Programming Assignment Multi-Threading and Debugging 2

Due Date: **Friday, May 27 @ 11:59 pm**

PAMT2 Assignment Overview

The purpose of this mini-assignment is to continue your introduction to parallel programming and multi-threading. It will cover aspects of implementation, as well as the benefits and tradeoffs involved. You will again be provided with skeleton code, and this time you will fill in the functionality where necessary to determine whether a large number is prime or not. This will be done both in parallel over many threads simultaneously, and sequentially in one. The program will also record and display the execution time of the calculations using both methods, allowing you to see the relative performance of each.

NOTE: Once again, due to the number of classes that use ieng9 and the CPU-intensive nature of this assignment, **we are developing and running the multi-threaded programming assignment on the workstations or ieng6**!

Debug2 Assignment Overview

You will also get to practice your debugging skills again. We have written a program that will start a command line tic-tac-toe game. We think it's close to working, but we didn't have time to debug it. It's now your job to help us track down the bugs and fix them so we can get it to work. You should **reference your notes on static vs. global** to help you finish this assignment. **The debugging exercise must be completed on ieng9.**

To summarize:

1. **PAMT2** must be developed, executed, and turned in with your cs30x account on the workstations in the lab (preferred) or ieng6.ucsd.edu remotely.
2. **Debug2** must be debugged, executed, and turned in with your cs30x account on ieng9.ucsd.edu.

PAMT2 Grading

- **README: 10 points** - See PAMT2 README File section.
- **Compiling: 10 points** - Using our Makefile; no warnings. If what you turn in does not compile with the given Makefile, you will receive 0 points for this assignment. **NO EXCEPTIONS!**
- **Correctness: 40 points** - Includes both abnormal and normal output, both to the correct destination (stderr vs stdout).

Debug2 Grading

- **README: 30 points** - See Debug2 README File section.
- **Correctness: 10 points**

NOTE: If what you turn in does not compile with given Makefile, you will receive 0 points for this assignment.

PAMT2 Overview

Once again, all functions for pamt2 are written in C. Your job is to fill in the missing functionality. Note that all development and running of pamt2 must take place on the workstations in the labs or on ieng6. Do not develop/run this programming assignment on ieng9.

Gathering Starter Files for PAMT2:

```
$ cd ~
$ cp -r ~/../public/pamt2 .
```

You should now have a directory named `pamt2`. This will contain the starter files for the project, listed below. You are responsible for filling in the code in `numOfFactors.c` and `parallel_numOfFactors.c`.

Starter files provided:

`parallel_numOfFactors.c`	`numOfFactors.c`	`main.c`
`pamt2.h`	`Makefile`	

PAMT2 Sample Output

This program takes a list of long long integers (64 bit numbers, up to about 9 quintillion in decimal) and displays information on whether each number is prime and how long it took to determine that.

Below are a few examples (bold indicates user input):

```
[cs30xzzz@ieng6]:pamt2$ ./pamt2 8081 246813607

Calculating number of factors for 8081 with sequential numOfFactors()
 [be patient with large values]
8081 is prime
Number of factors: 2
Completed in 0.000175 sec

Calculating number of factors for 8081 with parallel numOfFactors()
 [be less patient]
Num of threads = 8
8081 is prime
Number of factors: 2
Completed in 0.000389 sec

*** numOfFactors Speed-up: 0.449167 ***


Calculating number of factors for 246813607 with sequential numOfFactors()
 [be patient with large values]
246813607 is prime
Number of factors: 2
Completed in 6.821402 sec

Calculating number of factors for 246813607 with parallel numOfFactors()
 [be less patient]
Num of threads = 8
246813607 is prime
Number of factors: 2
Completed in 1.554082 sec
```

```
*** numOfFactors Speed-up: 4.389346 ***

--------------
[cs30xzzz@ieng6]:pamt2$ ./pamt2 8589934592

Calculating number of factors for 8589934592 with sequential numOfFactors()
 [be patient with large values]
8589934592 is not prime
Number of factors: 34
Completed in 304.404836 sec

Calculating number of factors for 8589934592 with parallel numOfFactors()
 [be less patient]
Num of threads = 8
8589934592 is not prime
Number of factors: 34
Completed in 67.111630 sec

*** numOfFactors Speed-up: 4.535799 ***
--------------
[cs30xzzz@ieng6]:pamt2$ ./pamt2 790738119649411319

Skipping 790738119649411319
num should be less than or equal to 8589934592

--------------
[cs30xzzz@ieng6]:pamt2$ ./pamt2 420420

Calculating number of factors for 420420 with sequential numOfFactors()
 [be patient with large values]
420420 is not prime
Number of factors: 144
Completed in 0.010200 sec

Calculating number of factors for 420420 with parallel numOfFactors()
 [be less patient]
Num of threads = 8
420420 is not prime
Number of factors: 144
Completed in 0.005737 sec

*** numOfFactors Speed-up: 1.777885 ***

--------------
```

Note that the first group of results for each number is for sequential mode (checking each possible factor of the number in sequence in a single thread), while the second group of results is for parallel mode.

Like last time, the speedup is the factor by which parallel computation of the result is faster than sequential. A speedup of 1 would mean that both methods are about equally fast, while a speedup of less than 1 would indicate that the sequential mode is better (the overhead of forking and joining threads can dominate small problems), and a speedup of greater than 1 would indicate that the parallel mode is better. Your output might not be exactly the same as the sample output, but it should look reasonable and justifiable.

C routines:

```
int main( int argc, char * argv[] );
long long numOfFactors( long long n, long long lo, long long hi );
long long parallel_numOfFactors( long long n, long long lo, long long hi );
```

PAMT2 Modules

---

main.c

```
int main( int argc, char * argv[] );
```

The driver of the program. For each number entered on the command line, `main()` calls `gettimeofday()` to get the start time, and then calls `numOfFactors()` to run the calculations sequentially. Upon return, `gettimeofday()` is called again to get the end time. The time difference is then calculated and the results of the sequential run are printed to `stdout`. Everything up to this point is given to you in the starter code.

It then does the same thing with `parallel_numOfFactors()`.

Note: There is a comment marked `TODO` in `main.c` that shows you where you should make additions/changes. You do not need to (and in fact should not) make changes to `main.c` anywhere except at this point. You should remove the `return 0` once `parallel_numOfFactors()` has been implemented.

---

numOfFactors.c

```
long long numOfFactors( long long n, long long lo, long long hi );
```

To implement `numOfFactors()`, use the algorithm provided in the header.

---

parallel_numOfFactors.c

```
long long parallel_numOfFactors( long long n, long long lo, long long hi );
```

To implement `parallel_numOfFactors()`, copy over `numOfFactors.c`. Then using what you learned in pamt1, create a OMP pragma to parallelize the loop you created for the third part of the algorithm. Remember to comment out the `return 0` on line 78 in `pamt2.c` when you are finished.

PAMT2 README File

Along with your source code, you will be turning in a README (use all caps and no file extension for example, the file is README and not README.txt) file with every assignment. Use vi/vim to edit this file!
Your README file for this and all assignments should contain:

- High level description of what your program does.
- How to compile it (be more specific than: just typing “make”--i.e., what directory should you be in?, where should the source files be?, etc.).
- How to run it (give an example).

- An example of normal output and where that normal output goes (stdout or a file or ???).
- An example of abnormal/error output and where that error output goes (stderr usually).
- How you tested your program (what test values you used to test normal and error states) showing your tests covered all parts of your code (test coverage). (Be more specific than diff'ing your output with the solution output--i.e., what are some specific test cases you tried?, what different types of cases did you test?, etc.)
- Anything else that you would want/need to communicate with someone who has not read the assignment write-up but may want to compile and run your program.
- Answers to questions (if there are any).

Questions to Answer in the README

1. Approximately what value gets the parallel version speedup of 1.0 or more, 2.0 or more, on one of the ieng6 servers and on one of the B240 workstations?
2. Which would you rather have, larger speed-up or faster elapsed time?

Debugging Exercise 2 Overview

The purpose of this program is to play tic-tac-toe from the command line. The `main()` function is written in C, and it will run the game using various helper functions.

We provide all of the code to you, but the code doesn't quite compile or work as it should. It is up to you to track down the bugs and fix them. There are a total of **6 bugs** in the source code.

You are required to record ALL of the bugs we placed and the solution for each bug. See the section on Debug2 README File for more details.

You **DO NOT** need to define any of the strings as constants but you **MUST** add file and function headers to the files.

Gathering Starter Files for Debug2:
For debug2 (the debugging exercise), you will develop on ieng9 as you do for most programming assignments. However, we will provide you with the buggy code. Simply go to your home directory and copy the whole folder to your home directory:

```
$ cd ~
$ cp -r ~/../public/debug2 .
```

This will provide you with all of the buggy source code. All you have to do is fix the code and detail in a README file exactly what fixes you had to make to get this code to work properly - line numbers, what the line looked before and after your fix, etc. Be sure to include a short explanation of how you debugged each problem.

Debug2 Sample Output

The program takes no arguments from the command line:

```
$ ./debug2
```

Below are a few examples (bold indicates user input):

If the user didn't pass in the correct format, an error message is printed.

```
[cs30xzzz@ieng9]:debug2$ ./debug2
-------
| | | |
-------
| | | |
-------
| | | |
-------
Player 1, please enter your move: (1,2

Missing right paren
Player 1, please enter your move: ^D
```

As you can see, the program will continue running until the game is over. If player 1 or player 2 wins, or there is a tie, the game will end.

```
[cs30xzzz@ieng9]:debug2$ ./debug2
-------
| | | |
-------
| | | |
-------
| | | |
-------
Player 1, please enter your move: (0,0)

-------
|X| | |
-------
| | | |
-------
| | | |
-------
Player 2, please enter your move: (1,0)

-------
|X| | |
-------
|O| | |
-------
| | | |
-------
Player 1, please enter your move: (0,1)

-------
|X|X| |
-------
|O| | |
-------
```

```
| | | |
-------
Player 2, please enter your move: (1,1)

-------
|X|X| |
-------
|O|O| |
-------
| | | |
-------
Player 1, please enter your move: (0,2)

-------
|X|X|X|
-------
|O|O| |
-------
| | | |
-------
Player 1 Won!
```

If the game is completed without a winner, a tie message will be printed.

```
[cs30xzzz@ieng9]:debug2$ ./debug2
-------
| | | |
-------
| | | |
-------
| | | |
-------
Player 1, please enter your move: (0,0)

-------
|X| | |
-------
| | | |
-------
| | | |
-------
Player 2, please enter your move: (0,1)

-------
|X|O| |
-------
| | | |
-------
| | | |
```

```
-------
Player 1, please enter your move: (1,0)

-------
|X|O| |
-------
|X| | |
-------
| | | |
-------
Player 2, please enter your move: (1,1)

-------
|X|O| |
-------
|X|O| |
-------
| | | |
-------
Player 1, please enter your move: (2,1)

-------
|X|O| |
-------
|X|O| |
-------
| |X| |
-------
Player 2, please enter your move: (2,0)

-------
|X|O| |
-------
|X|O| |
-------
|O|X| |
-------
Player 1, please enter your move: (0,2)

-------
|X|O|X|
-------
|X|O| |
-------
|O|X| |
-------
Player 2, please enter your move: (1,2)

-------
```

```
|X|O|X|
-------
|X|O|O|
-------
|O|X| |
-------
Player 1, please enter your move: (2,2)

-------
|X|O|X|
-------
|X|O|O|
-------
|O|X|X|
-------
It's a tie!
```

C routines:

```
int main( int argc, char * argv[] );
long parsePlay( long board[][3], char * input, long * xIndex, long * yIndex );
long checkWin( long xIndex, long yIndex, long turn );
void printBoard( long board[][3] );
```

Assembly routines:

```
long checkMove( long board[][3], long xIndex, long yIndex );
```

Debug2 C Modules

---

debug2.c

```
int main( int argc, char * argv[] );
```

The main driver of this program. It combines all of the game's logic as well as controlling the players and the placement of the pieces. It also contains the main game loop.

---

parsePlay.c

```
long parsePlay( long board[][3], char * input, long * xIndex, long * yIndex );
```

This function parses the user input from `stdin`. The input must be of the form (X,Y). If it does not have this form, an error message is printed and 0 is returned. This function uses `strchr()` and `strtol()` to parse the X and Y arguments one at a time. It also replaces the comma and right paren with null characters to allow `strtol()` to parse X and Y. This function returns 0 on failure, and 1 on a successful parsing of the argument.

---

checkWin.c

```
long checkWin( long xIndex, long yIndex, long turn );
```

This function checks whether or not a player has won the game. It keeps track of the number of times that each player has played in each row, column, and diagonal. It increments the count if player 1 plays, and decrements

the count if player 2 plays. If any row, column, or diagonal adds up to 3 or -3, then the player has won. This function returns 1 if a player has won, and 0 otherwise.

---

printBoard.c

```
void printBoard( long board[][3] );
```

This function simply prints a formatted board to `stdout`. 'X' represents player 1, 'O' represents player 2, and ' ' represents empty.

Debug2 Assembly Modules

---

checkMove.s

```
long checkMove( long board[][3], long xIndex, long yIndex );
```

This function checks if `board[xIndex][yIndex]` is a valid position. It first checks if the indices are in range, and then checks if the position has been filled yet. This function returns 0 if false, 1 if true.

Debug2 README File

For the debugging assignments only, you do not have to include the usual high level description, how tested, etc. in your README file. You will, however, have to list the compilation error you encountered and the fix you made to correct the error (include the compilation error, the file name, the line number, and the new code).

You will also have to solve several logic errors. Again, for each problem, describe the error and describe your solution, including the file name, line number, and code for your fix.

As a guideline, there should be 1 compilation error and 5 functionality problems. Make sure you locate all of them! (Note: When we say there is 1 compilation error, we mean that there is one fix you'll have to make, not that there is one error printed to the screen). Reference your lecture notes on static vs. global to help you with this assignment. You will not lose points for extra errors.

Turn-in Instructions

Complete Turnin - due Friday night, May 27 @ 11:59 pm

Once you have checked your output, compiled, executed your code, and finished your README files, you are ready to turn in your assignments.

How to Turn in an Assignment

Use the following scripts to submit your assignments before the due date as follows:

PAMT2: In your cs30x account on the lab workstations of **ieng6**. You will not be able to turn in pamt2 on ieng9.

```
$ cse30turnin pamt2
```

Debug2: In your cs30x account on **ieng9**. You will not be able to turn-in debug2 on ieng6.

```
$ cse30turnin debug2
```

Style Requirements

You will be graded on style for all the programming assignments. The requirements are listed below. Read carefully, and if any of them need clarification do not hesitate to ask.

- Use reasonable comments to make your code clear and readable.
- Use file headers and function header blocks to describe the purpose of your programs and functions. Sample file/function headers are provided with PA0.
- Explicitly comment all the various registers that you use in your assembly code.
- In the assembly routines, you will have to give high level comments for the synthetic instructions, specifying what the instruction does.
- You should test your program to take care of invalid inputs like non-integers, strings, no inputs, etc. This is very important. Points will be taken off if your code doesn't handle exceptional cases of inputs.
- Use reasonable variable names.
- Error output goes to stderr. Normal output goes to stdout.
- Use #defines and assembly constants to make your code as general as possible.
- Use a local header file to hold common #defines, function prototypes, type definitions, etc., but not variable definitions.
- Judicious use of blank spaces around logical chunks of code makes your code easier to read and debug.
- Keep all lines less than 80 characters, split long lines if necessary.
- Use 2-4 spaces for each level of indenting in your C source code (do not use tab). Be consistent. Make sure all levels of indenting line up with the other lines at that level of indenting.
- Do use tabs in your Assembly source code.
- Always recompile and execute your program right before turning it in just in case you commented out some code by mistake.
- Do #include only the header files that you need and nothing more.
- Always macro guard your header files (#ifndef ... #endif).
- Never have hard-coded magic numbers (any number other than -1, 0, or 1 is a magic number). This means we shouldn't see magic constants sitting in your code. Use a #define if you must instead.