# CSCE 3600: Systems Programming

## Minor Assignment 8 – Compiler Execution & Construction

### Due: 11:59 PM on Monday, April 25, 2016

1. **(50 Points)** Please read the accompanying PDF document **Linux time Description** prior to continuing with this problem. Then, consider the given CPU bound (i.e., calculation driven as opposed to user input) program compOpt.c that has several loops and typically takes a few seconds to run on our CSE Linux machines. Note that this program requires the –lm flag when compiling to use the math library.
   1. Compile as directed below and run the program using the time command. For example, time ./a.out. Record the “real”, “user”, and “sys” times for each of these program executions.
      1. Compile using gcc only (with the –lm flag).
         1. There is no optimization here. This is slower and larger than the other optimization options.
         2. This code has the slowest real and user time.
         3. The real time is greater than the user time. The process is I/O bound and execution on multiple cores wouldn’t help.
         4. There is no sys time.

real 0m2.540s

user 0m2.528s

sys 0m0.000s

* + 1. Compile using gcc –Os (with the –lm flag)**.**
       1. This optimizes for code size. The compiler doesn’t care about the execution performance.
       2. This has the 3rd fastest real and user time.
       3. The real time is greater than the user time. The process is I/O bound and execution on multiple cores wouldn’t help.
       4. There is no sys time.

real 0m0.073s

user 0m0.064s

sys 0m0.000s

* + 1. Compile using gcc –O2 (with the –lm flag).
       1. This turns on most optimizations, except for the ones that could drastically increase the code size. CPU stalling is minimized because instructions are moved around to avoid data hazards and dependencies. This code is small and fast.
       2. This code has the fastest real and user time.
       3. The real time is about equal to the user time. The process is CPU bound and takes no advantage of parallel execution.
       4. There is no sys time.

real 0m0.024s

user 0m0.020s

sys 0m0.000s

* + 1. Compile using gcc –O3 (with the –lm flag).
       1. Turn on even more optimizations that could drastically increase the code size. The performance is improved, however.
       2. This code has the 2nd fastest real and user time.
       3. The real time is greater than the user time. The process is I/O bound and execution on multiple cores wouldn’t help.
       4. This has the slowest sys time, because all the other sys times were 0.

real 0m0.035s

user 0m0.028s

sys 0m0.004s

Using your knowledge on compiler optimization (or refer to the man pages for gcc or your lecture notes) to compare and contrast the timing results for each of the different optimizations (i.e., why were some parts – real, user, sys – faster or slower for the various optimizations?). Note that you may want to look at the code and perhaps which specific optimizations are enabled for each one.

* 1. Now, modify the compOpt.c source code to add the following statement just prior to one of the for loops:

#pragma omp parallel for

Then compile using gcc –fopenmp (with the –lm flag) and run using the time command. The –fopenmp flag allows the program to run as many threads as available cores/CPUs. Did it make a difference? What about compiling with this option, plus the optimizing flags in the first part of this problem? Compare and contrast perhaps why the timing results were different (and why).

1. gcc –fopenmp –lm
2. This makes the time the 2nd slowest out of all tried so far.
3. Real time is less than user time. The process is CPU bound and takes advantage of parallel execution on multiple cores/CPUs.

real 0m1.545s

user 0m4.452s

sys 0m0.012s

ii. gcc –Os –fopenmp –lm

1. This goes faster than with no optimizations.
2. Real time is less then user time. The process is CPU bound and takes advantage of parallel execution on multiple cores/CPUs.

real 0m0.903s

user 0m2.552s

sys 0m0.008s

iii. gcc –O2 –fopenmp –lm

1. This is even faster real and user time. The system time increases.
2. Real time is less than user time. The process is CPU bound and takes advantage of parallel execution on multiple cores/CPUs.

real 0m0.859s

user 0m2.488s

sys 0m0.012s

iv. gcc –O3 –fopenmp –lm

1. This has the fastest real and user time, and the fastest system time using the parallel for loop.
2. Real time is less than user time. The process is CPU bound and takes advantage of parallel execution on multiple cores/CPUs.
3. There is no sys time.

real 0m0.849s

user 0m2.472s

sys 0m0.000s

1. **(15 Points)** For each of the following actions, state whether it happens at:

* *compiler construction time* (i.e., when developing the compiler)
* *compile time* (i.e., when compiling or linking a program)
* *run-time* (i.e.. when executing the code generated by the compiler)

For example, *Evaluating an arithmetic expression* will mostly occur at *run-time* (but in reality can occur at *compile time* if all operands have known constant values). Consider only normal scenarios for these actions.

* 1. Compute address of a local variable.

Run time

* 1. Elimination of a common sub-expression.

Compile time

* 1. Choose in which area of memory (static data, heap, or stack) a particular variable will be stored.

Compiler construction time

1. **(35 Points)** On Linux, flex is the fast lexical analyzer generator. The file scanner.lex contains the description of tokens to generate a simple and very basic scanner using flex. To generate the scanner, use the following command:

flex scanner.lex

You should notice that flex created the file lex.yy.c in your current directory. Now compile this file into an executable program as follows:

gcc –o scanner lex.yy.c

Note that you must have the zcalc.h header file in the same directory for the compile to be successful. Now you can run the scanner executable and see what it does by typing in text and checking if it is recognized as a token. You may want to view the scanner.lex file to see how tokens are specified. The scanner will output the type of symbol that it recognized for the input that you typed. You may use Ctrl-D to terminate the scanner.

* 1. Run the scanner to enter operators, words, parentheses, etc., but make sure you get results for at least all of the specified tokens and take a screen shot of your output (note that you may do so using the script command with no options, then your screen shot is saved in a file called typescript).
  2. Find at least one common operator or symbol that you use when writing your own programs that is not handled by the scanner and describe what happens. Now modify the scanner.lex file to add support for your operator or symbol and take a screen shot of your output with the now supported operator or symbol. Also include your modified scanner.lex file.
     1. The divide symbol is not recognized by the scanner. When a symbol is not recognized, nothing is printed out as a “Lex” or a “Token”.
     2. I added the divide symbol into the scanner and tested it with typescript.

**REQUIREMENTS:**

* This is an individual programming assignment that must be the sole work of the individual student.

**SUBMISSION:**

* You will electronically submit your solutions and whatever additional files (such as your screen shots or modified source code file) to the **Minor Assignment 8** dropbox in Blackboard by the due date.