1 Lab 13

Date: Nov 14, 2019

This document first describes the aims of this lab. It then provides necessary background. It then describes the exercises which need to be performed.

1.1 Aims

The aim of this lab is to introduce to you to the C library functions used for doing input/output (I/O). After completing this lab, you should be familiar with the following topics:

- The use plain `char`'s as return values.
- Opening and closing files.
- Reading/writing blocks of bytes.

1.2 Background

Unlike other languages which were prevalent at the time C was first implemented, the C language itself does not specify any I/O facilities. Instead, all I/O in C is relegated to library functions. C comes with I/O functions in its standard library; usually, these functions are declared in the standard header file `<stdio.h>`.

The C I/O functions will work irrespective of which operating system the function is running on. However, they will be implemented using the native I/O facilities; under Unix they will wrap Unix primitives like *file descriptors*, `read¬()` and `write()`.

1.3 Starting Up

Use the startup directions from the earlier labs to create a `work/lab13` directory and fire up a terminal whose output you are logging using the `script` command. Make sure that your `lab13` directory contains a copy of the files directory.

1.4 Standard Input Character Count

Change over to the stdin-char-count directory. Look at the stdin-char-count.c program as well as the Makefile. The program simply outputs the encoding for `EOF` followed by the number of characters read from standard input. However, note that the Makefile uses a special `-f` flag to ensure that a plain `char` type

is unsigned; hence the declaration for `char c` in the program is equivalent to `unsigned char c`.

Compile and run the program on itself:

```
$ ./stdin-char-count <stdin-char-count.c
```

The program should output the value of EOF as `-1` but will then get stuck in an infinite loop. `^C` out of it.

The problem is that we are assigning the output of `getchar()` to an `unsigned char` and then comparing it with a negative value; that comparison will always be false, resulting in the infinite loop.

Without changing the Makefile, change the declaration for `c` so as to avoid the infinite loop. If you now run the program:

```
$ ./stdin-char-count <stdin-char-count.c
```

it should output the number of characters in `stdin-char-count.c`. Compare your result with that from `wc stdin-char-count.c`.

1.5 File Character Count

Change over to the file-char-count directory. Look at the program in the file-char-count.c file. It outputs a count of the number of characters in the file specified by its single command-line argument.

The key function is fopen(). It is used to open the file with name specified by its first argument for reading as specified by its second `"r"` **mode** argument. If the open succeeds, `fopen()` returns a non-`NULL` pointer to a `FILE` ADT. This `FILE` pointer can be used by other functions for performing operations on the file.

Compile the program using `make`. Run it on itself:

```
$ ./file-char-count file-char-count.c
```

Compare the result with that produced by `wc`.

Check to see if the program handles non-existent files:

```
$ ./file-char-count xxx
```

It seems to run ok, but has a major bug. Can you spot the bug?

If you cannot spot the bug, try running the program using valgrind.

```
$ valgrind ./file-char-count file-char-count.c
```

and you will see that the program is leaking memory. So it is allocating memory at some point which is never being released. Try to figure out where that is and fix the bug; test using valgrind.

1.6 File Copy

Change over to the file-copy directory. Look at the program in the file-copy.c file. It copies the file specified by its first command-line argument into the file specified by its second command-line argument.

Again, the key function is fopen(). Not only is it being used to open the file with name specified by the first command-line argument for reading, but it is also being used to open the file with name specified by the second command-line argument for writing. In the first `fopen()`, the mode argument is specified as `"r"`, but the second `fopen()` has its mode argument specified as `"w"`.

Compile the program using `make`. Run it on itself and verify that it makes a correct copy:

```
$ ./file-copy                  #should output usage
$ ls -l file-copy.c t          #t should not be there
$ ./file-copy file-copy.c t
$ ls -l file-copy.c t          #t be there with same char count
$ cmp file-copy.c t            #should compare ok
```

Notice that `t` got created.

Now use your `file-copy` program to copy some other file to `t`.

```
$ ./file-copy Makefile t
$ cmp Makefile t                #should compare ok
```

Set up the destination file so that it cannot be written into:

```
$ chmod a-w t                #turn-off write perms for t
$ ./file-copy file-copy.c t #should fail in the fopen("w")
$ cmp Makefile t             #should compare ok as t unchanged
```

The program would not have noticed the fact that the `fopen()` for writing failed if we had not checked the return value from the `fopen()`. If you look at the code for the function, notice that almost all library calls are checked for errors; that is absolutely necessary for writing good quality code which fails fast when encountering an error situation.

Unfortunately, the `file-copy` program does not do a complete job of error checking. If you look at the *man page* for `fgetc()`, the return value is documented as "EOF on end of file or error". This means that the `while`-loop in the `doCopy()` function will terminate on either a real end-of-file or on error. Add code to `doCopy()` to have the program terminate when an error occurs on `fgetc()`. **Hint**: look at the *man page* for `ferror()`.

1.7 Appending To A File

Stay in the file-copy directory.

Using the "w"-mode argument to the second fopen() will clobber the contents of the file if it already exist. To avoid this, open the file for *append* by changing the mode argument from "w" to "a". Compile and test.

```
$ rm -f t                       #ensure t not present
$ ./file-copy file-copy.c t     #create empty t and append file-
copy.c
$ ./file-copy Makefile t        #append Makefile to t
```

So the destination file t should contain the contents of Makefile appended to the contents of file-copy.c. Verify using a text editor. You can also verify that the sum of the line counts of the two source files match the total number of lines in t:

```
$ wc -l file-copy.c Makefile
$ wc -l t
```

1.8 Buffering

When a function such as fputc() writes a character, the path taken by the data is as follows:

1. Normally, fputc() does not do any I/O. Instead it is set up to write the character into a buffer in memory. This buffer is controlled by the stdio library and is in *user space*.
2. When the buffer becomes full, the contents of the buffer is written from the user space buffer to a kernel space buffer using a OS call like write(). Since this involves calling the OS, it can be quite slow compared to a normal memory write by one or two orders of magnitude.
3. When the kernel buffer becomes full, it is actually written out to the file. This I/O is **extremely slow** compared to normal memory writes by several orders of magnitude.

When reading a character using fgetc() the data flows in the other direction: from the file to the kernel buffer and then into a stdio buffer using read() and finally into the program using a function like fgetc().

Without root access, we cannot control (3). However, the stdio library does allow us to control the stdio buffer using setvbuf() and friends.

Change over to the no-buffer-copy directory and look at the no-buffer-copy.c file. The program uses an optional extra command-line argument: stdio buffering is turned on iff that extra argument is specified and equal to 1. Compile and measure the difference when copying the gcc executable into the /dev/null data sink:

```
$ time ./no-buffer-copy `which gcc` /dev/null 1
$ time ./no-buffer-copy `which gcc` /dev/null 0
```

You should see an appreciable difference in performance.

1.9 Record I/O

Change over to the rec-io directory and look at the gen-rand-points.c. This program generates a number (given by its first argument) of random 2-dimensional points with coordinates in [0, 1000], while writing them to a file (given by its second argument) **in binary**.

Build the program by typing `make gen-rand-points` and run it.

```
$ ./gen-rand-points 100 points.dat
```

It will print out the average magnitude of all the generated points.

Look at the generated `points.dat` file using a text editor. You should see that it looks like garbage as it contains the binary representation of the points.

Each point is written out to the file using fwrite(). Look at its documentation to understand what it does.

Given this binary dump of the points, it is possible to read back the points using fread(). Build the `stat-points` program and run it on the generated points in `points.dat`:

```
$ ./stat-points points.dat
```

It will print out statistics about the magnitude of the points. Note that the average should match the average printed out by the `gen-rand-points` program.

The `stat-points` program reads each point in to a dynamically grown array and sorts the array in order to determine the `min`, `max` and `median`. The code in dyn-array.h provides a specification for a dynamic array ADT and dyn-array.c provides its implementation.

These programs shows that using `fread()` and `fwrite()` it is possible to dump out binary data. However, the binary data has severe portability problems:

- If `points.dat` was written out on a little-endian system and read back on a big-endian system, the results would be garbage.
- If `points.dat` was read back in on a system using a different `int` size than the one on which it was written, then the results would be garbage.
- Even if `points.dat` was read and written on the same system, but the reading and writing programs were compiled using separate compilers, or the same compiler with different options, the result could be garbage (this is likely to be prevented because of ABI compatibility reasons).

Nevertheless, with `gen-rand-points` and `stat-points` running on the same system, they seem to work. Unfortunately, they contain a major bug.

By default, when a file is opened by the C library, it is assumed that it is a **text** file containing a sequence of lines. However, that depends on the definition of lines which is system-dependent:

- The C libraries assume that a line is a maximum sequence of characters not containing a newline `'\n'` character and terminated by a newline.
- Under Unix systems, a line is a maximum sequence of characters not containing a linefeed `'\n'` character and terminated by a linefeed.
- Under Windows, a line is a maximum sequence of characters not containing the sequence carriage-return `'\r'` line-feed `'\n'` and terminated by the two character sequence carriage-return, line-feed.
- Under classic Mac-OS systems (not OS/X which is basically Unix), a line is a maximum sequence of characters not containing a carriage-return `'\r'` character and terminated by a carriage-return.

When doing I/O on text files, the C library I/O routines translates between system line-endings and the C line-endings. On Unix, this translation is the identity function but is non-trivial on other systems. For example, the character sequence `'\r' '\n'` in a Windows file is read into a C program as a single character `'\n'`.

Hence doing I/O of binary data using text files is wrong. It will work fine under Unix, but will be incorrect under Windows. For example, the integer 10 (which is the Ascii code for `\n` will be output as `'\r'` followed by a `'\n'`.

The fix for this is to open the files for binary I/O; this can be done by adding a `b` to the second mode string argument of `fopen()`.

1. Fix the modes in the `gen-rand-points` and `stat-points` program.
2. Add an optional second file-name argument to `stat-point`. If that argument is provided, the statistics should be appended to that file instead of being written on standard output. Test and ensure that your modified program is `valgrind`-clean.