Homework #2

Linux Shell Implementation

Part 1 - Command Execution

```
execvp(ecmd->argv[0],ecmd->argv);
```

I used the execvp command. This command takes performs the same functions as execv command by taking in the absolute file name to be executed as the first argument and the arguments to the program as an array of pointers. The parameters passed here to the function are from the ecmd structure. ecmd->argv[0] contains the program to be executed and ecmd->argv contains all the arguments to be passed to the program. The only difference is that execvp command will find the file name from the PATH variable. If the path variable is not defined the path list defaults to the current directory followed by the list of directories returned by confstr(_CS_PATH).

Part 2 - I/O Redirection

```
int file = open(rcmd->file,rcmd->mode,00644);
dup2(file, rcmd->fd);
runcmd(rcmd->cmd);
```

In order to open the file I used to the open command and passed the parameters from the rcmd structure. rcmd->file stores the name of the file to be opened/created, rcmd->mode stores the mode in which the file has to be created. 00644 determines the permission to be set to the file in case a new file is created. 00644 will give rw-r--r-- permissions to the file, i.e., the file will be writeable only by the owner and will be readable by everyone else. None of the users will have the permission to execute the file. The output of the open command will be a file descriptor.

The dup2 command will then duplicate the file descriptor from the open file to the rcmd->fd. This will allow the program being executed to either write to the file or read from it.

Lastly, the runcmd function is called in order to execute the program.

Part 3 - Pipes

```
    if(pipe(p)!=0)
        exit(0);
    int pid = fork();
    if (pid == -1)
        exit(0);
    else if (pid == 0){
        close(p[0]);
        dup2(p[1],1);
        runcmd(pcmd->left);
        close(p[1]);
    }
    else {
        close(p[1]);
        dup2(p[0], 0);
        runcmd(pcmd->right);
        close(p[0]);
    }
    wait(&pid);
    break;
}
exit(0);
```

Here, the program first checks if a pipe was successfully created. The output of the pipe command is 0 if a pipe gets created without any errors. The program then creates a child process using the fork command. It then checks if the command ran successfully.

On successful creation of the child process, in the else if statement, the program will close the read end of the pipe, duplicate the write end of the pipe to STDOUT and execute the program by calling the runcmd function. This will allow the process to the right of the pipe operator to take the output from STDOUT and treat as input.

The else statement will allow the parent process to close the write end of the pipe, duplicate the read end of the pipe to STDIN and execute the program by calling the runcmd function.

The wait command will then suspend the execution of the parent process while the child process is running.

Part 4 - Summary Questions

Ans 1. In order to execute the commands successively we can use a semicolon between the commands.

```
(cd /XXXdirectory); find . -name 'a*' -exec rm {} \ ; ls -la
```

The semicolon ensures that the command after the semicolon gets executed regardless of the successful execution of the previous command.

The && operator ensures that the command gets executed only if the previous command exited with status code 0 i.e., did not error out.

```
(cd /XXXdirectory) && find . -name 'a*' -exec rm {} \ && ls -la
```

In this case, if the change directory command fails, none of the commands that follow will be executed.

Ans 2. In order to implement a subshell, we can create a fork of the parent process. Everything inside of the parenthesis can then be passed to the runcmd function. Inside the case ' ' we can implement a check whether this is a subshell execution or not, if so we can execute the child process using execve command. This command allows us to run the commands inside of the subshell with with newly initialized stack, heap, and data segments. After the execution of the subshell, the parent process can finish it's execution just like our implementation in pipe.

Ans 3. In order to run a command in the background, if we can first check if there is a & at the end of the command. If it is, we can track it using a variable and we can create a fork and execute the entire program inside of the child process. We can then use setpgid(0, 0); inside the child call to put the child in a new process group. The parent process can then just continue without calling the wait. This way the child process execution will continue in the background.