Assignment 0

CS 4414 - Operating Systems

January 26, 2017

Abstract

Please read all the text in this assignment very carefully. It is quite long and may take 10-20 minutes but it will save you a lot of hassle during this assignment and later ones.

Introduction

No one likes more assignments. So it may be rather easy for you to hate this assignment and me for writing it. That is the last thing I want to have happen so please allow me to explain my rationale behind doing this; at the end, I hope you agree that although this may give you a bit of a hard time right now, it will make the rest of the semester much easier, more enjoyable and more beneficial.

The purpose of this newly added "assignment 0" is simple: to ensure you are comfortable doing common tasks in C. Many of you have not coded in a low-level language since CS2150 and thus are a bit rusty. Even in CS2150, you were exposed to so much new material that your focus most likely was not learning the nuances of C/C++.

In CS4414, you will face programming assignments with scope and scale that you most likely have no experience with. The problems are fairly large and there are very few, if any, instructions provided concerning the *how* - for most assignments, you will simply be provided a set of specifications (the spec) and are expected to design, implement, debug and test your code yourself. The *design* procedure for some of the assignments is non-trivial as you will have to keep a lot of details in your head, think about a lot of different cases, and understand many Unix and POSIX specifications and contracts. In the past, many students have had an extremely tough time designing and implementing solutions to these complicated machine problems *all while being unfamiliar with the very tool they are using to solve the problems*, i.e. the C/C++ programming language.

Restated a bit differently, the main purpose of this assignment is to isolate all the language-related frustrations and "aha" moments from assignments 1-5. We sincerely believe that spending a few hours (if that) right now, making sure you understand the nuances of C and common tools and design patterns of low-level programming will greatly reduce overall frustration (pronounced /hātrd/) for the course and its content.

A couple notes before we begin:

- All modules within this assignment are to be done in C (meaning we will use a std99 C compiler [gcc -std=c99] to grade your code, not a C++ compiler). This is to ensure that you learn the importance of designing your own data structures as well as common patterns in procedural languages.
- All modules are to be turned in using the client executable, which you will download from Github. More directions will follow in the next section.
- The "official instructor solution" file for each module will be automatically downloaded by the client program when you successfully turn in that module. The purpose behind giving you the solution after you have completed each module in your own way is to show you good design patterns that you may not be aware of.
- Every module asks you to write a program that does something meaningful. As a side-effect, you will be able to easily find their solutions on stackoverflow or forum posts. Understand that the purpose of this assignment will not be fulfilled even if you understand the code you copy-paste. If you would like to take away maximum value from this assignment, **do not look at any sample code in any way shape or form.** It is so important that you learn how to read man pages - I will go so far to say that how comfortable you are understanding a function or library routine **just** from its man page is directly and strongly correlated with how you do in OS assignments. Try getting used to reading man pages while doing these small modules so that it is like second-nature when you start the real assignments.
- Many of you will most likely ignore the previous point and use stackoverflow when you run into issues or unknowns; this is understandable - old habits die hard and when something useful is there why not take advantage? Just understand that usage and behavior are very different things. When you look up how str_tok() is used in a stackoverflow post and say "aha got it," you're understanding *how that person used str_tok() in that code snippet*, not the full behavior and contract of that routine. This is quite possibly the most dangerous scenario in low-level computing: you not understanding the contract of a routine you're using. This is not a huge problem in higher-level languages due to robust error reporting, but it's a huge source of frustration when dealing with C, which is why it's so important that you get in the habit of fully understanding library routines and system calls before using them.

General Submission Instructions

As mentioned before, the grading of these modules is done through an automated system. For each module, you will need to write and submit a single .c file. The client executable provided will be your main means of submitting your file and getting feedback. As with all machine problems in this course, **you are expected to write, test and submit your programs on a 64-bit Linux system.**

In order to download the client executable, first clone the Github repository for the client (Github.com/PedramPejman/os-grader-client):

```
$ git clone https://github.com/pedrampejman/os-grader-client
```

Then cd into the project's root directory (where you see the Makefile) and make the project:

```
$ cd os-grader-client
$ make
```

Now you can submit your programs using the client executable. As with most Unix utilities, when you run the executable with no arguments it prints out its usage; but here is an example usage:

```
$ ./client pp5nv 0 solution_0.c
```

We will grade your submissions in the following way. Compile your submitted file using gcc. Run it with a number of input files (./your_compiled_prog < input.txt) and check its output against our expected output files. You may want to follow this procedure for your own test-cases.

To make testing easier for you, we have provided example input and expected output files in the repo. To test your program, named sol_0.c, do the following:

```
$ gcc std=c99 sol_0.c -o mysol
$ ./mysol < 0/example_input.txt > my_output.txt
$ diff my_output.txt 0/out_example_input.txt
```

The diff should be empty - otherwise your program is incorrect.

Modules

And 2048 years later, let's look at the actual module write-ups.

Module 0 - I/O

Write a C program that will expect one line of input at a time from stdin. If the input is "exit" the program should terminate; otherwise it should write the number of characters of the input to stdout and wait for input on a new line. You may assume each line of input is fewer than 1024 characters. Please note, your program should produce nothing other than the required output (this means no "Please enter input" prompt..).

Pay attention to how fgets() or read() (or whatever system call you use to read from stdin) works - it's a bit more complicated than you may expect. For example, is it always a blocking call? When does it return? How does it handle special characters (`\n`, `\t`, `\0`, etc.). Test your program thoroughly with various types of inputs. *I strongly urge you to re-familiarize yourself with c-strings.* This is an incredibly simple program, you can do it in less than 10 lines of code - you just have to understand c-strings and the system calls well. Also just as a general note, anytime you are done printing to stdout, you should write a newline character so that the cursor goes to the next line.

Example Interaction:

```
$ gcc module0.c -o module0
$ ./module0
hello
5
goodbye
7
exit
$
```

Module 1 - Simple Calculator

In this module, you will write a simple 4-function calculator. Specifically, write a C program that will expect a line of input from stdin. The input will have the following format: <uint_32><operator><uint_32> (General note: the angle brackets won't actually be supplied in the input, it's a convention for specifying formatting. Also, uint_32 means unsigned 32-bit integer.) Upon receiving this input, your program should calculate and output the answer (write it to standard output) and terminate. Your program should be able to support the four basic operators (+-*/). You may assume both integers as well as the answer will be integers in the range [0, 1<<32) (This means from 0 to 2^{32}, including 0, excluding 2^{32}) .

Example Interaction:

```
$ gcc module1.c -o module1
$ ./module1
5*30
150
$
```

Module 2 - Word Selector

Write a C program that allows the user to input a series of strings, and then ask for the kth string. Specifically, your program should expect a number (let's call it n), then expect n space-separated strings on the next line. On the third line, there will be another number, k. It should then output the kth string and terminate. A couple notes: we are, of course, using 0-indexing. You may assume n is in the range (0, 1<<6), k is in range [0, n) and that each individual string is shorter than 32 characters. Obviously, since we are using ASCII character 0x20 (space) as the delimiter, none of the strings can contain it. More specifically, strings will only be composed of [A-Za-z].

Example Interaction:

```
$ gcc module2.c -o module2
$ ./module2
4
this is a test
1
is
$
```

Module 3 - Word Selector (Hard)

You are to rewrite a slightly more difficult version of the name selector program. Write a C program that allows the user to input a series of strings, and then ask for the kth string. However, now the user will not tell you n; instead, she will supply an unknown number of words (one per line) and simply enter a number when ready to supply k. So when your program starts, it is to keep prompting the user for a word until the user enters an integer, k, at which point it should print the kth word inputted and terminate. You may still assume all words are shorter than 32 characters and only contain character [a-zA-Z].

Note, the major source of difficulty is that you cannot know how many names the user will enter before asking for the kth one. Think about what data structure is useful when you don't know the number of items to be kept.

Example Interaction:

```
$ gcc module3.c -o module3
$ ./module3
Pedram
is
a
piece
of
3
piece
$
```

Hint (*please read only after struggling for a little*)
This module requires you to answer the following question: "How do I devise a mechanism to allocate space (malloc) on the fly?" Meaning, what data structure (struct) do I need to implement in order keep track of a list whose length I don't know? Note that you do know the maximum length of each item (word). Please do this first on paper and make sure your algorithm makes sense before implementing any of it. The logic isn't hard but it's not trivial. If you're still stuck, please do come to office hours.

Wrap-up

As I have mentioned hundreds of times in this write-up, as you start getting annoyed with portions of this assignment, please keep its purpose in mind. My hypothesis is that with the help of this assignment, i) on average, the total number of hours students spend on OS homeworks will be much lower this semester and ii) we will hopefully observe fewer students just giving up and not turning assignments in.

If you follow the directions (specially reading manual pages) you should not find yourself struggling too much with any portion of these modules. But if you do to the point that it is leaving a bitter taste in your mouth, please do attend office hours and ask the instructor, TAs and myself for help - struggling is a good thing but if it's just upsetting you with no progress then it's just a waste of time and we'd be happy to push you in the right direction.

My "office hours" will be **TBD**. Please do come by if anything doesn't make sense and/or you're having significant problems with any of the modules.

Happy Happy Coding :)

Cheers,
Pedram