Lecture 02: File Systems, APIs, and System Calls

Principles of Computer Systems

Spring 2019

Stanford University

Computer Science Department

Lecturer: Chris Gregg

PDF of this presentation

`umask` Redux

- On Monday, we introduced the idea of file permissions, and we discussed `umask`, which is a *default permissions set* for creating files in a directory. Based on the questions after class, I wanted to clear up some things about it.
  - 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 a terminal command:

```
cgregg@myth51:~$ umask
0077
cgregg@myth51:~$ umask 0066
cgregg@myth51:~$ umask
0066
cgregg@myth51:~$
```

- For reasons known to the original creators, the bits that are set in the `umask` *disable* creating permissions for the permissions bits that a program is attempting to set. Example: let's say a program attempts to set the following permissions:
- `rw-rw-rw-` This is a permission set of binary `110110110`, or octal `0666`, meaning that the user, group, and other fields are all set to `rw`.
- If the user's `umask` is set to `0077`, or `000111111`, then all of the permissions for the group and other fields will not be set, even if a program tries to set them when creating a file (unless the umask is changed *in* the program itself).
- You can think of the mask being applied with a bitmask as follows (using our example):
  - `attempt &~umask = 110110110 &~000111111 = 110110110 & 111000000 = 110000000`
  - So, the permissions that will be set will be `110000000`. Example on next slide.

`umask` Redux

- Example:

```
// open_ex_minimal.c
// attempts to create a file with permissions 0644
int main(int argc, char *argv[]) {
    if (argc == 1) {
        printf("Usage: %s filename\n",argv[0]);
        return -1;
    }
    // attempt to set permissions to rw-r--r--
    int file_descriptor = open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0644);
    close(file_descriptor);
    return 0;
}

cgregg@myth51$ ./open_ex_minimal test_file1
cgregg@myth51$ ls -l test_file1
-rw------- 1 cgregg operator 0 Apr  3 06:44 test_file1
```

- If a user *changes* the umask at the terminal, the program will have different behavior:

```
$ umask 0000
cgregg@myth51$ ./open_ex_minimal test_file2
cgregg@myth51$ ls -l test_file2
-rw-r--r-- 1 cgregg operator 0 Apr  3 06:48 test_file2

$ umask 0477
cgregg@myth51$ ./open_ex_minimal test_file3
cgregg@myth51$ ls -l test_file3
--w------- 1 cgregg operator 0 Apr  3 06:49 test_file3
```

Assignment 1: Six Degrees of Kevin Bacon

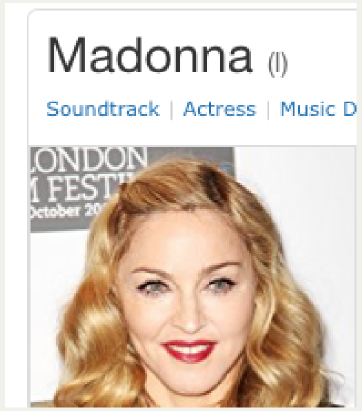
- The first assignment is meant to get you up to speed on the coding you need to be able to do for the class. It is a mix of CS106B and CS107 ideas. **Please re-download if you have already downloaded the handout (there were minor changes that affected copy/pasting the examples)**
- The program you will write is able to determine how to link two film actors through a series of films they have been in. Examples:

```
cgregg@myth65$ ./search "Meryl Streep" "Jack Nicholson (I)"
Meryl Streep was in "Close Up" (2012) with Jack Nicholson (I).
```

- You can see if an actor is in the database as follows:

```
cgregg@myth65$ ./imdbtest "Meryl Streep"

  Meryl Streep has starred in 104 films, and those films are:

    1.) 100 Years (2017)
    2.) A Century of Cinema (1994)

...
    4428.) Zoe Sternbach-Taubman
    4429.) Zvonimir Hace

cgregg@myth65$
```

- Be careful: because some actors have the same name, they may not be in the database without a roman numeral. To check an actor, look them up at imdb.com, and you will see a roman numeral in parentheses next to their name. E.g. for Madonna:



- Note that you would search for Madonna as follows:

```
$ ./imdbtest "Madonna (I)"
```

Assignment 1: Six Degrees of Kevin Bacon

- There are two files that link movie actors to the movies they have acted in. Both have been created in a format that allows *fast binary searching*. The **actorFile** is built on a data structure that allows binary searching for actor names, and the **movieFile** is built on a data structure that allows binary searching for movie titles. 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 data and once you have compiled a data structure for the specific actors your program's user is searching for, you need to perform a *breadth-first search* algorithm to link the two together. This is the CS106B part of the assignment.

Assignment 1: Lambda Functions

- To go back to the `lower_bound` function for a moment: part of the assignment says, "*I am requiring that you use the STL lower_bound algorithm to perform these binary searches, and that you use C++ lambdas (also known as anonymous functions with capture clauses) to provide nameless comparison functions that lower_bound can use to guide its search.*"
- 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:

```cpp
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:

```cpp
    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.

Back to file systems: 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;
}
```

- 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
$
```

- 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.

```
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.

```
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 **`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);
}
```