CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Homework 1 (document version 1.1)
Files, Strings, and Memory Allocation in C

Overview

- This homework is due by 11:59:59 PM on Friday, September 15, 2017.
- This homework will count as 8% of your final course grade.
- This homework is to be completed **individually**. Do not share your code with anyone else.
- You **must** use C for this homework assignment, and your code **must** successfully compile via `gcc` with absolutely no warning messages when the `-Wall` (i.e., warn all) compiler option is used. We will also use `-Werror`, which will treat all warnings as critical errors.
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v16.04.3 LTS. Note that the `gcc` compiler is version 5.4.0 (`Ubuntu 5.4.0-6ubuntu1~16.04.4`).

Homework Specifications

In this first homework, you will use C to implement a directory and text file manipulation program. The goal is to become more comfortable programming in C on Linux, in particular handling strings, working with the filesystem, and dynamically allocating (and re-allocating) memory.

In brief, your program must open and read a directory and process all regular files within that directory. For each regular file, your program must parse all words from the file (if any), store each parsed word into a dynamically allocated array, then display the words and their respective number of occurrences across all files. Note that you must use a second array to keep track of the word occurrence counts.

Command-Line Arguments

The input directory is specified on the command-line as the first argument. The second command-line argument, which is optional, is an integer specifying the number of words to display in the output. Note that words are displayed in the order that they are first encountered. Regardless, your program must process all of the words in all of the regular files in the given directory.

Directories, Regular Files, and Other File Types

Given a directory path, you must use the `opendir()`, `readdir()`, and `closedir()` functions to open, read from, and close the directory. As noted above, the first command-line argument will specify the input directory path to use. For each given file, you must use the `lstat()` system call and the obtained `st_mode` field to determine the file type. To do so in this assignment, use the `S_ISREG(m)` POSIX macro to identify regular files. All other file types should be ignored. Be sure to see the `man` page for `lstat()` for details.

Program Execution

As an initial example, you could execute your program and have it read all files within the current working directory by using `"."` as the directory path, as follows:

```
bash$ ./a.out .
```

Here are other examples of how you could specify relative and absolute paths:

```
bash$ ./a.out /cs/goldsd/OS/assignments/hw1/code/
...
bash$ ./a.out ~/OS
...
bash$ ./a.out ../xyz
...
bash$ ./a.out ~/public.html/docs
...
```

What is a Word?

For this assignment, words are defined as containing only alphanumeric characters and being at least two characters in length. Further, words are case sensitive (e.g., "`Lion`" is different than "`lion`"). Consider using `fopen()`, `fscanf()`, `fgetc()`, `isalnum()`, and other such string and character functions here. And be sure to check out the details of each function by reviewing the corresponding `man` pages from the terminal.

Error Handling

If improper command-line arguments are given, report an error message to `stderr` and abort further program execution. In general, if an error is encountered, display a meaningful error message on `stderr` by using either `perror()` or `fprintf()`, then aborting further program execution.

Error messages must be one line only and use the following format:

```
ERROR: <error-text-here>
```

Required Data Structures

For your data structures, you must use two parallel arrays, meaning that the elements stored in each array at common index j correspond to one another. In this case, each word is tied to its corresponding count. More specifically, the first array contains words encountered in the given input files; and the second array contains integers that represent the number of occurrences of the corresponding word in the given input files.

Your program must store words in the first array as you discover them in a file, initially setting the corresponding integer in the second array to `1`. If a word already exists in the data structure (use linear search here), do not add it again; instead, incremement its corresponding count in the second array.

Note that you must store **all** words in memory, not just the words to be displayed. Further, both arrays must be dynamically allocated on the runtime heap. To do so, dynamically create the first array as an array of character pointers (`char*`) using `calloc()`. Next, dynamically create the second array as an array of integers, also via `calloc()`. Start with an array size of `16`. And if the size of the arrays needs to be increased, use `realloc()` to do so, increasing the size of the array by `16` each time.

You must also dynamically allocate the memory to store each word (since a `char*` is simply a pointer). To do so, use `malloc()` or `calloc()`; be sure to calculate the number of bytes to allocate as the length of the given word plus one, since strings in C are implemented as `char` arrays that end with a `'\0'` character.

To read in each word from a file, you may use a statically allocated character array of size `80`. In other words, you can assume that each word is no more than `79` characters long.

Finally, be sure to use `free()` to ensure all dynamically allocated memory is properly deallocated. Be sure to use `valgrind` to verify that there are no memory leaks.

Required Output

When you execute your program, you must display a line of output each time you allocate or re-allocate memory for the two parallel arrays.

As an example, if you are processing a directory called `xyz` that only contains the following `hw1-input01.txt` file:

```
Once when a Lion was asleep a little Mouse began running up and down upon him;
this soon wakened the Lion, who placed his huge paw upon him, and opened his
big jaws to swallow him. "Pardon, O King," cried the little Mouse: "forgive me
this time, I shall never forget it: who knows but what I may be able to do you
a turn some of these days?" The Lion was so tickled at the idea of the Mouse
being able to help him, that he lifted up his paw and let him go. Some time
after the Lion was caught in a trap, and the hunters, who desired to carry him
alive to the King, tied him to a tree while they went in search of a wagon
to carry him on. Just then the little Mouse happened to pass by, and seeing
the sad plight in which the Lion was, sent up to him and soon gnawed away the
ropes that bound the King of the Beasts. "Was I not right?" said the little Mouse.
         "LITTLE FRIENDS MAY PROVE GREAT FRIENDS."
```

Executing the code as follows should display the output shown below:

```
bash$ ./a.out xyz 10
Allocated initial parallel arrays of size 16.
Re-allocated parallel arrays to be size 32.
Re-allocated parallel arrays to be size 48.
Re-allocated parallel arrays to be size 64.
Re-allocated parallel arrays to be size 80.
Re-allocated parallel arrays to be size 96.
Re-allocated parallel arrays to be size 112.
All done (successfully read 174 words; 104 unique words).
First 10 words (and corresponding counts) are:
Once -- 1
when -- 1
Lion -- 5
was -- 4
asleep -- 1
little -- 4
Mouse -- 5
began -- 1
running -- 1
up -- 3
```

Match the above output format **exactly as shown above**.

If the second command-line argument is omitted **(v1.1)** or if the second command-line argument is greater than or equal to the total number of words in the given file(s), the output format is as follows:

```
bash$ ./a.out xyz
Allocated initial parallel arrays of size 16.
Re-allocated parallel arrays to be size 32.
Re-allocated parallel arrays to be size 48.
Re-allocated parallel arrays to be size 64.
Re-allocated parallel arrays to be size 80.
Re-allocated parallel arrays to be size 96.
Re-allocated parallel arrays to be size 112.
All done (successfully read 174 words; 104 unique words).
All words (and corresponding counts) are:
Once -- 1
when -- 1
Lion -- 5
was -- 4
...
FRIENDS -- 2
MAY -- 1
PROVE -- 1
GREAT -- 1
```

As with the previous example, match the above output format **exactly as shown above**, **(v1.1)** replacing the ellipses with all words of the given regular file(s).

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty, the homework submission server. The specific URL is on the course website.

Note that this assignment will be available on Submitty a few days before the due date. Please do not ask on Piazza when Submitty will be available, as you should perform adequate testing on your own Ubuntu platform.

That said, to make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, as discussed in class (on 9/7), output to standard output (`stdout`) is buffered. To ensure buffered output is properly flushed to a file for grading on Submitty, use `fflush()` after every set of `printf()` statements, as follows:

```
printf( ... );     /* print something out to stdout */
fflush( stdout );  /* make sure that the output is sent to a */
                   /*  redirected output file, if specified  */
```

Second, also discussed in class (on 8/31), use the `DEBUG_MODE` technique to make sure you do not submit any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of x is %d\n", x );
    printf( "the value of q is %d\n", q );
    printf( "why is my program crashing here?!" );
    fflush( stdout );
#endif
```

To compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE homework1.c
```