CIS*3210
Assignment 2
Deadline: Tuesday, October 22, 9:00am
Weight: 10%

Description

Expand your Assignment 1 here to add threading and some extra functionality. To keep things simple, you will keep using TCP sockets. You will create a "upload server", whose job will be to receive files from various clients and save them to disk. It will also provide the user with a minimal UI.

This assignment can be done **in teams of two**. You are welcome to use any of the code examples that I have spotted for this course.

The server will be threaded. The main thread will listen for connections. When a new client connects, the main thread will start a "transfer" thread and pass the client address info to the thread function.

A transfer thread will receive a file (with a file name) from the client, and will save it to disk on the server side. Once the file has been received and saved, the transfer thread exits.

The server will run a simple command-line user menu in a separate UI thread. It will provide the user with the following options:
- Display active transfers. This shows the names of the files being downloaded.
- Shut down the server - which tells the main thread to terminate the program.

The client functionality will not change much from Assignment 1, but we will impose a simple protocol to ensure that everything works.

Protocol:
The client must send the server the following information:
- File size - 8-byte int
- Chunk size - 8-byte int. The client can decide what an appropriate chunk size is.
- File name - 21 chars (bytes): 20-character string+string terminator).
- The file contents

Bonus: smart exit (10%)

If the user selects the shut down option, give them a choice:
- Hard shut down, which works as before - the UI server tells the main server to exit, the main server terminates immediately.
- Soft shut down: the server stops accepting new connections and checks if there are any active connections
  - If there are none, the server exits.
  - If connections are in progress, the server asks the user if they should be terminated or if the user wants for them to be completed. If the user wants to exit immediately, exit the server. Otherwise, the main thread will wait for the transfers to be completed using `pthread_join` before exiting.

Implementation

Telling the main server to quit

The UI thread can set a boolean flag, which the main thread will read. If it is false, it continues to accept connections. If it is true, the main thread exists. This variable must be protected by a mutex. You don't have to use flags, however, and are welcome to use other options as well.

File name collisions

What happens if two clients connect at the same time and want to save the file with the same name? You will need to decide how to handle these situations, and describe - and justify - your design decisions. Include this information in the README file.

Sharing data between threads

The display functionality a race condition. It requires you to share the information about the active transfers between the UI thread and the transfer threads.

Since the active transfers list is dynamically growing/shrinking, a linked data structure is appropriate here. I suggest that you recycle code from my threading examples, which provide you with a simple queue implemented as a linked list. It will be protected by a mutex and a condition variable. The list is modified by transfer threads, and is read by the UI thread and main thread.

The synchronization would work as follows:

- When a transfer starts, a transfer thread adds its transfer record to the list
- When a transfer ends, a worker thread removes its transfer record to the list
- When the user asks to display active transfers, the UI thread iterates through the list and displays active transfer information
- When the user asks to exit, UI thread sets the exit flag and terminates. The main thread reads the exit flag and terminates the whole program.

Unless your files are reasonably large (multi-megabyte), transfers will be nearly instantaneous, and it will be hard to see multiple transfers in progress. To make it easier to see the UI in action, add sleep functionality to the clients.

You don't need to sleep after every send. However, do put the client to sleep for a second after you send every 10% of the file, so each client will take at least 10 seconds to transfer its file.

Data Synchronization

Each time you **add** to the active transfers list, you:

- Lock the mutex
- Add new element to the list
- Signal the condition variable
- Unlock the mutex

Each time you a **remove** from the active transfers list, you:
- Lock the mutex
- Find the element in the list
- Remove element from the list
- Signal the condition variable
- Unlock the mutex

Each time you **read** from the list, you:
- Lock the mutex
- While the list is empty, wait on condition variable (remember, this releases the mutex, so you won't be blocking anyone)
- Iterate through the list and display contents
- Unlock the mutex

If you decide to implement the bonus option, you need to decide:
- How to share active download information with the main thread
- What to do in the UI thread when the soft shutdown option is selected and the main thread is waiting for workers to terminate.

Grading

A grading scheme will be posted before the assignment is due. You code must compile and run. An assignment that doesn't compile will receive a grade of 0 (zero). Potential deductions will involve:
- Missing, incorrect, or incomplete functionality
- Compiler warnings
- Crashes
- Unresolved race conditions

Testing

Make sure that your server can handle more than one client. Create a script (bash or python) that spawns several clients that try to write to the server simultaneously.

You will need to test your client and server in two different environments:
1. Clients and server the same machine
2. Server on one of the `linux.socs.uoguelph.ca` hosts, client(s) on a different machine (doesn't matter where). You are welcome to test this with multiple clients on multiple machines.

Required files

- All the .c and .h files for your client and server
- A Makefile that compiles the server and the client
  - `make all` creates executables server and client
  - `make clean` deletes all executables and intermediate files
- A script for spawning and running multiple clients simultaneously

- A README file that describes how to run the script(s) and provides all other instructions necessary to run your assignment

Submission

Create a zip archive with all your deliverables and submit it on Moodle. If you are working as a team, you only need one submission per team. The filename must be FirstnameLastnameA2.zip (for team submissions, these are first and last names of the submitter). Do not include any binary files in your submission.