CPSC 213: Assignment 8

Due: Monday, March 19, 2012 at 6pm.

Late assignments are accepted until Wednesday, March 21 at 6pm with a 25% penalty per day (or fraction of a day) past the due date. This rule is strictly applied and there are no exceptions.

Goal

In this assignment you will gain experience with programming with threads in Java and will examine the use and implementation of user-level threads in C.

Notes on FunWithThreads

In the first part of the assignment you do some thread programming in Java. You are provided with a file, "`FunWithThreads.java`", which includes a simplified model of a disk with an operation called `read`. Unlike disk hardware, this `read` operation is synchronous. Like disk hardware it takes around 10ms (1/100 of a second) to complete a disk read. During this time the `read` method waits by sleeping (leaving the CPU idle while it waits).

The program is parameterized by command-line arguments that are listed at the beginning of the class. If you run the class without arguments (or providing incorrect argument syntax), it prints a description of the arguments. You use these arguments to select from one of three different implementations (*sequential*, *threaded*, and *executor*). The first of these is provided for you. Your first goal is to implement the other two. Note that your implementations must store the result of calls to `read` in the `val` array like the *sequential* version does.

The *threaded* version should create and start a thread, using Java's standard `Thread` interface (as described in class), for every `read` and then join with these threads to record the `read` result in the `val` array. And so, for example, this version will create 1000 threads to perform 1000 reads.

The *executor* version should use a fixed thread pool with the number of threads in the pool specified by the argument "`threadCount`", which is set by a command-line argument. The statement that creates the executor is provided. Also provided is an `ArrayList` to store the `future` values returned by executor `submit` method. This collection is provided because you might be tempted to create an array to store the `future` values for each `read` call, but `future`s are generics and Java does not allow you to create an array of generics. Collections of generics are fine, however.

To test your implementation, use the "`-verbose`" argument and small number of reads (e.g., "`-count 20`") and examine the output. The output comes as a list of number pairs. The first number is a sequence number assigned by the disk read when it completes and the second is a sequence number you assign when you request the read (it is the argument to the `read` method). The request number should be monotonically increasing, but you may see some small variation between the request and completion numbers and even some repeated or out-of-order *completed* sequence numbers; this is okay. As an aside you might ask yourself why this is happening (we will talk about this problem next week).

Once you are certain your two implementations work, your next task is to compare the runtime performance (i.e., running time) of these three alternatives under various setting of the `readCount` and `threadCount` parameters.

In UNIX you can time any operation by placing the word "`time`" before the command on the command line. For example, to time the *sequential* version you would type:

```
time java FunWithThreads -c 1000 -s
```

For the timing to be valid you must not specify the `verbose` option and you must perform a large number of reads (e.g., at least 1000) and run the command a number of times. Running times will vary depending on what else is running on the system (and some other factors) and so the best value to record is the minimum of several executions. Best results will be achieved running this on your laptop since you can control what else is running better there than on a department server.

The output of the `time` command is a little complicated. It shows you elapsed time and some other things (e.g., user and system time etc.) and looks something like this.

```
1.704u 1.402s 0:01.82 170.3%  0+0k 0+0io 12pf+0w
```

We are only interested in the elapsed time, which is the third number, in this case `1.82s`.

To begin, compare all three implementations performing 1000 reads (i.e., "`-count 1000`"). For the *executor* implementation, vary the size of the thread pool to find the best value to the nearest 50 threads or so (e.g., "`-e 200`" and then "`-e 250`" etc.). This does not have to be exact and there will be enough variation that an exact answer will be elusive.

Now, compare only the *threaded* and *executor* versions, varying read count (i.e., "`-count 1000`" and then "`-count 10000`", etc.) to determine when executors are faster than threads. Again, vary the size of the thread pool to get the best executor performance.

Notes on UThreads

In the second part of the assignment you will switch to C and the `uthread` user-level thread package we have discussed in class.
The `uthread` package runs on Intel x86 machines running Linux, MacOS or Cygwin. Department machines that run Linux include `remote.ugrad.cs.ubc.ca`, and all machines named `linXX.ugrad.cs.ubc.ca`, where XX is a two-digit number like '02' or '12'. To compile on Linux or Cygwin it is necessary to explicitly include the `pthread` library by adding "`-lpthread`" to the `gcc` command; this parameter is optional on MacOS.

Requirements

Here are the requirements for this week's assignment.

1. Implement the "`threaded`" and "`executor`" methods of `FunWithThreads.java` as described above.
2. Test your implementation of these two methods.
3. Compare the running time of the *sequential*, *threaded*, and *executor* implementations for a `readCount` of 1000, varying the size of the executor thread pool to optimize its performance. Carefully describe what you observe including saying which implementation is fastest/slowest etc., which thread-pool size gives the best results for executor, and what you think accounts for the performance differences you see.
4. Compare the running time of *threaded* and *executor* for larger read counts (e.g., increase by a factor of 10 each time) to determine when *threaded* is slower than *executor*, varying the size of the executor thread pool to optimize its performance. Carefully describe what you observe including the `readCount` (to the nearest 1000-5000 or so) at which *executor* begin to perform better than *threaded*, the ideal thread pool size for executor for this number of reads, and what you think accounts performance differences you see. Be sure you answer this question as best you can: why does the *thread*ed version slow down faster than the *executor* version does?
5. Compile the files `ping_pong.c` and `uthread.c`. Run the resulting program

   ```
   gcc -lpthread -o ping_pong ping_pong.c uthread.c
   ```

6. Read `uthread.c` carefully. Describe the control-flow path involved in creating and starting a new thread by listing the `uthread` procedures that execute, in order, starting with the creation of a thread and ending when the thread's start procedure begins executing.
7. Execute the `ping_pong` program and examine its output. Carefully explain this output by describing the execution of the `ping` and `pong` threads. Your explanation should be detailed and should include a description of control flow paths (i.e., procedure names executed) in the `uthread` package relevant to explain the execution of these two threads.
8. Modify the `ping_pong` procedure to add a call to `uthread_yield` in both `ping` and `pong` at the end (but inside) of the iteration loop (i.e., just after the "`j`" for loop, but inside of the "`i`" for loop). Run this modified program, examine its output and compare it to the previous output. Explain what you see by again describing the execution of the two threads in a detailed fashion including the relevant `uthread` thread control flow.
9. Modify the `ping_pong` procedure to change the argument to `uthread_init` from 1 to 2 to change its running environment from a uni-processor to a 2-processor system. Run `ping_pong` again (you should run it at least 3-5 times; they should be different from each other), compare and explain its output as you did in question 4. Carefully explain what this change did and why it produced the output it did.
10. Implement the problem from Assignment 7 using threads instead of calls to `doAsync`. Change the "`Triple`" struct to remove the "`result`" field and now have the `add` and `sub` routines return the resulting value (they will return this as an opaque pointer (i.e., `void*`). Use `uthread_join` to get this value, cast it back to its actual type, and to synchronize the three phases of the computation (the inner additions, the subtraction, and outer addition)

Material Provided

The files `FunWithThreads.java`, `uthread.h`, `uthread.c` and `ping_ping.c` are provided in the archive `code.zip`.

What to Hand In

Use the **handin** program. The assignment directory is **a8**. Please hand in exactly the following files with the specified names. Do **NOT** hand in class files, or your entire Eclipse project, or a README in formats like `.doc` or `.rtf`.

1. `FunWithThreads.java` with `threaded` and `executor` methods implemented, as specified in Requirement 1.
2. `notAsync.c` that implements the same computation as the `async` program from Assignment 7, but using the `uthread` package, as specified in Requirement 10.
3. `README.txt` that contains:
   - header with your name, student number, four-digit cs-department undergraduate id (e.g., the one that's something like a0b1)
   - statement that "I have read and complied with the collaboration policies" at http://www.ugrad.cs.ubc.ca/~cs213/winter11t2/policies.html
   - answers (i.e., comparisons of running time and what accounts for the differences) to Requirement 3 and 4.
   - description and explanation of the control-flow path involved in creating and starting a new thread, as specified in Requirement 6.
   - description and explanation of the execution (and the output) of the `ping` and `pong` threads when running the `ping_pong` program, as specified in Requirement 7.
   - description and explanation of the execution (and the output) of the `ping` and `pong` threads when running the modified `ping_pong` program, as specified in Requirement 8.
   - comparison and explanation of the outputs produced by the modified `ping_pong` program by changing the argument to `uthread_init` function, as specified in Requirement 9.