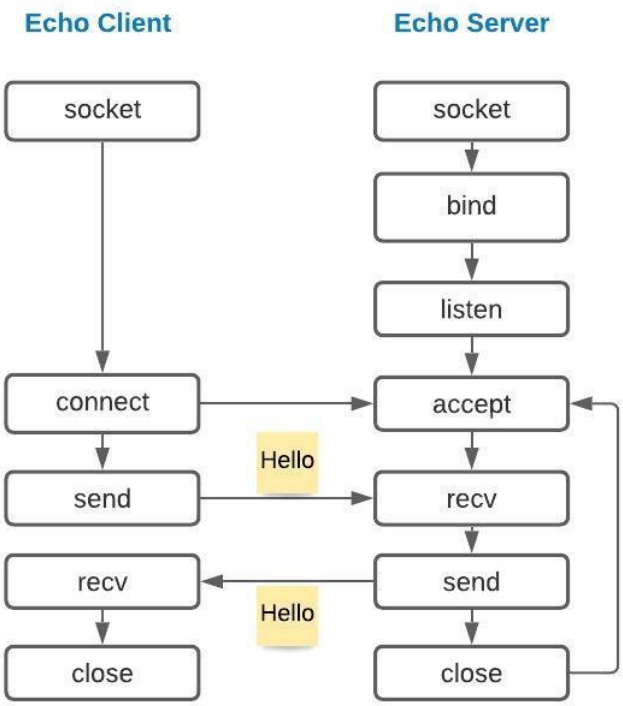
README

Introduction

This Readme file is a consolidated documentation of the four sections of project I.

1. Echo client / server
2. Transfer client / server
3. Getfile client / server
4. Multithreaded Getfile client / server.

Echo Client – Server



The Echo client and server implementation involves communication between the client and server using the C socket API. Once connection is successful, the server should echo or print the message sent from the client and vice versa. An addrinfo struct is defined with the required specifications and passed on to the getaddrinfo method. To make the client / server accept connections from both IPv4 and IPv6(IP agnostic), used the AF_UNSPEC for the addrinfo element ai_family and ai_socktype to SOCK_STREAM

for TCP protocol. Additionally , the ai_flags element was set to AI_PASSIVE for the server addrinfo specification to make the socket suitable for binding. The getaddrinfo returns a list of address information which can be iterated in a loop to successfully create client and server socket. For the echoclient, once the socket file descriptor is created, the connect method is called upon to connect with the server.

For the echoserver, after socket creation, the server socket is set to SO_REUSEADDR option to allow the server to reuse the address for future communications. The socket file descriptor is bound to the available address using the bind() method. The listen() function is invoked for the echoserver socket to specify the maximum client connections that can be served.

The methods socket , connect , listen and accept return a positive value after successful operation. The list of addrinfo structs is freed using freeaddrinfo method. Since this is a streaming protocol, the message content will be transmitted in bytes which may not be null terminated . Therefore, the buffer used to receive the message should be memset to null characters '\0' or a null character should be added at the end of the received message for successful printing of the same.

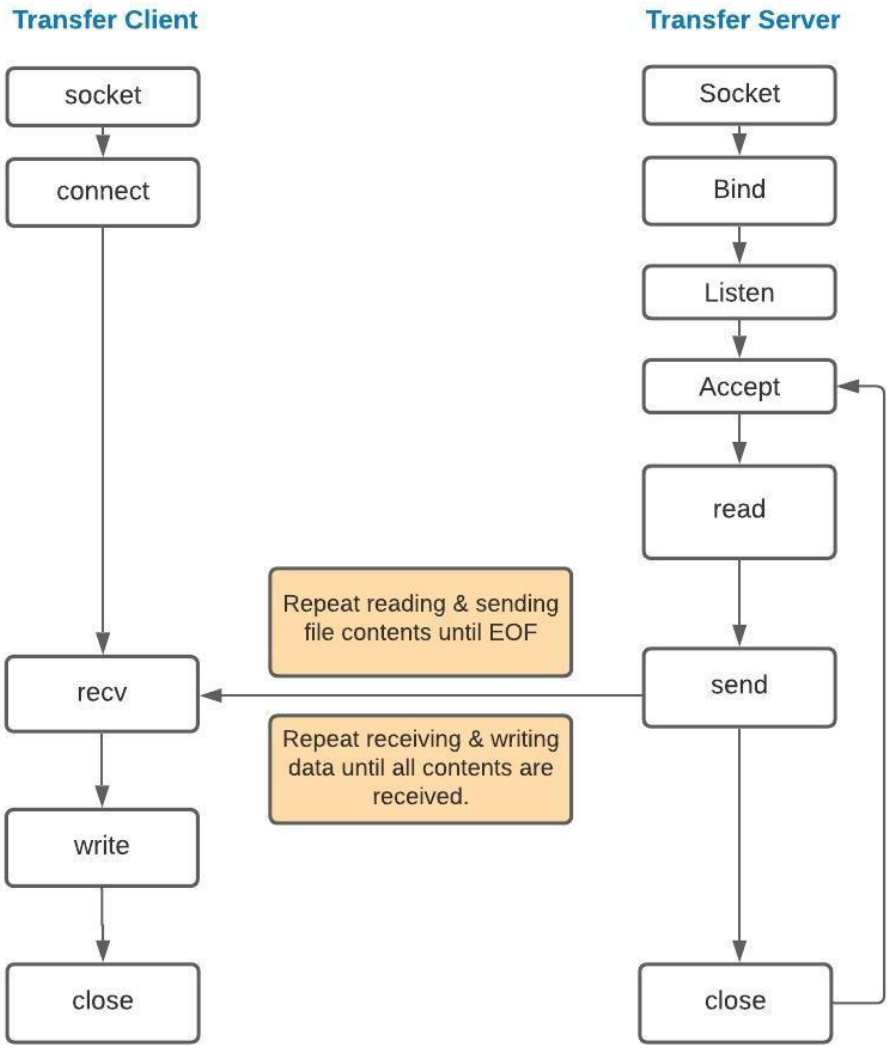
Once the message from both client and server is sent and received, the server reverts to accepting new connections and the client closes the socket.

Test done locally:

Successful printing of message (fixed length) passed from client to server and vice versa.

Transfer Client – Server

In this part of the assignment ,once the connection is established, the transfer server reads the contents of a requested file and sends it to the client via the socket. The data read from the file is stored in a buffer and a pointer to the buffer is passed to the send operation specified with the number of bytes read. Since information is passed from one socket to another via network, there are limitations in socket level read / write and send /recv. It is not guaranteed that all contents will be transferred or received in a single transmission.



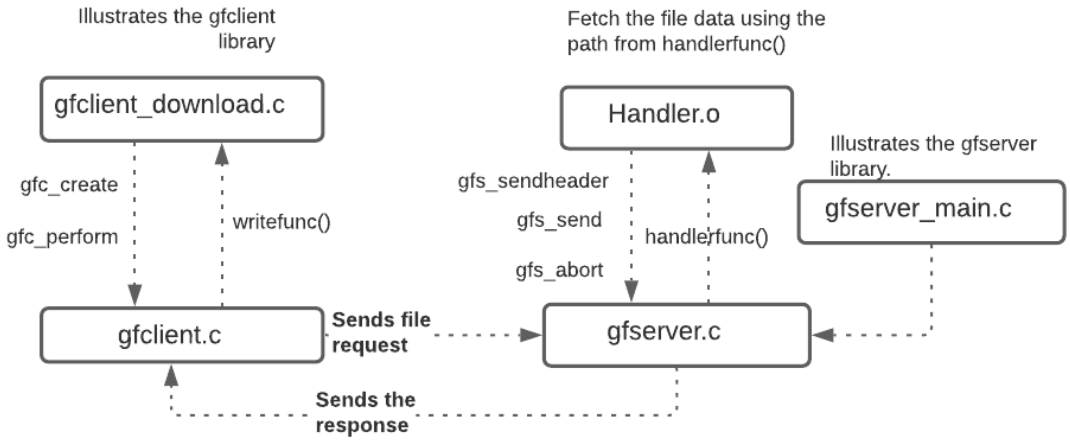
Therefore , the read / write and send / receive processes must be repeated in order to successfully transfer the file. Each iteration of read provides the number of bytes successfully read which can be passed as an argument to the send operation . Similarly, each iteration of recv gives the number of bytes received ,which will be useful for write operation. Since the size of file is unknown to the client , the receive operation is repeated multiple times so that all the data is completely transferred, post which the server closes the socket signalling the client the end of file transfer.

To avoid memory leaks, made sure variables declared are not left un-initialized and free the memory used for the file transfer.

Tests done locally :

Entered the name of the file to be transferred as a command line argument , the transfer server successfully transmitted the file content by printing the number of bytes sent in total which matched with the number of bytes received in the transfer client. Able to view the transferred file content (image and a text file).

Part I : GETFILE client / server



In this implementation, we are to create a client and server library that interact with each other using the GETFILE protocol. Upon parsing the request, the gfserver responds to the client with the file status ,file size and file data obtained from the registered call-back function to the handler. The client receives the server response parses it and begins to save the file content by using the write call-back registered.

gfclient:

The client_download.c modelled as a client library, invokes the methods gfc_create, gfc_perform and others implemented in gfclient.c by passing the opaque pointer gfcrequest_t *gfr defined inside gfclient. Since the struct pointer **gfr is passed to all methods necessary for successful establishment of connection and file transfer, defined struct with elements such as the server, port , writefunc call-back, status, headerfunc call-back. Inside the gfc_perform method the request header string is constructed as per the GETFILE protocol along with the path from the workload and sent to the

server. If the request satisfies the protocol specifications, the client receives the response header containing the file status, size of the file , end marker followed by the file content. The response from the server is received in bytes in a continuous manner and checked for the presence of header marker '\r\n\r\n'.

Each call to recv gives the number of bytes successfully received. Calculate the length of the response header to find the possible amount of file data fetched in bytes received in the first response (Total bytes received – (header length + 4 bytes request marker). Using pointer arithmetic move the buffer pointer past the marker and stream the file data to the write function call-back registered with the library. The receive operation is performed repeatedly until file size is reached.

The response from the server can be of status GF_OK,GF_ERROR,GF_FILE_NOT_FOUND,GF_INVALID which is saved in the status element of gfcrequest_t *gfr. The server status is converted to a string status (OK,ERROR,FILE_NOT_FOUND and INVALID) by passing the status from the (*gfr)->status to the gfr_getstrstatus method. The number of bytes received in total is saved in the bytes_received element of the opaque pointer. After completion of a request, the gfc_perform method returns 0 or -1 based on the file status along with the number of bytes of file data received printed out in the console.

gfserver:

The gfserver_main.c modelled as server library uses the methods such as gfserver_create, gfserver_serve, gfs_sethandler implemented in gfserver.c. The opaque pointer gfserver_t is used by the gfserver to pass to all implemented methods, register the call-back function handlerfunc, set the handler argument, set port and maximum connections for the server. Another opaque pointer , gfcontext_t *ctx is used by the gfserver to save the information of the currently serving request which will be needed once the call-back handler returns with file status and data. The request received in the gfserver_serve method is parsed to check for the presence of request marker, the correctness of scheme, method and file path.

If the scheme, path or method is incorrect, the gfserver_serve method sends a response with the header GF_INVALID back to the client and breaks out of the connection to

serve the next request. If the request sent is as per the protocol, the file path and gfcontext_t pointer is passed to the handler call-back handlerfunc().The handler receives the file path , reads the data from the file requested , calls the gfs_sendheader and gfs_send to pass the status of the request , file size and file content.

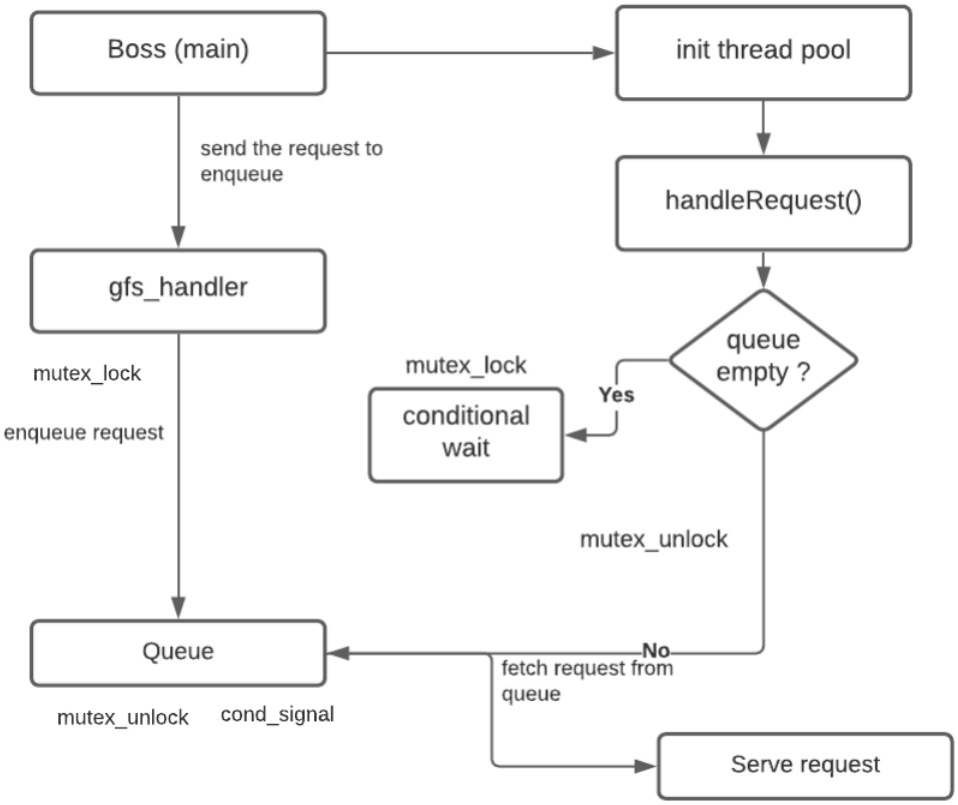
If the file requested is present or not present, or if the handler encounters errors such connection interruption, failed file read operation , a numeric file status is returned to the gfs_sendheader method which is compared to the status defined in the gfserver.h to send the appropriate response to the client. The handler calls the gfs_abort method to close the connection using the gfcontext_t pointer which contains the information of the associated client .

The response and request are sent across the socket in bytes and therefore decided to read the server response, client request byte-by-byte to check for header marker and used string functions such as strstr, strtok to possibly split the header.

Tests done locally:

1. Tested the server response to invalid request scheme , method and file path. The client console printed out the expected result .
2. Checked for the number of file bytes received in the client side to the actual file size in the server.
3. Checked for appropriate status return from server to client in cases FILE_NOT_FOUND and ERROR.
4. Verified if the client closes the socket after successfully receiving the header and file data based on the file size passed from the server.
5. Verified if the gfc_perform method returns the appropriate return code for each status from the server.

Part II Multithreaded GETFILE client / server



In the Multithreaded Getfile server implementation , the main method or the Boss thread initiates a pool of worker threads by calling the init_threads method and passes the number of threads mentioned in the command line. Upon initiation, the worker threads execute the void thread function where they are placed in pthread_cond_wait under pthread_mutex_lock until the queue is empty. The Boss thread then sends the requests received from gfserver to the handler call-back which in turn enqueues them in the Queue under pthread_mutex_lock. Once enqueued, the mutex is unlocked and the worker threads are signalled using pthread_cond_signal variable.

The conditional wait ends when the queue is not empty , mutex is unlocked , and the worker threads are signalled to pop a request from the queue and process them to read and send the file data. After successfully sending the response header and the file contents, the worker threads revert to serving the next request.

Multithreaded gfserver_main.c

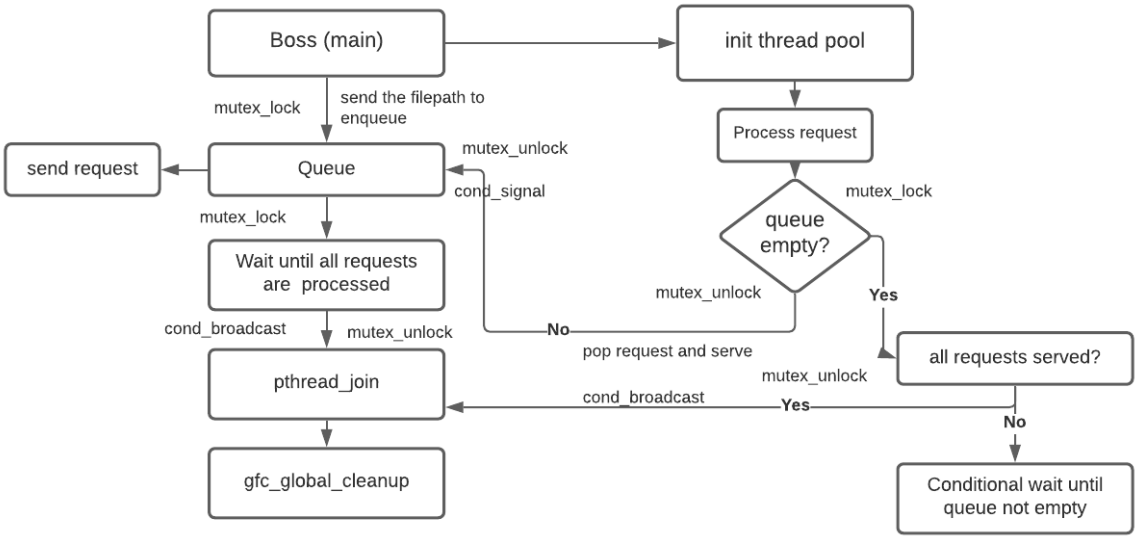
Declared the global variables mutex, request queue and the conditional variable. In the main function, allocated memory for the steque and called the steque_init method to initiate the queue. The init_threads() method is called to initiate the worker threads pool. Passed the request queue as void argument to the handler call-back . Inside the void thread function handleRequest, the worker threads will be in conditional wait inside mutex lock until the queue is empty.

Declared and defined a struct (requestInfo) to hold the context and file path from the handler. The global variables defined in gfserver_main mutex and conditional variable are declared as extern variables in the handler.c.The handler upon receiving the gfcontext_t double pointer , file path and void argument, stores the value of the pointer in the struct requestInfo pointer and sets the ctx to NULL. Under mutex_lock the request struct is enqueued as a steque_item to the request queue. Post enqueuing, the mutex is unlocked and the threads are signalled by pthread_cond_signal.

When the wait condition ends(queue not empty) , a request is popped from the queue . The file path from the request struct is passed to the content_get method which returns a file descriptor if file is found. If not, a status of FILE_NOT_FOUND is sent as response to client. Using fstat, the size of the file is determined and passed to the gfs_sendheader along with the context which holds the information of client currently being served. The file contents are transferred to the client via gfs_send method until the file length is reached after which the request and context are freed.

Multithreaded gfclient_download:

In the multithreaded client implementation of the Getfile protocol, the boss thread creates a pool of worker threads. Based on the number of requests the file path from workload is enqueued in the request queue. The worker threads upon launching, stay in conditional wait until the queue is empty. When a request is enqueued under mutex lock , the threads are signalled using the pthread_cond_signal and the mutex is unlocked. After the conditional wait ends, the worker threads dequeue a request and process it to send to the gfc_perform method of gfclient.



The process is repeated until all requests are dequeued, processed and sent to the server and all files are successfully received. The boss awaits until all the requests are processed, the mutex is unlocked so that the worker threads join the boss thread to terminate. Finally, the gfc_global_cleanup() method is called to free the resources that are allocated memory during the process.

Control Flow:

The global variables mutex ,conditional variable, requestQueue ,requests_processed and requests_enqueued are declared and defined. In the main method, steque is allocated memory and initiated using steque_init. The requests are enqueued in a loop under mutex_lock and the worker threads are notified by pthread_cond_signal after mutex unlock. The requests_enqueued variable is incremented during each enqueue. Similarly, after each successful receiving of the file content, the request_served variable is incremented inside a mutex lock and unlock followed by pthread_cond_broadcast .

Meanwhile, the boss thread waits for the worker threads in a conditional wait until the requests_processed and requests_served are less than the actual number of requests.

When the requests_served and the request_processed reach the actual number of requests, the conditional wait mutex is unlocked and the control breaks out of the loop returning to the main method. The pthread_join function is called upon in a loop for all the threads to join the boss thread and terminate. Once the worker threads are terminated ,the gfc_global_cleanup is invoked to free the memory used for global resources.

As per suggestion made in a piazza post , used pread() to successfully read file data since it allows for multiple threads to perform I/O operations without undefined behaviour.

Tests done locally:

For client: Tested the client for varied number of threads and requests.

When no of threads = no of requests , the worker threads successfully processed the request and terminated once the files are received.

When the number of threads is (greater or lesser)than the number of requests, the threads failed to join with the main thread and continued to wait under queue empty mutex lock.

For server: Tested if the worker threads start serving the request fetched from the queue in a synchronized manner and return to wait condition to serve upcoming requests.

References

Echo client/server :

- "Beej's Guide to Network Programming." n.d. Beej.Us. https://beej.us/guide/bgnet/.
- "Sockets Tutorial." n.d. Www.Cs.Rpi.Edu. Accessed September 23, 2020. https://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html.
- zx1986. 2013. "Zx1986/XSinppet." GitHub. March 17, 2013. https://github.com/zx1986/xSinppet/tree/master/unix-socket-practice.

- "An Advanced Socket Communication Tutorial." n.d. Users.Pja.Edu.Pl. Accessed September 23, 2020. http://users.pja.edu.pl/~jms/qnx/help/tcpip_4.25_en/prog_guide/sock_advanced_tut.html.
- Eduonix Learning Solutions. 2017. "Socket Programming Tutorial In C For Beginners | Part 1 | Eduonix." YouTube Video. *YouTube*. https://www.youtube.com/watch?v=LtXEMwSG5-8.
- "C Library Function - Memset() - Tutorialspoint." n.d. Www.Tutorialspoint.Com. Accessed September 23, 2020. https://www.tutorialspoint.com/c_standard_library/c_function_memset.htm.
- "How to Convert Unsigned Short to String in C?" n.d. Stack Overflow. Accessed September 23, 2020. https://stackoverflow.com/questions/63666912/how-to-convert-unsigned-short-to-string-in-c.

Transfer client /server:

Reused the socket level code part from Echo client / server

- "C++ - Sending Image (JPEG) through Socket in C Linux." n.d. Stack Overflow. Accessed September 23, 2020. https://stackoverflow.com/questions/15445207/sending-image-jpeg-through-socket-in-c-linux.
- "Beej's Guide to Network Programming." n.d. Beej.Us. Accessed September 23, 2020. https://beej.us/guide/bgnet/html/#sendrecv.
- "C Cut Char Arry and Save Binary Data from Socket." n.d. Stack Overflow. Accessed September 23, 2020. https://stackoverflow.com/questions/32908299/c-cut-char-arry-and-save-binary-data-from-socket.
- recv. (n.d.). Retrieved September 23, 2020, from pubs.opengroup.org website: https://pubs.opengroup.org/onlinepubs/009695399/functions/recv.html
- "Send." n.d. Pubs.Opengroup.Org. Accessed September 23, 2020. https://pubs.opengroup.org/onlinepubs/009695399/functions/send.html.

- "C - Socket Programming -- Recv() Is Not Receiving Data Correctly." n.d. Stack Overflow. Accessed September 23, 2020. https://stackoverflow.com/questions/30655002/socket-programming-recv-is-not-receiving-data-correctly.

Getfile client / server:

Reused the socket level code part from Echo client / server

- "Strtok - How to Split HTTP Header in C?" n.d. Stack Overflow. Accessed September 23, 2020.https://stackoverflow.com/questions/22732119/how-to-split-http-header-in-c.
- "C - Differ between Header and Content of Http Server Response (Sockets)." n.d. Stack Overflow. Accessed September 23, 2020. https://stackoverflow.com/questions/16243118/differ-between-header-and-content-of-http-server-response-sockets.
- "Writing to/Reading from File Using Pointers, C." n.d. Stack Overflow. Accessed September 23, 2020. https://stackoverflow.com/questions/31388934/writing-to-reading-from-file-using-pointers-c.
- Referred the below Github file to understand the process control flow https://github.com/pumaatlarge/OMSCS/blob/master/gfclient.c
- "C Double Pointer to Structure." n.d. Stack Overflow. Accessed September 23, 2020. https://stackoverflow.com/questions/7638612/c-double-pointer-to-structure#:~:text=If%20you%20only%20need%20to.
- "C Cut Char Arry and Save Binary Data from Socket." n.d. Stack Overflow. https://stackoverflow.com/questions/32908299/c-cut-char-arry-and-save-binary-data-from-socket.

Multithreaded Getfile client /server:

During the process of implementation , had to rework and create a new gfserver_main and gfclient_download file multiple times(ruined the provided source files with too much coding) copied code from old file.

- GIOS Pr1 high-level code design. 2020. "GIOS Pr1 High-Level Code Design." Google Docs. 2020. https://docs.google.com/drawings/d/1a2LPUBv9a3GvrrGzoDu2EY4779-tJzMJ7Sz2ArcxWFU/edit.

- “POSIX Threads Programming.” n.d. Computing.Llnl.Gov. Accessed September 23, 2020. https://computing.llnl.gov/tutorials/pthreads/#PassingArguments.
- “Multi-Threaded Programming With POSIX Threads.” 2019. Kent.Edu. 2019. http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html
- 2020. Kent.Edu. 2020. http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/thread-pool-server.c.
- Producer consumer example code shared in the lecture videos.

- “Pread(2) - Linux Manual Page.” n.d. Man7.Org. Accessed September 23, 2020. https://man7.org/linux/man-pages/man2/pread.2.html.
- “C Using Fstat() to Read Size of File.” n.d. Stack Overflow. Accessed September 23, 2020. https://stackoverflow.com/questions/39896316/c-using-fstat-to-read-size-of-file.
- “How Do I Share Variables between Different .c Files?” n.d. Stack Overflow. Accessed September 23, 2020. https://stackoverflow.com/questions/1045501/how-do-i-share-variables-between-different-c-files.