Yogesh Chaudhari
SUID: 244195971

Task 1: Manipulating environment variables

In this task we use export and unset commands to modify, add and delete environment variables for a shell. And we use **printenv** or **env** command to print all environment variables or **printenv var_name** or **env | grep var_name** to print a specific environment variable.

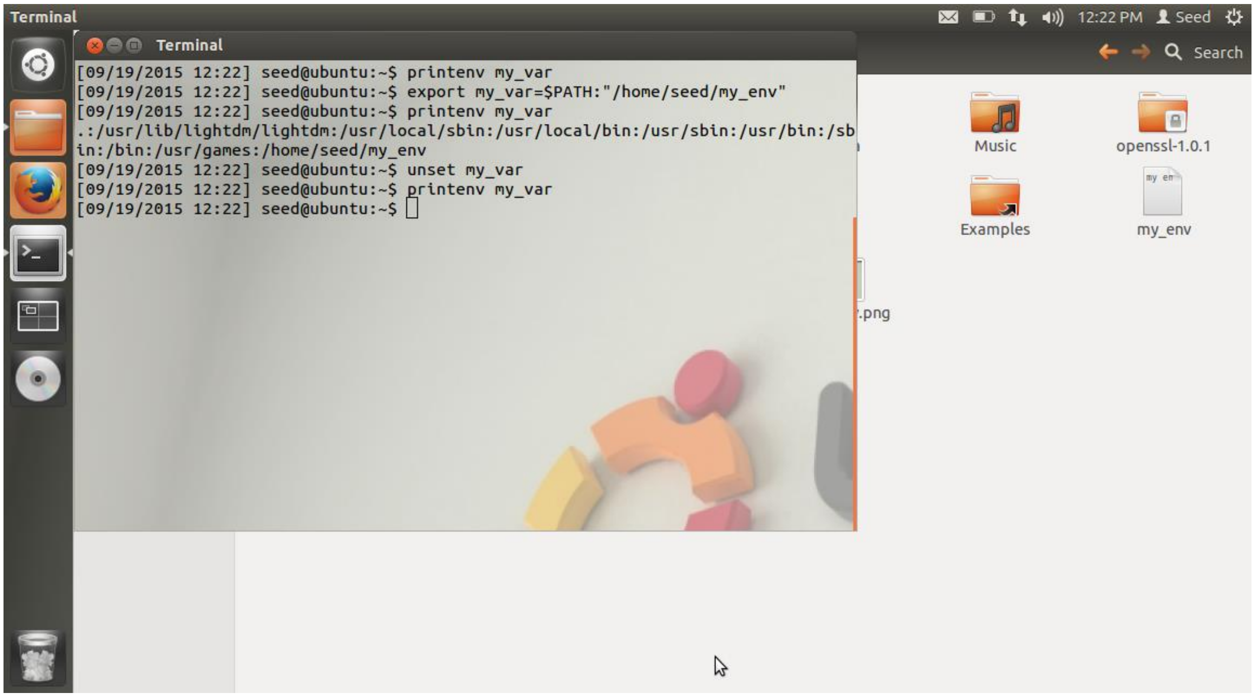
Commands run in default shell of seed user and their respective outputs are shown below in screenshots:

Printenv and env commands:

In above screenshot, we can see the default value stored for **PWD** environment variable using **printenv** and **env** commands in default shell of seed user.

Yogesh Chaudhari
SUID: 244195971

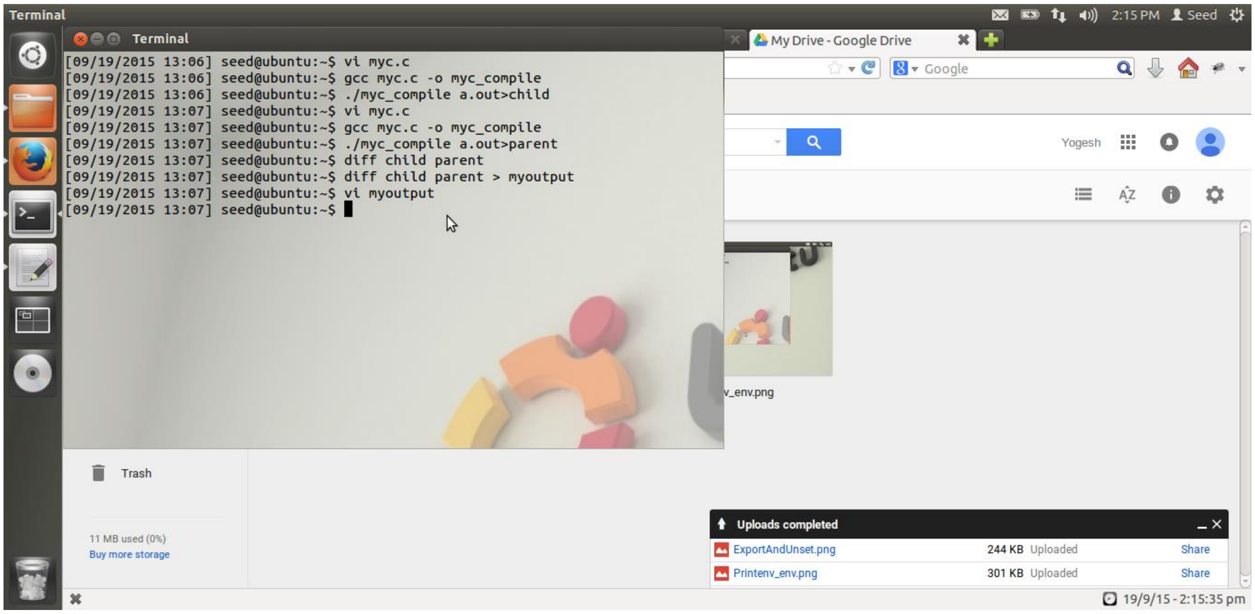
export and unset commands with printenv:



We have set a **my_var** environment variable using **export** command and given it a reference to user-defined file. By using **printenv** we can see that **my_var** environment variable has been added to the shell with given path as reference value. By using **unset** command we have removed **my_var** environment variable and by using **printenv** command we can verify that, as we get no value as output for **pirintenv** command.

Task2: Inheriting environment variables from parents

Screenshots for task 2 are as follows,



<u>For child process:</u>

In this we write a code given as follows in myc.c ,

```
#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

extern char **environ;

void printenv()

{

int i = 0;

while (environ[i] != NULL) {

printf("%s\n", environ[i]);

i++;

}

}
```

```
void main()

{

pid_t childPid;

switch(childPid = fork()) {

case 0: /* child process */

printenv();

exit(0);

default: /* parent process */

//printenv();

exit(0);

}

}
```

We compile this program and then run it, such that its output should get return in child file.

In this program we are creating a child process by using **fork**() and printing environment variables for this process into child file using **printenv**() function in switch case with 0 value. Because of switch on **fork** method, when a child process is running this method will give 0 as output value to switch and for parent process it will run the default case.

<u>For parent process:</u>

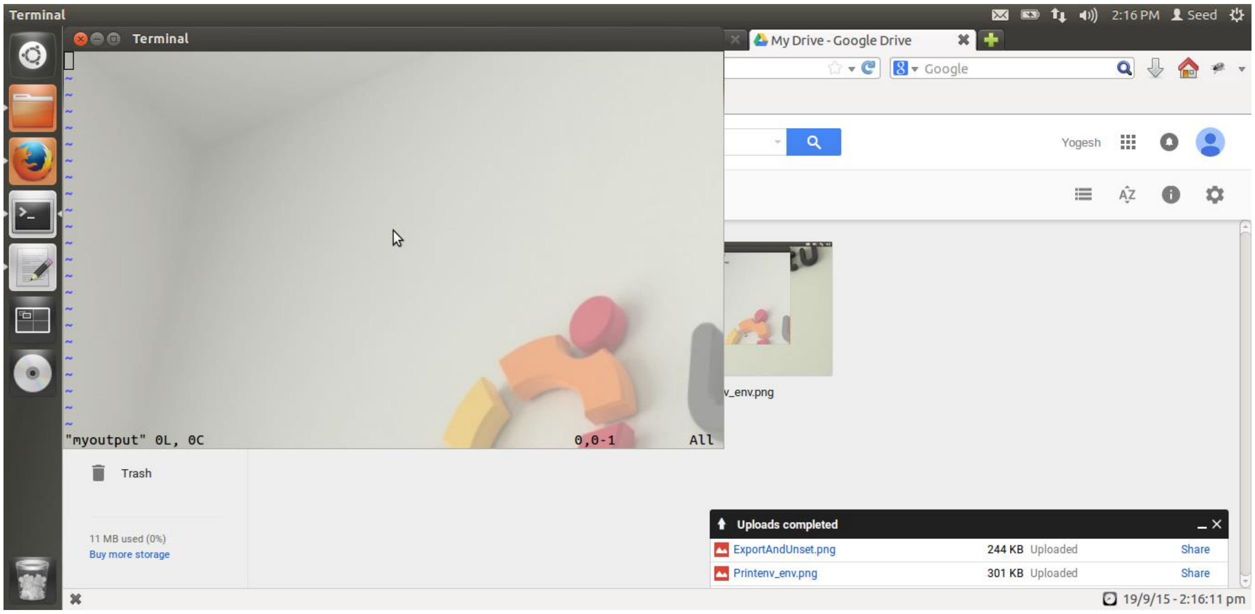
By modifying the above program we are printing environment variables for parent process. We saved the output to parent file. For printing environment variables for parent process the following code is used,

```
#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

extern char **environ;

void printenv()

{

int i = 0;

while (environ[i] != NULL) {
```

Yogesh Chaudhari
SUID: 244195971

```
printf("%s\n", environ[i]);
i++;
}
}
void main()
{
pid_t childPid;
switch(childPid = fork()) {
case 0: /* child process */
//printenv();
exit(0);
default: /* parent process */
printenv();
exit(0);
}
}
```

Now, we used diff command on these two files - child and parent to compare environment variables for both child and parent processes. Output of diff command is written into myoutput file.

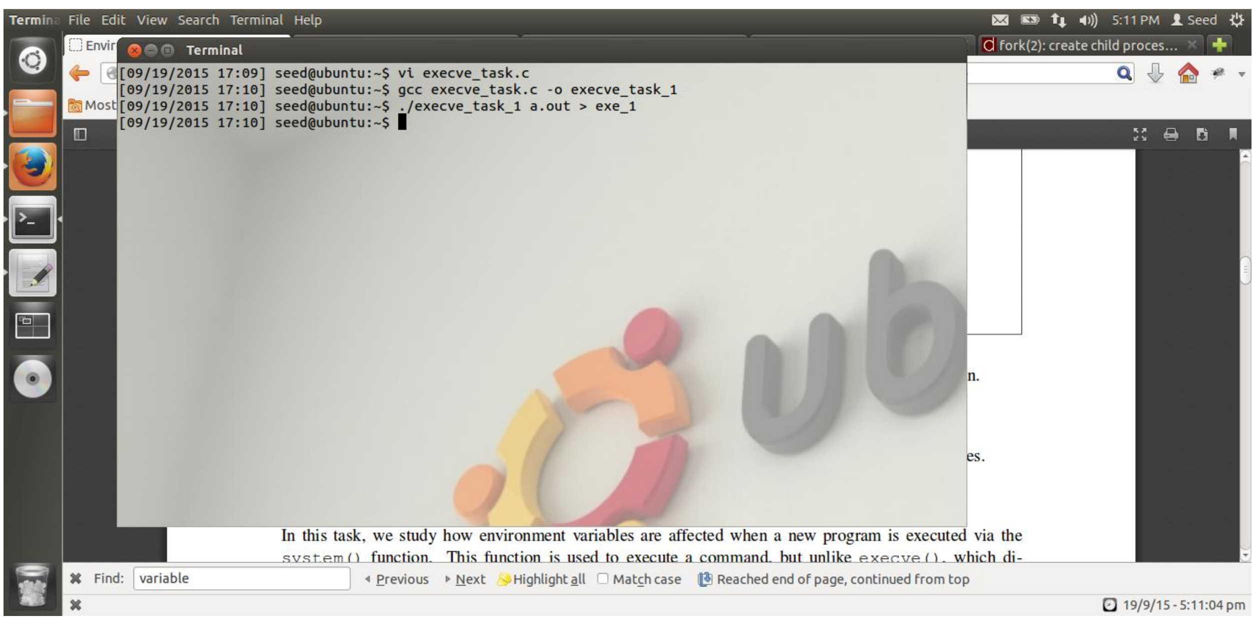


As we can see myoutput file is empty, we observed that for parent and child processes environment variable value remain same. These values are not dependent on processes, but the environment (i.e. shell environment) in which processes have been initiated.

Task 3: Environment variables and execve()

step 1: calling execve() with NULL value as 3rd argument in following code,

```
#include <stdio.h>

#include <stdlib.h>

extern char **environ;

int main()

{

char *argv[2];

argv[0] = "/usr/bin/env";

argv[1] = NULL;

execve("/usr/bin/env", argv, NULL);

return 0 ;

}
```

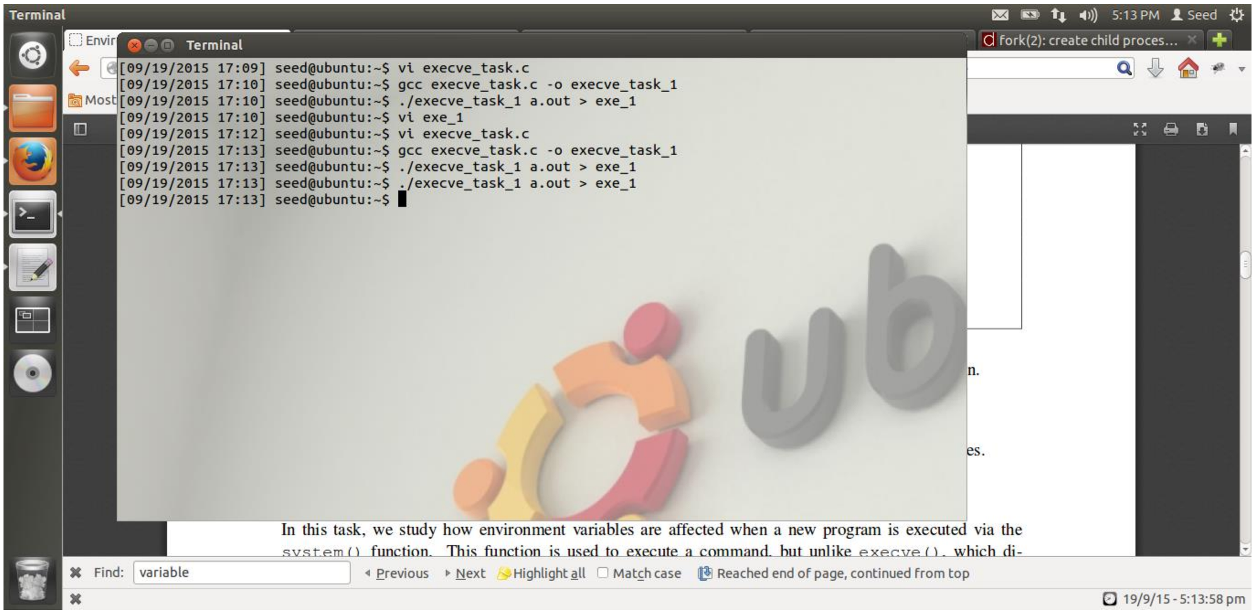


Output file: exe_1

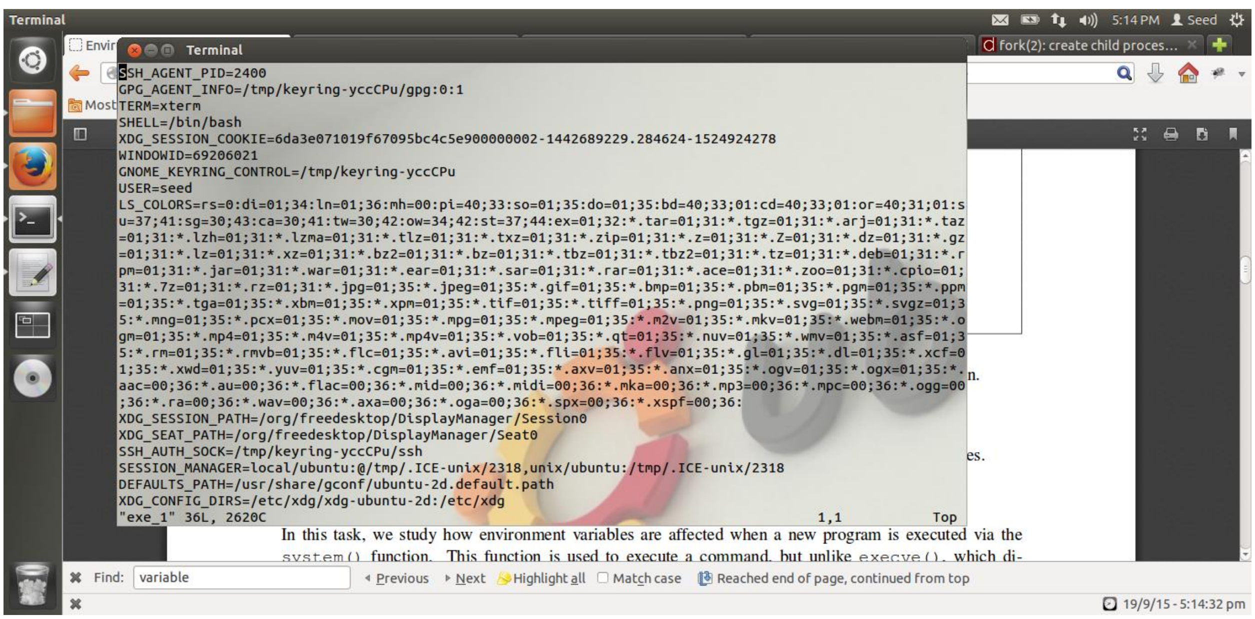
From the above screenshots we can see that when we call **execve**() with **NULL** value as 3rd argument, the process which gets created in parent process by overriding all data and stacks of parent process doesnot get any environment variables by default.

Step2: calling execve() with environ as 3rd argument value in following code,

```
#include <stdio.h>

#include <stdlib.h>

extern char **environ;

int main()

{

char *argv[2];

argv[0] = "/usr/bin/env";

argv[1] = NULL;

execve("/usr/bin/env", argv, environ);

return 0 ;

}
```



<u>The output file exe_1:</u>

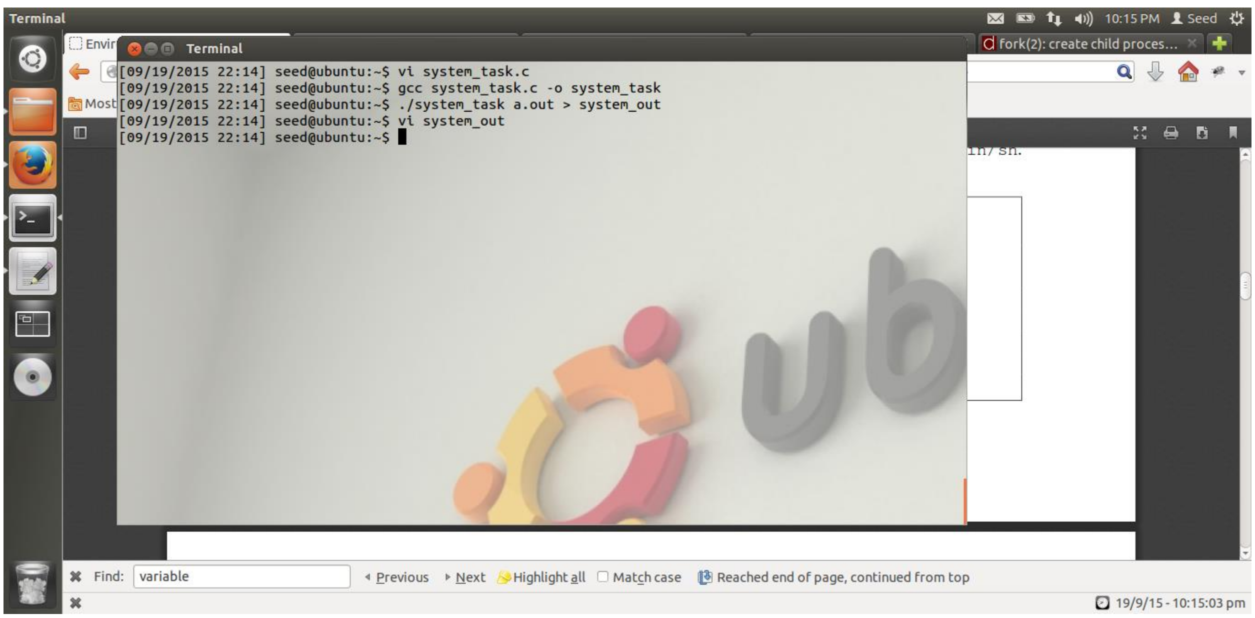


From above screenshot we can see that we make a call to **execve** with **envp** pointer array as argument, the new process gets environment variables by default. This is the situation because, the arguments specified by a program with the ***execve*** function are passed on to the new process image in the corresponding ***main***() arguments. The **envp** argument (3rd argument) constitute the environment for the new process image when we pass it as argument for **execve**() call.

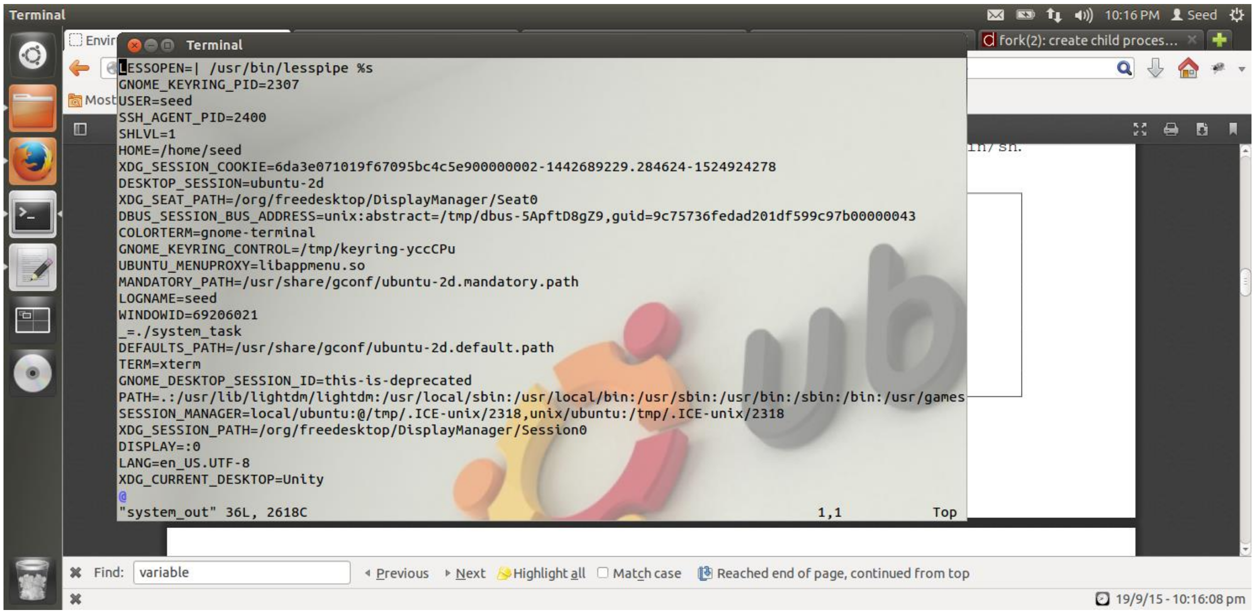
Task4: creating new process using system() method

Output after compileing following code is given as,

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

system("/usr/bin/env");

return 0 ;

}
```



Yogesh Chaudhari
SUID: 244195971



As from the above output file system_output you can see that, when we create a new process using **system**() call, the new process gets environment variables from its parent process. When system() call is made system invoke a new method call to excel(), which then calls execve() with environment variables. Thus, we have verified that in **system**() call, environment variables array is passed as an argument to **execve**() call. Hence, we have verified the implementation of **system**().

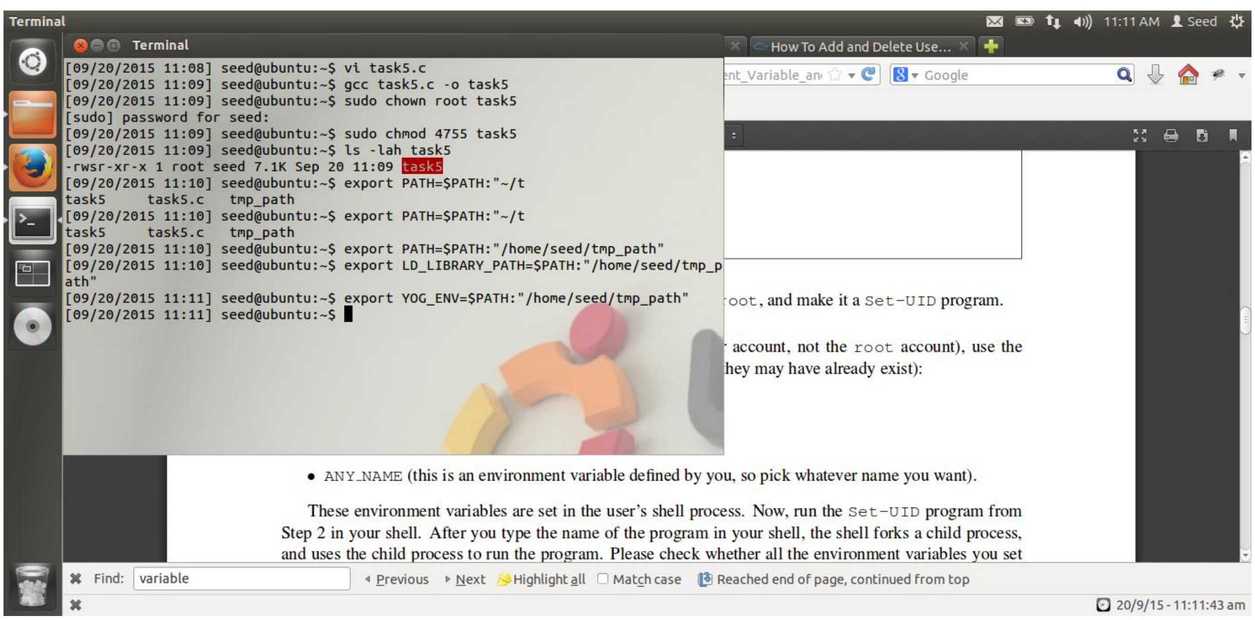
For **execve**() call we have to pass environment variables array as argument, new process doesnot get them by default and for system() call new process gets environment variables by default.

Task5: Environment Variables and Set-UID program

We are running following code as task5.c program,

```
#include <stdio.h>

#include <stdlib.h>

extern char **environ;

void main()

{

int i = 0;

while (environ[i] != NULL) {

printf("%s\n", environ[i]);

i++;

}

}
```

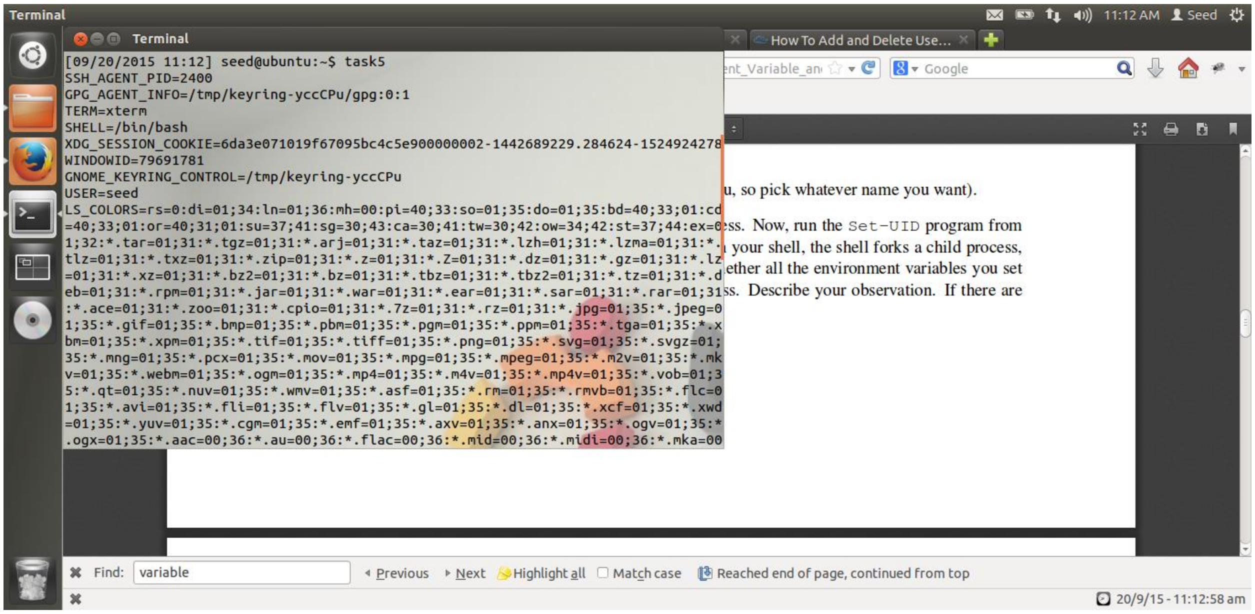
The above program prints all the environment variables from environ array.

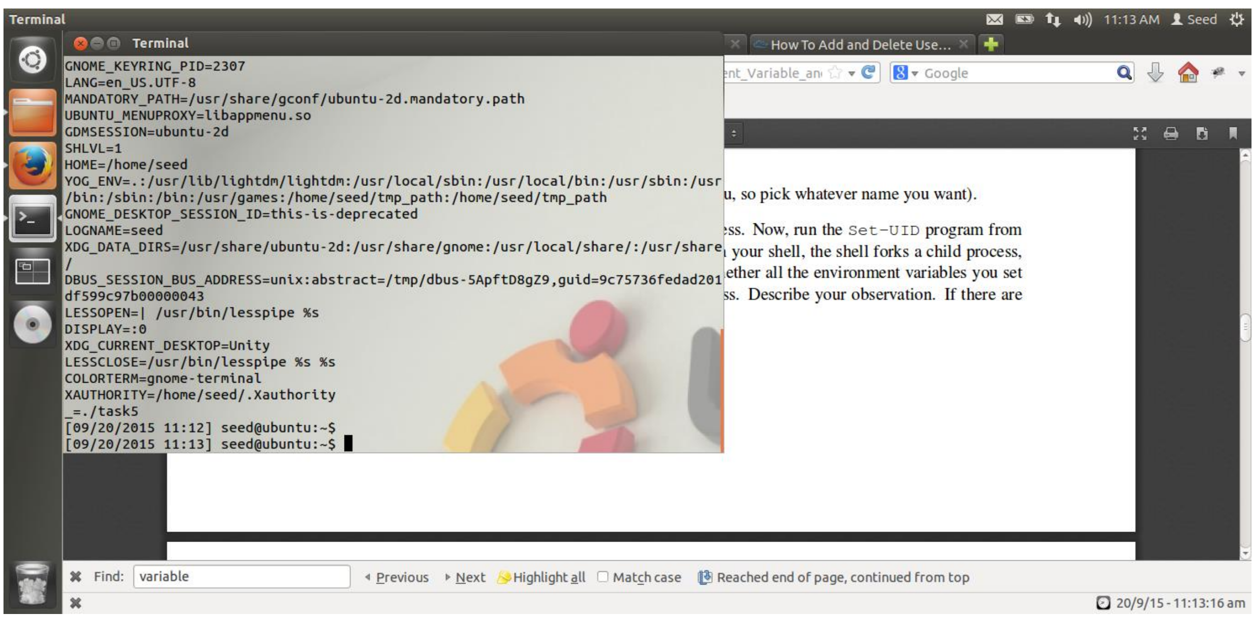


By using **chown root** command we are making root as owner of our program. And using **chmod 4755** command we are making it a **Set_UID** program. The bit **4** from **4755** is **set_UID** bit. Its binary value **100** sets the **set_UID** bit value to **1**. From this screenshot we can see that task5 file has been set as set_UID program and it has root as owner. From access privilege level given by '-**rwsr**' we can deduce that it is an set_UID program, as 's' bit stand for set_UID.

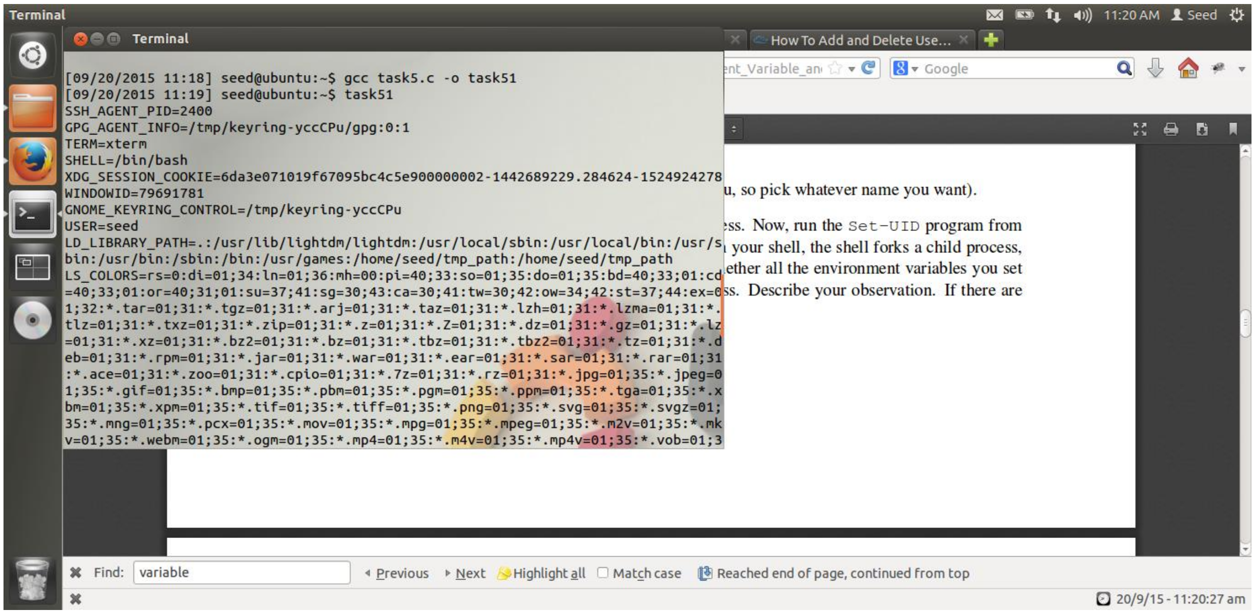
By using export command we are setting three environment variable **PATH**, **LD_LIBRARY_PATH** and **yog_env** with respective reference path values.

Here, we are running task5,

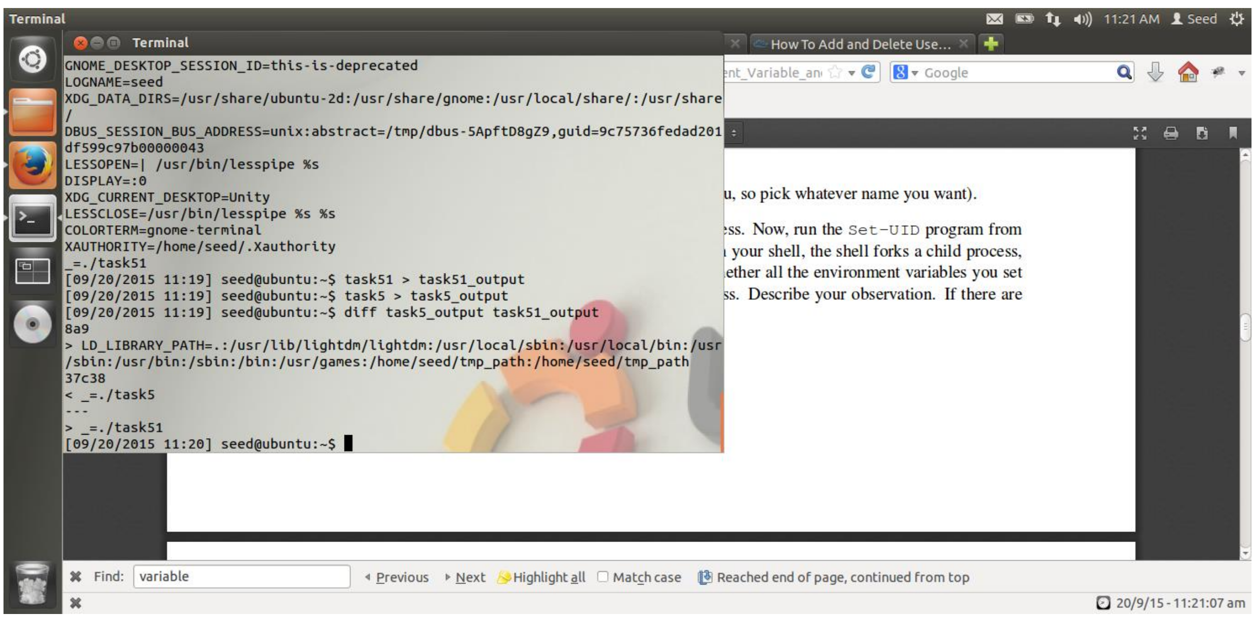




From above two screenshots we can see that, the new child process which got created has inherited all of the parent's environment variables.

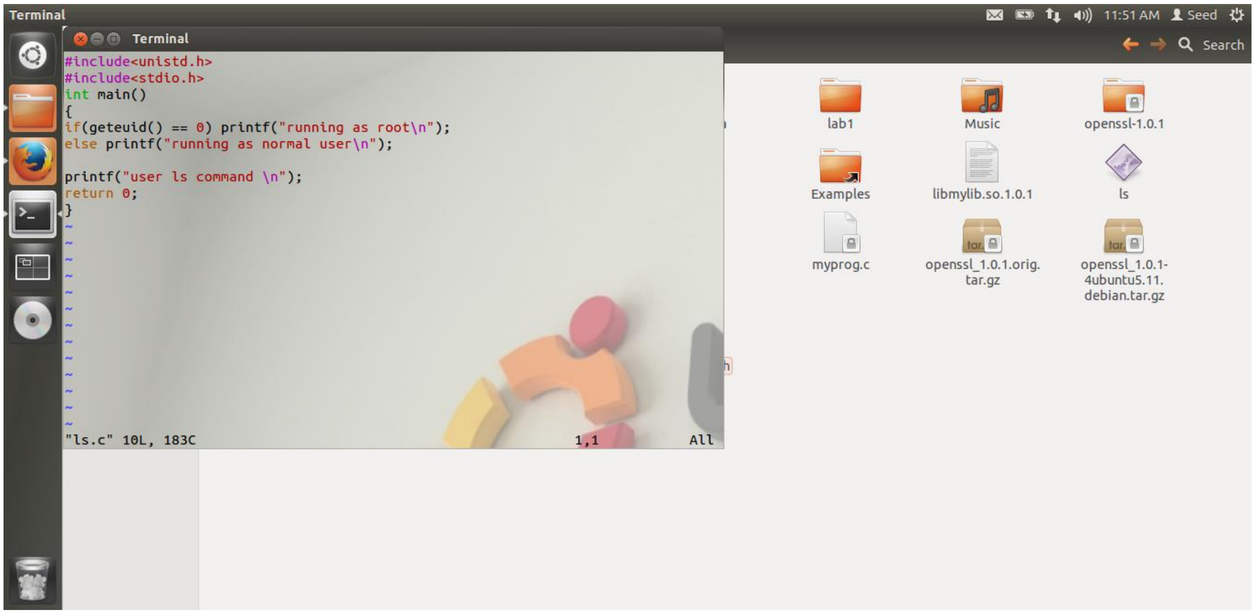


Here we are running task51 as non set-UID user program.

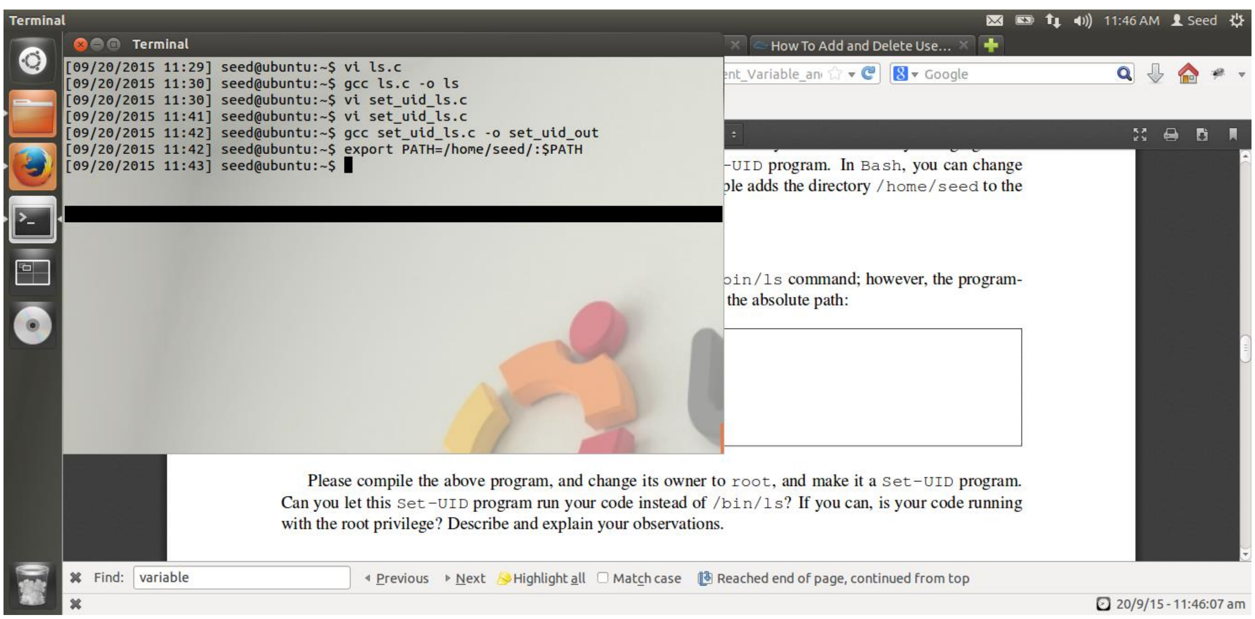


We are writing output of both set-UID root program and non Set-UID user program to task5_output and task51_output respectively. After that we used **diff** command on **env_output** and **set_uid_output** files. We can see that **LD_LIBRARY_PATH** environment variable was not inherited by child process when run as Set-UID program. The reason for this environment variable not to be inherited is that, it is an environment variable that specifies user local libraries to be added to the shell.

Task6: The PATH Environment variable and Set-UID Programs



In above screenshot we can see user defined **ls** command code.



Here by using export command we have changed the PATH variable value to refer to user defined directory where our user defined **ls** code resides.

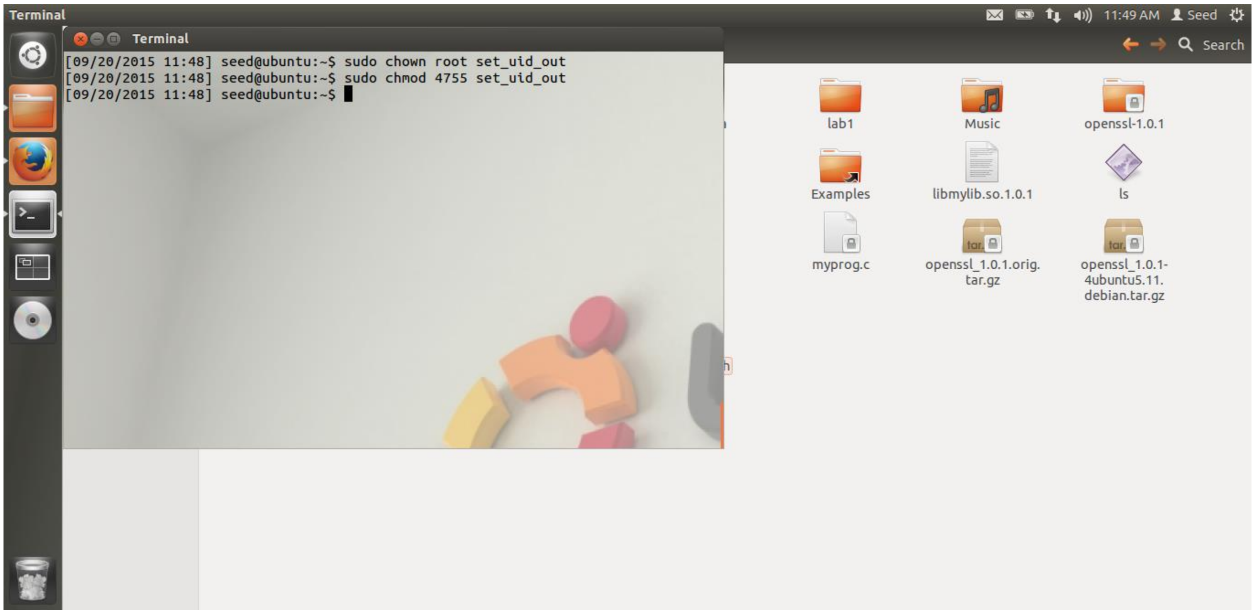
In above screenshot we can see that we have compiled following code in **set_uid_ls.c** file and stored the compiled file with name **set_uid_out**, int main()

{

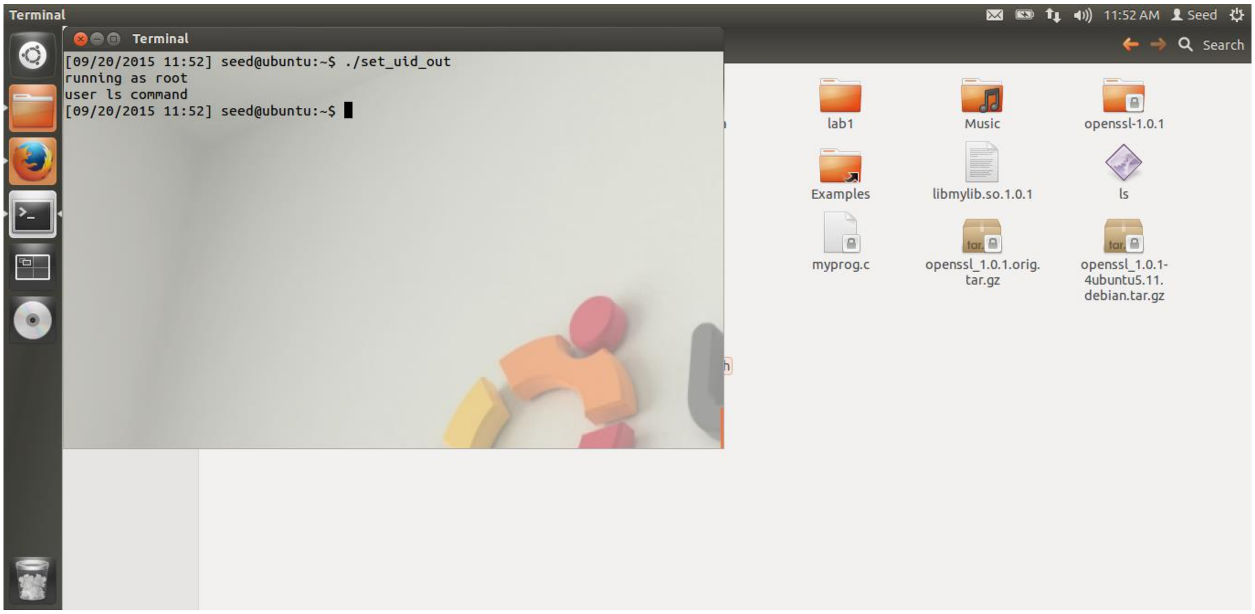
system("ls");

return 0;

}

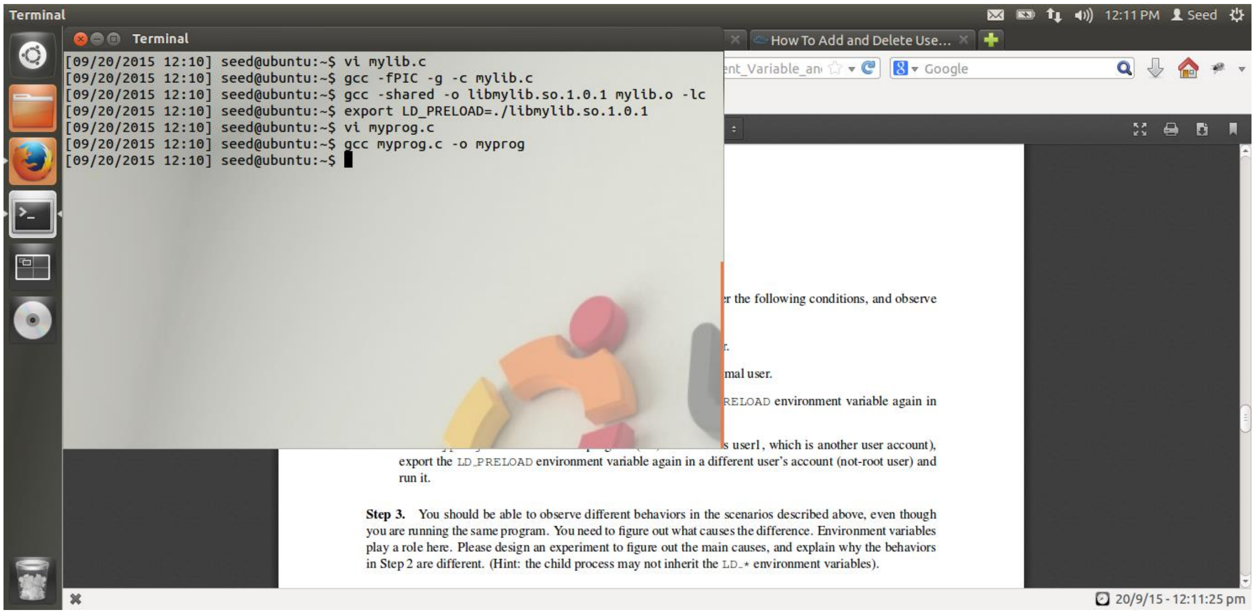


We have changed the owner of set_uid_out to root using **chown root** command and made it a set-UID program using **chmod 4755** command.



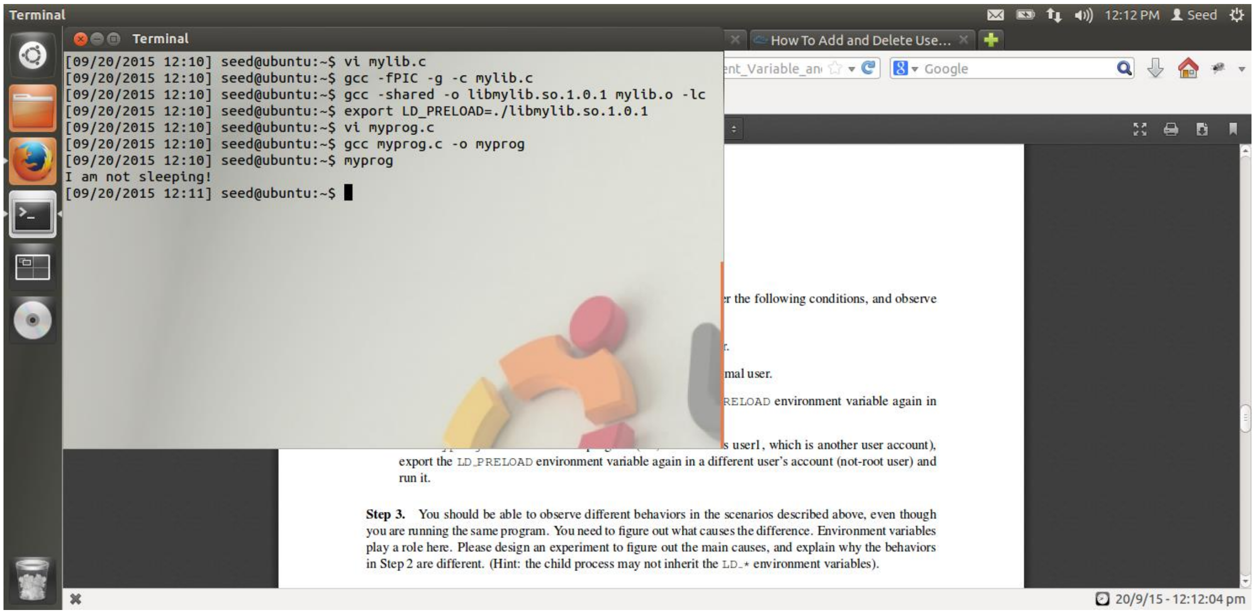
From above screenshot we can see that when system tried to access **ls** environment variable it found path to user defined directory because of value we modified in **PATH** environment variable and ran user defined **ls** command rather than **ls** command residing in **/bin/ls**. This user defined code is run with root privileges. We have used **geteuid()** method to check for programs effective user ID.

TASK7: The LD PRELOAD environment variable and Set-UID Programs

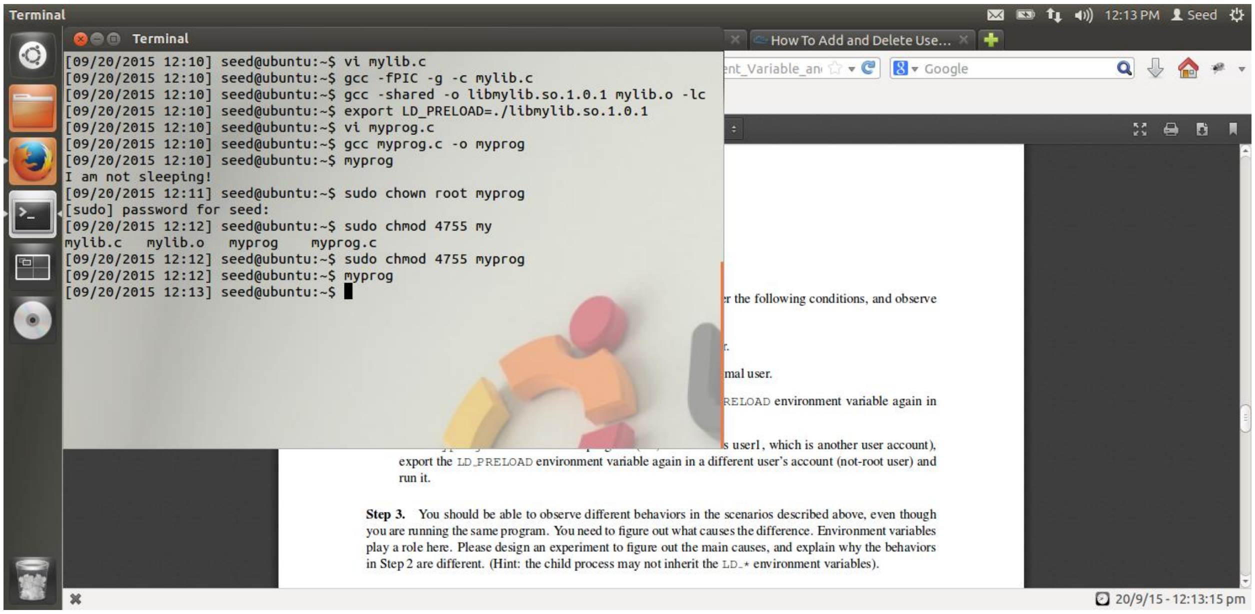


In the above screenshot we have compiled **mylib.c** dynamic library using **–fPIC** and **–shared** option in gcc compiler. –**fPIC** option makes our code suitable for inclusion in library. We set **LD_PREOAD** variable with our shared library **libmylib.so.1.0.1**. **LD_PRELOAD** variable defines libraries which will be loaded before loading any other libraries. Then, we compiled **myprog.c** file with following code,

```
/* myprog.c */

int main()

{

sleep(1);

return 0;

}
```

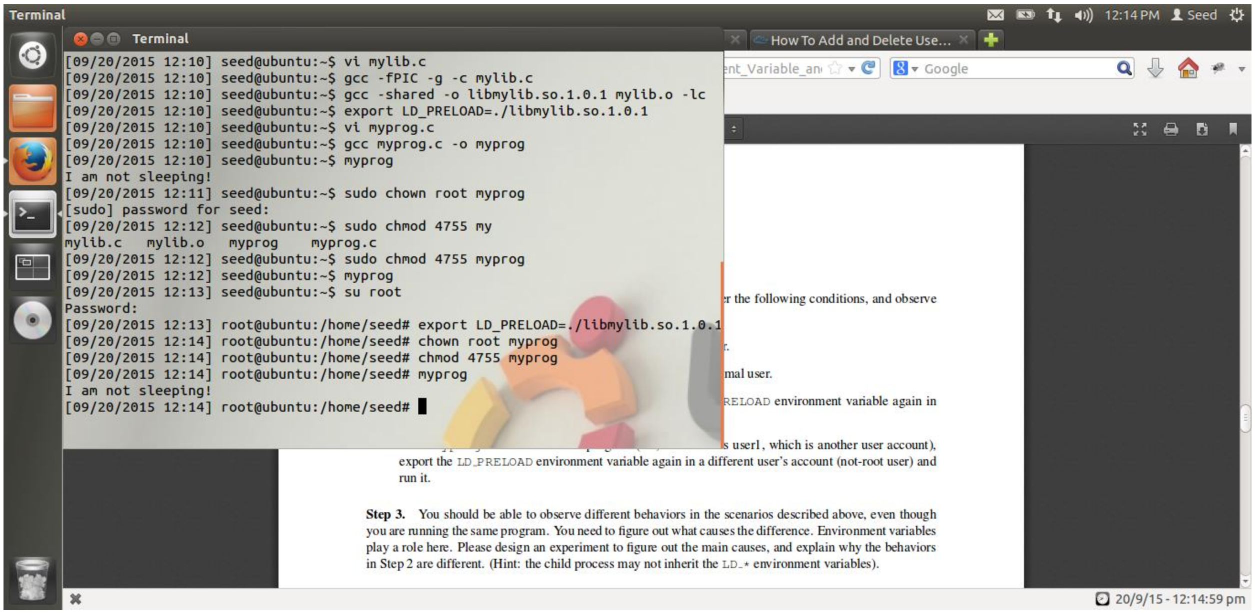


<u>Running **myprog** as normal program with normal user privileges:</u> Here, as we have exported **LD_PRELOAD** environment variable with our predefined library program, so when **myprog** is run, new child process is created in user shell environment. In which system tries to make **sleep()** call and our user-defined **sleep()** gets called. This is because, the child process inherits all the environment variables from parent process.



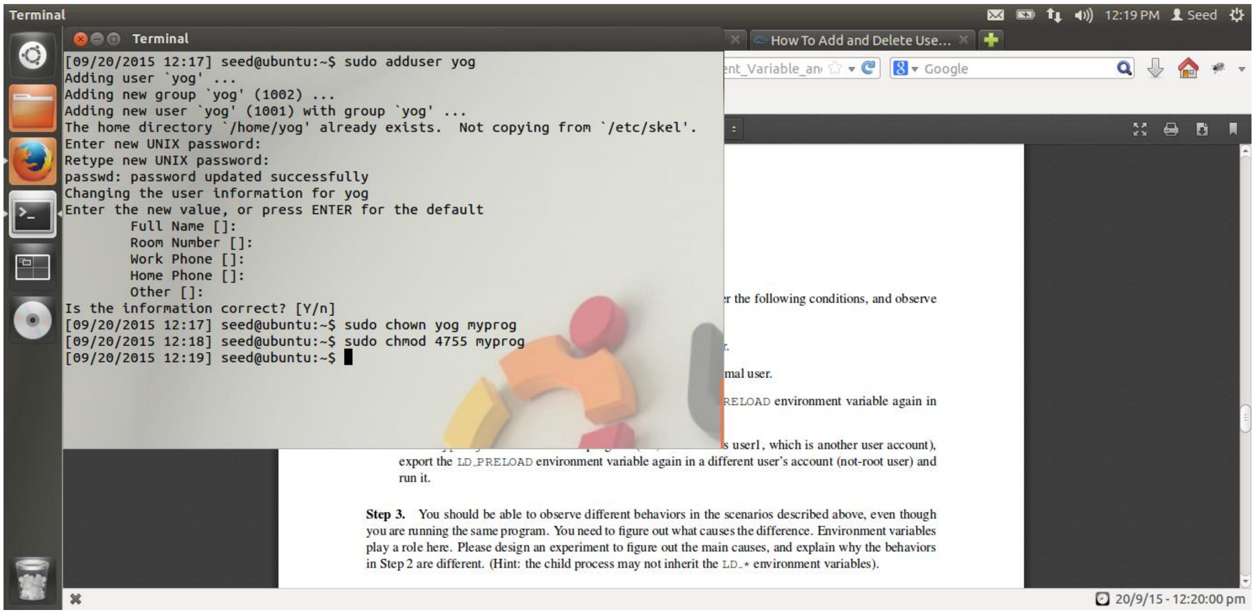
In above screenshot, we have changed ownership of **myprog** to **root**. We changed **myprog** to **set_UID** root program using **chown root** and **chmod 4755** commands respectively.

<u>Running **myprog** as **set-UID** root program from user shell:</u> Here, we have exported **LD_PRELOAD** environment variable in user shell with our predefined library program. When **myprog** is executed, new child process is created in root shell. In which system tries to make **sleep()** call. But, here user-defined **sleep()** doesnot get called as **LD_PRELOAD** environment variable doesnot exist in root shell in which **myprog** is running. Child process doesnot inherit any **LD environment variables (LD_*)** from parent process when its invoked in different shell environment.

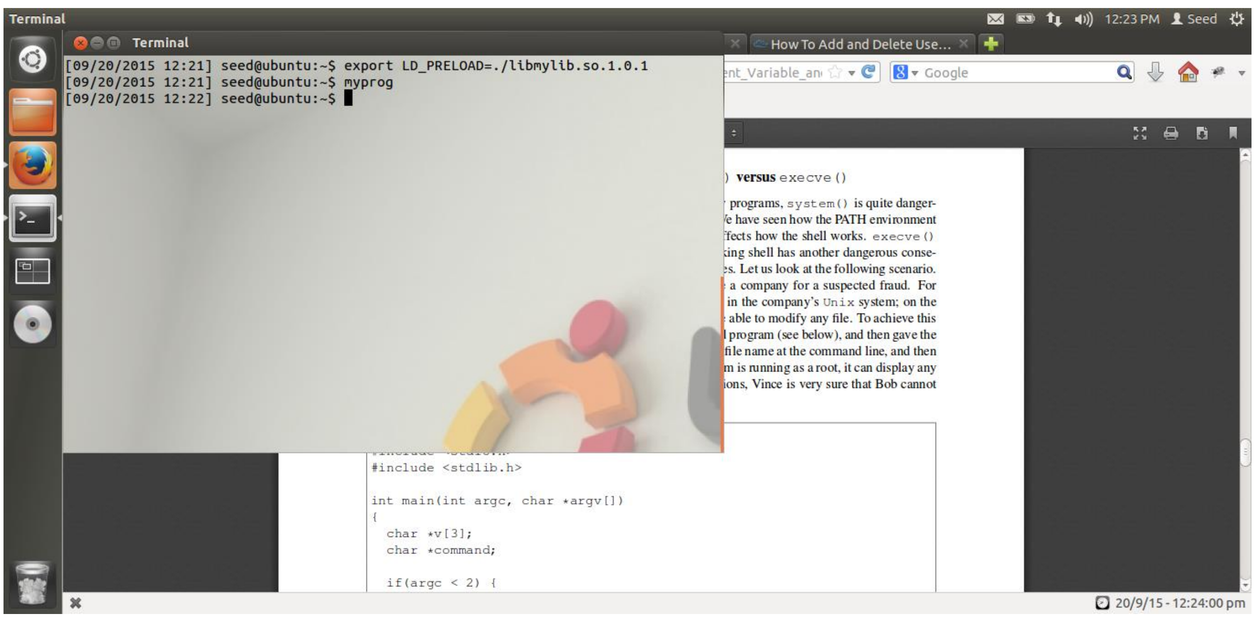


In above screenshot, we have changed ownership of **myprog** to **root**. We changed **myprog** to **set_UID** root program.

<u>Running **myprog** as **set-UID root** program from root shell</u>: Here, we have exported **LD_PRELOAD** environment variable in root shell with our predefined library program. When **myprog** is executed, new child process is created in same shell. In which system tries to make **sleep()** call. Here user-defined **sleep()** gets called as **LD_PRELOAD** environment variable exists in root shell in which **myprog** is running.

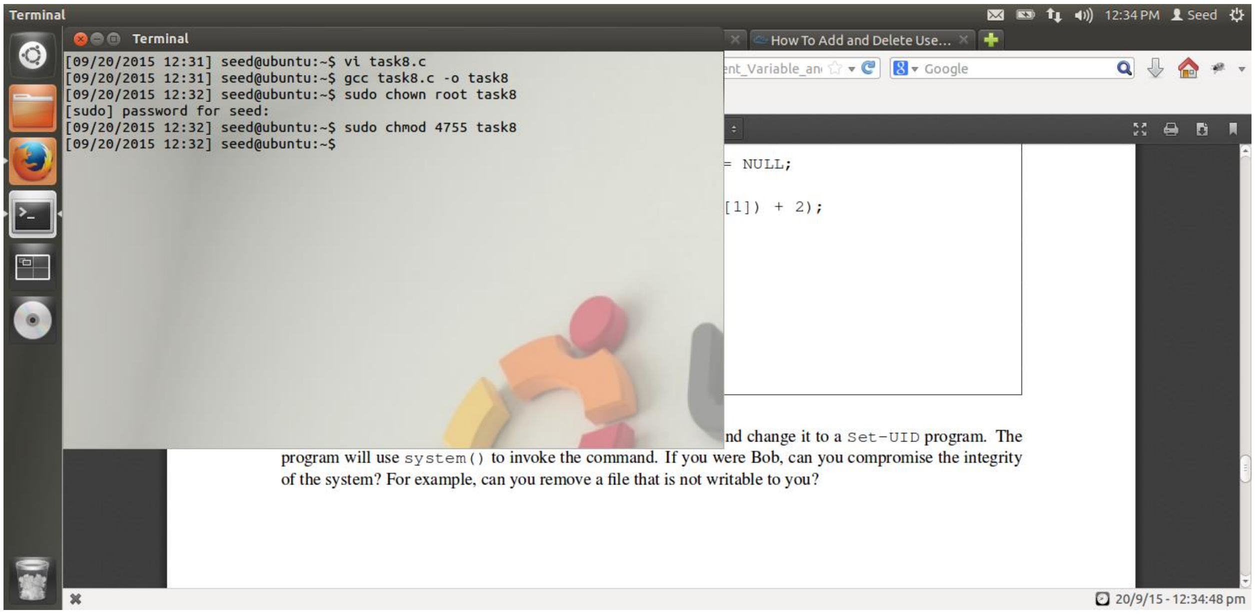


In above screenshot, we have added a new user **yog** to the system and changed ownership of **myprog** to user **yog** using **adduser yog** and **chown yog** commands respectively. We changed **myprog** to **set_UID yog** program.



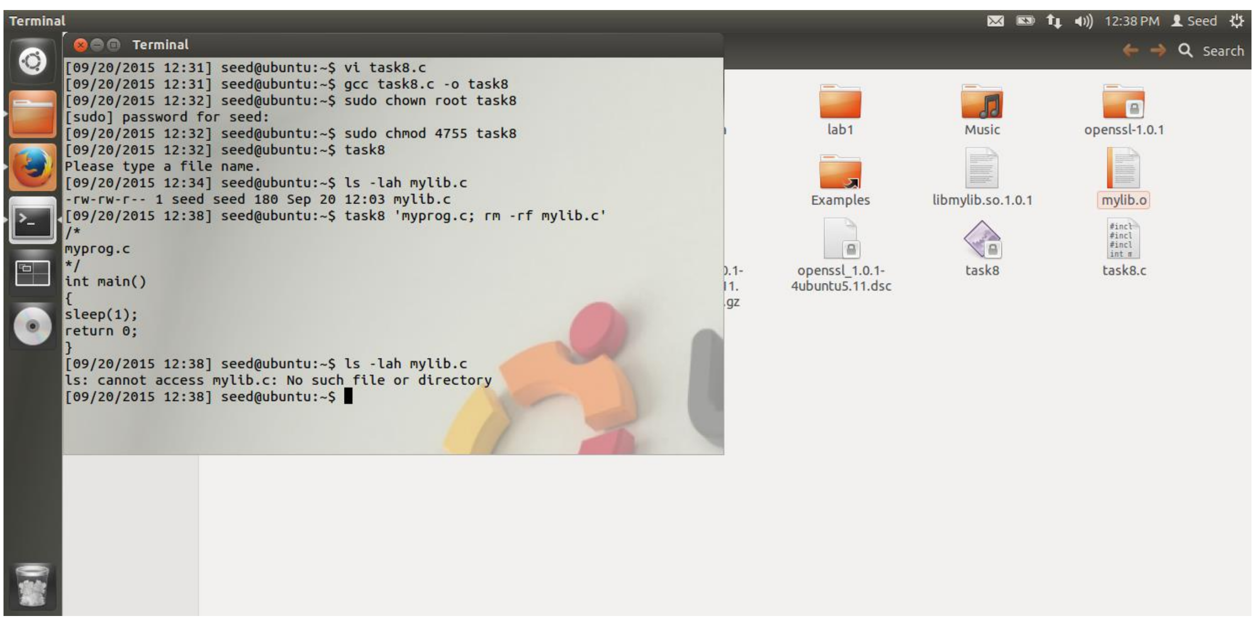
<u>Running **myprog** as set-UID user1 program from user2 shell</u>: Here, we have exported **LD_PRELOAD** environment variable in user2 shell with our predefined library program. When **myprog** is executed, new child process is created in user1 shell environment. In which system tries to make **sleep()** call. But, here user-defined **sleep()** doesnot get called as **LD_PRELOAD** environment variable doesnot exist in user1 shell in which **myprog** is running. Child process running in shell environment of user1 doesnot inherit **LD_* environment variables** form user2 shell.

Task8: Invoking external programs using system() versus execve()



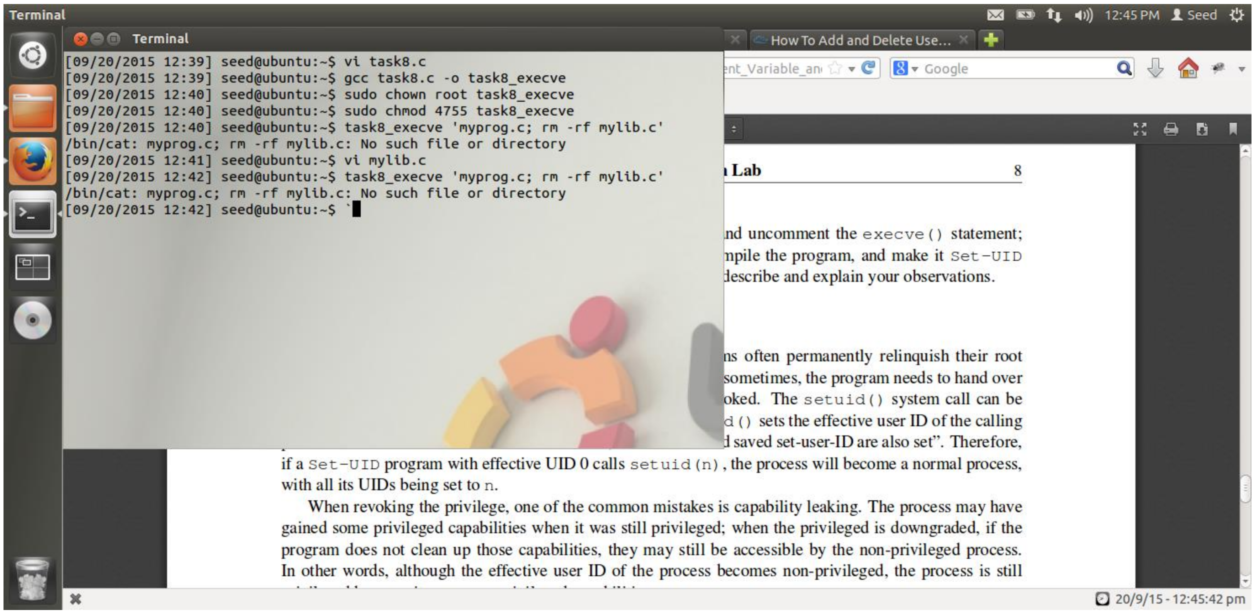
Here, we have compiled **task8.c** code to **task8** file and changed its ownership to **root** by using **chown root** command. We have modified **task8** program to **Set-UID root** program using **chmod 4755** command.

Using system() call:



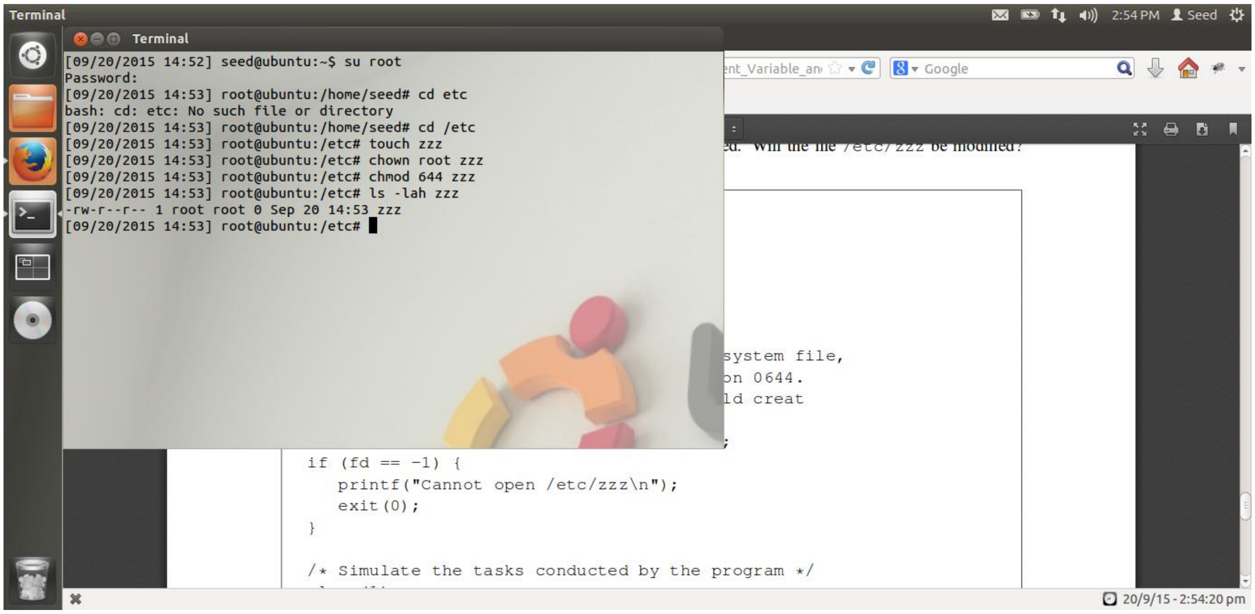
In this scenario, the **mylib.c** file was deleted from the main system. Hence, our attack was successful when **system()** was used. This is because when system is called a new process is created which invokes a root shell and user-input is directly run in this shell as commands without any validation scheme. User can attack such system using user-input in the format 'expected_user_input; attack_command1; attck_command2;....'& so on.

Using execve() call():

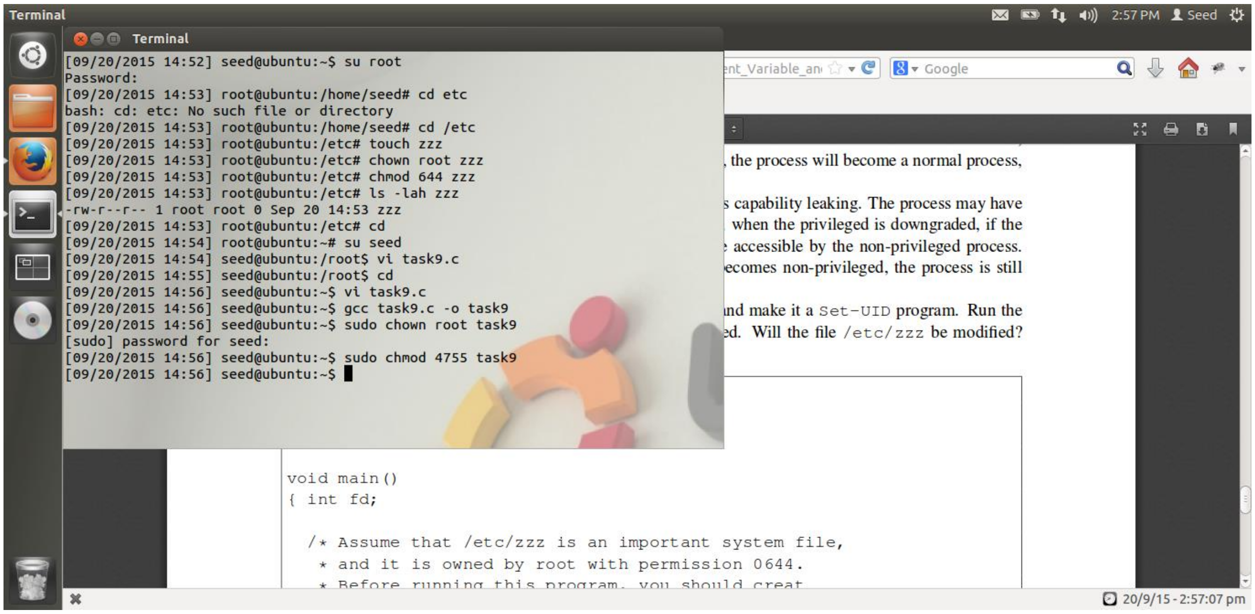


In this scenario, file doesnot get deleted and we get error message from user shell. Hence, our attack was not successful. This attack doesnot work in this case because when user-input is given to **execve()** call, new shell is not invoked rather input is given as arguments to **execve()** call. **execve()** expects input filename from user and when user gives user-input it tries to find that file in system. If attacker has given more than one command to **execve()**, error is invoked as system tries to find file with given commands as filename.

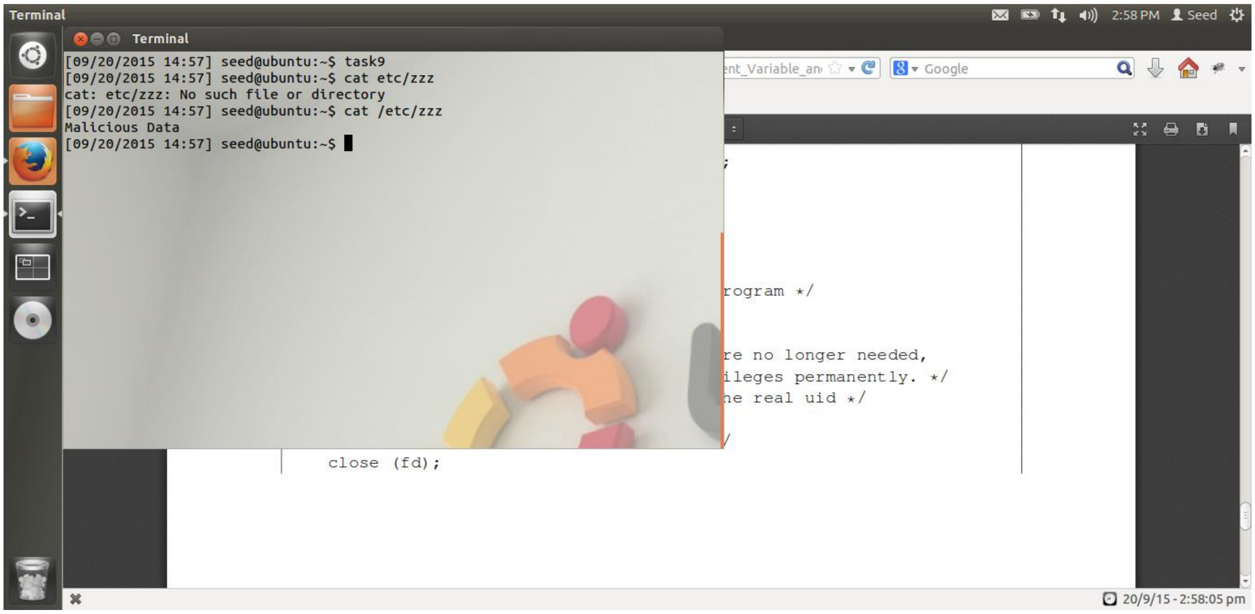
Task9: Capability Leaking



Here we are changing user to **root** using **su root** command and then created a new **zzz** file in **etc** folder using touch command. By using **ls –l** command we can check the privilege access level and owner of zzz file. **644** is **access level** and **root** is **owner**.



Here we have compiled **task9.c** and saved the executional file to **task9** in **seed user**. Now, using **chown root** we are changing owner to **root**. Using **chmod** we set **task9** as **set_UID** program.



After running **task9** and displaying contents of **etc/zzz** using **cat** command, we can deduce that capability to edit etc/zzz file was not revoked by system when we changed the effective userID of task9. When task9 is executed a new child process is created with exact same data and userID, which runs in the root shell environment. When we use **setuid(getuid())** command, the real user id is set as effective userId of parent process, but child process still runs with root privilege, thus allowing attacker to attack system using this child process. As the child process's capability is not revoked by **setuid(getuid())** call, it can still modify etc/zzz file and write malicious data into it. Thus, we can conclude that even if parent processes privilege is downgraded, its capabilities are not revoked.