COMP 304 Project 1 Report

Ahmet Koca 0076779 – Yusuf Çağan Çelik 0079730

Note: All project implementation is done together on zoom calls. Two branches are created on the github team for each of us, even though we worked on everything simultaneously. Our branches include the same code and github push. Main branch is empty and it should be discarded.

PART 1-

Exit and cd commands are the commands which we are going to implement so we started with exit command.

To make exit work we just checked if command-> name is “exit”. If so, we returned EXIT.

To make cd work we handled cd like this: At first check if command-> name is “cd”. Then we check the argument count being more than 1 because the typical working with cd command is “cd path”. Then we get the path and assign it to a variable with a chdir command. Then we write a bash command with printf which looks like this:

```
printf("-%s: cd: %s\n", sysname, strerror(errno));
```

and then we return success with error handling.

For the other built-in bash commands, we are forking and making a child process and then we are trying to find the executable file in the PATH environment. After finding it we reconstruct the whole path again. Then we execute it.

We handle the background processes by checking if command->background is true or not. If command->background is true, parent process returns SUCCESS without waiting. If command-> background is false, parent process waits child to finish its execution then return SUCCESS.

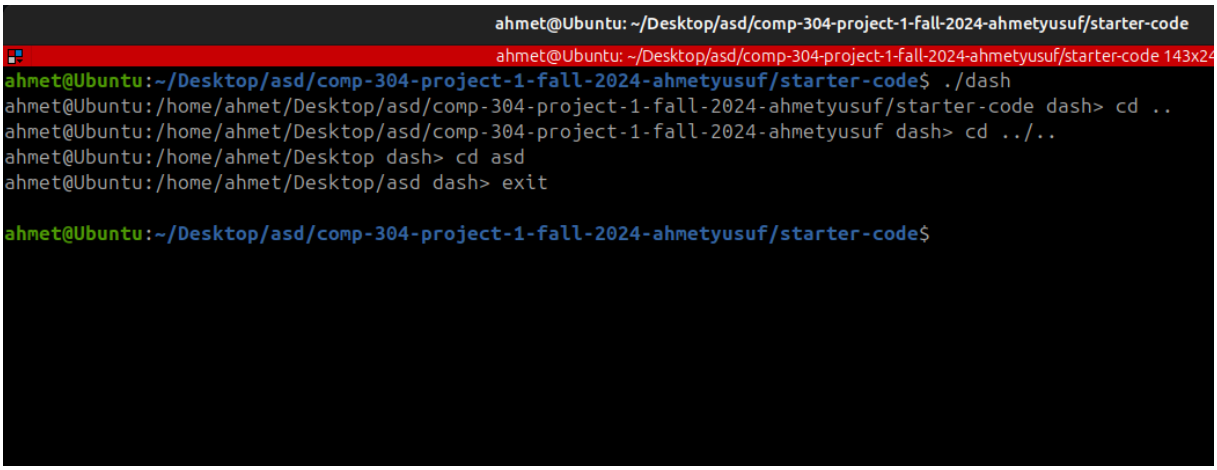

Figure 1: Example usage of cd and exit.

PART 2-

For I/O redirections, we are using command_t->redirects. The parser sets the correct thing to correct places with redirect_index as follows: 0 for "<" , 1 for ">", and 2 for ">>".

If command_t->redirects[0] != NULL it means that we are trying to read the file. We open the file with open command. Then in a child process, we dup2 the file we opened to STDIN_FILENO and we close the file. The same logic applies for write and append as well with checking redirects[1] != NULL for writing and redirects[2] !=NULL for appending. If they are not null then we understand that we are trying to write or we are trying to append.

Our redirection is not working with space after the redirection symbols. So the example usage for our redirections is like this:

`dash> ls >output.txt.` With no spaces after >, >> or < operations.

Pipeing is done like this: Our command struct has command->next. We check if command->next != NULL to see that if there is another command in the pipeline. If command->next is not null, we create a pipe using pipe(pipefd) and pipefd is an integer array with length 2. Then for each command in the pipe we fork a new process. The child handles the execution of the command. And if there is a next command, we connect the stdout to the write end of the current pipe and the same thing goes again.

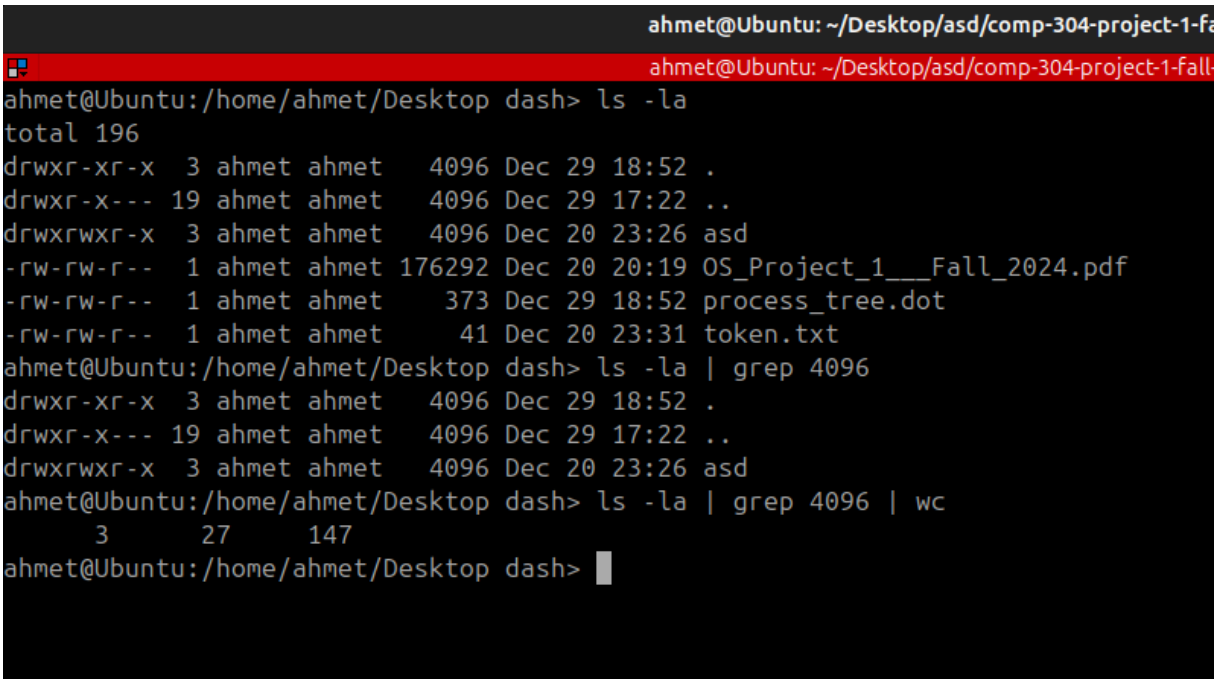

Figure 2: Example usage of piping.

PART3-

We could not handle the autocomplete part and it is not working as expected. We worked on it and its code can be found in the shell-skeleton.c.

We handled kuhex with a void kuhex_dump(filename,group_size). The function opens the file, and it reads it with 16 byte chunks with a buffer to hold. Then each byte is printed in two-digit hexadecimal format with grouping. The group size is given to the function. Then the ASCII representation is printed.

We check the usage of kuhex with 2 if statements after getting into the kuhex statement which is : if (strcmp(command->name, "kuhex") == 0)

We first check if command->arg_count < 3 and print the usage of the kuhex if it is smaller then 3.

It is 3 because in the skeleton code if you enter 2 arguments the argument[2] which is the third one is initialized but set to NULL. That is why we always have one more arg_count.

And the we have another if for checking the usage with “-g “ flag of the kuhex. We check it by if (command->arg_count == 5 && strcmp(command->args[1], "-g") == 0) and we change our group size accordingly in this if statement. The example usage is like this with “-g” flag.

dash> kuhex -g 2 input.txt

And for the general usage, we make group size set to 1 between those 2 if statements for using the kuhex without “-g” flag like this:

dash> kuhex input.txt

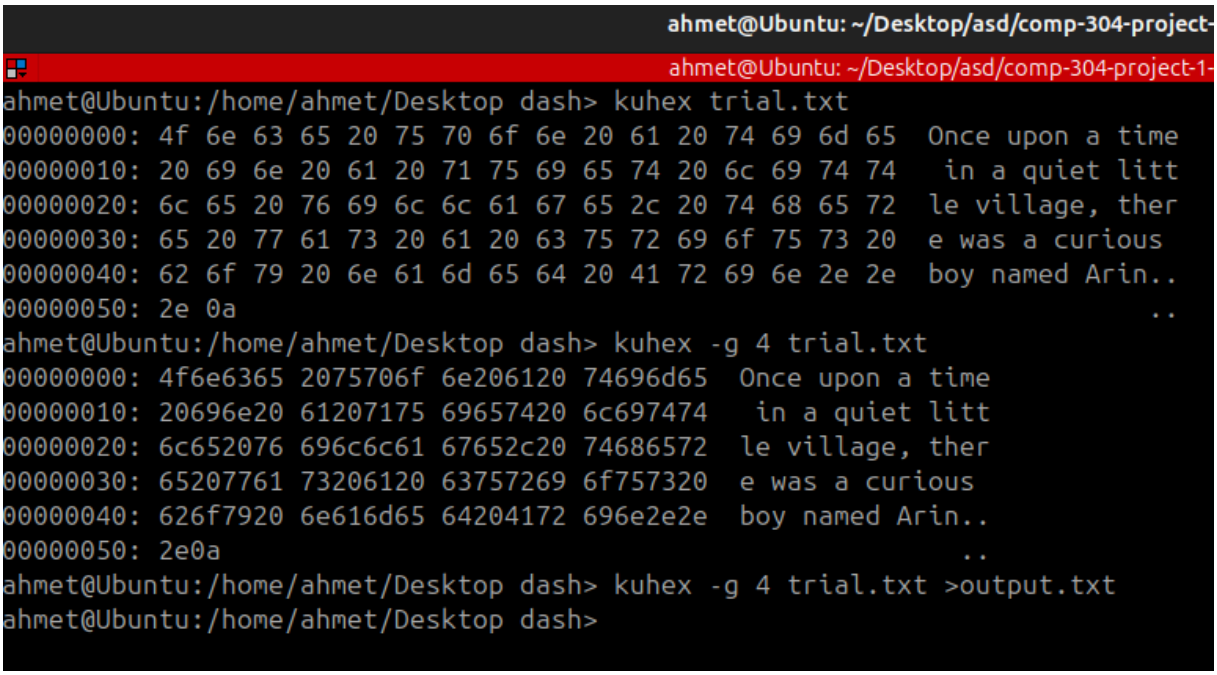

Figure 3: Example usage of kuhex and I/O redirection.

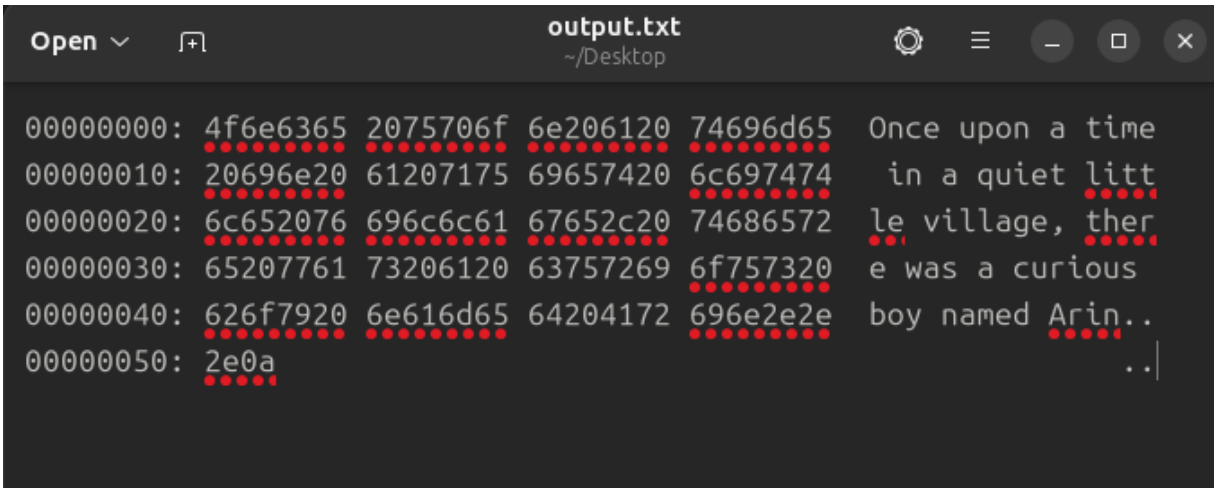

Figure 4: output.txt

Part4-

For the psvis command, we created a kernel module named mymodule. The psvis operation consists of two separate steps, kernel-space operations and user-space operations. The kernel module is responsible for retrieving the process tree according to the parameter given to it. In the user space, in skeleton code, where we handle the existing commands, we added a case for psvis command since it will be a built-in command. A PID is given as an input with psvis command. The command's format is like "psvis <PID> <output.png>". In the code, this PID number is sent to the kernel module with proc file-system. The kernel module gets the PID, finds the relevant process and retrieves its children. It is then sent to the program and then parsed and a png file is created through graphiz.

When psvic command is entered, if the kernel module is not loaded, then it is loaded to the kernel first. If it is already loaded, then the user is informed that the module is already loaded. After that, the necessary operations are done. The output .png file is saved to the directory where the command is entered.

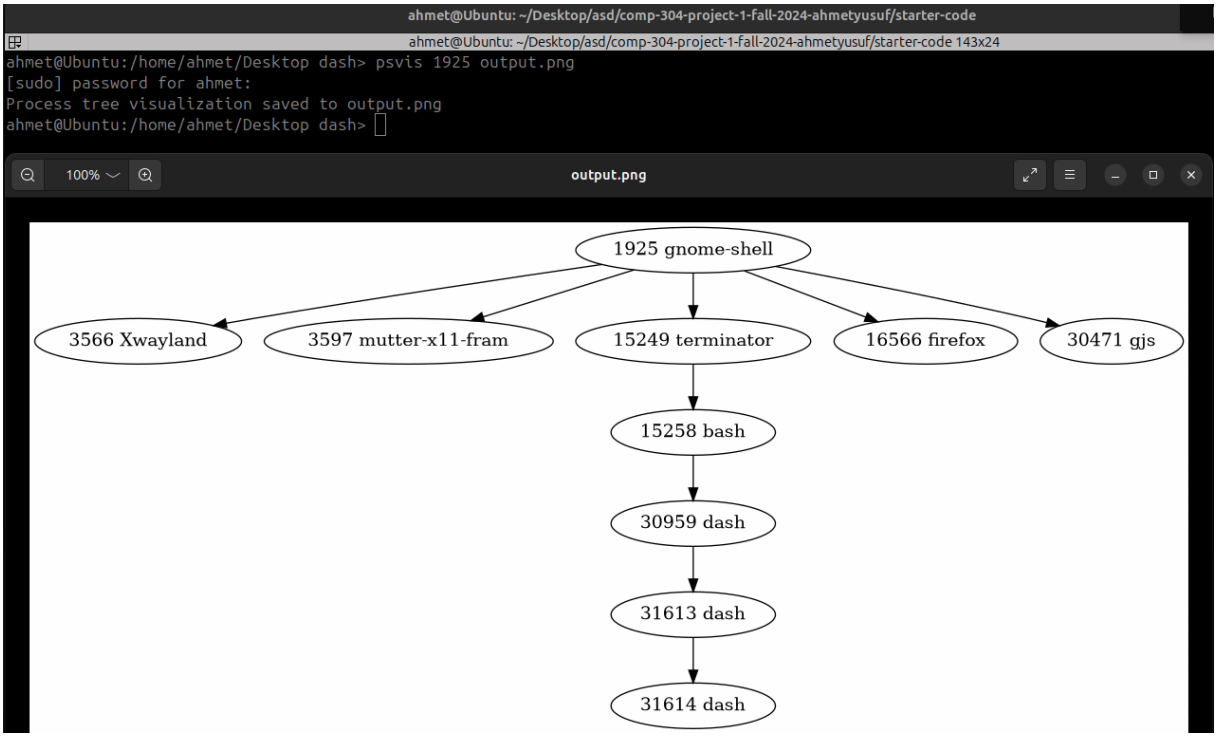

Figure 5: psvis command and an example output (PID and the process).