ECGR 6181/8181 - Project
A Priority Based Pre-Threaded Image Processing Server

The goal of the project is to develop an image processing server that interacts with clients through a pool of worker threads using the producer consumer model. The server consists of a main thread and a set of worker threads with the main thread running at a higher priority than the worker threads. The main thread repeatedly accepts connection requests from clients and places the resulting connected descriptors in a bounded buffer. Each worker thread repeatedly removes a descriptor from the buffer, services the client, and then waits for the next descriptor. All the threads are scheduled with the FIFO policy. The image server applies image processing operations on the image input by the client. These image processing operations are derived from the open source OpenCV library.

For this project, you can use the following C/C++ source code included in the project zip file.

1. Multi-threaded producer-consumer: An array is used to implement a bounded buffer. Producer and consumer threads write and read data from this buffer synchronizing with locks and condition variables using the Pthread API

2. Real time scheduling and priorities in Linux: Demonstrates how to assign real time priority and scheduling policy to threads on Linux. Note that this code needs root permission to run (sudo in Ubuntu).

3. Delay: Spins in a loop for delay seconds. Useful to emulate the execution of long-running threads.

The project has the following milestones

1. . Install OpenCV and follow the tutorial for converting a color image to greyscale. `http://docs.opencv.org/3.0-last-rst/doc/tutorials/introduction/table_of_content_introduction/table_of_content_introduction.html`

   `http://docs.opencv.org/3.0-last-rst/doc/tutorials/introduction/load_save_image/load_save_image.html#load-save-image`

2. Using the echo client and server from the Systems book as a template, develop the image processing server and client. The client inputs the user-input color image to the server. The server listens to connection requests at any unused port. The client connects to the server on the loop back IP address. The server should process the image converting to greyscale and send it back to the client. The client displays both the color and greyscale image. Note that the client and server are different processes running on the same machine.

3. Write a threaded version of the image processing server. On every connection request, the server spawns a new thread to process the request. Note that each connection requires a separate connection descriptor (connfd) to prevent races. Use malloc to dynamically allocate memory for the individual connection descriptors.

4. Write a pre-threaded version of the image processing server. Using a thread for each connection request is expensive due to the overheads involved in thread creation and destruction. A better approach is to have a pre-threaded server where a fixed number of worker threads process the connection requests. The main thread (master) that accepts connection requests puts the connection descriptors in a bounded buffer and the individual worker threads remove requests from the bounded buffer before processing.

5. Extend the pre-threaded image processing server so that the manager thread has higher priority than the worker threads.

6. Bonus 1: Host the server on a free cloud service such as available from Amazon and Google.

7. Bonus 2: I will strongly consider giving a grade pull up if on borderline!): Implement a RESTful Image processing API, based the Tiny webserver from Assignment 15. For inspiration on the API and how clients access the image processing service, see the following GitHub project where the server is a node.js, and client uses curl - `https://github.com/rickydunlop/node-image-processing-api`

Demo: Set the number of worker threads to the one less than the number of cores on your system. Set the buffer size to twice the number of threads. Insert a delay of 30 seconds for the worker threads. Now send a series of different client requests using multiple clients. Since the manager is running at a higher priority and the worker threads are slow to process the requests (due to the delay), the manager will preempt the workers whenever there is a request from the client, resulting in manager finding the connection descriptor buffer full. Print appropriate messages from the manager and worker threads to demonstrate that the manager thread is indeed running at a higher priority.

Please organize the different parts of your code in separate C/C++ files and use the gnu make utility in compiling your code. Use of a version control system such as Git to manage your project is highly recommended (`https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control`). You could also use GitHub as well.