
# 377 Operating Systems

## File System Introduction


# Key OS Abstractions

Thus far we have seen the development of two key operating system abstractions:

- **Process:** virtualization of the CPU
- **Address Space:** virtualization of memory

In tandem, these two abstractions allow a program to run as if it is in its own private, isolated world.

# Key OS Abstractions

Thus far we have seen the development of two key operating system abstractions:

- **Process:** virtualization of the CPU
- **Address Space:** virtualization of memory

In tandem, these two abstractions allow a program to run as if it is in its own private, isolated world.

The third abstraction:

- **Persistent Storage:** virtualization of a persistent storage device (disk)

## CRUX: HOW TO MANAGE A PERSISTENT DEVICE

How should the OS manage a persistent device? What are the APIs? What are the important aspects of the implementation?

# Files and Directories

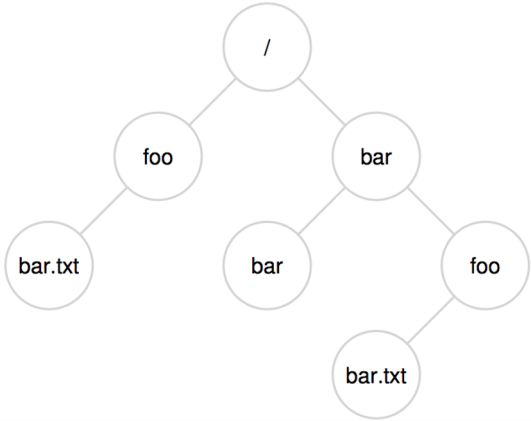
Two key abstractions have been developed over time:

- Files
- Directories

Note that this is definitely not the only way to organize persistent storage!

- Other examples include databases, persistent key-value stores, etc.
- However, files and directories are inherent to most operating systems

# Files and Directories




Two key abstractions have been developed over time:

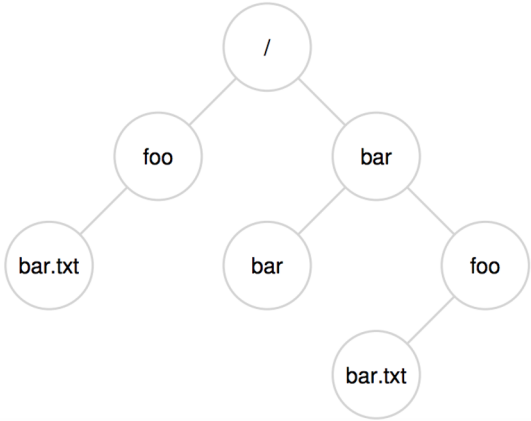
- **Files**
- Directories

A **file** has the following two properties:

- A linear array of bytes that can be read and written
- Has a low-level name which is usually a number (called an inode number)

In most systems, the OS does not know much about the structure of the file (e.g., whether it is a picture, or a text file, or C code).

# Files and Directories




Two key abstractions have been developed over time:

- Files
- **Directories**

A **directory** has the following two properties:

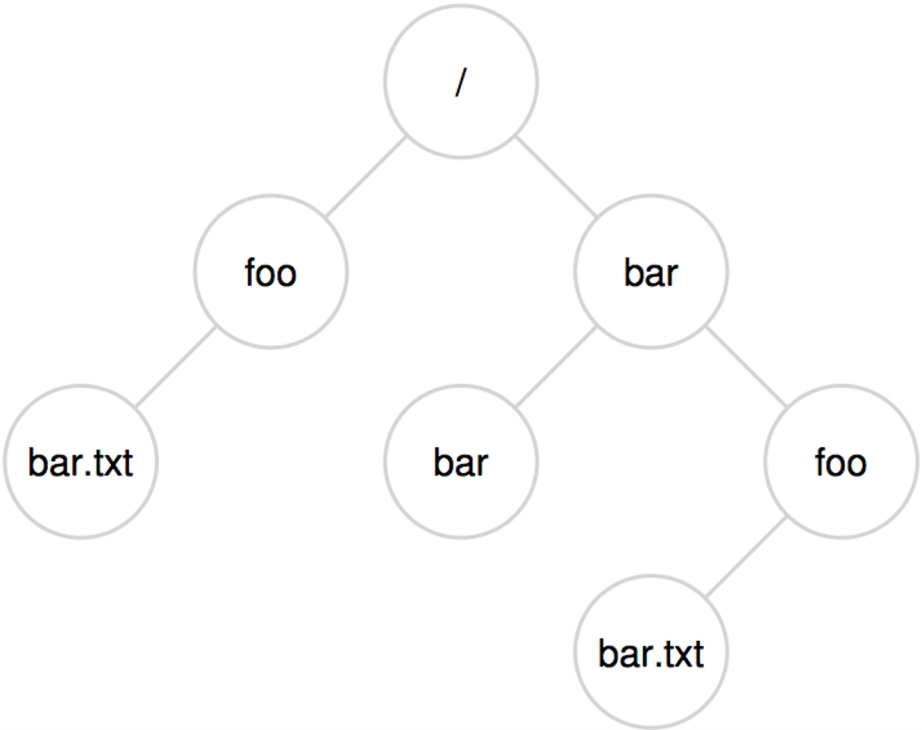
- Contains a list of (user-readable name, low-level name) pairs
- Also has a low-level name which is usually a number (called an inode number)

For example, let's say there is a file with the low-level name "10", and it is referred to by the user-readable name of "foo". The directory that "foo" resides in thus would have an entry ("foo", "10").

# Directory Tree

Each entry in a directory refers to either files or other directories.

By placing directories within other directories, users are able to build an arbitrary directory tree, under which all files and directories are stored.

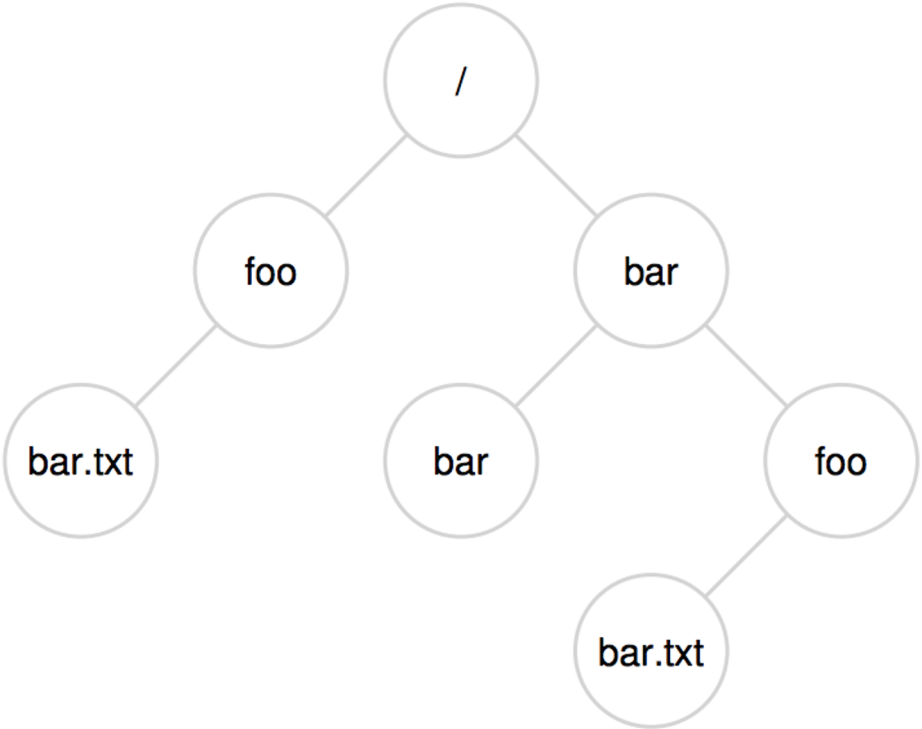


# Directory Tree

The directory hierarchy on a Unix system starts at / which is called "root".

It uses some kind of separator (e.g., /) to name subsequent sub-directories until the desired file or directory is named.

The absolute path starts from the root until we reach the desired file or directory.



## TIP: THINK CAREFULLY ABOUT NAMING

Naming is an important aspect of computer systems [SK09]. In UNIX systems, virtually everything that you can think of is named through the file system. Beyond just files, devices, pipes, and even processes [K84] can be found in what looks like a plain old file system. This uniformity of naming eases your conceptual model of the system, and makes the system simpler and more modular. Thus, whenever creating a system or interface, think carefully about what names you are using.

## TIP: THINK CAREFULLY ABOUT NAMING

Naming is an important aspect of computer systems [SK[redacted]]. In UNIX systems, virtually everything that you can think of is named through the file system. Beyond just files, devices, pipes, and even processes [K84] can be found in what looks like a plain old file system. This uniformity of naming eases your conceptual model of the system, and makes the system simpler and more modular. Thus, whenever creating a system or interface, think carefully about what names you are using.

## TIP: THINK CAREFULLY ABOUT NAMING

Naming is an important aspect of computer systems [SK09]. In UNIX systems, virtually everything that you can think of is named through the file syst  Beyond just files, devices, pipes, and even processes [K84] can be found in what looks like a plain old file system. This uniformity of naming eases your conceptual model of the system, and makes the system simpler and more modular. Thus, whenever creating a system or interface, think carefully about what names you are using.

## TIP: THINK CAREFULLY ABOUT NAMING

Naming is an important aspect of computer systems [SK09]. In UNIX systems, virtually everything that you can think of is named through the file system. Beyond just files, devices, pipes, and even processes [K84] can be found in what looks like a plain old file syst  This uniformity of naming eases your conceptual model of the system, and makes the system simpler and more modular. Thus, whenever creating a system or interface, think carefully about what names you are using.

## TIP: THINK CAREFULLY ABOUT NAMING

Naming is an important aspect of computer systems [SK09]. In UNIX systems, virtually everything that you can think of is named through the file system. Beyond just files, devices, pipes, and even processes [K84] can be found in what looks like a plain old file system. This uniformity of naming eases your conceptual model of the system, and makes the system simpler and more modular. Thus, whenever creating a system or interface, think carefully about what names you are using.

# File Extensions

You may also notice that the file name in this example often has two parts: **bar** and **txt**, separated by a period.

The first part is an arbitrary name, whereas the second part of the file name is usually used to indicate the type of the file, e.g., whether it is C code (e.g., .c), or an image (e.g., .jpg), or a music file (e.g., .mp3).

However, this is usually just a convention: there is usually no enforcement that the data contained in a file named main.c is indeed C source code.

# File System Interface: Creating Files

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

Here, we are creating a file called "foo" in the current working directory.

# File System Interface: Creating Files

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

Here, we are creating a file called "foo" in the current working directory.

The **`open()`** system call takes a number of different flags.

File creation flags:

- **O_CREAT:** create the file if it does not exist
- **O_WRONLY:** ensure that the file is write only
- **O_TRUNC:** if the file exists truncate it to a size of 0 bytes

# File System Interface: Creating Files

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

Here, we are creating a file called "foo" in the current working directory.

The **`open()`** system call takes a number of different flags.

File permission flags:

- **S_IRUSR:** set the file readable by the owner
- **S_IWUSR:** set the file writable by the owner

## ASIDE: THE CREAT() SYSTEM CALL

The older way of creating a file is to call `creat()`, as follows:

```
int fd = creat("foo"); // option: add second flag to set permissions
```

You can think of `creat()` as `open()` with the following flags: `O_CREAT | O_WRONLY | O_TRUNC`. Because `open()` can create a file, the usage of `creat()` has somewhat fallen out of favor (indeed, it could just be implemented as a library call to `open()`)

## ASIDE: THE CREAT() SYSTEM CALL

The older way of creating a file is to call `creat()`, as follows:

```
int fd =                    // option: add second flag to set permissions
```

You ca at() as open() with the following flags: O_CREAT O_TRUNC. Because open() can create a file, the usage somewhat fallen out of favor (indeed, it could just be im brary call to open()); however, it does hold a special pl Specifically, when Ken Thompson was asked what he would do differently if he were redesigning UNIX, he replied: "I'd spell creat with an e."

# File Descriptor

One important aspect of **open()** is what it returns: a file descriptor.

A file descriptor is just an integer, private per process, and is used in UNIX systems to access files

Once a file is opened, you use the file descriptor to read or write the file, assuming you have permission to do so.

In this way, a file descriptor is a capability, i.e., an **opaque handle** that gives you the power to perform certain operations.

What is another opaque handle we have seen earlier in this course?

# Reading and Writing Files

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

# Reading and Writing Files

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)          = 3
read(3, "hello\n", 4096)                   = 6
write(1, "hello\n", 6)                     = 6
hello
read(3, "", 4096)                          = 0
close(3)                                   = 0
...
prompt>
```

# Reading and Writing, But Not Sequentially

```
off_t lseek(int fildes, off_t offset, int whence);
```

- **fildes**: file descriptor
- **offset:** the offset to position to
- **whence:**

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
the file plus offset bytes.
```

# Writing Immediately

Most times when a program calls **write()**, it is just telling the file system:
please write this data to persistent storage, at some point in the future.

The file system, for performance reasons, will buffer such writes in memory for some time (say 5 seconds, or 30).

At that later point in time, the **write(s)** will actually be issued to the storage device.

From the perspective of the calling application, writes seem to complete quickly, and only in rare cases (e.g., the machine crashes after the **write()** call but before the write to disk) will data be lost.

What if we want to guarantee that everything is written to the file?

# Writing Immediately with fsync()

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

When a process calls **fsync()** for a particular file descriptor, the file system responds by forcing all dirty (i.e., not yet written) data to disk.

The **fsync()** routine returns once all of these writes are complete.

# Renaming Files

```
rename(char *old, char *new)
```

One interesting guarantee provided by the **rename()** call is that it is (usually) implemented as an atomic call with respect to system crashes.

If the system crashes during the renaming, the file will either be named the old name or the new name, and no odd in-between state can arise.

# Rename Example

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

Let’s be a little more specific here. Imagine that you are using a file editor (e.g., emacs), and you insert a line into the middle of a file.

# Rename Example

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

What the editor does in this example is simple: (1) write out the new version of the file under a temporary name (foo.txt.tmp)

# Rename Example

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

What the editor does in this example is simple: (1) write out the new version of the file under a temporary name (foo.txt.tmp), (2) force it to disk with fsync()

# Rename Example

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

What the editor does in this example is simple: (1) write out the new version of the file under a temporary name (foo.txt.tmp), (2) force it to disk with fsync(), and (3) then, when the application is certain the new file metadata and contents are on the disk, rename the temporary file to the original file's name.

# Rename Example

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

What the editor does in this example is simple: (1) write out the new version of the file under a temporary name (foo.txt.tmp), (2) force it to disk with fsync(), and (3) then, when the application is certain the new file metadata and contents are on the disk, rename the temporary file to the original file's name. **This last step atomically swaps the new file into place, while concurrently deleting the old version of the file, and thus an atomic file update is achieved.**

# Getting Information About Files

Beyond file access, we expect the file system to keep a fair amount of information about each file it is storing.

We generally call such data about files **metadata**.

To see the metadata for a certain file, we can use the **stat()** or **fstat()** system calls.

# Getting Information About Files

These calls take a pathname (or file descriptor) to a file and fill in a stat structure as seen here:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

# UNIX Stat

```
prompt> echo hello > file
prompt> stat file
  File: ‘file’
  Size: 6   Blocks: 8   IO Block: 4096   regular file
Device: 811h/2065d Inode: 67158084    Links: 1
Access: (0640/-rw-r-----) Uid: (30686/remzi)
  Gid: (30686/remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

# Removing Files

```
prompt> strace rm foo
...
unlink("foo")
...
```

Why is it called unlink?

First, we need to look at directories.

# Making Directories

How do we make a directory?

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)
...
prompt>
```

# Making Directories

An empty directory has two entries:

- one entry that refers to itself
- one entry that refers to its parent

```
prompt> ls -a
./  ../
prompt> ls -al
total 8
drwxr-x---  2 remzi remzi    6 Apr 30 16:17 ./
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ../
```

# Reading Directories

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}
```

```
struct dirent {
    char           d_name[256]; /* filename */
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* offset to the next dirent */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file */
};
```

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}
```

# Deleting Directories

You can delete a directory with a call to **rmdir()**.

Unlike file deletion, however, removing directories is more dangerous, as you could potentially delete a large amount of data with a single command.

Thus, **rmdir()** has the requirement that the directory be empty (i.e., only has “.” and “..” entries) before it is deleted. If you try to delete a non-empty directory, the call to **rmdir()** simply will fail.

Dangerous: **rm -rf /**

# Hard Links

We now come back to the mystery of why removing a file is performed via **unlink()**.

We will do this by understanding a new way to make an entry in the file system tree:

Through a system call known as **link()**

The **link()** system call takes two arguments:

- Existing pathname
- New pathname

When you “link” a new file name to an old one, you essentially create another way to refer to the same file.

# Hard Links

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello

prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

# Hard Links

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello

prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

The “inode” numbers are the same, which means **file** and **file2** refer to the same file on the file system.

# Symbolic Links

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

As you can see, creating a soft link looks much the same, and the original file can now be accessed through the file name file as well as the symbolic link name file2.

# Symbolic Links

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

beyond this surface similarity, symbolic links are actually quite different from hard links. The first difference is that a symbolic link is actually a file itself, of a different type. We've already talked about regular files and directories; symbolic links are a third type the file system knows about.

# Symbolic Links

```
prompt> stat file
  ... regular file ...
prompt> stat file2
  ... symbolic link ...
```

A **stat** on the file system reveals all...

# Symbolic Links

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May  3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ../
-rw-r-----  1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

Running **ls** also reveals this fact...
If you look closely at the first character of the long-form of the output from **ls**, you can see that the first character in the left-most column is **a** - for regular files, a **d** for directories, and an **l** for soft links.

# Symbolic Links

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May  3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ../
-rw-r-----  1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

Why is the symbolic link 4 bytes?

# Symbolic Links

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May  3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ../
-rw-r-----  1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

The reason that **file2** is 4 bytes is because the way a symbolic link is formed is by *holding the pathname* of the linked-to file as the data of the link file. Because we've linked to a file named file, our link file file2 is small (4 bytes).

# Symbolic Links

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi  6 May  3 19:17 alongerfilename
lrwxrwxrwx 1 remzi remzi 15 May  3 19:17 file3 -> alongerfilename
```

If we link to a longer pathname, our link file would be bigger

# Permission Bits and Access Control Lists

**The abstraction of a process provided two central virtualizations:**

1. CPU
2. Memory

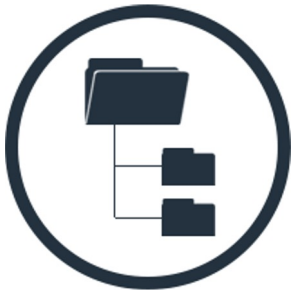
Each of these gave the illusion to a process that it had its own **private CPU** and its own **private memory**.

In reality, the OS underneath used various techniques to share limited physical resources among competing entities in a safe and secure manner.

# Permission Bits and Access Control Lists

- **The file system also presents a virtual view of a disk.**
  - Transforming it from a bunch of raw blocks into user-friendly files and directories.

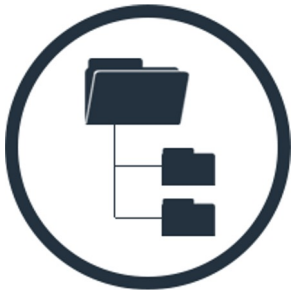
- **The abstraction is notably different from that of the CPU and memory**
  - Files are commonly shared among different users and processes are typically not.



# Permission Bits and Access Control Lists

- **The file system also presents a virtual view of a disk.**
  - Transforming it from a bunch of raw blocks into user-friendly files and directories.

- **The abstraction is notably different from that of the CPU and memory**
  - Files are commonly shared among different users and processes are typically not.



Thus, a more comprehensive set of mechanisms for enabling various degrees of sharing are usually present within file systems.

# UNIX Permission Bits

```
prompt> ls -l foo.txt
-rw-r--r--  1 remzi wheel  0 Aug 24 16:29 foo.txt
```

The first form of such mechanisms is the classic UNIX permission bits.

# UNIX Permission Bits

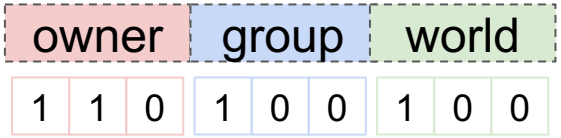
```
prompt> ls -l foo.txt
-rw-r--r--  1 remzi wheel  0 Aug 24 16:29 foo.txt
```

owner group world

The first three characters of the output of **ls** show that the file is both **readable** and **writable** by the owner (rw-), and only **readable** by members of the group *wheel* and also by anyone else in the system.

# UNIX Permission Bits

```
prompt> ls -l foo.txt
-rw-r--r--  1 remzi wheel  0 Aug 24 16:29 foo.txt
```

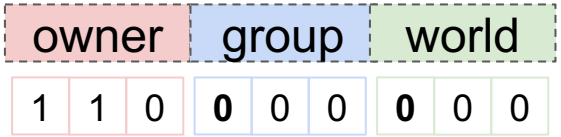



The permission bits are broken up into 9 bits.

3 bits for each permission (read/write/execute) designation.

# UNIX Permission Bits

```
prompt> chmod 600 foo.txt
```




The owner of the file can readily change these permissions, for example by using the chmod command.

Note - this is using octal for historical (and sensible) reasons.

# Mounting a File System

```
prompt> mount -t ext3 /dev/sda1 /home/users
```

If successful, the mount would thus make this new file system available. However, note how the new file system is now accessed. To look at the contents of the root directory, we would use `ls` like this:

```
prompt> ls /home/users/
a b
```


The End