I take the HTTPSERVER implementation from ASGN1 and make it multithreaded so it can handle several concurrent requests. Logging is also implemented: each byte of the file being GET/PUT is encoded into its appropriate hex value and stored in a log-file named at the initialization of the server.

## MULTITHREADING

I looked at several videos on the producer-consumer problem and noticed a similarity shared by the problem and this assignment. All of the clients can be thought of as one producer: the main thread which gets the client socket IDs and all of the threads can be thought of as multiple consumers. The producer and consumers share one global array. The producer adds to the array while the consumer extracts from the array. I use semaphores to ensure consistency with the dispatcher/producer and the workers/consumers. I didn't use a linked list because I felt it would be harder and also because it would use a lot of memory. I believe my method is cleaner too.

sem_t **empty**; // counting semaphore for number of empty slots in the array

sem_t **full**; // counting semaphore for number of filled slots in the array

sem_t **sem**; // locking semaphore used for critical sections

sem_t **result**; // locking semaphore used for critical sections

int **in = 0**; // initialized to 0: dispatcher inserts starting at 0

int **out = 0;** // initialized to 0: threads extract from the array starting at index 0

**empty** is initialized to BUFFERSIZE (a large number defined as a global header). Empty is the number of available slots in the shared array.

**full** is initialized to 0. Full is the number of occupied slots in the global array: initially 0

**sem** is initialized to 1. If reduced to 0, all other threads that try to enter the critical region will sleep until woken up by another thread: sem_post(&sem)

The main function has as infinite while loop which acts as the producer. The first thing it does is accepts() new connections from the queue of pending connections.

- ➔ Accept()
- ➔ sem_wait(&empty) -> if there are open slots in the array, continue. Else, sleep
- ➔ sem_wait(&sem) -> lock for critical region. Only 1 thread in the program may ever enter
- ➔ array[in] = client socket
- ➔ in = (in + 1) mod BUFFERSIZE

- ➔ sem_post(&sem) -> exit critical region: now the consumer thread waiting may enter its critical region
- ➔ sem_post(&full) -> increment full since we just filled one slot in the array

The consumer threads are created before the dispatcher starts doing its job. I use pthread_t [numThreads] to declare threads and use pthread_create() in a for loop to start the number of specified threads. (Number of specified threads will be acquired from getopt, explained later)

Pthread_create takes NULL as attributes. It executes the function executeWorker() and takes thread_id[i] as its only argument: the thread's integer ID.

**executeWorker(threadID)**

My thread function: the consumers.

Int socket;

Forever:

- ➔ sem_wait(&full) -> sleep until at least 1 slot in the array is filled
- ➔ sem_wait(&sem) -> thread enters critical region
- ➔ socket = array[out]; -> extract the client socket from the buffer
- ➔ out = (out + 1) mod BUFFERSIZE
- ➔ sem_post(&sem) -> exit the critical region
- ➔ sem_post(&empty) -> increment the number of empty slots in the buffer since we just extracted one
- ➔ processRequest(socket) -> does everything done in Assignment 1, with two extra features: logging and healthcheck

Only 1 thread among the dispatcher and worker threads can ever modify the global array, since they share the semaphores used to enter the critical region. This ensures data consistency and avoids race conditions.

## LOGGING

Global variable universal_offset = 0;

Int logFlag = -1;

In the main function, I find out if logging is specified using getopt (explained later) and set the global variable logFlag to 1;

I also open the logfile specified with flags O_CREAT, O_TRUNC, O_WRONLY and permissions 0600: only user will have read and write permissions.

I close the logfile here and open so that each thread can open and close it again in executeWorker.

At the end of a successful request, I check if logFlag == 1 and then calculate the space needed for the request's log contents. I calculate the space taken by the header and the body separately.

For example, the first line of a PUT/GET will take up: 14 + number of digits in content-length value + the length of the fileName

The body will have contentLength / 20 number of lines that store 20 hex-bytes each. If the contentlength isn't divisible by 20, then it will take up a total of num_lines * 69 + ((contentL % 20)*3 + 9) bytes.

// Critical region

I store the universal_offset in a new local variable: start_offset and increment universal_offset by the total Spsce taken by the file being logged.

// End of critical region

I pass the start offset to my logger function.

The logger function takes as its arguments the response Code, the contentLength, the operator number, the start offset, and the first line of the request.

If the responseCode is 200 or 201: It reads 20 bytes from the file by opening it. It prints the padded zeroes and then the 20 bytes by converting each byte into a hex representation. The last byte printed ends in a new line. Every byte is written to the logfile using pwrite. After every pwrite, the start_offset is incremented so the next byte is written after the previous byte.

If the responseCode isn't 200 or 201, then the request has failed. In that calse, the first line of the headers is extracted using sscanf. I build the header for the logfile using snprintf and then I pwrite it to the logfile.

## HEALTHCHECK

I keep two global variables numClients and numErrors. They're initialized to 0. Every time I add a client socket into by global array, I increment numClients. Every time I discover an error when processing the requests, I increment numErrors. The increments are done in a critical region protected by a semaphore. When responding to the healthcheck, I have a function countDigits

that checks the number of digits in numClients and numErrors. I add 1 to their sizes for the newline character and then I send the response using dprintf.