CS 407/591 – Adv. Linux Programming – Fall 2017

Lab #4
(Due: 10/30/17)

In Lab #4, we are going to implement a *thread pool* that can be added to the server from Lab #3, to easily make that server process clients using concurrent threads. (Adapting the Lab #3 server to use the thread pool will be the subject of Lab #5).

We will also consider how to provide the thread pool to your server program as a *library*. This will require investigating what must go into header files to be able to use a library's functions in other programs, how to create libraries, and how to get personal (non-system) libraries to be linked with for compilation and execution.

Finally, the assignment will require you to make use of *structs* and *functional arguments* (e.g., *callbacks*) in your code, to make certain that everyone understands these C elements.

Thread pools are a popular mechanism for making programs *concurrent*. A thread pool consists of two main components: a *task queue* and a pool/set of *worker threads*. When the program utilizing the thread pool has tasks it needs done that can be carried out concurrently, it adds those tasks to the task queue. When *idle*, a worker thread will take the next task from the queue and carry it out. Concurrency is achieved because multiple worker threads can be concurrently carrying out separate tasks.

The thread pool approach has several advantages over the alternative of having a program explicitly create new threads to handle each new task, particularly when a non-concurrent version of the program exists:

- The cost of repeatedly creating and destroying Pthreads is eliminated.
- The number of Pthreads being used can be tuned to the CPU/core capacity of the machine, reducing the number/cost of *context switches* (among Pthreads).
- Only minimal modifications may be require to add concurrent execution to an existing program (e.g., simply add task enqueue calls wherever concurrently executable tasks are identified).
- Few if any *synchronization* syscalls will need to be added (synchronization will primarily occur in thread pool code).

Program Submission:
For this lab, you are to submit several files:

1. `tpool.c` – your thread pool *source code*
2. `tpool.h` – *header file* for using your thread pool
3. `tpool.a` – *static library* containing your thread pool implementation
4. `tpool.so` – a *dynamic library* containing your thread pool implementation
5. `tpool-test.c` – a program that tests your thread pool implementation

Because you are submitting several files, you are to submit them as *gzipped tar archive*, in a single file named `lab4.tgz`, to the Lab #4 Dropbox on the course D2L website.

The following command will tar up a directory:

```
tar czf lab4.tgz directory_containing_required_files
```

The `directory_containing_required_files` should be your name, formatted analagously to: `Norman_Carver`

Improperly packaged/submitted files will not be accepted! Note also, that your *libraries must be compiled for 64-bit Linux systems!* 32-bit libraries will not be able to be tested, so will not be accepted.

Make sure you have *only the desired files in the directory* when you create the tar archive!

Thread Pools:
Our thread pool implementation will be required only to meet the specific requirements of our server. In particular, "tasks" will be required to be represented only as `int`'s, representing file descriptors ready to be processed (read/written from/to, etc.). This will simplify the task queue and task execution. The worker threads will then always use a single function to process each FD/task. That function will be specified when the thread pool is created. More general thread pool implementations will allow each task to be represented by a function ("callback") and its argument(s). If you wish to develop that sort of more general thread pool, you are free to do so (you will have to then modify the below specs as appropriate).

The (external) interface to your thread pool is to consist of just two functions:

- `int tpool_init(void (*process_task)(int))`
  - This function is initialize the thread pool, including creating the pool's Pthreads.
  - It is to be called once by the program using the thread pool, prior to generating tasks.
  - It is to take a function as an argument: the function that will be used by the worker threads to process each task (which will be represented by a FD `int`).
  - Since we are going to create only a single thread pool, this function should initialize the thread pool object in a *single fixed global variable.*
  - Return indicates success/failure: 1 means success/true, 0 means failure/false.
- `int tpool_add_task(int newtask)`
  - This function is to be called to add a new task to the thread pool's task queue.
  - It must take exactly one parameter: the newtask, an `int` representing an FD.
  - Return indicates success/failure: 1 means success/true, 0 means failure/false.
  - Note that if you use a *bounded* (fixed size) task queue, this call may *block* if the queue becomes full.

Since this is all that programs using your thread pool need to know about, this is all the information that needs to go into the header file `tpool.h`.

There are several thread pool implementations you can easily find online for use with C programs (from Oracle, Github, and libcprops). While it may be instructive to look at some of these, because they are all more general than what is required for our server, they are more complicated than what we require. (So don't get freaked out by their complexity!) In addition, there is no guarantee that the people who wrote them are experts. For example, I noticed that the Oracle version uses a single *mutex* with multiple *condition variables.* This is unusual, since it can result in the mutex being locked for a fairly significant amount of time (e.g., tests plus creating a new Pthread). Generally, one will want a separate mutex for each condition variable, to achieve finer-grained locking. Takeaway: don't blindly follow the design of one of these examples!

And of course: every line of code you submit must have originated with you, so absolutely do not copy code from these online thread pools!!!

Structs:
A thread pool consists of a number of values/objects that must be accessible to `tpool_add_task()` and all the Pthreads of the thread pool. The standard approach for maintain such a set of information is by defining a C *struct* that represents all the required info, and then creating an *instance* of this struct to represent the thread pool *object*.

Because the thread pool struct type will be required by most of your thread pool functions, it must have *global scope*—i.e., it must be a "global type." This means that you will define it near the top of the source code file, outside of any functions. Global types will generally need to be defined before function prototypes and global variable definitions, because the type may be used in these entities. You may want to instead define the type(s) in the required header file.

Some of the online thread pool functions include thread pool object *arguments*. This allows multiple thread pool instances to be in use simultaneously. As we need only a single thread pool, we can simply use a *global variable* to hold the single thread pool struct instance, and the thread pool functions can access the thread pool via that global. (Thus, our thread pool functions do not require thread pool object parameters.)

The thread pool `struct` definition must include fields for all info/elements that is required for the thread pool. The key items are: the task queue components, mutexes and condition variables, and the function `process_task()`.

Structs are covered in Lecture #12 of the instructors C Basics slides. Be clear that the type created by defining a `tpool` struct is the two word name "`struct tpool`" so it is common to use a `typedef` to give structs a single word type name, such as `tpool_t` or `tpool_type`.

Circular Queues:
Your thread pool will have to maintain a *queue* of tasks, with tasks being frequently added and removed, asynchronously. Remember that the defining characteristic of a queue is its FIFO nature. This is what we want, since tasks should be assigned to worker threads in the same order that they are generated.

The standard data structure for efficient queues is a *circular queue* (or *circular buffer/array*). Circular queues are standard topics in data structures courses, so will be found in any data structures book or online. Be aware that there is more than one standard approach for implementing circular queues—in particular, for determining whether a circular queue is full.

The standard operations on a queue data structure are named as *enqueue* (add new item) and *dequeue* (remove next item). You will need to implement functions for these operations. These functions are to be *internal* to the thread pool—i.e., they should not be accessible to functions outside of the thread pool (such as the server). C does not include encapsulation techniques like Java, etc., but we can limit the scope of functions to the same file (*compilation unit*), but including a `static` declaration with the function definition:

```
  static int enqueue_task(...)
```

It is *your choice* whether to implement a *bounded* (*fixed size*) circular queue or an *unbounded* (*expandable*) circular queue. While the logic for enqueue with an unbounded/expandable circular queue is more complex, it turns out that a *bounded queue requires additional synchronization.* Generally we would think of an unbounded queue as "better," since it can adapt to the relative speeds of generating tasks (ready FDs) vs. processing them with the available worker threads. On the other hand, if the worker threads cannot keep up with the rate at which clients want processing, you could end up with the task queue expanding to the number of clients or more (i.e., at least one task per client queued up). That doesn't really make much sense, so unlimited expansion is probably unwise for a production version of our server.

If you choose to implement an *unbounded queue*, the enqueue function may need to expand the queue when called. Because of this, you will need to use *dynamic* (*heap*) memory for the queue array/buffer, so that you can `realloc()` it if expansion is required. Expanding the queue array by *doubling* it has advantages, so you are advised to do that when expansion is required. You will have to carefully think through the logic required for expanding your circular queue, in particular, think about the possible cases and how expansion must happen with each (i.e., front<back, back<front).

Please note that using a *linked list* for your unbounded task queue is not appropriate and so not allowed. Linked lists are good when lists have to have objects added/removed from "the middle" of the list. They are not efficient for *queues*, that must only expand (there is absolutely no reason to ever downsize your queue). So, gain, you are not to use a linked list, you must use circular arrays/buffers.

If you choose to implement a *bounded* queue, you should use a regular array. One issue with a bounded queue is deciding how big to make the queue. Obviously it needs to be able to hold more tasks than the number of worker threads you have, to avoid delaying worker threads from acquiring new tasks when idle. It probably makes sense to allow several tasks to be queued per worker thread, so to make it easy to parameterize, please include the following preprocessor constant:

- `TASKS_PER_THREAD`

You should set this to a reasonable value, and have the queue size be: number_of_worker_threads $*$ `TASKS_PER_THREAD`. If you are implementing an *unbounded* queue, use this value as your *initial* queue size (and then double the queue size on expansion).

You may want to implement a function to print out the contents of your queue so that when testing, you can view it (and make certain your enqueue and dequeue functions are working). Obviously, make certain testing causes your enqueue function to expand the queue!

Pthreads:
You will make significant use of Pthreads functions in this assignment. As noted in Lab #3, there are the following resources:

- instructor's Threads slides (available through the course webpage)

- textbook chapter 29 (Threads: Introduction)
- textbook chapter 30 (Threads: Thread Synchronization)
- CS306 textbook chapters on threads (e.g., Matthew-Stones chapter 12)

The key Pthreads issue for the thread pool is the need for *synchronization*. Please make sure you understand how to create and use Pthreads' *mutexes* and *condition variables*, as you will need them for this lab. Their use is covered in Lectures #3 and #4 of the instructor's Threads slides.

The task queue is a *shared resource*, and operations on it to add/remove tasks are *critical sections* that must be protected by a *mutex*. I.e., you cannot be concurrently adding/removing queue items or you may end up with a corrupted queue. So this is one instance of synchronization that will be required: *mutual exclusion* for manipulation of the thread pool's queue.

A second synchronization issue is to have the worker threads block if the task queue is empty, and awaken when new task(s) get added. This requires use of a mutex and associated condition variable. The standard pattern for using a mutex+condition variable is shown in the instructor's Threads Lecture #3.

If implementing a bounded queue for your thread pool, you will also have to implement blocking for `tpool_add_task()` if the queue is currently full. This requires another mutex and associated condition variable, with similar logic to dealing with an empty queue.

Worker Threads:
It generally makes sense for your server to have the same (or close) number of threads as there are CPUs/cores on your server machine (we will discusss NGINX material related to this). In the final version of your server, all server operations (after server initialization) will be carried out by the thread pool worker threads, and these operations will all be *nonblocking*. If you then have exactly the same number of worker Pthreads as there are (logical) CPUs on your machine, context switches should be minimized: less threads than CPUs means CPUs may go unused, while more threads than CPUs means context switches may be required to concurrently execute all the threads.

The following call will tell you how many currently active CPUs/cores the machine has:
`sysconf(_SC_NPROCESSORS_ONLN)`

Remember that when you create each worker thread with `pthread_create()`, you must specify the function that it runs. The basic worker thread function will be the same for all worker threads, and should be quite simple:

- loop forever:
  - get next task (FD `int`) or block if no tasks available
  - carry out task (using `process_task()`)

Libraries:
For information on creating and using both *static and dynamic libraries*, please see Lecture #6 of the instructor's Development Tools slides, the textbooks for this class or CS306, or search the Web.

You can initially test your `tpool.c` code with your `tpool-test.c` program by simply compiling it all up with a single `gcc` command, e.g.:

```
  gcc -Wall -std=gnu99 -otest tpool-test.c tpool.c -pthread
```

Once you are sure your code is working, create a *static library* version of `tpool.c`, named `tpool.a`:

```
  gcc -c -std=gnu99 tpool.c && ar -cr libtpool.a tpool.o
```

Then, recompile and rerun your test program using that library:

```
  gcc -Wall -std=gnu99 -otesta tpool-test.c libtpool.a -pthread
```

After that, you must also create a *shared library* version of `tpool.c`, named `tpool.so`:

```
  gcc -Wall -std=gnu99 -fpic -c tpool.c && gcc -shared -olibtpool.so tpool.o
```

Then, recompile and rerun your test program using that library:

```
  gcc -Wall -std=gnu99 -otestso tpool-test.c libtpool.so -pthread
```

As with all labs in this class, all students are expected to work completely independently on this assignment!! The code that you submit is to have been written by you and you alone. Working in groups and/or submitting code written by someone else (including code from the Internet) will be considered **cheating!!**