Ocuis - An Unnecessarily Convoluted Factorizor

Nick van der Merwe - s5151332 - nick.vandermerwe@griffithuni.edu.au

October 7, 2020

Ocius - Latin. Swifter, more rapid

1 Problem Statement

A server-client system is required where the server factorises 32 right rotated numbers and sends them to the client using shared memory.

2 User Requirements

From the user's perspective (the client) they should enter a number, then see the latest factor and the progress of each query. They should be able to enter a new query request at any time, except for when there are ten or more queries already running. In that case, a warning should pop up saying that it cannot take any input at the current moment. Additionally, once query finishes it announces that the query finished alongside how much time it took to do so.

3 Software Requirements

There should be two Cygwin compiled executables, server and client. Out of these the server should be launched first, and the user should launch the client and connect to the port. In terms of the requirements of the backend,

1. The client side should consist of either a single thread or multiple threads
2. Unsigned 32-bit integers should be inputted - that is, the unsigned int type on a 64-bit machine
3. The server will swap this into an unsigned long and open 32 threads for factorisation. Each one will be the original number rotated right - that is, if we insert 1 its binary is .....001 and when its rotated it will be 100.....000 as it pushes it right and loops around to the start. In other words, the actual base-10 value will go $1 \rightarrow 18 * 10^{18}$
4. This will be factorised with the trial division method, and not the prime factors. It will return the actual factors and find them by finding the modulus less than the input of the input.
5. The server should be able to run 10 queries at once (320 threads)
6. The client reports responses instantly and when a query is complete the time will be reported
7. Shared memory will be used between the server-client to manage the locks and conditionals. In the actual requirements, this is stated twice in two different ways. This solution uses a mix of both This handshake protocol will go into detail later
8. The server is not allowed to use a buffer - it must be passed right away
9. Originally the requirement is that the progress bar updates 500ms after an update from the server, but even with very large numbers this never occurs. Instead, the progress bar will update every 500ms. That should show both the requirement of measuring time and the progress bar, while being neat

4 Software design

This time we used C++, but only for classes and some STL.

Diagram and overall algorithms

Earlier it was mentioned that the handshake protocol was replaced with a custom method. First, the original requirement is either redundant or contradicting - it was requested that integers are used to handle the read/write, then later down that mutexes and conditionals are used instead.

This solution uses mutexes for the read/write, but integers as the conditionals. Due to the fact that conditionals are in fact built for buffered systems - or at least, every documentation example used a queue style structure - there is no wait on on the write side for when data is ready. Instead, a boolean is used to mark that the slot has been written into or read.

Additionally, there is the tidbit of saying whether a slot is in use or not. For this, it was decided if a slot is holding 0, it means that it is not in use. While zero is a factor of zero, the trial division method would in fact crash if that input is given as the modulus operation would get called on a zero. So to notify the client that a socket is done, its set to zero.

However, the client is single threaded and does not actually rely on this to find out that a slot is inactive. Every loop it searches over all the sockets for ones that are not zero and have their flags set that they need reading.

The easiest way to communicate the overall algorithm is in a small diagram.

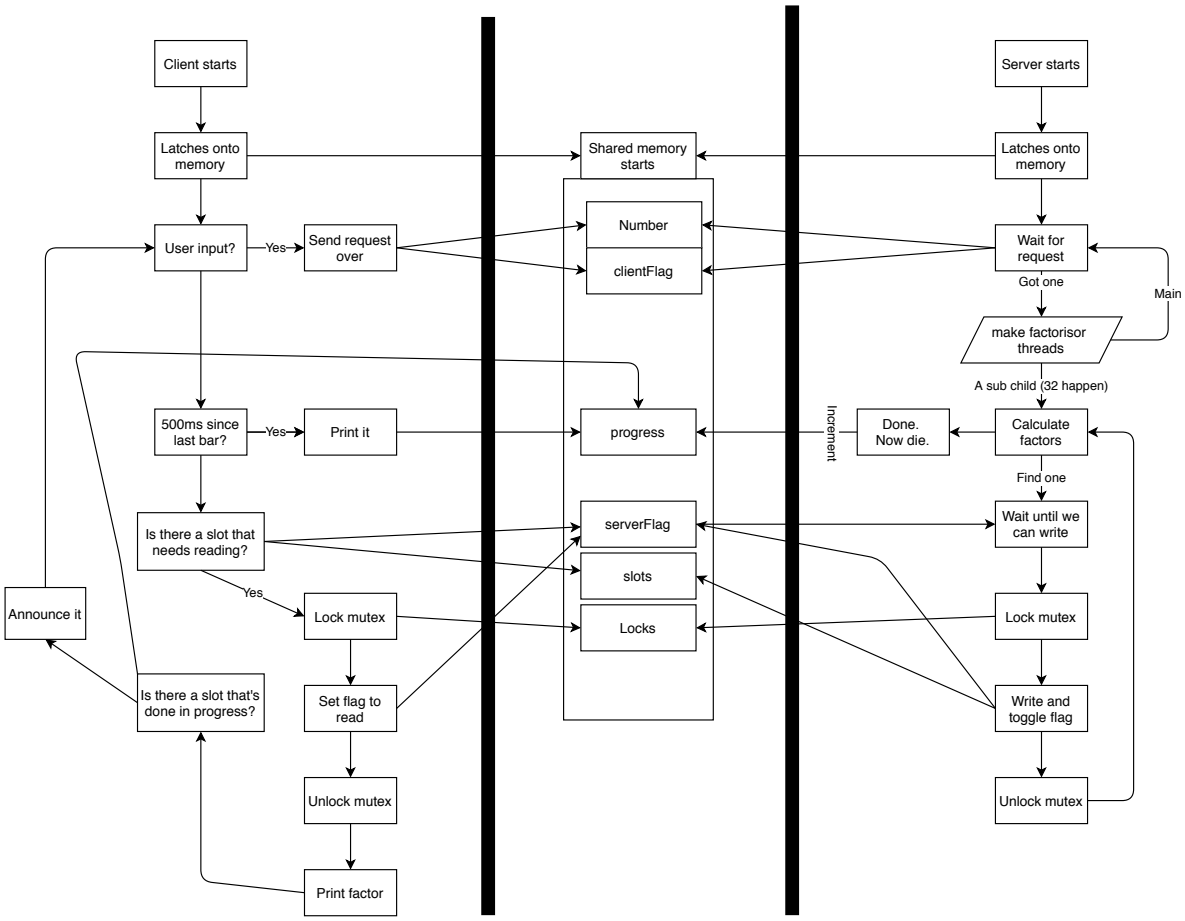

Figure 1: A small model of the program

In terms of algorithms, there are no overly complicated ones. For clarification though, this is the method used for factorisation:

```
def printFactors(number):
    for i in range(1, number): # inclusive
        if i % number == 0: # if it evenly divides
            print i # its a factor!
            # Well, we actually write to memory
```

Figure 2: Pseudo code for how factorisation is done

Client

Clock class

This is basically just a custom clock class so that we can start a timer and finish it very cleanly from the outside perspective

1. void startTimer()
2. bool isRunning() // returns a boolean for whether the clock was actually set
3. double getSecondsAndDisable() // disables the boolean and calculates how many seconds passed since it was started
4. private: timespecs and isCounting (whether its counting)

Shared memory class

This is so *all* of the mutex operations, read and write function from the outside. Essentially, the idea is that you can just call readSlot() without doing any odd mutex operations from the outside perspective. On the client side here, it consists of:

1. SharedMemory() - the constructor. This latches on, or creates, the shared memory for us
2. SharedMemory() - the destructor. Since this is the client side, we don't bother calling shmctl() or freeing our mutexes - that's the server's job when it shuts down. We just disconnect from the memory.
3. requestReservation() - This handles finding a reservation for us, it waits until the clientFlag says we're allowed to write, then does so.
4. grabReservationIndex() - Since the server can take a second to handle reservations, this is managed separately. It waits until the flag is set for reading, then grabs the reservation index. if it returned a value outside of [0, 9] then it tells the user that all the query slots are in use.
5. readSlot() - This sets our lock, waits for the read flag to be set (however, it is only called when the read flag is set. This is a safety precaution)., reads the slot and sets the flag that it was set , then unlocks the mutex and returns the value it read.
6. Subclass - shared data. This is the actual structure that exists in the shared memory. It contains everything we need - number, clientFlag, slots, progress, serverflag, and locks.
7. Private things - key and shmid. These are to initialise the memory and its position

Functions

1. startingMessage() - This just prints the title screen and a number of newlines so that our escape character for moving up a line is functional
2. readAnyNewFactors() - scans over the slots if anything can be read, and reads them. Reads one from every slot (if one is there)
3. clearWaitForQuery() - Clears the warning message for waiting for a query
4. sayQueryFinished() - Announces a query finished and how long it took to finish
5. showProgressBar() - This manages printing the progress bar by checking the progress variable in the shared memory
6. printWaitForQuery() - prints the warning message when a user attempts to type in a request for a query when 10 are already running
7. handleAllPrints() - Runs through all the print functions for you. Requires a timer variable of the counts of all the timers for the sayQueryFinished()
8. Main - Thanks to using select on our stdin to check for input, this got complicated. Essentially there's a select function which constantly checks if the user typed in any input, and then calls handleAllPrints() every loop to update all the information.

Server side

SharedMemory

This is mostly the same as the client side of shared memory, however it has a few different functions:

1. SharedMemory() - the same as the client side, except we initialise the mutexes
2. SharedMemory() - the same as the client side, except we free the mutexes and call shmctl() to delete it
3. setToDefault() - Between runs it can occur that it just reuses the same shared memory as the previous client/server. This is a safety check that's enforced at the start of the program - it sets everything the default values again
4. waitForReservation() - This waits until a flag is set by the client that a new query should start, and handles allocating it
5. writeSlot() - This locks our mutex, waits for the write flag to get set, sets the write flag and writes the factor into the slot, and then locks the mutex again. We also have a 100 micro second pause here to slow down the program
6. SharedData class - the same as the client
7. Private variables are the same too

Factorizor

This class handles all the threads and factorisation internally. Essentially we throw in a number and a memory index, then can go back to waiting for queries on the main thread.

1. Factorizer() ; ctor - This launches up 32 threads and specifically detaches them; that gives them permission to shut down on their own. We also have a sleep between each thread creation, because the data passed through *pthread_t* can have problems if it is too fast
2. struct ThreadData() - This is the information each thread needs; which socket to write to and the number it needs to factorise

3. static void * factoriseThread - This specifically needs to be a static function, without that inside of a class, pthread_t has major problems. Those problems are also related to why the shared memory variables - flags specifically - were not just handled internally in the class.

Functions

These are minimal compared to the client.

1. manageQuery() - takes in an index and launches up our factorizor
2. startingMessage() - same as the client, except we don't have a number of newlines
3. main() - We set our shared memory to default then wait for queries and call manageQuery() for each one

5 Detailed Software Testing

Everything works.

Requirement	Test	Implemented	Pass/fail
Multithreaded server	It factorises multiple at a time	Yes	Pass
Factorisation is correct	Check the prints	Yes	Pass
Server is non block (up to 10)	Run two or more queries	Yes	Pass
Client is non blocking	Run more than one query	Yes	Pass
Shared memory is used	Read implementation	Yes	Pass
Client-server synchronisation	Doesn't get stuck and no missing factors	yes	pass
Non buffered output	Read that no queues are used	yes	pass
Semaphore implementation	Read code; check task manager that the system is being used	yes	pass
Correct server thread synchronisation	No factors are lost; task manager is active	Yes	Pass
Progress reporting	Read prints	Yes	Pass
Displays results correctly	Read prints	Yes	Pass
Server can run up to 10 queries at once	Run 10 queries	Yes	Pass
Progress bar	Read	Yes	Pass
Warning for when there are 10 requests	Try 10+ requests and read	Yes	Pass

Figure 3: Basic tests that it functions

Test	Expected output	Output
Number too large	Does nothing	Does nothing
non-digit input	Does nothing	Does nothing

Figure 4: Basic tests that it functions

6 User Instructions

Compile with:

```
g++ server.cpp -o server -pthreads -O3
g++ client.cpp -o client -pthreads -O3
```

and run

```
./server
./client
```

Warning: Due to terminal differences and how specific some of the macros I use are, the output might not be as pretty for you as it is for me.