Advanced Homework 13

Due: Wednesday, December 13th, 11:59PM (Hard Deadline)

Submission Instructions

To receive credit for this assignment you will need to stop by someone's office hours, demo your running code, and answer some questions. **Make sure to check the office hour schedule as the real due date is at the last office hours before the date listed above.** This applies to assignments that need to be gone over with a TA only. **Extra credit is given for early turn-ins of advanced exercises. These details can be found on the website under the advanced homework grading policy.**

`gprof` and `gcov`

For this assignment, use the code from Question 1 of this week's homework.

`gprof` and `gcov` are slightly different *profiling* and *coverage* tools. They are deeply integrated with the `gcc` compiler. When you pass the `-p` ("profile") flag to `gcc`, the compiler will actually modify your program, causing it to record profiling information while it runs. What's weird here is that you simply run your compiled program to generate profiling information and don't use the `gprof` tool until you want to parse the output. Unlike with `perf`, where it was a run-time decision to profile, with these tools profiling is a compile-time choice.

To get started, try this:

```
$ gcc -g -p main.c -o main_with_profiling
$ ./main_with_profiling
# Notice that because of the -p flag, running the program made a gmon.out file
$ ls
gmon.out  main_with_profiling  main.c  Makefile
# Now when we run gprof, your program is not running, you're simply matching
# up the program that was run with the profile data it generated
$ gprof main_with_profiling gmon.out | less
```

Do the % time values at the top look familiar?

Adding the `-p` flag caused `gcc` to emit everything that `gprof` needed. The first thing to figure out is the additional flags needed for gcc to generate `gcov` data and how to use `gcov` to parse the generated file.

One interesting thing that gcov can show you is "taken/not taken" statistics for branches (how often was this if statement true).

Modify the code from the homework to include some branches. Include some that are always taken, some that are never taken, and some that are taken probabilistically with 25, 50, and 75% chance (`man 3 rand` will be helpful). The probability that each branch is taken should be read from a file or command-line argument, i.e. you must be able to change the probability that a branch will be taken *without* recompiling your code. Your branches should run many, many times.

Show the output of gcov's branch metrics and show that your probabilities are working as expected.

Show how changing the probabilities leads to different results.

Gcc supports something called *Profile Guided Optimization*. The idea here is that gcc can use the result of a profiling run to guide it compilation. For example, if a branch is likely to be taken, it can provide hints to the CPU that that branch will be taken. In some cases, it will even emit code that removes the branch, assumes it was taken, and checks at the end whether it was right, undo-ing the code that was run if it was wrong.

Show how to add coverage information to compilation and optimization.

Can you find a program that runs faster with coverage information? Any previous projects that do searches are good candidates. Anything long-running will be better. If you can't find something that runs noticeably faster, can you at least show an example where the compiled output (`objdump -Sd a.out | less`) changes? Can you give a *rough* explanation of what changed (you *do not* need to understand x86 assembly! An educated guess is fine). objdump gives a lot of information, you only need to look at specific functions though – use search.