Assignment 0

CS 301 - Operating Systems

1 Introduction

Knowing how to program in C is important for finishing the assignments and projects in the Operating Systems course. This exercise would recap various concepts of C for you. In particular, you need to be comfortable using features like `structs`, lists, pointers, arrays, `typedef` etc.

2 Assignment Description

The task is to write a program called `words` that counts (1) the total amount of words and (2) the frequency of each word in a file(s). It then prints the results to `stdout`. Like most real world programs, your program should read its input from **each** of the files specified as command line arguments, printing the cumulative word counts. If no file is provided, your program should read from `stdin`.

Header files (suffixed by .h) provide an abstraction of the implemented methods in C. They define the objects, types, methods, and documentation. The corresponding .c file provides the implementation of the abstraction. You should be able to write code with the header file without peeking under the covers at its implementation.

In this case, `words/word_count.h` provides the definition of the `word_count struct`, which will be used as a linked list to keep track of a word and its frequency. This has been `typedef`'d into `WordCount`. This means that instead of typing out `struct word_count`, `WordCount` can be used as shorthand. The header file also gives us a list of functions that are defined in `words/word_count.c`. Part of this assignment is to write code for these functions in `words/word_count.c`.

We have provided you with a compiled version of `sort_words` so that you do not need to write the `wordcount_sort` function. However, you may still need to write your own comparator function (i.e. `wordcount_less`). The `Makefile` links this in with your two object files, `words.o` and `word_count.o`. Note that `words.o` is an ELF formatted binary.

For this section, you will be making changes to `words/main.c` and `words/word_count.c`. After editing these files, `cd` into the `words` directory and run `make` in the terminal. This will create the `words` executable. (Remember to run `make` after

making code changes to generate a fresh executable). Use this executable (and your own test cases) to test your program for correctness.

For the below examples, suppose a file called `words.txt` that contains the following content:

```
abc def AaA
bbb zzz aaa
```

2.1 Total word count

Your first task will be to implement total word count. When executed, words will print the total number of words counted to `stdout`. At this point, you will not need to make edits to `word_count.c`. A complete implementation of the `num_words()` and `main()` functions can suffice.

A word is defined as a sequence of contiguous alphabetical characters of length greater than one. All words should be converted to their lower-case representation and be treated as **not case-sensitive**. The maximum length of a word has been defined at the top of `main.c`. After completing this part, running `./words words.txt` should print:

```
The total number of words is:  6
```

2.2 Word frequency count

Your second task will be to implement frequency counting. Your program should print each unique word as well as the number of times it occurred. This should be sorted in order of frequency (low first). The alphabetical ordering of words should be used as a tie breaker. The `wordcount_sort` function has been defined for you in `wc_sort.o`. However, you will need to implement the `wordcount_less` function in `main.c`.

You will need to implement the functions in `word_count.c` to support the linked list data structure (i.e. `WordCount` a.k.a. `struct word_count`). The complete implementation of `word_count.c` will prove to be useful when implementing `count_words()` in `main.c`.

After completing this part, running `./words -f words.txt` should print:

```
1 abc
1 bbb
1 def
1 zzz
2 aaa
```

3 user limits

The operating system needs to deal with the size of the dynamically allocated segments: the stack and heap. How large should these be? Poke around a

bit to find out how to get and set these limits on Linux. Modify `limits.c` so that it prints out the maximum stack size, the maximum number of processes, and maximum number of file descriptors. Currently, when you compile and run `limits.c` you will see it print out a bunch of system resource limits (stack size, heap size, etc.). Unfortunately all the values will be `0`. Your job is to get this to print the **actual** limits (use the soft limits, not the hard limits). (Hint: run `man getrlimit`; `man` pages will be your best friend during these lab exercises.) You should expect output similar to this:

```
stack size:  8000000
process limit:  30000
max file descriptors:  1000
```

You can run `make limits` to compile your code.

4 A-Z of GDB

Now we're going to use a sample program, `map`, for some GDB practice. The `map` program is designed to print out its own executing structure. Before you start, be sure to take a look at `map.c` and `recurse.c` which form the program. Once you feel familiar with the program, you can compile it by running `make map`. Write down the commands you use to complete each step of the following walk-through. Be sure to also record and submit your answers to all questions in bold.

(a) Run `GDB` on the map executable.

(b) Set a breakpoint at the beginning of the program's execution.

(c) Run the program until the breakpoint.

(d) **What memory address does `argv` store?**

(e) **Describe what's located at that memory address. (What does `argv` point to?)**

(f) Step until you reach the first call to `recur`.

(g) **What is the memory address of the `recur` function?**

(h) Step into the first call to `recur`.

(i) Step until you reach the `if` statement.

(j) Switch into assembly view.

(k) Step over instructions until you reach the `callq` instruction.

(l) **What values are in all the registers?**

(m) Step into the `callq` instruction.

(n) Switch back to C code mode.

(o) Now print out the current call stack. (Hint: what does the `backtrace` command do?)

(p) Now set a breakpoint on the `recur` function which is only triggered when the argument is 0.

(q) Continue until the breakpoint is hit.

(r) Print the call stack now.

(s) Now go up the call stack until you reach `main`. What was `argc`?

(t) Now step until the return statement in `recur`.

(u) Switch back into the assembly view.

(v) **Which instructions correspond to the `return 0` in C?**

(w) Now switch back to the source layout.

(x) Finish the remaining 3 function calls.

(y) Run the program to completion.

(z) Quit `GDB`.

5 Submission

Submit the solutions in a pdf document to the github repo where you push the code. Include your name, roll no., email and discipline in the submission.