SURVEY PROJECT AND DESIGN JUSTIFICATION – REPORT

SID GIDELA, DAVID LIANG, JULIUS MAZER

Jack Baskin School of Engineering, University of California, Santa Cruz, 95064

ABSTRACT

The purpose of the Machine Instruction Simulator was designed to utilize a practical approach of all the object-oriented techniques and concepts learned in class to execute a set of instructions. In lateral with software development, the program was initially built with a general object oriented design in mind and finally constructed and modified consistently to fit into a client-server model. Essentially, due to the complexity nature of software, object-oriented programming reduces and separates into problems into smaller blocks that connects with one another. As of result, this project emphasized the importance that object-orient allowed in terms of flexibility over other styles such as strictly functional languages.

HOW TO RUN OUR PROGRAM

1. To compile server type in:
   make
2. To compile client1 which contains “test.mis” :
   make client
3. To compile client2 which contains “test2.mis” :
   make client2

BACKGROUND

Software is inherently complex in such cases that it is hard to manage the development process due to unpredictable amount of behavioral system problems. One of the solution is to use algorithmic and/or object-oriented decomposition to better control the difficulty. With object-oriented structure and the use of abstraction, it will furthermore streamline the program in the form of generic programming. It is mainly achieved with the help of the class and object hierarchy to wrap common structure and operations. Through hierarchy, it greatly simplifies the complexity problem due to the grouping of similar objects under one superclass. One of the important features of OOP is virtual and pure virtual functions. It allows the base class to achieve the derived classes' methods through inheritance. Essentially, all group operations are used by the "superior" class and they can effectively overload the same method name. This allows the user to use one identity name throughout the program and increase readability. Another feature is encapsulation in which class members are used privately within the class to reduce system complexity to keep a constant state. It allows the implementation of an abstract class to transact between its members. At a higher scope, modularity is the separation of program into different sets of classes or components with a well-defined boundary. On the other hand, it separates on the hardware level as seen with header and implementation files rather by classes.

Up to a higher scope of learning, concurrency allows the operation of multiple tasks through various processes. With object-oriented environment, there are active and inactive objects, and in the case of the client-server model, they are threads. They individually execute their methods in respective to the other threads in a seemingly simultaneous movement. In which case, the concept of persistence is of greater importance. Traditionally, any object that occupied a set of memory would be destroy at the end of execution of the program or the main function. Now, the objects are typically alive longer than the process itself and will continue existing until it completes the objective it was set out to do.

In the virtual machine project, there are some important note to be said. There was no windows operating system being used to test while OSX and Linux were primarily the host. Due to some complications on the understanding of helper code given, OSX could connect to "server.cpp" flawlessly with no "invalid argument" errors. For Ubuntu on the other hand, "server.cpp" works periodically depending on how fast and efficient the operating system cleans up the process and if the computer was restarted. It was suggested in Piazza that a known problem in one of the given code distorts the compilation, and for that reason, it is likely the source of error for Linux users. Another note to make of, there are issue with "TCPServerSocket.cpp" particularly the bind function. In OSX, it requires the use of "::bind" to run, and Ubuntu, it does not.

There is various amount of options within the Makefile that uses either and/or POSIX threads and C++14 library. As of the final report, the first option will sufficiently compile the server and each option (primarily the first two) afterwards will compile the client in respective order

DISCUSSION & ANALYSIS

The overall design of the client-server model was based on TCP and not UDP for various reasons. The foremost motive was that we did not process the wealth of knowledge required to easily access and manage the inherent difficulty of UDP. As much as we would like to be efficient, TCP is known to be much more reliable to a higher degree than UDP at the same time. UDP could be reliable but to know where understand the locations to safeguard was a different story given the steep learning curve. Also, it would not make sense to use UDP if a byte of the file was to likely go missing while receiving the packets. Since the instructions are taken directly from the file, a missing packet of an instruction could dramatically change the outcome of the result. This is where TCP comes in handy. It does a three-way method known as the three-way handshake. The server sends a signal for a specific user, then said user sends a signal of intent to connect to in which the server would send a response to see if the user has received it. If not it either resends or timed out. A great analogy to look at this decision is like listening through a telephone versus a radio. If there is a problem, we could ask the person to repeat what they were saying which cannot be done with a radio host without some safeguard such as a recorder.

The general design of the project was separated into two parts. The client would read and check for any instruction errors much like spelling. The server would initiate and check the instruction for any errors and proceed to send the result to the ".out" file and any single error obtained to ".err" file. The idea came from not wanting to waste time sending an incorrect file with minor errors to the server only for it to return it right away. This would be an easy fix and scan on the reader's part and will not further burden the server's resources.

POSIX thread was the choice over C++11 library thread. Initially, it was due to platform reasoning. We had three different operating systems at first but now we moved over to just OSX and Linux. Another great reason we decided to do POSIX threads was because it enforced proper network programming behavior.

We included three maps for the project. One map takes in a "VAR" type which was our way of holding any data types to the variable. The other one holds string pairs that is used for the label method and is necessary for the implementation of all the jump functions. The third one holds the length of the character array from the string values in order to make it easier for the set and get methods. The decision to do this was because if a user said they wanted to allocate 100 space for a five-letter string like "hello" then they could. The biggest problem was because the

user could use set or get at any index that is beyond the length of the actual string. This was our assumption due to the mandatory need to allocate memory for strings. Thus, if they use set function at an index of 40 in a string called “hello” with a memory allocation of 50, then it would produce the outcome as hello with 36 whitespaces in between “hello” and the letter. On the other hand, there would be no whitespace after the letter or after index 40 to keep the “.out” organized and clean. It would only appear like that if the user indicates they would want to do that. It is the same idea for the get function. If the user wants a letter at an index beyond the actual string length then they would obtain an empty response, “ “.

There are sixteen instructions with about four parent classes to them of similar functionalities. Like how add, subtract, multiply, and divide would be under the operation class. There is a main class, Instruction, that all the classes inherited from to streamline the functions for the cloning process. However, the methods in Instruction are not pure virtual because there would be no point in having to implement four to five methods to each class when they do nothing relative to one important function. “Misc” encapsulates out and sleep functions since they do not match with the others. An interesting note after the inclusion of “\t”, “\n”, and “\r” is that the “out” class will not include whitespace in between a variable or constant until the users say so.

All the jump functions rely on the initiation of “label” before they could be used or it would produce an error. On the topic of jump, there are seven classes each with a different jump functionality. We went through with this route because of our style of parsing the files and implementing the actions based on the exact word. If we had to put in “JMPN/NZ” and “JMPLT/GT/LTE/GTE” it would be a strange way of putting it into a hierarchy. With seven classes, it would be easier to determine which word the user decides to use right away.

There is no error exception handling class to the same extent as the general project. One of the biggest reasoning is that all of the classes usually have their own separate and unique errors. With the class exception, we felt that it would be too vague to satisfy all the needs to the functions. We wanted to give the user the ability to figure out what went wrong easily instead of manually searching through the instructions to find the error. We made sure that any error produced by the program will exit out right away. Realistically, we would not want to give any consumer an error-prone program. Also, we assume no developers of good reputation would want to give a buggy program to the market or risk the fear of losing consumers. Mainly, all our

error checking start at the beginning of each function to ensure they match the ideal instruction. If not, they are sent into the ".err" file.

There is an assumption due to the confusion we experienced from reading the instruction. It was mainly on how multi-thread works on each individual ".mis" files. We assumed that we are doing this based on TCP model that the threads would not necessarily move in an unordered style. Main reason is that some of the variables and results heavily rely on the program going through a sequential process.

Another assumption that we made in the assignment was whether there should be one ".out" and one ".err" file as the result. Nowhere in the instructions did the professor mentioned producing multiple files for each respective thread. In our case, we automatically let multiple threads assign the results into one ".out" file.

An interesting case that we noticed during the second phase with multi-threads that the results were mixed match. Our assumption is primarily since we are sharing one global map between multiple clients. As result, it may seem like our operations were producing wrong results when it was the variable within the map being overwritten and messed around by other threads. In order to ensure that the operations are fully correct, one client is the best example in determining how accurate the they are.

CONCLUSION

The Machine Instruction Simulator project was used to apply object-oriented design and concepts to execute sets of instructions. With that knowledge, we effectively created a generic program that initiate the instructions. However, changes were constantly made to fit the client-server scheme in which generic programming allowed us to make an easier transition instead of having to make major changes in functional languages. The main goal of this project was to discover why object oriented program was necessary to use for a large-scale project.

REFERENCE & WORK-CITED

Hall, Brian. *Beej's Guide to Network Programming Using Internet Sockets.* N.p.: Jorgensen, n.d. *Beej's Guide to Network Programming.* Jorgensen Publishing. Web. 22 Nov. 2016

Hong, K. "C++ TUTORIAL MULTITHREADING/PARALLEL PROGRAMMING IPC." 20 Nov. 2016, http://www.bogotobogo.com/cplusplus/multithreading_ipc.php

Stroustrup, Bjarne. *Programming: Principles and Practice Using C.* Upper Saddle River, NJ: Addison-Wesley, 2015. Print.