Darren Eam
SID: 1536854

Design Document for Assignment 2:

Idea: construct a client-server model based on HTTP/1.1 Protocol, accepts GET, HEAD and PUT requests. This time, we should be able to process concurrent requests.

Command: “httpserver [-N (num_threads)] [-l (logfile_name)] (PORTNUM)”

Restrictions/details:

- Valid resource names are 27 characters long, of only ASCII characters.
- GET requests: Take a file from server and write it to the client
- PUT requests: Take a file from the client and write it to the server
- HEAD requests: Like GET requests, but the file isn’t sent back to the client. Just check if its there I guess
- Must implement multithreading so that simultaneous requests can be completed smoothly

Modules:

- readWriteFD(inputFD, outputFD, content_len)
- Subroutine to validate resource name (int validateName(char* recName))
- Subroutine to process GET requests (void processGET(socketFD, getFD))
- Subroutine to process PUT requests (void processPUT(socketFD, resource_name, content_len))
- Subroutine to process HEAD requests (void processHEAD(socketFD, getFD))
- Subroutine to take a FD and process a request (void processRequest(socketFD))
    - Read from the socket FD
    - Parse the first line, break into sections via strtok()
        - Request type
        - Resource name
        - Protocol number
    - validateName() to check if resource name follows specs
    - If the name is good, check the request type and handle accordingly (if PUT, GET, HEAD; then …)
        - processGET()
        - processPUT()
            - Get the content len from the header before calling function
        - processHEAD()
- Thread process function(void *threadProcess(void *thisThread))
    - ~~Initial wait() conditional variable, only proceeds when signalled (from main())~~
    - Semaphore wait()
    - Get a job from the queue
    - Semaphore post()
    - Process the job

Darren Eam
SID: 1536854

- Queue struct, with push/pop functions to go with (enqueue(queue, obj), dequeue(queue), front(queue), rear(queue))
- Subroutine to write to a logfile: (void rdWrLogFile(int log_fd, int data_fd, char request, char resource, int content len))
    - Sem wait
    - Format the header of the log entry
    - If the request is not a HEAD request (we have data to write)
        - Counter = 0
        - While counter != content len
            - Make buffers for initial read, saving the conversion of the initial read, and converting this byte
            - Read a chunk
            - For each byte
                - If the index % 20 == 0, then we're at the beginning of a line
                - "\n%08d %02x", counter + i, this byte
                - Else we aren't
                - " %02x", this byte
                - memset the buffer
            - Add the chunk size read to the counter
            - Write to the log file
            - Memset the converted and initial read buffers
    - Format the end of the log entry
- Subroutine to write an error to a logfile (void rdWrLogFileError(int log_FD, char* request, char* resource_name, int response_code))

Darren Eam
SID: 1536854

Global Variables:

- Log_file: file descriptor for lockfile
- Log_bool: boolean to signify presence of log file
- ~~Lock: lock for adding a thread to the critical region~~
- ~~Cond_var: conditional variable to wait/go, inside thread process~~
- Semaphore full: for a full queue
- Semaphore empty: for an empty queue

Main():

- [insert starter code]
- Create global variable for log_file, initially should be some impossible val like -1
- Create variable for number of threads (num_threads)
- Use getopt() to parse the command line. This should iterate thru argv[] to find specified flags (‘optind’ is for the current index, ‘optarg’ is the current argument)
    - Check for -N
        - If so, set num_threads variable to what is specified (optarg)
    - Check for -l
        - If so, create a file with that name (optarg), save FD to global log_file var
- If -N is not specified, then set num_threads to 4 (default)
- ~~Initialize the locks~~
- Initialize the semaphores
- Make the threadpool
- while(true)
    - Accept the incoming connection
    - Semaphore wait() if queue is full
    - Push the FD to the queue
    - Semaphore post() to increase number of jobs in queue