Multithreaded Client Server System

Advanced Software Development (7420ICT)
Hallvard Westman (s2873575)

Contents

Problem Statement

The objective of this assignment is to write a multithreaded client server system for multiprocessing.

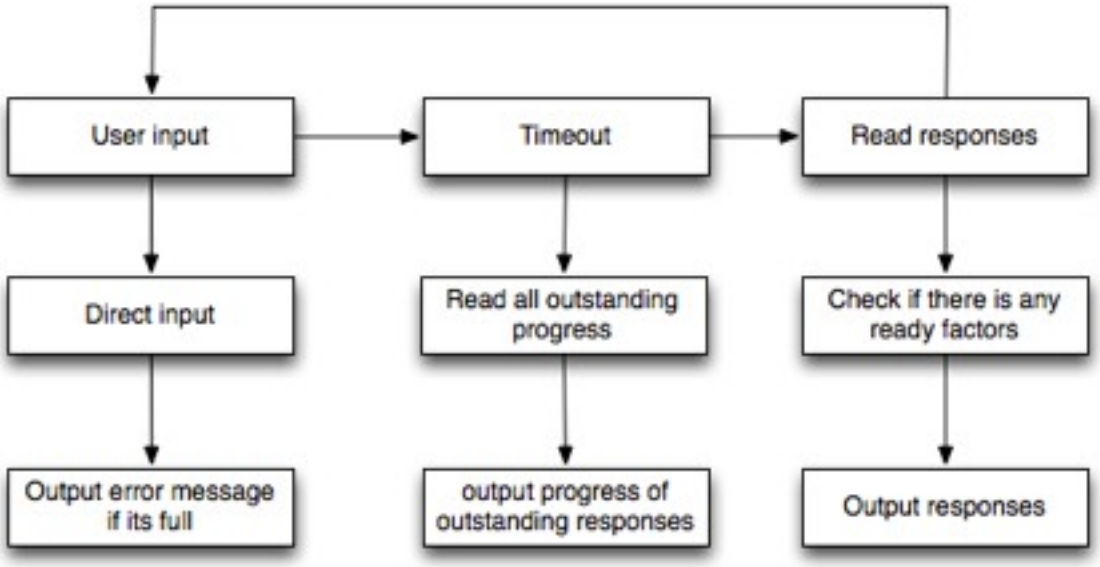
The client will take user input that will be passed to the server for processing. The processing will consist of rotating the bits of the user input and factoring each result from the rotation. These factors will be passed back to the client.

User Requirements

The following outlines the user requirements for the program:

- 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.
- 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.
- The system will have a test mode that will be activated by the user entering 0 as the number to be factored.
- If the client is terminated by the user entering ‘q’ or by typing Ctrl-C it will first cleanly shutdown the server and then print out a nice termination message.
- 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)
- When the server has finished factoring 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.
- The client is non-blocking. A 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.

Software Requirements

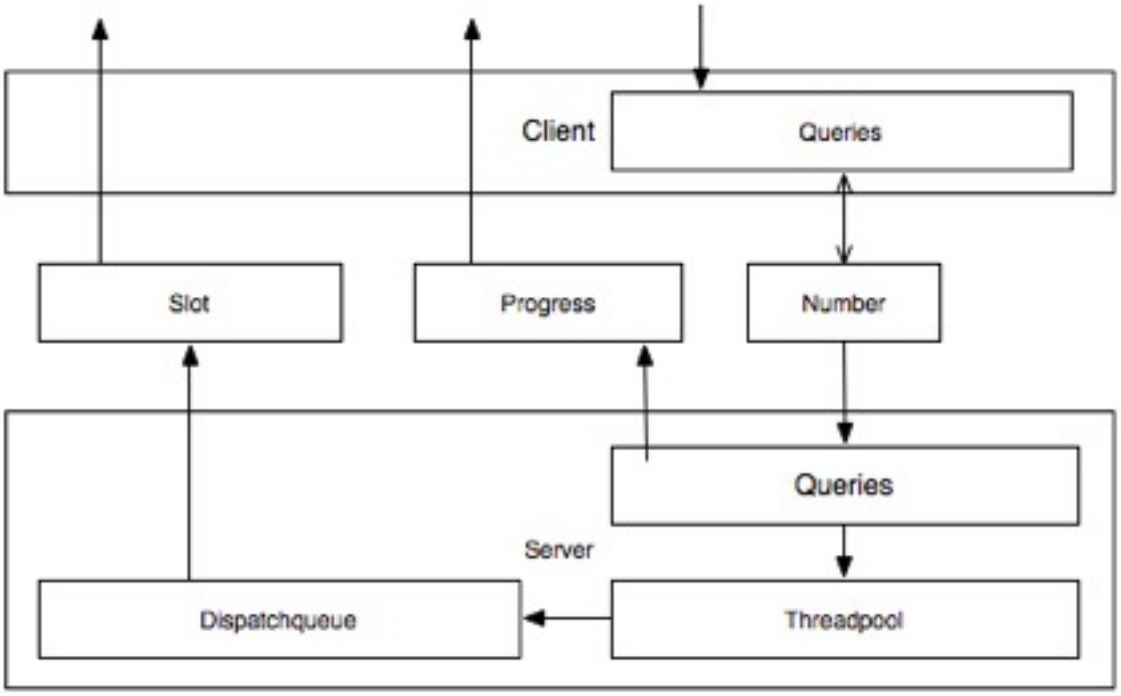


The following outlines the software requirements for the program:

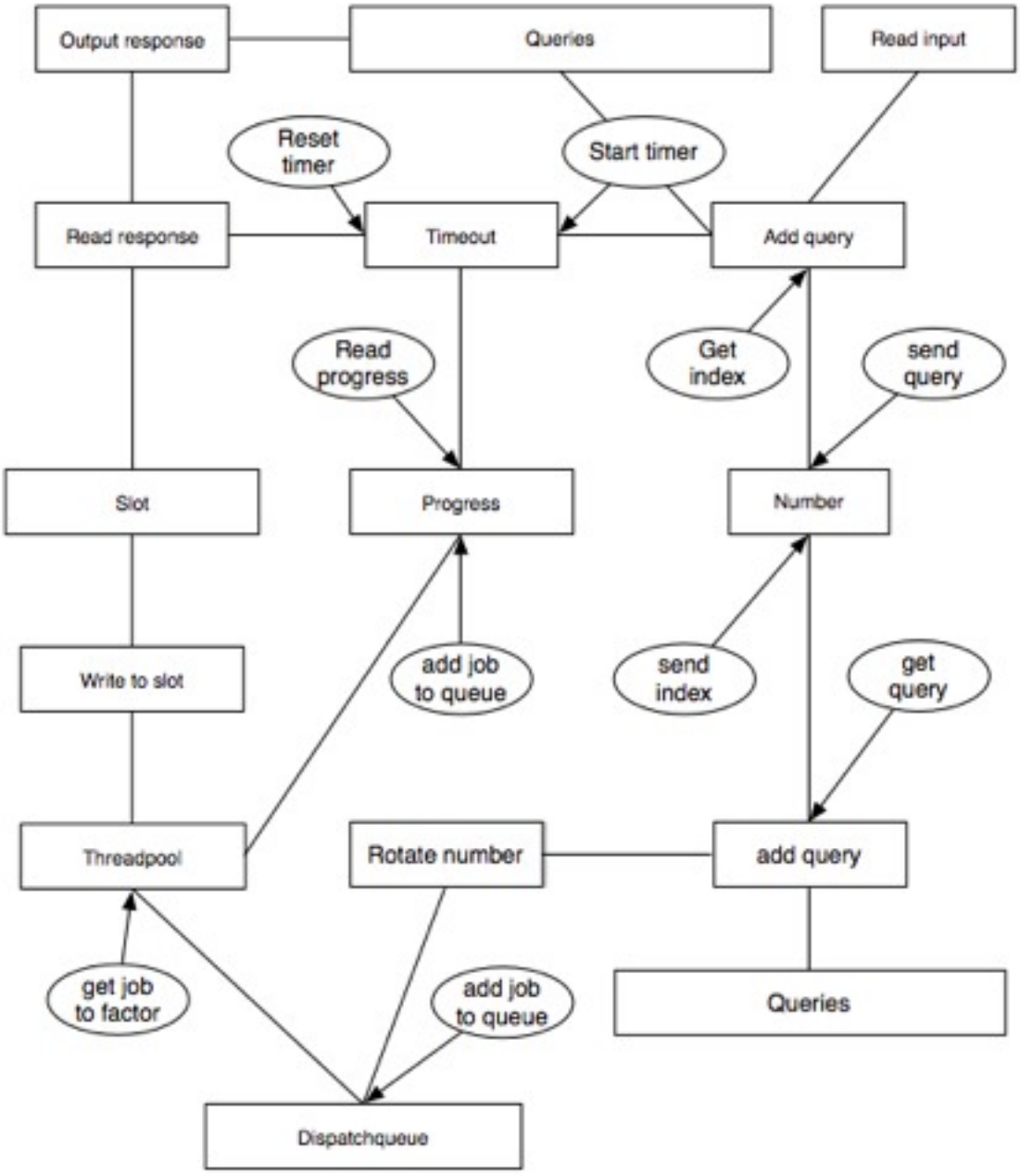
1. The server will take each input number (unsigned long) and will start up either the number of specified threads if given (see Req.17) or as many threads as there are binary digits (i.e. 32 threads). Each thread will be responsible for factoring an integer derived from the input number that is rotated right by a different number of bits.
1. The trial division method should be used for integer factorisation.
2. The server is must handle up to 10 simultaneous requests without blocking.
3. The client and server will communicate using shared memory.
4. A handshaking protocol is required to ensure that the data gets properly transferred.
5. The server will not buffer factors but each thread will pass any factors as they are found one by one back to the client.
6. Access to the shared memory slots needs to be synchronised so that no factors are lost
7. Each factoring thread will report its progress as a percentage in increments not larger than 5% as it proceeds to the server primary thread
8. 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
9. The source code shall be portable so that it can be compiled and run on Unix and Visual Studio

10.Software Design

High Level Design – Logical Block Diagram



Structure Chart



List of all functions in the software.

int main(int argc, char *argv[])

* Forks a process to start the server process, and runs the startclient function
* on the main thread

void initiateSignals()

* Initiates signals for IDE-quit, ctrl-c quit and terminal quit

void eventHandler (int sig)

* Sets the global quit flag signaling processes and threads to exit,
* takes the signal recieved as param in order to perform potential actions

void startClient(sharedMemory sm)

* Initiats the main loop of the client that is taking non-blocking input and
* outputing responses from the server

void writeToNumber(int req,int *number,char *clientFlagPtr,int &nrOfOutstandingQueries)

* Writes int to shared variable if its available for writing

void readResponse(outstandingQuery *queries,int *slot,int &nrOfQueries,char *serverflag,timeval &timer)

* Reads responses from the server according to active queries

int getInNum(char *input)

* Takes a string and checks if it is the quit and returns -1, else it converts
* input to int,

int getKBHitString(char *string,int &strLen)

* Checking if the user is done inputing a string, its delimited by CR
* should be used after the kbhit function returns true and unread input is
* residing in the buffer
* If input is not CR, it appends it to the parameter string and incrementing the strlen

int kbhit()

* Checks for keyboardhits, returns true/false if it has occured or not

void nonblock(int state)
* Setting the terminalstate to non-canonical, reading input imidiately from
* console imidiately

float TimerStop(timeval &timer)
* Takes a timer structure and returns the time passed since the given structure
* was initiated in milliseconds

void checkTimer(timeval &timer,int *progress)
* Takes a timer structure and a pointer to the progress array.
* it checks if the time since given structure was initiated has passed 500 ms
* if so it passed the progress structure to the printprogress function

void printProgress(int *progress)
* Takes the progress array as parameter and prints out the progress for each
* outstanding query

void printBar(int percentile)
* Prints a progressbar from 0-20 based on a percentile given as param

void startServer(sharedMemory sm,int poolSize)
* Starts up the server side of the application, reading queries in a continous
* loop checking for a new querie. writing progress to the shared progress
* array for each iteration as well

void *slave(void *arg)
* A function given as parameter to start up new threads that
* executes jobs in the dispatchqueue

void factoriseNumber(job *theJob)
* Factoring the number given in the job structure, and sends results to
* the global queries based on the indexes in the job struct

int getPercent(int t,int b)
* Returns the percentile of given parameters

void initiatedSharedVariables(sharedMemory *sm)
* Initiates all the shared memory need to communicate between server/client

void releaseSharedVariables(sharedMemory *sm)
* Releasing the shared memory reserverd in initiatesharedvariables, also
* releasing mutex and conditionvariables with corresponding attributes

void getSharedVariablePtrs(sharedMemory *sm,sharedMemoryPtrs *smptrs)
* initiates the given smptrs structure with pointer to the shared memory
* of the sm structure

List of all data structures in the software.

Only the non-selfexplanitary functions are documented

struct outstandingQuery
*A structure for the outstanding queries clientside
struct job
* A job to be picked from the dispatchqueue and executed by the threadpool
struct sharedMemory
* Structure containing the identifiers for the shared memory
struct sharedMemoryPtrs
* Containing the pointers of the shared memory

class SemLock{
* A lock object synchronizing access to the shared memory
private:

```
pthread_mutex_t m;
pthread_cond_t c;
bool running;
int *slot;
char *serverFlag;
void Lock(){ pthread_mutex_lock(&m); }        // <- helper inlines added
void Unlock(){ pthread_mutex_unlock(&m); }
```

public:

```
SemLock();
void setSlot(int *aSlot,char *aServerFlag);
```
* Initiating the lock with the pointer to the shared memory
```
int write(int factor);
```
* Writing to the shared memory
```
bool isRunning();
void startRunning();
void stopRunning();
void killIt();
```

};

class ThreadPool{
* A standard threadpool with a dispatchqueue initiated with poolsize and locks to synchronize threads access to the dispatchqueue. The access to add items is not synchronized due to the fact that there will only be one thread adding items.
private:

```
bool running;
pthread_t **threadPool;
pthread_mutex_t lock;
pthread_cond_t  condition;
 job dispatchQueue[QUEUESIZE]
```

```
        int poolSize,queueCount,queueStart,queueEnd;
        void Lock(){ if(pthread_mutex_lock(&lock)!=0){cout << "i cant lock this
                                hsit"<<endl;} }          // <- helper inlines added
        void Unlock(){ pthread_mutex_unlock(&lock); }
        void popJob(job &j);

    public:
        ThreadPool();
        ~ThreadPool();
        void initialize();
        void setPoolSize(int size);
         void pushJob(job j);
        * Adding a job to the end of the queue
        void destroyPool();
        * Destroying the pool by joining the threads and destroying the locks
        void executeJob();
        * Picks off the oldest job on the queue(from the start)
        void stopRunning();
        bool isRunning();
};

class Queries{
* A class containing a semlock object for each slot in the the shared memory,

private:
        SemLock queries[MAXQUERIES];
        int nrOfQueries;
        int testRunCount;
        int getAvailableIndex();
        int mainProgress[MAXQUERIES][INTBITS];
        void rotateAndFactorize(int number,int index);

    public:
        Queries();
        int sendFactor(int index,int factor);
        int addQuery(int number);
        int endQuery(int index,bool testmode);
        void initiateSlots(int *slots,char *serverFlags);
        void writeProgress(int *progress);
        void updateProgress(int index,int jobIndex,int percent);
        void stopRunning();
};
```

11.Requirement Acceptance Tests

Software Requirement No	Test	Implemented (Full / Partial/ None)	Test Results (Pass/ Fail)	Comments (for partial implementation or failed test results)
1	Multithreaded server implementation	Full		
2	Correct factorisation	Full		
3	Non bocking server	Full		
4	Non blocking client	Full		
5	Shared memory use as specified	Full		
6	Client – server synchronisation	Full		
7	Non buffered server output	Full		
8	Semaphore implementation (non-busy waiting)	Full		
9	Correct server thread synchronisation	Full		
10	progress reporting by server	Full		
11	progress display by client	Full		
12	Test mode operation	Full		
13	Ctrl-C handling / clean server shutdown	Full		
14	Dispatch queue driven thread pool architecture	Full		
15	Compiles and Runs on windows using win32 api	Full		

User Instructions

Files required:
*** common.h**
*** Headers.h**
*** SemLock.h**
*** Client.cpp**
*** Init.cpp**
*** main.cpp**
*** Makefile**
*** SemLock.cpp**
*** Server.cpp**

On unix(Cygwin)
initiate shared memory;
/usr/sbin/cygserver &
CYGWIN=server
make
./program.exe <size of pool, default 32>
On Windows with Visual studio

* add dependencies: pthreads (files reside under pthreads dir)
    * add .dll to /C/windows
    * add .lib to visualstudio/VC/lib
    * add headers to visualstudio/VC/include
    * add lib to project dependencies under projectprop->linker->output->additional dep
* Open project assignment_2 from the win/assignment_2 directory with visual studio and build, arguments are set in visual studio options

General instructions for usage of application

Input a number to rotate and the enter key to add a query.

This will imidiately start outputing factors to the screen ,you will still be able to input another number to rotate, each keystroke will be recorded and just press enter to send the number.

In order to initiate testmode, input 0 then enter

In order to quit at any time just press the letter q then enter.