Lecture 02: Unix Filesystem APIs

- **Software layered over hardware, filesystem API calls**
  - First off, we'll take a first pass at understanding how the physical hardware of a disk drive can be made to look like software to store traditional files. I'll leave some details out, but will provide enough to be clear how regular files of wildly different sizes can be stored on disk and retrieved via file sessions managed using data types like `FILE *`, `ifstream`, and `ofstream`.
  - We'll learn how programmers can interact (either directly, or indirectly though the `FILE *` and `[io]stream` implementations) with the filesystem via *system calls*, which are a collection of kernel-resident functions that user programs must go through in order to access and manipulate system resources. Requests to open a file, read from a file, extend the heap, etc, all eventually go through these system calls, which are the only functions that can be trusted to interact with the OS on your behalf.

Lecture 02: Unix Filesystem APIs

- Today's lecture examples reside within `/usr/class/cs110/lecture-examples/filesystems`.
  - 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 .` at the command prompt to create a local copy of the master.
  - Each time I mention there are new examples, descend into your local copy and type `git pull`. Doing so will update your local copy to match whatever the master has become.

Lecture 02: Implementing `copy` to emulate `cp`

- Implementation of `copy`
  - The implementation of `copy` (designed to mimic the behavior of `cp`) illustrates how to use `open`, `read`, `write`, and `close`. It also introduces the notion of a file descriptor.
  - `man` pages exist for all of these functions (e.g. `man 2 open`, `man 2 read`, etc.)
  - Full implementation of our own `copy`, with exhaustive error checking, is right here.
  - Simplified implementation, sans error checking, is on the next slide.

Lecture 02: Implementing `copy` to emulate `cp`

```
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 buffer[1024];
  while (true) {
    ssize_t bytesRead = read(fdin, buffer, sizeof(buffer));
    if (bytesRead == 0) break;
    size_t bytesWritten = 0;
    while (bytesWritten < bytesRead) {
      bytesWritten += write(fdout, buffer + bytesWritten, bytesRead - bytesWritten);
    }
  }
  close(fdin);
  close(fdout)
  return 0;
}
```

Lecture 02: Implementing `copy` to emulate `cp`

- Pros and cons of file descriptors over `FILE` pointers and C++ `iostream`s
  - 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++ `iostream`s 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++ `iostream`s assume they can rewind and move the file pointer back and forth freely, but that's not the case with file descriptors linked to network connections.
  - File descriptors, however, do 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.

Lecture 02: Implementing `t` to emulate `tee`

- Overview of `tee`
  - The `tee` builtin 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`.

```
myth60$ cat alphabet.txt | ./tee one.txt two.txt three.txt
abcdefghijklmnopqrstuvwxyz
myth60$ cat one.txt
abcdefghijklmnopqrstuvwxyz
myth60$ cat two.txt
abcdefghijklmnopqrstuvwxyz
myth60$ diff one.txt two.txt
myth60$ diff one.txt three.txt
myth60$
```

Lecture 02: Implementing `t` to emulate `tee`

- Overview of `tee` (continued)
  - If the file `vowels.txt` contains the five vowels and the newline character, and `tee` is invoked as follows, `one.txt` would be rewritten to contain only the English vowels, but `two.txt` and `three.txt` would be left alone.

```
myth60$ more vowels.txt | ./tee one.txt
aeiou
myth60$ more one.txt
aeiou
myth60$ more two.txt
abcdefghijklmnopqrstuvwxyz
myth60$
```

  - Full implementation of our own `t` executable, with error checking, is right here.
  - Implementation replicates much of what `copy.c` does, but it illustrates how you can use low-level I/O to manage many sessions with multiple files. The implementation inlined across the next two slides omits error checking.

Lecture 02: Implementing `t` to emulate `tee`

```c
int main(int argc, char *argv[]) {
  int fds[argc];
  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;
}
```

Lecture 02: Implementing `t` to emulate `tee`

```
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);
  }
}
```

- Features:
  - Note that **`argc`** incidentally provides a count on the number of descriptors that we write to. That's why we declare an **`int`** array (or rather, a file descriptor array) of length **`argc`**.
  - **`STDIN_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.
  - I assume all system calls succeed. I'm not being lazy, I promise. I'm just trying to keep the examples as clear and compact as possible. The official copies of the working programs up on the **`myth`** machines include real error checking.

Lecture 02: 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:

```
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 (e.g. **`man 2 stat`**, **`man 2 lstat`**, etc.)

Lecture 02: Using `stat` and `lstat`

- The `struct stat` contains the following fields (source)

```
struct stat {
  dev_t st_dev;         // ID of device containing file
  ino_t st_ino;         // file serial number
  mode_t st_mode;       // mode of file
  // many other fields (file size, creation and modified times, etc)

};
```

  - 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.
  - The next two examples illustrate how the `stat` and `lstat` functions can be used to navigate and otherwise manipulate a tree of files within the file system.

Lecture 02: Implementing `search` to emulate `find`

- `search` is our own imitation of the `find` builtin.
  - 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.

```
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
myth60$
```

Lecture 02: Implementing `search` to emulate `find`

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

```
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); // buffer overflow impossible
  listMatches(path, length, pattern);
  return 0;
}
```

Lecture 02: Implementing `search` to emulate `find`

- Implementation details of interest:
  - This is the first example to call **`lstat`**, which extracts information about the named file and populates the **`st`** with that information.
  - You'll also 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`** decided whether the file is a link. We'll use all of these in our next example.
  - Most of what's interesting is managed by the **`listMatches`** function, which does a recursive (CS106B, ftw!) depth-first traversal of the filesystem to see what files coincidentally to match the **`name`** of interest.
  - The implementation of **`listMatches`**, which appears on the next slide, makes use of these three library functions to iterate over all of the files within a named directory.

```
DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
```

Lecture 02: Implementing `search` to emulate `find`

- Here's the implementation of `listMatches`:

```
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);
}
```

Lecture 02: Implementing `search` to emulate `find`

- Implementation details of interest:
  - My 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 I ignore them to avoid cycles and infinite recursion.
  - I use `lstat` instead of `stat` so I know whether an entry is really a link. I ignore links, again because I want to avoid infinite recursion and cycles.
  - If the `stat` record identifies an entry as a *regular file*, I print the entire path iff the entry name matches the name we're searching for.
  - If the `stat` record identifies an entry as a *directory*, I recursively descend into it to see if any of its named entries match the name we're searching for.
  - `opendir` returns access to a record that eventually must be released via a call to `closedir`. That's why my implementation ends with it.

Lecture 02: Implementing `list` to emulate `ls`

- I also present the implementation of **`list`**, which emulates the functionality of **`ls`** (in particular, **`ls -lUa`**). Implementations of **`list`** and **`search`** have much in common, but implementation of **`list`** is much longer.
  - Sample output of my own **`list`** is presented right here:

```
myth60$ ./list /usr/class/cs110/WWW
drwxr-xr-x  8    70296 root       2048 Jan 08 17:16 .
drwxr-xr-x >9 root     root       2048 Jan 08 17:02 ..
drwxr-xr-x  2    70296 root       2048 Jan 08 15:45 restricted
drwxr-xr-x  4 poohbear operator   2048 Jan 08 17:03 examples
-rw-------  1 poohbear operator   2395 Jan 08 15:51 index.html
// others omitted for brevity
myth60$
```

  - Full implementation of **`list.c`** is right here.
  - I don't present the entire implementation here or in lecture. I just show one key function: the one that knows how to print out the permissions information (e.g. **`drwxr-xr-x`**) for an arbitrary entry.

Lecture 02: Implementing `list` to emulate `ls`

- Here's the implementation of `list`'s `listPermissions` function, which prints out the permission string consistent with the supplied `stat` information:
  - First, a helper function and a collection of constants that assist in the implementation of `listPermissions`:

```
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 mode_t kPermissionFlags[] = {
  S_IRUSR, S_IWUSR, S_IXUSR, // user flags
  S_IRGRP, S_IWGRP, S_IXGRP, // group flags
  S_IROTH, S_IWOTH, S_IXOTH  // everyone (other) flags
};
static const size_t kNumPermissionFlags =
   sizeof(kPermissionFlags)/sizeof(kPermissionFlags[0]);
```

Lecture 02: Implementing `list` to emulate `ls`

- Here's the implementation of `list`'s `listPermissions` function, which prints out the permission string consistent with the supplied `stat` information:
  - Finally, the implementation of `listPermissions` itself, which assumes the helper function and constants from the previous slide.

```
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);
}
```