Allakhverdiyev, Dmytro - dallakhv
Darling,Bailey - bdarling
Li, Wilbur - wiwili
CMPS 109

Project Phase 2 Report

Building with G++

Like before, building the programs with G++ requires preprocessing, assembling into object files, and linking. G++'s terminal command shorthand for all of these tasks is g++ (all cpp files) -o (output) -std=c++(ver) -phreads (for POSIX threads). This can be time consuming for large projects such as this one, so a makefile is used instead. Makefiles build each object file individually and links them all at the end. The advantage of this is that individual object files that are already built but not updated do not need to be rebuilt, significantly reducing compile time. A makefile was provided for both the client and server of the misvm, each in their respective folders. To run the server, type ./server_misvm <port> in the terminal and to run the client, ./client_misvm <ip address> <port> <filename.mis>.

When Valgrind is run on the client from Ubuntu, there is a still reachable leak of 72,704 bytes in 1 block. At first, this appears to be a potential cause of networking constantly giving “Error on accept: invalid argument” on attempt, However, this appears to be a known issue as the Valgrind documentation itself states that this is not a bug, but rather a feature, due to the many implementations of C++ standard library having their own implementation of memory pool allocation. In this case, it appears that this memory pool for certain destructed objects are not immediately freed and returned to the OS. Since this is the case, Valgrind reports it as a still reachable leak as the pools were not freed on program exit.

Design

One of the big decisions early on was the choice between TCP and UDP. Both protocols have their advantages and disadvantages, as well as their own use cases. For this project, we decided very early on that TCP would be the way to go. TCP has the advantage of reliability over UDP, where TCP will establish a connection and guarantee that packets get sent and ordered properly unlike UDP's unordered, non-guaranteed packets. In a situation such as a machine instruction simulator where programs are run sequentially, it is imperative that lines are processed in the correct order with nothing missing, otherwise unexpected results will occur. The disadvantage that TCP has of being somewhat heavy with a lot of overhead is a non-issue in the scope of this project. UDP's advantage of being faster is more applicable to something like an online game, where the speed of data transmission is more important than the reliability. In the case of the project, we are mostly dealing with the processing of text, so it is not as important.

Single-Threaded Networking

As explained earlier, we are utilizing the TCP protocol for implementing networking. To accomplish this, the local misvm was split into 2 components, the server and the client. The client will handle accepting the file, sending it to the server, and waiting

for responses to output to the respective files. The server accepts the file and processes the code much like how it used to; loading the code into the parser, preprocessing labels, then executing command sequentially. The individual design aspects will be detailed below in respective sections.

Single Client

Implemented by Wilbur using the TCPSocket helper classes provided by teaching staff. As The client has very simple implementation. It accepts arguments for IP, port, which in this case is 7777 hard coded by the server, and an mis file. Like the local machine misvm, it will check for argument count, file extension, and save the file name. This file name will be used for output of <filename>.out and <filename>.mis. Following that, the client will send the entire file over to the server by first getting the file size, allocating a buffer of that size, reading the file into the buffer, and sending it over to the server. This was where the first design decision came in. We could either send the whole file over, or send it one line at a time. Sending it one line at a time would have more safety because the maximum packet size of a TCP packet is 64k, but would require some reworking of our parser implementation. But, since we are working with text files, the size is realistically much smaller. As such, the decision was to send the entire file over.

Following the sending, the client will then wait for a response with timeout. This means that if the client does not received data for a set time, which in this case is 60 seconds, the socket will timeout. When a packet is received, the buffer is then printed to the respective file. How the client knows which file is that the server prepends information, either ERROR or OUTPUT before each buffer. This will be further detailed below in the next section. When the client receives a return byte of 0, it means that the process has ended and the client breaks out of the loop, cleaning up and ending the program.

Single Server

Implemented by Wilbur using the TCPSocket helper classes provided by teaching staff. One thing to note, however, is that for some reason the server gets invalid argument shutdowns quite frequently on the same compiled executable with the same command from the client. The root cause of this was unknown, and the way to get around this is to just reboot the server and keep trying until it goes through. For the port listening, the instructions say this:

"To execute this program you need your MIS server daemon to be running on the local or remote node listening on a specific network port"

This is pretty relatively ambiguous and left me unsure of what port to listen on. When the statement specific is made, it implies either that it is one specific port, and one specific port should be hard-coded into the server, or that it can only listen for one port at a time. However, since the instructions below that say that client should be called through ./client_misvm <ip address> <port> <filename>.mis, it is assumed that the port is a variable. As a result, the server takes an argument of port when called through the console, ./server_misvm <port> and it will listen on that port.

In terms of process, the server did not change much from the local machine routine. The server generates a TCP socket and accepts a client. Then it accepts a file, which in this

case the maximum buffer size is half of the maximum TCP socket size processing it in the parser much like the local machine version. The rest is nearly identical with a few differences, the biggest being the logger. Instead of the main server modifying the logger's file name, it passes the logger a TCPSocket pointer, which it will then use to write to the proper buffer and send messages back to the client. These messages are prepended with either ERROR or OUTPUT, signifying where the client will output its data. These prefixes are intended to be shedded by the client before outputting. In addition, these messages have a small delay after the socket sends them, which allow the client to order the messages properly, at least on the Ubuntu loaded laptop this was tested on. This may change for other machines. The reason why only the logger was changed was to reduce the code rewriting that needs to be done, but this implementation fails in multi-threading as it was soon figured out when the server was converted to handle multiple clients, detailed further below.

It is strange, but the error “Error on accept: invalid arguments” occurs somewhat randomly. Somehow on some calls it is fine but others will output this error when the terminal command is identical. This seems to stem from maybe a new socket being unable to be created or a socket was not being freed properly, but I was still unable to pinpoint where this error comes from, as viewing the code for TCPServerSocket's getConnection() showed where the "Error on accept" error came from but not where the invalid arguments part. Using Wireshark to try and find the cause did not yield any results, and the belief that the leak detailed in the sections above would be the cause led to the result that it was not the issue. This issue persisted even in the multi-threaded server implementation. To bypass this, the only (un)reliable solution is to just run the server and client multiple times until it connects.

Multi-Threaded Networking

The next part of this phase is to convert the server into a multi-threaded server, handling multiple clients transparently. While the client itself had very little changes, the server had to be overhauled in several ways, mostly with the implementation of a thread manager and the way outputs were handled.

Multiple Clients

As explained earlier, the client itself did not undergo many, if any changes. The changes were made mostly to the back-end server, making it completely transparent to the end-user. As such, there will not be any explanation beyond this.

Multi-Threaded Server

Implemented by Wilbur utilizing the classes and information provided in the lecture slides. The operation of said classes will be explained, clarifying that they were not just blindly copied from the slides. As provided in the slides, there are 3 classes. The main server, a connection class, and a garbage collector class. Each of these classes comprise a sort of thread manager, instantiating new sockets when needed and cleaning up old sockets when finished. Discussion will begin from the changes made to server code, connection, garbage collector, and finally the main server.

Starting out, the changes made to the server itself. The major change to the server was the way outputs will be handled. Early on in testing, it was quickly realized that the static logger class did not work. When multiple clients tried to access it, each client would "hijack" it, so to speak. This is because there is only one logger, and it held information for only one socket at a time. There were two solutions that arose for this problem implemented into the project for demonstration, but either one will work, with preference leaning towards the latter.

Both solutions required having a pointer to the socket somewhere in the code, either as a passed parameter or stored in a private member. This requires the changing of every class object either during instantiation or during its execution step. For this implementation, the choice was to store the information as private members with the storage and factories being passed with the socket information and generating objects with initialization of socket information when cloned. The other option of being passed as parameters may be more or less viable, but it was not explored in depth as this implementation seemed to work without changing too much of the code. I am unsure of whether or not this implementation needs a mutex or not or how to even implement this, but in testing it appeared that it did not, though the logger would definitely be a critical section.

The first solution was to eliminate the logger in its entirety and pass the socket information to the instantiation of each class, where they will store the pointer as a private member and write to each socket individually. This is very cumbersome and required rewriting all of the logger output lines and doing some other string, char, variable appending conversion to get more detailed output, as well as including delays individually if necessary.

The second solution was to change the signature of logger to accept a TCP socket as a parameter. This also required going through every class and changing lines that invoke the logger to accept 2 parameters instead of 1, the string and the socket information. I believe this is considered thread-safe from what was tested, as the logger does not need to change any variables and simply just writes to sockets sequentially as it becomes available. The advantage of this is that many of the classes do not have to have their logger messages and commands entirely rewritten, just add a socket information parameter following.

During the combing of each class, some of them such as the DataTypes, have had some minor revisions done on them for the storage of invalid data, but they are nothing significant to the program logic so they will not be explored in detail.

The next class to be discussed is the connection class, which inherits from the thread class also provided by teaching staff, represents a single connection. It holds information for the thread's main body and a TCP connection, as well as a pointer to the next connection for the garbage collector. This is where the main body of our program, our server, will be included. In other terms, this class represents one instance of the single-threaded server in a multi-threaded environment. In terms of main body, the TCP size stated in the single-thread version was intended to be changed to 1024*1024*10 like the default values of getConnection(), but the program gets segmentation faults when trying to run with those sizes, so it was left as is.

The garbage collector is a class that manages the TCP connections, holding them in a linked list and cleans up expired connections as they come. Its private members represent

said linked list, with pointers to the head and tail of the list. Functionally, it is like a thread manager. It accepts new connections and has functions for cleaning up expired connections, and waiting for all threads to finish before cleaning all of them up, somewhat like a barrier.

Finally, the main server class brings everything together. The server accepts a port from argument, creating an instance of TCPServerSocket with that port, and an instance of the garbage collector. There follows a main loop, where it waits on connections and when one is made, it will invoke the garbage collector to cleanup any expired connections, create a new connection object with the created socket, start the connection thread, and add said connection to the garbage collector where it will be cleaned up when finished. In the main loop, if the connection returned by TCPServerSocket's getConnection() is NULL, then the loop terminates, memory freed, and server shuts down.

The loop was changed to an error output and the loop continues for testing purposes because of the "Error on accept: invalid arguments" error that kept arising, although shutting down and restarting the server seems to be the solution sometimes. It would be xpreferred to have the server survive errors and clean itself up properly on exit, but the cleaning up on exit part could not be verified properly for some reason.

Multi-Threading MIS

Implementation by Bailey. The current implementation of multi-threading has the connection, the main body of a single thread, going through the code and parsing the information until a THREAD_BEGIN is reached, in which case a vector of code is generated until THREAD_END is hit. This line of code is then passed to the storage, where it is saved until use later by the expression execution itself. When execution starts, the parser is set to the line where THREAD_END is, to skip the thread and continue down the main loop. The thread body code will then start a thread using that segment of code. Currently, this implementation does not support LOCK, UNLOCK, and BARRIER.

Another implementation was attempted by Dmitry before his laptop died (faulty PSU) over the course of the project. This approach was to have execution sequentially until THREAD_BEGIN is hit. When execution of THREAD_BEGIN commences, the execution loop will append strings to its private data member until THREAD_END is hit, then start a thread. This thread will then generate a new parser and expression factory with that data and execute the instructions until the end of its parser, being THREAD_END. As this implementation was incomplete at time of laptop destruction, it does not support LOCK,UNLOCK, and BARRIER.

An alternative idea thought of by Wilbur in hindsight took a slightly different approach. This was not thought up of until after attempts at implementation have already been made, so time was quite limited which prevented testing or validation of this method. It is significantly more time consuming as it requires rewriting storage, parser, and any instruction that utilizes labels, like the jumps, in addition to the new functionality it needs.

The rewriting behind this is to move all label storage and functionality to parse, which it should have been in the first place, as they are commands more closer to the parser rather than the storage. The parser will store labels within its own context, addressing the issue of scope that plagues threads. In addition, there needs to be a default constructor that

creates an empty parser. As a result, all code that interacts with storage's label will need to be rewritten to interact with the parser instead.

Functionality-wise, there would need to be a thread management class managed through storage and a queue of parsers in storage. How instruction will go is is as follows. The single thread of connection will parse through the code for the first time, looking for LABEL and THREAD_BEGIN. LABEL will be saved as usual, but in the parser instead of storage. When THREAD_BEGIN is found, a new empty parser will be generated and individual lines of code will be fed into this parser until THREAD_END is reached. Then, the new parser is handed over to the storage, where it will be saved to a queue of parser pointers. This is repeated until the code is finished parsing. At this point, normal code execution begins. When THREAD_BEGIN is reached, operation of this expression is as follows.

The THREAD_BEGIN would inherit from thread and have a member for holding the storage pointer, with initialization and cloning being similar to other expressions, and a different constructor for starting the thread. The function definition for threadMainBody would generate a new expression factory, as it was unable to get the main one into its scope, the passed storage, and a parser that was popped off of storage's queue of parsers. Then, the execution is much like a main thread, where it will look for labels and execute sequentially. This addresses the issue of scope, where the labels outside do not exist and labels inside are not exposed to the outside. The execution command would pass the instantiated object back to storage, where it will then pass it to a thread manager class, store the thread object, and start the thread, skip the code in the main parser until THREAD_END is reached, and have execute return NULL so this object does not get deleted in the main loop.

The thread manager class is where BARRIER can get implemented. A BARRIER instruction will invoke the thread manager class through storage and run its own barrier command until all threads are terminated before returning control to the main program. This will only work if the idea that the THREAD_BEGIN objects can be passed to

Unfortunately, the part where this idea slightly falls apart is LOCK and UNLOCK. In honesty, I still do not understand how mutexes work with members of data structures, so this implementation may not work at all. The storage of variables in the map would have to be changed to be a pair of mutex and data. Then, when LOCK is invoked, if the variable exists in the map then lock is called on that mutex, locking the data from access except by that thread. Likewise, unlock will unlock the mutex allowing other threads access to that data. This idea was thought of too late and with a significant lack of time, there were no attempts at implementation.