Assignment 2 Multiprocessing

Carl Humphries s5084150

Contents

1. Problem Statement

The Goal of the assignment was to create two programs server and client. Where the client would query the server and the server would use threading to calculate all the factors of that number and 31 other numbers that have been bit shifted from that number.

The server and client need to use shared memory to communicate and each factor needs to be returned as soon as it is found. Using syncronisation techniques, like mutex and semaphores it enables for no data to be lost.

The program will need to be portable and be able to work on unix using pthreads and windows using win32 api.

2. User Requirements

The following outlines the user requirements for the program:

- The user can enter upto 10 queries at a time and regardless of the number of threads on the server the queries should be handled.
- if the user enters a 11th query the program should tell the user that system is busy.
- Each query can be from 1-(max 32 bit int).
- Queries can be entered at any time meaning the program should not block the user from entering if there is other queries.
- Factors should be reported as soon as they arrive.
- If there is no reponse from the server the client should display the current progress of the queries.
- The progress bar should be non scrolling and disapear if more factors are found.

3. Software Requirements

The following outlines the software requirements for the program:

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 ‘q’ to quit.
3. The server will start up either the number of specified threads if given (see Req.17) or as many threads as there are binary digits x 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 input number is K, each thread #X will factorise K rotated right by (K-1) bits. For example say K and it has N significant bits, then 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. Rotating an integer K by B bits = (K >> B) | (K << 32 – B). CLARIFICATION: C= K << (32 – B); Rotated = (K >> B) | C
4. The trial division method should be used for integer factorisation.
5. The server is 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. Each factorising thread will report its progress as a percentage in increments not larger than 5% as it proceeds to the server primary thread. The server will use the individual thread progress values to calculate an overall progress % for each request being processed and pass this back to the client in increments not larger than 5%. This will require a second shared array char progress[10] that is 10 elements long but no handshaking protocol or synchronisation since it does not matter if the client misses some values or reads them multiple times.
13. 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 that does not scroll up (see example lab 3). 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 ie
> Progress: Q1:50% ▓▓▓▓▓______| Q2:30% ▓▓▓______| Q3: 80% ▓▓▓▓▓▓▓▓__|

A sample session with only a single query may look like this (the example is contrived)….
o Enter numbers to factor:
12345678912345
factors: 13 17
Progress: Query 1: 30% ▓▓▓______|
factors: 333 1234
Query complete for 12345678912345

14. 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.
15. The system will have a test mode that will be activated by the user entering 0 as the number to be factored. This will be passed to the server which will launch 3 sets of 10 threads each, each set will simulate one of three user requests. Each thread in each set will return to the client 10 numbers starting with the thread # times 10 and incrementing by 1, with a

random delay of between 10ms100ms between return values. For example thread #0 will return numbers 0-9, thread #1 will return numbers 10-19, thread #2 returns numbers 20-29. Progress output will be disabled during the test and it will only run if the server is not processing any outstanding requests instead a warning will be issued to the user. IF THE TEST MODE IS NOT IMPLEMENTED NO MARKS WILL BE AWARDED FOR REQUIREMENTS 9 & 11.B.

16. If the client is terminated by the user entering ‘q’ or by typing Ctrl-C (cygwin only) it will first cleanly shutdown the server and then print out a nice termination message.

17. The server will have a dispatch queue driven thread pool architecture where the thread pool size will be configured at startup by means of a command line argument

18. The source code shall be portable so that it can be compiled and run on Unix and Windows

4. Requirement Acceptance Tests

Software Requirement No	Test	Implemented (Full /Partial/ None)	Test Results (Pass/ Fail)	Comments (for partial implementation or failed test results)
1	The program will consist of a multi-threaded server and single- or multi-threaded client process.	FULL	PASS	
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 ‘q’ to quit	FULL	PASS	Forces number to be 32bit unsigned integer
3	The server will start up either the number of specified threads if given or 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.	FULL	PASS	Using thread pool shown in lab references I made it be able to work with any number of threads using a que.
4	The trial division method should be used for integer factorisation.	FULL	PASS	Doubled check that using a for loop is what it means.
5	The server is must handle up to 10 simultaneous requests without blocking.	FULL	PASS	
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.	FULL	PASS	Only blocks while user is entering this is because of a design choice.
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.	FULL	PASS	
8	The client and server will communicate using shared memory.	FULL	PASS	

Software Requirement No	**Test**	**Implemented (Full /Partial/ None)**	**Test Results (Pass/ Fail)**	**Comments (for partial implementation or failed test results)**
9	Since the client and server use shared memory to communicate a handshaking protocol is required to ensure that the data gets properly transferred	FULL	PASS	
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.	FULL	PASS	
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.	FULL	PASS	Using custom made semaphores. They are slightly altered to make coding with them for this project work better.
12	Each factorising thread will report its progress as a percentage in increments not larger than 5% as it proceeds to the server primary thread. The server will use the individual thread progress values to calculate an overall progress % for each request being processed and pass this back to the client in increments not larger than 5%.	FULL	PASS	The threads report progress from 0-100 and the main thread calculates a (double) average from it.
13	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. The repeated progress message should be displayed in a single row of text that does not scroll up.	FULL/BUG	PASS	Because of the way terminal and CMD work if there is too many queries to fit on one line it causes some weird display issues although its still visible to see what is happen.
14	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.	FULL	PASS	Returns 0 since it can never be a factor.
15	The system will have a test mode that will be activated by the user entering 0 as the number to be factored.	FULL	PASS	

Software Requirement No	**Test**	**Implemented (Full /Partial/ None)**	**Test Results (Pass/ Fail)**	**Comments (for partial implementation or failed test results)**
16	If the client is terminated by the user entering ‘q’ or by typing Ctrl-C (cygwin only) it will first cleanly shutdown the server and then print out a nice termination message.	FULL (unix) PARTIAL (windows)	PASS/Not windows	Windows doesn’t handle ctrl+c but does handle q. this was because windows is lame.
17	The server will have a dispatch queue driven thread pool architecture where the thread pool size will be configured at startup by means of a command line argument	FULL	PASS	Using a linked list que.
18	The source code shall be portable so that it can be compiled and run on Unix and Windows.	FULL	PASS	

5. Detailed Software Testing

No	Test	Expected Results	Actual Results
1.0	**Query Numbers**		
 1.1 1.2 1.3 1.4	Trying numbers in range: 1 4294967295 1234567890 2	For each case 1.1 - 1.4: a. Should query and find factors.	As expected
 1.5 1.6 1.7 1.8	Trying numbers out of range: -1 4294967296 123456789012345 2000000000000	For each case 1.5 - 1.8: a. Should query and find factors but force into range 0 -MAX_32_UINT.	1.5 is treated as largest number 1.6 is treated as 0 this is as expected but neat to see. 1.7, 1.8 get treated as smaller numbers derived from them.
 1.9 1.10 1.11 1.12	Trying non numbers: 1.1 a1 bbbb q	For each case 1.9 - 1.12: Because I am using atol it takes reads only leading numbers otherwise 0. So for all these bar 1.1 and q they should be treated as 0. Aand q should quit. 1.9 should be treated as 1 since leading 1	As expected
2.0	**Threads**		
 2.1 2.2	Server with 1 thread: - one query - two queries	For each case 2.1 - 2.2: Using thread pool archetecture and job que it should deal with the jobs as the thread finishes each job	As expected Note: longer than multithreading ~30s to ~10s (windows).
 2.3 2.4	Server with 32 threads: - one query - two queries	For each case 2.3 - 2.4: Using thread pool archetecture and job que it should deal with the jobs as the threads finishes each job	As expected
 2.5 2.6	Server with 320 threads: - one query - max queires	For each case 2.5 - 2.6: Because each thread take one job with one query there is no improvements. But should be able to handle all 10 queries.	As expected

No	Test	Expected Results	Actual Results
3.0	**Test Mode**		
3.1	Running test once with no queries	Should run	As expected
3.2	Running test once with queries	Shouldn’t run	As expected
3.3	Running test twice	Should run once and stop second time	As expected

6. User Instructions

- Start Server fist with Server <number of threads>
- Start Client
- In client type a number to be factorised
- Watch progress or type another number upto 10 max
- Press q to quit or ctrl+c unix