Assignment 5. C programming and debugging

Useful pointers

- Ian Cooke, C for C++ Programmers (2004). This describes C89, also known as C90 and ANSI C. C11, the current version of C, supports `//` comments, declarations after statements, and (if you include `<stdbool.h>`) `bool`.
- Brian Kernighan and Dennis Ritchie, *The C Programming Language*, 2nd edition (1988), Prentice Hall, ISBN 0-13-110362-8. This classic book on C is still the language's best description, albeit limited to C89.
- Tom Plum, Introduction to C11 (2013).
- Parlante, Zelenski, et. al, Unix Programming Tools (2001), section 3 — `gdb`.
- Valgrind Quick Start Guide (2018)
- Valgrind User Manual (2018)
- Richard Stallman, Roland Pesch, Stan Shebs, *et al.*, Debugging with GDB (2018)

Laboratory: Debugging a C program

As usual, keep a log in the file `lab5.txt` of what you do in the lab so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a true lab notebook, in which you briefly note down what you did and what happened.

You're helping to maintain an old stable version of `coreutils`, but that version has a bug in its implementation of the `ls` program. (Actually, it has two bad bugs, but for now we'll just look at the first one.)

The bug is that `ls -t` mishandles files whose time stamps are very far in the past. It seems to act as if they are in the future. For example:

```
$ tmp=$(mktemp -d)
$ cd $tmp
$ touch -d '1918-11-11 11:00 GMT' wwi-armistice-cs35L
$ touch now
$ sleep 1
$ touch now1
$ TZ=UTC0 ls -lt --full-time wwi-armistice-cs35L now now1
-rw-r--r-- 1 eggert csfac 0 1918-11-11 11:00:00.000000000 +0000 wwi-armistice-cs35L
-rw-r--r-- 1 eggert csfac 0 2018-10-29 16:43:16.805404419 +0000 now1
-rw-r--r-- 1 eggert csfac 0 2018-10-29 16:43:15.801376773 +0000 now
$ cd
$ rm -fr $tmp
```

Build this old version of coreutils as-is, and then again with this renaming patch. What problems did you have when building it as-is, and why did the renaming patch fix them?

Reproduce the problem. Use a debugger to figure out what went wrong and to fix the corresponding source file.

Construct a new patch file `lab5.diff` containing your coreutils fixes, in the form of a ChangeLog entry followed by a `diff -u` patch.

Also, try to reproduce the problem in your home directory on the SEASnet Linux servers, instead of using the `$tmp` directory. When running the above test case, use the already-installed `touch` and `ls` utilities instead of the old version of coreutils. How well does SEASnet do?

Homework: Sorting encrypted text

The basic idea is that we want to sort obfuscated data without deobfuscating and reobfuscating it. That is, our input is an obfuscated file, and we could compute the output by deobfuscating the input, sorting the deobfuscated data, and then reobfuscating the resulting output—except that we do not want to obfuscate or deobfuscate anything.

Write a C function `frobcmp` that takes two arguments `a` and `b` as input and returns an `int` result that is negative, zero, or positive depending on whether `a` is less than, equal to, or greater than `b`. Each argument is of type `char const *`, and each points to an array of non-space bytes that is followed by space byte. Use standard byte-by-byte lexicographic comparison on the non-space bytes, in the style of the `memcmp` function, except that you should assume that both arrays are frobnicated, (i.e., trivally obfuscated via `memfrob`) and should return the equivalent of running `memcmp` on the corresponding unfrobnicated arrays. If one unfrobnicated array is a prefix of the other, then consider the shorter to be less than the longer. The space bytes are not considered to be part of either array, so they do not participate in the comparison.

For example, `frobcmp ("*{_CIA\030\031 ", "*`_GZY\v ")` should return a positive `int` because `"*{_CIA\030\031"` is `"\0Quick23"` frobnicated and `"*`_GZY\v"` is `"\0Jumps!"` frobnicated, and `"\0Quick23"` is greater than `"\0Jumps!"` in the ASCII collating sequence. As the example demonstrates, null bytes `'\0'` are allowed in the byte arrays and do contribute to the comparison.

Your implementation should not invoke `memfrob`, as that would mean that memory would temporarily contain a copy of the unfrobnicated data. Instead, it should look only at frobnicated bytes one at a time, and unfrobnicate them "by hand", so to speak.

Use your C function to write a program `sfrob` that reads frobnicated text lines from standard input, and writes a sorted version to standard output in frobnicated form. Frobnicated text lines consist of a series of non-space bytes followed by a single space; the spaces represent newlines in the original text. Your program should do all the sorting work itself, by calling `frobcmp`. If standard input ends in a partial record that does not have a trailing space, your program should behave as if a space were appended to the input.

Use `<stdio.h>` functions to do I/O. Use `malloc`, `realloc` and `free` to allocate enough storage to hold all the input, and use `qsort` to sort the data. When debugging, you may find the AddressSanitizer (asan) and the Undefined Behavior Sanitizer (ubsan) useful; these can be enabled with the GCC options `-fsanitize=address` and `-fsanitize=undefined`, respectively.

Do not assume that the input file is not growing: some other process may be appending to it while you're reading, and your program should continue to read until it reaches end of file. For example, your program should work on the file `/proc/self/status`, a "file" that is constantly mutating: it always appears to be of size 0 when you `ls` it, but it always contains nonempty contents if you read it. You should make sure your program works on empty files, as well as on files that are relatively large, such as `/usr/local/cs/gcc*/libexec/gcc/*/*/cc1plus` on SEASnet.

If the program encounters an error of any kind (including input, output or memory allocation failures, it should report the error to `stderr` and exit with status 1; otherwise, the program should succeed and exit with status 0. The program need not report `stderr` output errors.

For example, the shell command:

```
printf '*~BO *{_CIA *hXE]D *LER #@_GZY #E\\OX #^BO #FKPS #NEM\4' |
./sfrob |
od -ta
```

should output:

```
0000000   *   h   X   E   ]   D  sp   *   {   _   C   I   A  sp   *   ~
0000020   B   O  sp   *   L   E   R  sp   #   N   E   M eot  sp   #   @
0000040   _   G   Z   Y  sp   #   F   K   P   S  sp   #   E   \   O   X
0000060  sp   #   ^   B   O  sp
0000066
```

because frobnicating `sfrob`'s input and then appending a trailing newline (because the last frobnicated byte is not a newline) yields:

```
^@The
^@Quick
^@Brown
^@fox
^Ijumps
^Iover
^Ithe
^Ilazy
^Idog.
```

where `^@` denotes a null byte `'\0'`, and `^I` denotes a tab byte `'\t'`. Sorting this yields:

```
^@Brown
^@Quick
^@The
^@fox
^Idog.
^Ijumps
^Ilazy
^Iover
^Ithe
```

and frobnicating this yields the output shown above.

Submit

Submit the following files.

- The files `lab5.txt` and `lab5.diff` as described in the lab.
- A single source file `sfrob.c` as described in the homework.

All files should be ASCII text files, with no carriage returns, and with no more than 200 columns per line. The C source file should contain no more than 132 columns per line. The shell commands

```
expand lab5.txt lab5.diff | awk '/\r/ || 200 < length'
expand sfrob.c | awk '/\r/ || 132 < length'
```

should output nothing.

$Id: assign5.html,v 1.44 2020/03/27 22:30:29 eggert Exp $