Lab Exercises for Posix Thread Workshop
2016 Ontario HPC Summer School
University of Ottawa, August 10, 2016

General Considerations:

All lab sample codes are in `~/Posix/C`. There are sub-directories that contain sample code For instance, the `hello` example resides in `~/Posix/C/hello`.

Most example directories contain the following files (using *name* as an example):

- The source code `name.c`.
- A `makefile` with instructions for compilation and a sample run.
- In some cases, an input file `name.in` with sample input for the code.
- Sometimes there is a `solution` directory as well.

The code is supposed to be examined and in some cases modified. The `makefile`'s are there as a reminder on how to compile and execute the code. The idea is, of course, to issue the commands in the `makefile` *manually* as a practice.

Compilations:

C compiler: `gcc`
Pre-processor: `cpp`
Optimization flag: `-O3`
Renaming executable: `-o name.exe`
Compilation only: `-c`
Pre-processor flag for keyword KEY: `-DKEY`
Use math library: `-lm`
Use pthread library: `-lpthread`

For instance:

C code `name.c` using Posix threads, compiled into `name.exe`, using math lib:

```
gcc -o name.exe -O3 name.c -lm -lpthread
```

Displaying/Editing code:

Editors on the system:

- `vi` Standard Unix editor, requires practice, only useful to people familiar with Unix
- `nano` simple text editor, almost self-explanatory, ideal for new users

Executing Posix code from command line:

The number of threads is usually an argument to the executable

```
./name.exe 2
```

for 2 threads.

Individual Labs:

Note: No separate sessions, we will do the exercises during the lectures.

1. **hello**: Two versions of “Hello World” using Posix threads. One (`hello_s`) uses different threads to print out the two words, the other (`hello_l`) prints a separate message for each thread. Run with number of threads as argument.
2. **rootsum**: Sum of square roots example from the course. Should scale reasonably well up to the hardware limitations. Run with two arguments: number of threads and maximum integer. Try increasing thread number exponentially (1,2,4…) and see when it “bails out”.
3. **erf**: This is for those who feel they can write their own already. You are supposed to make a Posix Thread program that computes and error function by “rectangle rule” numerical integration. The easiest way is to copy the “rootsum” example and modify it.
4. **once:** Super-boring example demonstrating the use of `pthread_once` “blocker” variables to protect initialization of a mutex.
5. **minimum:** Code finding the minimum of a function over an interval. If executed with a large number of points and many threads, the unprotected version occasionally results in errors due to a race condition. Use “watch” command to try. Fix the race condition by inserting a mutex. A solution is in the “solution” directory.
6. **dot:** Forming the dot product of two vectors. Since the total is summed to a global variable, the latter must be protected through a lock. Also an example for the impact of Amdahl’s law. Compare external (total) and internal (partial) timing.
7. **cond**: Demonstrates the use of a condition variable. Three threads count up a variable, and a fourth one “butts in” once a threshold is reached.
8. **tsd:** Demonstrates thread specific data.
9. **destruct:** Demonstrates usage of “destructors” to clean up after thread specific data.
10. **barrier:** Uses a barrier to separate two parts of code. Multiple threads are delayed by different times to “spread out”. Barrier syncs them before moving on.
11. **slaves**: Basic skeleton of a “all slaves” model. This can be used to work through any “bag of tasks” workload by modifying the function DoJob().
12. **mandel**: Specific implementation of “All slaves model” to compute the Mandelbrot set. Jobs are lines in the complex plane with a fixed imaginary part. There are two versions, “smart” and “stupid”. Examine the code, look at their scaling and determine why they got these names.
13. **mixed**: This is an example on how to combine Posix threads with MPI. Note that the MPI parallelism and the multithreading parts have to be kept well separated. Follow the instructor about execution.