

# 2803ICT ASSIGNMENT 3

Sebastian perry

S5132483

## Contents

Problem Statement .....	1
User Requirements .....	2
Software Requirements .....	2
File Structure .....	2
Software Design .....	3
Shared .....	3
Includes .....	3
Global constants .....	3
Global Variables .....	3
Structs .....	4
Functions .....	4
Server .....	5
Global constants .....	5
Global Variables .....	5
Structs .....	5
Functions .....	6
Client .....	7
Global Variables .....	7
Functions .....	7
Flag play by play .....	9
Requirement Acceptance Tests .....	10
Software Testing .....	11
User Instructions .....	12
Preparations .....	12
Server .....	12
Client .....	12
Usage .....	12
Controls .....	12
Phases .....	12

## Problem Statement

This assignment requires the construction of code which can produce 2 separate executables, a server and client. The server and client work as two separate processes running on the same machine using shared memory to communicate with each other. The client gets input numbers from the user (or command in the case of quit) and sends it to the server for processing before reporting and results it sends back.

## User Requirements

The following will be a collection of dot points outlining the user requirements:

1. The User should be warned if the client is started first (provided the programs were closed properly before with q).
2. The user should be able to input a 32 bit unsigned integer into the client.
3. The user should be able to input 0 to enter a test mode.
4. During the test mode normal request can not be made and the user will be warned.
5. During the normal request mode, a test cannot be started, and the user will be warned.
6. The user should be able to input q in order to quit, notifying both the client and server.
7. The user inputs should be non-blocking.
8. Status updates should be non-blocking.
9. If an input is entered with 10 requests already running, the client will be warned.

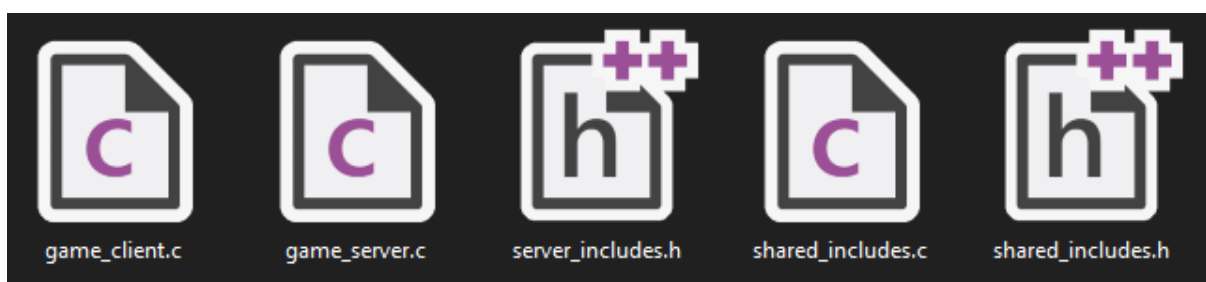
## Software Requirements

The following will be a collection of dot point outlining the software requirements stated by the task sheet:

1. The two applications should use shared memory to communicate.
2. The client input and output should be non-blocking.
3. The server should be able to use multiple threads to process requests from the client.
4. The server should be able to handle at least 10 requests at one time.
5. The server should use a system of flags in shared memory to perform handshakes and ensure that the client has read calculated results before they are replaced with new ones.
6. The server should calculate progress of the solution throughout the solving of the request and report it back to the user whenever an input or output has happened on the client for more than half a second.
7. The server should make use of mutexes to protect vital resources from the race condition problem.

## File Structure

The following body of code is dispersed appropriately throughout the following files:



# Software Design

## Shared

### Includes

```
<stdio.h> <stdlib.h> <string.h> <time.h> <math.h> <sys/types.h>  
<sys/ipc.h> <sys/shm.h> <pthread.h> <unistd.h>
```

### Global constants

GENSIZE 1024 : generic size - the default size allocated to a string

MINVAL 0 : minimum value - the minimum size allowed for input

MAXVAL 4294967295 : maximum value - the maximum size allowed for input

LENVAL 11 : value length - the maximum length of input

SLOWDOWN 0.7 : slow down ammount - a number used by strategic sleep commands throughout the program to induce a longer run time

SLOTS 10 : slots - the maximum number of slots the server has for taking requests

Flags - these were used along with a struct in shared memory to handle states between the programs.

OPEN 20 : slot is open - a flag signalling that a slot is ready to receive a request

READY 21 : slot ready - a flag signalling that a slot is ready to start processing the request on the client's signal

REQUEST 22 : a request in slot - a flag signalling that a slot contains a new request for the server or that the server has finished a request

FILLED 23 : slot filled - a flag signalling that a slot is currently filled with a value to be taken

Taken 24 : slot value taken - a flag signalling that the value that was in this slot has been taken

QUIT 25 : the quit flag - this states that the client has quit and the server should end

TEST 26 : test mode - a flag signalling that the server is in test mode

### Global Variables

struct Shared\_Memory \* sm\_ptr : this is a variable which holds the a pointer to shared memory for both the client and the server.

## Structs

Number : is a shared variable struct that will be used to send and receive numbers between the server and the client.

```
struct Number {  
    uint32_t main_slot;  
    uint32_t slot [SLOTS];  
};
```

Number\_Handshake : is a shared variable struct that will be used for signalling states between the client and the server.

```
struct Number_Handshake {  
    char main_slot_status;  
    char slot_status[SLOTS];  
};
```

Progress : is a shared variable struct that will be used for relaying the progress of each slot in its calculations.

```
struct Progress {  
    char slot_progress[SLOTS];  
};
```

Shared\_Memory : is a shared variable struct that will be used as a container to hold a single instance of all the other shared memory structs.

```
struct Shared_Memory {  
    struct Number number;  
    struct Number_Handshake status;  
    struct Progress progress;  
};
```

## Functions

```
double timeElapsed (clock_t start);
```

when called this function will return the time since it the given time

start : a clock\_t containing the start time to subtract from the current time

returns - a double containing the time since the time given

```
struct Shared_Memory * getSharedMemoryBind();
```

this function binds a variable to the shared memory

returns - a struct pointer of type Shared\_Memory that bound to the shared memory

```
int detectQuit ();
```

detects whether or not the quit signal has been sent

returns - an int either 1 or zero indicating whether or not the quit flag was detected

## Server

### Global constants

SOLVERS 32 :SLOVERS - solving threads count : the number of solving threads which must be started.

### Global Variables

extern pthread\_mutex\_t slot\_reply\_control[SLOTS] : slot reply control : is a mutex responsible for stopping the race condition when sending answers back to the client

extern pthread\_mutex\_t test\_control : test request control : is a mutex responsible for stopping the race condition when enacting dummy requests from the server

### Structs

request\_information : is a struct that will be given to the request handler

```
struct request_information {  
    int slot;  
    uint32_t my_input;  
};
```

solver\_progress : is a struct that will be given to each solver allowing them to report their progress

```
struct solver_information {  
    int * progress;  
    int slot;  
    int solver;  
    uint32_t my_input;  
};
```

test\_holder : is a struct that will be given to each of the test threads

```
struct test_holder {  
    int my_number;  
    int slot;  
};
```

### Functions

```
void *requestHandler (void * arg);
```

this thread function handles the handling of a request to the server starting the various solvers

arg : the arguments variable for the thread function, in this case it will pass an instance of request handler

returns : void \*

```
void *solver (void * arg);
```

this thread function handles the individual solving of a given number derived from the input number of the requests

arg : the arguments variable for the thread function, in this case it will pass the number to solve

returns : void \*

```
uint32_t rightRotate(uint32_t n, int b);
```

this function rotates a given unsigned long by a given number

returns : 32 bit interger containing the time since the function was last called

```
void runTestMode();
```

this function runs the test mode for the server

returns : void

```
void *testHandler (void * arg);
```

this thread function handles the handling of the test threads

arg : the arguments variable for the thread function, in this case it will pass an instance of test\_holder

returns : void \*

```
void *tester (void * arg);
```

this thread function handles the individual solving of a given number derived from the input number of the requests

arg : the arguments variable for the thread function, in this case it will pass an instance of test\_holder

returns : void \*

## Client

### Global Variables

extern pthread\_mutex\_t send request control : is a mutex responsible for stopping the race

### Functions

```
void *communicationReceiver (void * arg);
```

this thread function handles the receiving of any messages from the server

Arg : the arguments variable for the thread function

Returns : void \*

```
void *requestSender (void * arg);
```

this thread function handles making a request to the server when appropriate

Arg : the arguments variable for the thread function, in this case it will pass the number to be requested to the server

Returns : void \*

```
void flushInBuffer ();
```

this function flushes the input buffer

Returns : void

```
int inValid (char input[LENVAL]);
```

returns whether or not the given input is valid

input : an int containing the input to be checked</param>

returns : int as either 1 or 0



```
uint32_t getInput (char * input);
```

processes user input and returns the number

Input : an int containing the input to be checked

returns : unsigned long containing the output as a number

```
void signalQuit ();
```

sets all of the status variables to quit, with the hope that surely the server will see one

returns : void

## Flag play by play

The following will be a play by play for the flags and how they're applied to the status struct in the shared memory structs to allow the two programs communicate:

1. Upon initialisation the main slot and the 10 processing slots are set to OPEN
2. When sending a request, it is put into the main slot of number, and its status is set to request
3. The server detects this fact taking in the number, finding it a slot before writing this slot number back to the main slot, and changing the status of it and the chosen slot to ready
4. Once the client reads the slot number and starts its timer replying with TAKEN the server returns the main slot to OPEN and the slot the input is being calculated on to TAKEN as well
5. From then on when ever it finds a value it will be written to the slot with the status FILLED, the client will then take the number and set it to TAKEN
6. When finished the server will set the slot to REQUEST, the client will take note and reset to OPEN

## Requirement Acceptance Tests

Requirement No.	Test	Implemented (Full/ Partial/ Not)	Test Result (Pass / Fail)	Comment
U1	I started the client first on a fresh start up	FULL	Pass	It returned an error
U2	Input numbers from the max to the minimum of the range a 32 bit number allows	FULL	Pass	The numbers were accepted
U3	Typed 0	FULL	Pass	The test mode activated
U4	Tried to make a request during test mode	FULL	Pass	A warning was returned
U5	Tried to activate test mode during a request	FULL	Pass	A warning was received
U6	q was input	FULL	Pass	Both the client and server exited successfully
U7	Tried making inputs at various times	FULL	Pass	All input were successful
U8	Tried typing through output	FULL	Pass	Output appeared regardless
U9	Tried giving 11 requests	FULL	Pass	The 11 request was met with a busy warning

## Software Testing

No.	Test	Expected result	Actual result
1	Input negative number into the client	It would be knocked back by the client	The client gave a warning about non numerical input and didn't send it
2	Input only multiple of the same number	The client would process the numbers individually and successfully	The client would processed the numbers individually and successfully

## User Instructions

Below will be a list of instructions that are necessary to run the project which was created for this assignment:

### Preparations

This application is designed to solely work on Linux based operating systems and as such is inoperable on windows without the use of bash emulation software such as Cygwin; such running conditions must be obtained. Furthermore, if Cygwin is being used, the following commands

**cygserver-config** then accept by typing **yes**

then **cygrunsrv -S cygserver**

### Server

In order to run the server, you should open the system terminal (or Cygwin terminal on windows), navigate to the directory of the server program and run it with a line similar to the one shown below

`./server.exe`

### Client

In order to run the client, you should open the system terminal (or Cygwin terminal on windows), navigate to the directory of the server program and run it with a line similar to the one shown below

`./game_client.exe`

### Usage

#### Controls

Once started the user can at any time input one of 3 inputs:

1. A 32-bit number that isn't 0 for the server to process.
2. A 0 which activates the test mode.
3. A q character which will safely close the client and server applications.

### Phases

#### *Request phase*

The main phase of the program consists of the server accepting numbers for each of it's 10 slots if free. When accepted into a slot, the program will create a new thread which will in turn create 32 of its own threads to calculate each right bit shifted possibility of the given 32 bit number. Each thread will use the trial division method on it's given number to find every prime factor of a given number, displaying the results and progress to the user intermittently.

#### *Test phase*

Is entered into by inputting 0, when no other requests are already running. It works by starting a thread which in turn, creates 3 threads of its own to send certain numbers back to the user at random intervals.