Environment Variable and *Set-UID* Program Lab

Task 1: manipulating environment variables

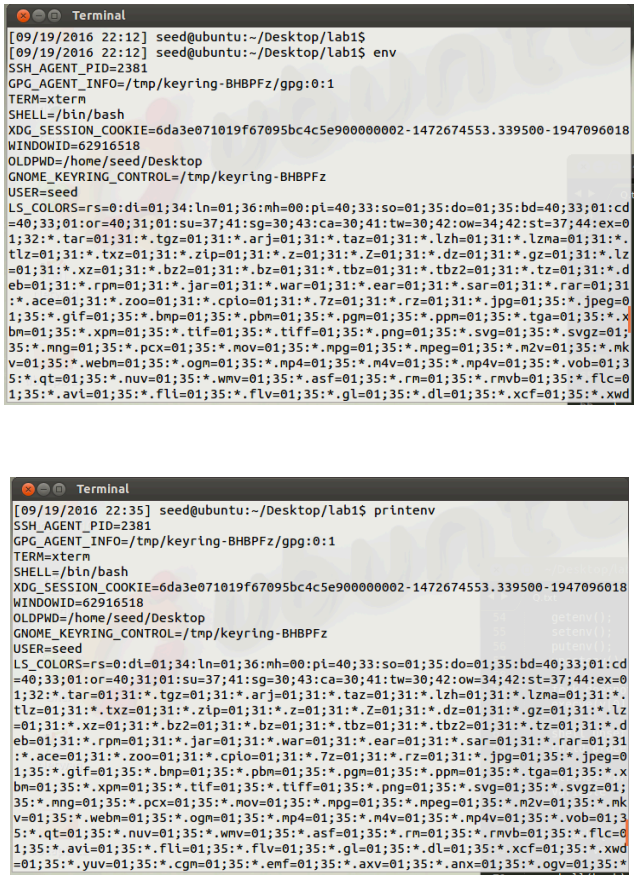

Fig 1. env command print out environment variables

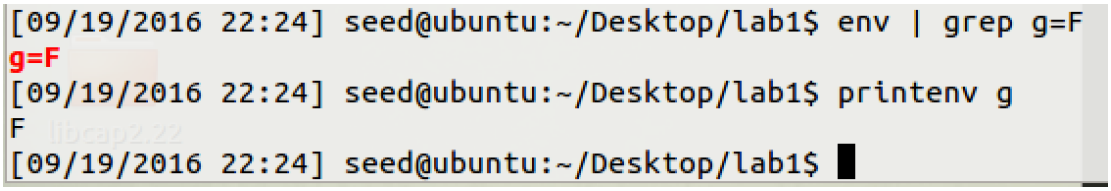

Fig 2. Specify the environment variable and print out

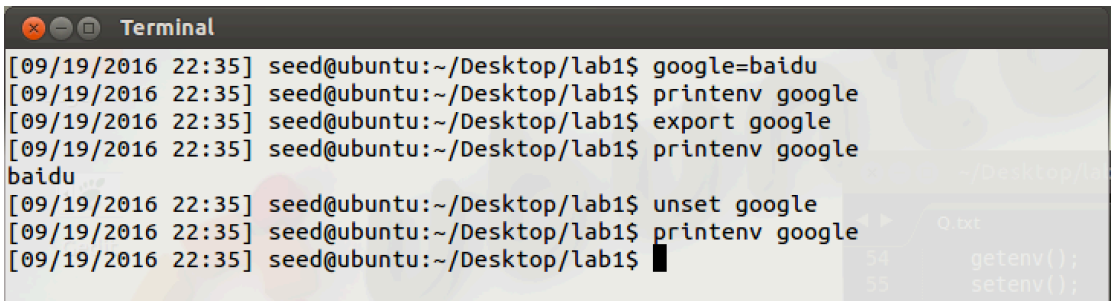

Fig 3. export and unset environment variables

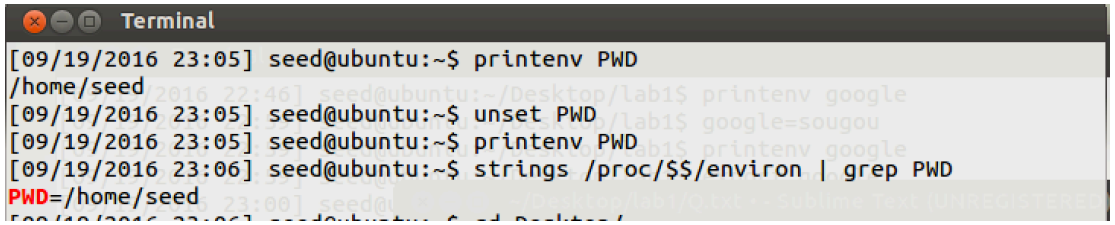

Fig 4. Unset variable effects shell vaiable

Observation:

1.1 From Fig 1, we can know 'printenv' and 'env' command they all print out all the environment variables, actually these printing out are shell variables. And from fig 2, we know that 'printenv [variablename]' and 'env | grep [variablename]' can print out specify environment variable.

1.2 From fig 3, we can use printenv [variablename] print out variable after we export the variable. As google, at first I did not export it. I cannot search it using printenv google, but after I export google, I can print it out using printenv google. The idea for this result is that shell gets deep copy from environment variables, and then I created google=baidu as shell variable, and then use export to make sure child process inherits this variable as environment variable.

1.3 As fig 3, we can use unset [variablename] to remove this variable from shell variables. Like fig 4, unset can remove shell variable but it cannot remove the environment variable.

Explanation:

1.4 For Fig 3, as google, at first I did not export it. I cannot search it using printenv google, but after I export google, I can print it out using printenv google. The idea for this result is that shell gets deep copy from environment variables, and then I created google=baidu as shell variable, and then use export to make sure child process inherits this variable as environment variable.

1.5 Shell can deep copy environment variables called shell variables. Then child process can access these variables after they were exported. Command printenv is kind of child process, so export should be used.

Task2: Inheriting environment variables from parents

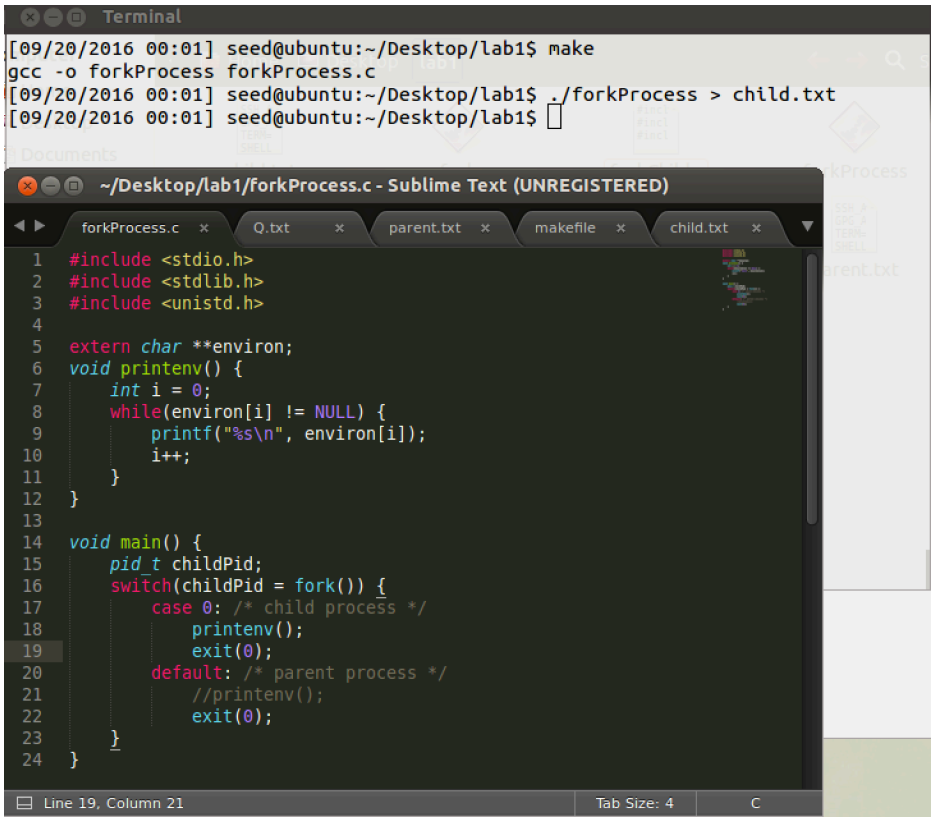

Fig 2.1 fork child process print environment variables

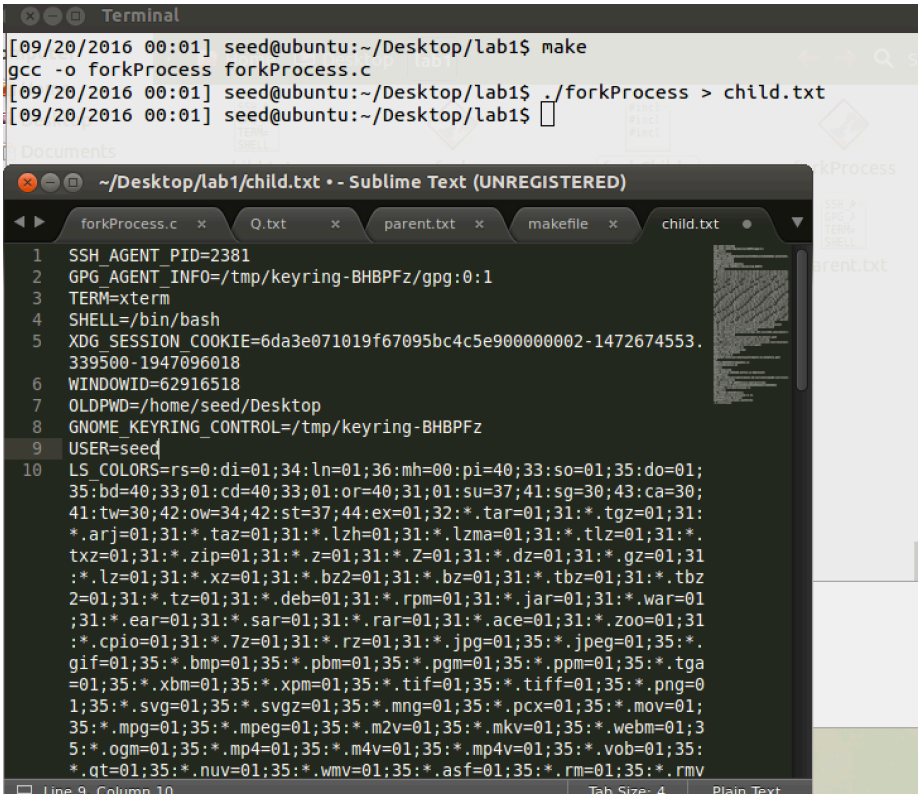

Fig 2.2 result of fork child process print environment variables

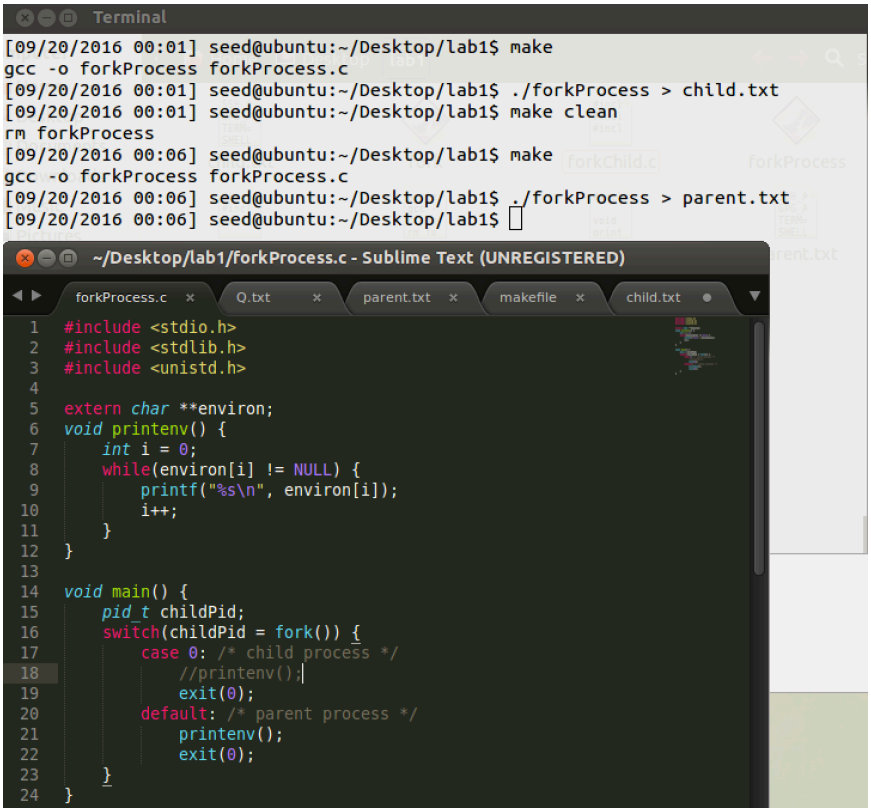

Fig 2.3 parent process print environment variables

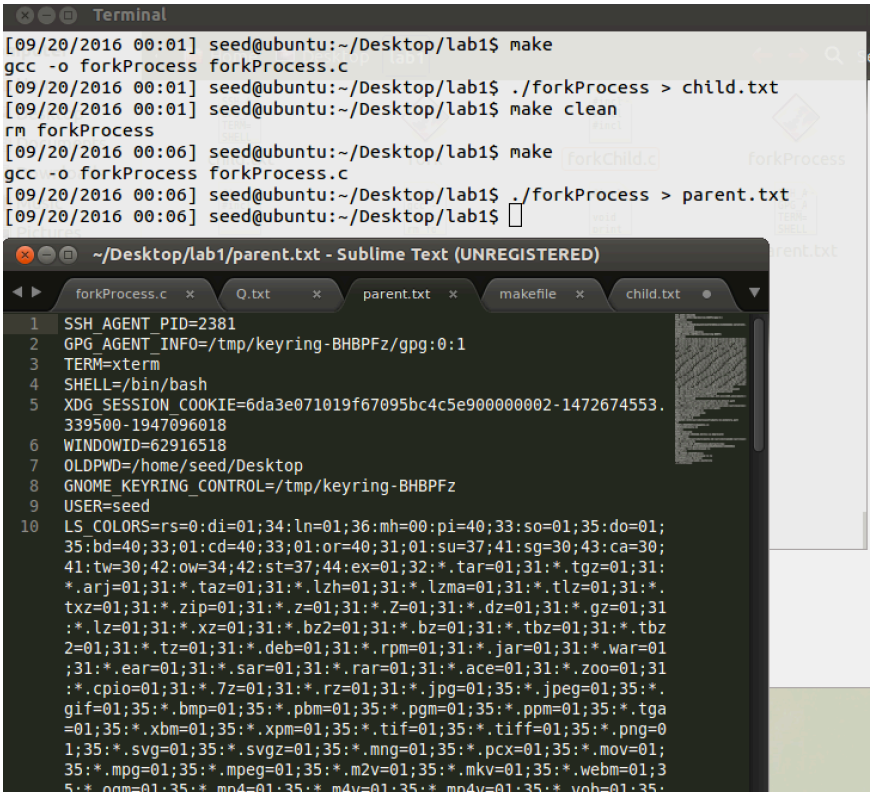

Fig 2.4 result of parent process print environment variables

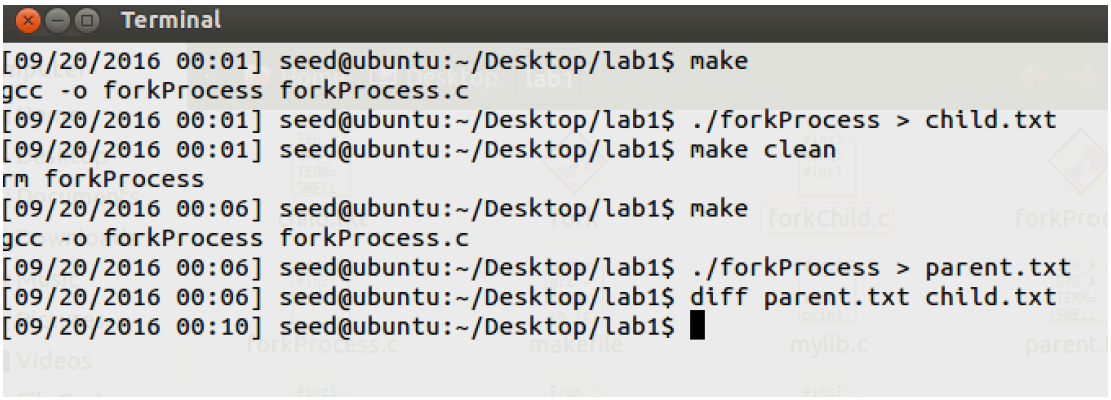

Fig 2.5 compare child.txt and parent.txt

Observation:

2.1 From Fig 2.1, and Fig 2.2 we can get information that we use fork function to make child process out put some environment variables and wrote them in child.txt.

2.2 From Fig 2.3, and Fig 2.4 we can get information that we use parent process out put some environment variables wrote them in parent.txt.

2.3 As Fig 2.5, we found that contents of child.txt and parent.txt have no difference.

Explanation:

2.4 No difference between child and parent process variables means that fork() can produce child process, and child process could inherit environment variables from parent process.

Task3: Environment variables and execve()

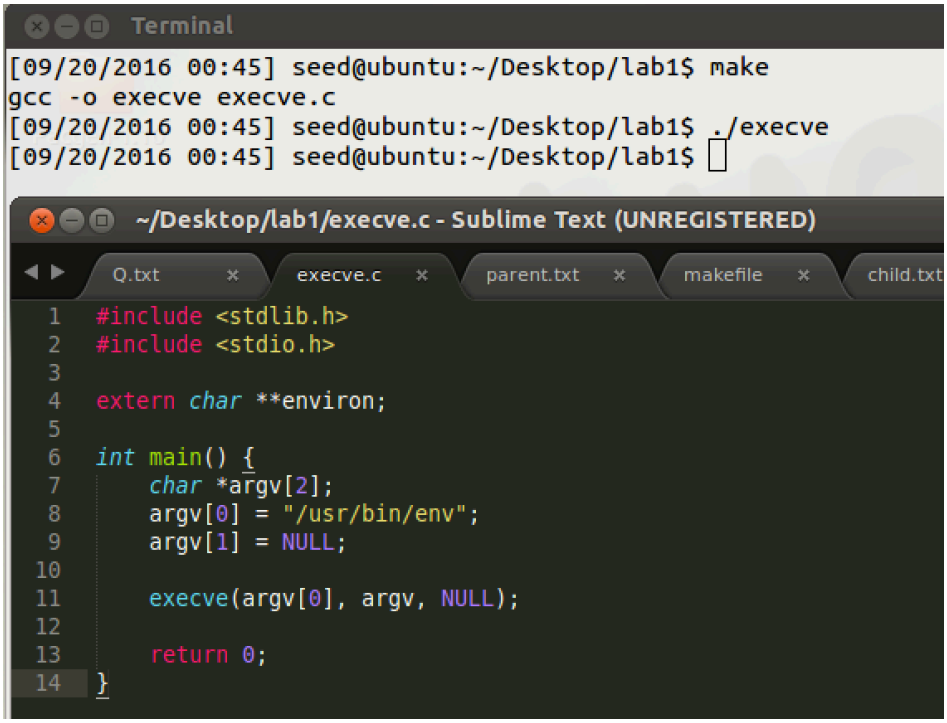

Fig 3.1 execve 3rd argument equals NULL

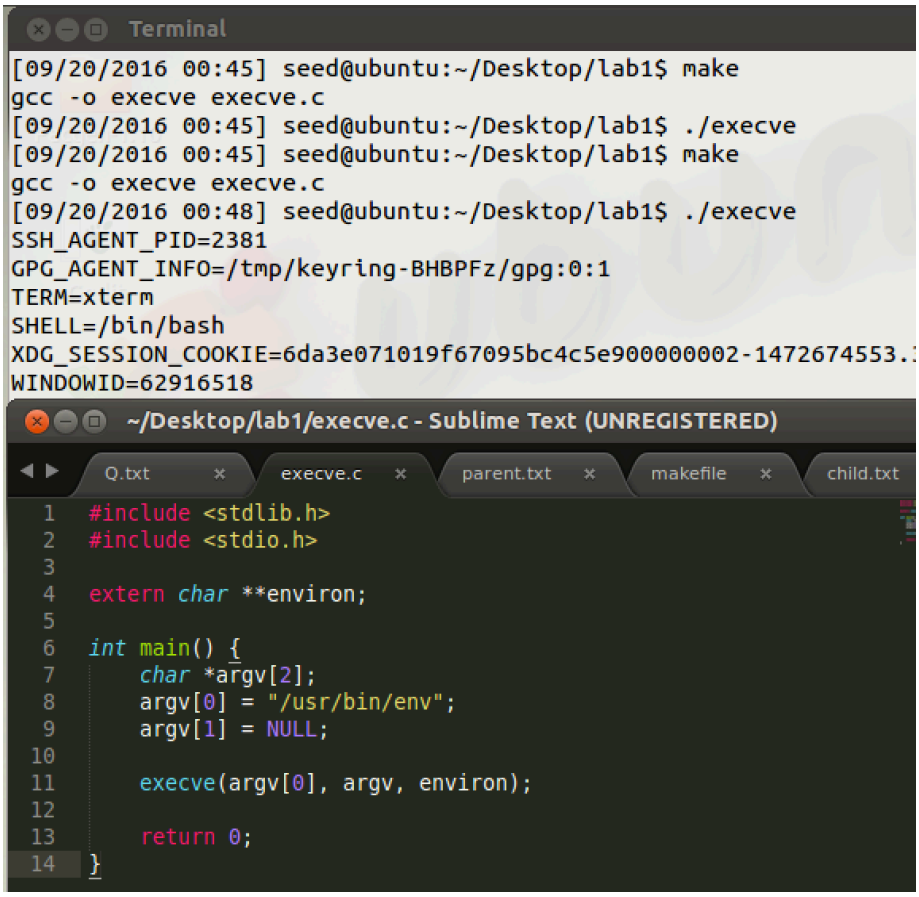

Fig 3.2 execve 3rd argument with environ

Observation:

3.1 From Fig 3.1, the 3rd argument with NULL, execve function did not print out environment variables.

3.2 From Fig 3.2, the 3rd argument with environ, execve function printed out environment variables.

Explanation:

3.3 3rd argument of execve is used for passing environment variables. We can specify single or several variables or the whole environ. The process which runs execve could get the environment variables according to 3rd argument. So execve could not get environment variables automatically, they must specify as argument.

Task4: Environment variables and system()

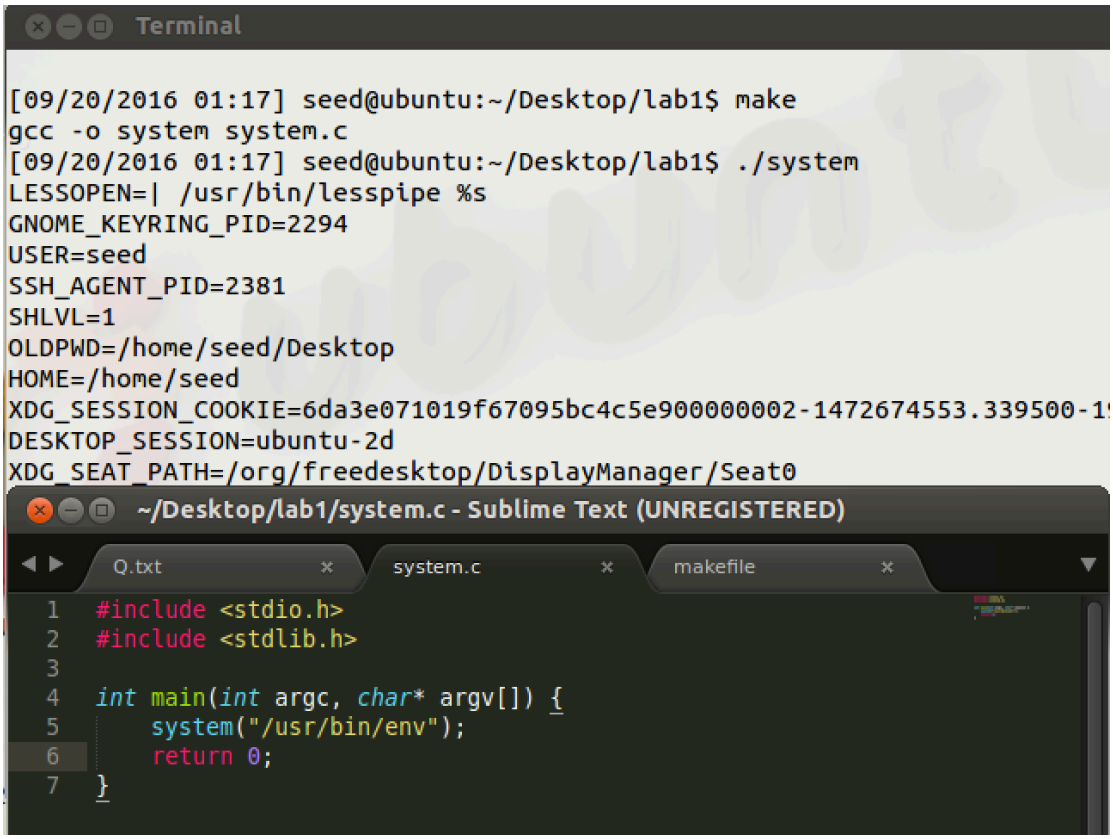

Fig 4.1 run system.c file

Observation:

4.1 From Fig 4.1, system did not specify any environment variable to this process, but this process could print out all the environment variables.

Explanation:

4.2 Function system call execl() to execute /bin/sh and execve(). Moreover, it pass current environ to execve(). Function system could have environment variables automatically. So it is quit dangerous if used in a privileged program.

Task5: Environment variables and Set-UID Programs

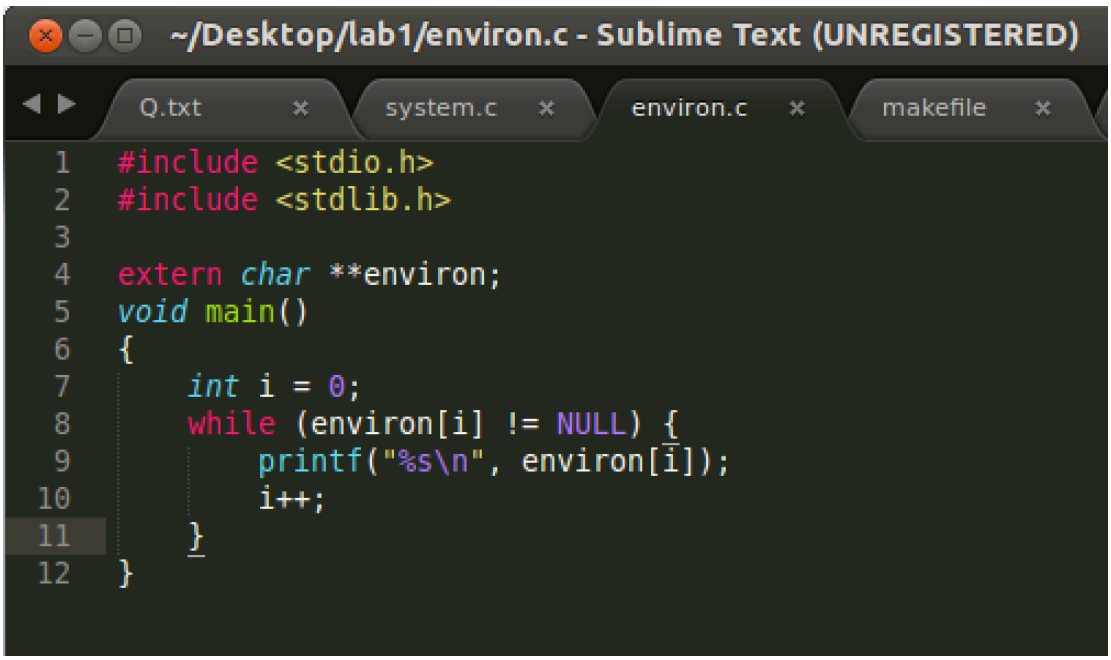

Fig 5.1 print environment variables program

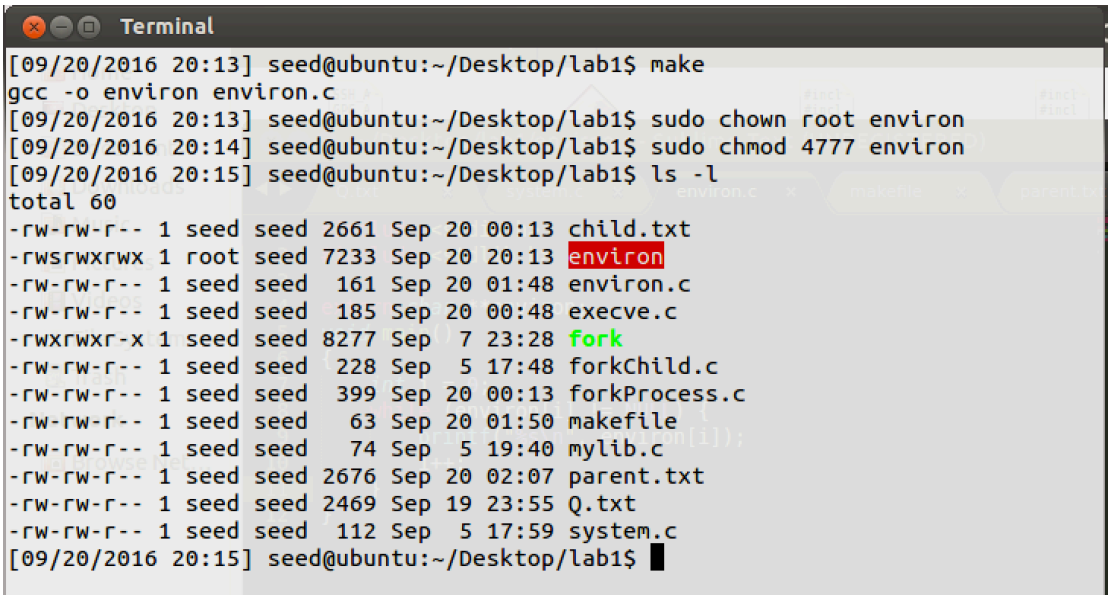

Fig 5.2 change environ owner to root and make it set-uid program

```
[09/20/2016 20:23] seed@ubuntu:~/Desktop/lab1$ ./environ > env.txt
[09/20/2016 20:23] seed@ubuntu:~/Desktop/lab1$ cat env.txt | grep google=
google=baidu
[09/20/2016 20:24] seed@ubuntu:~/Desktop/lab1$ cat env.txt | grep PATH=
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
DEFAULTS_PATH=/usr/share/gconf/ubuntu-2d.default.path
PATH=/seed/Wenbin:.:/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/usr
/sbin:/usr/bin:/sbin:/bin:/usr/games
MANDATORY_PATH=/usr/share/gconf/ubuntu-2d.mandatory.path
[09/20/2016 20:24] seed@ubuntu:~/Desktop/lab1$ cat env.txt | grep LD_LIBRARY_PAT
H
[09/20/2016 20:25] seed@ubuntu:~/Desktop/lab1$
```

Fig 5.3 run ./environ program and make it out write in env.txt and then search variables

Observation:

5.1 Fig 5.1, this is the program to print out environment variables.

5.2 As Fig 5.2, I compiled the environ.c file, and changed owner as root and set this program as a Set-UID program.

5.3 Run this program. From Fig 5.3, I printed out all the environment variables in env.txt. Then I searched google, PATH, LD_LIBRARY_PATH variable in env.txt. The result is that we found google and PATH. But, I did not find LD_LIBRARY_PATH variable.

Explanation:

5.4 The reason of why LD_LIBRARY_PATH not be displayed is kind of security feature. This is one. Set-UID program will ignore LD_LIBRARY_PATH. Otherwise, this is a security hole. For example, you can make a self-made program referenced by path of Set-UID program. So that self-made program could run as root privilege.

Task6: The PATH Environment variable and Set-UID Programs

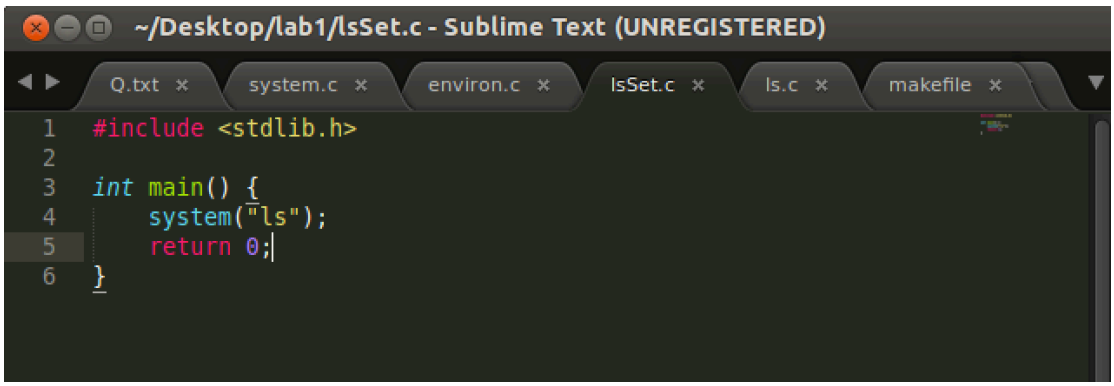

Fig 6.1 a program call system().

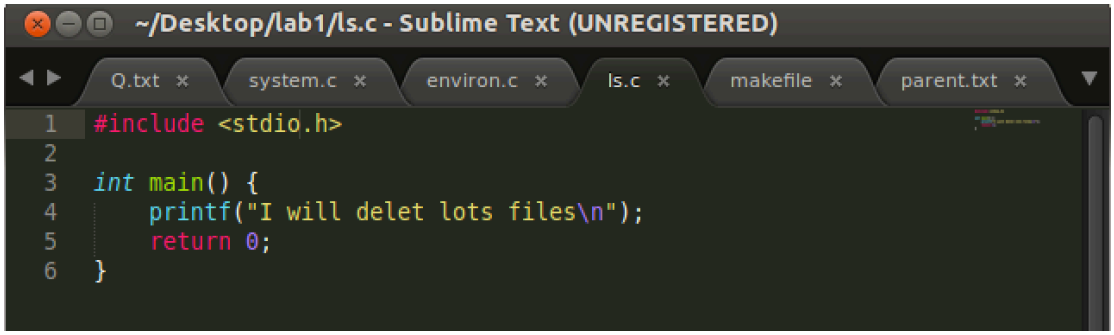

Fig 6.2 a program file named ls.c

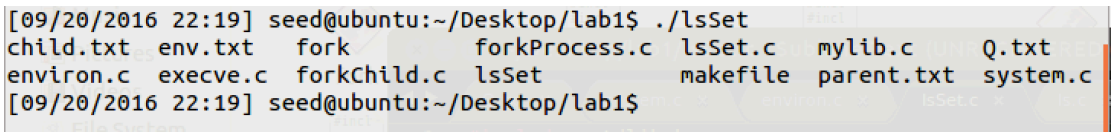

Fig 6.3 run ./lsSet afer compile this file

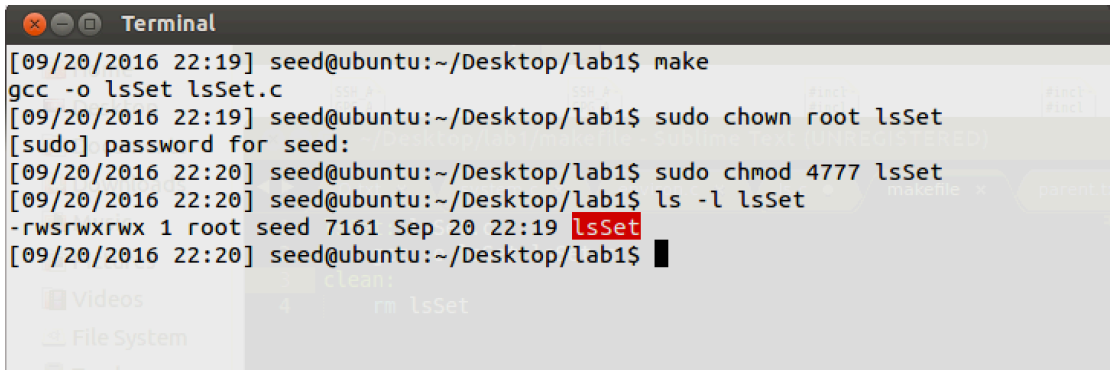

Fig 6.4 compile lsSet.c and change owner to root, and make it Set-UID program

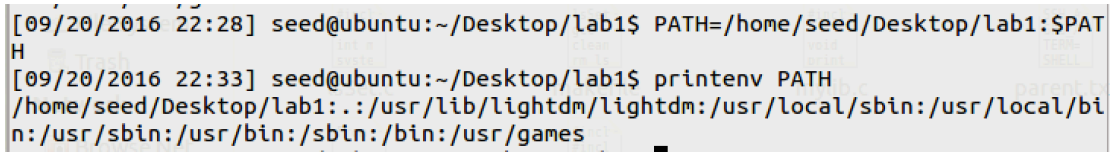

Fig 6.5 change PATH variable.

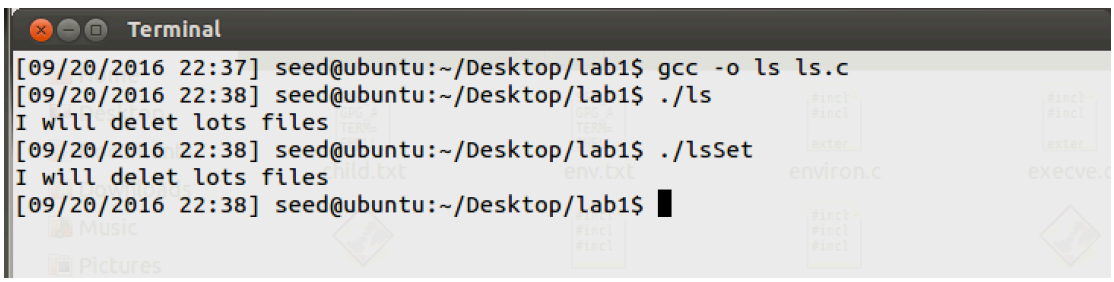

Fig 6.6 compile ls.c file and run ls and lsSet.

Fig 6.7 change ls.c file and print uid and euid

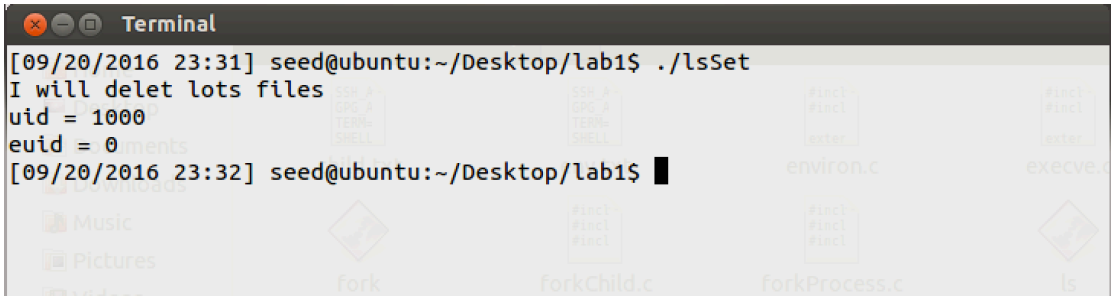

Fig 6.8 call ./lsSet to make sure ls has root privilege.

Observation:

6.1 Fig 6.1 and Fig 6.2 are two program used in this task. As Fig6.4, compile lsSet.c file and change owner to root and then make it Set-UID program.

6.2 As Fig6.5, change PATH variable: PATH=/home/seed/Desktop/lab1:$PATH

6.3 From Fig6.6, compile ls.c file, and then run this file. Later, run lsSet. We can find it print "I will delet lots files" rather than list files, compare to Fig 6.3.

6.4 From Fig 6.8, we can get information uid = 1000, euid = 0. So that ./ls has root privilege.

Explanation:

6.5 system() will get environment variables from shell variables. Because I changed PATH variable, I redirected /bin/ls to /home/seed/Desktop/lab1/ls. So sytem("ls") will call shell to invoke /home/seed/Desktop/lab1/ls.

6.6 When I run ./lsSet, I get root privilege, because it is a Set-UID program. So root privilege process' child process still has root privilege.

Task7: The LD_PRELOAD environment variable and Set-UID Programs

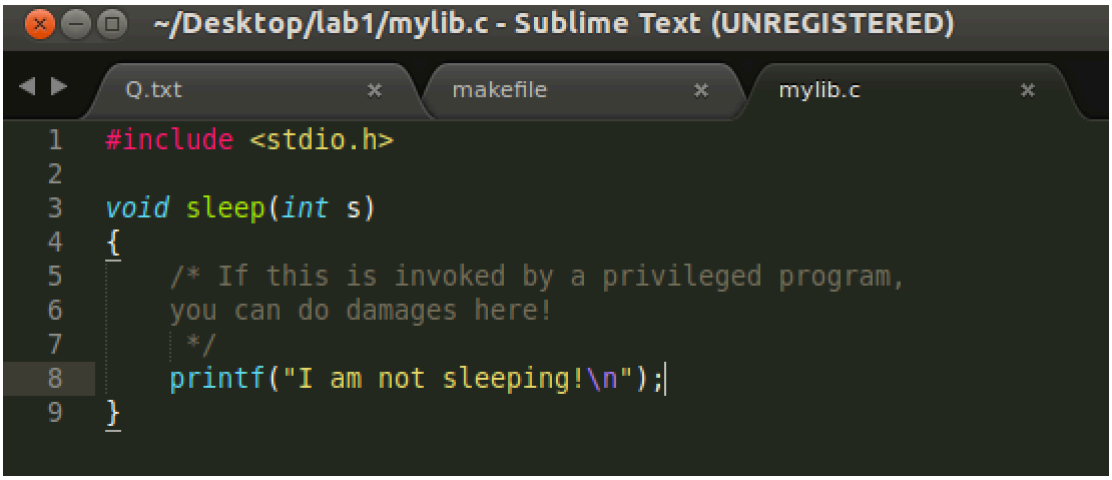

Fig 7.1 mylib.c

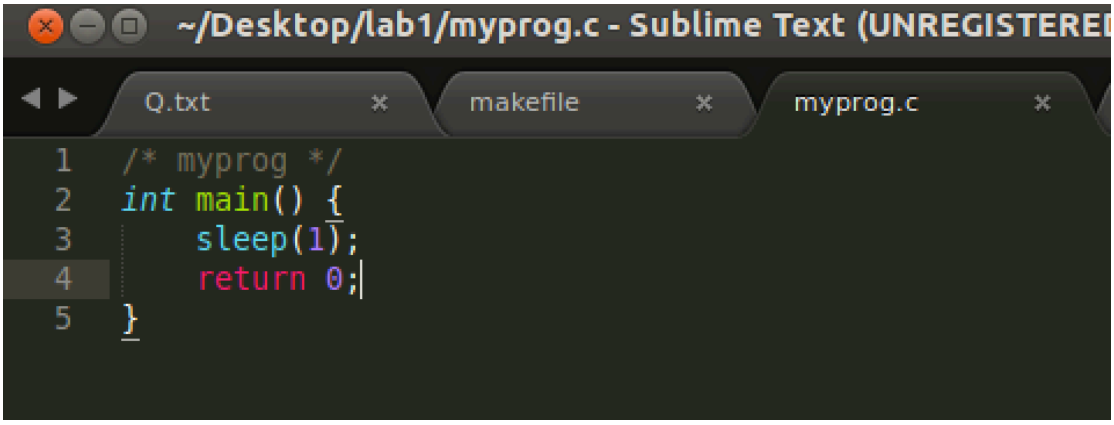

Fig 7.2 myprog.c

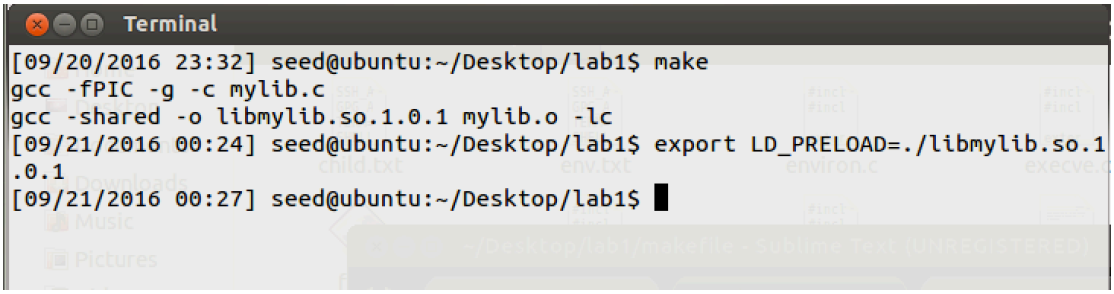

Fig 7.3 create libmylib.c and export LD_PRELOAD

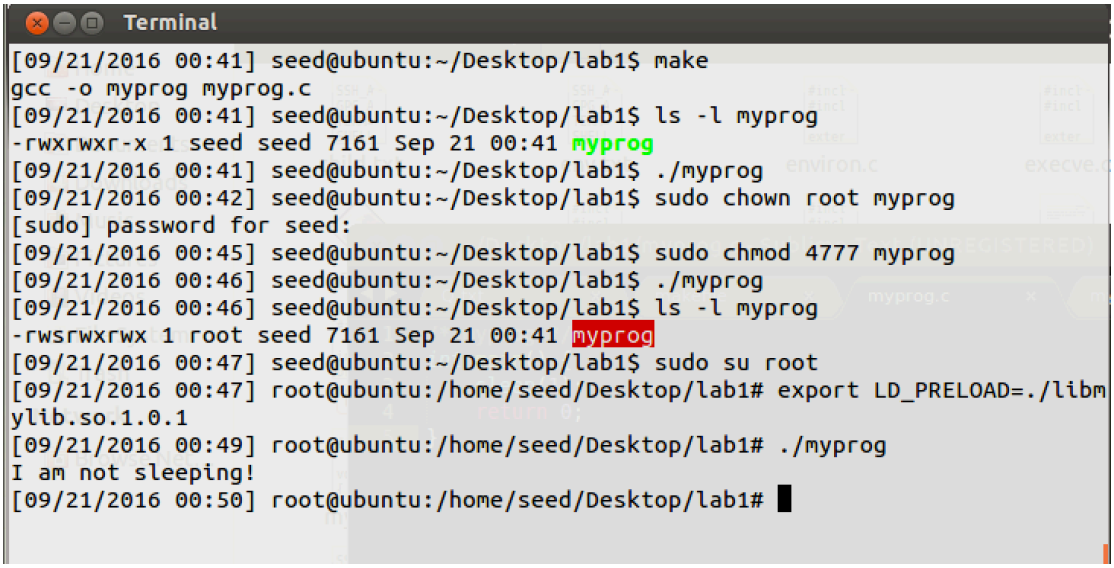

Fig 7.4 three process for step 2

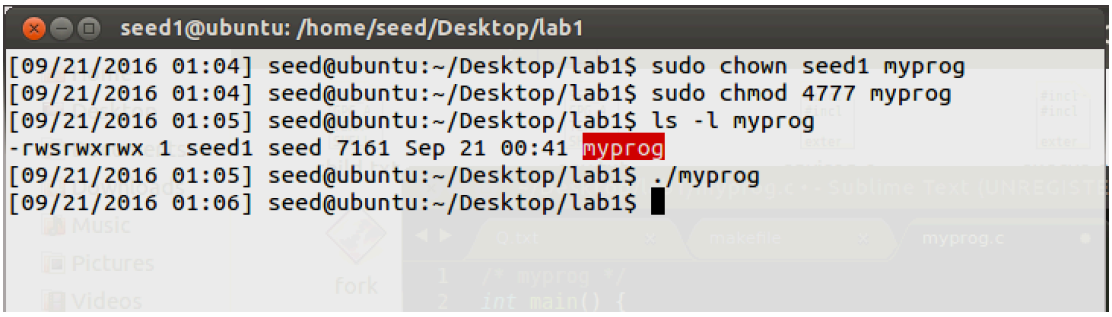

Fig 7.4 last process for step 2

Observation:

7.1 seed user run regular program, sleep() be called normally.

7.2 seed user run Set-UID root program, sleep() be called normally.

7.3 Root account run Set-UID root program, sleep() defined in libmylib.so.1.0.1 be called.

7.4 seed user run Set-UID root program, sleep() be called normally.

Explanation:

7.5 from the result of this experiment we can get conclusion that when normal user run Set-UID program LD_* variables will be ignored.

7.6 If root account, export variable and run Set-UID LD_* will not be ignored. So only root account can run export LD_* effected program.

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

~/Desktop/lab1/task8.c • - Sublime Text (UNREGISTERED)

Q.txt | makefile | task8.c | mylib.

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    char *v[3];
    char *command;
    if (argc < 2) {
        printf("please type a file name\n");
        return 1;
    }
    v[0] = "/bin/cat";
    v[1] = argv[1];
    v[2] = NULL;
    command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
    sprintf(command, "%s %s", v[0], v[1]);
    //use only one of the following
    system(command);
    //execve(v[0], v, NULL);
    return 0;
}
```

Fig 8.1 call system()

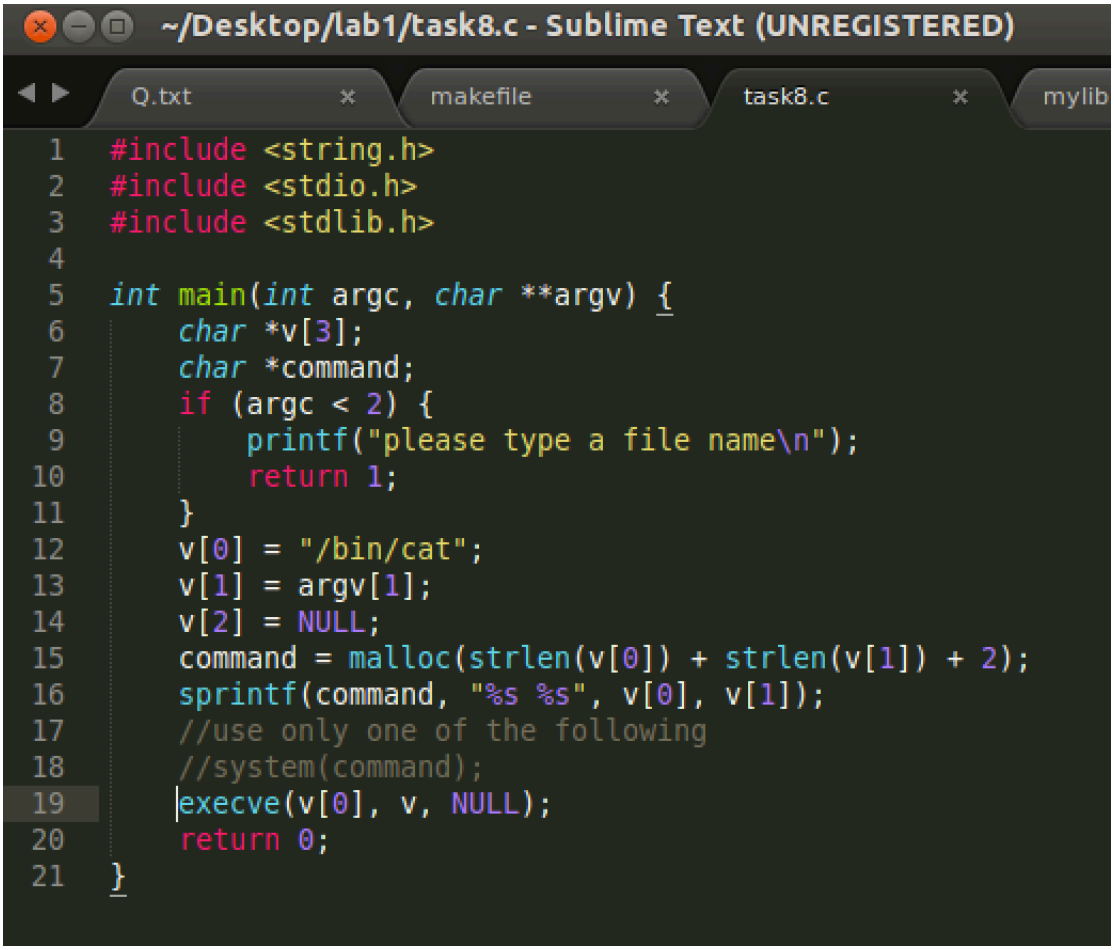

Fig 8.2 call execve()

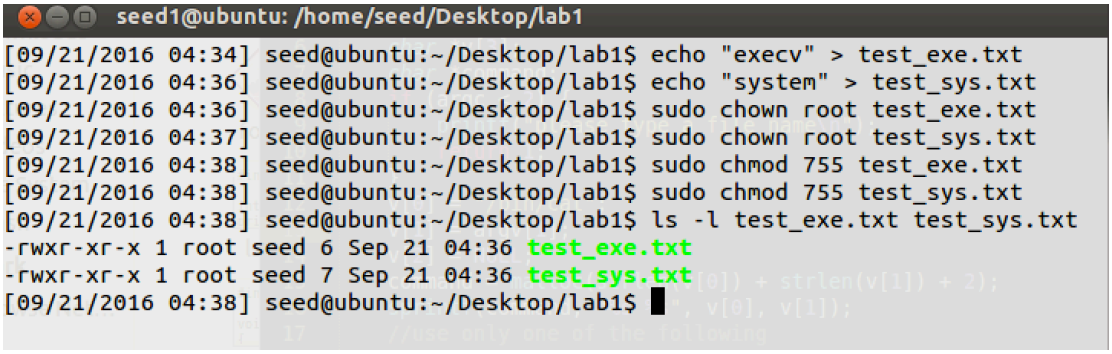

Fig 8.3 create two not writeable root owner file

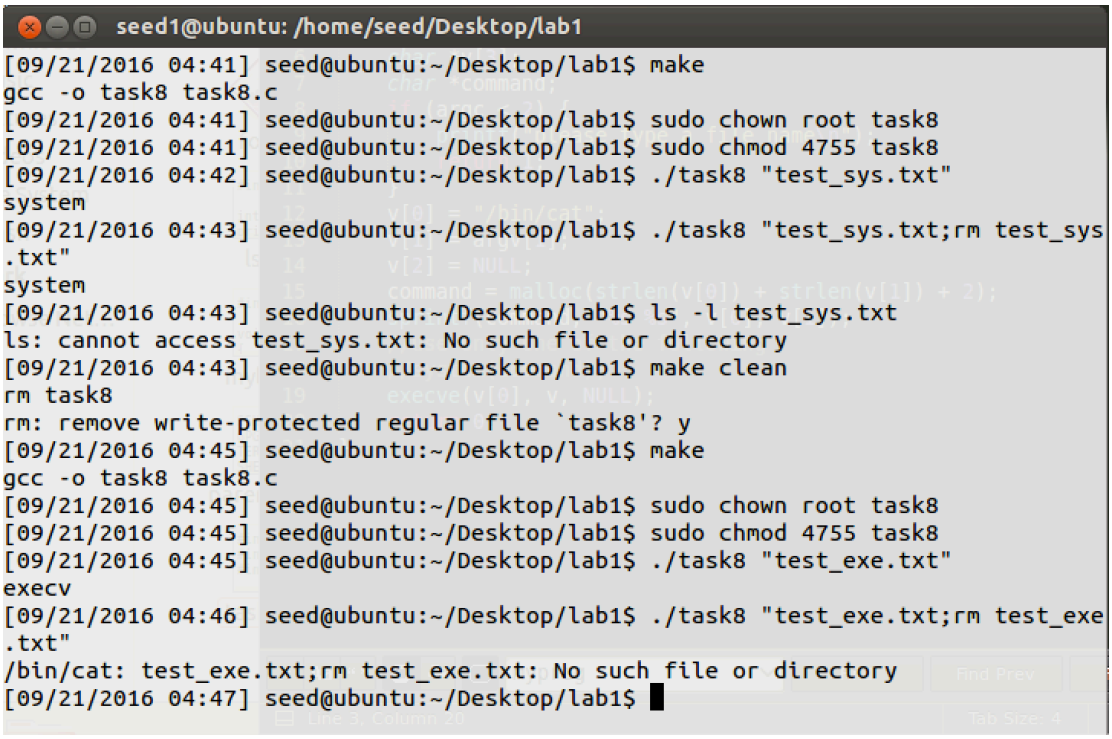

Fig 8.4 procedure for lab

Observation:

8.1 From Fig 8.4, when system() is called, input "./task8 test.txt" will get contents of test.txt. "./task8 test.txt;rm test.txt" will remove not writeable test_sys.txt

8.2 From Fig 8.4, when execve() is called, input "./task8 test.txt" will get contents of test.txt. "./task8 test.txt;rm test.txt" will print error message.

Explanation:

8.3 Set-UID program can help user get file owner's privilege. As file owner is root, so this program could get root privilege. System() will invoke shell to run command. If "file;command2" is input for this program, command2 will be executed. So the file can be removed.

8.4 Execve is not invoked from shell. If we use "file;command2" as input, this string will be pass to execv(file, args[], enrp) as second argument. This function will not understand input arguments.

Task9: Capability Leaking

Fig 9.1 task9 Set-UID program code

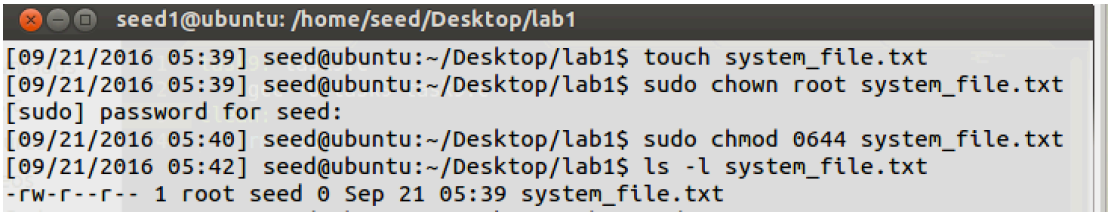

Fig 9.2 make a test file owned by root and permission with 0644

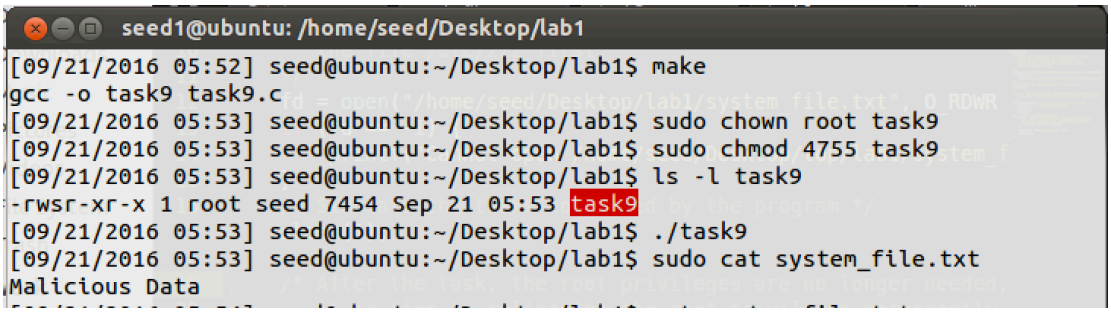

Fig 9.3 make task9.c owned by root and Set-UID, compiele and run task 9 code

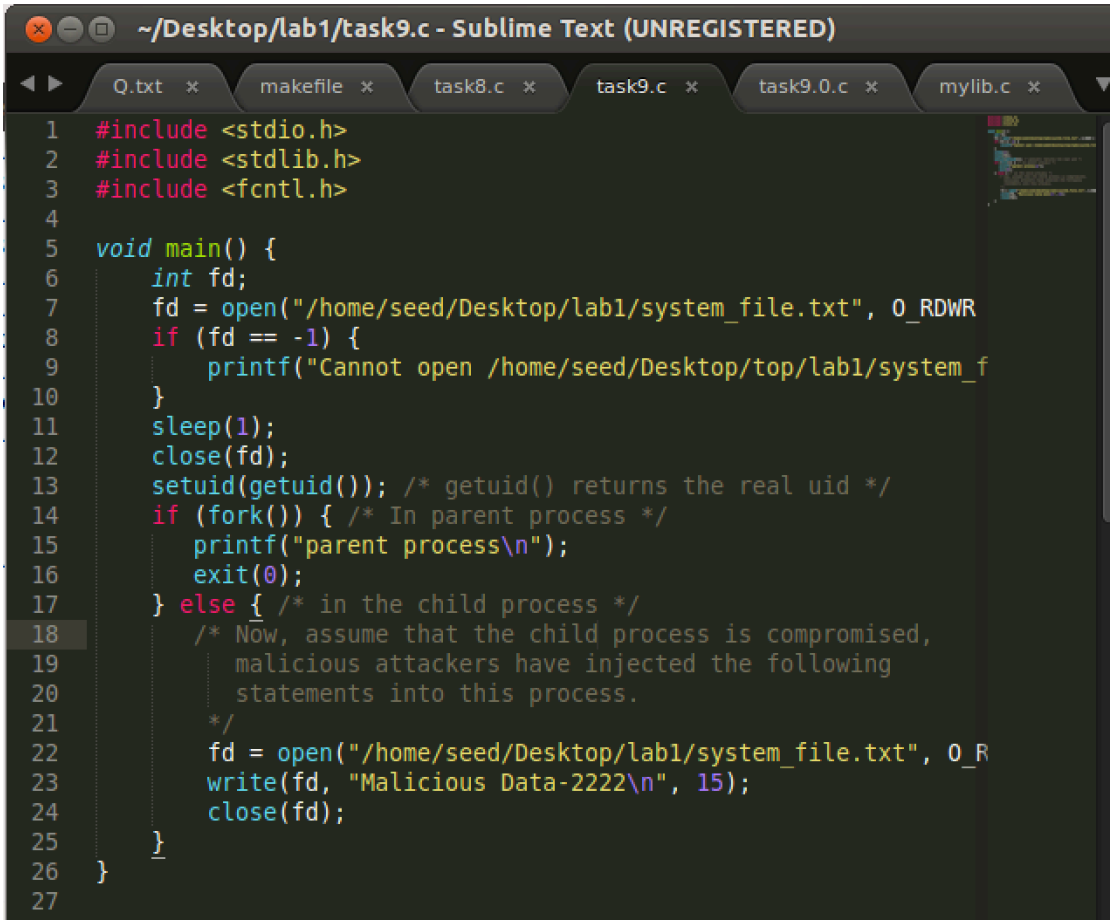

Fig 9.4 change code to test child process privilege

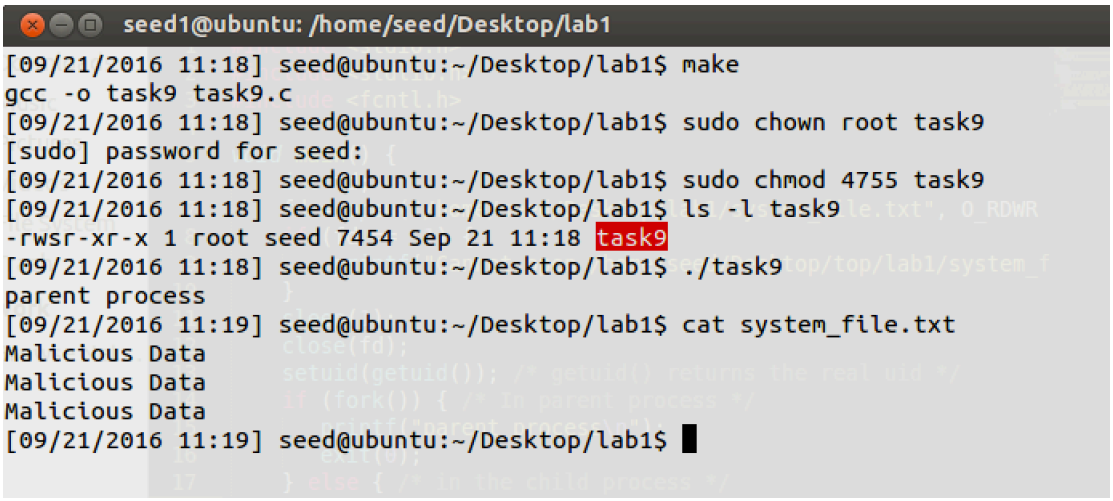

Fig 9.5 new text is not be written in test file.

Observation:

9.1 From Fig 9.3, after I run ./task9, I can cat contents of “Malicious Data”.

9.2 As Fig 9.4, 9.5, new text “Malicious Data-2222” cannot be written in test file.

Explanation:

9.3 The test file system_file.txt is root owned and permission 0644, so that only user with root privilege could write something in it. So child process forked by ./task9 had root privilege.

9.4 Because task9 is owned by root and Set-UID program, when it is run, user could get root privilege.

9.5 getuid() returns real uid number, so setuid(getuid()) can set euid back and equal to real uid. After euid back to real uid, its privilege downgraded. But, the process is still privileged because it possesses privilege capabilities.

9.6 In this program, file descriptor was opened with root privilege and then child process was forked. Child process had a fd with root privilege opened and child process can run it. So child process can write its thing in this file.

9.7 As Fig 9.4, I closed fd, before fork. Then child process will not have root privilege to open this file when they open it in child process with normal privilege.