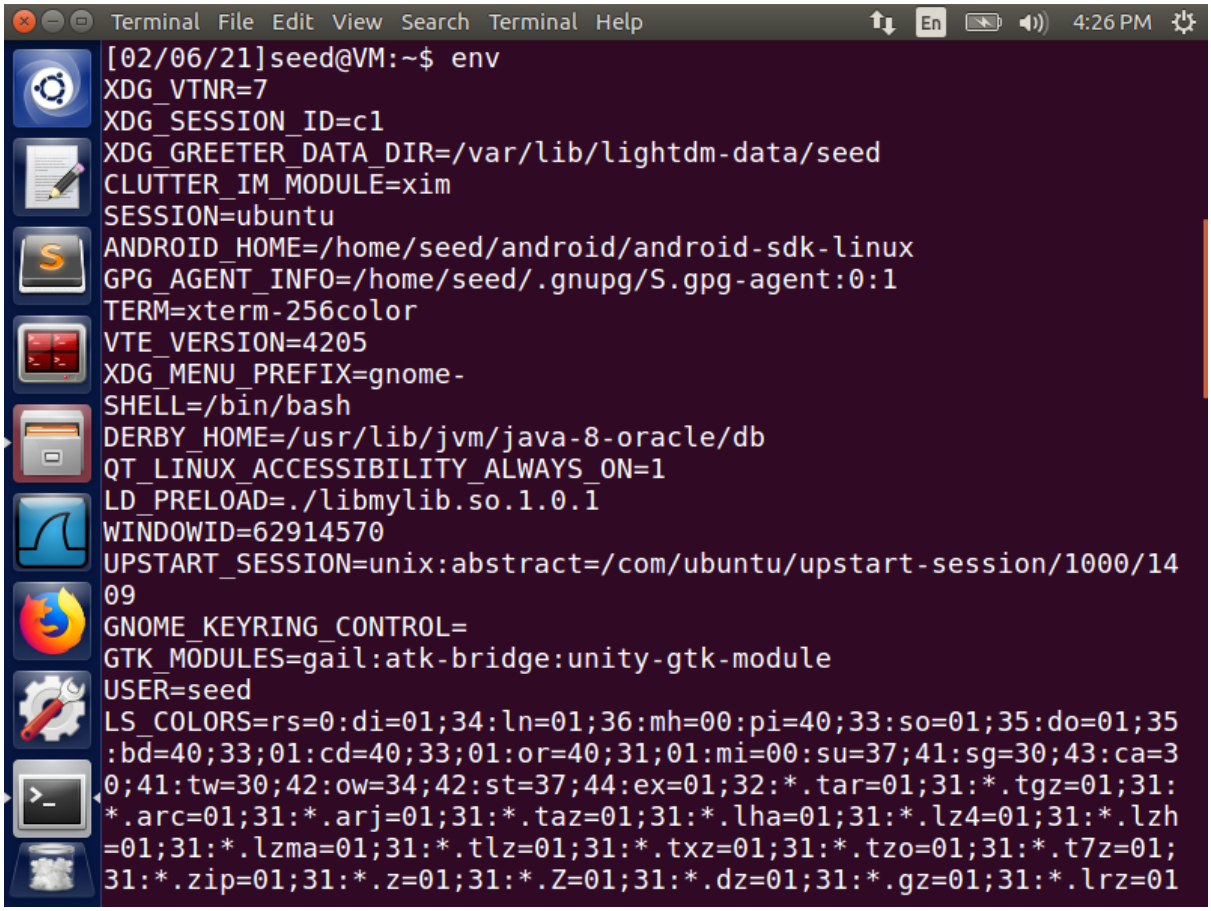
ASSIGNMENT -1

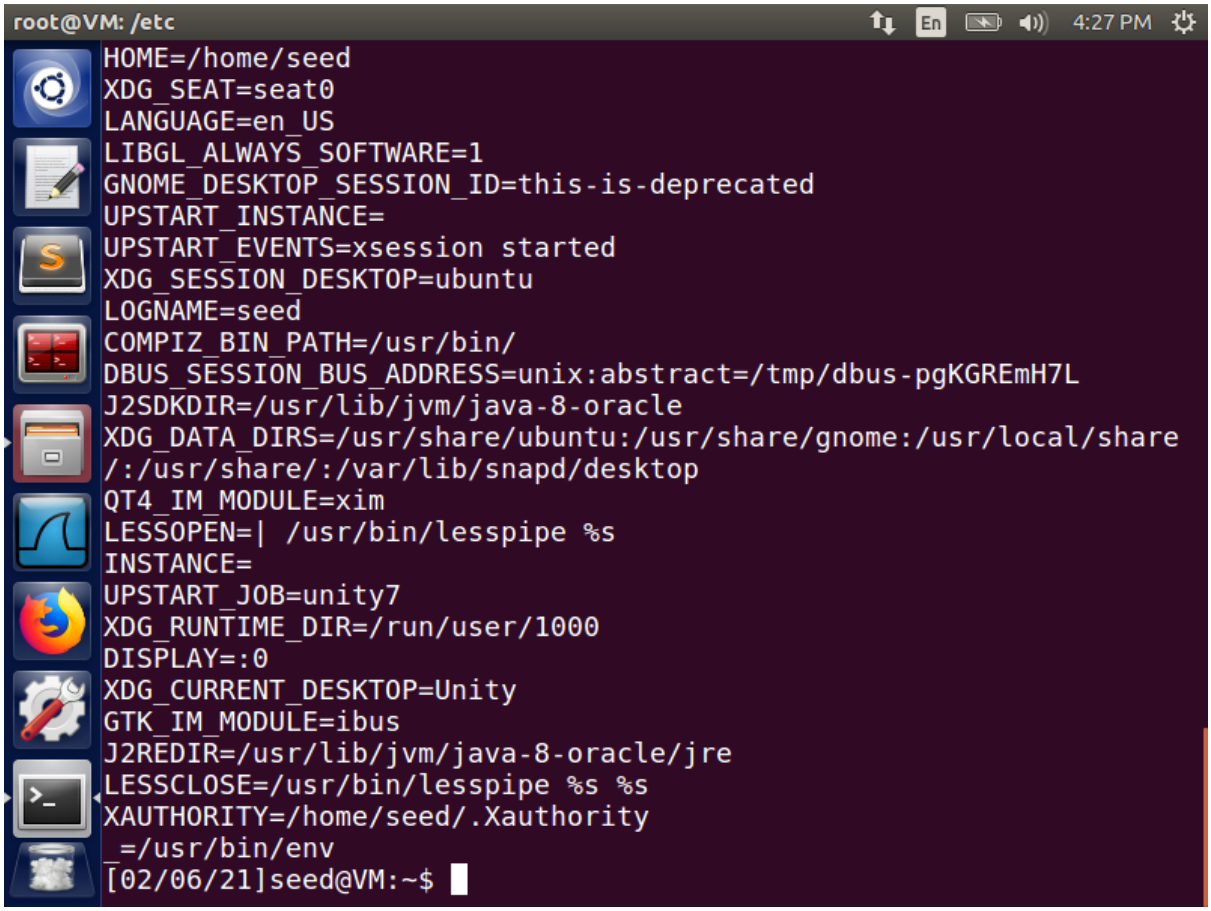
Name : Sudharsan Srinivasan

ID : 1001755919

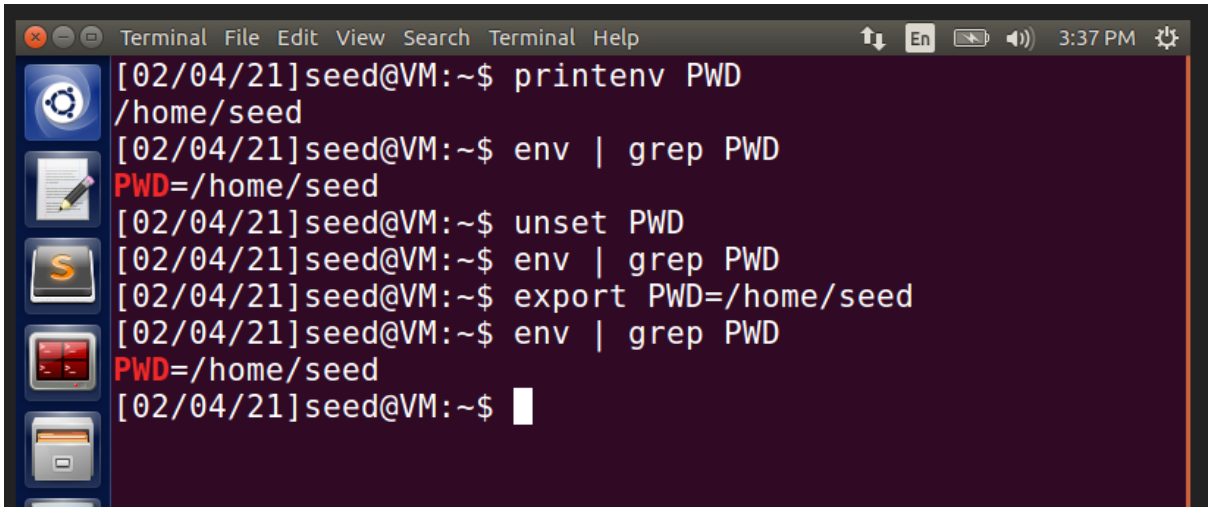
Task 1 – Manipulating Environment Variables

- Here, using the **env** command to display all the environment variables. It can also be noted that **printenv** command also does the same function as env (displaying environment variables)





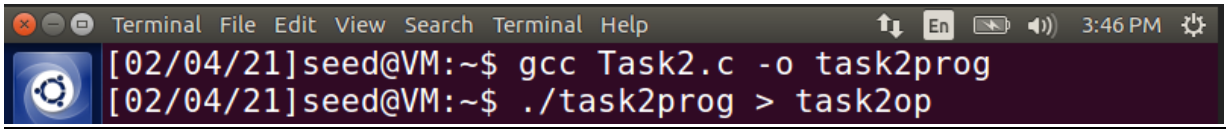
- Using the **env | grep** with a variable next to the command will display all the substring which contains the variable. For example, in the above image, using the command **env | grep PWD** will display all the substrings in which PWD is present.



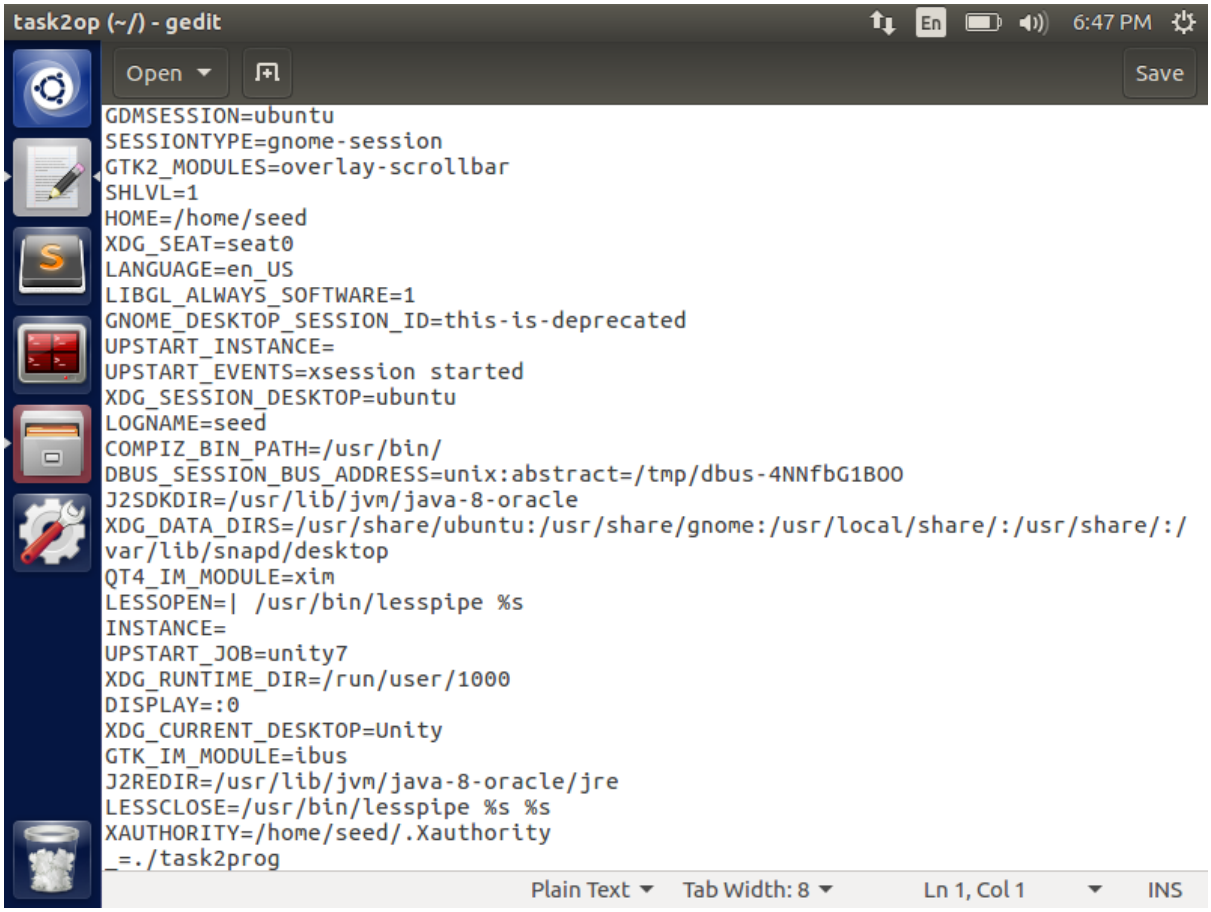
- **unset** command deletes the environment variable. As seen in the above image, after unsetting the PWD variable, if we try to view the variable using env command, it does not display anything.
- **export** command is used to set the environment variable. In the above image, using export command, environment variable is set, which can later be viewed using env command.

Task 2 – Passing Environment Variables from Parent Process to Child Process

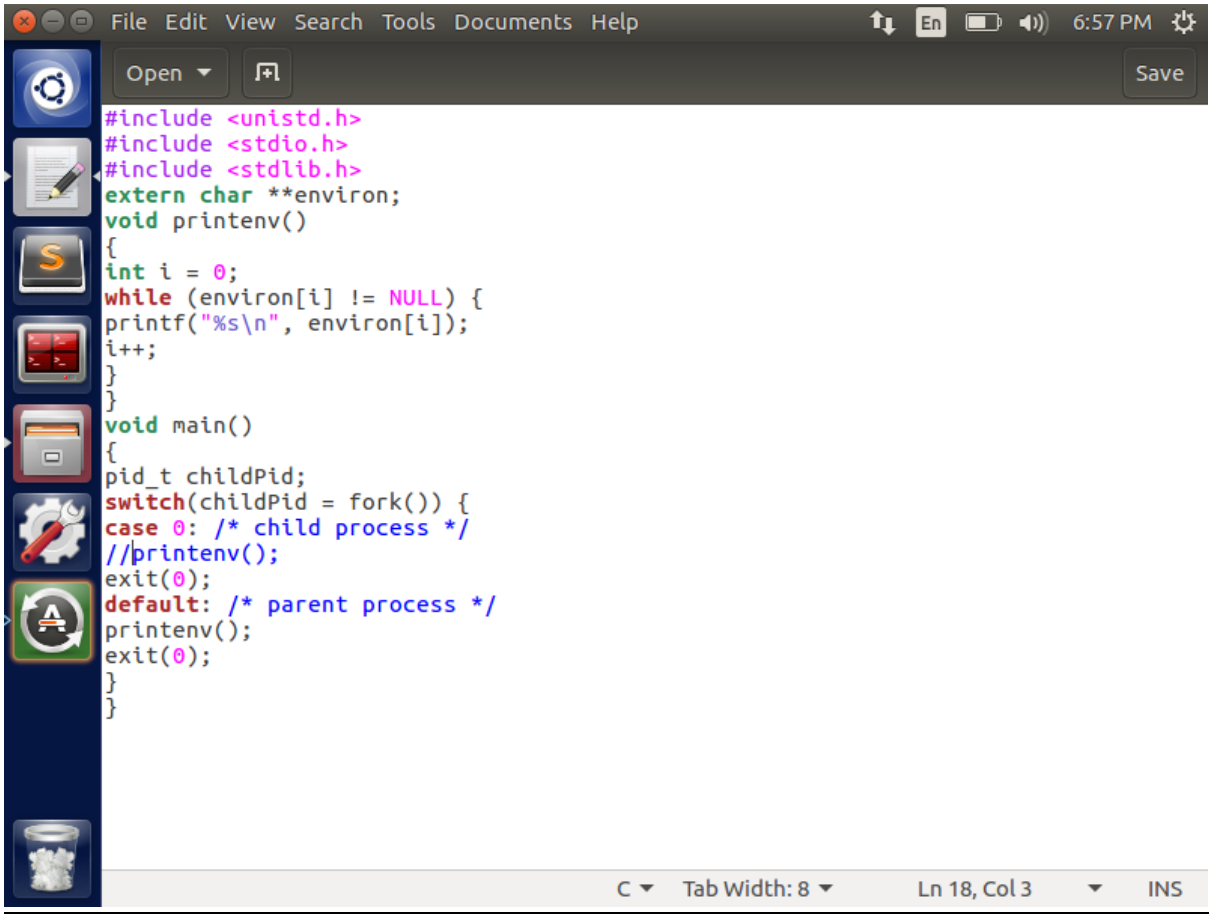
- For this task, the given program is stored as **Task2.c**, compiled and its output is stored in file **task2prog**. The environment variables from this, is stored into **task2op** file. (Child Process)



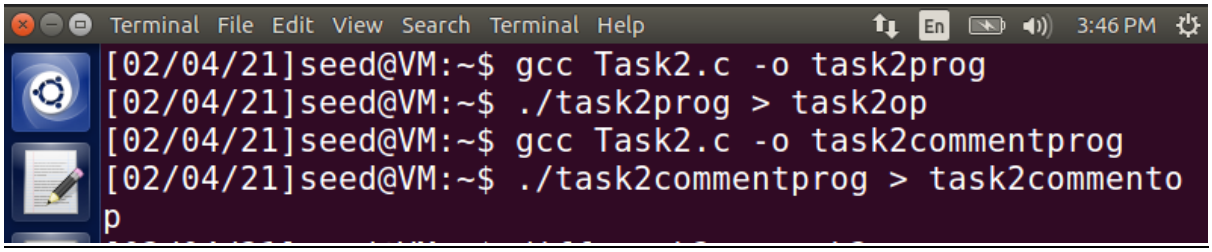
- In the task2op file, in the 70th line of the file, it reads **_=./task2prog**



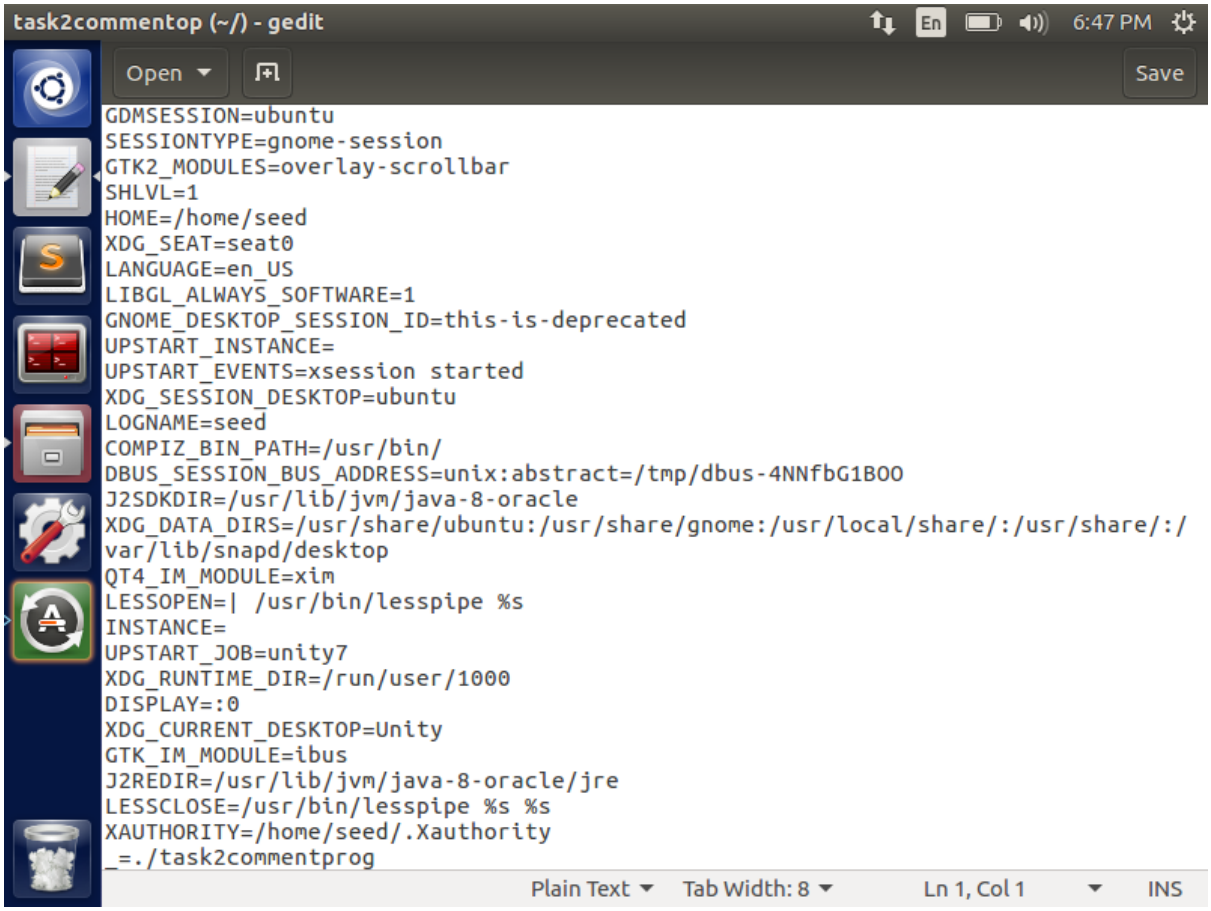
- Next, we comment out the **printenv** of child process and uncomment the **printenv** of Parent process as seen below in the code. Then run the program again.



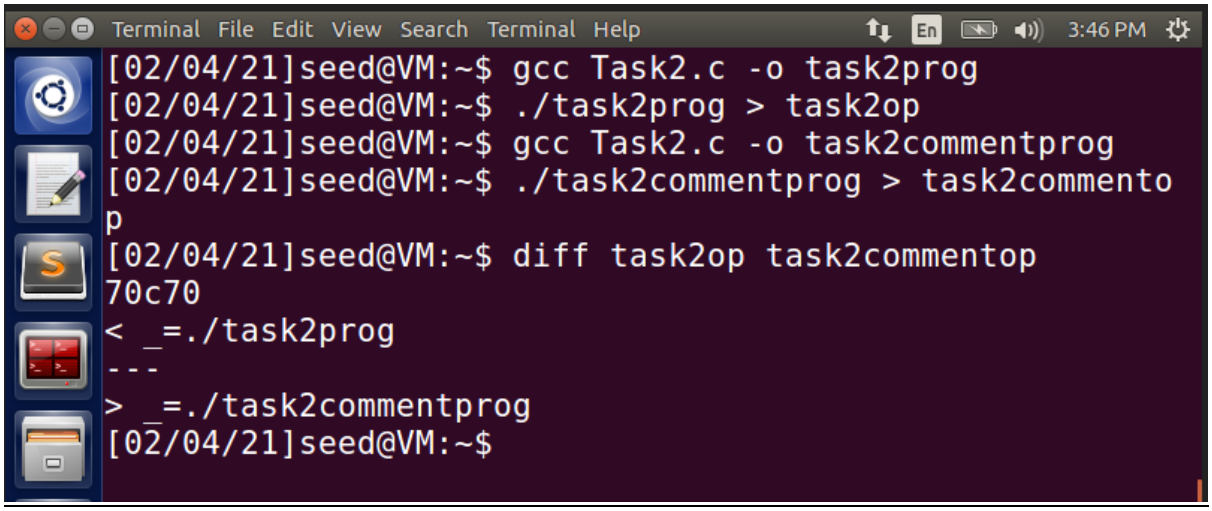
- Task2.c is compiled again and stored as **task2commentprog** and its environment variables are stored in **task2commentop** file.



- In the task2commentop file, the 70th line reads as **_=./task2commentprog**



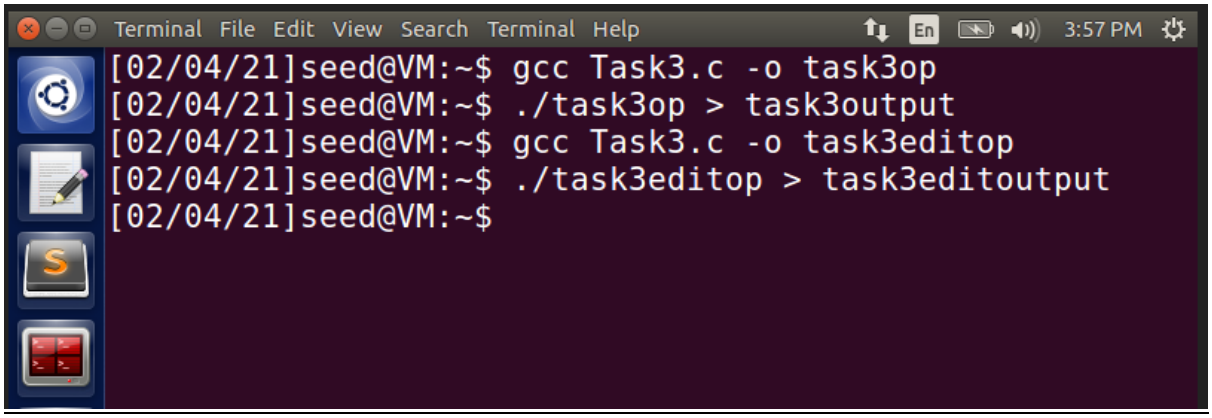
- To compute the difference between the two files, we use **diff** command, which says **70c70**, as show in the below image which means that the 70th line of child process is changed into 70th line of parent process. If the difference is compiled into the same file name, it would not result in any difference because even though they are from different process, they still point to the same output file.

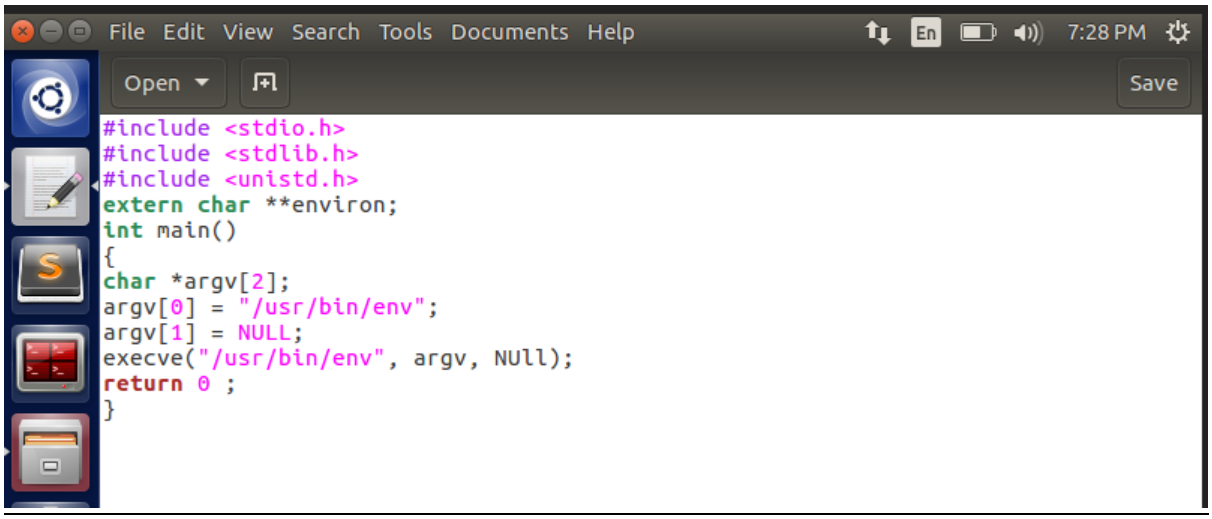


- This shows that the **printenv** passed from whichever function gets populated into the output file. Here, the environment variables are passed from Parent process (default) into child process.

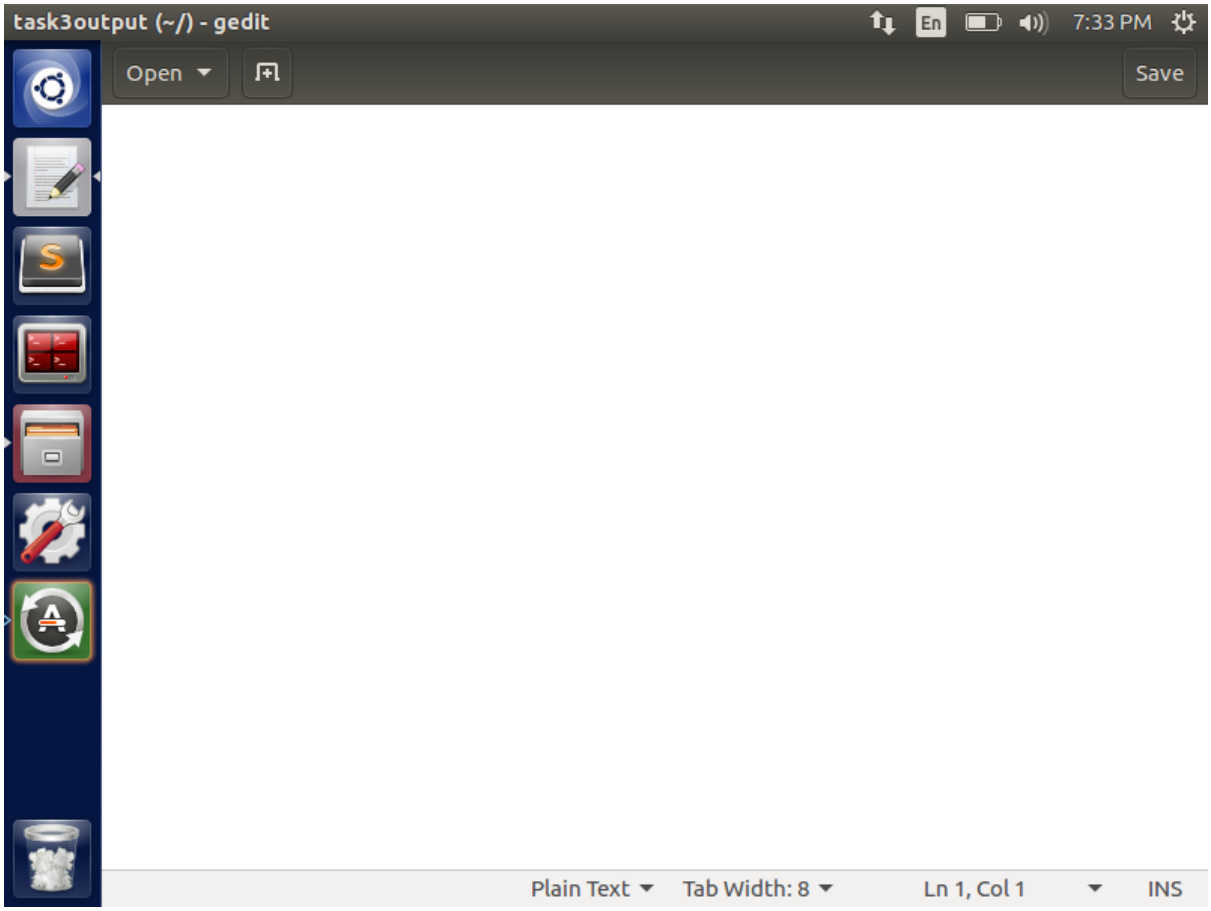
Task 3: Environment Variables and execve()

- For this task, the given program is stored as **Task3.c**, compiled and its output is stored in file **task3op**. The environment variables from this, is stored into **task3output** file.

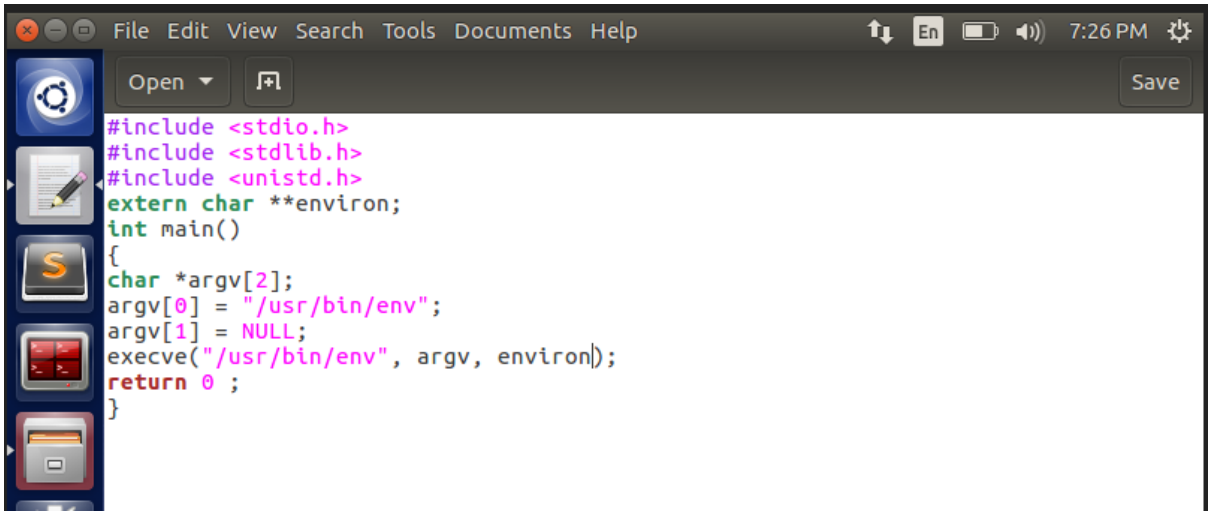




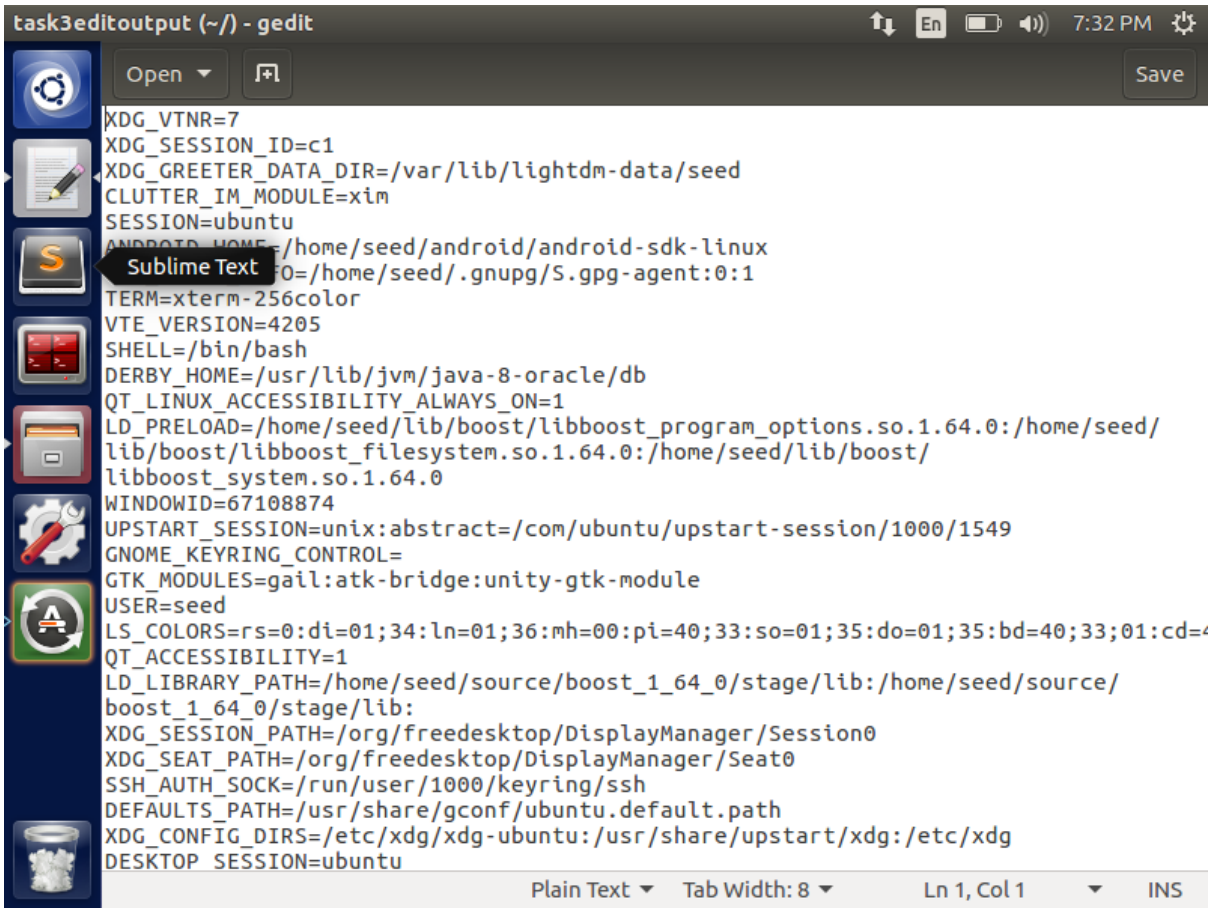
- In the task3output file, there is no output to display as seen below. An empty file.



- Now, the Task3.c code is modified and the modified **execve** function is shown below, where the argument **environ** is passed.



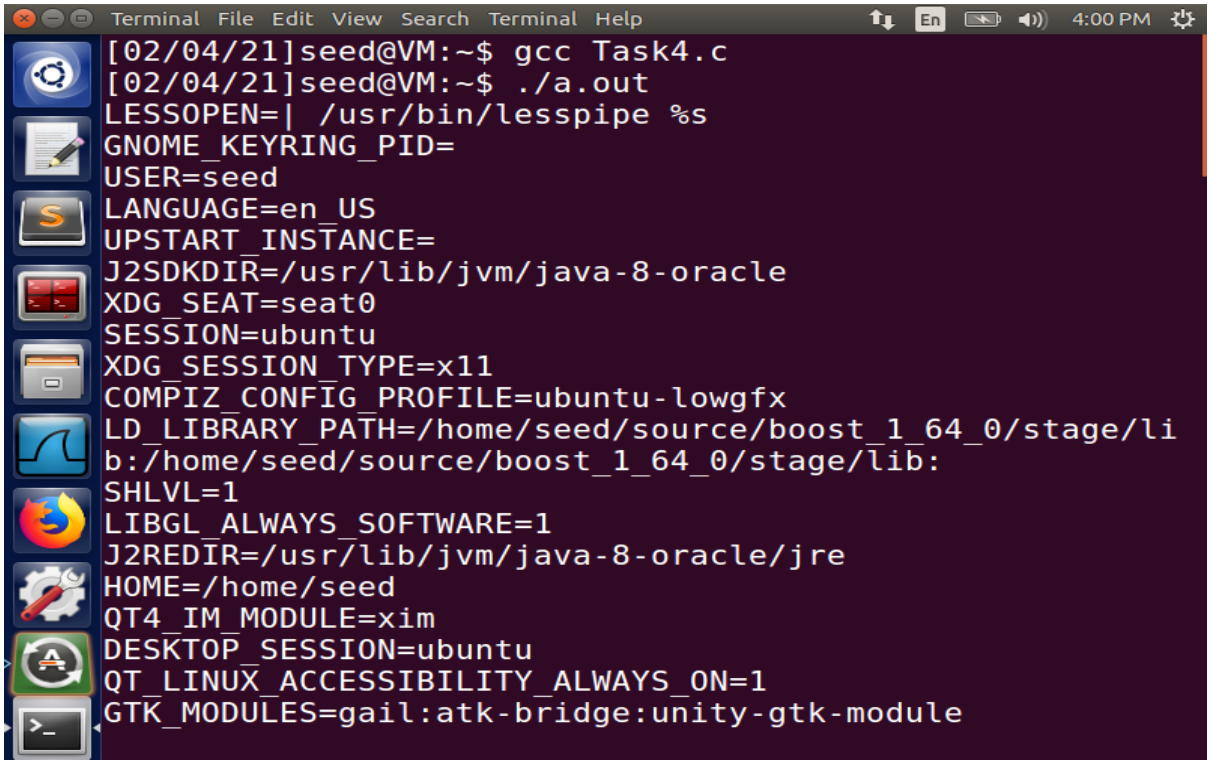
Now the output file is populated with environment variables.

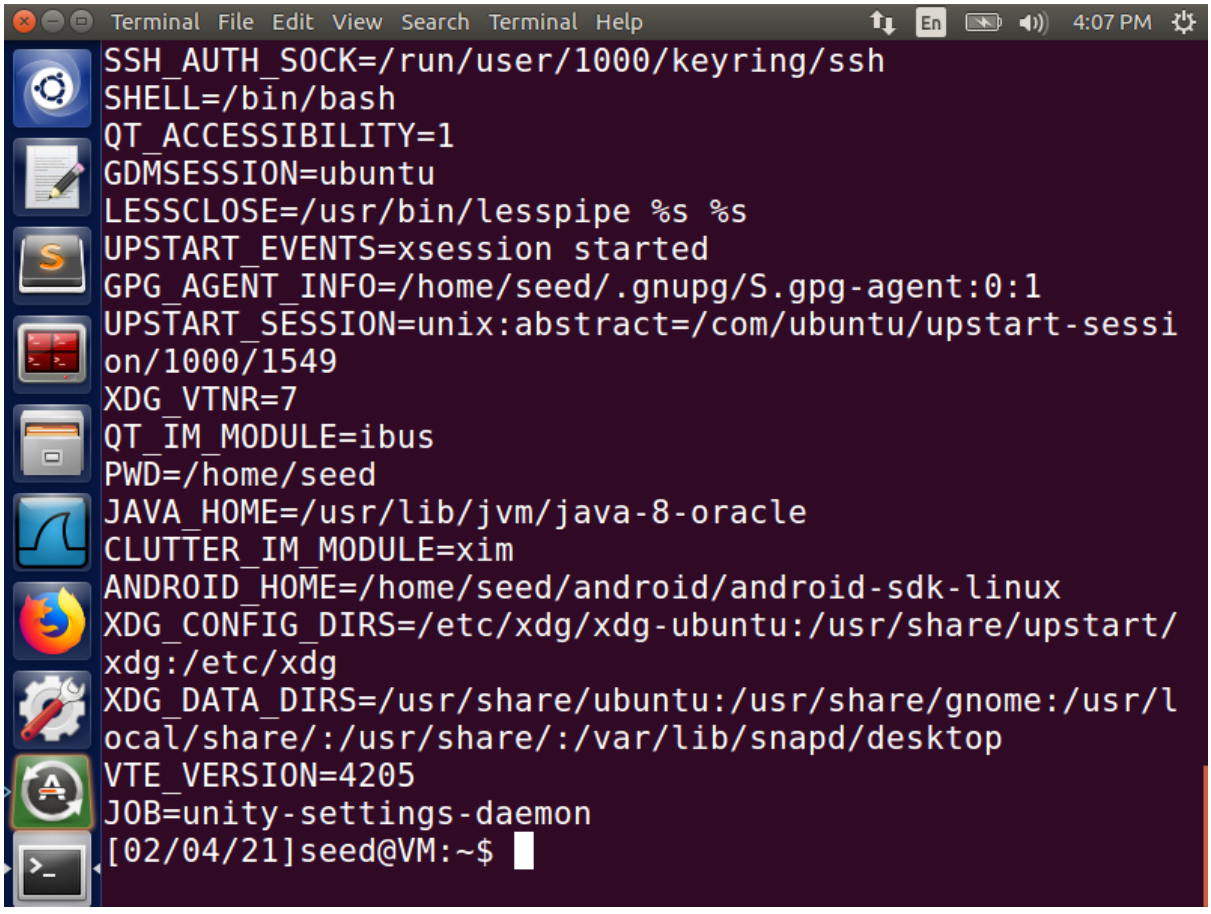


- This is because, in the first function, the argument **NULL** was passed to execve() function, which means there is nothing for the function to display. But after edit, **environ** (environment variables) were passed as arguments to the same function, which populates the same into the output file.

Task 4: Environment Variables and system()

- For this task, the given program is stored as Task4.c, compiled and run.

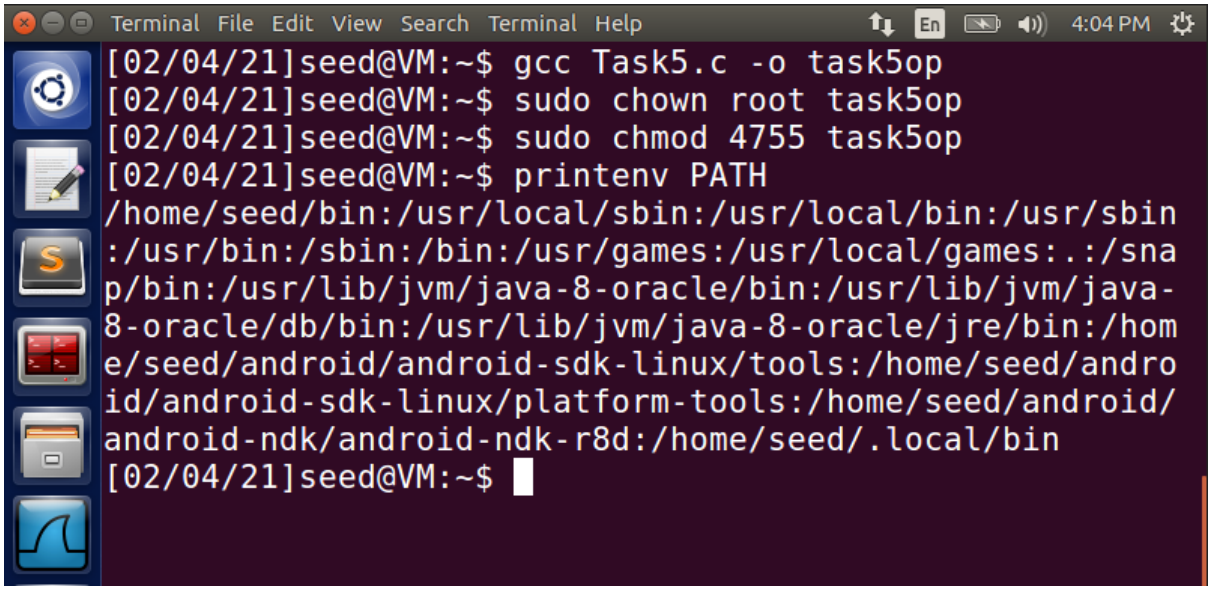




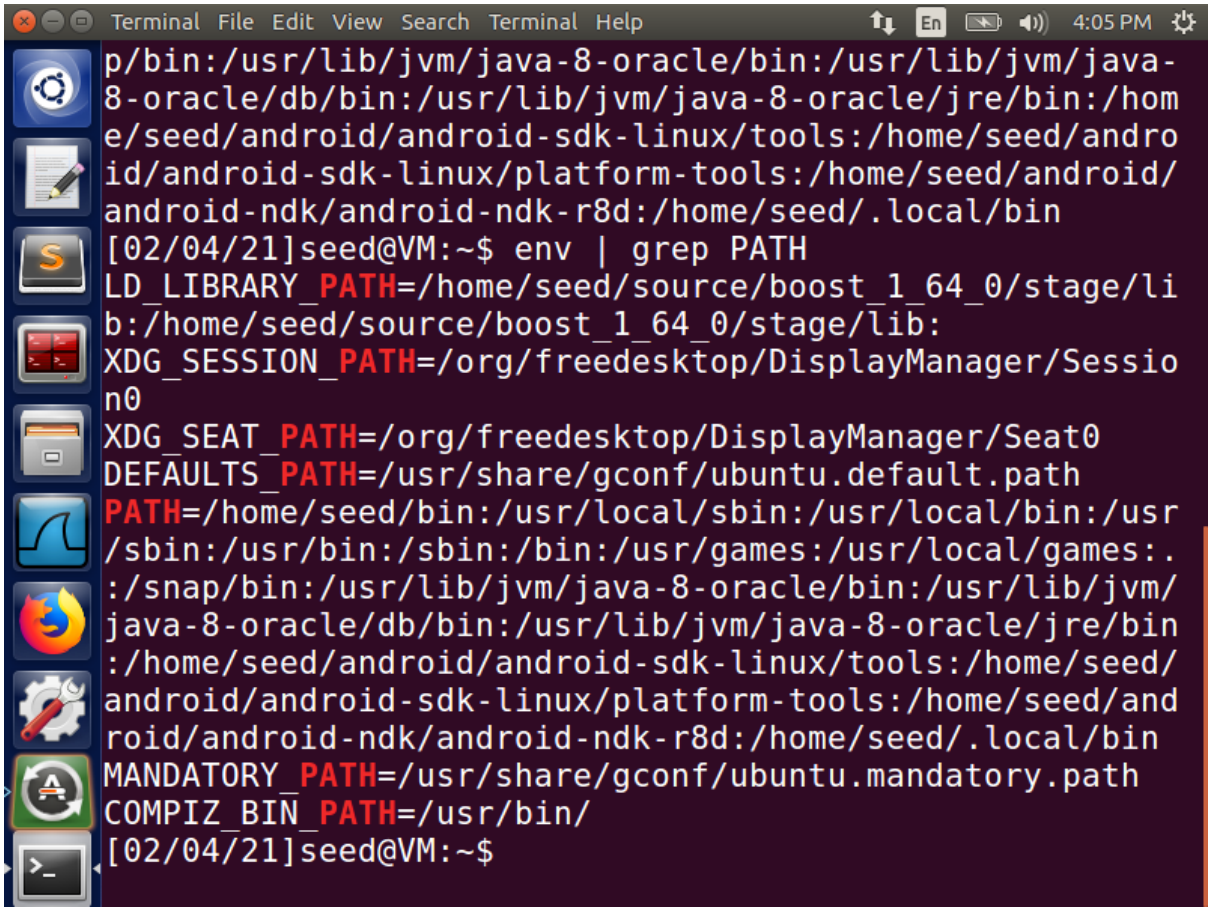
Even though there is nothing explicitly passed from our side to the program, just compiling the program and displaying the output, display the following as showing in the images above. This is because, the **system("usr/bin/env")** function loads the environment variables implicitly during run time.

Task 5: Environment Variables and SET-UID programs

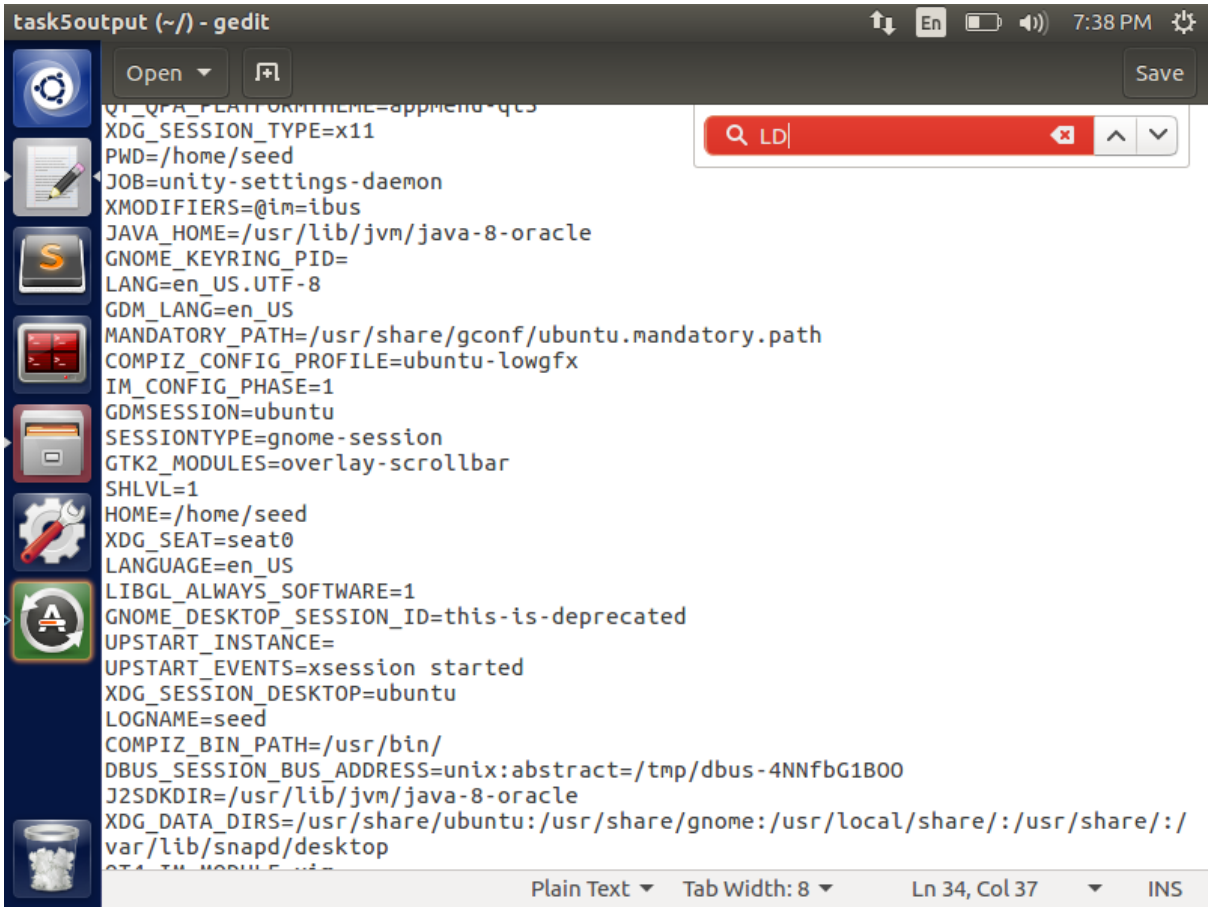
- For this task, the given program is stored as Task5.c, compiled and output is stored in task5op. Now, to access the op file the ownership of the file is changed and changed to root using **chown root** command.



- Here, using env | grep PATH, we check the LD_LIBRARY_PATH and it is seen that the child process inherits the PATH variable.

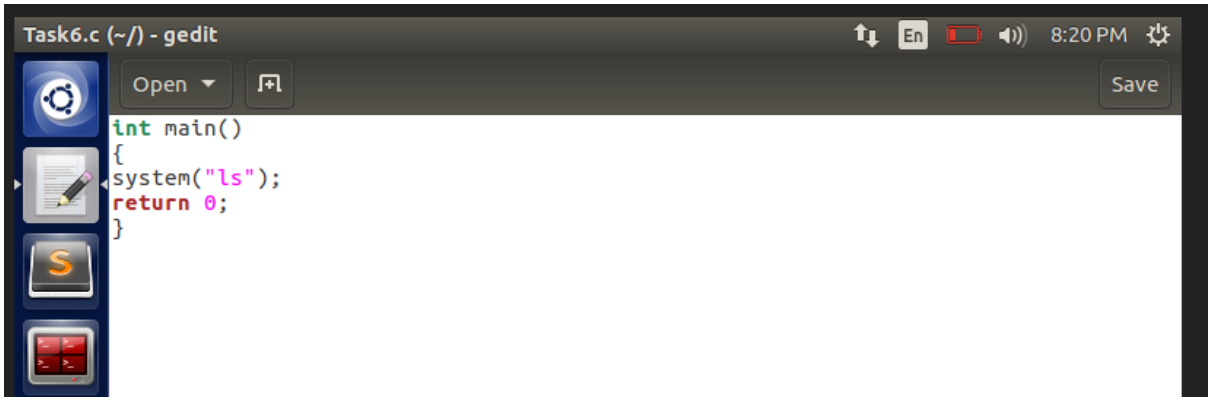


- But if we check for the LD PATH variable in the output, it is not present. This is because of ownership issues where the child process may not be able to inherit all the environment variables from the parent and that is the reason why LD_LIBRARY_PATH is not inherited in the output file.

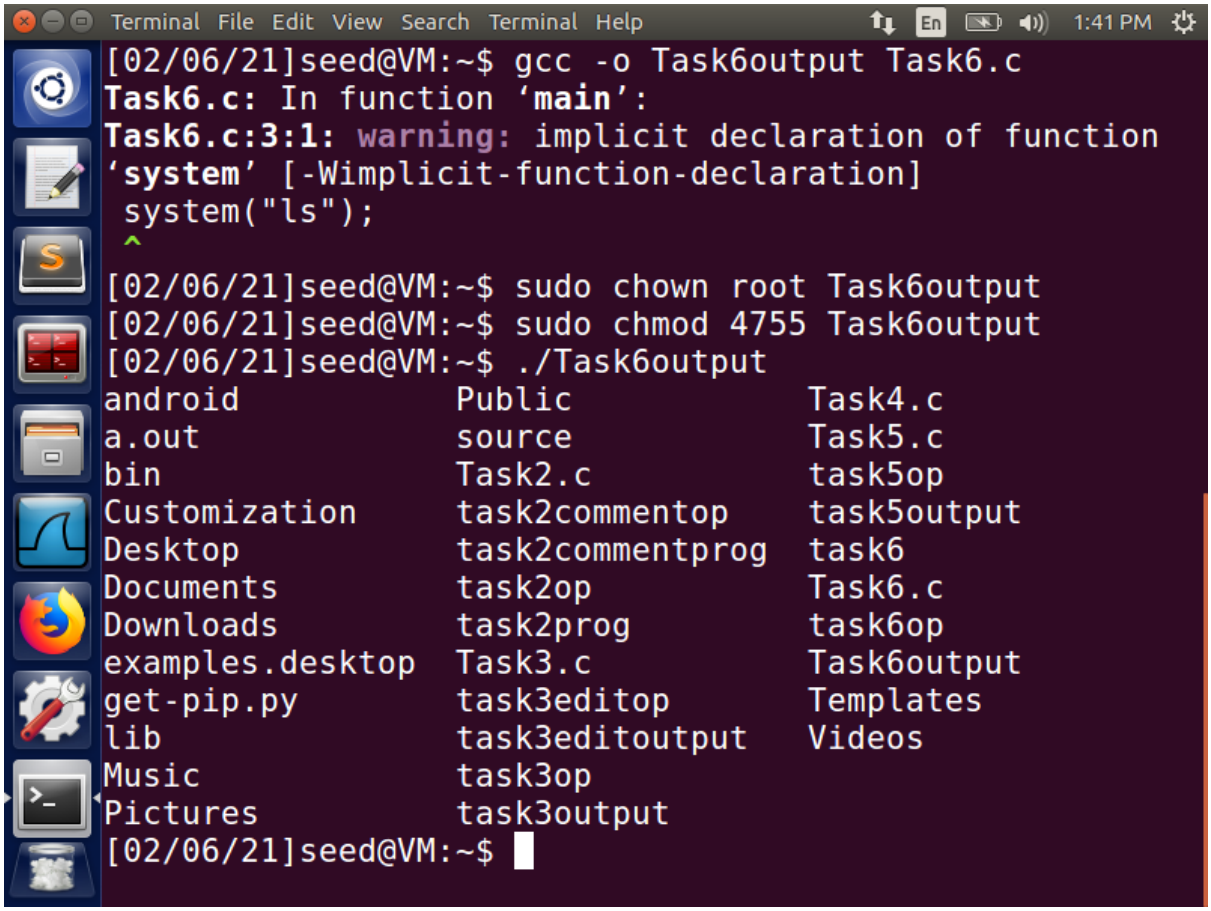


Task 6: The PATH Environment Variable and SET-UID Programs

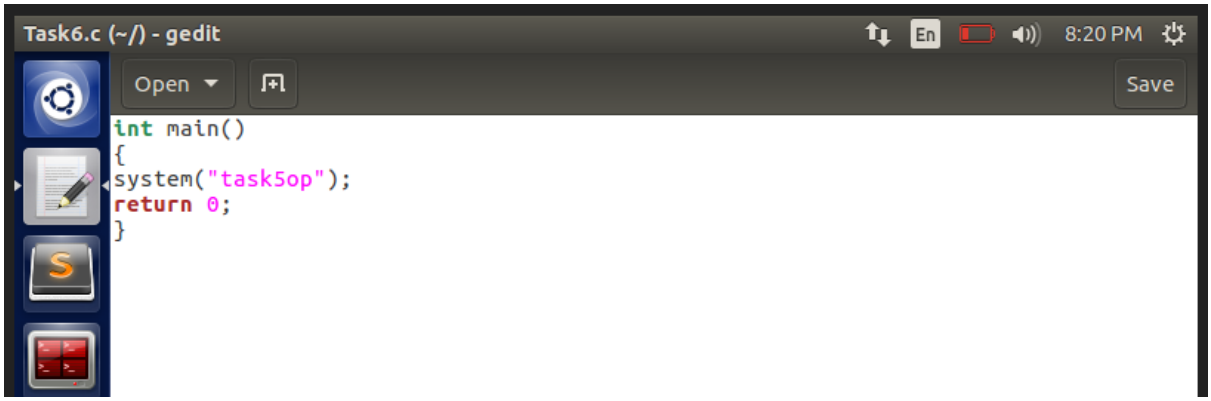
- For this task, the given program is stored as Task6.c, compiled and output is stored in Task6output. Now, to access the op file the ownership of the file is changed and changed to root using **chown root** command.



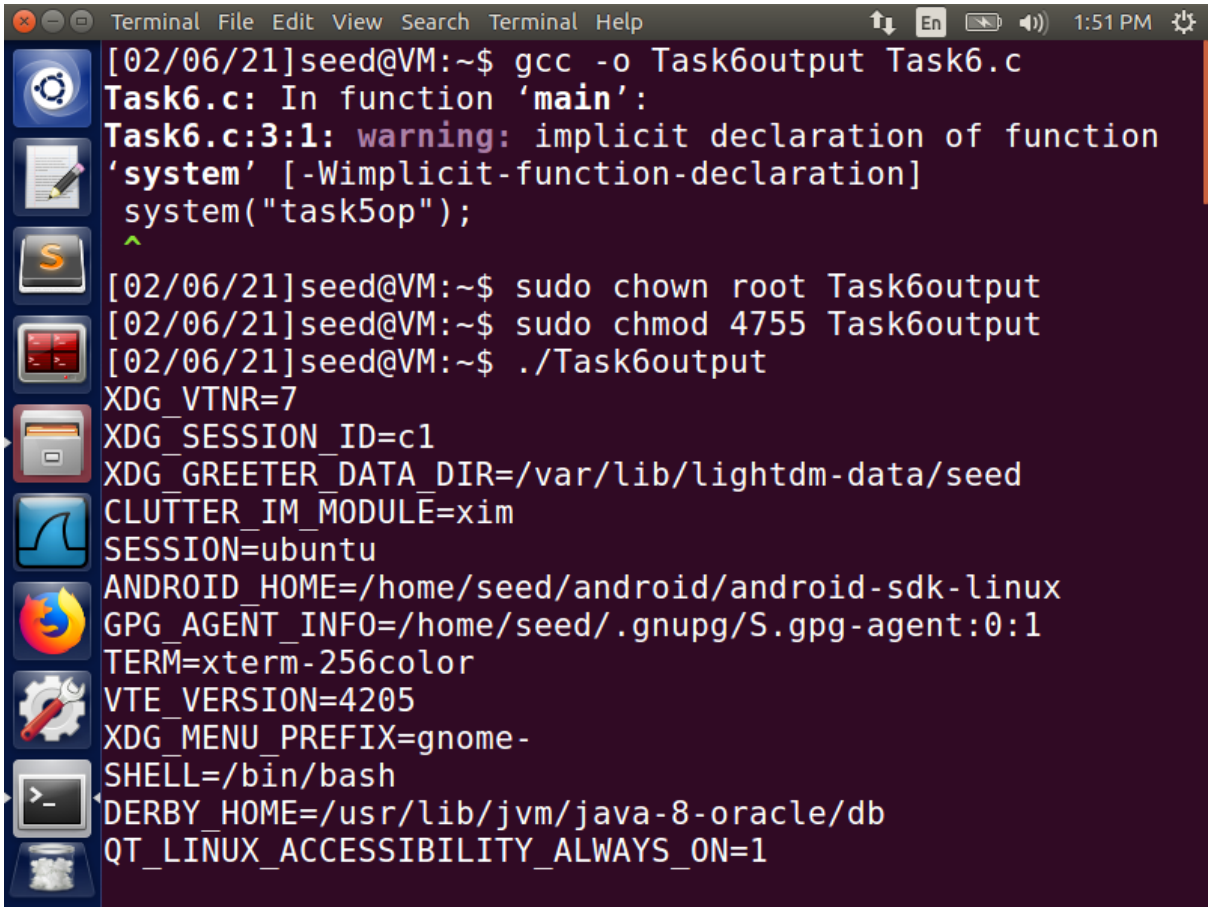
- After providing ownership to the file, on running the output file, it displays all the contents of the file, since the system() has ls.
- The output is displayed below as shown:



- Now, the contents of the file Task6.c is changed and "task5op" file is passed as an argument to the system() function.



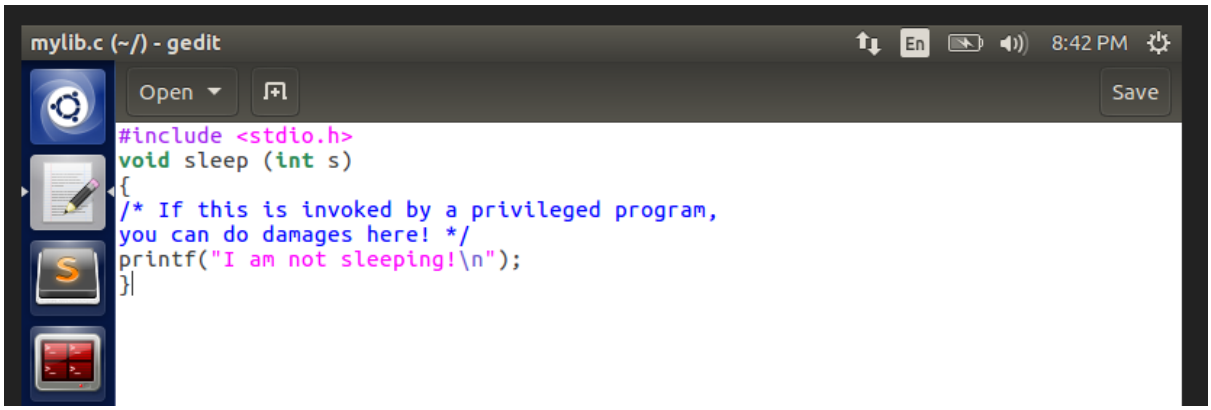
- Now, the code Task6.c is run again and the output is again overwritten into the file Task6output. Then the root privileges are provided to the file using **chown root** command.



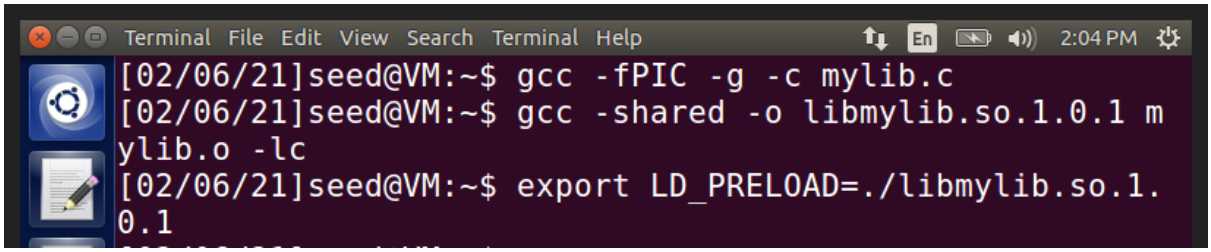
- On running the Task6output file, it runs **task5op** and displays the environment variables of Task 5. Here, we use the system() function to run another program's output file and display the same instead of ls command. It can be observed that the output file is run only after providing the root privileges.

Task 7: The LD_PRELOAD Environment Variable and SET-UID Programs

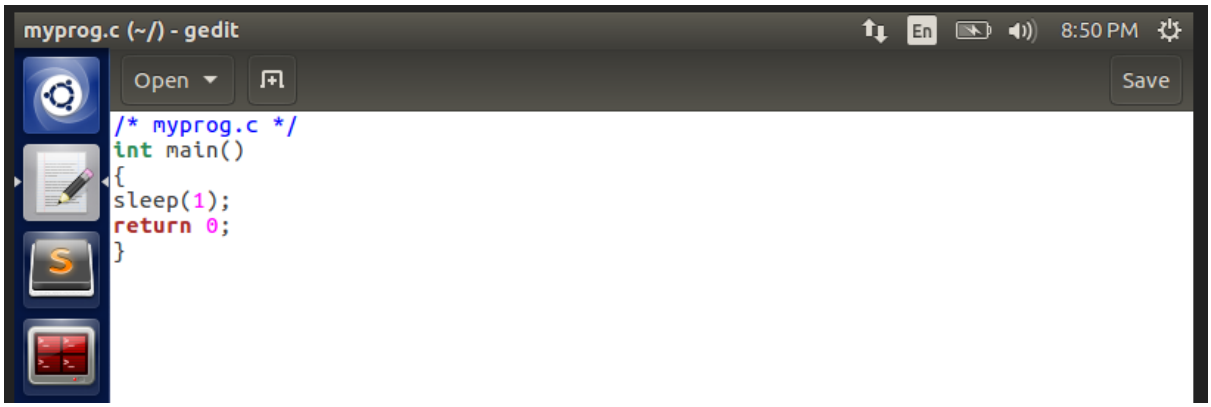
- For this task, the given program is stored as mylib.c



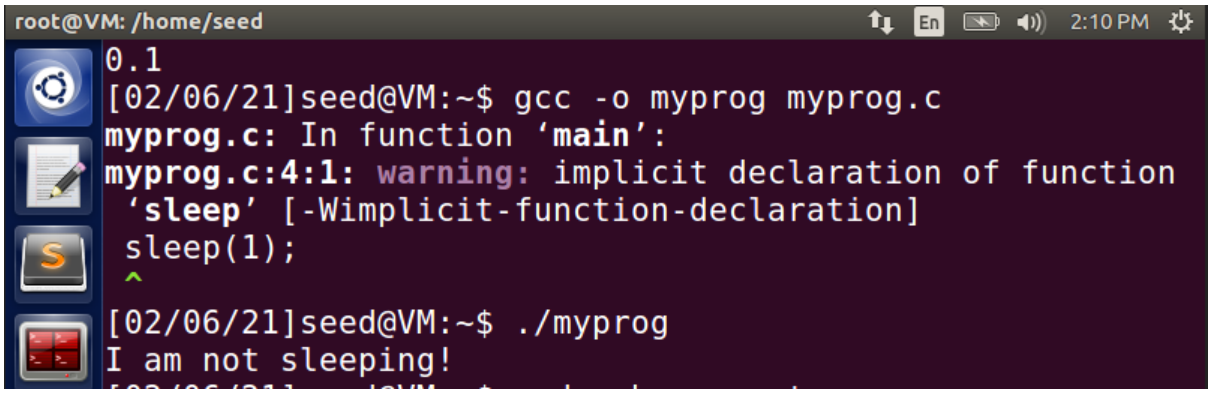
- The above program is compiled using the following commands, as shown below. LD_PRELOAD variable is set after this.



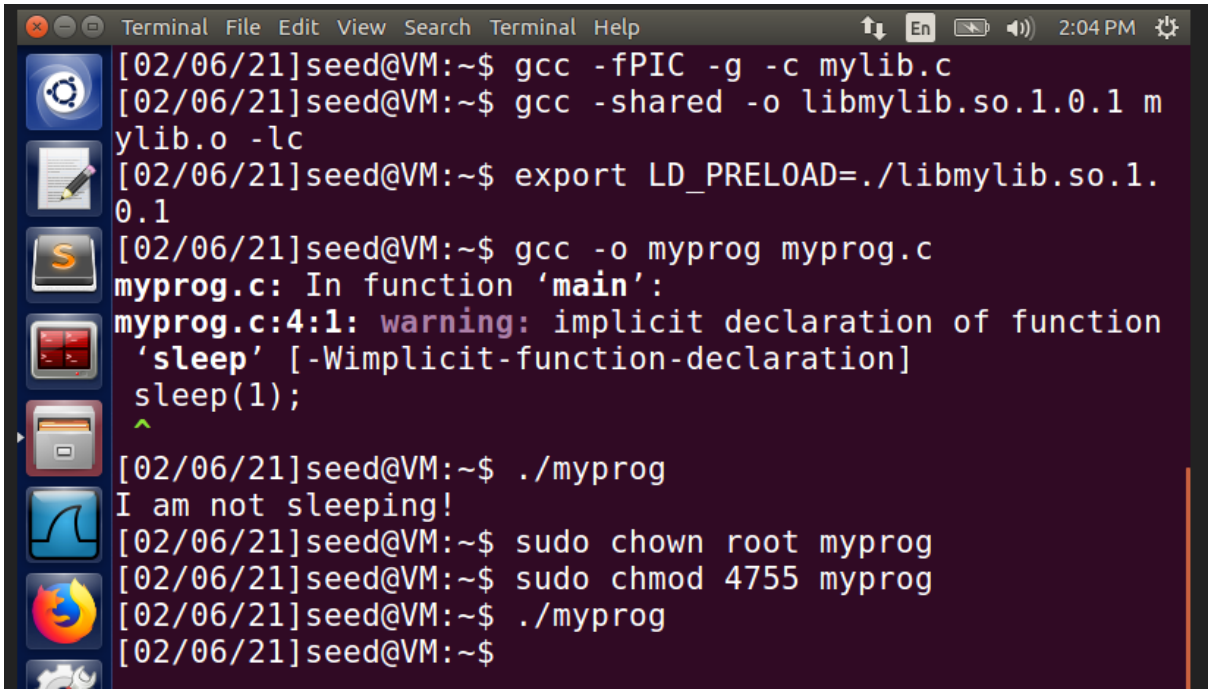
- The given program is saved as myprog.c



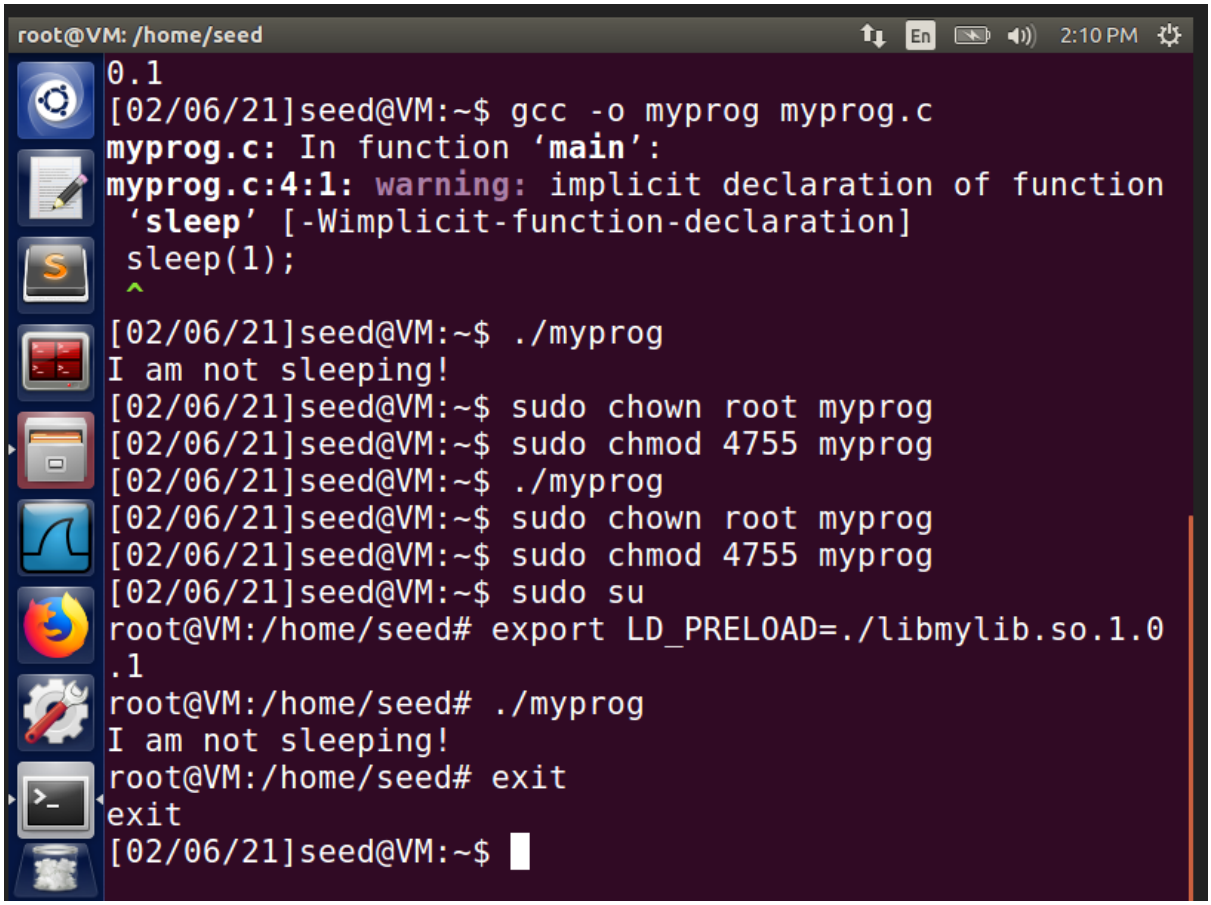
- The above program is run as a normal user and the output is seen below:



- Here, we can see that the myprog.c is run as a normal user and the printf present in mylib.c "I am not sleeping" is displayed. This shows that the sleep() function present in myprog.c is overridden by the function in mylib, which was earlier compiled and pre-loaded.
- Now, we give root permission to the myprog function and try running the program again. We can observe that on giving root permission, the program performs the root task (task defined in the myprog.c) and the program goes to sleep, which is what is defined in the program.

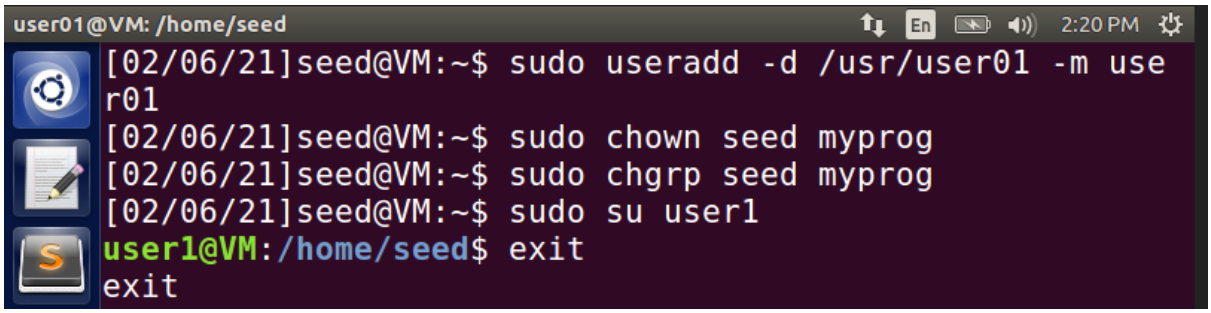


- Now, we again give root permission to myprog. This time, before trying to run the program, we export the LD_PRELOAD variable and then try running the program again.

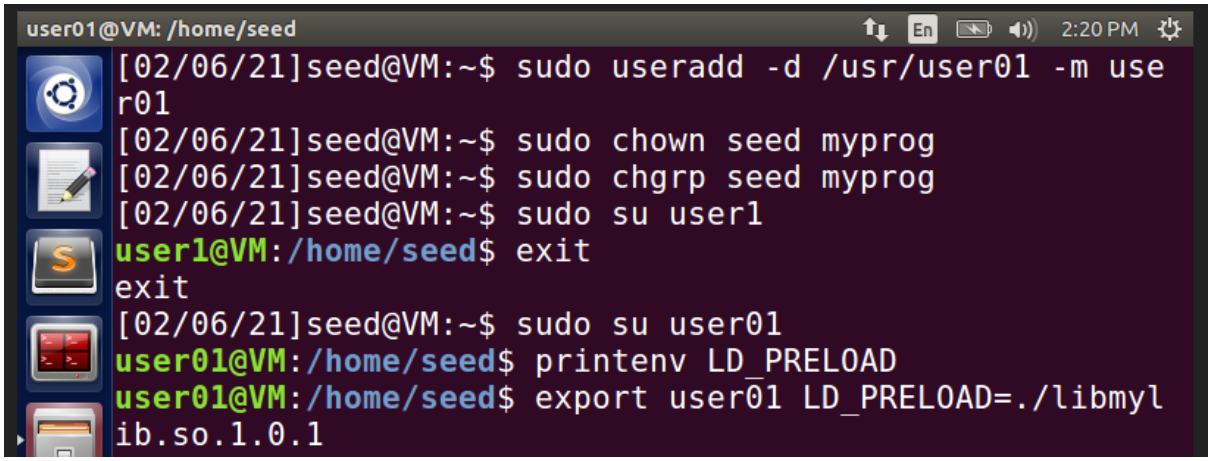


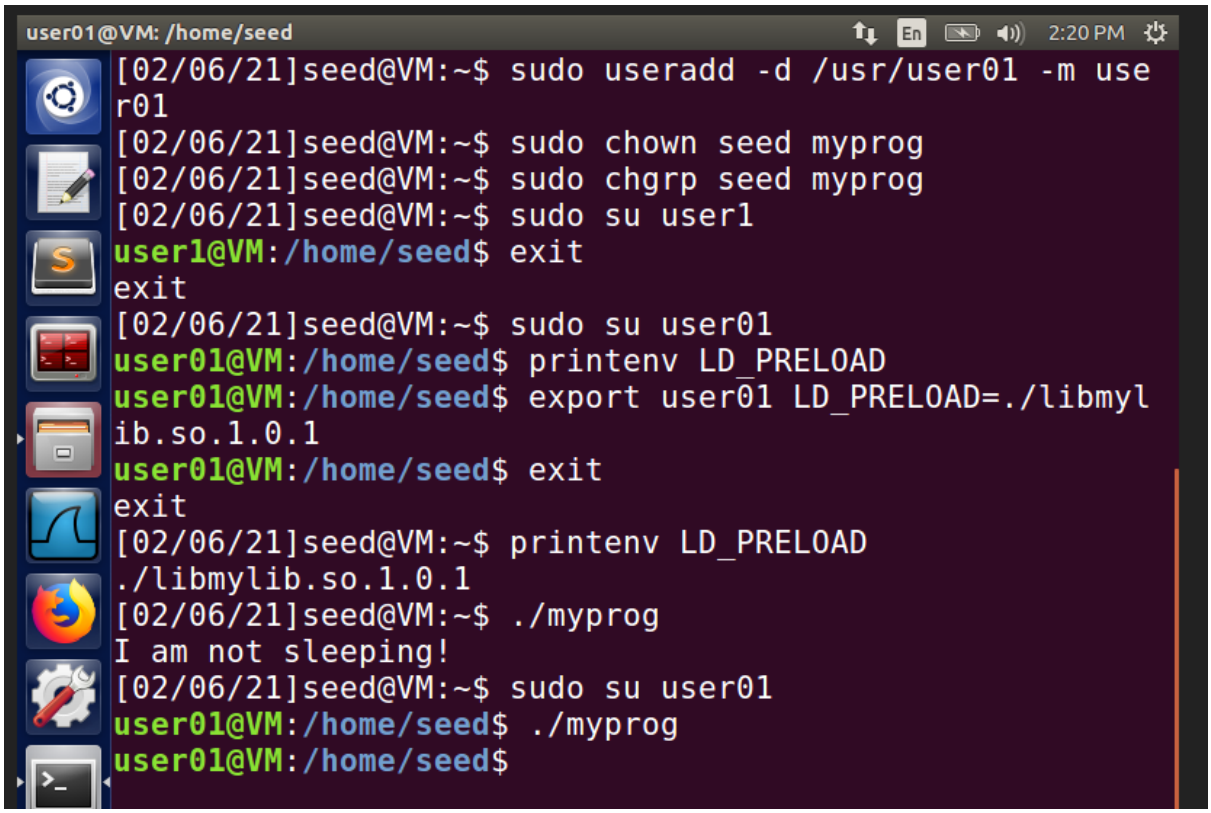
- In this case, the general sleep() function is overridden with the mylib.c printf() function, which is loaded through the libmylib.so.1.0.1

- Now, a new user **user1** is created and make myprog a Set-UID seed program as shown below:



- LD_PRELOAD variable is exported into that user account and myprog is run as below:

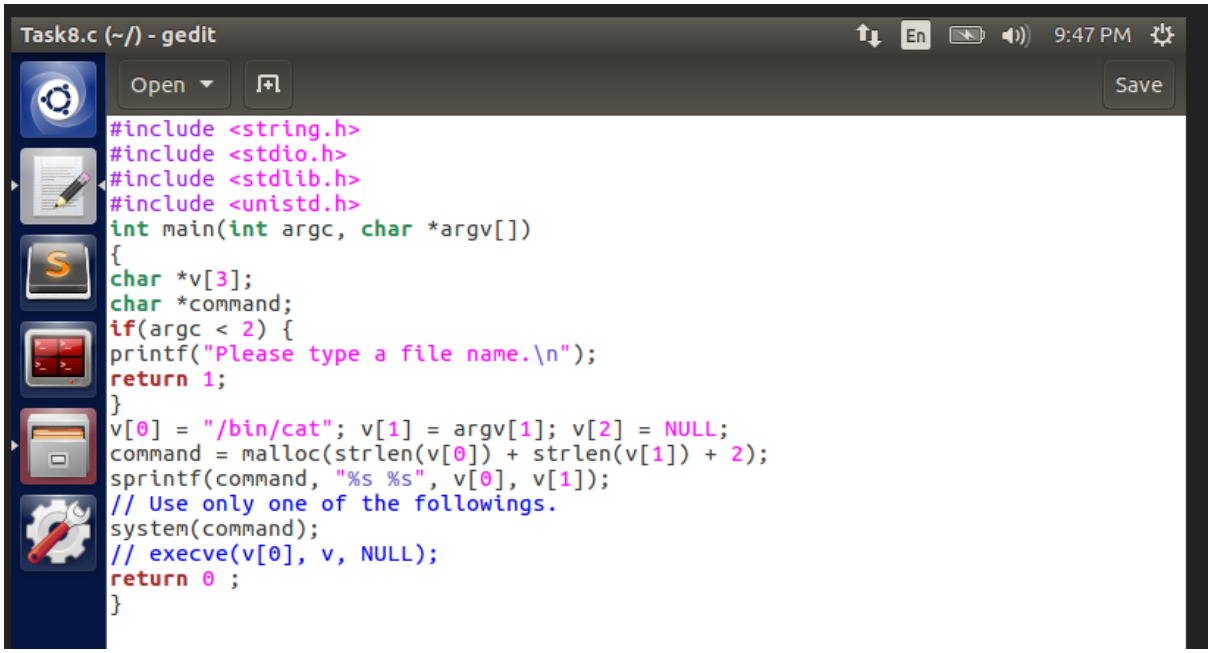




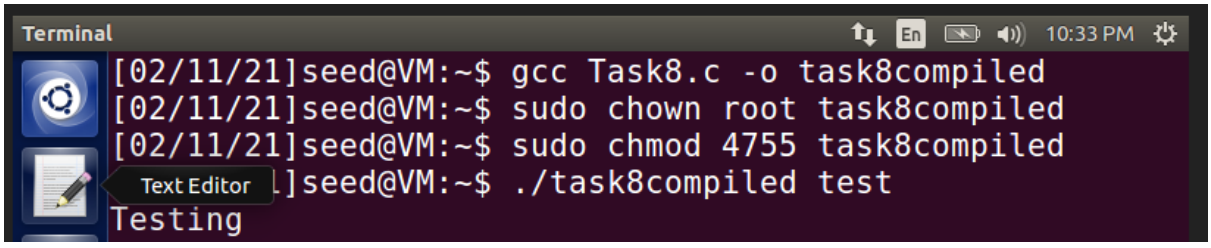
- It is observed from the above images that if the LD_PRELOAD variable is exported into the user01 account, then running the myprog will override the general sleep() function and execute the commands we have defined in mylib. But when we do not export the LD_PRELOAD variable, the general sleep() function is invoked.
- This is because, only when the user account and the LD_PRELOAD exporting user account are the same, only then our defined mylib function will work or else default functions will only work.

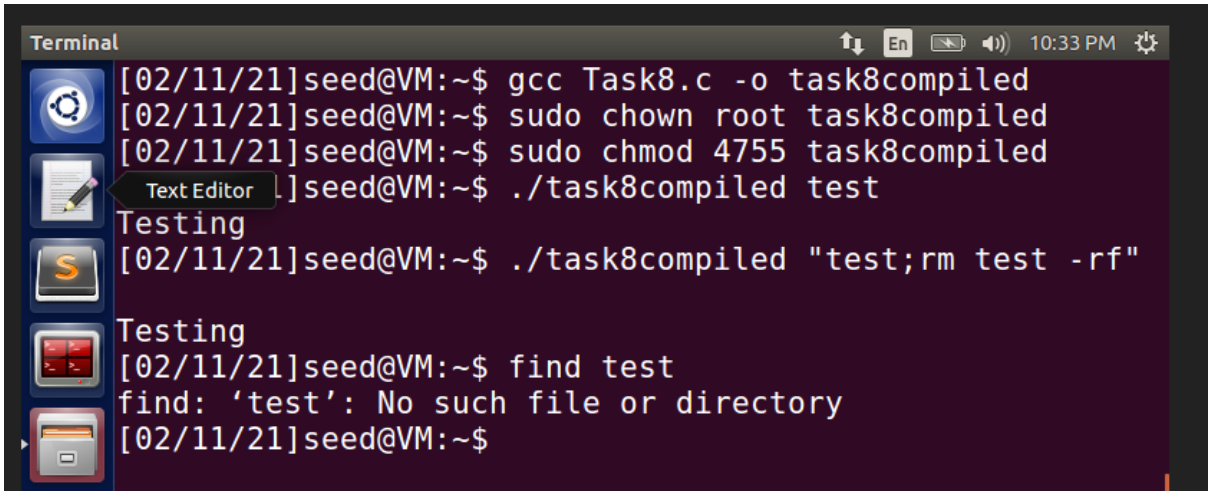
Task 8: Invoking External Programs Using system() versus execve()

- For this task, the given program is stored as Task8.c

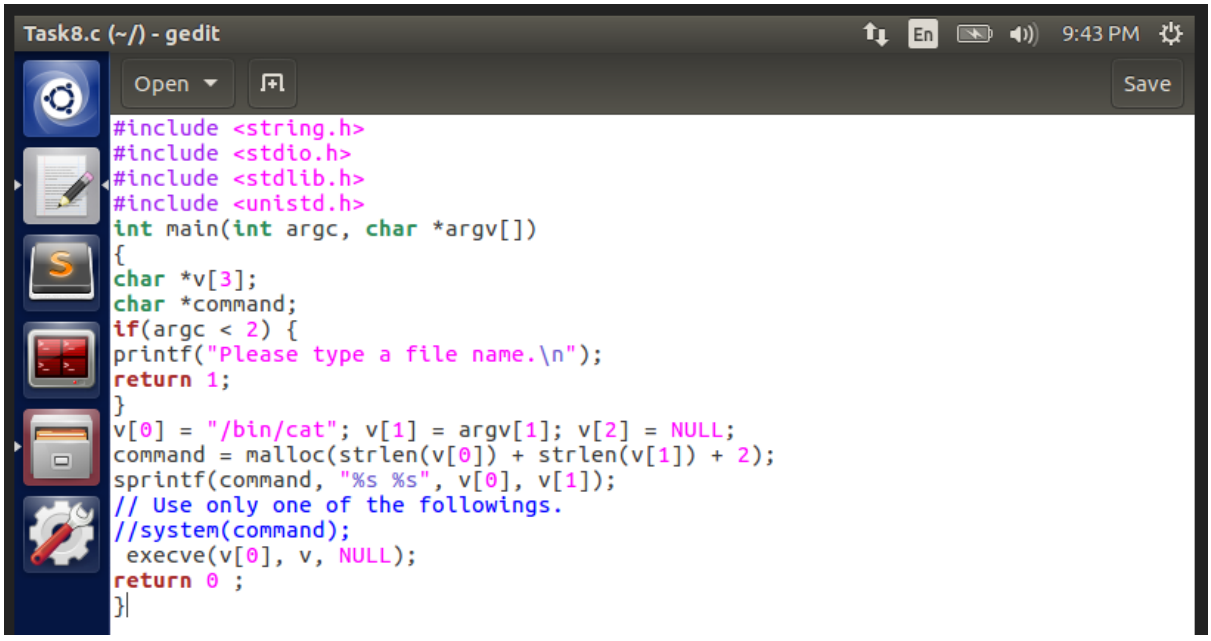


- The Task8.c is compiled and output is stored into task8compiled. Root access privileges are given to task8compiled using **chown** and **chmod** commands. A test document is created and run with the task8compiled. The test file is read and its contents are displayed.

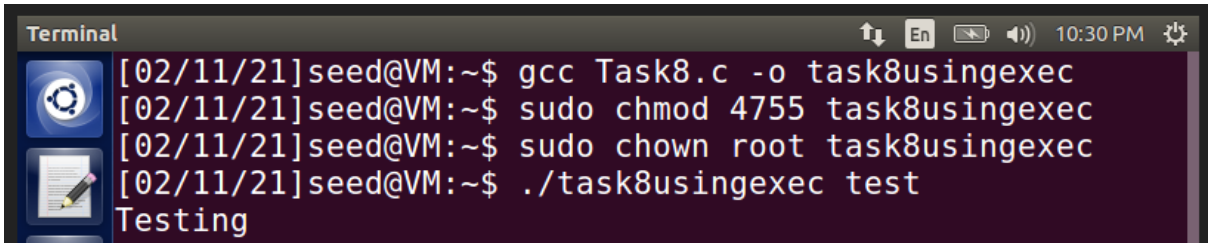




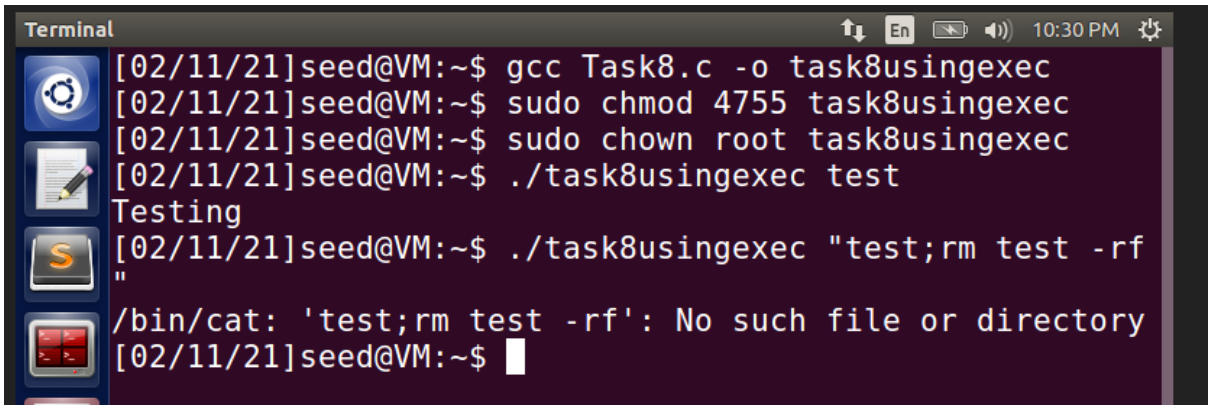
- Here, if we add ;rm test -rf after the test document, it deletes everything from the root directory, in this case, it deletes the test file. That is the reason why it says “No such file or directory” if we try to find the test file.
- Now, we comment out the system() command and uncomment the execv() function in Task8.c file.



- The Task8.c is compiled again and output is stored into task8usingexec. Root access privileges are given to task8usingexec using **chown** and **chmod** commands. A test document is created and run with the task8usingexec. The test file is read and its contents are displayed.



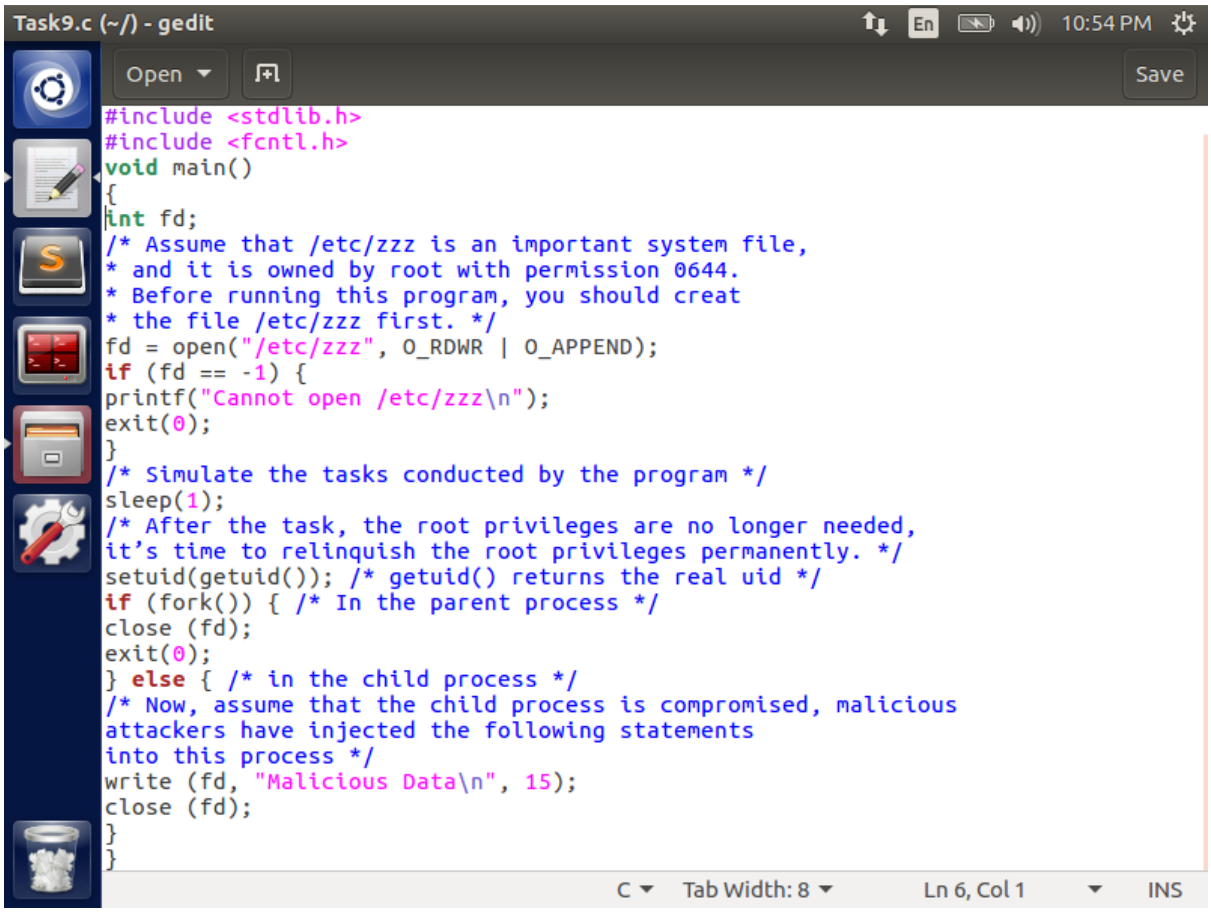
- Here, if we add ;rm test -rf after the test document, unlike system() command which deletes everything, in the case of execv() command, it does not delete anything.



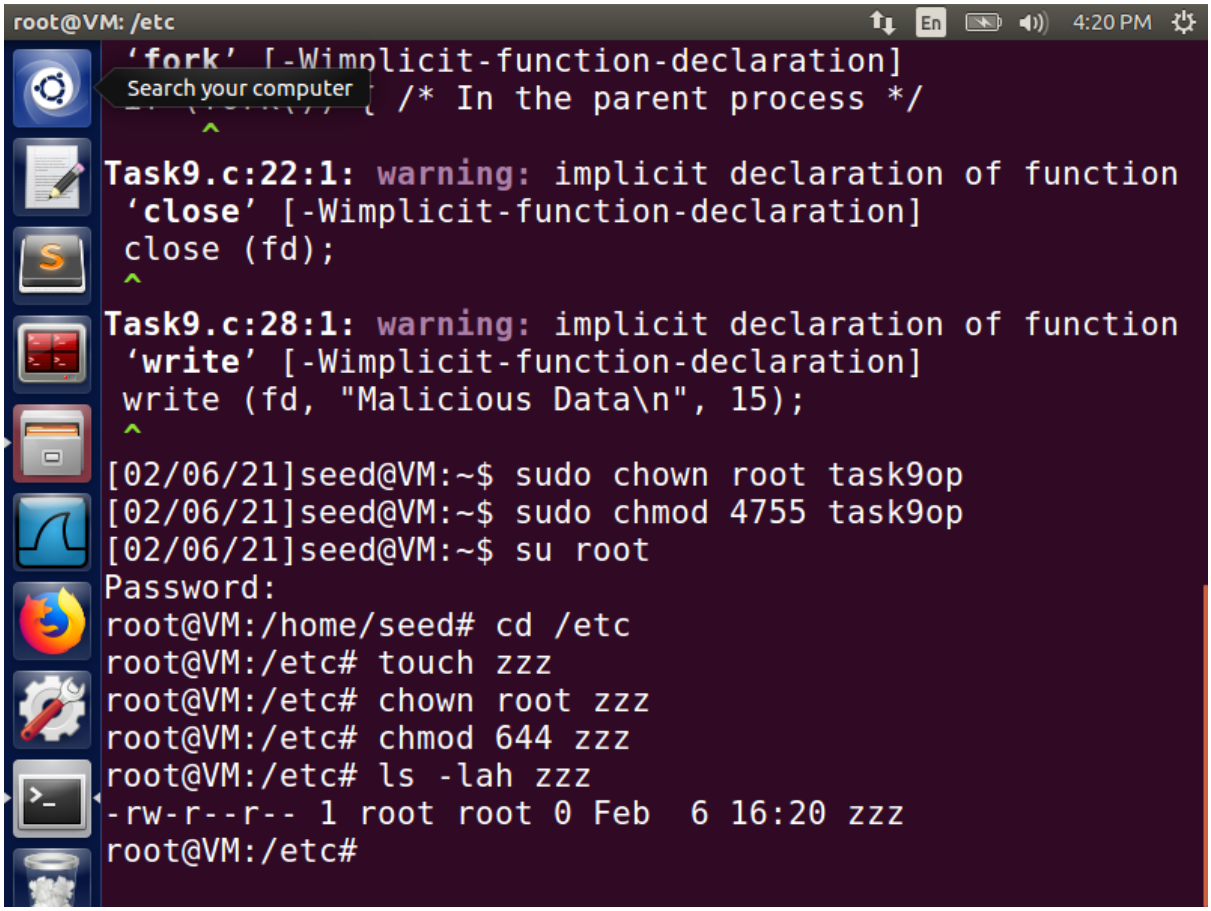
- Rather, it treats the whole of **"test;rm test -rf"** as a single string and tries to search for that string in the directory which will result in the above output. This is because, **execv() function does not have the root access privileges unlike system() function**.

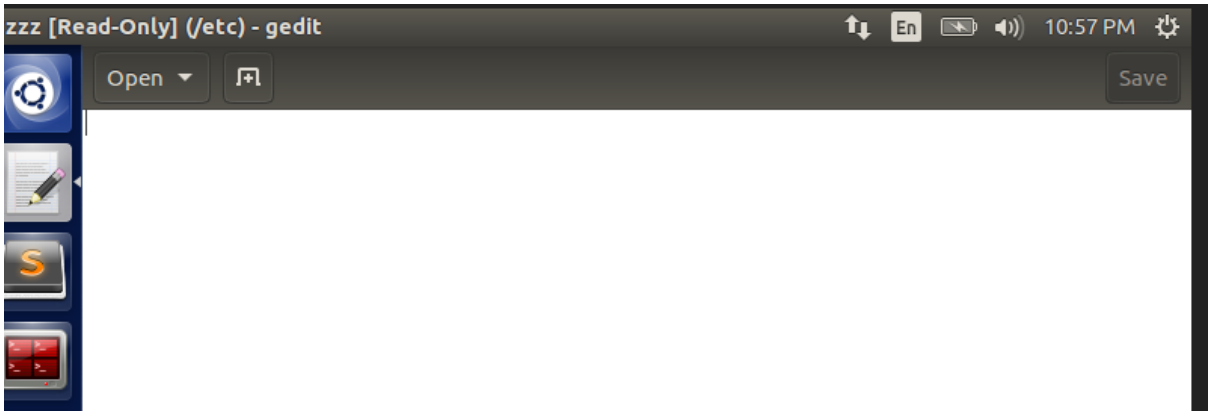
Task 9: Capability Leaking

- For this task, the given program is stored as Task9.c

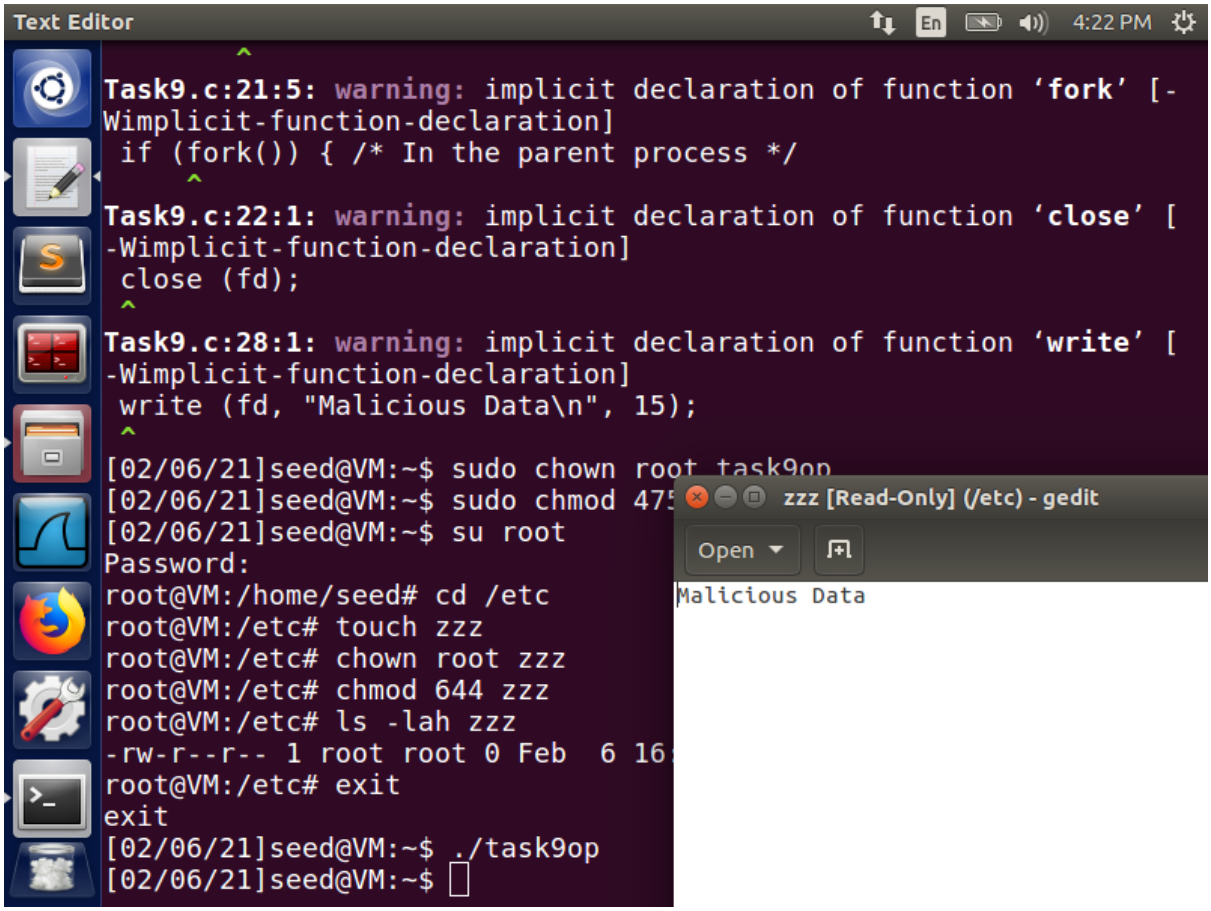


- Root privileges are given to the file task9op and we find that the file zzz is empty at first.





- Now, we run the same program as a normal user with the command **./task9op** and we find that “Malicious Data” is written into the zzz file as seen below.



This is because, when the user has root access, it does not allow them access the malicious file and edit its contents. But when the privileges are downgraded, we are able to launch the attack and write into the file.