Multi-threaded Client/Server Assessment

Thomas Carney
s5130828

Problem Statement

The aim of this project was to create 2 programs, one being a client and the other a server. The client queries the user for a 32-bit integer as input and sends it to the server. The server first creates 31 more numbers by rotating the bits in the original numbers (An example of this on a smaller scale would be creating 3 numbers from a 3-bit integer. If 2 was entered (010), the server would create 3 numbers 2 (010), 1 (001) and 4 (100)). The factors for all 32 of these numbers are then calculated and sent back to the client to be displayed. Each number's factors will be calculated in its own thread (32 threads per request). The server can handle up to 10 requests simultaneously, each having its own threads (up to 320 threads at once).

User Requirements

- The user runs both the client and the server in the same directory.
- The user can then enter a 32-bit integer into the client window.
- The client will allow up to 10 different outstanding requests to be made.
- The client will display a loading bar that shows the progress of all processing requests
- Once a request is completed the client will print all of the factors found by the server.

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

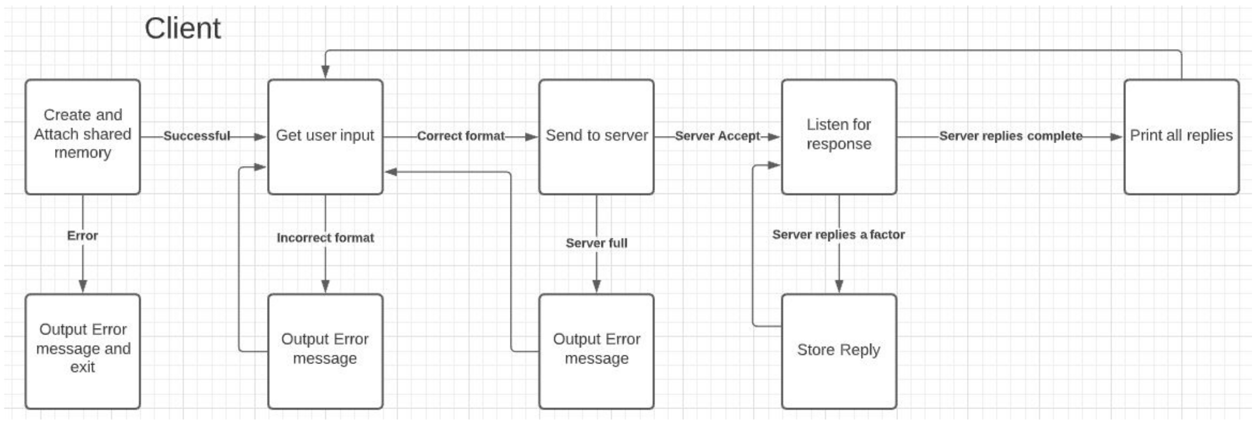
mutexes and condition variables used for controlling access to the shared memory so that data is not lost.

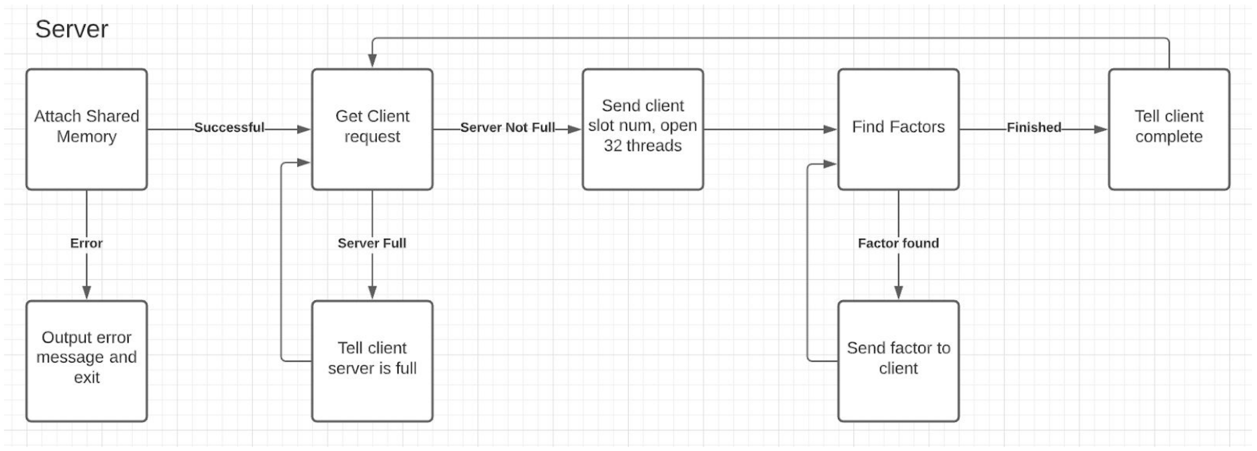
12. While not processing a user request or there has been no server response for 500 milliseconds, the client should display a progress update messages for each outstanding request (repeating every 500ms until there is a server response or new user request). The repeated progress message should be displayed in a single row of text. The message should be in a format similar to: > Progress: Query 1: X% Query2: Y% Query3: Z% If you want you can use little horizontal progress bars i.e. > Progress: Q1:50% ▓▓▓▓______| Q2:30% ▓▓______| Q3: 80% ▓▓▓▓▓▓▓__|

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.

Software Design

High Level Design - Logical Block Diagram





Functions

Client Functions

```
int format_input(char *user_input, long *num_p);
```
Takes user input and a pointer to a long. If the user input is a long integer the number will be put into long *num_p and 0 will be returned. Any other input entered results in 1 being returned, telling the main function that no number was entered.

```
void *listen(void *arg);
```
This function is called with a new thread every time a request is made. The thread then running the function is responsible for listening for all results given by the server for that specific request. When a result is received, it is inserted into a linked list. Once the server finishes finding all the factors, the client calls to print and delete the linked list. This function also records the time elapsed for the server to find all the factors.

```
void print_list(struct Node* n);
```
This function prints the linked list that stores all the factors found for 1 request. (So there would be 10 different linked lists if 10 requests were made.)

```
void push_front(struct Node ** head, long factor);
```
This function puts a factor on the head of the linked list. I chose to insert at the head because the order of the results doesn't matter and insert at head is O(1) compared to O(n) for tail insert.

```
void delete(struct Node *head);
```
Iteratively deletes and frees the linked list.

```
void delete_bar(int length);
```
Prints a backspace character to stdout for the length given (plus a constant number for the format of the bar e.g. the “9 : |” at the start etc).

```
void display_bar(long num, long full);
```
Prints a loading bar to represent the progress that has been made on a request. The bar is filled for the percentage of threads finished. Num is the original request number, and full is the percentage the bar should be filled too (Note: It’s not actually a percentage value).

```
int get_length(long num);
```
Function to find the number of digits in a number. (E.g. get_length(12345) would return 5). This is used to help calculate length to tell delete_bar() how many ‘\b’s to print.

```
void *loading_bar(void *arg);
```
This is a driver function for both display_bar() and delete_bar(). It just calls each for the number of requests made.

Server Functions

```
long bit_rotate_right(long num, unsigned int rotations);
```
This function returns the num parameter bit rotated right by the rotations value

```
void *solve(void* arg);
```
This function is called by a new thread from main for each request made. It then creates 32 threads, each running the find_factors function. Each thread calls find+factors with a different index so each thread can keep track of how many times it needs to be bit rotated.

```
void *find_factors(void *arg);
```
This function is run by 32 threads for each request made. Each thread bit rotates the original number by the correct number of rotation. Then iterates through 1 - n to find all the factors of the new number. When a factor is found it immediately sends it to the listening client. It also keeps track of the number of threads that have been completed to update the loading bar on the client side.

```
void delay(int milli);
```
Delays the program by a number of imputed milliseconds.

```
int slot_request(int server_flag[]);
```
Returns the index of the next available slot. Returns -1 if there are no available slots.

Data Structures

Shared memory structure for all of the server to client information sending.
```
struct Memory{
   long number; // new number
   long slot[NUM_REQUESTS]; // slots for responses.
   int client_flag; // flag for new number
   int server_flag[NUM_REQUESTS]; // flags for response slots.
   long original_num[NUM_REQUESTS]; // track the original values.
   int threads_finished[NUM_REQUESTS]; // track the number of threads finished
for each req.
   int index, current_slot; // index -> for bit rotation, cur_slot -> for temp
slot value.
};
```

A linked list to store each factor as it is found.
```
struct Node{
   long factor;
   struct Node* next;
};
```

Requirements Acceptance Tests

Req. No.	Test	Implementation (Full/Partial/None)	Test Results (Pass/Fail)	Comments
1.	Server uses multiple threads to find Factors	Full	Pass	
2.	Client queries user for 32-bit integer. Quit exits the program.	Full	Pass	
3.	Server starts up 32 * number of requests threads. Which bit rotates and finds factors.	Full	Pass	
4.	Trial division method used.	Full	Pass	
5.	Server handles 10 simultaneous requests without blocking.	Full	Pass	
6.	Client is non-blocking, busy message when request is made and the server has 10 outstanding requests.	Full	Pass	
7.	Client immediately receives server responses and records total query time.	Full	Pass	
8.	Server and client communicate using shared memory	Full	Pass	
9.	Flags used	Full	Pass	
10.	Server does not buffer factors. Sends factors as soon as they are found.	Full	Pass	
11.	Semaphore and mutex	None	Fail (no test).	Could not manage to implement in the given time. Flags were used instead.

12.	Loading bars showing progress of all outstanding queries.	Partial	Pass (Majority of the time)	Works majority of the time. Can be buggy with lots of requests.
13.	Server tells the client when all factors have been found so that the client can calculate the time taken.	Full	Pass	

Detailed Software Tests

No.	Test	Expect Result	Actual Result
1.0	User Inputs		
1.1 1.2 1.3 1.4 1.5	- No Input - Letters - Multiple arguments - Larger than a 32-bit integer	1.1 - 1.5: Error message is displayed, user is queried for new input.	As Expected
1.6	- A 32-bit integer	1.6: Number is sent to the server, all factors are found. Results are printed once they are all found	As Expected
1.7	- multiple 32-bit integers in succession (less than 10).	1.7: Each number is requested to the server and solved simultaneously. Results are printed once each request is finished.	As Expected
1.8	- multiple 32-bit integers in succession (more than 10).	1.8 The first 10 are processed as usual. Every request that would result in more than 10 current requests results in an error message. The	As Expected

		other 10 requests are still satisfied	
2.0	Loading bar		
2.1 2.2	- One request given - Multiple request given	2.1-2.2: Loading bar is displayed showing the progress of all requests that are currently being found.	Usually works as expected. Sometimes the loading bar can be buggy and not delete properly resulting in nonsense being printed. This usually only happens with 7+ requests.
3.0	Output		
3.1 3.2	- One request given - Multiple request given	3.1-3.2 Displays all factors found for all requests given.	Can miss some factors, I believe this is due to the lack of the semaphore in the program.

User Instructions

- If there are no executable files, compile both client.c and server.c with the flag -lpthread. (E.g. gcc client.c -o client -lpthread)
- Run the client executable first, then run the server executable while they are in the same directory.
- Once they are connected enter a number into the client. (Must be 32-bit number).
- The server will display a loading bar as your request is processed, once complete all the factors will be displayed on the client.
- The server supports up to 10 simultaneous requests.