Files, Directories and File I/O

Eric Missimer

File System = Files + Directories

- File system - organizes data in a computer
- Two "basic" types of files in a file system: regular files and directories files
  - Both have a *name* i.e. a string of characters
- Regular files contain data, directory files contain other files
- File system forms a tree with directories as non-leaf nodes and files as leaves

File Types

- Regular files - contain data (text or binary)
- Directory files - contain names of other files pointers to information about them
- Character files - used for character devices such as terminal devices (`/dev/tty1`)
- Block special files - refer to block devices such as disks
- FIFOs - a.k.a. named pipes used for IPC
- Sockets - file types used for communication between processes that may reside on different machines, connected via a network
- Symbolic links - these are aliases that link to a specific file accessible via another name

File Status Information

- The status of a file can be obtained using `stat` (or the `fstat` and `lstat` variants)
- `int stat(const char *path, struct stat *buf)`
- `lstat` - if the path is symbolic link is stat-ed instead of what it points to
- `fstat` - the same as `stat` except it takes a file descriptor instead of a path string

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

Permissions: Read, Write and Execute

- Each file has 3 sets of permissions: owner, group and other
- Permission for regular files is pretty self explanatory
- For directories:
  - read - allows enumerating the directory entries
  - write - allows changing the directory attributes (e.g. its modification time) and creating and removing entries in it
  - execute - allows accessing files inside the directory

File System = Files + Directories

- Root of the file system (tree) is "/"
- By following the tree and concatenating the strings together with "/"s in-between we get a pathname. If we don't start with root with have a relative path, otherwise we have an absolute path.
- Relative paths use the working directory to determine the absolute path

Basic File I/O Operations

- `open`, `read`, `write`, `lseek` and `close`
- These are system calls, the application requests that the operating system perform these actions for it

open

`int open(const char *pathname, int flags, mode_t mode)`
(mode is optional)

- `pathname` is either relative or absolute
- `flags` must include at least: `O_RDONLY`, `O_WRONLY` or `O_RDWR`
- it can bitwise-or'ed with creation flags (see `http://linux.die.net/man/2/open`[1] for details)

[1] `http://linux.die.net` is your friend for this class

write and read

```
ssize_t write(int fd, const void *buf, size_t count)
```

- `fd` - file descriptor returned by `open`
- `buf` - buffer containing the data to be written
- `count` - amount of data to be written

```
ssize_t read(int fd, void *buf, size_t count)
```

- Arguments are analogous to write's arguments

Middle of the file

When we open a file we start reading and/or writing at the beginning of the file.

Middle of the file

When we open a file we start reading and/or writing at the beginning of the file.

What if we want to read or write at the middle of the file?

What if we want to go back to the beginning of a file?

What if we want to write data to the end of a file?

Middle of the file

When we open a file we start reading and/or writing at the beginning of the file.

What if we want to read or write at the middle of the file?

What if we want to go back to the beginning of a file?

What if we want to write data to the end of a file?

That's what `lseek` is for

lseek

`off_t lseek(int fd, off_t offset, int whence)`

- `fd` - file descriptor returned by `open`
- `offset` - numerical value, its meaning depends on `whence`
- `whence` - one of the following:
  - `SEEK_SET` - The offset is set to offset bytes.
  - `SEEK_CUR` - The offset is set to its current location plus offset bytes.
  - `SEEK_END` - The offset is set to the size of the file plus offset bytes.
- What do you think happens if we seek beyond the file size?

lseek

`off_t lseek(int fd, off_t offset, int whence)`

- `fd` - file descriptor returned by `open`
- `offset` - numerical value, its meaning depends on `whence`
- `whence` - one of the following:
  - `SEEK_SET` - The offset is set to offset bytes.
  - `SEEK_CUR` - The offset is set to its current location plus offset bytes.
  - `SEEK_END` - The offset is set to the size of the file plus offset bytes.
- What do you think happens if we seek beyond the file size?
- We create a hole of \0's

close

```
int close(int fd)
```

- `fd` - file descriptor of file being closed
- What do you think happens if you do not close the file descriptor?

Code Example: `file_io`

- Why was the file descriptor 3?

Code Example: `file_io`

- Why was the file descriptor 3?
- 0 to 2 were already in use

Code Example: `file_io`

- Why was the file descriptor 3?
- 0 to 2 were already in use
- But wait that was the first file opened

Code Example: `file_io`

- Why was the file descriptor 3?
- 0 to 2 were already in use
- But wait that was the first file opened
- Standard Streams:
  - 0 - Standard Input
  - 1 - Standard Output
  - 2 - Standard Error

Directories

```
int mkdir(const char *pathname, mode_t mode)
```
(mode specifies the permissions)

```
DIR *opendir(const char *name)

struct dirent *readdir(DIR *dirp)

struct dirent {
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* offset to the next dirent */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported
                                   by all file system types */
    char           d_name[256]; /* filename */
};
```

Code Example: `myls`

Code Example: `myls`

- What was the "." and ".."?

Code Example: `myls`

- What was the "." and ".."?
- The working directory and "one up"

System calls are tedious

- Working with system calls is tedious

System calls are tedious

- Working with system calls is tedious
- There has to be a better way?!?!

System calls are tedious

- Working with system calls is tedious
- There has to be a better way?!?!
- You could wrap the system call in a function

System calls are tedious

- Working with system calls is tedious
- There has to be a better way?!?!
- You could wrap the system call in a function
- Now you have to wrap each system call you plan on using

System calls are tedious

- Working with system calls is tedious
- There has to be a better way?!?!
- You could wrap the system call in a function
- Now you have to wrap each system call you plan on using
- And cut and paste the code around to each program

System calls are tedious

- Working with system calls is tedious
- There has to be a better way?!?!
- You could wrap the system call in a function
- Now you have to wrap each system call you plan on using
- And cut and paste the code around to each program
- Wouldn't it be nice if someone already did this for you?

System calls are tedious

- Working with system calls is tedious
- There has to be a better way?!?!
- You could wrap the system call in a function
- Now you have to wrap each system call you plan on using
- And cut and paste the code around to each program
- Wouldn't it be nice if someone already did this for you?
- They did, its call `libc`

Libraries and `libc`

- Libraries consist of code you can *link* against and utilize
- `libc` is the C standard library
- Side note: C comes with another library, `libm` the C math library, `libm` is not linked by default you need to specify `-lm` to `gcc`
- `libc` contains functions that make it easier to do things such as open, write to and read from a file.

libc

- `int printf(const char *format, ...)`
- I/O is buffered
- `buffer_io`, `buffered_stdout` and `unbuffered_stdout`

libc

- `int printf(const char *format, ...)`
- I/O is buffered
- `buffer_io`, `buffered_stdout` and `unbuffered_stdout`
- full buffering - all I/O is stored in a buffer until it is full and then it is written
- line buffering - I/O is stored until a newline '\n' is encountered

libc - File Operations

- `FILE* fopen(const char *filename, const char *mode)`
- `size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream)`
- `size_t fread(void *ptr, size_t size, size_t count, FILE *stream)`
- `int fseek(FILE *stream, long int offset, int origin)`
- `int fclose(FILE *stream)`

libc - File Operations

- `FILE* fopen(const char *filename, const char *mode)`
- `size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream)`
- `size_t fread(void *ptr, size_t size, size_t count, FILE *stream)`
- `int fseek(FILE *stream, long int offset, int origin)`
- `int fclose(FILE *stream)`
- `int fflush(FILE *stream)`

libc - File Operations

- `FILE* fopen(const char *filename, const char *mode)`
- `size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream)`
- `size_t fread(void *ptr, size_t size, size_t count, FILE *stream)`
- `int fseek(FILE *stream, long int offset, int origin)`
- `int fclose(FILE *stream)`
- `int fflush(FILE *stream)`
- `void setbuf(FILE *stream, char *buffer)`

libc - Memory Management

- `void* malloc(size_t size)`
- `void* calloc(size_t num, size_t size)`
- `void* realloc(void* ptr, size_t size)`

libc - Memory Management

- `void* malloc(size_t size)`
- `void* calloc(size_t num, size_t size)`
- `void* realloc(void* ptr, size_t size)`
- `void free(void* ptr)`

libc - Memory Management

- `void* malloc(size_t size)`
- `void* calloc(size_t num, size_t size)`
- `void* realloc(void* ptr, size_t size)`
- `void free(void* ptr)`
  - What happens if `ptr` does not point to a block of allocated memory (and isn't `void`)

libc - Memory Management

- `void* malloc(size_t size)`
- `void* calloc(size_t num, size_t size)`
- `void* realloc(void* ptr, size_t size)`
- `void free(void* ptr)`
  - What happens if `ptr` does not point to a block of allocated memory (and isn't `void`)
  - *Undefined behavior*