James Iwamasa
Sean Wakasa
Jeffrey Zheng

Final Report

Description:

MIS_server/MIS_client is a client/server application for the execution of machine language programs (mis programs). A client reads in and sends a file to the server, where it is processed. The output is then returned to the client.

Compilation:

After downloading all necessary files, run “make” at the terminal to create executables “MIS_server” and “MIS_client”.

Usage:

On the server computer:

MIS_server [*address*] *port*
address: The address the server should listen on. If none specified, listens on all available.
port: The port the server should listen on.

Once the server is running, on the client computer:

MIS_client [-f] *address port filename*
address: address of the server you wish to connect to
port: the port of the server you wish to connect to
filename: name of the mis program file (must be of form “filename.mis”)
-f option: since we need to send programs over the network, and output is printed to files, by default, program length and instruction execution count is limited. The -f flag removes this limit.

Known bugs:

LABELs can’t have numbers in them

Detailed report

Networking:

-To facilitate the client/server model, we use TCP network sockets for moving the data around.

-The MIS_client reads in a file, and sends it as a file stream over to the server. The server reads, parses, and executes the program. Finally, the server sends the output back as a stream of tokens to the user, where they are printed to the appropriate output and error files.

-The MIS_server can also process multiple requests in parallel via multithreading. Whenever a new connection is accepted by the server socket, a new thread is created to process the request.

Networking Reasoning:

-Since the mis language is very simple, there is not much preprocessing the client itself can do. So sending over the file as a whole for the server to parse made the most sense.
-But for returning the output, we had to distinguish between two separate files, the .out file for normal output and the .err file for error output. Sending the output as a stream of distinct tokens made it very easy to signal the client (by sending a special signal token) to switch between the outputs.

-TCP was used over UDP for various reasons:

1. In order packets: since all the server does is parse and execute a sequential program, it doesn't really make sense to use UDP, since we would have to reorder the packets anyway.
2. Security: for the mis programs to execute correctly, we obviously need all the instructions intact, so using UDP would have required us to force consistency, which TCP does for us.
3. File streaming: While output is returned as a stream of tokens, the input is sent as a continuous file stream, which TCP is much better for than UDP.

Class/Object hierarchy:

MIS_client:

-Simply has one central class related to it – Client – which does all the file reading, network sending, and output receiving

Client:

- -Reads in a file, sends it to the server, and receives the output, printing it to the appropriate .out .err files

MIS_server:

-Again, we have one central class – Server – which calls other classes to perform all the other functions

Server:

- -Is created once in the main program of MIS_server
- -Contains the server socket and creates all new connections with clients
- -Whenever a new connection is made, it is given to a Connection object to do the processing

Connection:

- -Class that takes a connection created by the Server object, and processes the request
- -Runs all of its processes in a detached thread, allowing our server to handle multiple requests at once

TCPSocket and TCPServerSocket:

- -Classes provided by professor Sobh to facilitate an object-oriented implementation of networking
- -Is used by Server, Connection, and Client for sending packets over the network

Parse:

- -A class containing several static methods for parsing and processing the file into a runnable mis Program object
- -Performs actions such as tokenizing the file, checking syntax and semantics, etc
- -In the end, creates a Program object which is runnable
- -Parses in two “passes”
    - -First pass: processes variables and jump labels
        - -We do these first since all variables/labels[1] are global
    - -Second pass: processes instructions

[1] Labels are actually only global to their thread, but this is handled by ThreadInstr

FactoryMap:

- -A container class for the middleware factory map for parsing the instructions
- -Is use by the Parse module for parsing
- -The Instruction map is a map from instruction command names (ADD, MUL, SLEEP, ETC) to pointers to the functions that perform said command (more details in "Instruction" section)
- -The Variable map is a cloner map, which maps the variable names (NUMERIC, CHAR, etc) to an instance of the Variable object, which is cloned to create a new instance (more details in "Variable" section

Program:

- -A manager class that contains everything needed to run the program
- -Contains a map from variable names to the actual Variable object
    - -Also contains references to all constants, which are treated as Variables (explained in "Variable" section)
- -Contains a map from label names to the line number they correspond with
    - -Line numbers are implemented as Integer Variable objects
- -Contains a vector of Instructions in sequence for execution
- -Contains buffers for the normal output and the error output
    - -Buffers are implemented as vectors of strings, rather than printing to a temporary file, which makes sending the output back as a stream of tokens easier
    - -Making temporary files may also be troublesome if multiple clients send files of the same name, so each Program instance has its own buffer
- -Contains a list of all detached threads running within the program
- -Contains a tokenized version of the file as a reference for debugging
- -execute() runs through the instruction list, executing the program
    - -This may include creating and running new threads (more in "ThreadInstr")

Instruction:

- -A class that holds the Instruction command and all its arguments
- -The arguments are stored as a vector of Variable object pointers
    - -These point into the Program's variable/constant maps
- -The command itself is stored as a pointer to a function
- -Use of a function pointer to differentiate between instructions was used as a lightweight way of supporting many instructions and instruction extendibility. To implement a new function all you have to do is define it and add it to the FactoryMap.[2]
- -In a manner similar to that of interpreted languages, argument checking is partially done in the Instruction at runtime, either within the Instruction function, or in the Variable objects

[2] This is not inherently compatible with DSOs, however.

Interrupts:

- Some instructions, instead of executing their own logic, cause “interrupts”
- These are for commands where something other than just a variable needs to be updated
- They throw a negative integer to the Program object for special actions:
  - Starting a new thread
  - Blocking until all threads are done
- Printing is in a way an interrupt which throws a vector of arguments to be printed to the Program’s internal buffer
- Jumps also act similarly, throwing the line number we should transfer control to

OpFuncs:

- A header that contains all of the function declarations for the various instructions
- They all must follow the form defined in the Instruction.h header
- Implementations are stored in appropriate .cpp file (StringFuncs.cpp, MathFuncs.cpp, etc)

Thread:

- Multithreading library provided by professor Sobh
- Used in ThreadInstr for threads within a mis program, and by Connection to support processing multiple client requests at a time

ThreadInstr:

- Represents a detached thread in a mis program
- Contains a vector of instructions
- These inherit from the provided Thread class, and execute the instructions in its list in a detached thread
- Are created when a BEGIN_THREAD interrupt is thrown

Variable:

- Object that represents a variable in the mis language
- Designed as a base Variable class from which all the types descend from (VarInt, etc)
- The single base type allows us to store all of our global variables in one place, which makes sense since we don’t need to worry about scope (even between threads)
- By use of selectively defined non-pure virtual functions, this provides us a very organic way of type-checking:
  - We attempt to call a getter/setter on a variable
  - If it is of the correct type, then the vtable guides us to the appropriate function
  - If it is incorrect, the function “falls-through”, calling the parent Variable’s function
  - These functions then throw the error at runtime
- Variable objects also represent constants, which have a constant flag set
- Variables can also be locked by a calling thread to give it exclusive use

MisException:

- The exception, error handling class
- Is thrown whenever an error occurs
- We can throw exceptions from a lower level with specific messages and bubble them up to attach more information
  - Typically, we throw a message at the Variable, Instruction level, it’s caught by the Program level, which then appends the line where it occurred