Assignment 1

CSI2110—01 Programming Practice, Fall 2011

Due date: 11:59pm, October 5 (Wednesday) 2011.

Introduction

The purpose of the first assignment is to get acquainted (1) with the Linux programming environment, and (2) with parallel programming using POSIX threads. We will apply the concepts from the lectures on "Parallelism" and on "POSIX threads". The objectives of this assignment are to

- practice creation and joining of POSIX threads,
- experience parallelism and task indeterminism, i.e., observe several threads executing in parallel, and
- implement task and data parallelism for a simple example.

This assignment consists of two examples. You are asked to implement those examples until the due date. You must hand in your programs as explained in Section "Deliverables and Submission" below.

Advice

Please allow enough time *to get acquainted with the course environment (remote login, the edit-compile-run cycle under Linux, aso)!*

This assignment is probably also your first encounter with the C programming language. The lecture slides contain code examples that you can use for this assignment. We will discuss selected C topics (e.g., pointers and memory allocation) in the tutorials as well. Recommended resources related to C programming are

- The classic textbook "The C Programming Language", second edition, by the inventors of C, Brian Kernighan and Dennis Ritchie (often referred to as the K&R book).
- Notes for the K&R book: from http://www.eskimo.com/~scs/cclass/notes/top.html
- A tutorial on pointers in C: http://pw2.netcom.com/~tjensen/ptr/pointers.htm
- The C Programming FAQs from http://c-faq.com/
- The online version of "The C Book" by Mike Banahan http://publications.gbdirect.co.uk/c_book/

Don't take chances when developing parallel programs. Programming errors can be very subtle and show up only at a later stage, e.g., when your program is being marked. If you are not sure about something, try to resolve the issue by consulting the course materials, discuss with your colleagues or post in our YSCEC discussion board. Don't just go by trial-and-error and rely on the "information" obtained from trial runs!

Always develop step-by-step: break down your programming task into a sequence of steps, and do one step at a time. When you have implemented one step, test it to ensure that it works correctly. Then move on to the next step. The idea behind this approach is that you can locate the cause of a problem quickly. If the program behaves unexpectedly, then it is most likely because of a problem introduced in the most recent step. Once you have changed 10 things at once in your code, it is much harder to track down the cause of a problem.

Environment

You can solve this assignment on any computer that has Linux, GNU GCC and the Pthreads librarary installed. A convenient way will be to do the assignment on our server (`elc1.cs.yonsei.ac.kr).` Our tutorial on the course environment contains all the information you need to access the server.
This assignment will be tested and marked on the server; it is therefore highly recommended that you test your programs on the server before submitting (see also the submission instructions below).

Example1: Parallel Hello World

In this example we focus on thread creation and observe how the operating system schedules threads. Your task is to write an extended version of the parallel hello world program from Lecture 'POSIX Threads', slide #8. Your program should provide the following functionality:

- The main task uses pthread_create to create three threads T1, T2 and T3. You should provide each thread with its *own* thread function (Therefore there is no need to pass an argument to a thread, use NULL instead). Furthermore, use NULL as the thread attribute argument.
- After thread creation, the main task uses pthread_join() to wait for the termination of thread T1. Then it uses again pthread_join() to wait for termination of thread T2. Finally it joins with thread T3 and terminates.
- Each of the thread functions of threads T1, T2 and T3 shall contain a loop where the name of the thread is output N times (you will use several runs of this example, for N=10, N=100 and N=100000). It is therefore recommended to use a macro

  ```
  #define MAXITER 10
  ```

  at the beginning of your program and use it as follows in your thread functions:

  ```
  for (i=0; i< MAXITER; i++) ....
  ```

  That way it is sufficient to change only the value for **`MAXITER`** and recompile in order to run this example for N=10, 100 and 100000.

- For thread T1, the **`printf()`** inside the **`for`** loop shall be

```
printf("T1");   // no newline after "T1"!
```

- Likewise for threads T2 and T3.
- After output of the above text N times, the thread shall terminate (no need for a **`sleep()`** statement!).

(1.) Compile and run your program on the server for N=10. What can you conclude from the output? Is this output as you would have expected it?

(2.) Recompile and run your program on the server for N=100. What can you conclude from the output?

(3.) Recompile and run your program on the server for N=100000. What can you conclude from the output? If you run the program several times, does the output change between runs or stay the same?

Note: you may want to pipe the output of your program to the less command as follows:

```
[you@elc1 ~] ./example1 | less
```

That way you can scroll the output with keys page up/down.

Example 2: Integer Array Computations

In this example, we compute minimum, maximum and average from a set of integer numbers. File /opt/pp/assignment1/data1.txt on the elc1 server contains 1000000 integer numbers, one number per line. Values range from 0 to 1000. Copy the above file to your homedirectory. Your program should read the numbers from the file and store them in an integer array. Consult the manpages on fopen() and fscanf() on how to open a file and read integers ("man fscanf()", …). Store the integers in a global array where your threads can access them (similar to the example in Slide #16 from the Lecture on ``POSIX Threads''). Consult the manpages on fclose() on how to close a file.

Part 1: Calculate max, min and average of the integer array sequentially. After execution, you should print the max, min and average values on the screen. See the Linux manual pages for fprintf ("man fprintf") on how to output integers and doubles in C. The format of your output must be as follows (each value on a separate line!):

```
min: <value>
max: <value>
average: <value>
```

The average value must have 3 digits after the decimal point, and no exponent (e.g., 234.444, and not 2.34.444e+02). So you should use the f conversion specifier with printf. **It is important that you follow these conventions exactly, because your program will be marked automatically.**

Our test-script will look for output in the specified format. Non-conformance to the above rules will make your program appear as being wrong!

Part 2: Calculate max, min and average in parallel, using *task parallelism*. You should use three threads for calculations – thread 1 for calculating the minimum value, thread 2 for calculating the maximum value, and thread 3 for calculating the average (again no need to pass arguments to threads). After all three threads have finished their computations, they must store the results in global variables that you provide. You should then output the results on the screen. Output must be done in the *main thread*, after threads 1-3 have terminated (**use pthread_join to synchronize the main thread with treads 1-3**).

Part 3: Extend Part2 to use data-parallelism. For each computation (min, max, average), create an additional thread. The two threads share the work: one thread works on the lower half of the array, one thread works on the upper half of the array. You are required to pass an argument to a thread function to distinguish between lower and upper half of the array! You need to create 6 threads in total (two for min, two for max, two for the average). You should provide global variables where threads can store the result of their computation, e.g.,

```
        int T1min, T2min, T3max T4max;
        double T5sum, T6sum;
```

The first thread computes the minimum of the lower half of the array (T1min), the second thread computes the minimum of the upper half of the array (T2min). Likewise for the maximum. For the average, we compute the sum of the lower half of the array in T5sum, and the sum of the upper half of the array in T6sum.
The main thread must join the 6 threads and do the output of the results. For the minimum, the main thread must select the minimum from T1min and T2min. For the average, the main thread must add T5sum and T6sum and divide by the total number of integers.

As outlined above, it is recommended to use a step-by-step approach for your assignments. For this example, a sensible work-breakdown could be as follows.
Step 1: read in integers from a file and store them in your integer array. Then output the integers from your array to a temporary file. You can then use the UNIX diff command to check that you read in the integers correctly. Create a file with only a few numbers if you have to debug.
Step 2: create the sequential version and test it on a file with a few numbers. Check the correctness of your computations. Run your program on the provided large file.
Step 3: Create the task-parallel version, compare the output with the output from Step2.
Step 4: Create the task-and-data-parallel version, compare the output with Step2.

Marking

For Example 1, we will check that you created threads and that the threads produce output (the exact order of the threads' output, relative to each other, will be irrelevant). For Example 2, we will use a different file of 1000000 numbers. We will compare the output of your Example 2 with the output of our solution. With Assignment 1 we do not evaluate program performance (we will study performance of parallel programs in subsequent lectures). **Testing will be done on the elc1 server, so please make sure that your programs compile and execute there. Programs which fail to compile, crash, or produce no or the wrong output will receive 0 points.**

Please note that all assignments are *individual* assignments. You are most welcome to discuss your ideas with your colleagues, but 'code-sharing' or 'code-reuse' is not allowed. Pls. see also the university policy on plagiarism from the course introduction slides.

Deliverables and Submission

You are required to hand in Example 1 and Example 2 (Part 1, 2 and 3) each as a separate file. The files should be named example1.c, example2_p1.c, exampe2_p2.c and example2_p3.c. Your submission should also contain a README.txt file where you explain briefly (1 paragraph) your observations with the output from example 1.

Summing up, you need to hand in 5 files:

- example1.c
- example2_p1.c
- example2_p2.c
- example2_p3.c
- README.txt

To submit these files, please log in on elc1.cs.yonsei.ac.kr, create a submission directory, e.g., ~/myassignment1, and copy the above files to this directory (the directory should contain only those 5 files and nothing else!).

After copying the files to your submission directory, you must cd to the submission directory and type 'submit' (return) to submit your Assignment 1. The submit command will collect all files from the directory and copy them to our repository.

```
[you@elc1 ~] cd ~/myassignment1
[you@elc1 myassignment1] submit
```

Please note: There is no submission of Assignment 1 unless you issue the 'submit' command! Failing to issue this command will result in 0 points. The timestamp of your submission will be recorded by the 'submit' command. Please note also that if you re-submit, then a previous submission will be overwritten and a new timestamp will be recorded.

Good luck! :)