CSC 345

Project 1 Report

Overview and Executing a Command in a Child Process

To display a prompt with the up-to-date, working directory, I used `getcwd()` to place the full path name into `char full_path[]`. I used `strtok()` to tokenize `full_path` and placed these tokens into `char* curr_directory[]`. I then included the final token of `curr_directory[]` (the working directory) in the `printf` statement for the prompt.

To execute a command specified by the user, I added a char array, `commandBuf[]`, to store the entire line of input from the user. If the command is "exit", then `should_run` is set to 0 and the program terminates. I used `strcmp()` to implement this. `strtok()` is used once again to tokenize the command line input. These tokens are placed into `char* args[]`. A child process is then created using `fork()`, and the command is executed using `execvp(args[0], args)`. In the parent process, the final token of `args[]` is checked to see if it is "&". If it is, then the parent and child will run concurrently. Otherwise, the parent calls `wait(NULL)` to wait for the child to finish before continuing execution.

Creating a History Feature

To provide a history feature, I created a doubly linked list based on a `struct HistoryItem`. A `HistoryItem` contains `char command[]`, `struct HistoryItem* next`, and `struct HistoryItem* prev`. Furthermore, I created the linked list related functions, `createItem()`, `rearrange()`, `insertItem()`, `getLastItem()`, `printHistory()`, and `getTail()`. When input is entered through the command line,

the “command” (it may or may not be a valid command) is saved into history using `insertItem()` (there are exceptions to this in cases of special commands). This function creates a new `HistoryItem* newItem` that holds the inputted command. The `newItem` is then placed into the first open spot in the linked list. However, if there are already five items in history, `newItem` is not appended to the list. Instead, `rearrange()` is called from inside `insertItem()`. The `rearrange()` function uses `strcpy()` to copy each linked list item’s `command` variable into the previous item’s `command`, effectively shifting the entire list up by one. For example, item two’s `command` is put into item one’s `command`, item three’s `command` is put into item two’s `command`, and so on. Subsequently, the newest command entered by the user is copied into the `command` variable of the fifth item of the linked list (the last position). In this fashion, the oldest command (which was item #1 in the list) gets overwritten.

If the user enters “history” into the command prompt, then `printHistory()` is called, which iterates through the linked list and prints out the value of the `command` variable of each item. Alternatively, if the user enters “!!”, the previous command is obtained through `char* prev_command = getLastItem(&historyHead)`. The `prev_command` is copied into `commandBuf` so that the previous command line arguments will be used. The command is echoed by printing `commandBuf` to the terminal. If there are no commands in history, `getLastItem()` prints the message, “no commands in history”.

Unfortunately, I was not able to implement the arrow key history feature in this project. However, I believe that the `getTail()` function could be useful for such a feature, as it returns a pointer to the last `HistoryItem` in the linked list, which holds the previous command.

Redirecting Input and Output

To check if a command contains either the “>” or “<” redirection operator, I utilized `strstr(commandBuf, ">")`, and so on. If the command contains either of these operators, the tokens in `args` that occur before the “>” or “<” are copied into `char* args_before_char[]`, and the index of the token that occurs after the operator is stored in `int i_after_char`.

In the child process, if the operator is “>”, then `fd = open(args[i_after_char], O_WRONLY)`. This sets `int fd` to the file descriptor of the user-given file (the file’s name is stored in `args[i_after_char]`). Then, `dup2(fd, STDOUT_FILENO)` is called. This duplicates `fd` to the standard output so that any writes to the standard output will be sent to the user’s specified file. Subsequently, the command is executed with `execvp()` using the command arguments before the “>”. The command’s output is redirected to the specified file.

Alternatively, if the operator is “<”, then `fd = open(args[i_after_char], O_RDONLY)`. Then, `dup2(fd, STDIN_FILENO)` is called. This duplicates `fd` to the standard input so that the file data will be sent to the standard input. Subsequently, the command is executed with `execvp()` using the command arguments before the “<”. The data of the specified file is redirected to be used as input to the command.

Communication via a Pipe

To check if a command contains “|”, `strstr(commandBuf, "|")` is utilized. If the command contains this operator, the tokens in `args` that occur before the “|” are copied into

`char* args_before_char[]`, and the index of the token that occurs after the operator is stored in `int i_after_char`.

In the child process, if the command contains "|", then a pipe is created with `pipe(pipe_fd)`. Then, a second `fork()` occurs to create a "grandchild" process. In the parent/"first child" process, `dup2(pipe_fd[WRITE_END], STDOUT_FILENO)` is called, which duplicates the write end of the pipe to the standard output. Subsequently, when `execvp()` is called using the command arguments before the "|", the output of the command is put into the write end of the pipe. In the grandchild process, the command arguments that occur after the "|" are saved into `char* args_after_char[]`. Then, `dup2(pipe_fd[READ_END], STDIN_FILENO)` is called to duplicate the read end of the pipe to the standard input. When `execvp()` is called using the command arguments that occur after the "|", the data from the read end of the pipe is used as input to the command. In this way, two commands may communicate with each other. For example, `ls -l | less`.