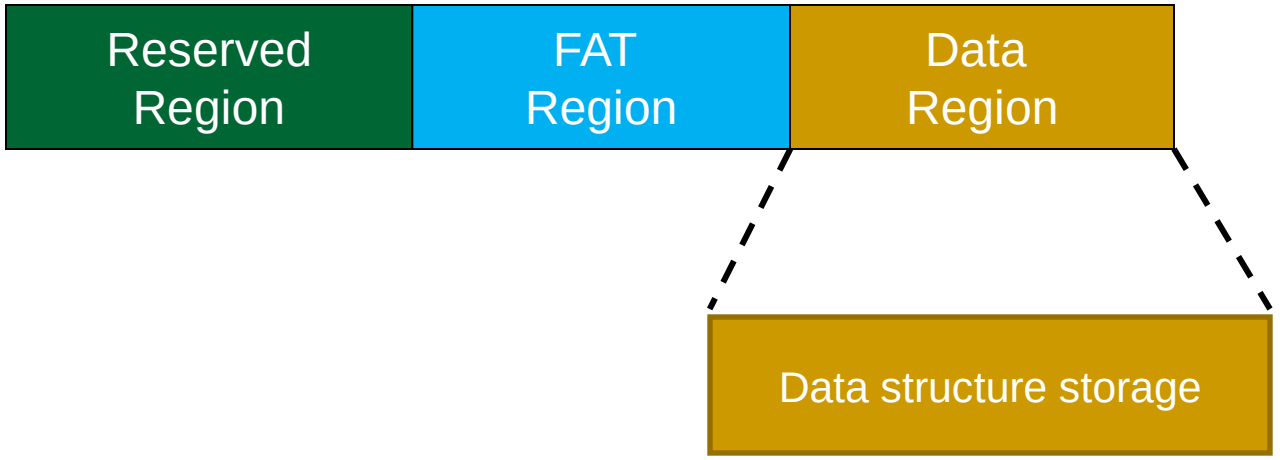
FAT Region

- **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?

Endianness

- Order to interpret a 1-dimensional array of bytes to form a larger data structure
  - E.g., ints, shorts, long longs
- Big-endian
  - Most significant byte first
  - *Big end first*
- Little-endian
  - Least significant byte first
  - *Little end first*
- FAT32 is assumes little-endian formatting

Endianness

- Order to interpret a 1-dimensional array of bytes to form a larger data structure
  - E.g., ints, shorts, long longs
- Big-endian
  - Most significant byte first
  - Big end first
- Little-endian
  - Least significant byte first
  - Little end first
- FAT32 is assumes little-endian formatting

FAT32 Endianness

- Endianness matters in your project
  - Integral values in FAT32 image
    - E.g., short
  - Directory entry attributes

Endianness Example

- 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?)

Endianness Example

- 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

- 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

Value = 13,371,337
(0x 00 CC 07 C9)

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

Until Next Time

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