Project 3: Airline Reservation System

CS-343 - Fall 2014

Important Dates

Out: *Thursday November 3rd, 2014.*

Due: *Wednesday November 17th, 2014 (11:59 PM CDT).*

Project Overview

For this project, you and your team are asked to help failing budget airline *AquaJet* upgrade their online reservation system. Their system currently runs a single-threaded web server, handling only one request at a time and,no surprise, has resulted in a poor user experience.

You are to work in groups of two (2) for this project. The skeleton code provided is the single threaded code for AquaJet's current seat reservation server. Your task is to update their ticketing server to handle multiple simultaneous connections through multi-threading. You will update the server from a single-threaded server to a multi-threaded server using a worker-queue and thread pool. Your implementation will be stress tested through scripts to see the number of simultaneous connection the server can handle.

Educational Objectives

This project introduces the concept of multi-threaded programming through the use of threads and synchronization.

Background

This project involves the creation of threads, managed through a thread pool and requests handled through a worker-queue. Let's review each of these items.

Threads

A thread is a basic unit of CPU utilization, each containing an ID, a program counter, a register set and a stack. Threads are a more economical way to achieve concurrent programming compared to multiple processes created through `fork()` since threads share the same memory and resources of the host process, thus incurring much lower creation and context-switch costs.

Threads are typically managed through a language level package and accessed in code through libraries. In Linux, threads are managed through the POSIX threads interface, `pthreads`. A useful overview of the Pthreads library can be found at `https://computing.llnl.gov/tutorials/pthreads/`

The Pthreads interface can be used by including `#include "pthreads.h"` in your source file, and using the `pthread_t` structure. A list of useful pthread commands is shown below.

- `pthread_create(thread,attr,start_routine,arg)`: Function which creates an executable thread, which uses the following arguments.
  - thread: An opaque, unique identifier for the new thread returned by the subroutine.
  - attr: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
  - start_routine: the C routine that the thread will execute once it is created.
  - arg: A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.
- `pthread_exit(status)`: Exits the current thread.
- `pthread_cancel(thread)`: Requests to cancel execution of the specified thread.
- `pthread_join(threadid, status)`: blocks the calling thread until the specified `threadid` thread terminates. It is one way to accomplish synchronization between threads.

Synchronization and Locks

Within concurrent programs, mutual exclusion of resources is controlled through synchronization primitives such as semaphores and mutexes. A Mutex (MUTual EXclusion) allows a single thread to maintain ownership of shared resources such as a shared counter or queue. In the Pthreads library, the mutex structure is a `pthread_mutex_t`. A list of commands to use with muxtes within the pthreads library is shown below.

- `pthread_mutex_init(mutex,attr)`: Initializes the specified mutex for use.
- `pthread_mutex_destroy(mutex)`: Destroys the specified mutex.
- `pthread_mutex_lock(mutex)`: Attempts to acquire a lock on the specified mutex. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.
- `pthread_mutex_unlock(mutex)`: Will unlock the specified mutex if called by the owning thread. An error will be returned if (a) the mutex was already unlocked or (b) the mutex is owned by another thread.

Semaphores

A semaphore is as an object with an integer value that we can manipulate with two routines (which we will call `sem_wait()` and `sem_post()` to follow the POSIX standard). Semaphores are a useful tool in the prevention of race conditions. Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores. Because the initial value of the semaphore determines its behavior, before calling any other routine to interact with the semaphor. Pseudocode for wait and post functions are shown below.

```
int sem_wait(sem_t *s) {
    wait until value of semaphore s is greater than 0
    decrement the value of semaphore s by 1
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by 1
    if there are 1 or more threads waiting, wake 1
}
```

Thread Pools

While creating a separate thread for concurrent execution is much faster than creating a separate process, there is still a performance penalty incurred during thread creation. Also, it may be beneficial to control the maximum number of threads run by your program. Too many active threads could exhaust system CPU time or memory resources.

One way to solve these issues is to manage a pool of existing threads – a *thread pool*. To work with a thread pool one creates a number of threads when the program begins, and have them sit and wait for work to be assigned to them. Once a thread completes its work, it returns to the pool. If there are no available threads the incoming work waits until one becomes available, typically in a work queue or other data structure.

One common way to manage a thread pool is through a *worker queue*, a thread-safe queue in which work requests are kept and dispatched to available threads as they finish their current work. For this project, each request will be queued in a worker queue, with the information for each request (connection, request string) stored in a structure and queued in the worker queue.

Creating a thread pool using pthreads involves the use of condition variables and signals within each thread. A list of signalling functions within pthreads is shown below.

- `pthread_cond_init(condition, attr)`: Condition variables must be declared with type `pthread_cond_t`, and must be initialized with the `pthread_cond_init()` routing before they can be used. The ID of the created condition variable is returned to the calling thread through the condition parameter. This method permits setting condition variable object attributes, attr.
- `pthread_cond_destroy(condition)`: This method destroys a thread condition variable.
- `pthread_cond_wait(condition, mutex)`: This method blocks the calling thread until the specified condition is signalled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, mutex will be automatically locked for use by the thread. The programmer is then responsible for unlocking mutex when the thread is finished with it.
- `pthread_cond_signal(condition)`: This method is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for `pthread_cond_wait()` routine to complete.

Project Specification

Web Front-end

The website for AquaJet is extremely simplistic and consists of only 2 pages, the *Seat Selection* page and the *Seat Confirmation* page. The Seat Selection page shows a diagram of seats and their corresponding states: *Available*, *Pending* or *Occupied*. The Seat Confirmation page shows the previously selected seat (now in the pending state) and offers two choices for the user to either confirm the seat reservation, thereby placing the seat into the occupied state, or cancel the reservation and placing the seat back in the available state.

Webservice Calls

The server functions are accessed through a webservice with the four procedures shown below. To access the webservice calls, simply make a request to your server with the specified call appended. For example if your server is running on `localhost` at port 8888, the call for listing seats would be `http://localhost:8888/list_seats`.

1. `/list_seats?user=USERID`: This methods returns a listing of seats and their current states expressed as the letters **A** available, **P** pending, or **O** occupied. Each seat pair is represented as a seat number and state separated by a space; consecutive seats are separated by a comma.
2. `/view_seat?user=USERID&seat=SEAT_NUM`: This is called when a user wants to reserve a seat but has not yet confirmed it. This call places the seat into the `pending` state within the server, and redirects the user to the *Seat Confirmation* page. `SEAT_NUM` refers to the seat number selected.
3. `/confirm?user=USERID&seat=SEAT_NUM`: This is called from the *Seat Confirmation* page to confirm a seat selection. This can only be called when the same user has placed the seat into `pending` state, and subsequently places the seat into the `occupied` state.
4. `/cancel?user=USERID&seat=SEAT_NUM`: This is called from the *Seat Confirmation* page to cancel a pending seat reservation. This can only be called when the same user has placed the seat into the `pending` state, and returns the seat to the `available` state.

Thread Pool

Each team will implement a Thread Pool to manage and dispatch incoming requests. As each request is handled by the server, it is placed into a queue. Each item in the queue contains the function pointer to the appropriate handler function and an argument structure containing the connection and any other relevant information.

When a thread becomes available for work, it checks to see there is any requests in the queue. If there is, if pops off the item and runs the function. If there is not, the thread blocks on a condition variable, waiting to be notified by the handler to begin work.

Standby List

As an additional wrinkle in the threadpool, you must also implement a *standby list* for passengers within the reservation system. Seat requests made when the plane is at full capacity are placed into a standby list. This list gets first priority whenever a seat reservation is cancelled. Practically what this means is that a reservation

is cancelled within the system, any requests in the standby list that can be satisfied should be run before any pending in the thread pool queue.

The stand by list is limited (defined by `STANDBY_SIZE` in threadpool.c), and thus represents a bounded buffer. You are to implement your own semaphore (found in semaphore.c) to control access to this list. If the standby list is full, then requests are returned unsuccessful.

For Extra Points

There are two ways to get extra points in this project. You can implement priority scheduling within the thread pool.

Priority Scheduling *Extra credit: 15 points*
AquaJet wants to add new passenger classes in order to attract higher paying customers. It wishes to add *first class* and *business class* to its existing *coach* class. Part of their plan is to give these new classes additional service priority throughout the airline, including within their online services. With this in mind, the airline wants to allow priority scheduling in their web server, having higher priority customers serviced first when reserving seats online, specifically that all requests in the queue of priority 1 are serviced before those of 2 or 3, and so on.

You can implement priority scheduling within the thread pool instead of the FIFO scheduling used during the project. To implement priority scheduling, modify the server to accept an additional query string parameter `priority=PRIORITY` where `PRIORITY` is a number between 1 and 3, with first class as 1, business class as 2 and coach as 3. When selecting the request to run next, instead of choosing the front of the queue, you will choose highest priority request, and then the earliest request.

Suggested Plan of Attack

- Begin to add a thread for each incoming request. Call `pthread_create()` on passing in the appropriate function pointer for the request as well an any appropriate arguments. Remember that only one argument is able to be passed into `pthread_create()`, therefore if multiple arguments need to be passed, you might have to create a new struct, and that all arguments must be cast as `(void *)`.
- Add mutual exclusion to the state data structure using mutexes. There are multiple ways to lock a resource. While you can lock on the entire data structure, keep in mind that this can cause a performance bottleneck since only a single resource can access it at a time. Another option with better performance may be to have a mutex for each seat, especially since the state of each seat does not depend on another.
- Create a thread pool with a fixed number of threads. For the thread pool you are only able to create at most `MAX_THREADS`, defined in `threadpool.h`. A few hints: (1) your thread pool should have a work queue where incoming requests are enqueued and this queue also needs to be synchronized (2) when each thread is started in the beginning, it should be passed a function which loops through the thread either pulling work from the queue and running it (if applicable) or waiting to be notified from a condition variable using `pthread_cond_wait()`.

Evaluation, deliverables, and hand-in instructions

Your project will be graded based on its ability to run correctly on the virtual machine distributed at the beginning of the term without modifications. The project is graded out of 100 points.

We will deduct points for the following reasons:

- 100 points!: it doesn't compile when we type make
- up to 10 points: no documentation (DOC file) or poorly self-documented/commented code
- up to 30 points: incorrect thread implementation
- up to 30 points: incorrect resource exclusion
- 2 points per compiler warning
- 1 point per memory leak warning
- 5 points per definitive memory leak

If you hand in your project late, we will deduct 10% from your final score per day or portion thereof. We will not accept submissions that are more than three days late.

The test-cases will (1) load test your multi-threaded server and see how many simultaneous connections it can handle and (2) run a scripted series of web service calls to test for concurrent ordering and resource mutual exclusion. The resulting assignment of seats will be tested against our reference implementation for accuracy.

The deliverable for this project should include:

1. Source code.
2. Evaluation report (DOC file) which describes your architecture for your thread pool, seat state management, resource mutual exclusion and any other design decisions you made during the project. Please make sure these files are included in the handin.
3. If you attempted to pass the extra credit test cases, please make a note of it in your man page.

To hand in your project:

- Change the working directory to the project directory.
- Set your team name in Makefile: replace 'whoami' with "yourNetid1+yourNetid2" for the TEAM variable.
- Invoke `make test-reg`, which builds the deliverable tar.gz and runs the test cases.
- **If and only if the test cases terminate, you may submit the handin**.
- Submission will be done through a dedicated webpage (check the project section of the class web site for the URL).
- After running the handin procedure, if you discover a mistake and want to submit a revised copy, follow the same procedure and resubmit. The submission page keeps track of all your submissions and considers the last one as your final submission.
- A few minutes after submitting your handin on the submission site, you will receive an email confirming your submission, with the output of the testsuite running on our submission server. If you haven't received a confirmation within an hour, contact the TA to verify that your submission was received.

Good Luck!