Final Project Report

**Team member: Jiaqi Liu, Hengqing Wang**

**vunetID: liuj19, wangh26**

**github repo: https://github.com/CS3281-2016/Jiaqi-Hengqing-Project.git**

**Launching:**

The project is divided into two parts, server and client. To launch the server, go to “Server” folder, make a new folder “build”, then go to “build” and type in command “cmake ..”. Finally, type in command “make” to build the server, there will be an executable called “server” in “build” folder. This executable has one parameter, which is the port number. A port number is required to execute the executable.

To launch the client, go to “Client” folder, make a new folder “build”, then go to “build” and type in command “cmake ..”. Finally, type in command “make” to build the client, there will be an executable called “client” in “build” folder. This executable has two parameter, one of which is the server host address, the other is coordinating port number. The port number is 20000 by default.

**Implementation and design:**

1. **Server:** Firstly, we set up a TCP server socket, and make it wait for a client to connect. Whenever a client gets connected, the server forks a child process exclusively for this client, and redirects stdin, stdout, and stderr to the socket. Therefore in this process, stdin will always read from socket, and stdout and stderr will write to the socket. Finally, execpl calls the shell we built in assignment3, to fulfill the shell functions.
2. **Client:** Firstly. We set up a TCP socket, connect the client to server, and set a timeout for recv function (we are using a blocking recv because unblocking recv is hard to control). Therefore, recv will receive output messages within the time span, and after that time, recv will not block and the code goes on to be executed. The timeout span is initialized to 1 second.

The client first receives a welcome message from server, and waits for user command to be typed in, send the command to server for executing, receive the output message from server, and finally print the message on client screen. If the command contains “sleep”, we reset the timeout span to be the sleeping time plus 1 second, so that the result message of sleep can be timely received by client and printed on client screen.

What the program does can be summarized as: sending a command to server 🡪 Server execute the command 🡪 send the output back to client 🡪 Client receive the output and print it to screen.

1. **Statistic Monitor:** As we know, the monitor has two basic parts – the command history and the CPU, memory recording. In order to record these parts clearly, we open two .txt files “commandsRecords.txt” and “utilizationRecords.txt” to record the command history and utilization information. They are created and written as the program goes.

For the command history part, we write every input command into “commandsRecord.txt” file (code for this part is in main.cpp). If the input command is “erase history”, the records before will be deleted and a line of reminder “You have already erased some commands!” will be written in the file. Finally, if “exit” command is entered, the total number of command you entered will be written into this file.

For the utilization part, “utilizationRecords.txt” records the user and system CPU time and the memory usage. In serverSocket.cpp, after we wait until all the children processes are terminated, the getrusage() function is introduced and it returns the resource usage statistics for all children of the calling process that has terminated and been waited for. Among that, the user and system CPU time usage and maximum memory size are extracted and written into the file and printed out to the screen.

**Pitfalls:**

We met countless pitfalls during the project, one of the most buggy ones is getline function. On client side, there is a getline function asking the user to input commands. In the shell code which lies on the server side, there is also a getline function redirected to read from socket. We found that the getline in shell code continues blocking even the command is sent to it from client. The reason is that when we type in a command on client side, the getline in client code discard the newline character and does not send it to the server, so we really need to append a new line character, as the delimiter, explicitly at the end of each command, so that the getline in shell knows where the command ends. This bug really cost us a long time to detect, and the lesson is that the computer will always do what you tell it to do, nothing more and nothing less.

As for statistic monitor, we also have a lot of gains related to operating system concepts. For example, at first, we wanted to put the getrusage() function which carries RUSAGE\_SELF flag in every child process that records every calling process’s usage separately. After that, by simply adding all the children’ usage, we can get the final total resource usage. But after revising the fork() function, such idea was abandoned. Because when a new process is forked, all the memories are copied into the new child processes, but unlike the threads, which share the same memory, none of the variables in parent process have relations to the children processes. So the proper way to achieve our goals in utilization monitor is to wait in the parent process until all the children processes are terminated and then implement getrusage() function with RUSAGE\_CHILDREN flag**.**