Computer Systems Principles

Word Frequency

Contents

Overview

This project assignment will help you understand how to work with larger C programs, use C structures, allocate and free memory with `malloc` and `free`, and use `make`. You should complete all the exercises below using a text editor of your choice. Make sure you follow the instructions exactly. The actual code you write is fairly short. However, the details are quite precise. Programming errors often result from slight differences that are hard to detect. So be careful and understand exactly what the exercises are asking you to do.

The goal of this project is to complete the implementation of a program that performs a simple glossary analysis. In particular, the program extracts *words* from a text file

and prints to the console the list of those words and their frequencies. For example, a working implementation will allow you to run the program `wordfreq` like this:

```
$ cat books/aladdin.txt | ./wordfreq
THE        :  1
ADVENTURES :  1
OF         :  1
ALADDIN    :  1
Once       :  1
upon       :  1
a          : 47
time       :  1
widow      :  5
had        : 21
an         :  6
only       :  4
son        :  6
...
```

Suggested Reading

You should do the reading for this week and the previous week. It will also be useful to read up on any of the library functions that we mention in this assignment documentation and other functions that you notice as you read the provided project code. Don't forget, you can look at the documentation for C functions directly from the command line using the `man` command like so:

```
$ man 3 <function name>
```

(The `3` requests lookup in the 'section' of the manual reserved for C library routines and system calls, as opposed to various other things.)

Part 0: Project Startup

Please download the project startup tarball. If you do not know what a "tarball" is you should read up on it. To do this, open a web browser by clicking on the left-most menu button followed by "Internet". We have installed Firefox for you to use. You should do this within your virtual machine so you can download the tarball to your virtual machine disk. You should download the tarball to your home directory. By default, chromium will download to the "Downloads" folder. You can change this by clicking on "student" in the save dialog box. Alternatively, you can download to the

"Downloads" folder and then run the following Unix command from the terminal inside your home directory:

```
$ mv Downloads/word-freq-proj-student.tgz .
```

Remember, the `.` represents your current working directory, in this case your home directory. This command will move the tarball to your home directory. You can verify this using the `ls` command. Once you have the tarball in your home directory you can execute the following command:

```
$ tar xzvf word-freq-proj-student.tgz
```

This will *unarchive* the contents of the tarball and you will see the `word-freq-proj` directory. You can then go into that directory from the terminal using `cd` (change directory):

```
$ cd word-freq-proj-student
```

Part 1: Understand The Code

We provide you with starter code for this assignment. Your first task is to **read** through each of the provided source files in detail so you understand the structure of the code. Here is a summary of the files that you will find:

- **Makefile**: the Makefile you must use to compile this assignment.
- **wordfreq.c**: one of the main files that will be compiled into an executable.
- **wordfreq-fast.c**: the other main file that will be compiled into an executable.
- **words.c**: this file contains the implementation of a function for reading "words" from standard input. You will need to implement the function in this file.
- **words.h**: this is the header file for words.c.
- **wset.c**: this file contains the implementation of a *set* data structure implemented as a linked list. There are a number of functions that you will need to implement in this file.
- **wset.h**: this is the header file for wset.c.
- **test/Makefile**: the Makefile for compiling the tests.
- **test/test-all.c**: this contains the tests.

To compile the project you need to use the following command:

```
$ make
```

This will produce several *object files* and two *executable files*. The two executable files are:

- `wordfreq`
- `wordfreq-fast`

The `wordfreq` program will read words from the standard input and generate a list of those words and their frequencies (as depicted at the beginning of this document). It uses the set data structure defined in the *wset.c* and *wset.h* files. The `wordfreq-fast` program does the same thing as the `wordfreq` program. However, it uses a smarter set *add* function to improve performance. You can run these two programs on the large text files we provide in the `books` directory with this project. Note that this can take some time (up to several minutes), especially using the non-fast version of the program. (We suggest testing on smaller files to make it easier to detect infinite loops, etc.) Here is en example of how to run the programs:

```
$ cat books/odyssey.txt | ./wordfreq-fast
```

To clean the project:

```
$ make clean
```

To run the tests you must issue the following command:

```
$ make test
```

This will compile and run the tests inside the `test` directory.

Testing

We have provided tests that you can run to see if you are on the right track toward a solution. Make sure you review the academic honesty policy on the course website.

The `test/test-all.c` file contains tests that use the check C unit testing framework. You do not need to know every detail of the check framework to use it. Each test is defined by using special macros that auto-generate test functions. If you want to know more about C macros you can read this article. Here is what a test looks like:

```
START_TEST(test_wset_new)
{
  WordSet* wset = wset_new();
  ck_assert_msg(wset != NULL,
                "The wset_new function is returning NULL!");
}
END_TEST
```

In this test you can see that it is testing to see if the value returned by `wset_new` is not equal to `NULL`. If it is, the test will fail and it will be reported when you run the public tests with `make test`:

```
$ make test
make -C test
make[1]: Entering directory '/home/student/word-freq-proj/test'
gcc -I/usr/include --std=c99 -Wall -g -c test-all.c
```

```
gcc -I/usr/include --std=c99 -Wall -g ../wset.o ../words.o test-all.o \
  -o test-all -lcheck -lsubunit -L/usr/lib/i386-linux-gnu -lrt -lm -lpthread
make[1]: Leaving directory '/home/student/word-freq-proj/test'
./test/test-all
Running suite(s): Word-Freq Assignment Tests
0%: Checks: 8, Failures: 8, Errors: 0
... list of failures ...
```

The important lines are the ones that start with `Running suite(s): Public Assignment Tests`. You will see after that some details about how the test ran (87% of the tests were successful) and a report of which tests failed (including the line number). When you run the tests for the first time you will see several failed tests - your job is to make sure that all tests pass by the time you submit the project. This is what you will see when all the tests pass:

```
$ make test
make -C test
make[1]: Entering directory '/home/student/word-freq-soln/test'
gcc -I/usr/include --std=c99 -Wall -g -c test-all.c
gcc -I/usr/include --std=c99 -Wall -g ../wset.o ../words.o test-all.o \
  -o test-all -lcheck -lsubunit -L/usr/lib/i386-linux-gnu -lrt -lm -lpthread
make[1]: Leaving directory   '/home/student/word-freq-soln/test'
./test/public-test
Running suite(s): Public Assignment Tests
100%: Checks: 8, Failures: 0, Errors: 0
```

You are welcomed and encouraged to introduce additional tests. To add a new test, you should copy one of the existing tests and add your own. After you add your own test you need to add it to the test suite. To do this you simply extend the `tester_suite` function to include an additional `tcase_add_test(tc_inc, <your test name>);`

Part 2: Reading Words

Your first task is to implement the function:

- `char *words_next_word()`

This function is in the `words.c` file. Its job is to read in characters from standard input until it encounters a "non-word" character. The acceptable word characters are the characters A-Z, a-z, and 0-9. You should use the function in the `ctype.h` header file called `isalnum` to identify acceptable word characters. Use `man` to find out more about `isalnum` and other related functions. You should read the comments we have provided in `words.c` to understand better how you should implement this function. Note that `getchar` returns an `int` so that `EOF` can be `-1`, different from any actual character value. Thus you should declare any variable to hold the result of a `getchar`

call as an `int`. Once you have verified it is not `EOF`, you can store it as a `char` since you know its value will be in the range 0 to 255.

Part 3: Set Implementation

Your second task is to complete the implementation of a set data structure. We have provided a starting implementation that you can find in `wset.c`. The implementation of the set is simply a linked list of `WordNode` objects. A `WordNode` object contains a word and its frequency. This file starts with structure definitions for `WordNode` and `WordSet`. You should also note that we typedef these structs in the corresponding `wset.h` header file. After these structure definitions we have two functions for allocating and deallocating `WordNode` objects. We provide the implementation of these functions for you!

After that begins the set implementation. The first two functions, `wset_new` and `wset_free`, allocate and deallocate the WordSet objects. You need to implement these using `malloc` and `free`. You should model your implementation after the allocation functions for WordNode. Do not forget to properly initialize your allocated objects! After the allocation functions you have a function for returning the size of the set - we provide the implementation for that!

Next comes an internal function called `search`. This very important function is used by most of the remaining functions in `wset.c` to find the `WordNode` for a given word. Read the comments on `search` carefully. Notice that it returns a `WordNode **`, *not* a `WordNode *` as you might first think. Returning a pointer to the cell that points to the found node is better than returning a pointer to the WordNode because it allows for easier adding and removing of nodes. Note: we want exact string matches, that is, case-sensitive, not case-insensitive. Thus "A" and "a" will count as different words.

After `search` comes the `wset_add` and `wset_fast_add` functions. The `wset_add` function is used to add a word to the set (linked list of `WordNode` objects), adding new words at the *end* of the list. The `wset_fast_add` function does the same except that it will move `WordNode` objects to the front of the list when they are added or accessed. You must implement both of these functions. See the comments associated with them for additional hints.

After these functions you have `wset_remove` for removing words from a set, `wset_exists` for checking the existence of a word in a set, `wset_freq` for returning the frequency of a particular word in a set, and lastly `wset_print` for printing out the words and associated frequencies (we have provided the implementation for this) in a set. You must implement each of these functions to complete the implementation of the set data structure (some of them may not be used by the main programs, but we will test them nevertheless.) For further hints look at the comments associated with each of these functions. Notice that, given the `search` function, *none* of the other functions you are to implement require iterating over the list. (We will take points

off if they do.) Note that `wset_remove` is not appropriate for the unchaining required in `wset_fast_add` to pull an existing node to the head of the list. This is because `wset_remove` will free the node (that approach will also search the list again, entirely unnecessarily).

Debugging Hints

C programs involving pointers can be tricky to debug. You should use the `valgrind` tool to identify problems in your code and any memory leaks (allocating memory without freeing it). To run `valgrind` from the command line you do this:

```
$ cat books/aladdin.txt | valgrind ./wordfreq
```

This will report any invalid access to memory and any memory that had been allocated and not freed before the program terminated. If your program tries to access memory in a way it is not allowed to do you will likely see `segmentation violation (core dump)` as the only output after running your program. Make sure you use `valgrind` to help better understand where things went wrong. Here is a list of how to go about debugging your C code:

1. Use `printf` to output debugging information. Do not underestimate the usefulness of this simple method of debugging.
2. Use `valgrind` to narrow the scope of where your problem is (which function), then use the `printf` method.
3. Use `gdb` if you are really stuck and need to step through your program one line or one call at a time.

Part 4: Performance

This last part simply requires you to run both the `wordfreq` and `wordfreq-fast` programs on some of the books in the `books` directory. Some of these books are quite large and make the performance of your programs more obvious. The `wordfreq` program uses an add-at-the-end list to store words and the `wordfreq-fast` program uses the pull-to-the-front-of-the-list implementation. When you run these two programs on some of the larger books you should notice a difference in performance. To get a more exact measurement of performance you should use the following command:

```
$ time cat books/iliad.txt | ./wordfreq quiet

real    0m1.221s
user    0m1.192s
sys     0m0.024s
```

```
$ time cat books/iliad.txt | ./wordfreq-fast quiet

real    0m0.518s
user    0m0.500s
sys     0m0.012s
```

The `time` command will time the execution of the command after it. (Your times may be quite different from the sample above.) Learn more about it using `man`. The output provides the elapsed "real" time, the number of seconds the program was executed by the CPU (user), and the amount of time the system (OS) executed (sys). You should notice a difference between the normal and fast versions of the program. Of course, the times above will be different depending on your system and what you currently are running. If you implemented the fast add set functionality properly you will observe a difference using the `time` command. The `quiet` argument to the `wordfreq` programs suppresses their usual output of the words and frequencies. For a large timing test (which may take some minutes!) you can use this command (and its fast variant):

```
$ time cat books/*.txt | ./wordfreq quiet
```

For some fun, try this:

```
$ cat books/aladdin.txt | ./wordfreq | sort -k3,3nr -k1,1 | more
```

What does it do? Do `man sort` to see what those command line arguments do.

Submission Instructions

You must submit your assignment as a *tarball*. After you complete the assignment you need to run the following command from the directory of the your `word-freq-proj-student` directory.

```
$ make tar
```

This will create the file `word-freq-proj-student-submit.tgz` which you need to upload to the word-freq assignment activity in Moodle. Please submit your assignment to Moodle by the assigned due date. **Be careful to submit the submit.tgz file, not the original tgz file!** Please make sure you have followed all the instructions described in this assignment. Failure to follow these instructions precisely will likely lead to considerable point deductions and possibly failure for the assignment.