Lab 6
File I/O and Simple File Retreival

Prof. Kredo

Due: Start of lab Friday, March 6

Desk	Name:	
	Name:	

Introduction

In this lab you will accomplish several goals:

- Learn basic file I/O concepts
- Develop simple programs that read and write files
- Create a file download program (a simple web client)

Work in pairs for this lab using the equipment at your desk. Distribute the work evenly to make sure both group members know the material, as you will be required to know the material for evaluation.

There are (at least) two common ways to access files through C and C++: the standard C libraries and through system calls. This lab introduces the system call approach since they match more closely with the other system calls used this semester. However, you are free to use either one as long as you satisfy program objectives and requirements.

Always check return values for system calls. Continuing to execute a program after an error results in undefined behavior. Many system calls set a special variable, `errno` to indicate the specific error that occurred. You may find the `errno(3)`, `perror(3)` and `strerror(3)` man pages useful.

Reading Files [30 Points]

In order to read a file, you must first open the file using the `open(2)` system call. Parameters to `open` inform the OS which file you want to access and how you wish to access it. A simple function call to read the `/etc/services/` file would be `open("/etc/services", O_RDONLY)`. The `O_RDONLY` argument informs the OS that you wish to only read the file. Upon opening a file (without error) `open` returns a value called a file descriptor. Whenever you want to reference the file when interacting with the OS, you use the file descriptor. As you'll see in later labs and in the programs, file descriptors are a common way to reference resources.

Once you have successfully opened a file, you can retrieve the contents with the `read(2)` system call. As you might expect, `read` takes an open file descriptor and copies a specified number of bytes to a location you provide. The `read` man page describes how you determine when you have read all the data from a file.

To release the resources associated with a file and close the file you use the `close(2)` system call.

In general, when reading files you will do the following:

1. Open the file with `open(2)`
2. Read the file, possibly multiple times, with `read(2)`
3. Close the file with `close(2)`

Practice what you have just learned by writing a program that reads a file and displays the contents to the terminal (you want to implement a simplified `cat(1)`). Your program should take one argument, the file to read. When you have finished your program, display it to the lab instructor for credit.

Writing Files [30 Points]

Writing a file follows similar procedures, but requires different arguments to `open(2)`. If you wish to only write the file, then you can use the `O_WRONLY` flag or you can use the `O_RDWR` flag if you wish to read and write the file. By default, the OS does not create or clear files and it writes at the start of a file. If you wish to change these settings, you can research the `O_CREAT`, `O_APPEND`, and `O_TRUNC` flags. You specify multiple file flags by using the OR operator between the flags in the call to `open`, such as `open("some file", OWRONLY | O_CREAT)`.

If you use the `O_CREAT` flag, be sure to set the file permissions using the `mode` argument to `open`. The file permission modes are also ORed to indicate multiple values, such as `S_IRUSR | S_IWUSR`.

As you might expect, writing data into an open file occurs with the `write(2)` system call. Be sure to `close` your file when you've finished writing or all your changes may not end up in the file.

In general, when writing files you will do the following:

1. Open the file with `open(2)`
2. Write the file, possibly multiple times, with `write(2)`
3. Close the file with `close(2)`

Practice what you have learned by writing a program that reads a file, converts all characters to upper case, and writes the results to second file with the added extension `.up`. Your program should take one argument, the original file to read. For example, you execute your program as `./a.out some_file` and your program creates a file called `some_file.up` with the same contents of `some_file`, but the characters are all upper case. You can use the `toupper(3)` function to convert characters to upper case. When you have finished your program, display it to the lab instructor for credit.

Web Client [40 Points]

Your last assignment has you write a very simple Web (HTTP) client that requests a resource and writes the server response to a local file. Your program should follow these steps:

1. Open a `STREAM_SOCK` socket to the server `farnsworth.ecst.csuchico.edu` on port 80 (the default HTTP port).

2. Send the request string "`GET /lab_docs/reset_instructions.pdf HTTP/1.0\n\n`" to the server. You don't need to understand the request string at this time. Note that two newlines are required and it is not a typo.

3. Receive the response from the server and write it to a file named "`local_file`" in the current directory. If the file does not exist, create it and if the file already exists, ignore (erase) all existing data. The server will close the connection after sending all the data (see "orderly shutdown" in `recv(2)`). You do not need to parse the server's response; simply dump everything the server sends you into the file.

After downloading the file run the commands `sed -e '/^Date/d' -i local_file` to delete the Date HTTP header and then `md5sum local_file` to generate the MD5 sum of your file. Your program works correctly if the MD5 sum of your file equals 47e87fc36db04c7d797de901b9c46de4.

You can develop your program on the lab machines, jaguar, or your own machine, but you will need to VPN into campus when working from home. If you can ping `farnsworth.ecst.csuchico.edu`, then you should be able to connect.

Turn in your program and an associated Makefile through Learn.