Announcements

- **Start the assignment** if you haven’t already!
  - Repos have been created. Email me if you still don’t have access.
- **Lab signups** are now available on the website
- Website **calendar** has been updated with office hours and lab schedules
- Remember to join **Piazza** and **Slack**
- I’ll try to keep each lecture at 90 minutes instead of the full 110 minutes. I’ll use the extra time to answer questions. I’ll also have a 10 minute break in the middle of lecture when possible.
- Use **cplayground.com** to test out code! Created by last year’s summer instructor, Ryan
- **Clarifications**
  - The mode that is passed into the open system call starts with a 0 to denote octal (base 8) versus using the default, decimal (decimal equivalent is 420):

```c
int file_descriptor = open(kFilename, O_WRONLY | O_CREAT | O_EXCL, 0644);
```
Lecture Overview

We need to invest some time learning about some C and Unix/Linux libraries that offer us programmatic access to the filesystem. There are a few data structures and a collection of Linux library functions that allow us to crawl over the tree of files and directories, and there are even more functions that grant us raw, low-level access to the file contents.

We'll invest some time discussing these functions and data structures—enough that we can implement a collection of obvious and not-so-obvious programs that emulate some of the terminal/shell builtins you've been using your entire Unix lives.
Accessing Code Examples

- Today's lecture examples reside within:
  
  /usr/class/cs110/lecture-examples/filesystems.

  - First `ssh` into a myth machine (ssh `yourusername@myth.stanford.edu`). When prompted for your password, it is normal for the text not to appear as you enter your password. Once logged onto a myth machine, `cd` into the above directory.

  - The `/usr/class/cs110/lecture-examples` directory is a `git` repository that will be updated with additional examples as the quarter progresses.

  - To get started, type:

    `git clone /usr/class/cs110/lecture-examples cs110-lecture-examples`

    at the command prompt to create a local copy of the master.

  - Each time I mention there are new examples (or whenever you think to), descend into your local copy and type `git pull`. Doing so will update your local copy to match whatever the master has become.
You should already be familiar with the Linux filesystem as a user. The filesystem uses a tree-based model to store files and directories of files. You can get details of a file in a particular directory with the \texttt{ls} command:

```
rcyrus@myth58:~/cs110/summer-2019/lecture-examples/filesystems$ ls
copy.c list.c Makefile search.c t.c
```

You can get a more detailed listing with the \texttt{ls -al} command:

```
rcyrus@myth58:~/cs110/summer-2019/lecture-examples/filesystems$ ls -al
total 23
  drwx------  2 cgregg operator 2048 Mar 29 12:33 .
  drwx------ 10 cgregg operator 2048 Mar 29 12:33 ..
  -rw-------  1 cgregg operator 1882 Mar 29 12:33 copy.c
  -rw-------  1 cgregg operator 5795 Mar 29 12:33 list.c
  -rw-------  1 cgregg operator 628 Mar 29 12:33 Makefile
  -rw-------  1 cgregg operator 2302 Mar 29 12:33 search.c
  -rw-------  1 cgregg operator 1321 Mar 29 12:33 t.c
```

- The first column is the \textit{file mode}, comprised of one \textit{entry type bit} followed by 9 \textit{permission bits}.
- There are different entry types, but only focus on these two: \texttt{d} for a directory and \texttt{-} for a regular file.
- Here, there are two files listed as directories (d), "." (the current directory) and ".." (the parent directory).
Permissions

The "rwx" designates the permissions for a file or directory, with "r" for read permission, "w" for write permission, and "x" for execute permission (for runnable files, including directories). Imagine the following entry:

```
$ ls -l list
-rwxr-xr-x 1 cgregg operator 19824 Mar 29 12:47 list
```

There are actually three parts to the permissions line, each with the three permission types available:

```
Permissions:
-rwx r-x r-x
```

In this case, the file owner (a.k.a. user) has read, write, and execute permissions, the group has only read and execute permissions (the group is one the owner belongs to), and other also has only read and execute permissions. Other represents any user that isn’t the file owner and isn’t in the group that owns the file.

Because each individual set of permissions can be either r, w, or x, there are three bits of information per permission field. We can therefore use base 8 to designate a particular permission set. Let's see how this would work for the above example:

- permissions: **rwx r-x r-x**
- bits (base 2): **111 101 101**
- base 8: **7 5 5**
- So the permissions for the file would be **755**

Roslyn Michelle Cyrus | Stanford University
The open System Call

In C, a file can be created using the `open` system call, and you can also set the permissions at that time. We will discuss the idea of system calls soon, but for now, simply think of them as a function that can do systems-related things. The open command has the following signatures (and this works in C, even though C does not support function overloading! How, you ask? See [here](#):)

```c
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

There are many flags that determine how a file can be handled after calling `open` (see `man 2 open` for a list of them), and they can be bitwise OR'd together. You must include one of the following flags in the `open` system call:

- **O_RDONLY** read only
- **O_WRONLY** write only
- **O_RDWR** read and write

We will generally only care about a few other flags when creating a file:

- **O_CREAT** If the file does not exist, it will be created.
- **O_EXCL** Ensure that this call creates the file, and fail if the file exists already
- **O_TRUNC**: If file exists, is a regular file, and is **O_RDWR** or **O_WRONLY**, then its length is truncated to 0
Changing Permissions with umask

- When creating a file, the third argument, `mode`, is used to attempt to set the permissions.
- The reason it "attempts" is because there is a default permissions mask, called `umask`, that limits the permissions of files created by each user. `umask` has a similar octal value to the permissions, although if a bit is set in the `umask`, then trying to set that bit with the `mode` parameter will not be allowed.
- The `umask` can be set with the following system call:

  ```c
  mode_t umask(mode_t mask); // see "man 2 umask" for details
  ```

- The return value is the old mask (the one that was already set).
- Note: The `umask` is set for a user in the shell, and when a program is run it inherits the user's `umask` settings. The user can adjust the default `umask` with the `umask` command in her terminal. The `umask` can also be changed in a program (to be shown soon).
Checking Permission Values in `umask`

If you want to simply check the `umask` value, you must call the function twice. Example:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

int main() {
    mode_t old_mask = umask(0); // set to 0, but get old mask as return value
    umask(old_mask); // restore to original
    printf("umask is set to %03o\n",old_mask);
    return 0;
}
```

Then run it:

```
$ gcc show_umask.c -o show_umask
$ ./show_umask
umask is set to 077
```

This output means that only the **owner** permissions can be set (rwx). The **group** and **other** permissions cannot be set because all three bits of their respective permissions are set in `umask`. 

Base Permissions

- The execute permission bit provides the right to execute a file. Only programs (and directories) should have the execute bit set; directories have the execute bit set so users can enter a directory (via `cd`).
- Regular files are created without the `x` (execute) bit set for security reasons.
- All files (including directories, which Unix treats as files) have **base permissions**: default permissions that are applied to files: **666** for regular files and **777** for directories.
- On the myth machines, the umask is set to 077 as previously discussed. Thus when new files are created, they are given `[base permission] & ~umask` permissions (recall that `~` is the bitwise NOT operator).
  - As an example, when a new regular file is created, it will have the following permissions:
    - `666 & ~077` (in octal)
    - `110110110 & 111000000` (in binary)
    - `= 110000000`, meaning that the file owner can read and write the file

```bash
$ touch new_file
$ mkdir new_dir
$ ls -l | grep new_
  drwx------ 2 rcyrus operator 2048 Jun 25 12:14 new_dir
  -rw------- 1 rcyrus operator    0 Jun 25 12:14 new_file
```

- Note how only the user permissions were set when creating a new regular file and directory.
- **Caveat**: AFS ignores permissions!
Overriding umask

- You can override umask if you need to set the permissions a particular way, either in terminal or a program.
- The following program creates a file and sets its permissions:

```c
#include <fcntl.h>    // for open
#include <unistd.h>   // for read, write, close
#include <stdio.h>
#include <sys/types.h> // for umask
#include <sys/stat.h>  // for umask
#include <errno.h>

const char *kFilename = "my_file";
const int kFileExistsErr = 17;

int main() {
    umask(0); // set to 0 to enable all permissions to be set
    int file_descriptor = open(kFilename, O_WRONLY | O_CREAT | O_EXCL, 0644);
    if (file_descriptor == -1) {
        printf("There was a problem creating '%s'!
", kFilename);
        if (errno == kFileExistsErr) {
            printf("The file already exists.
");
        } else {
            printf("Unknown errno: %d
", errno);
        }
        return -1;
    }
    close(file_descriptor);
    return 0;
}
```

After compiling:

```
$ ./open_ex
$ ls -gG my_file
-rw-r--r-- 1 0 Jun 25 14:44 my_file
```

Note that the file’s permissions were successfully set to 644.

If I try to run the program again:

```
$ ./open_ex
There was a problem creating 'my_file'!
The file already exists.
```

This is because the O_EXCL flag was provided.

File descriptors will be explained later in the course.
Unix Filesystem APIs

- We have already discussed two file system API calls: `open` and `umask`. We are going to look at other low-level operations that allow programmers to interact with the filesystem. We will focus on the direct system calls, but when writing production code (i.e., for a job), you will often use indirect methods, such as `FILE *`, `ifstream`, and `ofstream`.

- Requests to open a file, read from a file, extend the heap, etc., all eventually go through system calls, which are the only functions that can be trusted to interact with the system on your behalf. The operating system kernel actually runs the code for a system call, completely isolating the system-level interaction from your (potentially harmful) program.

Implementing copy to emulate cp

- We will show how to implement `copy` (designed to mimic the behavior of `cp`), which illustrates how to use `open`, `read`, `write`, and `close`. It also introduces the notion of a file descriptor. It is simplified and only copies files, not directories.
  - man pages exist for all of these functions (e.g. `man 2 open`, `man 2 read`, etc.)
  - The full implementation of our own `copy`, with exhaustive error checking, is right here.
  - The simplified implementation, sans error checking, is on the next slide.
Implementing copy to Emulate cp

```c
int main(int argc, char *argv[]) {
    int fdin = open(argv[1], O_RDONLY);
    int fdout = open(argv[2], O_WRONLY | O_CREAT | O_EXCL, 0644);
    char buf[1024];
    while (true) {
        ssize_t bytesRead = read(fdin, buf, sizeof(buf));
        if (bytesRead == 0) break;
        ssize_t bytesWritten = 0;
        while (bytesWritten < bytesRead) {
            bytesWritten += write(fdout, buf+bytesWritten, bytesRead-bytesWritten);
        }
    }
    close(fdin);
    close(fdout);
    return 0;
}
```

- To run this program:
  ```
  ./copy <source-file> <destination-file>
  ```
- `fd` is a file descriptor (as seen in the return value of `open`) and is an integer.
- From man 2 read:
  `read()` attempts to read up to `count` bytes from file descriptor `fd` into the buffer starting at `buf`.
- From man 2 write:
  `write()` writes up to `count` bytes from the buffer `buf` to the file referred to by the file descriptor `fd`.
- For buffer: “Read in, write out”
- Read the man pages for more info.

**read** and **write** are defined as follows. To use:
```c
#include <unistd.h>
```

```c
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```
Implementing copy to Emulate cp (continued)

Both `read` and `write` return a `ssize_t`, which is like `size_t` but can have negative values.

- Normal return values are the number of bytes successfully read or written. Thus the return value is not always the same as `count` (the argument in the function prototypes).
- A return value of `-1` indicates an error, and `errno` is updated.

The `read` system call will block until bytes have been read or when there are no more bytes to read (e.g., the file has reached the end or has been closed).

```c
#include <unistd.h>

int main(int argc, char *argv[]) {
    int fdin = open(argv[1], O_RDONLY);
    int fdout = open(argv[2], O_WRONLY | O_CREAT | O_EXCL, 0644);
    char buf[1024];
    while (true) {
        ssize_t bytesRead = read(fdin, buf, sizeof(buf));
        if (bytesRead == 0) break;
        ssize_t bytesWritten = 0;
        while (bytesWritten < bytesRead) {
            bytesWritten += write(fdout, buf+bytesWritten, bytesRead-bytesWritten);
        }
    }
    close(fdin);
    close(fdout);
    return 0;
}
```

`read` and `write` are defined as follows. To use: `#include <unistd.h>`

```c
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```
Implementing copy to Emulate cp (continued)

- If `read` or `write` returns a value less than `count`, it means that the system couldn't read or write all the bytes at once. This is why the `while` loops are necessary, and the reason for keeping track of `bytesRead` and `bytesWritten`.

- You should close files when you are done using them, although they will get closed by the OS when your program ends. We will use `valgrind` to check if your files are being closed.

```c
int main(int argc, char *argv[]) {
    int fdin = open(argv[1], O_RDONLY);
    int fdout = open(argv[2], O_WRONLY | O_CREAT | O_EXCL, 0644);
    char buf[1024];
    while (true) {
        ssize_t bytesRead = read(fdin, buf, sizeof(buf));
        if (bytesRead == 0) break;
        ssize_t bytesWritten = 0;
        while (bytesWritten < bytesRead) {
            bytesWritten += write(fdout, buf+bytesWritten, bytesRead-bytesWritten);
        }
    }
    close(fdin);
    close(fdout);
    return 0;
}
```

`read` and `write` are defined as follows. To use: `#include <unistd.h>`

```c
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```
The file descriptor abstraction provides direct, low level access to a stream of data without the fuss of data structures or objects. It certainly can't be slower, and depending on what you're doing, it may even be faster.

**FILE pointers and C++ iostreams** work well when you know you're interacting with standard output, standard input, and local files.

- They are less useful when the stream of bytes is associated with a network connection.
- **FILE pointers and C++ iostreams** assume they can rewind and move the file pointer back and forth freely, but that's not the case with file descriptors associated with network connections.

File descriptors, however, work with **read** and **write** and little else used in this course.

**C FILE pointers and C++ streams**, on the other hand, provide automatic buffering and more elaborate formatting options.
Implementing t to Emulate tee

The `tee` program that ships with Linux copies everything from standard input to standard output, making zero or more extra copies in the named files supplied as user program arguments. For example, if the file contains 27 bytes—the 26 letters of the English alphabet followed by a newline character—then the following would print the alphabet to standard output and to three files named `one.txt`, `two.txt`, and `three.txt`.

```bash
$ cat alphabet.txt | ./tee one.txt two.txt three.txt
abcdefghijklmnopqrstuvwxyz
$ cat one.txt
abcdefghijklmnopqrstuvwxyz
$ cat two.txt
abcdefghijklmnopqrstuvwxyz
$ diff one.txt two.txt
$ diff one.txt three.txt
$
```

- The full implementation of our own t executable, with error checking, is [right here](#).
- The implementation replicates much of what `copy.c` does and illustrates how you can use low-level I/O to manage many sessions with multiple files.
- The implementation inlined in the next slide omits error checking.
Implementing t to Emulate tee (continued)

```c
static void writeall(int fd, const char buffer[], size_t len) {
    size_t numWritten = 0;
    while (numWritten < len) {
        numWritten += write(fd, buffer + numWritten, len - numWritten);
    }
}

int main(int argc, char *argv[]) {
    int fds[argc];  // array of integers that represent file descriptors
    fds[0] = STDOUT_FILENO;
    for (size_t i = 1; i < argc; i++)
        fds[i] = open(argv[i], O_WRONLY | O_CREAT | O_TRUNC, 0644);

    char buffer[2048];
    while (true) {
        ssize_t numRead = read(STDIN_FILENO, buffer, sizeof(buffer));
        if (numRead == 0) break;
        for (size_t i = 0; i < argc; i++) writeall(fds[i], buffer, numRead);
    }

    for (size_t i = 1; i < argc; i++) close(fds[i]);
    return 0;
}
```

- **O_TRUNC**: If the file exists and is a regular file, and the file is `O_RDWR` or `O_WRONLY`, its length is truncated to 0.
- Note that `argc` incidentally provides a count of the number of descriptors to write to. That's why we declare `fds`, an array of length `argc`.
- `STDOUT_FILENO` is a built-in constant for the number 0, which is the descriptor normally attached to standard input. `STDOUT_FILENO` is a constant for the number 1, which is the default descriptor bound to standard output.
- For brevity, I assume all system calls succeed. The official copies of the working programs on the `myth` machines include real error checking.
Using stat and lstat

- **stat** and **lstat** are functions—system calls, actually—that populate a **struct stat** with information about some named file (e.g. a regular file, a directory, a symbolic link, etc).
  - The prototypes of the two are presented below:

  ```c
  int stat(const char *pathname, struct stat *st);
  int lstat(const char *pathname, struct stat *st);
  ```

- **stat** and **lstat** operate exactly the same way, except when the named file is a link, **stat** returns information about the file the link references, and **lstat** returns information about the link itself.
  - **man** pages exist for both of these functions (**man 2 stat** and **man 2 lstat**)
Using stat and lstat (continued)

- The **struct stat** contains the following fields (source):

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev_t</td>
<td>ID of device containing file</td>
</tr>
<tr>
<td>ino_t</td>
<td>file serial number</td>
</tr>
<tr>
<td>mode_t</td>
<td>mode of file</td>
</tr>
<tr>
<td>nlink_t</td>
<td>number of links to the file</td>
</tr>
<tr>
<td>uid_t</td>
<td>user ID of file</td>
</tr>
<tr>
<td>gid_t</td>
<td>group ID of file</td>
</tr>
<tr>
<td>dev_t</td>
<td>device ID (if file is character or block special)</td>
</tr>
<tr>
<td>off_t</td>
<td>file size in bytes (if file is a regular file)</td>
</tr>
<tr>
<td>time_t</td>
<td>time of last access</td>
</tr>
<tr>
<td>time_t</td>
<td>time of last data modification</td>
</tr>
<tr>
<td>time_t</td>
<td>time of last status change</td>
</tr>
<tr>
<td>blksize_t</td>
<td>filesystem-specific preferred I/O block size for this object.</td>
</tr>
<tr>
<td>blkcnt_t</td>
<td>number of blocks allocated for this object</td>
</tr>
</tbody>
</table>

- The **st_mode** field—which is the only one we'll really pay much attention to—isn't so much a single value as it is a collection of bits encoding multiple pieces of information about file type and permissions.
- A collection of bit masks and macros can be used to extract information from the **st_mode** field.
- We’ll see how the **stat** and **lstat** functions can be used to navigate and otherwise manipulate a tree of files within the file system.
Using `stat` and `lstat` (continued)

- **search** is our own imitation of the **find** program that comes with Linux.
  - Compare the outputs of the following to be clear how **search** is supposed to work.
  - In each of the two test runs below, an executable—one builtin, and one we'll implement together—is invoked to find all files named **stdio.h** in **/usr/include** or within any descendant subdirectories.

```sh
myth60$ find /usr/include -name stdio.h -print
/usr/include/stdio.h
/usr/include/x86_64-linux-gnu/bits/stdio.h
/usr/include/c++/5/tr1/stdio.h
/usr/include/bsd/stdio.h

myth60$ ./search /usr/include stdio.h
/usr/include/stdio.h
/usr/include/x86_64-linux-gnu/bits/stdio.h
/usr/include/c++/5/tr1/stdio.h
/usr/include/bsd/stdio.h
```

---

Roslyn Michelle Cyrus | Stanford University
Using *stat* and *lstat* (continued)

- The following `main` relies on *listMatches*, which we'll implement a little later.
  - The full program of interest, with error checking we don't present here, is online [here](#).

```c
int main(int argc, char *argv[]) {
    assert(argc == 3);
    const char *directory = argv[1];
    struct stat st;
    lstat(directory, &st);
    assert(S_ISDIR(st.st_mode));
    size_t length = strlen(directory);
    if (length > kMaxPath) return 0; // assume kMaxPath is some #define
    const char *pattern = argv[2];
    char path[kMaxPath + 1];
    strcpy(path, directory);
    listMatches(path, length, pattern);
    return 0;
}
```

- This example calls *lstat*, which extracts information about the named file and populates the `st` with that information.

- Note the use of the `S_ISDIR` macro, which examines the upper four bits of the `st_mode` field to determine whether the named file is a directory.

- `S_ISDIR` has a few cousins: `S_ISREG` decides whether a file is a regular file, and `S_ISLNK` decides whether the file is a link. We'll use all of these in our next example.

- The *listMatches* function does a depth-first traversal of the filesystem to see what files just happen to match the name of interest. It uses the three library functions shown to the left at the bottom.
Using `stat` and `lstat` (continued)

Here's the implementation of `listMatches`:

```c
static void listMatches(char path[], size_t length, const char *name) {
    DIR *dir = opendir(path);
    if (dir == NULL) return; // it's a directory, but permission to open was denied
    strcpy(path + length++, "/");
    while (true) {
        struct dirent *de = readdir(dir);
        if (de == NULL) break; // we've iterated over every directory entry, so stop looping
        if (strcmp(de->d_name, ".") == 0 || strcmp(de->d_name, "..") == 0) continue;
        if (length + strlen(de->d_name) > kMaxPath) continue;
        strcpy(path + length, de->d_name);
        struct stat st;
        lstat(path, &st);
        if (S_ISREG(st.st_mode)) {
            if (strcmp(de->d_name, name) == 0) printf("%s\n", path);
        } else if (S_ISDIR(st.st_mode)) {
            listMatches(path, length + strlen(de->d_name), name);
        }
    }
    closedir(dir);
}
```
Using stat and lstat (continued)

Implementation details of interest:

- Our implementation relies on `opendir`, which accepts what is presumably a directory. It returns a pointer to an opaque iterable that surfaces a series of `struct dirent`s via a sequence of `readdir` calls.
  - If `opendir` accepts anything other than an accessible directory, it'll return `NULL`.
  - When the `DIR` has surfaced all of its entries, `readdir` returns `NULL`.
- The `struct dirent` is only guaranteed to contain a `d_name` field, which is the directory entry's name, captured as a C string. `.` and `..` are among the sequence of named entries, but we ignore them to avoid cycles and infinite recursion.
- We use `lstat` instead of `stat` so we know whether an entry is really a link. We ignore links, again because we want to avoid infinite recursion and cycles.
- If the `stat` record identifies an entry as a regular file, we print the entire path if and only if the entry name matches the name of interest.
- If the `stat` record identifies an entry as a directory, we recursively descend into it to see if any of its named entries match the name of interest.
- `opendir` returns access to a record that eventually must be released via a call to `closedir`. That's why our implementation ends with it.
Using stat and lstat

- Here is the implementation of \texttt{list}, which emulates the functionality of \texttt{ls} (in particular, \texttt{ls -1Ua}). Implementations of \texttt{list} and \texttt{search} have much in common, but the implementation of \texttt{list} is longer.
  - Here is sample output from Jerry Cain's \texttt{list}:

\begin{verbatim}
myth60$ ./list /usr/class/cs110/WWW
drwxr-xr-x  8 70296 root       2048 Jun 23 20:17 .
drwxr-xr-x >9 root     root       2048 Jun 25 21:53 ..
drwx------  2 rcyrus operator   2048 Jun 23 18:14 templates
drwx------ >9 rcyrus operator   2048 Jun 18 00:35 static
drwx------ >9 rcyrus operator   2048 Jun 17 23:27 examples
-rw-------  1 cgregg operator   4559 Jun 24 17:24 index.html
drwxr-xr-x  2 root     root       2048 Jun 17 22:15 restricted
// others omitted for brevity
myth60$
\end{verbatim}

- Full implementation of \texttt{list.c} is \texttt{right here}.
  - We will just show one key function on the slides: the one that knows how to print out the permissions information (e.g. \texttt{drwxr-xr-x}) for an arbitrary entry.
Using `stat` and `lstat` (continued)

Here's `list`'s `listPermissions` function (it prints out the permission string consistent with the given `stat` info:

```c
static inline void updatePermissionsBit(bool flag, char permissions[], size_t column, char ch) {
    if (flag) permissions[column] = ch;
}
static const size_t kNumPermissionColumns = 10;
static const char kPermissionChars[] = {'r', 'w', 'x'};
static const size_t kNumPermissionChars = sizeof(kPermissionChars);
static const size_t kNumPermissionFlags = sizeof(kPermissionFlags)/sizeof(kPermissionFlags[0]); // perm. flag = 4 bytes
static void listPermissions(mode_t mode) {
    char permissions[kNumPermissionColumns + 1];
    memset(permissions, '-', sizeof(permissions));
    permissions[kNumPermissionColumns] = '\0';
    updatePermissionsBit(S_ISDIR(mode), permissions, 0, 'd');
    updatePermissionsBit(S_ISLNK(mode), permissions, 0, 'l');
    for (size_t i = 0; i < kNumPermissionFlags; i++) {
        updatePermissionsBit(mode & kPermissionFlags[i], permissions, i + 1, kPermissionChars[i % kNumPermissionChars]);
    }
    printf("%s
", permissions);
}
```
End of Lecture 2