2803ICT – Assignment 2 Report

Himath Ratnayake

Contents

1. Problem Statement

The goal of this project was to run a client-server system that is multithreaded and can be used for multiprocessing. Interprocess communication will be done using shared memory and threads are used to factorise 32 numbers. In this implementation mutexes are used with both the server and client being multithreaded.

2. User Requirements

The following outlines the user requirements for the program:

1. The user must be able to input a number equal to or above 1 that will then be used for factorising
2. The user is able to input the letter "q" to quit the program at any point
3. If the user doesn't enter any inputs into the program for more than 0.5 seconds, the client will show current progress of each query
4. When a query is complete, the time taken to complete the query is returned to the user
5. The user is able to enter "0" if there are no current ongoing requests, which will initiate a test mode

3. Software Requirements

The following outlines the software requirements for the program:

Server Requirements

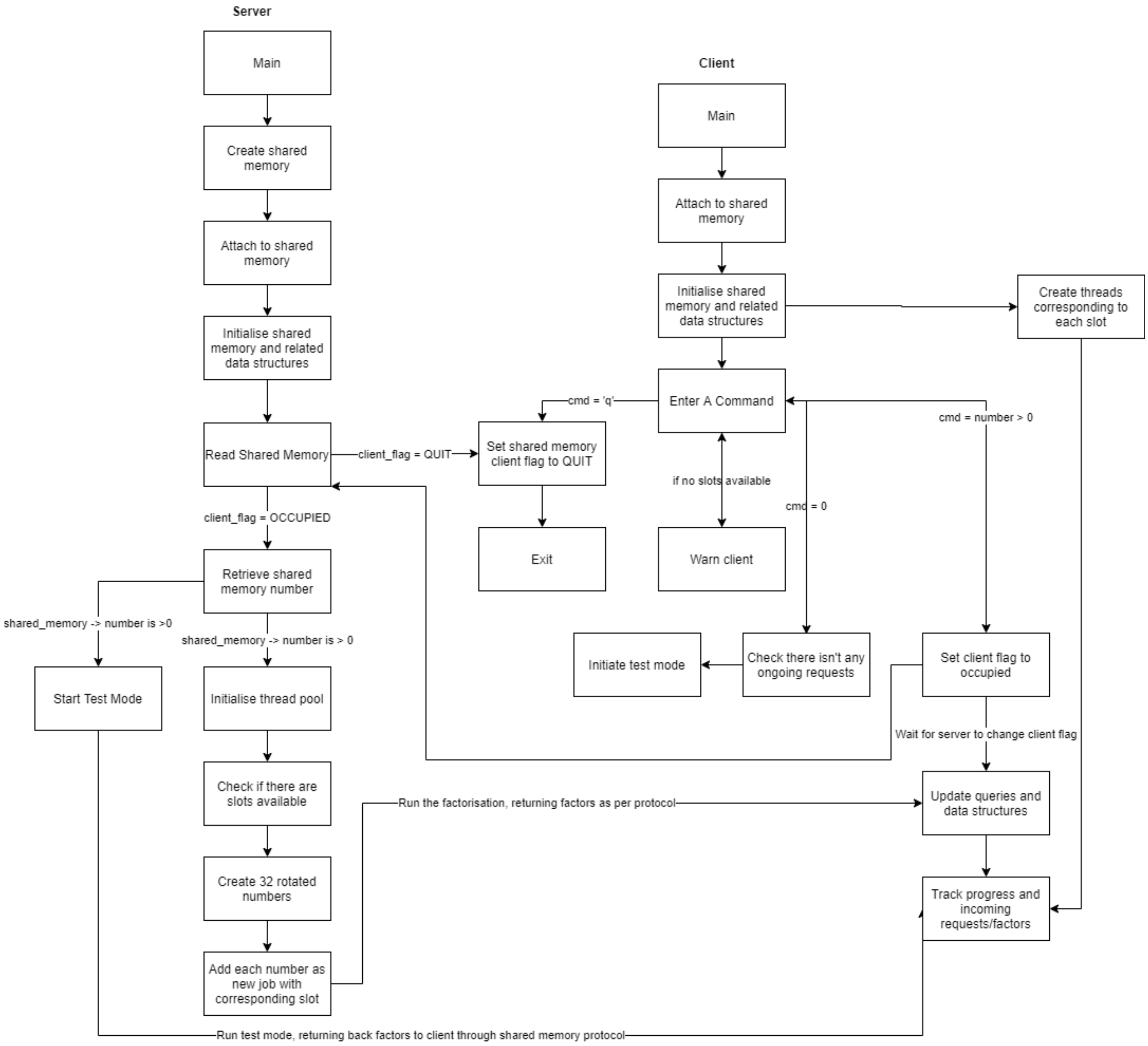
- The server must be multi-threaded, with two modes: normal and test mode
- If the server receives the integer 0 from a client, it will start test mode
- If the server receives the letter 'q' from the client, it will terminate and disconnect from shared memory
- When an integer that is not 0 is given, the server will then create 32 numbers, comprising of the original digit and 31 binary rotations of that digit.
- After the 32 digits are created, they are then factorised through trial division using threads.
- The server needs to be able to handle up to 10 requests without being blocked – this is signified by all the slots available in shared memory being set to occupied.
- Each thread needs to pass the factors it has found to the client as soon as they are found back to the client in the correct slot they were found.
- The slot used by the server for responding to its request will be identified to the client by the shared 'number' variable in the handshaking protocol
- The thread access needs to be synchronised so that factors are not lost
- Each thread's progress will be reported in percentage increments of 5%; individual thread progress is used to calculate overal progress, and is passed back to the client through the progress array in shared memory.
- When a query is done, the server needs to return an appropriate message back to the client
- In test the mode, the server simulates 3 user queries, creating 10 threads per query, such that the 10 threads all together report the numbers 0-99.
- There is a random delay of between 10 and 100 ms in the test mode when sending back factors

Client Requirements:

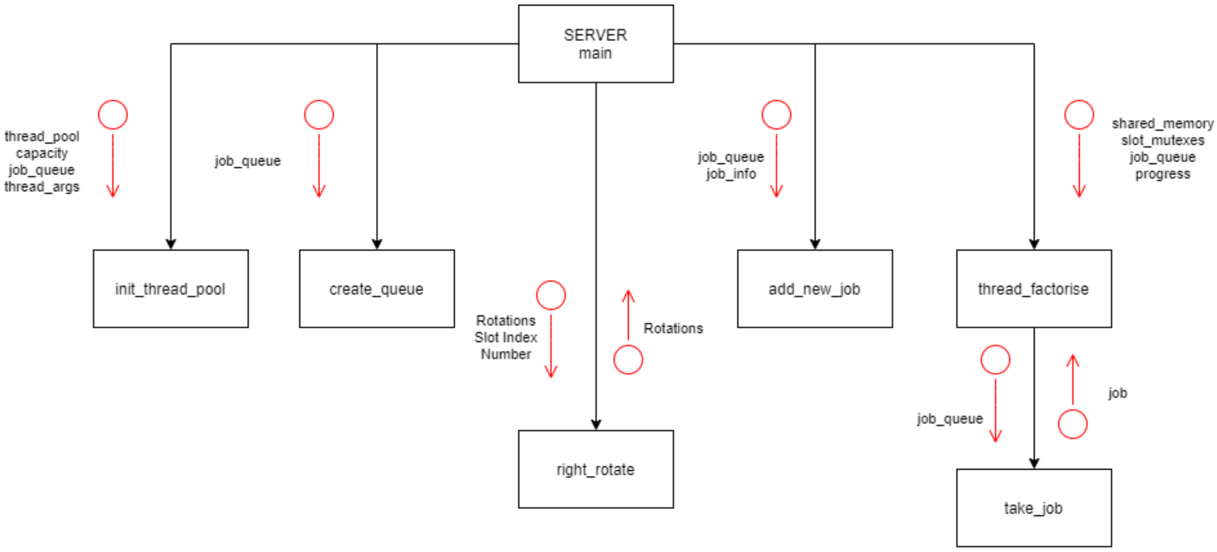
- This implementation uses a multithreaded client
- Client will prompt user for an input request
- This input is sent to the server, with data written to 'number' in shared memory, and a message is received back accordingly
- The client always immediately prints out server responses, as well as query completions and time taken to complete a query
- The client is non blocking
- Up to 10 server responses are outstanding at a time, and if there's more than 10 the client needs to warn the user that the server is busy
- Progress update messages need to be given every 500s until theres a server response or user request
- Progress format should consistent of the query number, the progress percentage read from the server, and a progress bar
- The time taken to complete a query will be printed to client screeen once the server finds all factors
- In test mode, the client will print out the numbers from 0 to 99 as they are received from the server, ensuring no numbers are lost.

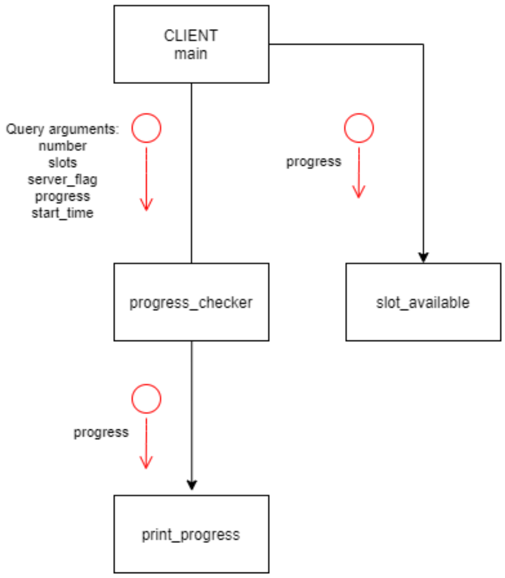
4. Software Design

High Level Design – Logical Block Diagram



Structure Chart





Software Functions (Server)

Main (server)	
Description	The main driver function for the server. This is responsible for running the main server and client loop and initialising all relevant data structures.
Information	First, the server sets up the shared memory. This is done by initialising a struct named shared_memory. A shared memory key is created using the ftok function and shmat() is used to attach this to the initialised shared_memory struct using a shared memory id generated using shmget. This shared memory id (shm_id) is set as a global variable and is only ever changed at the start of this initialisation with shmget, and is also used when terminating the program. The shared memory struct is then initialised with default values so that is then ready to be used. A 2 dimensional array known as rotations is then initialised and allocate_rotations is used to allocate memory for it. A slot mutex array of size SLOT_NUM, as defined in the shared attributes .h file (default value of 10) is then created and initialised using pthread_mutex_init() for each mutex in the slot_mutex array. A number of structs are also initialised. They are Thread_Pool, Job_Queue, and Job. Explanations of these structs is done in the data structures section below. The other struct initialised is Thread_Args, which will be passed to the thread function so it can access shared memory, slot mutexes defined earlier, the job queue and the remaining jobs.

right_rotate	
Description	This fuction is used for creating 32 rotations of the input number given by the client and storing it in the rotations array
Information	The input parameters for this function are the rotations 2D unsigned long int array, the int slot number that client request will occupy in shared memory, and the unsigned long int number that needs to be rotated. In a loop, an unsigned long int number known as rotated_number defined within the function is assigned a value of: (number >> i) | (number << ROTATIONS_NUM – i). This essentially acts as a bit shift to the right by a certain amount for the original number, with the amount of shift being dependent on "i" (the amount of iterations done so far). This rotated number is then assigned to the 2d array at rotations[slot][i], to signify one added rotation variant of the original input number for its corresponding slot. This process is repeated for up to 32 times – the amount of rotations is dependent on the value of ROTATIONS_NUM, as defined in the shared_attrs.h header file.
Return value	The return type of this function is void.

allocate_rotations	
Description	This function is responsible for dynamically allocating memory for the "rotations" 2d array.
Information	The unsigned long int array rotations is passed into the function. Then memory is dynamically allocated using malloc based on the size of the usngined long int x SLOT_NUM as defined in shared_attrs.h (default value of SLOT_NUM is 10). After this, at each slot of *rotations[i], further memory is allocated for each slot i to contain ROTATIONS_NUM amount of unsigned long ints (once again, ROTATIONS_NUM si defined in shared_attrs.h with a default value of 32). This will then result in the rotations 2d array initialised in the main function being dynamically allocated.
Return value	The return type of this function is void.

thread_factorise	
Description	This is the thread function that carries out the factorising for a given number.
Information	This function is invoked when creating the threads for factorisation. pthread_create() expects a pointer as the argument (which will be passed to the thread function). This pointer is to the thread_args struct, which contains a pointer to the shared_memory, slot mutex, job queue, progress and the remaining jobs that need to be done. Within the function, the job_queue structure is extracted from the thread_args by Job_Queue *job_queue = thread_args->job_queue. Within the thread_args, this function will also changed attributes within the shared memory; namely the server flags for each slot when slots are updated with new factors, and the slots within the shared memory themselves when the new factors are found. (thread_args->shared_memory->slots[slot_number]). The slot mutexes within the thread args are also used for locking and unlocking critical sections as needed, and the job queue from the thread args is needed for taking and completing the new jobs as they come in.
Return value	The return type of this function is void.

create_thread_pool	
Description	This function is responsible for initialising the thread pool. This is what is used to launch the threads as needed for each slot and start the factorisation process.
Information	This function takes the Thread_Pool struct, job_queue struct and the thread_args structs as arguments as well as a capacity value as defined by the program for the amount of threads to be made. Furthermore, it also takes another argument called "function" which denotes the actual function that the threads will run – this is either the "test_function" or "factorise_thread" function depending on whether the program is run in test mode or normal mode.

	The thread_pool struct's variables are then initialised with memory allocated for the pool of pthreads itself based on the defined capacity. It then creates the threads using pthread_create, passing in the thread args as an argument. As mentioned earlier, pthread_create() expects a pointer to thread_args as the argument (which will be passed to the thread function).
Return value	The return type of this function is void.

create_queue	
Description	This function is used to create the queue data structure used for processing jobs
Information	The job queue is passed into this function that was initialised in main. Then, the queue's head, tail, size and length are set. Memory is also then allocated for the number of jobs that need to be processed. Mutexes are also initialised for popping and adding jobs using pthread_mutex_init().
Return value	The return type of this function is void.

Add_new_job	
Description	This is a helper function for the Job queue, responding for adding new jobs into the queue
Information	This function accepts the current job queue structure, as well as a new job struct. First the job queue's size is checked to make sure it hasn't exceeded the maximum allocated size. Then the job is added at the tail of the queue, and the queue size incremented.
Return value	The function returns an int of 1 for successful completion, or 0 if the queue length is exceeded.

Take_job	
Description	This is another helper function for the job queue, responsible for retrieving the next job at the head of the queue to be processed
Information	This function accepts the current job queue structure, as an argument. The job to be popped is defined as the job at the head of the queue. This head is then removed, and the queue size decremented. The head job is then returned to be used.
Return value	The function returns a struct "Job".

Finish_program (Server and Client)	
Description	This function is for exiting the the program. When called it will terminate the client program, set the flient flag to QUIT in shared memory, then exit, freeing all allocated resources with it.
Information	Shmctl() is used to remove the shared memory current set up. This uses the shm_id, which is a global variable.
Return value	The return type of this function is void

Test_function	
Description	This function is for initiating the test mode for the server.
Information	The arguments from thread_args are passed into the test function as well as the job queue, similar to thread_factorise. The behaviour of this function is similar to the thread_factorise function, except that instead of using trial division, numbers from 0 to 99 are simply returned instead.
Return value	The return type of this function is void

Main (client)	
Description	The main driver function for the client. This is responsible for accepting and verifying client inputs, as well as initiating the threads for progress checking.
Information	First, the server sets up the shared memory. This is done by initialising a struct named shared_memory. A shared memory key is created using the ftok function and shmat() is used to attach this to the initialised shared_memory struct using a shared memory id generated using shmget. Next the query structs are initialised and the threads for progress checking are made. Then, in a while loop user inputs are asked for. A number of tests are done in each function to ensure that the code is robust to responding to a number of user inputs such as invalid ones, test mode under various conditions, and also to quitting. When test mode is initiated successfully, a global variable known as test_mode is set to 1; this is needed as the progress_checker function needs to know whether it needs to print progress (it does not need to in test mode).

Progress_checker (client)	
Description	This function is responsible for updating the progress of each slot, as well as outputting the server responses of each slot.
Information	The query struct in the client function is passed in as an argument to the thread function, and has information from the shared_memory within it. The timeval struct is a global variable that is also updated within this function with each update that the client does in response to the server. This is used for the progress checker to calculate the time between sending the progress bars and ensure it happens every 500ms. The progress mutex is another global variable, and is locked while the progress for each query is being printed, and then unlocked after.
Return value	The return type of this function is void

Data Structures

This sections aims to detail the data structures within the client and server programs.

CLIENT: Query	
Type of Structure	struct
Description	This is the structure used for storing information regarding each query, and acts as the bridge between the client to shared memory communication. This struct is necessary so that it can be passed into the progress_checker function as an arg and can be used to print key information from shared memory like progress.
Data Members and their purpose	Copied from shared memory: • num (unsigned long int) • slots (unsigned long int array) • slot (a specific slot number) • server_flag (char array) • progress (float) • start_time (timeval struct)
Functions that use it	The query struct is first initialised in the main client function, then passed in as a thread argument in the progress_checker function.

CLIENT: Job	
Type of Structure	struct
Description	This is the structure that contains the information for a specific “job” which the thread function must do
Data Members and their purpose	• num (unsigned long int): rotated variation of a number • slot timeval struct) slot number corresponding to the thread
Functions that use it	The Job struct is used by create_queue, take_job, add_new_job functions.

CLIENT: Job_Queue	
Type of Structure	Queue (struct)
Description	The job queue is the data structure that contains all the jobs that are added by the main function.
Data Members and their purpose	• *jobs (Job data structure): all the jobs that are still yet to be processed • Pop_mutex (pthread_mutex_t): mutex for popping jobs off queue • Add_mutex (pthread_mutex_t): mutex for adding jobs into the queue • Head, tail, size, length (int): variables that convery basic information about the queue
Functions that use it	The Job_queue is used by create_queue, take_job, add_new_job functions, as well as the create_thread_pool and thread_factorise functions.

CLIENT: Thread_Pool	
Type of Structure	struct
Description	This data structure is used to store all the threads created and the related jobs through job_queue
Data Members and their purpose	• Capacity (unsigned int): the number of threads to be made • Used (unsigned int): the number of threads that are currently being used • Pool (pthread_t): data type used to uniquely identify a thread • Job queue (struct): data structure defined above, which keeps a list of the jobs to be completed
Functions that use it	The thread_pool is used by create_thread_pool function as well as the main function.

CLIENT: Thread_Args	
Type of Structure	struct
Description	This data structures stores all the relevant arguments necessary for the thread factorisation to run.
Data Members and their purpose	• Shared_memory (struct): the interprocess communication method for this project, which is used to communicate with the client • Slot_mutex (Pthread_mutex_t): mutex object used to control process synchronisation for each slot • Job_Queue (queue struct): defined as above, used to store the jobs left to do • Progress (char array): used to store the progress at each slot • Remaining_jobs (int array): used to store the jobs at each slot
Functions that use it	The thread_args above are first initialised in the main function, and passed into the create_thread_pool function, which in turn invokes the creation of threads through the thread_factorise function. Hence, thread_args will also be used by the thread_factorise function to get the relevant information needed for factorisation and subsequent communication with shared memory.

Detailed Design

CLIENT main function

Attach to shared memory
Initialize the queries struct for each slot with the items in shared memory
Initialize 10 threads in the client for checking progress of each slot, passing in a queries struct corresponding to each slot number for each thread
In a while loop:

> Prompt for an input
> If command length is 1, or an invalid letter continue from new loop
> If command is ‘q’, set the client flag in shared memory to ‘QUIT’, then exit
> If there isn’t a slot available in shared memory slots, warn client and continue looping
> If the command entered is zero, and there’s no ongoing requests, tell client test mode is starting, and set test_mode global variable to 1
>
> Put the number inputted into shared_memory->number
> Set the shared_memory->client_flag to occupied
> While the client flag is occupied, the server will read the number in shared memory
> Once the client flag is no longer set as occupied, the new number in shared memory will be set as the slot being used for the original number. Save this slot number
>
> Reset the progress in shared memory and update queries struct in client program with new information like the number and slots occupied as well as start time
> Record the current time in a separate variable for “last update”, signifying when the last update was for the program client

CLIENT progress_checker Function

In the progress checker function, retrieve the queries struct arguments
Create a time struct for keeping track of the end time and current time

In a while loop:

> If test_mode global variable is 0 meaning we aren’t in test mode:
>
> > Get the current time
> > Start mutex lock for tracking progress
> > Check if the current – last update’s time difference is greater than or equal to 500 ms
> >
> > > If it is, iterate through the slot numbers, and for the server_flags in shared memory that are not yet finished, and print out the progress from the progress array in shared memory,
> > > Then, make the last_update the current time, as client has been updated again
>
> From the query struct, If the server flag’s slot is occupied:
>
> > Print out the query and factor from the server
> > Set the server_flag for that specific slot to empty again to signify that new data was received
>
> From the query struct, if the server flag’s slot is finished:
>
> > Print that the query is complete
> > Print the time taken in total to finish that request.
> > Set the slot back to -1, to stop the thread from doing anything more to it

SERVER main function
Create shared memory
Attach to shared memory
Initialised shared memory variables
    Set client_flag to 0
    Set number to 0
    For i in range 10 (corresponding to number of slots):
        Shared_memory->slots[i] is set to -1
        Shared_memory->server_flag[i] is set to FINISHED
        Shared_memory->progress[i] is set to 0
Next, define a 2D rotations array, and allocate memory for this
Create 10 slot_mutex pthread mutexes (one for each slot), and initialise each one

Initialise thread pool, job queue, and thread args
Initialise a queue that is 320 jobs long, corresponding to 32 rotations * 10 threads (SEE BELOW)

In a while loop:
    If the client flag in shared memory is set to QUIT, exit program
    If the client flag is set to OCCUPIED:
        Check the number in shared_memory->number
        If the number is 0, start test mode (SEE BELOW)
        Otherwise, do the factorisation of the number (SEE BELOW)

Factorisation Pseudocode
If it's the first run of the while loop, initialise the thread pool with 320 threads
Go through all the slots in shared memory to see if there is a slot available
If there is, pass this slot number into the right_rotate function along with the rotations 2d array previously allocated

The add_mutex object for the job queue is locked
Then, in a loop from 0 until the number of rotations (32):
    A job is initialised with the slot to use and one of the 32 rotated numbers
    New_job function called to add this job into the queue
The add_mutex is then unlocked

Once that job is finished, set the shared memory variables for current slot to EMPTY, as well as the client_flag and server_flag for said slot to EMPTY too.

SERVER factorise thread function

Retrieve thread arguments
Read the current job queue into the thread arguments
Define a new job
In a while loop:

```
    The pop_mutex object from the job queue is locked
    Then, if there is a job available, it is popped from the job queue
    (This job as defined earlier contains the slot number relating to it, as well as the
    rotated number that needs to be factorised)
    The pop_mutex object is then unlocked
    Initialise bool isPrime to 0
    For I in range 2 to the number:
        If the number % I is 0:
            The number is prime so bool isPrime is set to 1
            For j in range 2 to i/2:
                If I % j is 0,
                The number is not prime
            If isPrime:
                Update the server_flag slot with the new factor only once the
                client has read the last one
                Lock the slot mutex for the given slot
                Update shared memory slot with the factor found (i)
                Unlock the slot mutex for the given slot
                Tell the client there is a new data to read by setting the
                server_flag slot slot to 1 as per handshaking protocol
    Once the above loop is done, we now know that all possible factors have been found
    and sent back to the client, so this thread is effectively "finished".
    Therefore, the remaining jobs for the current slot can be decremented
    Then, the progress can be calculated based on the jobs left, and placed in the
    shared_memory->progress[slot] based on the current slot to be read by the client
    If there are no more jobs left, the thread slot is set as finished, which signals the
    client to print out that their query is done, as well as how long the thread took to
    finish
```

Test Mode Pseudocode

```
For i in range(10):
    New Number = i * 10
    Store in array

For t  in range (3):
    Lock add mutex
    For number in range 10:
        Add new job, with slot = t and number stored in testmode numbers array
    Unlock add mutex
```

In Test Function:

Retrieve args

In while loop:

Take new job

For k in range(10):

Update the slot with the number only once client has read last one (server flag is no longer 1)

Delay by random amount between 10 and 100ms

Lock slot mutex

Add to shared memory at given slot the number in job + k

Unlock slot mutex

Tell client there is new factor to be read (server flag[slot] = 1)

5. Requirement Acceptance Tests

Software Requirement No	Test	Implemented (Full /Partial/ None)	Test Results (Pass/ Fail)	Comments (for partial implementation or failed test results)
1	Server and client connect to shared memory successfully	Full	Pass	
2	Client prompts user for initial input, and then continues accepting input requests	Full	Pass	
3	The server needs to be able to handle up to 10 requests without being blocked	Full	Pass	
4	If there's already 10 outstanding requests, the client needs to warn the user that the server is busy	Full	Pass	
5	Any client inputs are written to number in shared memory so that they can be read by server	Full	Pass	
6	If the server receives the integer 0 from a client, it will start test mode	Full	Pass	
7	If the server receives the letter 'q' from the client, it will terminate and disconnect from shared memory	Full	Pass	
8	When an integer that is not 0 is given, the server will then create 32 numbers, comprising of the original digit and 31 binary rotations of that digit.	Full	Pass	
9	After the 32 digits are created, they are then factorised using threads.	Full	Pass	
10	Each thread needs to pass the factors it has found to the client as soon as they are found back to the client in the correct slot they were found; The slot used by the server for responding to its request will be identified to the client by the shared 'number' variable in the handshaking protocol	Full	Pass	
11	The client always immediately prints out server responses, as well as query completions and time taken to complete a query	Full	Pass	
12	The thread access needs to be synchronised so that factors are not lost	Full	Pass	

Software Requirement No	Test	Implemented (Full /Partial/ None)	Test Results (Pass/ Fail)	Comments (for partial implementation or failed test results)
13	Each thread's progress will be reported in percentage increments of 5% by the server; individual thread progress is used to calculate overall progress, and is passed back to the client through the progress array in shared memory.	Full	Pass	
14	When a query is done, the server returns an appropriate message back to the client	Full	Pass	
15	Progress update messages need to be given every 500ms to the client until theres a server response or user request	Full	Pass	
16	Progress format consists of the query number, the progress percentage read from the server, and a progress bar	Full	Pass	
17	The time taken to complete a query will be printed to client screeen once the server finds all factors	Full	Pass	
18	There is a random delay of between 10 and 100 ms in the test mode when sending back factors	Full	Pass	
19	In test mode, the client will print out the numbers from 0 to 99 as they are received from the server, ensuring no numbers are lost	Full	Pass	

6. Detailed Software Testing

No	Test	Expected Results	Actual Results
1	Run server and client from command line	Server will set up shared memory and client successfully connects, with message prompts to show this successful connection	As Expected
2	Enter a letter that is not ‘q’	Error message is given and the server re-prompts for the user to start	As Expected
3	Enter the letter ‘q’	Server and client successfully terminate	As Expected
4	After setting client and server up, enter a number greater than 0	Server returns back factors for all rotations of that number as soon as they are found by the server as well as other server responses – this is done through updating the correct slot with the factor in shared memory; factors are then read by client and printed to client screen.	As Expected
5	After entering a number, check to see if the progress output is sent every 500 ms	The progress bar is shown in one line, with percentage increments of 5%	As Expected
6	Enter a factor, and then wait until the query is completely finished	Once finished, the client will be informed of the number and slot that was finished, and the time taken from the server will be printed to the screen	As Expected
7	Enter 10 numbers and observe the output	The server returns the factors for each number with their corresponding query number, as well as the percentage completion in the form of a progress bar. It is able to handle all 10 queries simultaneously.	As Expected
8	Keep entering factors until all 10 slots are occupied, then enter another extra factor in the client	The client will inform the user that the server is already full	As Expected
9	Enter the letter 0 in client and observe the client output	The client alerts user that test mode is starting. The server threads all together report the numbers 0-99 to the client without any numbers lost. Progress output is also not shown.	As Expected
9	Enter the letter 0 in client while there are still outstanding requests	Error message is sent to the user saying that they need to wait until the requests are finished	As Expected

7. User Instructions

For The Server:

Compile the program and run it in the command line with the syntax below.

- Compiling Syntax: gcc server.c -o server -lpthread && ./server

The server will then start assuming all parameters are correct, and wait for a client to connect to the shared memory. Once requests from the client start coming in, the server will print out information in response to the behaviors of the client and progression of the factorization.

For The Client:

In another terminal, compile the client program, and run it in the command line, following the syntax below.

- Compiling Syntax: gcc client.c -o client -lpthread && ./client

NOTE: Ensure that the server is set up beforehand, with the same shared memory key. If this doesn’t happen, error checking is in place and the program will automatically terminate.

Once connected, the user can then input one of 3 inputs:

- Inputting ‘0’ will make the server start test mode and return to the client screen 3 sets of numbers from 0 to 99.
- Inputting any number greater than or equal to 1 will then return all the factors for the inputted number, and its rotations as denoted by the ROTATIONS_NUM (defined in shared_attrs.h – default is 32 as per the task sheet specifications).
- Inputting ‘q’ will make the client and server terminate

Compiler Notes:

This program was coded in Windows subsystem for linux, and verified to be working in Cygwin64.