**Homework 1**

**Environment Variable and Set-UID Program Lab**

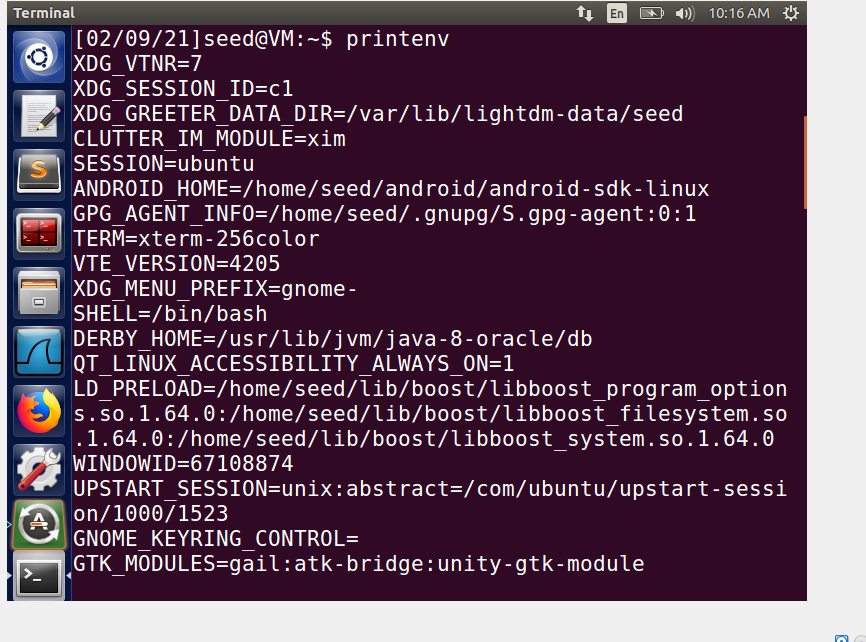
**Name:- Aparna Krishna Bhat**

**UTA ID:- 1001255079**

**Task 1: Manipulating Environment Variables**

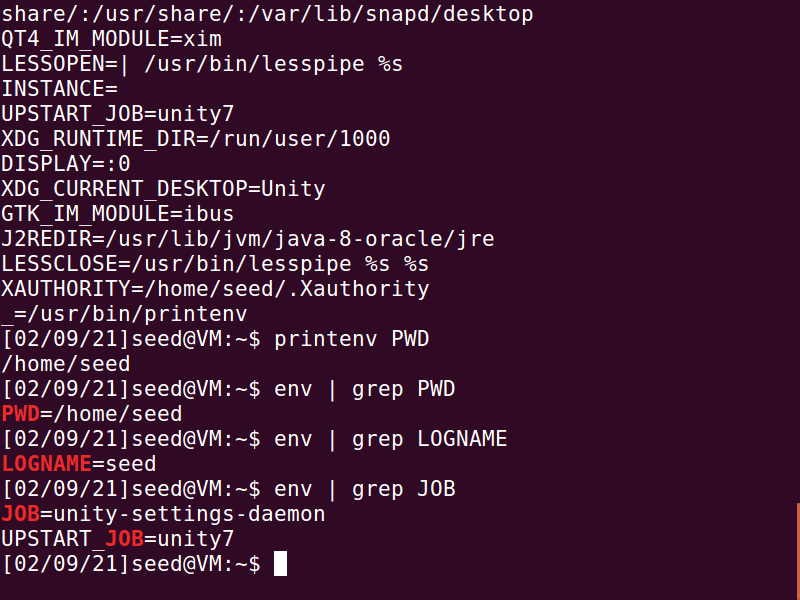
**The aim of this task is to study the commands that can be used to set and unset environment variables.**

In any shell there are a good number of environment variables, set either by the system, or by own shell scripts and configuration. All the environment variables can be printed on the terminal using the ***printenv*** or ***env*** command.



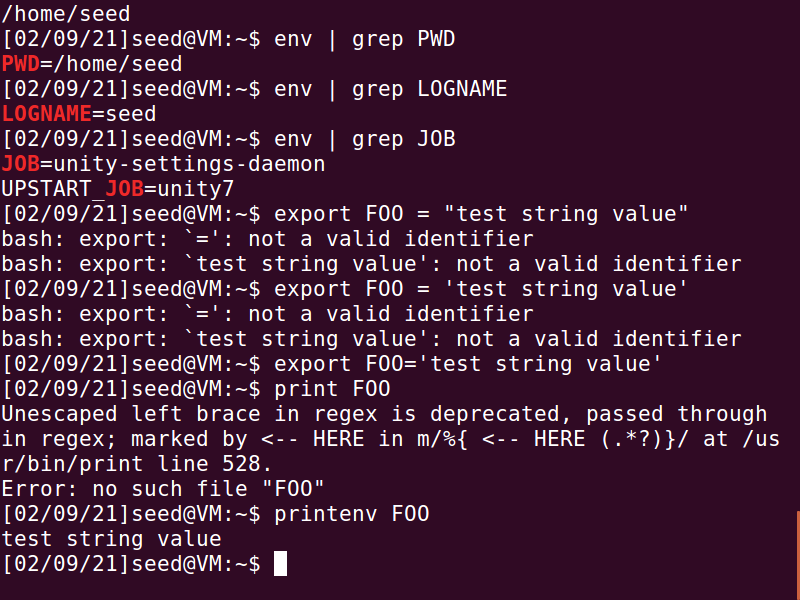
**Fig 1:- Output for printenv command with a few more lines(not shown)**

We can append a variable name as a parameter, to only show that variable value. Fig 2 shows the output for ***env | grep PWD*** and few more



**Fig 2**

Using ***export***, the environment variable will be set for the current shell session. Therefore, if we try to open it another shell or if we restart the VM, the environment variable will not be accessible anymore. As seen in the output environment variable FOO has been set using export with a value ‘test string value’.

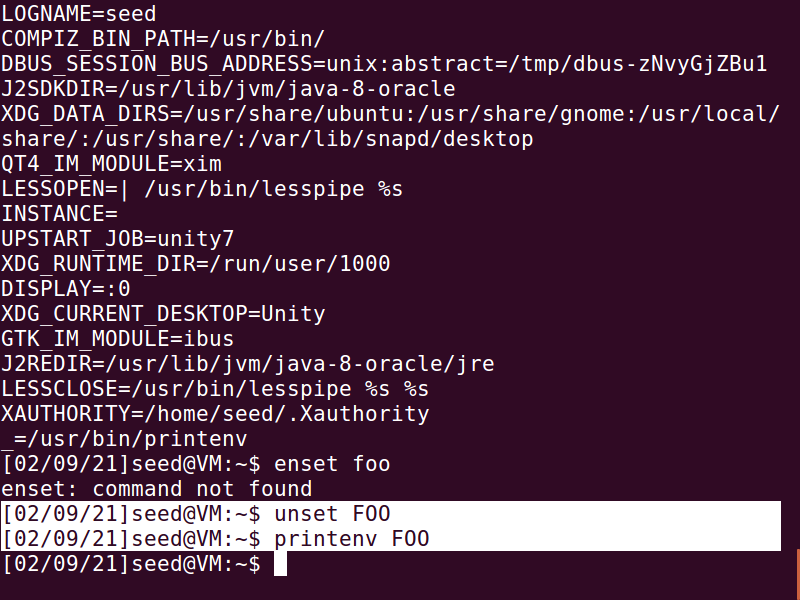


**Fig 3:- Output for command *export***

***unset*** command is used to unset any local environment variable temporarily. Once we unset FOO and then try to find it using env command, it returns nothing because there is no variable FOO.



**Fig 4**

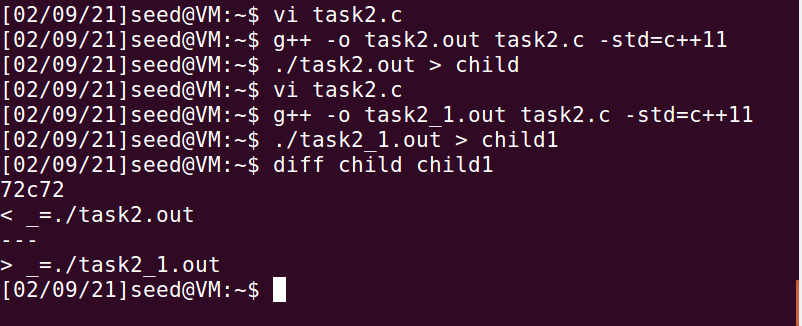


**Fig 5**

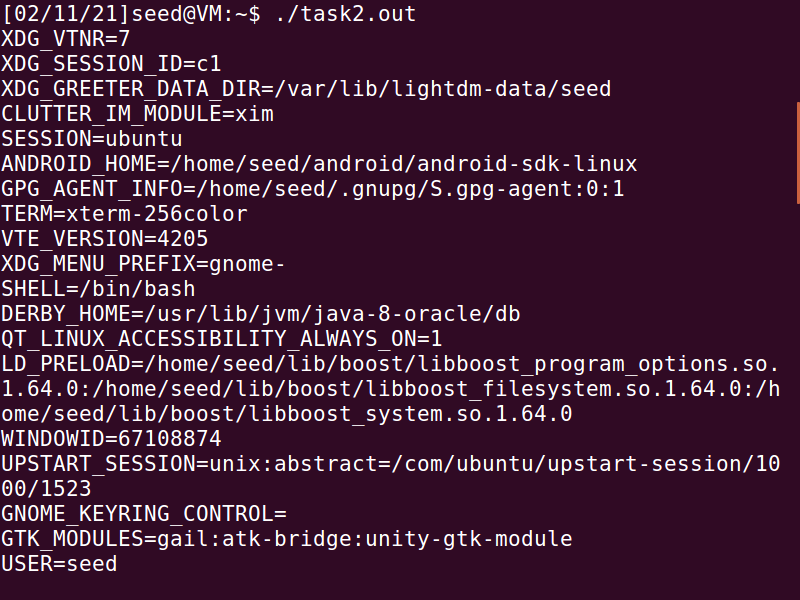
**Task 2:- Passing Environment Variables from Parent Process to Child Process**

**The aim of this task is to study how a child process gets its environment variables from its parent.**

The content of the output of task2.c containing child process with printenv is stored in file named task2.out. It displays all the environment variables of the child process. The parent process with printenv command gives the similar output.



**Fig 6**



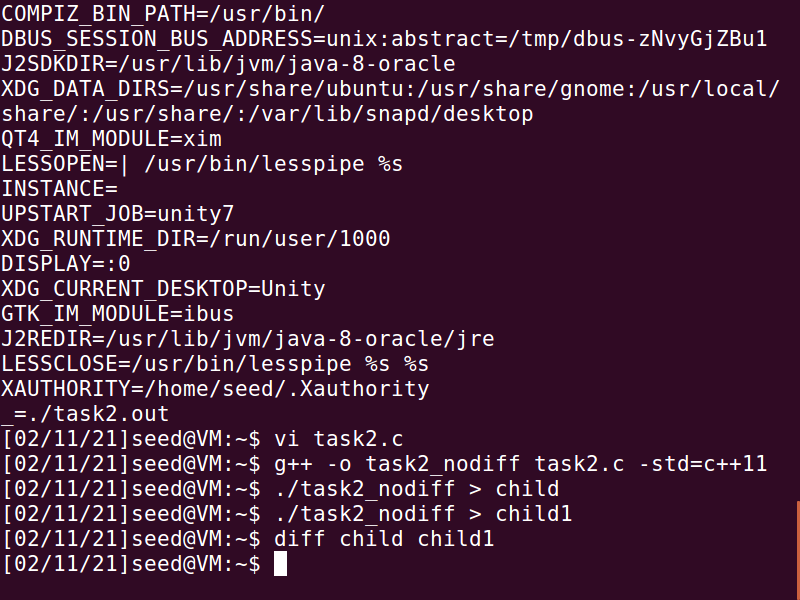
**Fig:- Environment Variables of the child process**

The following are created in this task:

1. Task2.c:- The file containing the code snippet of the task
2. Task2.out:- The executable file with child process printenv()
3. child:- Output from the file Task2.out file
4. Task2\_1.out:- The executable file with parent process printenv()
5. child1:- Output from the Task2\_1.out file

The output for ***diff*** command is interpreted as follows:

72c72 means that in the 72nd line (left) in left file is changed to the 72nd line (right) in the right file, where c stands for changing and the left and right numbers indicate the line number. The < denotes lines in the left file and > indicates in the right file showing the changed content. If both the programs were compiled into a file with the same name, there would not be any difference between the output of the parent and child process.

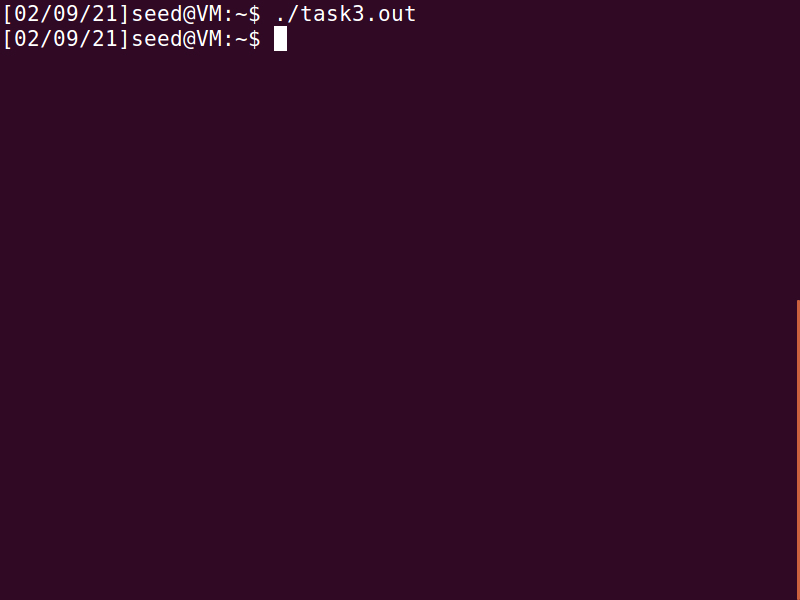


**Fig:- No difference scenario**

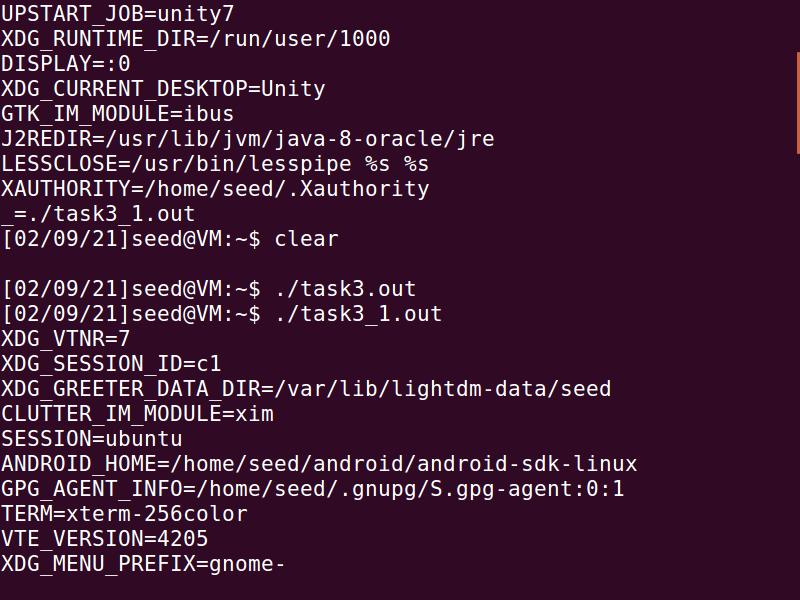
**Task 3:- Environment Variables and execve()**

**The aim of this task is to study how environment variables are affected when a new program is executed via execve()**

Created a program task3.c where the 3rd argument of the execve command i.e., environment variable is set to ***NULL***. The output of this program is nothing as only the shell is returned (as shown in Fig 7) . We then modify the 3rd argument of the execve command and set the environment variable to ***environ***. The output of this program is that the environment variables are printed (as shown in Fig 8).



**Fig: 7 :- program output with the NULL parameter**



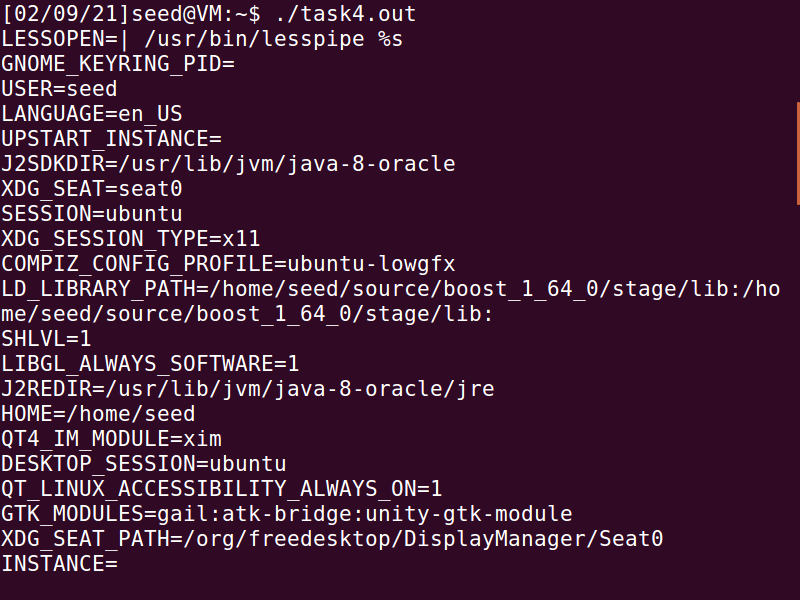
**Fig 8:- program output with environ as 3rd parameter**

The program does not have any environment variable when we pass ***NULL*** to execve() . By passing ***environ*** to execve, all the environment variables of the current process are passed to the program. Thus, we can conclude that the parameters for execve decides the environment variables of the process.

**Task 4:- Environment Variables and system()**

**The aim of this task is to** **study how environment variables are affected when a new program is executed via the system() function.**

The program is compiled and executed and as seen in Fig 9, even though we do not explicitly send any environment variables in the program, the output shows the environment variable of the current process. This happens because the system function implicitly passes the environment variables to the called function /bin/sh.



**Fig 9**

When the system function executes, it does not execute the command directly. It calls the shell instead and the shell executes the command. The shell internally calls the execve command, and the environment variables of the calling process are passed to the shell and the shell passes it to the execve command.

**Task 5: Environment Variable and Set-UID Programs**

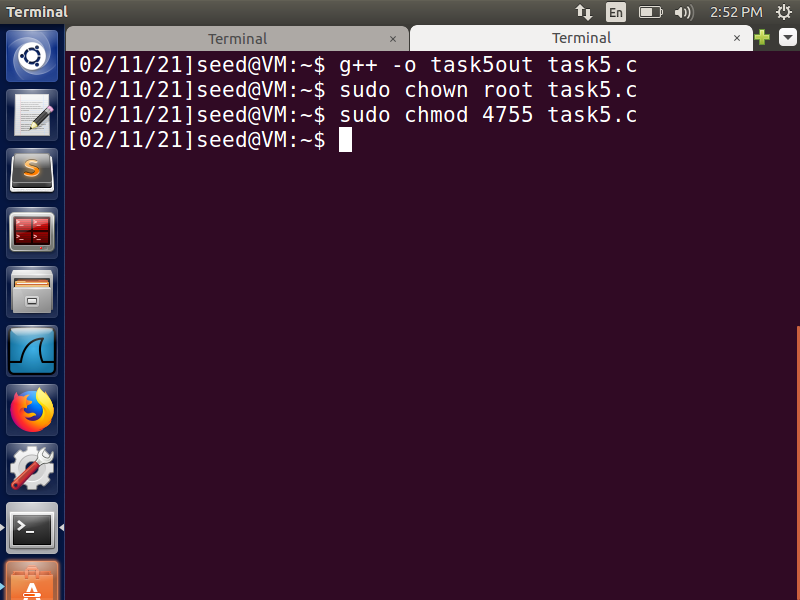
**The aim of this task is to study about Set-UID which is an important security mechanism in Unix operating systems.**

After compiling the given program, we change the ownership and permission of the file using the following commands:

***sudo chown root filename (making the root as the owner of filename)***

***sudo chmod 4755 filename (making the program a SET-UID program by setting set-uid bit)***

This makes the program a SET-UID root program. I initialized a ***new variable with name AB and value ‘seedLabs’ , Set PATH value as /home/seed:$PATH, set LD\_LIBRARY\_PATH value as ‘foo bar’*** using ***export*** command. The following screenshots shows the performed steps:

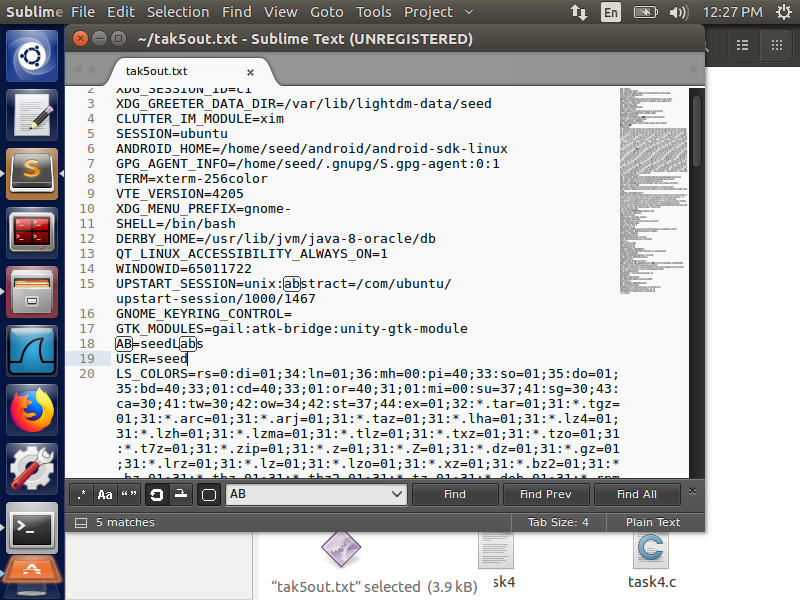


**Fig 10:- Compiling and making it a set-UID program**



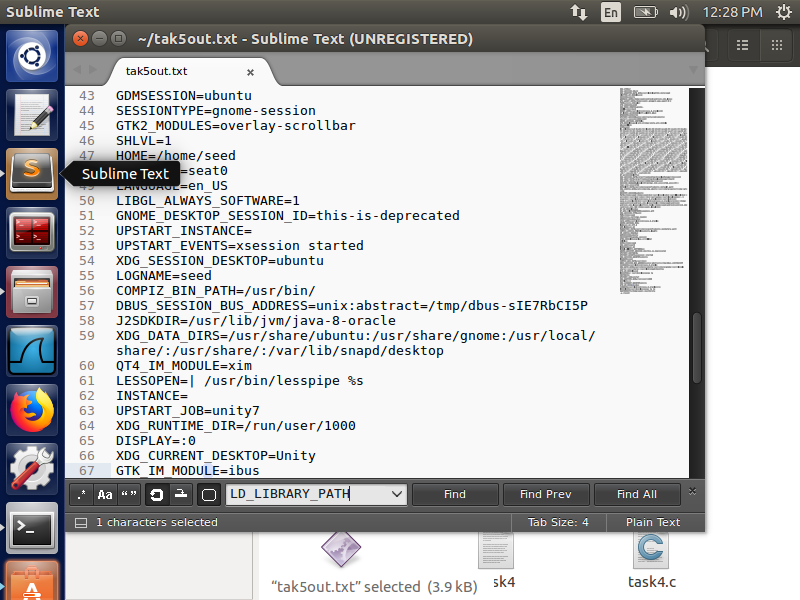
**Fig 11:- Setting values to enironment variables in the user shell**

On running the above compiled program and storing the output in a text file named task5out.txt, it can be noted that the child process inherits the PATH and custom AB environment variable but there is no LD\_LIBRARY\_PATH environment variable.



**Fig 12:- task5out.txt file**



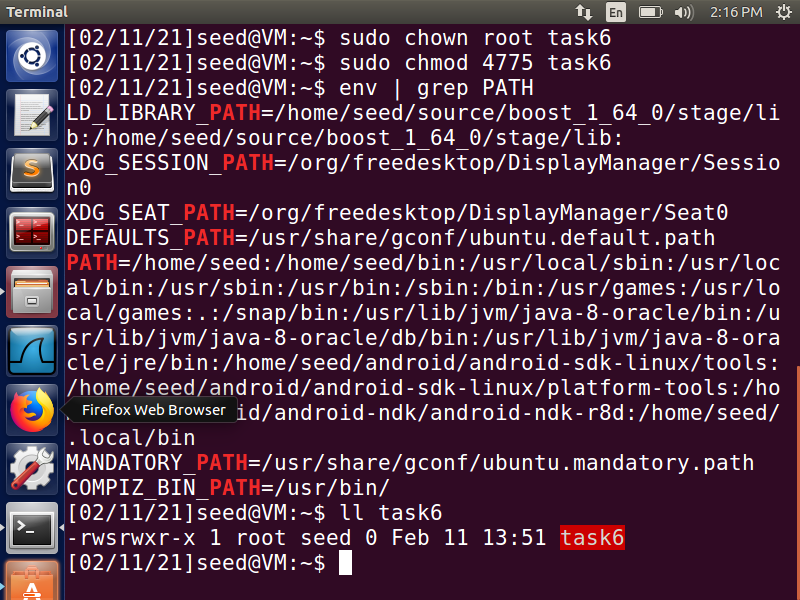


This shows that the SET-UID program’s child process may not inherit all the environment variables of the parent process, LD\_LIBRARY\_PATH being one of them over here. It is a protection mechanism against malicious files being placed into shared libraries. The LD\_LIBRARY\_PATH is ignored here because the RUID and EUID are different. That is why only the other two environment variables are seen in the output.

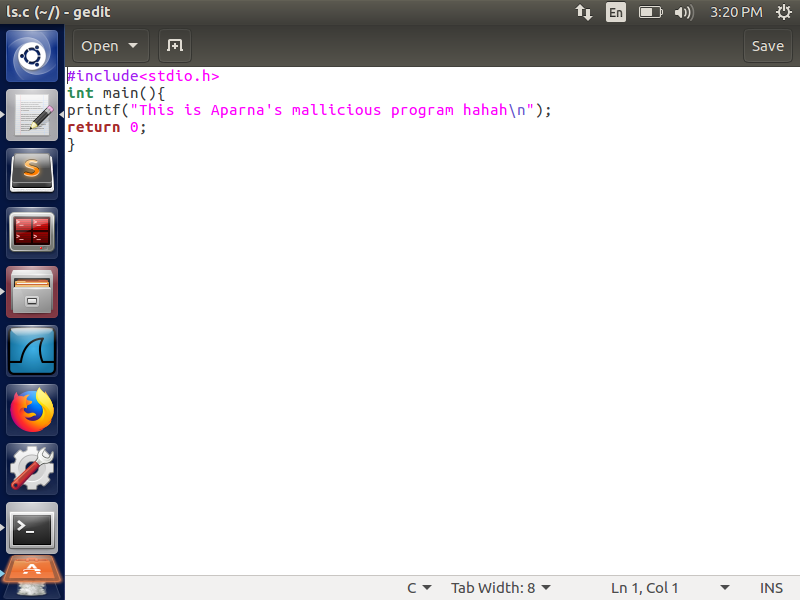
**Task 6:- The PATH Environment Variable and Set-UID Programs**

**The aim of this task is to study the effect of shell program invoking, system() within a Set-UID program which can be quite dangerous.**

The given program is written in the file named task6.c and compiled into task6 file. Then the compiled program’s owner is changed into root and its converted into a SET-UID program. Also, confirming that task6 is a SET-UID program with root as the owner

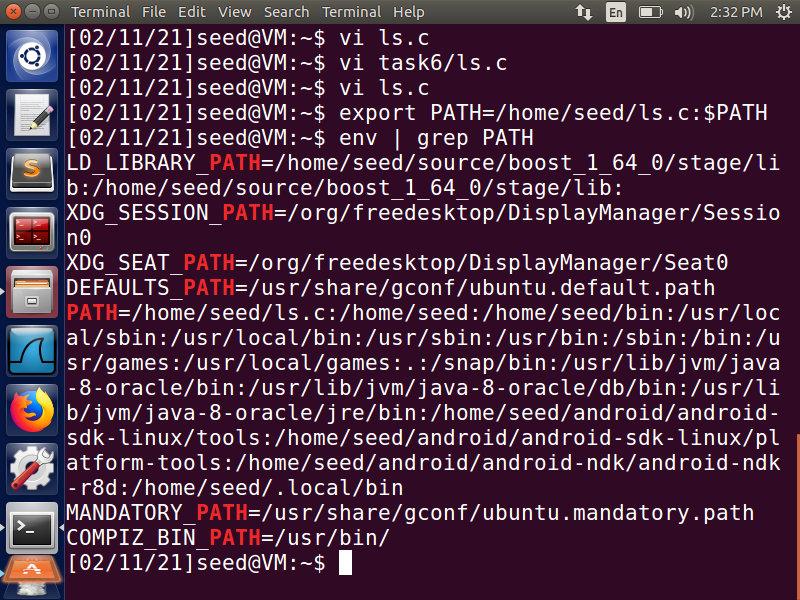


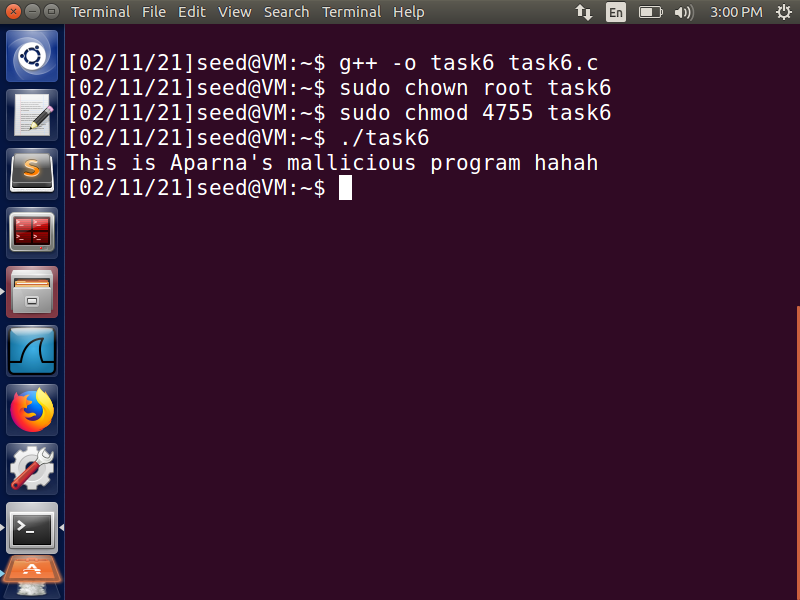
**Fig 13**

I created a new program ls.c and compiled it. This is the malicious file that I will be placing in the path. This file will replace the functionality of the ls command

**Fig 14:- My malicious ls.c program**

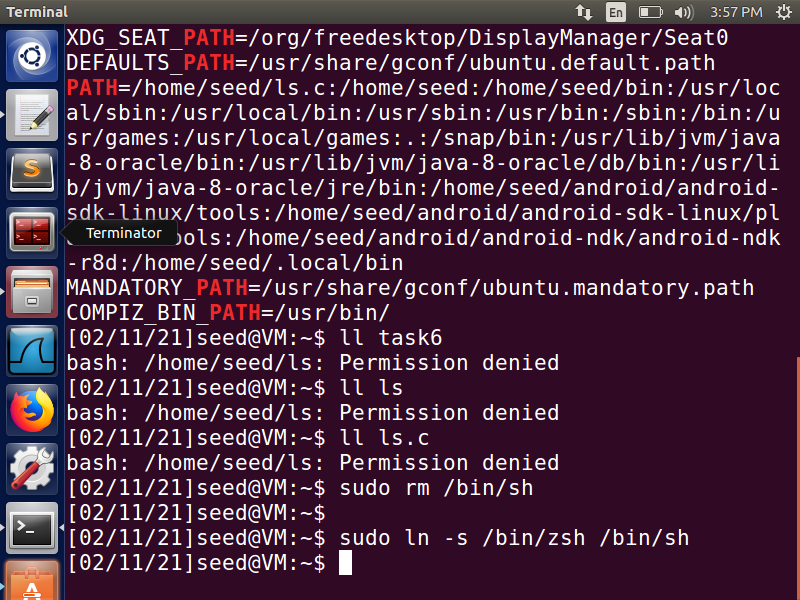
To run my program instead of the standard ls program, I changed the value of environment variable PATH and provided the path to my file as the first value of the variable. This makes the program to search for the file in my directory first before any other directory and since I have the file with the same name as ls, the current program will execute my program.





This illustrates how it is possible to modify the PATH environment variable to point to a desired file and run user-defined program(s) that may be malicious. It is potentially risky because of the inclusion of shell and environment variables because we use system(). Also, we have specified the relative path of the program instead of specifying the absolute path. Due to this, the system() will spawn a shell which will look for a ls program in the location specified by the PATH environment variable. Thus, the attacker can run a malicious code with root privileges by changing the PATH value to a malicious file with the same name as defined in the program since it is a root-owned SET-UID program. Hence using relative path and system function in a SET-UID program could lead to severe attacks.

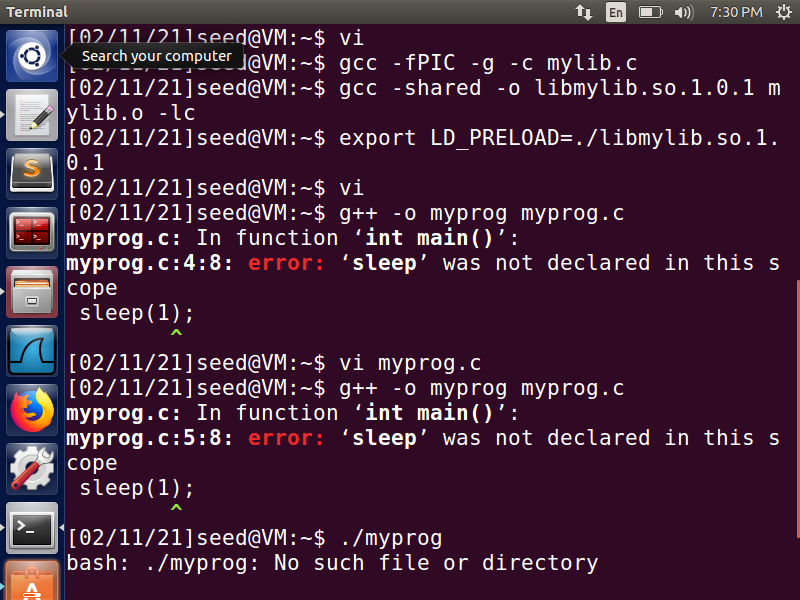
To overcome the dash shell security mechanism of dropping SET-UID program’s privileges on being called from one, link the /bin/sh to another shell



**Task 7: The LD PRELOAD Environment Variable and Set-UID Programs**

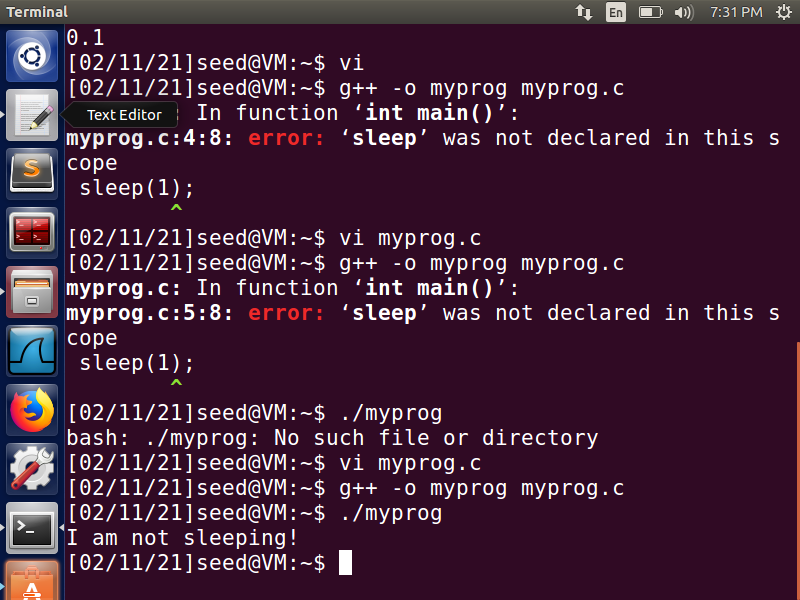
**The aim of this task is to study how Set-UID programs deal with some of the environment variables**.

We create a program ***mylib.c***, compile and make it a dynamic link library. We set the ***LD\_PRELOAD*** environment variable using the export command to point to the dynamic library link (DLL) we just created. We then create a program ***myprog.c*** and compile it. Below Fig 15 shows these operations.



**Fig 15**

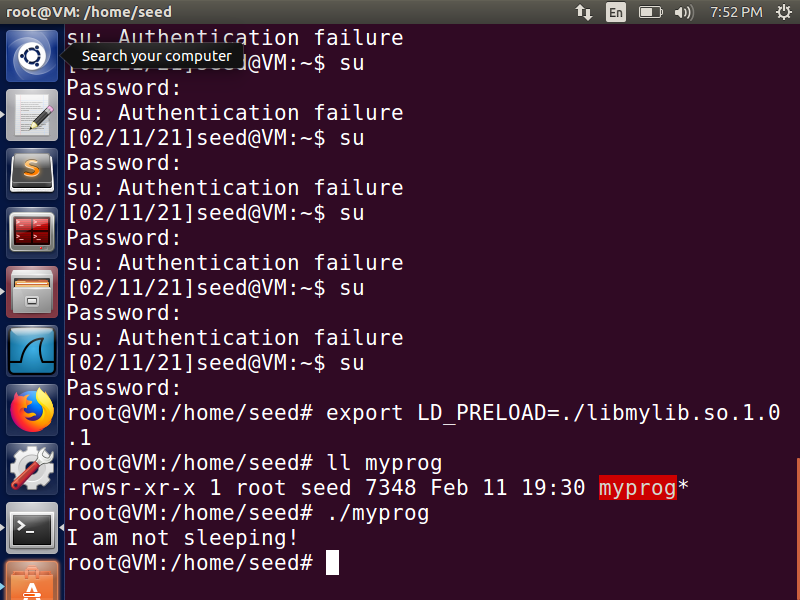
Running it as a normal user program our own library function gets invoked here. It means that this program calls the mylib DLL that we just created instead of the mylib.c DLL.



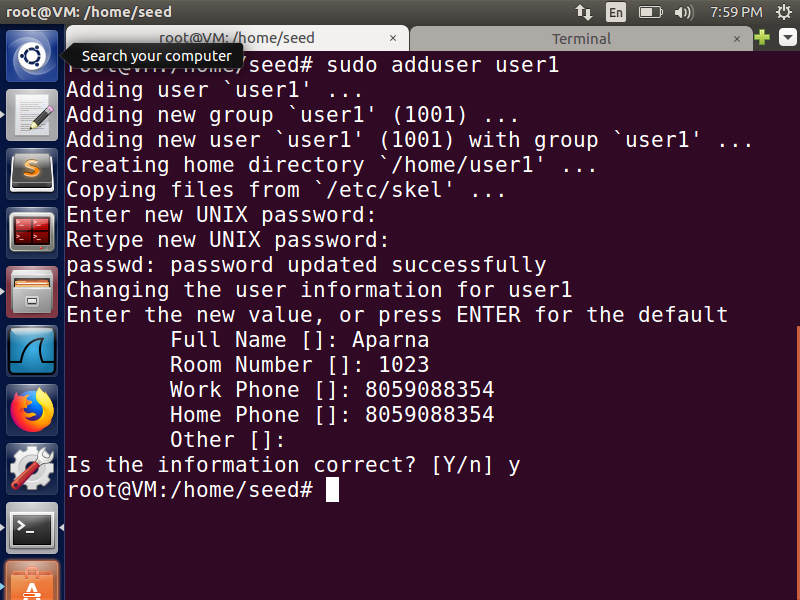
Running it as a set-UID root program as a normal user



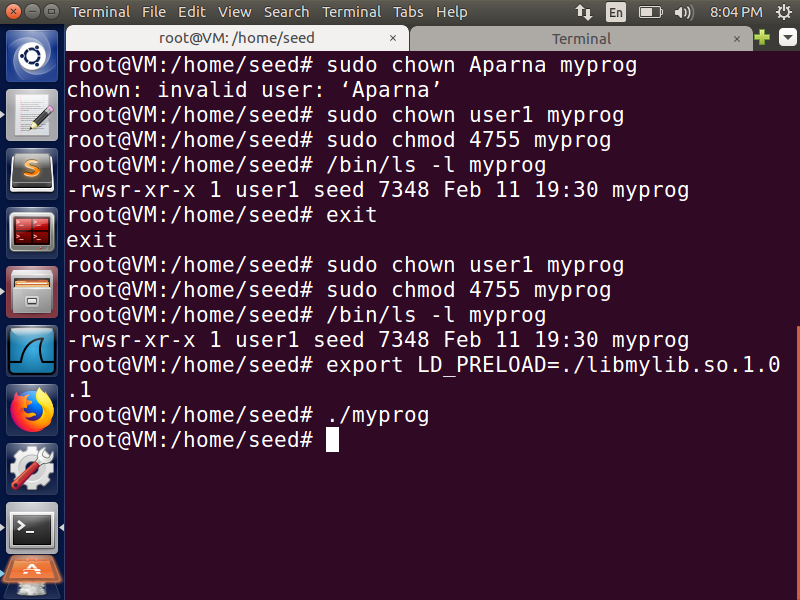
Running it as a set-UID program as a root user



The above screenshot shows that we are executing the myprog.c program from root account. It is a set-UID program owned by the root. We set the LD\_PRELOAD environment variable pointing to the DLL we created. When we execute the program, the program calls the mylib.c DLL that we have created.



**Fig 16: Creating a new User**



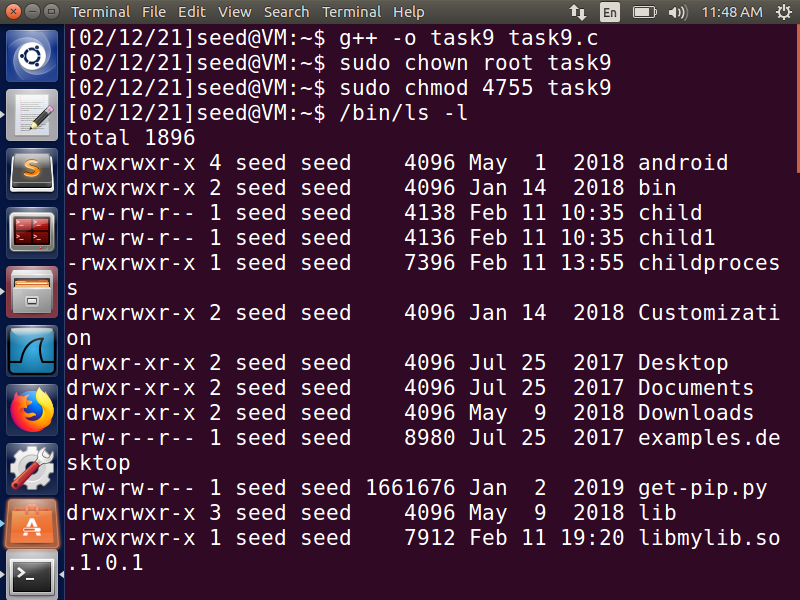
**Fig 17**

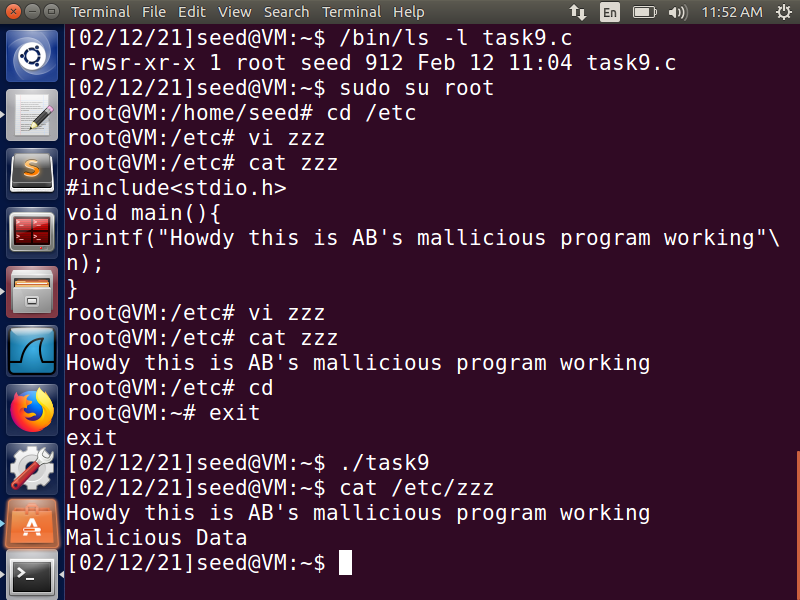
Fig 16 shows how we have created a new user user1. In Fig 17 we can see how to make the myprog.c program a set-UID program owned by user1. We then set the LD\_PRELOAD environment variable pointing to the DLL we created. When we execute the program from another user account like seed, the program sleeps for some time. This means that the program does not invoke the DLL we created.

This behavior indicates that the LD\_PRELOAD variable is present if the effective and real ID are the same and is ignored if they are different. It is basically a protection mechanism of the SET-UID program. In the first and third scenario, since the owner and the account executing the file were the same, the LD\_PRELOAD variable was present and user-defined library was preloaded. Whereas, in the second case, the effective ID was of root and real ID was of seed, the LD\_PRELOAD variable was ignored, and system-defined sleep function was called instead. In the fourth scenario, myprog is a Set-UID program owned by a user and run by another user. Hence LD\_PRELOAD is ignored again because it is a Set-UID program.

**Task 9: Capability Leaking**

**The aim of this task is to study the principle of Least Privilege, Set-UID programs that often permanently relinquish their root privileges if such privileges are not needed anymore.**



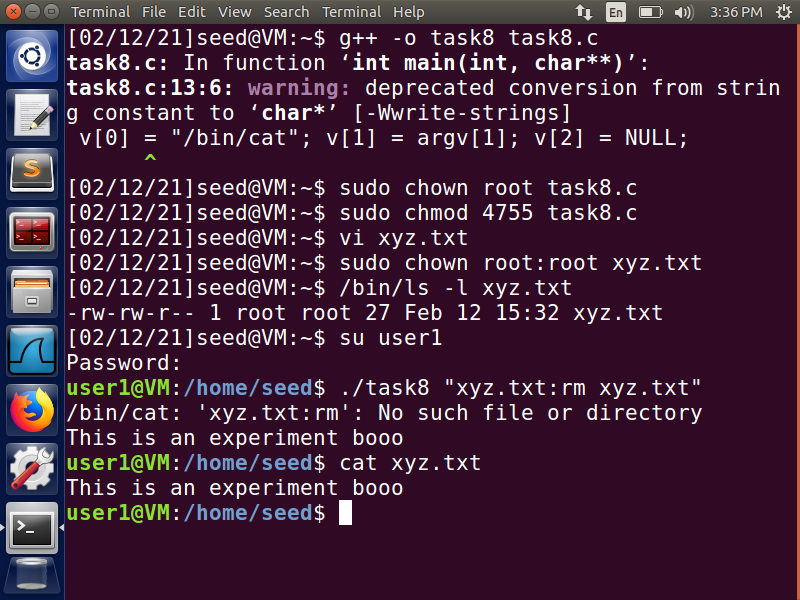


Created a program task9.c, compiled it and made it a Set-UID program owned by root. I then Logged in as root and created a program ***zzz*** in the ***/etc*** directory. Exited from the root account and executed the program from seed account. We can notice that the file ***/etc/zzz*** is modified by appending the content of the child process into the file. This is capability leaking. The above screen shots show the mentioned operations.

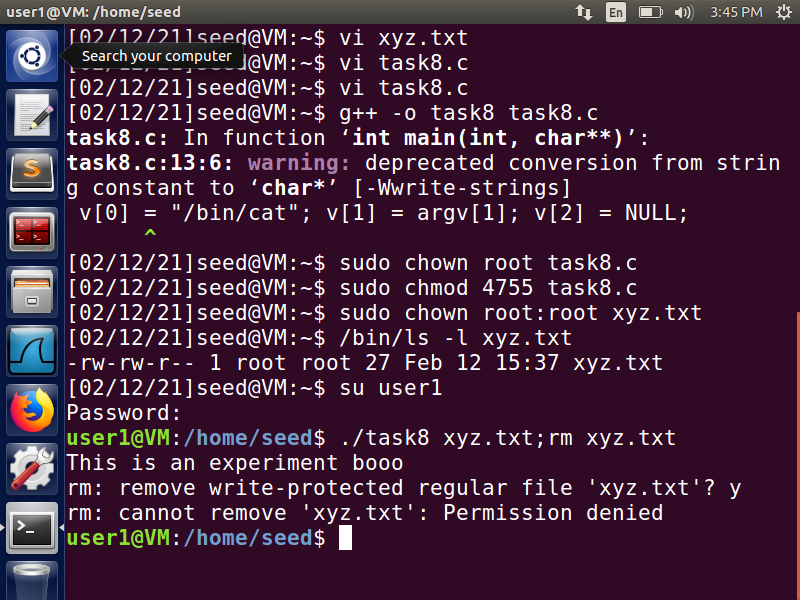
This happens because even though in the program, we dropped the privileges, we did not close the file at the right time and hence the file was still running with privileged permissions that allowed the data in the file to be modified, even without the right permissions. The child inherits copies of the parent's set of open file descriptors during fork call. Hence the malicious user can successfully modify the content of a privileged file. This shows that it is important to close the file descriptor after dropping privileges, for it to have the appropriate permissions. To avoid such attacks, the file descriptor must be closed before the fork call.

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

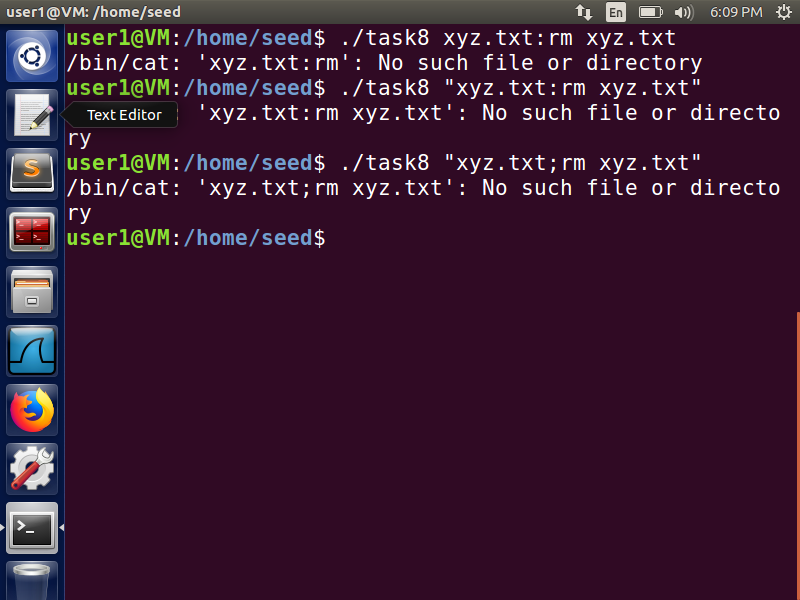
Here, first I compile the program provided into a file named task8.c. Next, this file is converted into a root-owned SET-UID program with executable permission to other users. I then created a xyz.txt file that is owned by root with no write privileges to other users. Now I login into user1 account and execute the program. The program displays the contents of the xyz.txt file and deletes the file on which it did not have the write permission by using the rm command after the : . This shows that even though user1 did not have any permission to write, it could remove a file easily by assuming the privileges of the root user.

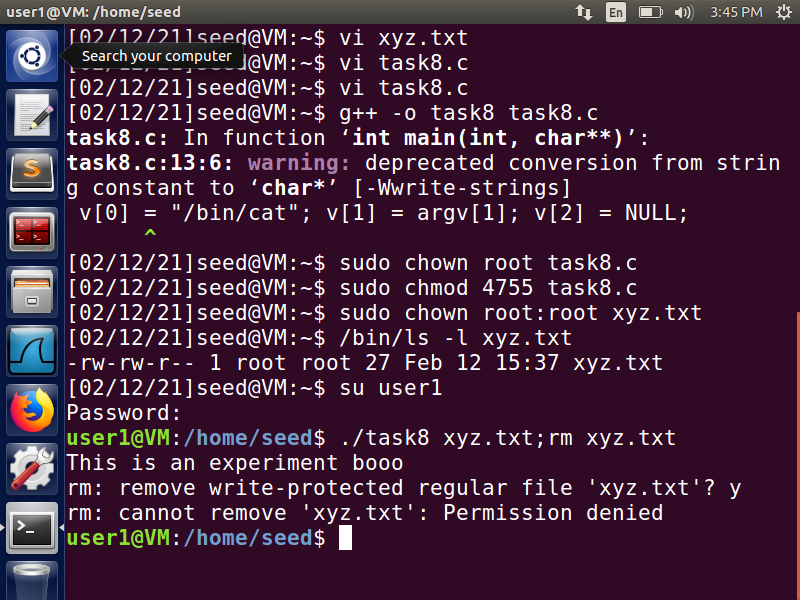


**Fig 18:- with system()**



**Fig 19:- With execve()**





I modified the code to have the execve() command and comment the system() command. When I execute the program without “”, the program executes only till the ; and displays the contents of the file. The remaining part of the argument consisting of the rm command is not executed since it does not have permissions. When I try to execute the program with “”, the program searches for the exact match . So, a file by that name would not exist.

When system() executes, it does not execute the command directly. It calls the shell instead and executes the command. So, if the program is a Set-UID program, the user will have temporary root privileges and can remove any file they wish to with root privileges. Multiple commands can be passed together using the “ ”. There is no input validation while using system (), but there is some when we use execve(). When we use execve() command it replaces the program with the called program and passes the argument strings exactly as specified and does not interpret quotes. Thus, when we pass the something after the ’ ; ‘ it is treated as a new command and root privileges would have been lost. This avoids this kind of attack.