Prof. Jingke Li (FAB 120-06, li@cs.pdx.edu); Class: MW 14:00-15:50 @ UTS 308; Office Hr: M 13:00-14:00 & by appt.

Programming Assignment 1
(Due Wednesday, 4/22/15)

This assignment is to practice multi-threaded programming using the Pthreads library. You'll develop a Pthreads program and run it on the CS multi-core Linux server machine `babbage.cs.pdx.edu` using the `gcc` compiler. You may also use the CS Linux lab machines (`linuxlab.cs.pdx.edu`) to develop and test your program. To compile Pthreads programs on these machines, you need to include the `-pthread` flag with `gcc`.

For this assignment, CS415 and CS515 students will work on two different programs. This assignment carries a total of 100 points.

Setup

Download the file `assign1.zip` to your CS Linux account and unzip it. You'll see three program files, `arraysum.c`, `prodcons0.c`, `qsort.c`, and a `Makefile`.

[For All Students] Array-Sum Programs (Warm-up)

Use an editor to open the `arraysum.c` program file. Read and understand the program; and then compile and run it:

```
linux> make arraysum
linux> ./arraysum 1000 10
...
The sum of 1 to 1000 is 500500
```

- Try the program with different array sizes and number of threads. What is the sum of 1 to 12345?
- Comment out all the occurrences of Pthreads locks in the program, and re-compile it. Now, this program has the potential for race conditions. Run the program until you see an evidence of a race condition occurring. Write down the array size, the number of threads, and the evidence.

This is a warm up exercise. Just report the numbers you get from the above parts.

[For CS415 Students] Producer-Consumer Program

Use `prodcons0.c` as the starting template, and `arraysum.c` for guidance, implement a full version of producer-consumer program. Name your program `prodcons.c`. The program should have the following features:

- It assumes a fixed total number of 100 tasks, and a fixed queue size of 20 tasks.
- It takes an *optional* command-line argument, `numConsumers`, which represents the number of consumer threads. If the argument is not provided, the program use the default of one consumer thread.

  ```
  linux> ./prodcons 10    // 10 consumer threads
  linux> ./prodcons       // 1  consumer thread
  ```

- Each consumer thread starts by printing out a message of the form:

  ```
  Consumer <tid> started on <cpu>
  ```

where `<tid>` is the the consumer thread's id, and `<cpu>` is the id of the CPU the thread is running on. (There is no requirement on where threads should run, *i.e.* no need to control CPU affinity.)

- Each consumer thread prints a tracking message for each task it performs:

  ```
  Consumer <tid> performed task <taskid>
  ```

  It also keeps track of how many tasks it has performed.

- The program terminates properly after all tasks are done, with a final message showing the distribution of tasks over threads:

  ```
  [1]:11, [2]:9, [3]:5, ..., total tasks = 100
  ```

  where the number inside a bracket is a thread id, and the number outside is the number of tasks the thread performed. The total number of tasks should add up to 100.

One important part of this program is the handling of consumer thread termination. The following are two possible approaches:

- The producer creates a bogus "termination" task and add `numConsumers` copies of it to the global queue. Upon receiving such a task, a consumer thread will termination itself by breaking out its infinite loop.
- Use a global count to keep track of the completed tasks. The consumer threads all participate in updating and monitoring the global count. When the count reaches the total number of tasks, all threads terminate.

You may use either of these approaches, or a totally different approach of your own.

[For CS515 Students] Quicksort Program

Your task is to convert a sequential quicksort program to a parallel Pthreads program using a task queue.

The Sequential Version

A sequential quicksort program, `qsort.c`, is provided to you as a starter version. The program has a simple interface:

```
linux> ./qsort <N>
```

where `<N>` is an integer representing the size of the integer array to be sorted. The program starts off with command-line processing to get the value of `<N>`. It then allocates an array of size `<N>`, and initializes it with a random permutation of values from 1 through `<N>`. After that, it follows the standard quicksort algorithm:

```
void quicksort(int *array, int low, int high)
{
  if (high - low < MINSIZE) {
    bubblesort(array, low, high);
    return;
  }
  int middle = partition(array, low, high);
  if (low < middle)
    quicksort(array, low, middle-1);
  if (middle < high)
    quicksort(array, middle+1, high);
}
```

It partitions an array segment into two smaller segments, and recurses on them. When a segment is smaller than the pre-defined threshold value (`MINSIZE`, currently set at 10), it switches to use a bubble sort to finish the sorting.

The Parallel Version

Name the Pthreads program you are going to write `qsortpthd.c`. It should have the following interface:

```
linux> ./qsortpthd <N> [<numThreads>]
```

It reads in one or two command-line arguments. The first argument `<N>` represents the array size, while the second (optional) argument represents the number of threads to use. When the second argument is omitted, the program uses one thread (*i.e.* the main thread itself) as the default.

For this program, you are *required* to follow the following outline, and use as much existing code from the sequential quicksort program as possible.

```
// A global array of size N contains the integers to be sorted.
// A global task queue is initialized with the sort range [0,N-1].

int main(int argc, char **argv) {
  // read in command-line arguments, N and numThreads;

  // initialize array, queue, and other shared variables

  // create numThreads-1 worker threads, each executes a copy
  // of the worker() routine; each copy has an integer id,
  // ranging from 0 to numThreads-2.
  //
  for (long k = 0; k < numThreads-1; k++)
    pthread_create(&thread[k], NULL, (void*)worker, (void*)k);

  // the main thread also runs a copy of the worker() routine;
  // its copy has the last id, numThreads-1
  worker(numThreads-1);

  // the main thread waits for worker threads to join back
  for (long k = 0; k < numThreads-1; k++)
    pthread_join(thread[k], NULL);

  // verify the result
  verify_array(array, N);
}

void worker(long wid) {
  while (<termination condition> is not met) {
    task = remove_task();
    quicksort(array, task->low, task->high);
  }
}
```

Implementation Details

- **The Task Queue** — The task queue representation and routines in `prodcons0.c` can be copied over and used. However, two changes are needed. (1) Instead of a single integer value, a task should contain

a pair of integer values, representing the low and high indexes of a segment of the global array. (2) There is no need to use queue size limit for this program.

- **The quicksort() Routine** — This routine is similar in structure to the sequential version in `qsort.c`. However, instead of recursing on the two smaller array segments, it places the first segment unto the task queue, and recurses only on the second one.
- **Synchronization** — Since the task queue is a shared resource, synchronization is needed. It is needed for two reasons: (1) Adding and removing tasks from the queue needs to be serialized to avoid race conditions; (2) When the queue is (temporarily) empty, a waiting/waking-up mechanism needs to be used.
- **Tracking Messages** — Have each worker thread print out a tracking message

  ```
  Worker <wid> started on <cpu>
  ```

  For debugging purposes, you may print out additional information to verify that each worker is indeed sorting for some ranges of the array.
- **Termination Condition** — By default, each thread will keep looking for new tasks to work on. How should the program terminate? The task queue being empty is a necessary condition, but not sufficient, since new tasks may still be added to it. Think about the two approaches discussed above in the Producer-Consumer section, and see which one fits this program better.
- **Result Verification** — Like in the sequential version, the `main()` routine should call `verify_array(array, N);` at the end to verify the sorting result.

[For All Students] Program Submission

Write a short (one-page) summary covering your experience with this assignment: what was easy, what was hard, and what lessons you've learned. Also include the numbers you got from the Warm-up section. Make a zip file containing your program and your write-up, and submit it through the Dropbox on the D2L site.