Introduction

> *simple HTTP server that responds to GET, PUT, HEAD,commands with filenames up to 27 ASCII characters. Now with pthread support will serve N clients at once. Option to create a logging file that logs all requests and data sent in hex. Healthcheck will return #errors/#total_number_of_requests.*

Data decisions : get it working before optimization

- Data Decisions
    - I started with a list structure I built in CSE 101 and added enqueue and dequeue functions
        - A single Queue will hold all incoming requests, and be enqueued by the main function
            - Benefits
                - Simple fast implemenation
                - Compared to a faster circular queue it does not have a limit to the amount of backlogged connections
                - Can backlock clients waiting if all threads are busy while more client connections are recieved
            - Drawbacks
                - My list is double linked, I can reduce overhead by recreating a single linked queue
                - A hashtable with queues stored inside would allow me to see which file each thread is holding and handle write data hazards
                - By having multiple queues for each thread I can decrease the amount of overlapped locks and increase parallelization
                    - This will more overhead
                    - Will also require a dedicated dispatch thread and algorithm to control which thread gets a request

- httpObject
    - Created during each handling of a client request locally, stores data such as :
        - request data :
            - filename
            - httpversion
            - command
            - content length
        - data buffer
        - any body_data transfered with header
- threadArg
    - Single object shared between all threads
    - Includes :
        - client queue
        - list of worker threads
        - queue_mutex

        - wakeup_condition
        - healthcheck variables
        - healthcheck_lock
        - log_file name
        - log_file offset

Design decisions : get it working before optimization

- Main Loop
    - On startup initalizes all variables mutexes and server socket and starts N threads that go to sleep
    - Daemon
        - Accepts client connections
        - Locks Queue_lock
        - Enqueues client connection
        - signals worker thread
            - From the manpage
                - "If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked"
                - This will prevent the thundering heard problem and only wakeup one thread to service the queue
        - Unlocks Queue_lock
- thread service function
    - Daemon
        - Locks Queue_lock
        - while queue is empty
            - pthread_cond_wait
                - Gives up the lock and goes to sleep waiting on main thread to signal to wakeup
        - Unlock Queue_lock
        - Preforms same service as HW#1
        - Locks healthcheck_lock
        - increments healthcheck variable
        - unlock healthcheck_lock
        - If logging file has been specified executes log_success or log_failure
- HealthCheck
    - Two ints shared between all threads by threadArgs
        - Mutex locks for access and incrementation
    - Get healthcheck returns both numbers seperated by newline
- Logging
    - I preform logging after sending my response inside the thread -This is a HUGE bottleneck for my threads

      ```
      - One solution would be to have a logging thread
      - However current solution is okay because after turning in my
      ```

```
assingment logging body data will be turned off and cost will be
negligible
```

- To write a log asynchronously I use pwrite
    - I calculate the size my logfile will need using arithmetic
    - I lock the log_file_lock
    - get the current file offset
    - increment the file offset by the calculated space needed
    - unlock the log_file_lock
    - open and read a large chunk of the file to be logged
        - This has glaring problems with if it's the most recent file, but it's this is the KISS implementation so we will ignore the fact that users can change the file between logs
            - A solution would either be to log while handling the requests, but this would add too much complexity to my handle_request function
            - A better solution would be a data structure that could generate locks for each file and lock the files during requests. A hashtable would be good here.
- Get healthcheck writes the log numbers to a temp file and then logs that file using the normal logging algorithm

---

Functions :

void* serveClient(void* threadArgs)

> *DESCRIPTION: Preforms client serving / handling hw#1 PRECONDITION: threadArgs have been initalized POSTCONDITION: runs forever*

void log_success(struct httpObject* message, void* threadArgs)

> *DESCRIPTION: Reserves log_file space and logs requests handled and body data into the log_file PRECONDITION: log_file has been defined POSTCONDITION: log_file has recorded the request*

void log_failure(struct httpObject* message, void* threadArgs)

> *DESCRIPTION: Reserves log_file space and logs requests handled into the log_file PRECONDITION: threadArgs have been initalized POSTCONDITION: runs forever*

int readSend(int fd, int client, struct httpObject* message)

> *DESCRIPTION: repeated read / sends data for GET requests PRECONDITION: fileName has been opened() POSTCONDITION: returns -1 if error sets message status else returns*

int recvWrite(int fd, int client, struct httpObject* message)

> *DESCRIPTION: repeatedly reads/writes file descriptor to stdout PRECONDITION: fileName has been opened() POSTCONDITION: returns -1 if error else returns 0*

void construct_http_response(struct httpObject* message)

> *Generates message based of message->status_code 200 OK 201 Created 400 Bad Request 403 Forbidden 404 Not Found – the case in GET and HEAD requests 500 Internal Server Error*

int read_http_response(ssize_t client_sockd, struct httpObject* message)

> *DESCRIPTION: parses through http request PRECONDITION: full header has been read POSTCONDITION: returns -1 if error sets message status, else sets message data accordingly*

int process_request(ssize_t client_sockd, struct httpObject* message)

> *DESCRIPTION: Executes request bassed of message PRECONDITION: message conents have beens set POSTCONDITION: returns -1 if error sets message status executes request*

ssize_t recv_full(ssize_t fd, struct httpObject* message){

> *DESCRIPTION: repeated recvs until full header is read in PRECONDITION: socket connection has been set POSTCONDITION: returns -1 if error sets message status, records any body data read*

int openfile(char* fileName)

> *DESCRIPTION: Given a char*(string) opens the file & returns the file descriptor number PRECONDITION: fileName is not null POSTCONDITION: returns -1 if error else returns file descriptor for fileName*

readWrite(int fd, char* fileName)

> *DESCRIPTION: repeatedly reads/writes file descriptor to stdout PRECONDITION: fileName has been opened() POSTCONDITION: returns -1 if error else returns 0*

Testing:

I tested modules as I added them by manually doing curl requests and checking the results by my expectations, most of the scripts I ran were shared by classmates, near the end I wrote bash scripts to measure preformance by generating 8 MiB files then requesting them and measuring time, then doing diff on the files with the original to measure accuracy.

Useful testing commands:

curl get

> *curl localhost:8080/filename.txt*

CURL PUT COMMAND (-T, --upload-file)

> *curl -T localfile.txt localhost:8080/filename.txt OR curl --upload-file localfile.txt localhost:8080/filename.txt*

CURL HEAD COMMAND (-I, --head)

> *curl -I localhost:8080/filename.txt OR curl --head localhost:8080/filename.txt -T, --upload-file -I, --head -w, --write-out*

curl -s http://localhost:8080/FILENAME curl -s -T FILENAME http://localhost:8080/FILENAME curl -s -I http://localhost:8080/FILENAME

Check http code only curl -o /dev/null -s -w "%{http_code}\n" http://localhost:8080/FILENAME Check content type curl -o /dev/null -s -w "%{header_size}\n" http://localhost:8080/FILENAME

telnet www.example.com 80 GET \foobar HTTP/1.1 Host: www.example.com