Prof. Jingke Li (FAB 120-06, li@cs.pdx.edu); Class: MW 2:00-3:50pm @ FAB 40-07; Office Hr: MW 1-2pm & by appt.

Assignment 1: Programming with Pthreads
(Due Wednesday, 4/20/16)

This assignment is to practice multi-threaded programming using the Pthreads library. You'll develop a Pthreads program, compile and run it on the CS Linux system (`linuxlab.cs.pdx.edu`). For this assignment, CS415 and CS515 students will work on two different programs. This assignment carries a total of 10 points.

Download the file `assign1.zip` to your CS Linux account and unzip it. You'll see several program files: `task-queue.h`, `task-queue.c`, and `qsort.c`, plus a `Makefile`.

Task Queue Representation and Supporting Routines

Your program will be task queue based. The task queue representation is shown below:

```
typedef struct task_ task_t;
struct task_ {
  int low;
  int high;
  task_t *next;
};

struct queue_ {
  task_t *head;   // head pointer
  task_t *tail;   // tail pointer
  int limit;      // queue capacity (0 means unlimited)
  int length;           // number of tasks in queue
};
```

Each task holds a pair of integer values and a pointer to next task. A queue has two pointers and two integer fields.

There are four task queue routines:

```
extern task_t *create_task(int low, int high);
extern queue_t *init_queue(int limit);
extern int add_task(queue_t *queue, task_t *task);
extern task_t *remove_task(queue_t *queue);
```

The task queue implementation is available in the two included files, `task-queue.h` and `task-queue.c`. To use the code in your program, add an include line:

```
#include "task-queue.h"
```

[CS415 Students] Producer-Consumer Program

Implement a producer-consumer program; name it `prodcons-pthd.c`. You may use code samples from this week's lecture as a starting template. The program should have the following features:

- It takes an *optional* command-line argument, `numCons`, which represents the number of consumer threads. If this argument is not provided, the program defaults its value to 1.

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

- The producer generates 100 total tasks. Each task is identified by an unique integer `i`, whose value is between 1 and 100. The value is stored in both the `low` and `high` fields of a task record.
- The task queue has a fixed capacity of 20 tasks.
- 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>
  ```

  where `<taskid>` is the value of task's `low` field. 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 has performed. The total number of tasks should add up to 100.

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

- The producer creates a bogus "termination" task and add `numCons` 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.

[CS515 Students] Quicksort Program

Your task is to convert a sequential quicksort program to a parallel Pthreads program using a task queue. The provided sequential program, `qsort.c`, 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 your Pthreads quicksort program `qsort-pthd.c`. It should have the following interface:

```
linux> ./qsort-pthd <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** — Use the provided task queue implementation. There is no need to set a limit on the queue's capacity 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. Specifically, (1) when adding and removing tasks from the queue, the operations need to be serialized to avoid race conditions; (2) when the queue is (temporarily) empty, a waiting/signaling mechanism is needed to coordinate the work.
- **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.

[All Students] Extra Challenges

If you are interested in extra challenges, you can work on both programs. For CS415 students, completing both programs will earn you an extra credit up to 5 points. For CS515 students, completing both programs will earn you an extra credit up to 2 points.

[All Students] Program Submission

Write a short (one-page) summary covering your experience with this assignment: what issues you encountered, how you resolved them, and what lessons you've learned. Make a zip file containing your program and your write-up, and submit it through the Dropbox on the D2L site.