2803ICT Systems and Distributed Computing
Mutithreaded Client/Server Factorisor

Caleb Howard

September 9, 2020

1. Problem Statement

The problem objective for this project is to write a multithreaded client server system for multiprocessing. The system should involve a client/server model over shared memory using C. The user can input numbers into a non-blocking client terminal, the number will be sent to the server where its binary representation will be rotated 32 times, where each rotation of the number creates a new thread to factorise the rotation, the server sends the factors back to the client as they are calculated. The client can have maximum 10 outstanding queries.

2. User Requirements

The following outlines the user requirements for the program (for details on how to use the system , refer to the *user instructions* section of this document):

- The user should be able to input numbers into the client without interrupting the terminal output.
- The client should be able to view a factor and the number of the query that was responsible for factorising it as soon as the factor is calculated.
- The user should be able to issue a maximum of 10 queries at a time. If the user tries to issue another query while the number of queries is 10 then an appropriate warning message should be printed.
- The user should be able to view a message when a query finishes, stating how long the query took to execute (seconds).
- The user should be able to to view the percentage of completion for each active query. The percentage is calculated using:
  (threads completed) / (32 (which is the total threads used for the input number)).

3. Software Requirements

1. The program will consist of a multi-threaded server and single- or multi-threaded client process.
2. The client will query the user for 32-bit integers to be processed and will pass each request to the server to process and will immediately request the user for more input numbers or ‘quit’ to quit.
3. The server will start up as many threads as there are binary digits × the max number of queries (i.e. 320 threads). The server will take each input number (unsigned long) and create 32 numbers to be factorised from it. Each thread will be responsible for factorising an integer derived from the input number that is rotated right by a different number of bits. Given an input number K, each thread #X will factorise K rotated right by increasing number of bits. For example, thread #0 will factorise the number K rotated right by 0 bits, thread #1 will factorise K rotated right by 1 bit, thread # 2 will factorise K rotated right by 2 bits etc.
4. The trial division method should be used for integer factorisation.
5. The server must handle up to 10 simultaneous requests without blocking.
6. The client is non-blocking. Up to 10 server responses may be outstanding at any time, if the user makes a request while 10 are outstanding, the client will warn the user that the system is busy.
7. The client will immediately report any responses from the server and in the case of the completion of a response to a query, the time taken for the server to respond to that query.
8. The client and server will communicate using shared memory. The client will write data for the server to a shared 32-bit variable called ‘number’. The server will write data for the client to a shared array of 32-bit variables called a ‘slot’ that is 10 elements long. Each element in the array slot will correspond to an individual client query so only a maximum of 10 queries can be outstanding at any time. This means that any subsequent queries will be blocked until one of the 10 outstanding queries completes, at which times its slot can be reused by the server for its response to the new query.
9. Since the client and server use shared memory to communicate a handshaking protocol is required to ensure that the data gets properly transferred. The server and client need to know when data is available to be read and data waiting to be read must not be overwritten by new data until it has been read. For this purpose, some shared variables are needed for signalling the state of data: char clientflag and char serverflag[10] (one for each query response/slot). The protocol operation is:
    - Both are initially 0 meaning that there is no new data available
    - A client can only write data to ‘number’ for the server while clientflag == 0; the client must set clientflag = 1 to indicate to the server that new data is available for it to read
    - The server will only read data from ‘number’ from the client if there is a free slot and if clientflag ==1. It will then write the index of the slot that will be used for the

request back to ‘number’ and set clientflag = 0 to indicate that the request has been accepted.

- A server can only write data to slot x for the client while serverflag[x] == 0; the server must set serverflag[x] = 1 to indicate to the client that new data is available for it to read.
- The client will only read data from slot x if serverflag[x] ==1 and will set serverflag[x] = 0 to indicate that the data has been read from slot x.

10. The server will not buffer factors but each thread will pass any factors as they are found one by one back to the client. Since the server may be processing multiple requests, each time a factor is found it should be written to the correct slot so the client can identify which request it belongs to. The slot used by the server for responding to its request will be identified to the client at the time the request is accepted by the server through the shared ‘number’ variable.
11. Since many threads will be trying to write factors to the appropriate slot for the client simultaneously you will need to synchronise the thread’s access to the shared memory slots so that no factors are lost. You will need to write a semaphore class using pthread mutexes and condition variables to used for controlling access to the shared memory so that data is not lost.
12. While not processing a user request or there has been no server response for 500 milliseconds, the client should display a progress update messages for each outstanding request (repeating every 500ms until there is a server response or new user request). The repeated progress message should be displayed in a single row of text. The message should be in a format similar to:
    > Progress: Query 1: X% Query2: Y% Query3: Z%
13. When the server has finished factorising all the variations of an input number for a query it will return an appropriate message to the client so that it can calculate the correct response time and alert the user that all the factors have been found.

4. Software Design

List of Software Functions:
The client/server program has 2 source files, client.c and server.c, a list of function definitions for each module is as follows.

<u>Client.c</u>
displayQueries(struct Memory *ShmPTR)

- Takes the shared memory struct and prints each query and its completion percentage.

receiveServerData(struct Memory *ShmPTR)

- This function is continuously called by the run function, it checks if the server has placed a factor in a given slot and prints the factor and query number if a factor is present. It also checks the server has finished factorising an input completely and prints an appropriate message.

sendNumber(struct Memory *ShmPTR, uint32_t number)

- This function is called when the user inputs a message, it takes the shared memory structure and the user input number. It sends the number to the client, then it retrieves and returns the slot number for which the factors of that number will be placed.

run(struct Memory *ShmPTR)

- Takes the shared memory pointer and runs a continuous loop, handles the non-blocking user input and calls all the above functions.

main(int, char**)

- Main program function, initialises the shared memory between the client and server. It initialises the shared memory values and calls the run() function.

Server.c

rotate(uint32_t number, int rotate_degree)

- This function takes a number and a rotate_degree, it it rotates the binary representation of the number to the right by a degree specified in the second parameter. It returns the uint resulting from the rotation.

*factorise(void *data)

- This function pointer is used when creating a thread to factorise a number. It uses the trial division algorithm to factorise and sends each factor to the client as soon as it is found.

initThreads(uint32_t number, int slot_num)

- This function takes a number and a slot number. It handles rotating the number 32 times, and for each 32 rotation it creates a thread using the *factorise() function pointer.

getFreeSlot()

- This function simply checks if there are any slots available and returns an available slot number. If no slots are available it returns 11;

run()

- This function runs the main server loop. It calls all the above server functions.

main()

- Main program function, connects to the client over shared memory, calls the run() function.

Data Structures used in program:

The data structures used in this program:

- Arrays
- Thread_data : a simply structure that is used to pass a slot number and the number itself to the *factorise() function when creating a new thread.
- Memory : A shared memory structure where the client and server communicate.

Pseudocode for the *factorise function:

```
BEGIN
    // number: number to be factorised.
    // sem_wait(): semaphore wait function so threads do not use a slot at the same time.
    // sem_post(): semaphore function to signal completion of critical section.
    // mutex_fact[ ]: contains a mutex for each slot
    // pthread_exit(): exit the current thread
    f = 2
    prev_ f = 0
    WHILE number > 1:
        IF number MODULUS f equals 0:
            // Ensure no duplicates are processed.
            IF prev_f  does not equal f:
                sem_wait(mutex_fact[slot_num])
                // Ensure client is ready
                WHILE server_flag[slot_num] does not equal 0
                    ;
                // Send factor and update prev_f
                slot[slot_num] = f
                server_flag[slot_num] = 1
                prev_f = f
                sem_post(mutex_fact[slot_num])
        number = number / f
        ELSE:
            f = f + 1
    PRINT “Query `slot_num`, Thread `threads_complete[slot_num]` complete”
    pthread_exit(NULL)
END
```

Pseudocode for the rotate function:

```
BEGIN
    // number: the number that will have its binary representation rotated.
    // rotate_degree: the number of times number will be rotated.
    // >> : shifts a number’s binary digits right by a given amount.
    // << : shifts a number’s binary digits left by a given amount.
    // |    : bitwise OR operator.
    IF rotate_degree equals 0:
        RETURN number
    bit_width = sizeof(number) * 8
    rotate_degree = rotate_degree MODULUS bit_width
    temp = number
    number = number >> rotate_degree
    // Build mask for carried over bits
    temp = temp << (bit_width - rotate_degree)

    RETURN number | temp
END
```

5. Requirement Acceptance Tests

Software Requirement No.	Test	Implemented (Full/Partial/ None)	Comments
1	Connect to the server via the client and input a number. Ensure 32 threads are created (The server prints a thread number once it has completed factorising).	Full	
2	Input 2 numbers back to back then issue the quit command and ensure the client and server terminate.	Full	
3	3.1) Input 10 different numbers into the client, ensure, wait for all 10 to complete, ensure (on the server side) that each query had a thread completed (and printed) from 1-32. 3.2) Create a simple version of the program without the rotate function. Input a few different numbers and ensure the factorise function is factorising correctly.	Full	
4	Covered in requirement test 5.	Full	
5	Input 10 numbers, ensure the client is displaying factors correctly. Input another number (11 total), ensure the appropriate warning message is displayed.	Full	
6	Input 10 numbers, ensure the client is displaying factors correctly. Input another number (11 total), ensure the appropriate warning message is displayed.	Full	
7	Input a number, wait for factorising to complete, ensure that an appropriate message is printed stating the query that is completed and the time the server took to complete factorisation.	Full	
8	Repeat requirement test 5. When one query finishes (given a total of 9 queries), input another number and ensure the query is accepted.	Full	
9	Covered in requirement test 8.	Full	
10	Covered in requirement test 6.	Full	
11	Test the the system with and without the semaphore mutexes, ensure that the mutexes are functioning correctly.	Full	
12	Input a number of queries less than 10, ensure that the progress of each query is displayed at the bottom of the terminal.	Full	
13	Covered in requirement test 7.	Full	

6. Detailed Software Testing

No	Test	Expected Results	Actual Results
1.0	**quit command**		
	Issue the quit command.	Client: “client has detached from shared memory”. Server: “server has detached from shared memory”. Both client and server terminate	As expected
2.0	**Greater than 10 queries**		
	Run the *get* command with less than 3 arguments.	“*** WARNING: Server busy.. wait for available query ***”	As expected
3.0	**Non-numeric or negative int input.**		
	Input a sequence of letters (other than quit) and press ENTER.	“ERROR: please enter an unsigned numeric value.”	As expected
4.0	**Appropriate query completion message.**		
	Input two numbers.	For each number the following is printed: “Query X has completed in Y seconds.”. Where X is the query number and Y is the seconds it took to complete.	As expected.
5.0	**Inputting number 1 of the 10 queries has completed**		
	Input 10 numbers, wait for 1 query to finish, input another number.	Another query, number by the same number as the query that just completed will begin. The progress bar will show this query at 0%.	As expected
6.0	**Correct progress displaying.**		
	Input a number.	Ensure the progress bar shows the query at 0% initially. Each time the server prints that a thread is completed, ensure the percentage increases on the client. Because we are using (threads completed / total threads), the percentage should increase by 3% apon each thread completion.	As expected.

7. User Instructions

1. Launch the client if it is not already running.
2. Launch the server.
3. Input a number followed by the ENTER key to submit the number and start a query. Up to 10 queries can be outstanding.
4. Input “quit” followed by the ENTER key to terminate the client and server.

8. Compiling Source

For the client I used the ncurses library, if you want to compile, ensure you have ncurses installed and compile with *gcc client.c -o client -lpthread -lncurses*.