CS110: API Programming Against the UNIX Filesystem

- **Learning filesystem API calls**
  - Next lecture, we'll be talking about how components of a UNIX filesystem are actually implemented. That alone should be motivation for learning how to code against a collection of APIs that manipulate files, directories, and so forth.
  - You'll benefit by understanding how clients interact with the file system through **system calls**, which are a collection of kernel-resident functions that user programs must go through in order to manipulate the hardware. Requests to open a file, read from a file, to extend the heap, etc, all eventually go through system calls, which are the only functions that can be trusted to touch the system.
  - Today's lecture examples reside in **`/usr/class/cs110/lecture-examples/spring-2015/filesystems`**.
  - The **`/usr/class/cs110/lecture-examples/spring-2015`** directory is a mercurial repository that will be updated with additional examples as the quarter progresses.
    - To get starter, type **`hg clone /usr/class/cs110/lecture-examples/spring-2015 cs110-lecture-examples`** at the command prompt to create a local copy of the master.
    - Each time I mention there are new examples, navigate into your local copy and type **`hg pull; hg update`**.
  - More importantly, read Sections 1 through 5 of the Saltzer & Kaashoek online textbook, paying special attention to the details in Section 5, which will help you with your first assignment (which goes out on Friday).

Filesystem API: Using **stat** and **lstat**

- **`stat` and `lstat`**

  - **`stat`** is a function that populates a **`struct stat`** with information about some named file (regular files, directories, links).
  - **`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.
  - Manual (**`man`**) pages exist for both of these functions (e.g. **`man 2 stat`**, **`man 2 lstat`**, etc.)
  - The **`struct stat`** contain the following fields (source)

```
dev_t     st_dev     ID of device containing file
ino_t     st_ino     file serial number
mode_t    st_mode    mode of file
nlink_t   st_nlink   number of links to the file
uid_t     st_uid     user ID of file
gid_t     st_gid     group ID of file
dev_t     st_rdev    device ID (if file is character or block special)
off_t     st_size    file size in bytes (if file is a regular file)
time_t    st_atime   time of last access
time_t    st_mtime   time of last data modification
time_t    st_ctime   time of last status change
blksize_t st_blksize a filesystem-specific preferred I/O block size for
                     this object.  In some filesystem types, this may
                     vary from file to file
blkcnt_t  st_blocks  number of blocks allocated for this object
```

  - The **`st_mode`** field 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 files within the file system.

Filesystem API: `search`

- **Implementation of `search`**

  - **`search`** is our own simplied implementation of the **`find`** built-in.
  - The following **`main`** relies on **`listMatches`**, which we'll implement a little later. (The full program of interest is online right here.)

```
static void exitUnless(bool test, FILE *stream, int code, const char *control, ...) {
  if (test) return;
  va_list arglist;
  va_start(arglist, control);
  vfprintf(stream, control, arglist);
  va_end(arglist);
  exit(code);
}

int main(int argc, char *argv[]) {
  exitUnless(argc == 3, stderr, kWrongArgumentCount,
             "Usage: %s <directory> <pattern>\n", argv[0]);
  struct stat st;
  const char *directory = argv[1];
  stat(directory, &st);
  exitUnless(S_ISDIR(st.st_mode), stderr, kDirectoryNeeded,
             "<directory> must be an actual directory, %s is not", directory);
  size_t length = strlen(directory);
  if (length > kMaxPath) return 0;

  const char *pattern = argv[2];
  char path[kMaxPath + 1];
  strcpy(path, directory); // no buffer overflow because of above check
  listMatches(path, length, pattern);
  return 0;
}
```

Filesystem API: `search` (continued)

- **Implementation details**
  - Don't worry about the **`va_list`** trickery unless you're curious. I just wanted to unify error checking to a single helper function. My **`exitUnless`** is basically a sexier version of the **`assert`** macro.
  - The first thing that's new to us is the call to **`stat`**, which lifts a bunch of information about the named file off of the file system and populates **`st`** with it.
  - 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 (or a link to one).
  - **`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 file system to see what files just happen to contain a named **`pattern`** as a substring.

Filesystem API: `search` (continued)

- Implementation of `listMatches`

```
static void listMatches(char path[], size_t length, const char *pattern) {
  DIR *dir = opendir(path);
  if (dir == NULL) return; // path isn't a directory
  strcpy(path + length, "/");
  while (true) {
    struct dirent *de = readdir(dir);
    if (de == NULL) break;
    if (strcmp(de->d_name, ".") == 0 || strcmp(de->d_name, "..") == 0) continue;
    if (length + strlen(de->d_name) + 1 > kMaxPath) continue;
    strcpy(path + length + 1, de->d_name);
    struct stat st;
    lstat(path, &st);
    if (S_ISREG(st.st_mode)) {
      if (strstr(de->d_name, pattern) != NULL) {
        printf("%s\n", path);
      }
    } else if (S_ISDIR(st.st_mode)) {
      listMatches(path, length + 1 + strlen(de->d_name), pattern);
    }
  }

  closedir(dir);
}
```

Implementation of `listMatches` (continued)

- **Implementation details**
  - My implementation relies on **`opendir`**, which accepts what is presumably a directory. It returns a pointer to an opaque iterable that surfaces a sequence of **`struct dirent`**s via a series of **`readdir`** calls.
  - If **`opendir`**'s parameter is something other than a directory, it'll return **`NULL`**.
  - When the **`DIR`** has surfaced all of its entries, **`readdir`** returns **`NULL`**. A return value of **`NULL`** says it's all over.
  - The **`struct dirent`** is only *guaranteed* to contain a **`d_name`** field, which is a C string expression of the directory entry's name. **`.`** and **`..`** are among the sequence of named entries, but I ignore them so I don't cycle through any single directory more than once.
  - I use **`lstat`** instead of **`stat`** so I know whether an entry is really a link.
  - If the status clearly identifies an entry as a regular file, then I print the entire path if and only if it contains the **`pattern`** of interest.
  - If the status identifies an entry to be a directory, then I recursively descend into it to see if any of its named entries match the pattern we're looking 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.

Filesystem API: `list`

- **`list` implementation details**
  - I also present the implementation of **`list`**, which emulates the functionality of **`ls`** (in particular, **`ls -lUa`**).
  - The implementation of **`list`** and **`search`** have many things in common, but the implementation of **`list`** is much longer.
  - Full implementation of entire **`list`** executable is right here.
  - Sample output (notice this is my own **`list`**, not **`ls`**!):

```
myth22:/usr/class/cs110/staff/lectures/filesystem> list /usr/
lrwxrwxrwx  1 root     root          26 Sep 19  2012 newsw -> /afs/ir/systems/@sys/newsw
drwxr-xr-x  2 root     root        4096 Oct 10  2012 games
lrwxrwxrwx  1 root     root          26 Sep 19  2012 pubsw -> /afs/ir/systems/@sys/pubsw
lrwxrwxrwx  1 root     root          26 Sep 19  2012 sweet -> /afs/ir/systems/@sys/sweet
drwxr-xr-x  5 root     root        4096 Sep 21  2012 lib32
drwxr-xr-x  2 root     root      118784 Apr 01 16:28 bin
drwxr-xr-x  2 root     root       12288 Mar 21 16:56 sbin
drwxr-xr-x >9 root     root        4096 Oct 17  2012 ..
drwxr-xr-x >9 root     root        4096 Sep 21  2012 local
drwxr-xr-x >9 root     root       73728 Mar 21 16:56 lib
lrwxrwxrwx  1 root     root          26 Sep 19  2012 class -> /afs/ir.stanford.edu/class
drwxr-xr-x >9 root     root       12288 Mar 13 22:28 include
drwxr-xr-x >9 root     root       16384 Oct 10  2012 share
drwxr-xr-x >9 root     root        4096 Sep 19  2012 .
drwxr-xr-x  8 root     root        4096 Sep 21  2012 src
```

  - I don't present the entire implementation. I just show one key function: the one that knows how to print out the permissions information for an arbitrary entry.

Filesystem API: `list`

- ## Implementation of `list`'s `listPermissions`:

```
static inline void updatePermissionsBit(bool flag, char permissions[],
                                        size_t column, char ch) {
  if (!flag) return;
  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);
}
```

  - Full implementation of **`list`** is in **`list.c`**.

Filesystem API: `copy`

- **Implementation of `copy`**
  - The implementation of **`copy`** (designed to mimic the behavior of **`cp`**) illustrates how to use **`open`**, **`read`**, **`write`**, **`close`**, **`stat`**. 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`** executable is right here.
- **Pros and cons of file desciptors 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 layering over 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 nothing else. C **`FILE`** pointers and C++ streams provide automatic buffering and more elaborate formatting options.

Filesystem API (continued)

- **Implementation of copy**

```
int main(int argc, char *argv[]) {
  if (argc != 3) {
    fprintf(stderr, "%s <source-file> <destination-file>.\n", argv[0]);
    return kWrongArgumentCount;
  }

  int fdin = open(argv[1], O_RDONLY);
  if (fdin == -1) {
    fprintf(stderr, "%s: source file could not be opened.\n", argv[1]);
    return kSourceFileNonExistent;
  }

  struct stat st;
  stat(argv[1], &st);
  int fdout = open(argv[2],
                  /* flags = */ O_WRONLY | O_CREAT | O_EXCL,
                  /* mode = */ st.st_mode & S_IRWXU);
  if (fdout == -1) {
    switch (errno) {
      case EEXIST:
        fprintf(stderr, "%s: destination file already exists.\n", argv[2]);
        break;
      default:
        fprintf(stderr, "%s: destination file could not be created.\n", argv[2]);
        break;
    }

    return kDestinationFileOpenFailure;
  }
```

Filesystem API (continued)

- **Implementation of `copy`, continued**

```
  char buffer[1024];
  while (true) {
    ssize_t bytesRead = read(fdin, buffer, sizeof(buffer));
    if (bytesRead == 0) break;
    if (bytesRead == -1) {
      fprintf(stderr, "%s: lost access to file while reading.\n", argv[1]);
      return kReadFailure;
    }

    size_t bytesWritten = 0;
    while (bytesWritten < bytesRead) {
      ssize_t count = write(fdout, buffer + bytesWritten, bytesRead - bytesWritten);
      if (count == -1) {
        fprintf(stderr, "%s: lost access to file while writing.\n", argv[2]);
        return kWriteFailure;
      }
      bytesWritten += count;
    }
  }

  if (close(fdin) == -1) fprintf(stderr, "%s: had trouble closing file.\n", argv[1]);
  if (close(fdout) == -1) fprintf(stderr, "%s: had trouble closing file.\n", argv[2]);
  return 0;
}
```

  - Full implementation is in **`copy.c`** file.

Filesystem API (continued)

- **We're equipped to understand another system call that knows how to duplicate a file descriptor: `dup2`.**
- **Here's the prototype:**

```
int dup2(int oldfd, int newfd);
```

- **`dup2`** attaches **`newfd`** to the same file **`oldfd`** is attached to, closing **`newfd`** first if necessary (but not closing **`oldfd`**).
- **`dup2`** is often used to support pipes and redirections, and it's often used to simplify program logic when characters can be either drawn from standard input or from a file.
- Here's an example of a program—I call it **`filedump`**—that emulates the **`cat`** executable that comes with most UNIX/Linux distributions.
  - Our version copies all data from either standard input (if the program is invoked without any arguments) or the contents of a regular file (if the program is invoked with a single extra argument, assumed to be the name of a readable, regular file) to standard output until the relevant **`read`** call returns 0.
  - I contrive a requirement that all bytes must be drawn from file descriptor 0 (aka **`STDIN_FILENO`**), thereby requiring the conditional use of **`dup2`**.
  - Full implementation is online right here.

```
int main(int argc, char *argv[]) {
  // error checking omitted
  if (argc == 2) {
    int fd = open(argv[1], O_RDONLY);
    dup2(fd, STDIN_FILENO);
    close(fd);
  }

  char buffer[1024];
  while (true) {
    size_t numBytesRead = read(STDIN_FILENO, buffer, sizeof(buffer));
    if (numBytesRead == 0) break;
    size_t numBytesWritten = 0;
    while (numBytesWritten < numBytesRead) {
      numBytesWritten +=
        write(STDOUT_FILENO, buffer + numBytesWritten,
              numBytesRead - numBytesWritten);
    }
  }

  return 0;
}
```