Lecture 02: File Systems, APIs, and System Calls

Principles of Computer Systems

Winter 2020

Stanford University

Computer Science Department

Instructors: Chris Gregg

and Nick Troccoli

PDF of this presentation

Assignment 1: Amazon Search

- The first assignment is meant to get you up to speed on the coding you need to be able to do for the class. It is a mix of CS106B and CS107 ideas.
- The program you will write will search through the (in some cases) enormous Amazon Review database, which we have packaged into two files: the reviews themselves, and an index to all the words in all the reviews.

```
cgregg@myth57:$ ./amazon_search "best thing since sliced bread"
Found 258 matching reviews out of 3093869 reviews in the database.
**********
Review index: 3092071
Product title: Ceiva Internet-Enabled Photo Frame
Product category: Electronics
Star rating: 5 stars
Review headline: Rave Review !   Perfect for mom and grandma !
Review body: This product is the best thing since sliced bread !   I bought 5 of
them for family and friends.  The best application for a ceiva is my grandmothers
room, at the retirement home.  She is too old to use AOL or  WebTV.  Ceiva is the
perfect bedside companion for her.  Everyday, she gets  a new set of 10 pictures
downloaded from the 250 I have  &quot;uploaded&quot; to her album storage section.
She has no computer  skills and doesn't know a jpeg from a gif, but she loves the
pictures I  have sent to her via her Ceiva frame...
Date: 2000-5-10

**********
```

Assignment 1: Amazon Search

- The first assignment is meant to get you up to speed on the coding you need to be able to do for the class. It is a mix of CS106B and CS107 ideas.
- The program you will write will search through the (in some cases) enormous Amazon Review database, which we have packaged into two files: the reviews themselves, and an index to all the words in all the reviews.

```
cgregg@myth57:/usr/class/cs110/staff/master_repos/assign1$ ./amazon_search '"almost killed me"' -k bodysize -r -n 1
Total number of keywords: 483621
Found 4 matching reviews out of 3093869 reviews in the database.
**********
Review index: 3043273
Product title: Segway Human Transporter (HT) p Series
Product category: Electronics
Star rating: 1 stars
Review headline: Segway Danger - almost Killed me.
Review body: I want to worn you before you get an the Segway and take off...I bought mine
here from Amazon and at that time took 4 months to get it...then I started riding it with
no problem everyday I could to get the mail up our 600 feet concrete driveway...when I
returned everytime I hooked it up to the power source. This one day while going up the driveway
which is on an incline...it powered down..and slammed me to the ground....without any notice..
I layed there until my brother came along and help me to the house....my back was brused...and
it knocked some bone spurs loose in my neck....I contacted Amazon/Segway ..and SEGWAY said
there was a recall to the computer system and I sent it back...and sold it ASAP...Segway never
offered and relief on my bills..etc...MY advise to riders is they nee some sort of fail safe
system for backway fails....I know how one could rig up a devise for the forward fall..but
short of some sort of backward fall...I don't...I have played sports in school but never have
been slammed a hard as that fall....I would say..be very careful and don't look for Segway to
offer any type of help if you are hurt. <br />RDM
Date: 2005-8-10
```

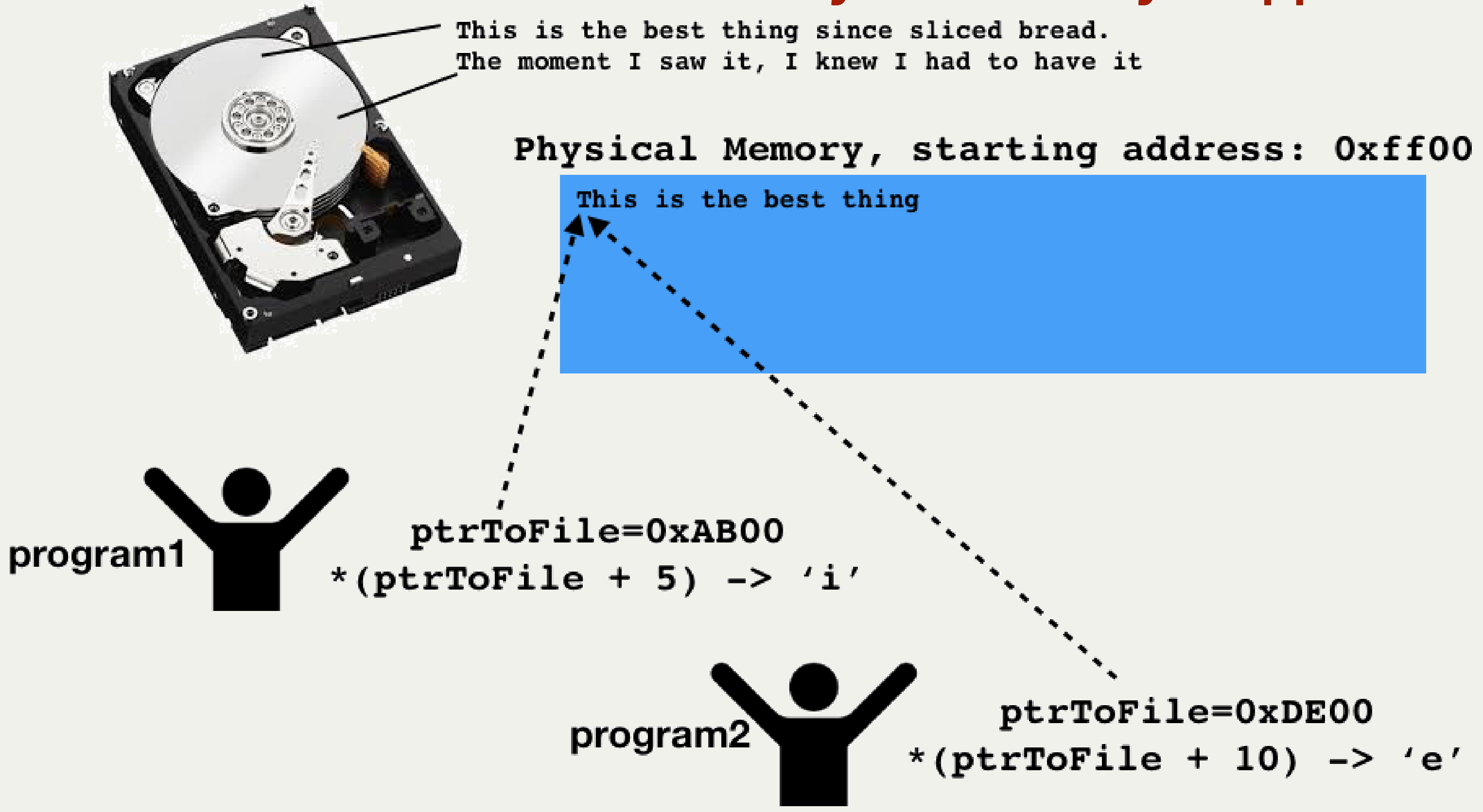
Assignment 1: Amazon Search

- The two files for each database have been created in a format that allows *fast binary searching*. The **databaseFile** is built on a data structure that allows O(1) access to a particular review, by index, and the **keywordIndexFile** is built on a data structure that allows binary searching for words that are found in the reviews. This is where the CS107 stuff comes in: you need to understand the file formats *exactly* and you need to use pointer arithmetic to parse them.
- You will also use C++ standard template library (STL) classes to do the binary searching in these files. Specifically, you will use the `lower_bound` function from the STL. The function is a bit subtle -- you need to take some time to understand how it works. For example, it takes an *iterator*, which in our case is just a pointer to the data. Also, when searching, it returns an "Iterator pointing to the first element that is not less than value, or last if no such element is found." (see the link above for details).
- Once you have worked out how to search for keywords (this is the CS106B part of the assignment), and once you have compiled a data structure that contains the indexes for all reviews that match the user's search query, you need to sort the reviews by a client-provided search function, so the user can get the reviews in a particular order.

Assignment 1: Amazon Search

- You might be wondering how 250 students are all going to be reading in gigantic database files into shared Myth machines without running out of memory. It turns out that you *don't* read the files into memory -- we set up what is called a *memory map* to the files on the disk. This is a CS 110 idea, actually, so let's spend a minute discussing it.
- When a file is memory mapped, the operating system keeps track of requests for particular parts of the file, and only loads the data that is requested into main memory (note: *main memory* is RAM. Each myth machine has 32GB of RAM, which seems like a lot, but can become scarce with many users).
- Here is where it gets really cool: if two users on the same machine happen to memory map the same file, and both ask for the same portion of data from that file, the OS recognizes this, and simply keeps a single copy in memory. Each user has a pointer that maps to the region of memory where the data is stored, and when they ask for a particular part of the file (using pointer arithmetic), if that particular part of the file is already loaded into memory, then the OS can translate their pointer to the physical memory location and return the data that is there, without needing to go back to the disk at all. It is both fast *and* saves memory -- in other words, a great idea!

Assignment 1: Amazon Search: Virtual Memory for memory mapped files



```
This is the best thing since sliced bread.
The moment I saw it, I knew I had to have it
```

```
Physical Memory, starting address: 0xff00
```

```
This is the best thing
```

```
ptrToFile=0xAB00
*(ptrToFile + 5) -> 'i'
```

```
ptrToFile=0xDE00
*(ptrToFile + 10) -> 'e'
```

There are *three* different representations of the location of the start of the file. The “real” memory location in physical memory is `0xff00`, and the *virtual memory* locations each user has (`0xAB00` and `0xDE00`). The OS translates the user’s pointer to a physical memory location, and returns the values found there. The users *think* they are accessing memory locations `0xAB00` and `0xDE00`, but they are *really* accessing memory location `0xff00`.

Assignment 1: Lambda Functions

- To go back to the `lower_bound` function for a moment: part of the assignment says, *"Remember, you must use the lower_bound function and a C++ lambda function to perform a binary search to find keywords."*
- What is this about a "C++ lambda"? This is likely a new concept for you, so let's discuss it.
- A *lambda function* is a function that is usually placed inline as a parameter to another function, which expects the parameter to itself be a function (I N C E P T I O N)
- Before we talk about lambdas specifically, let's back up a bit and recall what it means to pass around function pointers (CS107 stuff)
  - Function pointers provide flexibility. Recall the **qsort** function:

```
void qsort(void *base, size_t nmemb, size_t size,
                  int (*compar)(const void *, const void *));
```

- The last parameter is a function pointer that defines the comparison function **qsort** will use when it sorts an array.
- The caller of the **qsort** function passes in the function pointer, and **qsort** itself simply calls it, expecting an `int` return value. **qsort** does not care about the details of how the comparison is done, it just relies on it to provide a legitimate result.

Assignment 1: Lambda Functions

- Let's look at an example program: (full program here)

```
 1 int add(int x, int y) { return x + y; }
 2 int sub(int x, int y) { return x - y; }
 3
 4 void modifyVec(vector<int> &vec, int val, function<int(int, int)>op) {
 5    for (int &v : vec) {
 6       v = op(v,val);
 7    }
 8 }
 9
10 int main(int argc, char *argv[]) {
11     string opStr = string(argv[1]);
12     int val = atoi(argv[2]);
13
14     vector<int> vec = {1, 2, 3, 4, 5, 10, 100, 1000};
15     printVec("Original",vec);
16     cout << "Performing " << opStr << " on vector with value " << val << endl;
17
18     if (opStr == "add") modifyVec(vec, val, add);
19     else if (opStr == "sub") modifyVec(vec, val, sub);
20
21     printVec("Result",vec);
22
23     return 0;
24 }
```

```
./fun_pointer add 12
Original: 1, 2, 3, 4, 5, 10, 100, 1000,
Performing add on vector with value 12
Result: 13, 14, 15, 16, 17, 22, 112, 1012,
```

- We've created two functions, **add** and **sub**, that get called by **modifyVec**.
- The `function<int(int, int)>` **op** parameter is a C++ way of creating a function pointer.
- Note on lines 18 and 19, the **add** and **sub** functions *do not get called immediately -- they get called when **modifyVec** gets around to calling them.*

Assignment 1: Lambda Functions

- With a *lambda* function, we can replace the **add** and **sub** functions with an *inline* function (full program here).

```
 1 void modifyVec(vector<int> &vec, int val, function<int(int, int)>op) {
 2    for (int &v : vec) {
 3       v = op(v,val);
 4    }
 5 }
 6
 7 int main(int argc, char *argv[]) {
 8     string opStr = string(argv[1]);
 9     int val = atoi(argv[2]);
10
11
12     vector<int> vec = {1, 2, 3, 4, 5, 10, 100, 1000};
13     printVec("Original", vec);
14     cout << "Performing " << opStr << " on vector with value " << val << endl;
15
16     if (opStr == "add") modifyVec(vec, val, [](int x, int y) {
17                 return x + y;
18             });
19     else if (opStr == "sub") modifyVec(vec, val, [](int x, int y) {
20                 return x - y;
21             });
22
23     printVec("Result", vec);
24
25     return 0;
26 }
```

- Lines 16-18 and 19-21 are where the magic happens.
- A lambda function has the following signature:

```
[ captures ] ( params ) { body }
```

- We will talk about *captures* in a moment, but for now, see that the *params* and the *body* comprise a similar form to our original functions for **add** and **sub**.

Assignment 1: Lambda Functions

- So a lambda function is just an inline function. But, it can be more than that. We *may* want to allow the function to utilize variables from the scope where the function is being called. Let's say we changed modifyVec from this:

```
void modifyVec(vector<int> &vec, int val, function<int(int, int)>op) {
   for (int &v : vec) {
      v = op(v,val);
   }
}
```

To this:

```
void modifyVec(vector<int> &vec, function<int(int)>op) {
   for (int &v : vec) {
      v = op(v);
   }
}
```

- In other words, now we want the function that calls **modifyVec** to also handle the value we are updating by. This would be difficult to accomplish with a regular function pointer.
- But, with a lambda function, it is possible.

Assignment 1: Lambda Functions

- Here is our new version, with a modified lambda function:

```
void modifyVec(vector<int> &vec, std::function<int(int v)>op) {
   for (int &v : vec) {
      v = op(v);
   }
}

int main(int argc, char *argv[]) {
    string opStr = string(argv[1]);
    int val = atoi(argv[2]);

    vector<int> vec = {1, 2, 3, 4, 5, 10, 100, 1000};
    printVec("Original", vec);
    cout << "Performing " << opStr << " on vector with value " << val << endl;

    if (opStr == "add") modifyVec(vec, [val](int x) {
                return x + val;
            });
    else if (opStr == "sub") modifyVec(vec, [val](int x) {
                return x - val;
            });

    printVec("Result", vec);

    return 0;
}
```

- In this version, we have *captured* the variable **`val`**, using the bracket notation. This allows the lambda function, when it is called (remember, it *isn't called immediately*) to use **`val`**.
- There are multiple ways to capture variables -- often, we want to capture them by reference. If we wanted to capture **`val`** as a reference, we would call it as follows:

```
    if (opStr == "add") modifyVec(vec, [&val](int x) {
                return x + val;
            });
```

Assignment 1: Lambda Functions

- Some more comments on lambda functions:
  - Lambda functions are critical when we have C++ classes, too -- without lambdas, you can't call class functions from a non-class function (this is a key reason why it is necessary for the `lower_bound` function for assignment 1!)
  - If you want to capture all class variables, you can use `[this]` as a capture clause.
  - You can capture multiple variables in a capture clause, e.g., `[this, val, &myVec]`
  - Basically, any in-scope variable you want to use in the lambda function must be captured in the capture clause.
  - We will use lambda functions a great deal when we get to threading, so learn it well on this assignment.

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 interaction with the file system. We will focus here 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`**s, and **`ofstream`**s.
- 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`

- 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.

Back to file systems: 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 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;
}
```

- The **`read`** system call will *block* until the requested number of bytes have been read. If the return value is 0, there are no more bytes to read (e.g., the file has reached the end, or been closed).
- If **`write`** returns a value less than **`count`**, it means that the system couldn't write all the bytes at once. This is why the **`while`** loop is necessary, and the reason for keeping track of **`bytesWritten`** and **`bytesRead`**.
- 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.

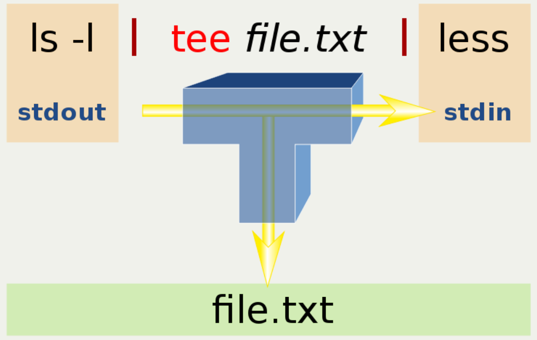
Pros and cons of file descriptors over `FILE` pointers and C++ `iostreams`

- 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 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`

- Overview of **`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`**.

```
$ 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
$
```



Source: https://commons.wikimedia.org/wiki/File:Tee.svg

- 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.

```
$ cat vowels.txt | ./tee one.txt
aeiou
$  cat one.txt
aeiou
```

- 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 omit error checking.

Implementing `t` to emulate `tee`

- Features:
  - Note that **`argc`** incidentally provides a count on the number of descriptors that write to. That's why we declare an integer 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.

```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;
}

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);
  }
}
```

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.)

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.

Using `stat` and `lstat`

- **`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.

```
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$
```

Using `stat` and `lstat`

- 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.

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

Using `stat` and `lstat`

- Implementation details of interest:
  - This is our first example that actually calls **`lstat`**, which extracts information about the named file and populates the struct **`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 depth-first traversal of the filesystem to see what files just happen 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);
```

Using `stat` and `lstat`

- 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`

- 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`

- We 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 Jerry Cain's `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 cgregg   operator   2048 Jan 08 17:03 examples
-rw-------  1 cgregg   operator   2395 Jan 08 15:51 index.html
// others omitted for brevity
myth60$
```

- Full implementation of `list.c` is 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. `drwxr-xr-x`) for an arbitrary entry.

Using `stat` and `lstat`

- Here's the implementation of **`list`**'s **`listPermissions`** function, which prints out the permission string consistent with the supplied **`stat`** information:

```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 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]);

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);
}
```