CSCI 330

UNIX System Calls for Files

Jon Lehuta

Northern Illinois University

August 17, 2020

UNIX System Calls for Files - Outline

Opening Files

Before a program can do anything to the contents of a file, it needs to ask the operating system to open that file for it. All file operations happen through system calls that ask the operating system to do things for it.

Northern Illinois University

Opening Files

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

- `pathname` – the path to the file you'd like to open (can be absolute or relative)
- `flags` – these affect how the file is opened (if you use more than one, use the bitwise OR, |, to combine them)
  - `O_RDONLY`, `O_WRONLY`, `O_RDWR` – read only, write only, or both
  - `O_TRUNC` – when writing to an existing file, get rid of the data that was there
  - `O_APPEND` – start writing to the end of the file
  - `O_CREAT` – if the file doesn't exist, create it
- `mode` – if you specify the mode here, then any new file that is created will have this mode

Northern Illinois University

Opening Files

You will notice that the `open` system call returns an integer. If there has been no error, the return value will be positive, and will denote the *file descriptor* for the newly opened file. If an error occurs, `open` will return `-1`, and set the value of `errno` so you can display the error using the `perror` function, or similar.

```
int fd; // file descriptor, or error code
fd = open("path_to_a_file", O_RDWR | O_CREAT, 0755);
if(fd == -1) {
  perror("opening file"); // print human readable error
  // deal with the error, maybe exit if you can't recover
}
```

Closing Files

The operating system tracks which files are open. This uses a small amount of memory, but if you have a file open in a program that is no longer using it, it may prevent other processes from doing what they need to with the file. It is a good practice to always close files when you are done with them. You do this with the `close` system call.

Northern Illinois University

Closing Files

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

- `fd` – the *file descriptor* of the file to be closed. Usually this comes from a previous call to `open`

The `close` function returns zero for success. If there has been an error, `-1` is returned instead, and `errno` is set.

Reading from Files

Reading a file is something that cannot be done without the operating system's help. It involves taking bytes from the contents of a specified file, and copying them into the memory used by your program so they can be worked with.

If you would like to read data from a file, you can use the `read` system call. You will be specifying which file to read from with its *file descriptor*, so you will need to have opened the filed with the system call `open` first, and you must not have called `close` on it yet.

Northern Illinois University

Reading from Files

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

- `fd` – the *file descriptor* of the currently open file to read data from
- `buf` – the data read will be stored at the location specified by this pointer
- `count` – the number of bytes to attempt to read from the file

The system call will return the **number of bytes successfully read**, unless there was an error, in which case `-1` is returned, and `errno` is set.

Northern Illinois University

Reading from Files

```
char stringbuffer[1024]; // 1024-byte array of chars
struct some_struct data; // structured data
// open a couple of files
int fd1 = open("file1", O_RDONLY);
int fd2 = open("file2", O_RDONLY);

ssize_t howmany; // how many bytes read?

// reading from a file into a string buffer
howmany = read(fd1, stringbuffer, 1024);
if(howmany==-1) { perror("reading file 1"); exit(1); }

// read data from a file directly into a data structure
howmany = read(fd2, &data, sizeof(some_struct));
if(howmany==-1) { perror("reading file 2"); exit(2); }
```

Writing to Files

Writing a file involves taking bytes from the memory of your program, and having the operating system copy them into the file.

To write data to a file, you can use the `write` system call. Once again, you will be using the *file descriptor* to specify which file to write into.

Northern Illinois University

Writing to Files

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

- `fd` – the *file descriptor* of a currently open file to write to
- `buf` – a pointer to the start of the memory location to be written to the file
- `count` – the number of bytes to attempt to write

Notice that this function looks a lot like `write`. They have the same parameters, but opposite purposes. The `write` system call returns the number of bytes successfully *written*, or `-1` if there was an error.

Northern Illinois University

Writing to Files

```
// string literal stored as character array
char mystring[] = "Some text to write.";
// pointer to start of character array from above
char * pointer = (char *) mystring;

// open a couple of files
int fd1 = open("file1", O_WRONLY);
int fd2 = open("file2", O_WRONLY);

ssize_t howmany; // how many bytes read?
// write the character array to file 1
howmany = write(fd1, mystring, sizeof(mystring));
if(howmany==-1) { perror("writing to file 1"); exit(1); }
// sizeof gives the wrong answer for pointers, use strlen
howmany = write(fd2, pointer, strlen(pointer));
if(howmany==-1) { perror("writing to file 2"); exit(2); }
```

Finding the Status of Files

The system calls we have talked about so far can be used to read the contents of files. Sometimes you may need information about the file other than what it contains. How big is it? Is it a symbolic link? What operations does it support? One system call that can answer many questions like these is `stat`, which is short for status.

Northern Illinois University

The `stat` System Call

There are three versions of the system call `stat`

```
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- `path` – the path to the file you would like to get the status for
- `fd` – the file descriptor of an already open file (for)
- `buf` – a pointer to the `struct stat` data structure to fill with the status of the file

All three versions return zero on success, and `-1` if an error has occurred. The is set so you can use to display what happened.

The normal version of this function is `stat` . `fstat` uses the *file descriptor* of an already open file instead of using the path. `lstat` is used to look up information on symbolic links, instead of the files they point to.

struct stat

All of these system calls work with a `stat` structure, which contains the following fields:

```
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 filesystem 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 */
};
```

Working with the `st_mode` field

Some of the fields in the structure used by `stat` are fairly straightforward and do exactly what they sound like. The `st_mode` field is a little more complicated.

Instead of being a simple integer that denotes a quantity, each of its bits is treated as an individual true or false value. To strip out the irrelevant bits, you can use bitwise boolean operations.

Using the bitwise AND operator, `&`, on the field, with a mask value, can make sure only the relevant bits are on.

Bitwise Operators

You should be familiar with the logical operators in C++; AND, `&&`, OR, `||`, and NOT, `!`. These operators treat integers as a single true or false value, 0 being false, everything else being true. They return a single true or false value as a result.

The bitwise logical operators; AND, `&`, OR, `|`, XOR `^`, and NOT, `~`; treat each bit of an integer as its own true or false value, and return a new integer as a result. In that result, each bit's value is the result of the logical operation performed on the input bits of the operand in matching positions.

Northern Illinois University

Bitwise Operators

```
bitwise AND   bitwise OR   bitwise XOR
1 if both 1   1 if any 1   1 if 1,0 or 0,1
   1100          1100         1100
 & 1010        | 1010       ^ 1010
 ------        ------       ------
   1000          1110         0110

   bitwise NOT (one's complement)
          ~ 11000001111
          ------------- (each bit inverted)
            00111110000
```

Northern Illinois University

Bitwise Operators - Masks

Each `struct stat` has a field, `st_mode` . It is a 32 bit integer. The last 9 bits of it contain the 9 bits of the file's privileges. There are other bits that have nothing to do with those that may be set in the integer returned. We can use our bitwise AND with a special *mask* that has 0's everywhere but in the bits we want to look at, with the result being an integer with only the relevant bits on.

```
  st_mode 10001111 01110011 11000001 11100100
& mask    00000000 00000000 00000001 11111111 (0777)
-------------------------------------------
  result  00000000 00000000 00000001 11100100 (0644)
```

Northern Illinois University

Example

```
// declare your struct stat -- NOT a pointer
struct stat filestatus;
// Notice the & - we are sending stat a pointer to
// where our data structure is stored in memory
if( stat("path_to_file", &filestatus) == -1) {
  perror("Error in stat"); exit(1); } // handle errors
mode_t &mode = filestatus.st_mode; // for convenience
// Notice that a 0 in front of a number in C++ causes
// the rest of the number to be interpreted as octal
cout << oct << (mode & 0777) << endl; // privileges
// the following will print either 0 or 1 as answer
cout << ((mode & 0111) != 0) << endl; // check for execute
cout << ((mode & 0200) != 0) << endl; // user has write?
cout << ((mode & 0040) != 0) << endl; // group has read?
```

Northern Illinois University

For More Information...

All of the functions dealt with in these slides are system calls and their documentaion can be found in section 2 of the manpages.

```
man 2 open
man 2 close
man 2 read
man 2 write
man 2 stat
man 3 errno
man 3 perror
```

`write`, `stat`, and `open` have other commands that share their name, so you will not get to the right manpage for them without specifying the section number.