CSE 451, Introduction to Operating Systems, Autumn 2013

Project 0: C Programming warm-up

Information

Out: Wednesday September 25
Due: Friday October 4 electronically @ 11:59 p.m.
Submission: submit your hash.c and queue.c to the Catalyst DropBox when you have finished.

Make sure to start early!

Background

Despite incredible advances in programming languages over the last 30 years, most serious systems programming is still done in C.

Why is this? Because C gives the programmer more control and power over the code's execution than do other, higher-level languages like Java or even C++. Also, C typically has less runtime overhead than higher-level languages, which can translate into increased performance. Suppose you have a function that takes an integer and returns a double. In a strongly typed language, all you can do with this function is call it while passing an integer and treat the result as a double. Of course, you can do this in C. But you can also call it with no parameters, call it with 5 parameters, take the result and store it in an integer. Even better, you could treat the function as an array and read each instruction as an integer if you like. Or, you could call not the first instruction in the function, but maybe the second, or the third, or ... there is a reason why C is sometimes referred to as a "high-level assembly language".

What's bad about this freedom? Bugs. Forgot a parameter? Maybe you did it on purpose. Or maybe (and probably) not. In Java, the compiler shows you your mistake. In C, the compiler is very easy to please, but when you run the program, it fails, generally in a very cryptic way - "segmentation fault (core dumped)" is a common error message indicating that something went wrong in the code.

All of this means that you need to program carefully and deliberately in C. If you do this, you can write programs that are as well structured and clear as you can in languages like Java. But, if you don't, you'll quickly have a big mess on your hands. Hard to debug. Hard to read. Hard to modify.

There are lots of good references for programming in C. The primary one is "C Programming Language (2nd Edition)" by Kernighan and Ritchie.

Assignment

You should do this assignment on a Linux machine, which will provide you not only with the compiler (gcc) but also a debugger (gdb). You can use any editor you like, but Emacs and Vim are good choices.

We recommend the use of the CSE home Linux virtual machine (http://www.cs.washington.edu/lab/homeVMs/homeVMs.shtml) for this assignment. It's handy to set up, and comes pre-configured with everything you need. (It's sufficient for all projects in this course, except for Project 1, which is done on a CSE Linux box devoted to this course and configured in an odd way.) The one gotcha is that the VM runs a 64-bit guest operating system, which requires that the host OS (the one running directly on your hardware) be 64-bit as well.

Getting the starter code

The code for this assignment is distributed via a Git repository. If you are not familiar with it, Git is a version control system developed by Linus Torvalds and others. To get the code, first make a new empty repository on attu,

Github, or wherever else you like (preferrably somewhere that keeps back ups of its data). If you are using attu, you can create an empty repository as follows:

```
snowden@attu2:~$ mkdir 451repo
snowden@attu2:~$ cd 451repo
snowden@attu2:~/451repo$ git init --bare
Initialized empty Git repository in /homes/iws/snowden/451repo/
```

Keep in mind that this is where your Git repository is stored, not the files that you will be editing. You'll notice that the directory contains some Git-related data already:

```
snowden@attu2:~/451repo$ ls
branches  config  description  HEAD  hooks  info  objects  refs
```

Next, navigate to a directory on *the machine where you are planning to complete the assignment* (this could also be attu). For the purpose of this writeup, I am using a starting directory of `~/cse451` on my personal laptop. Clone the Git repository that you created from wherever it is hosted; if you have it on attu, you would do something like:

```
# Replace this username with your own
elliott@elliott-hp:~/cse451$ git clone snowden@attu.cs.washington.edu:~/451repo
Cloning into '451repo'...
snowden@attu.cs.washington.edu's password:
warning: You appear to have cloned an empty repository.
```

Next, set up your working repository to be able to fetch from the course repository that contains the starter code. To do this, add the repository as a remote source (MAKE SURE TO CHANGE DIRECTORY TO YOUR WORKING REPOSITORY), fetch its attributes, and then merge the code in:

```
elliott@elliott-hp:~/cse451/451repo$ git remote add course \
snowden@attu.cs.washington.edu:/cse/courses/cse451/13au/451repo
elliott@elliott-hp:~/cse451/451repo$ git fetch course
From attu.cs.washington.edu:/cse/courses/cse451/13au/451repo
 * [new branch]      master     -> course/master
elliott@elliott-hp:~/cse451/451repo$ git merge course/master
# (No output)
```

And now you have the files! If you want to save incremental changes as you go (this is a very good idea), you can persist your changes by `git add`ing the files that you modified, then doing `git commit -m "message here"`, then using `git push` to push your commit back to your remote repository. Here is a sample workflow:

```
# Edit one of the provided files
elliott@elliott-hp:~/cse451/451repo$ emacs project0-skeleton/hash.c
# After making a change and saving it, add the file to the
# list of files to be committed
elliott@elliott-hp:~/cse451/451repo$ git add project0-skeleton/hash.c
# Now make a new commit
elliott@elliott-hp:~/cse451/451repo$ git commit -m "Change _____ in hash.c"
[master 345d6fc] Change _____ in hash.c
 1 file changed, 2 insertions(+)
# Finally, push the commit to the remote repository
elliott@elliott-hp:~/tmp/451repo$ git push origin master
snowden@attu.cs.washington.edu's password:
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (18/18), 42.64 KiB, done.
Total 18 (delta 3), reused 0 (delta 0)
To snowden@attu.cs.washington.edu:451repo
 * [new branch]      master -> master
```

At any time, you can use `git status` to see which files have been modified, which are going to be part of the next commit, and some other information. After pushing this first time, you can just use `git push` (without the extra flags) to push to your repository in the future. If you lose the contents of your working repository (as in the files you have been editing) or want to check what your remote repository contains, you can use the `git clone` command from above to clone your repository again in a different directory. For further reading, take a look at the Git tutorial provided by kernel.org.

Part 1: The Basic Queue

The queue is one of the most important data structures you'll be dealing with in this class. Consequently, it's a good one to start working with early.

For this part of the assignment, we will be providing you with a complete interface and mostly complete implementation for a queue. Starting with this, you will:

1. Find and fix two critical bugs in the implementation in queue.c. (We've put these bugs in after getting the code working). You will need to extend the test infrastructure in queuetest.c significantly and ensure that it functions correctly. You should include comments that make it clear which problems you fixed either in queue.c or in queuetest.c.
2. Implement the two declared, but not implemented, methods: queue_reverse() and queue_sort(). Both methods must work in-place: they can't create a new queue and move or copy elements from the original queue to build the result. The time efficiency of the sorting algorithm is not important, as long as it's something reasonable (not worse than `O(n^2)`, not better than `O(n log(n))`. `queue_reverse()` should execute in `O(n)` time.

You should follow the coding style (indentation, naming, etc) that you find inside queue.c and queue.h. You should not change the interface that is already defined in queue.h.

Part 2: The Hash Table

Write the code in hash.c that implements the hash table interface defined in hash.h. The abstract hash table allows you to store and retrieve pointers to values of any kind based on keys of any type. You should not change the interface that is already defined in hash.h in the skeleton code.

Internally, you're free to use any hash table implementation you like. You learned several variations, such as linear probing or separate chaining, in your data structures course. You *must* resize the hash table once it reaches a certain size / capacity ratio (the exact value of the ratio is up to you) in order to minimize memory overhead while maximizing speed of insertions.

Testing

We have provided two small test files, queuetest.c and hashtest.c, that test the functionality of the queue and the hash table. These are very basic, limited test files that do not test the complete functionality of the data structures. You should extend these test files (or write your own) with additional test cases that test the complete data structure interfaces, as well as any corner cases you can think of. We will grade your data structures with test files that are much more comprehensive than the skeleton test files.

A good practice is to write your test code BEFORE implementing your data structures, to make sure that you understand the expected behavior of the data structure's interface; you can then write the internal data structure code and see if your test cases pass or fail.

Additionally, you should run the queuetest and hashtest programs under the valgrind tool to check for memory leaks and other memory bugs. More information about valgrind will be given in section or in a separate tutorial on the class website, but it is very easy to use; just run:

```
valgrind ./queuetest
valgrind ./hashtest 10  # Or some other number of insertions
```

Style

Please make use of the provided clint.py script for checking that your code is stylistically sound. While we won't penalize in this case for poor style, it will make your code much more readable for follow the same standards as everyone else. You will be penalized for submitting files that have warnings when compiled with `gcc -Wall`, however.

Submission

Submit both hash.c and queue.c to the Catalyst DropBox when finished. Both files should compile when used with the original Makefile, hash.h, and queue.h files.