Systems & Distributed Computing (2802ICT)

Assignment Report

Matthew Prendergast (s5283740)

CONTENTS

PROBLEM FORMULATION

FILE COMPILATION

To compile these files, please follow the instructions below, using GCC in the program directory.

COMPLIATION INSTRUCTIONS: make all

GENERAL OVERVIEW

The problem requires the writing of a multithreaded server and a multithreaded client system. The serve must accept communication with the client using a shared memory structure.

USER REQUIREMENTS

The following user requirements were extracted from the clients brief:

1. The client must repeatedly query the user for a 32-bit integer to be processed, or the letter ‘q’ to terminate the program.
2. When the server is initiated, it must create a thread for each process, and each rotated number that each process will factorise.
3. A rotated number is defined as the output of any 32-bit, where the binary equivalent has been rotated one instance to the right. For each given process, a number will be rotated 32 times – once for each bit – and each of these rotated numbers will be factorised.
4. The factorising method used should be “trial division”.

SOFTWARE REQUIREMENTS

The following software requirements were extracted from the clients brief:

1. The program must be able to run in test mode or in standard mode.
2. The server must accept 10 simultaneous processes. Whenever a process is completed, it will become available again for use.
3. The client and server must communicate using shared memory, defined by handshaking protocol.
4. The progress of each process should be printed to the client terminal.
5. The program should allow for two printing formats.

SOFTWARE DESIGN

HIGH-LEVEL DESIGN

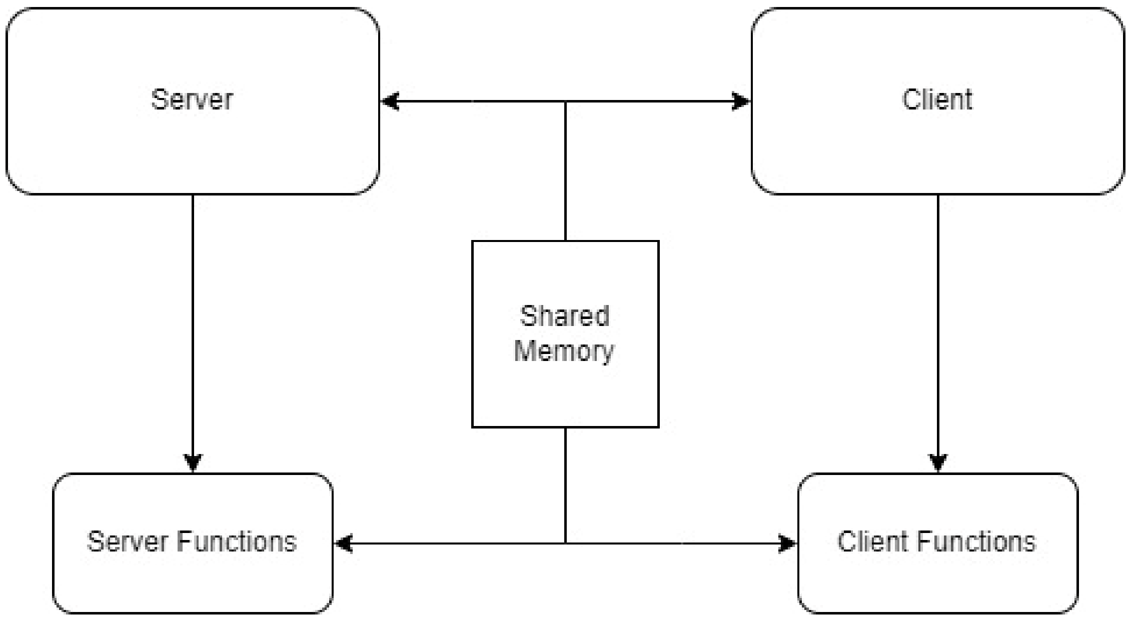

Figure 1.0 – Functional Flow Diagram

PROGRAM DATA (STRUCT MEMORY*)

ENUM

client_states	Enumeration of the states the shared data case be in at any time.

BOOL

connected	Triggers when the client first connects to the server.
standard_mode	Triggers if the user proceeds in standard mode.
test_mode	Triggers if the user proceeds in test mode.

BOOL*

progress_percentage_flag	Triggers when each thread has completed its task and has incremented the percentage.
server_flag	Triggers when a process is reading to be read by the threads.

UNSIGNED LONG*

numbers	An array of numbers which have been inputted by the client for each process.
rotated_numbers	All rotations of the number inputted by the client for each process.

INT

client_flag	Monitors the status of the shared memory in regard to manipulating data.
current_index	The index number which corresponds to the current process.
end_queue	The index number of the last position in the queue.
process_count	The current number of processes running.
test_completed	A count of how many test threads have been completed.
thread_count	A count of how many threads have been completed.

INT*****

available_process_queue	A queue of index numbers to correspond with the process rotation.
progress_percentage	An array to monitor the progress for the factorising of each process.
test_numbers	An array to store the test numbers.

#DEFINE

SHM_NAME	A predefined identifier representing the name of the shared memory segment used in the program.
SEM_CLIENT	A predefined identifier representing the name of the semaphore used for client synchronisation.
SEM_SERVER	A predefined identifier representing the name of the semaphore used for server synchronisation.
MAX_BITS	The maximum number of bits to use in the program.
MAX_32BIT	The highest acceptable input number the user can make.
MIN_32BIT	The highest acceptable input number the user can make. The number 0 is reserved for testing.
MAX_PROCESS	The maximum number of processes the program can run at once.
MAX_LENGTH	The maximum input length for the user loop.
PRINT_FORMAT	The print format that will be used by the program to print progress.

TEST_CASES	The number of test cases used in test mode.
TEST_THREADS	The number of threads used by each test case in test mode.

PROGRAM FUNCTIONS

MAIN() – SERVER

INPUT: None.

RETURN: None.

When initialised, the main() function will launch the server and create populate the shared memory to be used within the program. Once running, the server will wait for a client to connect and begin to process the data.

MAIN() – CLIENT

INPUT: None.

RETURN: None.

When initialised, the main() function of will launch the client and connect to the server using the shared memory that has been initialised. Once the client has connected to the server, it will initialise two threads, one to interact with the user, and the other to print the results from the server.

CLIENT_CONNECTED()

INPUT: void* args

RETURN: None.

The client_connected() function will handle all interactions and inputs from the user. When an input has been received, it will be sent to the server for processing.

CREATE_SERVER_THREADS()

INPUT: struct Memory* shared_data

RETURN: None.

When initialised by the server, the create_server_threads() will create a thread for each process required by the program.

POP_REQUEST_QUEUE()

INPUT: struct Memory* shared_data

RETURN: None.

When initialised by the client, the pop_request_queue() will pop the next process index from the queue to be used by the program. The function will then reorder the queue so it can be used again for the next process.

PRINT_CLIENT_PROCESSES()

INPUT: void* args

RETURN: None.

The print_client_process() function is initialised by the client when it creates a thread to handle the printing of all server progress reporting. This function will print the progress of each process that is currently being run by the system.

PUSH_REQUEST_QUEUE()

INPUT: struct Memory* shared_data, int i

RETURN: None.

When initialised by the client, the push_request_queue() will push the current process index that has just been terminated, back to the queue to be used by the program. The function will then reorder the queue monitoring variables so it can be used again for the next process.

SERVER_CONNECTED()

INPUT: struct Memory* shared_data, sem_t *sem_client, sem_t *sem_server

RETURN: None.

The server_connected() function is responsible for handling all client communications and responses. When initialised, the function will create all required threads to be used by the program, and begin listening for messages and managing all functionality.

SERVER_THREAD()

INPUT:	void* args
RETURN:	None.

The server_thread() function is initialised by the server when it creates the threads required to process the client inputs. Each thread will listen for a flag which has been set in the shared memory. When the flag has triggered, the function will begin to factorise the number it has been provided and notify the client when it has been completed.

TERMINATE_PROCESS()

INPUT:	struct Memory* shared_data, int i
RETURN:	None.

The terminate_process() function is responsible for terminating a process when it has completed. This will include pushing the process index back to the queue and resetting the flags that are being polled by the server threads.

TEST_MODE()

INPUT:	void* args
RETURN:	None.

The test_mode() function is initialised by the server when it created the threads required by test mode. This function will manage the test mode memory and pass all required information to the client.

TRIGGER_PROCESS()

INPUT:	struct Memory* shared_data, int index, int number
RETURN:	None.

When a process is ready to begin, the trigger_process() function will ensure that all of the data required by the process has been initialised and reset in shared memory.

TEST RESULTS

REQUIREMENT ACCEPTANCE TESTING

Software Requirement	**Test Description**	**Implemented** • **Full** • **Partial** • **None**	**Result** • **Pass** • **Fail**	**Comments** **(If Test Failed)**
1	The client can repeatedly query the user for a 32-bit integer to be processed.	Full	Pass	None
2	Any input of the letter ‘q’ will terminate the program	Full	Pass	None
3	When the server is initiated, all threads are created.	Full	Pass	None
4	For each user input, the number is correctly rotated 32 times and stored in shared memory.	Full	Pass	None
5	For each rotated number, each number is factorised correctly.	Full	Pass	None
6	The program is able to run in standard mode.	Full	Pass	None
7	The program is able to run in test mode.	Full	Pass	None
8	The server is able to handle up to 10 processes at one time.	Full	Pass	None
9	The server will send a busy message when there are no more available processes.	Full	Pass	None
10	The server will send a ready message when more processes become available.	Full	Pass	None
11	The server and client communicate using shared memory.	Full	Pass	None

Software Requirement	**Test Description**	**Implemented** • **Full** • **Partial** • **None**	**Result** • **Pass** • **Fail**	**Comments** **(If Test Failed)**
12	The progress of each process is printed correctly to the client terminal.	Full	Pass	None
13	The program allows for two printing formats.	Full	Pass	None

DETAILED SOFTWARE TESTING

No	Test	Expected Results	Actual Results
1.0	**Run Program**		
1.1	Run Server ./server	ALL SERVER THREADS HAVE BEEN CREATED	As expected.
1.2	Run Client ./client	------ INSTRUCTIONS ------ 1. TYPE '0' TO RUN TEST MODE (ONCE ONLY) 2. TYPE 'q' OR 'Q' TO QUIT (AT ANY TIME) 3. ENTER A 32-BIT INTEGER (AS REQUIRED)	As expected.
2.0	**Client Input**		
2.1	Test Mode Initialised	Please Wait - Running Test Mode... RESULTS…. RESULTS….	As expected.
2.2	Quit Initialised	Please Wait - Disconnecting you now...	As expected.
2.3	Invalid Input	ERROR: USAGE - INTEGER 1 <= N <= 4294967295	As expected.
2.4	Valid Input	Progress: Q1:100% ▓▓▓▓▓▓▓▓▓▓|	As expected.
2.5	Max Processes Reached	SYSTEM BUSY: Please Wait... SYSTEM READY: Please Continue...	As expected.