CS109 MIS Phase I Report

Alex Ou aou4 1441135

Andrew Guterres aguterre 1416340

Max Zhao mlzhao 1466223

To build the system, simply use the makefile provided at the root directory of the project. Compile using the command “make” and run with the command “./main test.mis”. To clean and remove objects and executables, write “make clean”.

The system is divided into two main hierarchies, with a small parser class various maps/vectors to hold and transfer data. The main hierarchy is the Instruction class hierarchy. The instruction class is an abstract base class that contains three important pure virtual methods: clone, initialize, and process. Clone calls instMap which holds string for key and pointer to Instruction object. When the parser parses the first string of the instruction, instMap will instantiate a new object of that type. Clone works as a middleware factory so we can allow for multiple operations on the same Instruction object in the map since we are cloning the map before we operate on it. Initialize parses the input file and initialize our Instruction object. They initialize the appropriate protected variable within our Instruction object. Process does exactly what it sounds like: execute the function that each Instruction is suppose to do. For example, the process method of the Add instruction will add all the parameters and store the result in the first parameter, which happens to be a variable. Sub will do the opposite, subtract third parameter from second parameter and store the result in the first parameter. The list goes on, and each instruction does it’s own job. The Instruction hierarchy breaks down like so:

Instruction

Add	Sub	Jmp	JmpZ	Ssc	JmpGT	Assign	Out	Sleep	Var
Mul	Div		JmpNZ	Gsc	JmpGTE				
					JmpLT				
					JmpLTE				

In the beginning of our design, we initially wanted to group together instructions based on their operations. This gave us the same hierarchy as above, but that would lead to JMP instructions being linear in inheritance. So we instead decided to break the hierarchy down mostly according to the number and types of parameters each instruction takes in. Add and Mul both take in 3-13 parameters, making it so that we can use the same initialize function for Mul as we did for Add. The same concept extends to Sub and Div in that Div uses Sub's initialize function. Many of the jump instructions follow a similar design since they take in the same number and type of parameters.

For our function/method implementation, we first started out with a brute force, messy way of figuring out the parameters. We would, for example, initialize 13 parameters in add and mult instead of using variadic functions. We did this to get the skeleton of the hierarchy and how each instruction would work together. After we broke down the hierarchy by parameters and type of instruction, we then worked out what functions and methods we could share, and what functions and methods should be private to certain classes. For example, the children of the intermediate classes would not taken their own initializers and would have to call their parents initializers since they take in the same parameters. We did this because each parent and child constructor take in the same amount of parameters and the same type of parameters. This would allow for less redundancy, but still allow us to have separate processes. Jump functions also had their own separate functions that allowed us to get the parameters we needed to test for different error cases in the assembly instructions, such as infinite loops and when to terminate loops.

What we could have done better is plan out our instruction vector a little more carefully. We had ideas about how to store instruction objects in maps,vectors, arrays, etc., however there ended up being errors in instruction order preservation, how to call instruction functions, how to point to variables stored in our map, etc. Although we ended up using a vector to store instruction objects, our code could have been a lot cleaner if we were able to separate them into their own classes. However, problems arose when we tried to iterate through the vector since calling a new iterator inside each jump object caused problems with instruction order. We also had some trouble implementing templates into our program and ended up using custom classes to differentiate between variable types. For this reason, we decided to use a map that holds a key and a pair of string variables that holds the value of the variable and the type of variable it is. This allows us to call the variable types when we need to compare variables with other variables. The problem with this map, however, is making sure that the variables are updated correctly, especially in loops. Initially in our jump instructions, we pushed wrong string values into the map by updating the variable keys to the variable values (i.e. if a variable key is “$myint” and its value is “9”, we would accidently update the key to “9”, which, of course, would mess us the map and cause errors while executing). To fix this, we used temporary variables to store the variable values in order to always keep the variable keys constant and unchanged.

Phase 2

For phase 2, we spent most of our time on the new threading instructions, since it was the hardest portion to do. We chose to use the TCP method because it is a more stable and reliable method than UDP. Unlike UDP, TCP offers no lost data, flow control, and keeps the data in order so that instructions won't get mixed up. For most of our code for the client server threading, we took inspiration from the helper code given to us by Karim. We changed our program layout by moving around the parser and vector execution in different classes. We replaced the threadmainbody in the connection class with our own file opening, parsing, and execution. The most trouble of this section was getting to know the concepts behind threading and sockets, and putting them into networks, since none of us have actually dealt with networking before. As for our instructions, the hardest part was figuring out how threading in the program would work. We were unsure about keeping track of the multiple threads while executing, how to lock later variables, and a few other problems. For our final approach towards these problems, we decided to use one vector for exclusively main instructions and multiple vectors for each thread instruction. We also created a linked list of pointers to vectors and then iterate through the linked list, starting each vector's execution iterator one by one, and then letting them run in parallel. We kept lock and unlock variables in a map to help with locking/unlocking threads and would lock the program if the variable is encountered(if it's in the map or not). Unlike part one, part two forced us to create new code in order to make sure this thread is working properly. We made code similar to connection.cpp since it deals with getting and executing each connection in a linked list in parallel, which went along with iterating through each vector in a linked list in parallel. However, we ran into problems trying to pass in the pointer to each vector we need to iterate through. It would give us an empty vector when trying to pass the instruction vector we made. Overall in phase 2, we were able to run multiple clients sequentially and in parallel, but we were unable to run threads within our program successfully. Some key points to be made: to run the program, you'll need one server and a few clients to test it out:

Instructions to run program

-type "make" to compile the server files needed

-to add a new client:

    -we provided our client socket, so you just need to duplicate it for how many clients you want

    -create new TCPClientSocket.cpp, and compile manually (g++ -std=c++14 TCPClientSocket.cpp TCPSocket.cpp -o client, g++ -std=c++14 TCPClientSocket2.cpp TCPSocket.cpp -o client2)

-to run server and client

    -server: ./main

    -clients: ./client1 127.0.0.1 <input file>… ./client2 127.0.0.1 <input file>...etc.

-be sure sure to delete error and out files after each program run through as to avoid run errors.