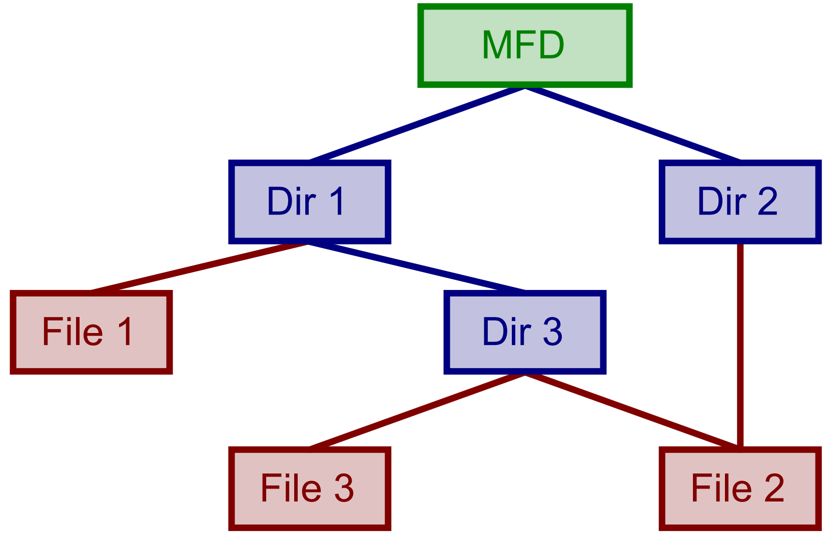
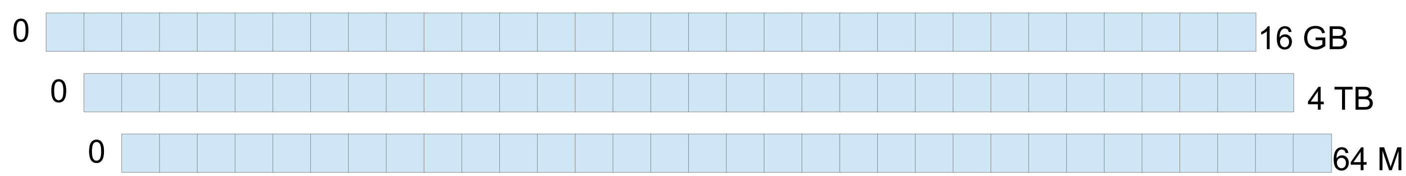
Directory Tree



Block Devices



Physical Storage

Files and Storage

- Block devices
- Filesystems
- Directories & UNIX File Conventions
- Common File Ops
- Caching

Block Devices

- Many different types of storage devices
  - Hard drives, CD-ROM, SSD, USB Thumbdrives, etc.
- Typically expose storage as an array of “blocks”
  - Fixed-size chunks of data, usually 512, 1024, or 2048 bytes
  - Can read/write blocks 0..n

Block Devices

- OSes provide a **block device** abstraction
  - `read()` / `write()` methods, and many others (`get_size()`, etc.)
  - Completely ignorant of physical details
- Device drivers talk to the OS
  - Implement the block device abstraction
  - Block devices (**not** disks) are visible to the user

Block Devices

- Other types of block devices are possible:
  - RAMdisk
  - Loopback device (file as a “disk”)
  - ???

Example from `lectura` :

```
russelll@lectura:~$ ls -al /dev/sda*
brw-rw---- 1 root disk 8,  0 Jul 27 06:32 /dev/sda
brw-rw---- 1 root disk 8,  1 Jun  1 19:56 /dev/sda1
brw-rw---- 1 root disk 8, 10 Jun  1 19:56 /dev/sda10
brw-rw---- 1 root disk 8,  2 Jun  1 19:56 /dev/sda2
brw-rw---- 1 root disk 8,  3 Jun  1 19:56 /dev/sda3
brw-rw---- 1 root disk 8,  4 Jun  1 19:56 /dev/sda4
brw-rw---- 1 root disk 8,  5 Jun  1 19:56 /dev/sda5
brw-rw---- 1 root disk 8,  6 Jun  1 19:56 /dev/sda6
brw-rw---- 1 root disk 8,  7 Jun  1 19:56 /dev/sda7
brw-rw---- 1 root disk 8,  8 Jun  1 19:56 /dev/sda8
brw-rw---- 1 root disk 8,  9 Jun  1 19:56 /dev/sda9
russelll@lectura:~$
```

T,P,S:
What do you think is the relationship between `/dev/sda` and the other devices?

Example from my home Linux box:

```
russ@russ-9020m-home:~$ ls -al /dev/sda*
brw-rw---- 1 root disk 8, 0 Jul 31 16:08 /dev/sda
brw-rw---- 1 root disk 8, 1 Jul 31 16:08 /dev/sda1
brw-rw---- 1 root disk 8, 2 Jul 31 16:08 /dev/sda2
brw-rw---- 1 root disk 8, 3 Jul 31 16:08 /dev/sda3
russ@russ-9020m-home:~$
```

I'll show you what `fdisk` shows me, on the next slide.

WARNING WARNING WARNING

It is safe to use `fdisk` to read your disk's parameters. But be careful! If you make changes, you can **destroy all the data on your disk.**

Example from my home Linux box:

```
russ@russ-9020m-home:~$ sudo fdisk /dev/sda

Welcome to fdisk (util-linux 2.38).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


Command (m for help): p


Disk /dev/sda: 3.64 TiB, 4000787030016 bytes, 7814037168 sectors
Disk model: ST4000LM024-2AN1
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
Disklabel type: gpt
Disk identifier: D01E1719-1F4F-4B1C-A9D0-E4917B088593

Device        Start        End    Sectors  Size Type
/dev/sda1      2048       4095       2048    1M BIOS boot
/dev/sda2      4096    1054719    1050624  513M EFI System
/dev/sda3   1054720 7814035455 7812980736  3.6T Linux filesystem

Command (m for help): q

russ@russ-9020m-home:~$
```

Filesystems

- A **filesystem** stores files and directories in a block device
  - Generally, can put any filesystem on any device
  - In practice, certain combinations are common

- OS provides a filesystem abstraction
  - Allows users to see all files, in all filesystems, as equivalent

Filesystems

NOTE:

- A few filesystems don't represent data on disk
  - NFS
  - FUSE
  - etc.
- These FSes expose the filesystem abstraction, but don't use a block device

Filesystems

Problem A:

- Block devices read/write in terms of sectors (fixed size) but files can be any size
  - Large files: many sectors
  - Small files: much less than a sector

Problem B:

- Need to track file metadata
  - Owner/group, permissions, modification time, etc.

Filesystems

- Filesystems keep track of metadata for each file
  - In UNIX, called an **inode**
- Includes basic file properties
- Includes indexing information

- Inodes are typically invisible to the user
  - Except that you can ask for an “inode number”
  - Different FSes store them in different places

Filesystems

- How to arrange the blocks on disk?
  - Sequential: All blocks of a file in a line, like an array
  - Linked list: Each block points to next
  - Indexed: Special block stores locations of the actual data

T,P,S:
Discuss the tradeoffs between various strategies.
What are common operations, that must be optimized for?

Filesystems

- What is used in the real world?
  - **It varies!** (Each FS can make a different choice)
  - Sequential is rare, good for RO medium (ISO 9660)
  - Linked list **very** rare

Surprised? It turns out that most OSes don't provide an "insert in the middle" op, so linked lists are not very useful.

- Common: indexed
  - Hierarchical for big files
  - But wasteful for small files
  - So, often a hybrid

Directories

- How to store a directory?
  - Some of the simplest FSes don’t
- Option A:
  - Directory is a special thing stored on disk
  - Maybe a limit on # of directories, or max depth
- Option B:
  - Directory is a file with a special “type”
  - Just stores lookup information

Directories and Hard Links

UNIX Design:

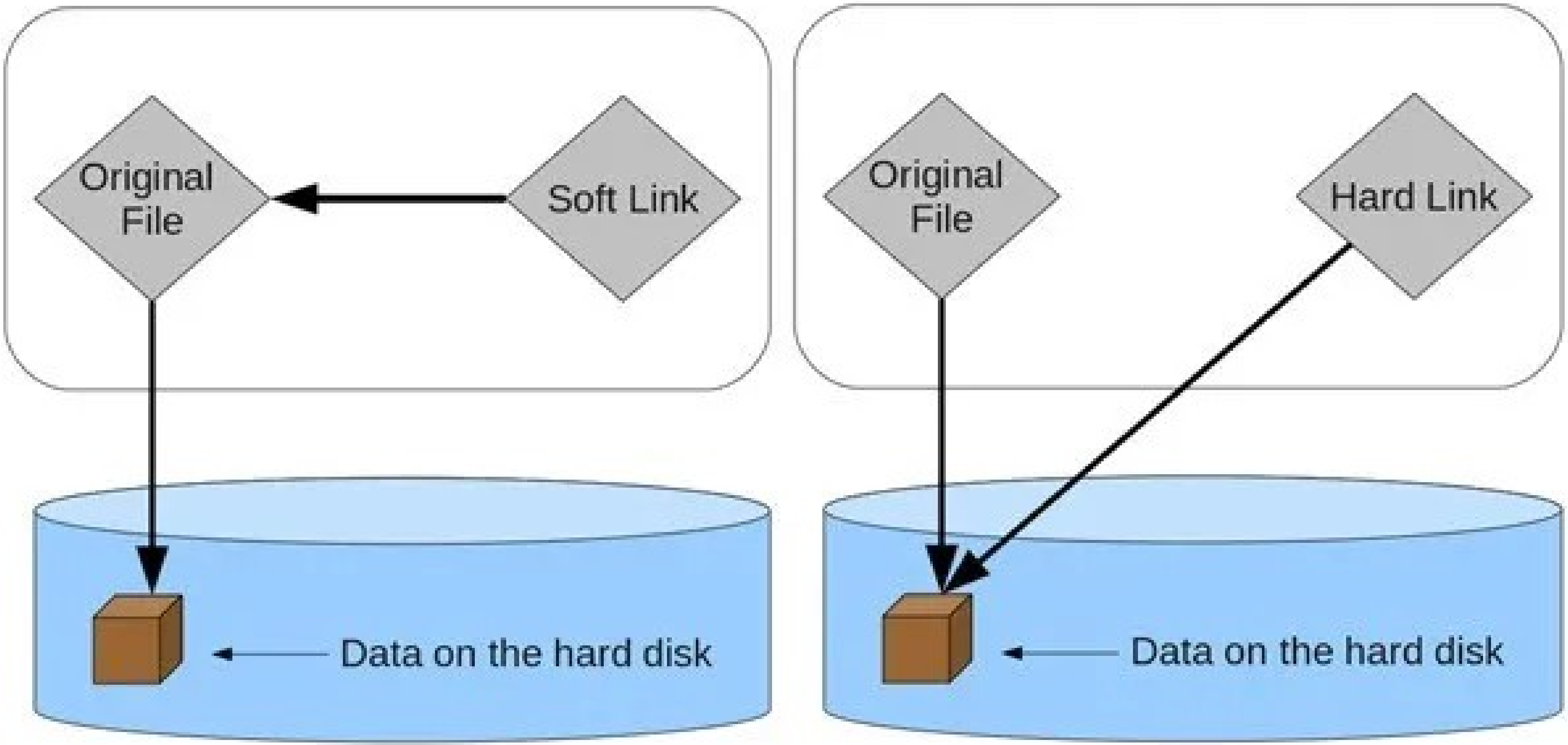
- A directory is a special type of file
  - Other types: regular, symlink, device, pipe
- Stores mappings of names to **inode numbers**
  - Not files!
  - Thus, 2 directories can “contain” the same file

- A **hard link** is when two directory entries point to the same underlying file

Symbolic Links

UNIX Design:

- A **symbolic link** (or **soft link**) is a special file which links to another file by *name* instead of by *inode number*
- Sometimes, the file doesn't exist
- Can point to a directory
- Can point to a file in a different FS



https://medium.com/@eitanmayer/hard-vs-symbolic-links-f0584a5d1db5

Investigation

UNIX Commands to Try:

```
ln -s old_file new_file    # symlink
ln    old_file new_file    # hard link

ls -al              # type L is symlink
ls -ali             # show inode #s
```

Common File Ops

- OSes typically require you to open a file before you use it

T,P,S:

What are the advantages of opening a file, as a separate operation from read/write ?

Common File Ops

- A **file handle** is an identifier (often, an integer) which represents an *open file,* by a *certain process*
  - Usually, when we `fork()`, child gets a duplicate
  - Some OSes allow you to delete the file & keep using it

- In C, we normally use a `FILE*` . But this is just an abstraction around the file handle

Common File Ops

- `open()`
- `close()`
- `read()`, `write()`
- `seek()`
- `unlink()`,`rename()`,`mkdir()`,`rmdir()`

Advanced File Ops

- `stat(),fstat()`
- `getdents()`
- `mmap()`
- `dup()`
- `pipe()`
- `socket()`

Caching

T,P,S:

What examples of **caching** have you seen in other classes?

Caching

- OSes often have a **file cache** and/or a **block cache.**

- File caches store the contents of files (and directory entries) so that opening & modifying files are faster
- Block caches store the data from block devices, so that we don't have to read them again

Caching

- Caching speeds reads (almost always)
- Caching speeds writes (sometimes)

T,P,S:

How could caching speed up writes, and what are the dangers involved?

Caching

- Caching speeds reads (almost always)
- Caching speeds writes (sometimes)

- Write caching alternatives:
  - **Write through:** synchronous writes
  - **Write on close:** only flush when you’re done
  - **Bounded write-back:** write occasionally

Caching

- Caching has to be done *carefully*
  - Network FSes: cache consistency issues
  - Local OSes: durability of writes

- **Cache consistency:** When you have multiple caches, they agree about the state of the data
- **Durability:** A write will survive a system crash or power loss