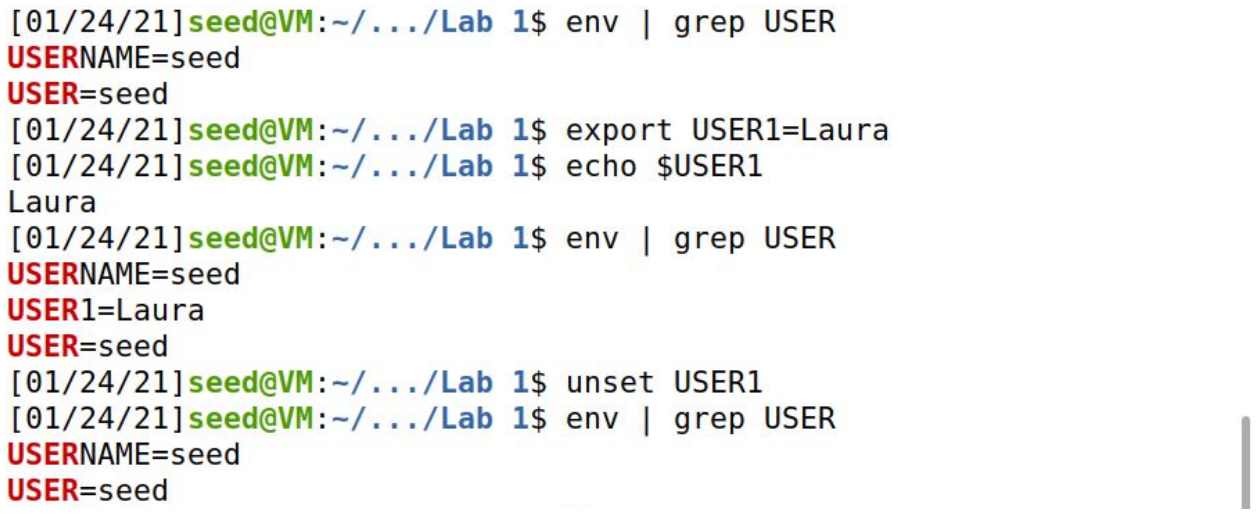
**Environment Variables and Set-UID Programs**

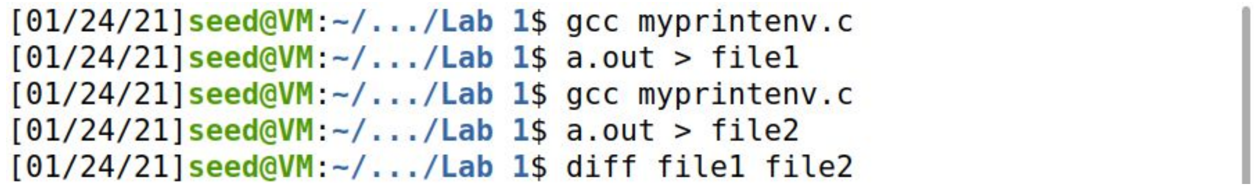
**Task 1:**



```
[01/24/21]seed@VM:~/.../Lab 1$ env | grep USER
USERNAME=seed
USER=seed
[01/24/21]seed@VM:~/.../Lab 1$ export USER1=Laura
[01/24/21]seed@VM:~/.../Lab 1$ echo $USER1
Laura
[01/24/21]seed@VM:~/.../Lab 1$ env | grep USER
USERNAME=seed
USER1=Laura
USER=seed
[01/24/21]seed@VM:~/.../Lab 1$ unset USER1
[01/24/21]seed@VM:~/.../Lab 1$ env | grep USER
USERNAME=seed
USER=seed
```

First I did a command using env and grep to only show environment variables with the substring "USER" in the name. (If I just did env, that would produce all the environment variables.) There were two variables: USERNAME and USER. Then I used export to create a new environment variable called USER1 and set that equal to my name. I used echo to see the value of this variable and confirm that USER1 was created properly. Then, when I did the same env command as before, my USER1 variable was there along with the two others. Once I unset USER1 and did the env command again, my variable was deleted. This shows how to monitor and alter environment variables.

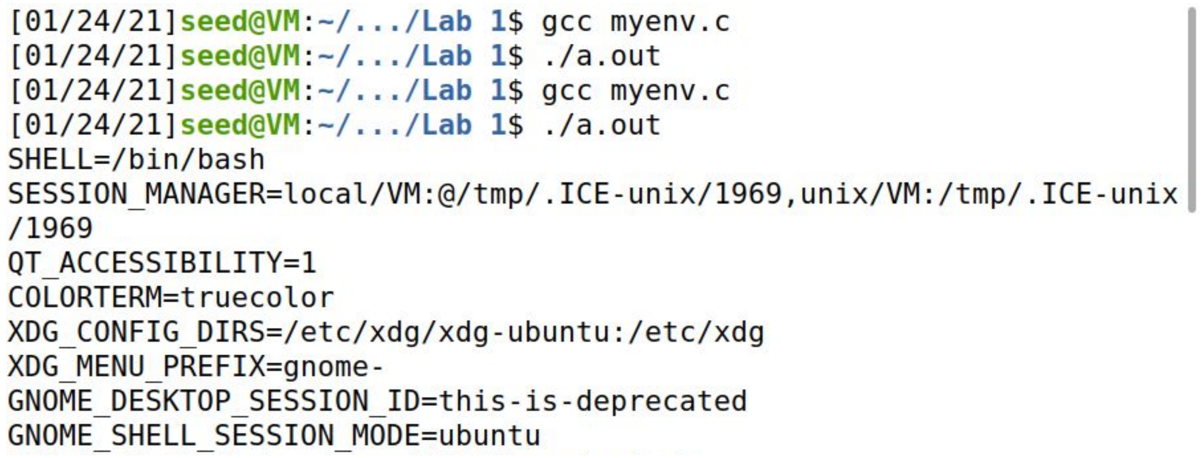
**Task 2:**



```
[01/24/21]seed@VM:~/.../Lab 1$ gcc myprintenv.c
[01/24/21]seed@VM:~/.../Lab 1$ a.out > file1
[01/24/21]seed@VM:~/.../Lab 1$ gcc myprintenv.c
[01/24/21]seed@VM:~/.../Lab 1$ a.out > file2
[01/24/21]seed@VM:~/.../Lab 1$ diff file1 file2
```

First I compiled myprintenv.c and put the output into file1. This means that file1 contained the environment variables of the child process spawned by fork. Then I modified myprintenv.c such that the program printed the environment variables of the parent process, recompiled, and saved that output to file2. Using the diff command, there was no result. This means that file1 and file2 were identical, which means that the parent and child process have the same environment variables. We conclude that a child process inherits it's variables from the parent process.

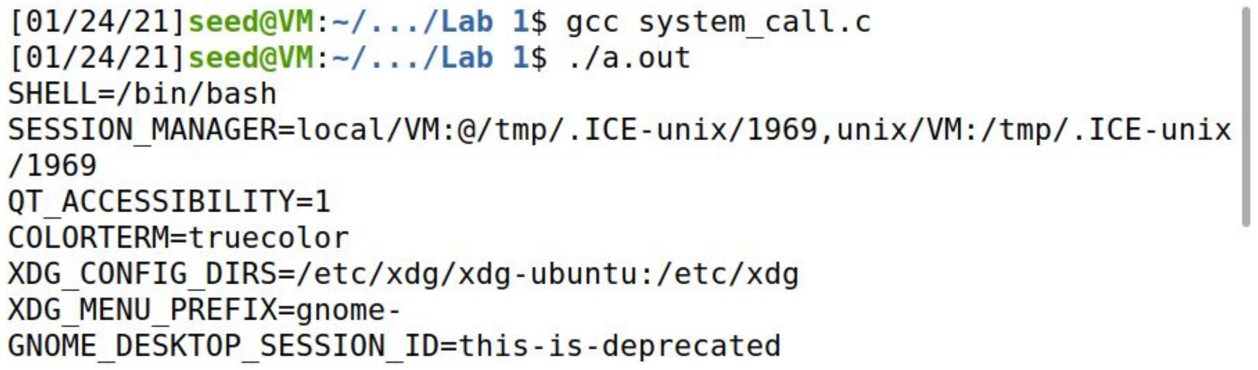
**Task 3:**



```
[01/24/21]seed@VM:~/.../Lab 1$ gcc myenv.c
[01/24/21]seed@VM:~/.../Lab 1$ ./a.out
[01/24/21]seed@VM:~/.../Lab 1$ gcc myenv.c
[01/24/21]seed@VM:~/.../Lab 1$ ./a.out
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1969,unix/VM:/tmp/.ICE-unix
/1969
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
```

Upon first compiling and running myenv.c there is no output. Then, once we change the NULL parameter to environ, the output is the environment variables of the calling process. Based on this, I assume the new program gets it's environment variables from the calling process through the third argument to execve.

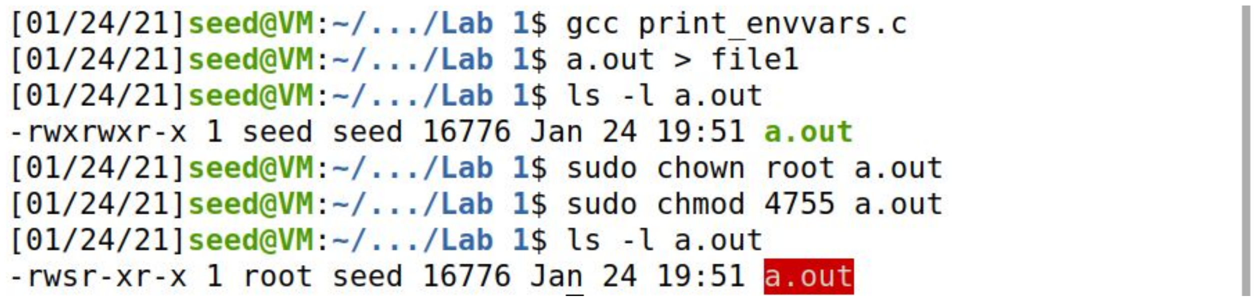
**Task 4:**



```
[01/24/21]seed@VM:~/.../Lab 1$ gcc system_call.c
[01/24/21]seed@VM:~/.../Lab 1$ ./a.out
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/1969,unix/VM:/tmp/.ICE-unix
/1969
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
```

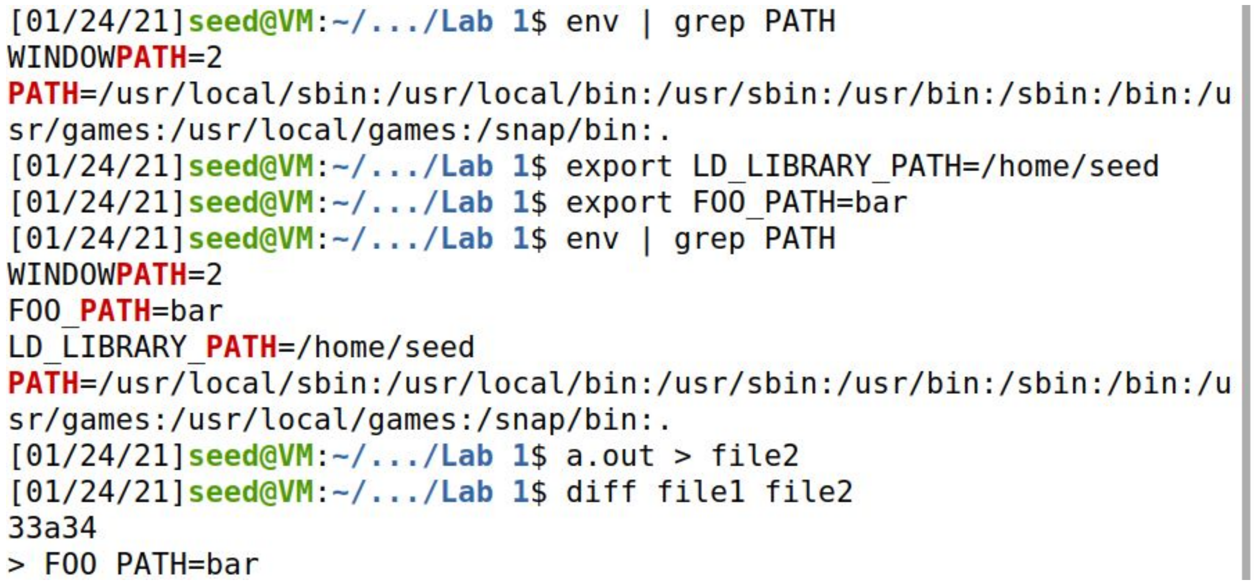
I compiled and ran the given code under the name system_call.c. The result was the environment variables of the calling process. This is because the system function implicitly calls execve and uses it to pass the variables along to the new program. So, when using system, environment variables are automatically transferred from one process to the other.

**Task 5:**



```
[01/24/21]seed@VM:~/.../Lab 1$ gcc print_envvars.c
[01/24/21]seed@VM:~/.../Lab 1$ a.out > file1
[01/24/21]seed@VM:~/.../Lab 1$ ls -l a.out
-rwxrwxr-x 1 seed seed 16776 Jan 24 19:51 a.out
[01/24/21]seed@VM:~/.../Lab 1$ sudo chown root a.out
[01/24/21]seed@VM:~/.../Lab 1$ sudo chmod 4755 a.out
[01/24/21]seed@VM:~/.../Lab 1$ ls -l a.out
-rwsr-xr-x 1 root seed 16776 Jan 24 19:51 a.out
```

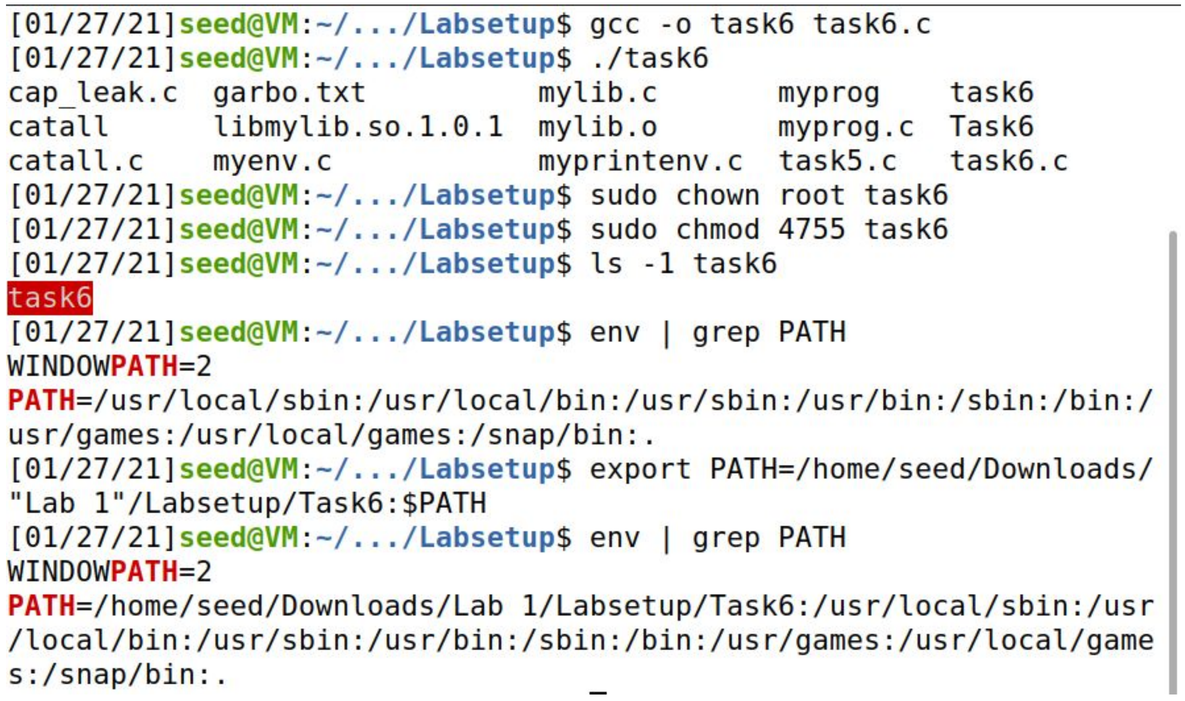
I named the given file print_envvars.c, compiled it and saved it's output to file1. I used the ls command to check the program's owner and mode before changing it to a root set-UID program using chown and chmod commands. Then I verified this change by using the ls command again, which showed the owner change from seed to root and a.out was highlighted red.



```
[01/24/21]seed@VM:~/.../Lab 1$ env | grep PATH
WINDOWPATH=2
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/u
sr/games:/usr/local/games:/snap/bin:.
[01/24/21]seed@VM:~/.../Lab 1$ export LD_LIBRARY_PATH=/home/seed
[01/24/21]seed@VM:~/.../Lab 1$ export FOO_PATH=bar
[01/24/21]seed@VM:~/.../Lab 1$ env | grep PATH
WINDOWPATH=2
FOO_PATH=bar
LD_LIBRARY_PATH=/home/seed
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/u
sr/games:/usr/local/games:/snap/bin:.
[01/24/21]seed@VM:~/.../Lab 1$ a.out > file2
[01/24/21]seed@VM:~/.../Lab 1$ diff file1 file2
33a34
> FOO_PATH=bar
```

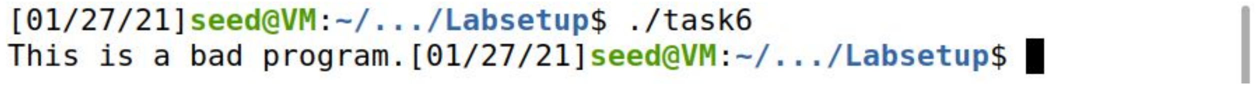
Then I used env | grep PATH to see if the environment variables PATH and LD_LIBRARY_PATH existed. PATH existed but LD_LIBRARY_PATH didn't, so I created the latter and a new variable called FOO_PATH. Then I verified that those variables were created. When I executed the program again I put the output in file2. Then I used the diff command to see the difference in environment variables between the output files. I found that all environment variables were transferred except for FOO_PATH, which was only present in the second one. This tells me that the child process does not always inherit all of the parent's environment variables.

**Task 6:**



```
[01/27/21]seed@VM:~/.../Labsetup$ gcc -o task6 task6.c
[01/27/21]seed@VM:~/.../Labsetup$ ./task6
cap_leak.c  garbo.txt          mylib.c       myprog    task6
catall      libmylib.so.1.0.1  mylib.o       myprog.c  Task6
catall.c    myenv.c            myprintenv.c  task5.c   task6.c
[01/27/21]seed@VM:~/.../Labsetup$ sudo chown root task6
[01/27/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 task6
[01/27/21]seed@VM:~/.../Labsetup$ ls -1 task6
task6
[01/27/21]seed@VM:~/.../Labsetup$ env | grep PATH
WINDOWPATH=2
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/
usr/games:/usr/local/games:/snap/bin:.
[01/27/21]seed@VM:~/.../Labsetup$ export PATH=/home/seed/Downloads/
"Lab 1"/Labsetup/Task6:$PATH
[01/27/21]seed@VM:~/.../Labsetup$ env | grep PATH
WINDOWPATH=2
PATH=/home/seed/Downloads/Lab 1/Labsetup/Task6:/usr/local/sbin:/usr
/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/game
s:/snap/bin:.
```

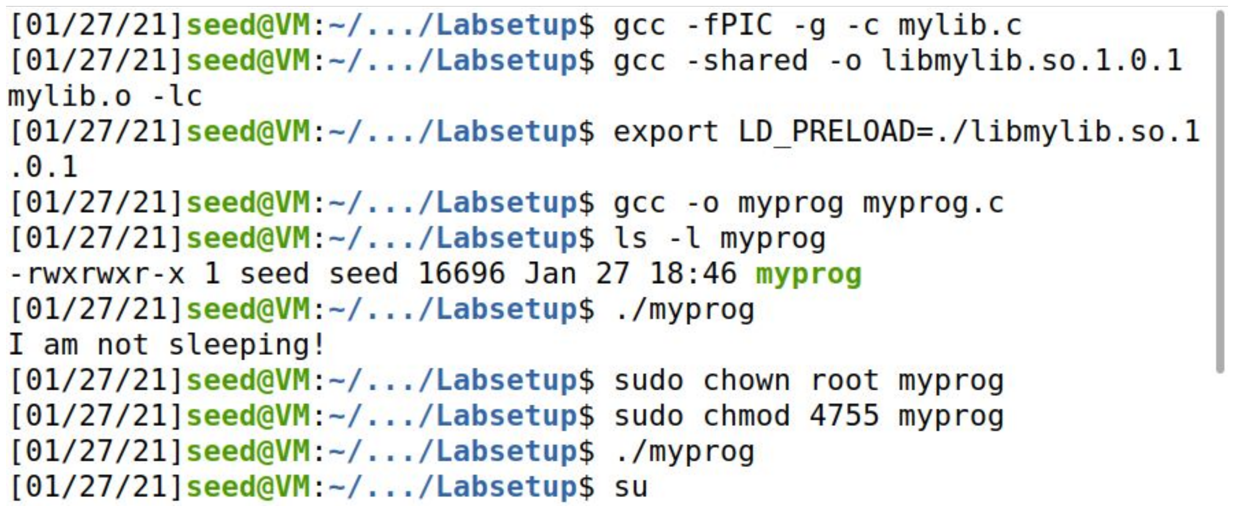
I compiled the given code in a file called task6.c. Then I changed the compiled file's privileges to be a root Set-UID program and checked to see if this was successful using ls -l. Then I used export to set the PATH variable to point to a new directory called Task6, where I stored a file I created called ls.c.



```
[01/27/21]seed@VM:~/.../Labsetup$ ./task6
This is a bad program.[01/27/21]seed@VM:~/.../Labsetup$
```

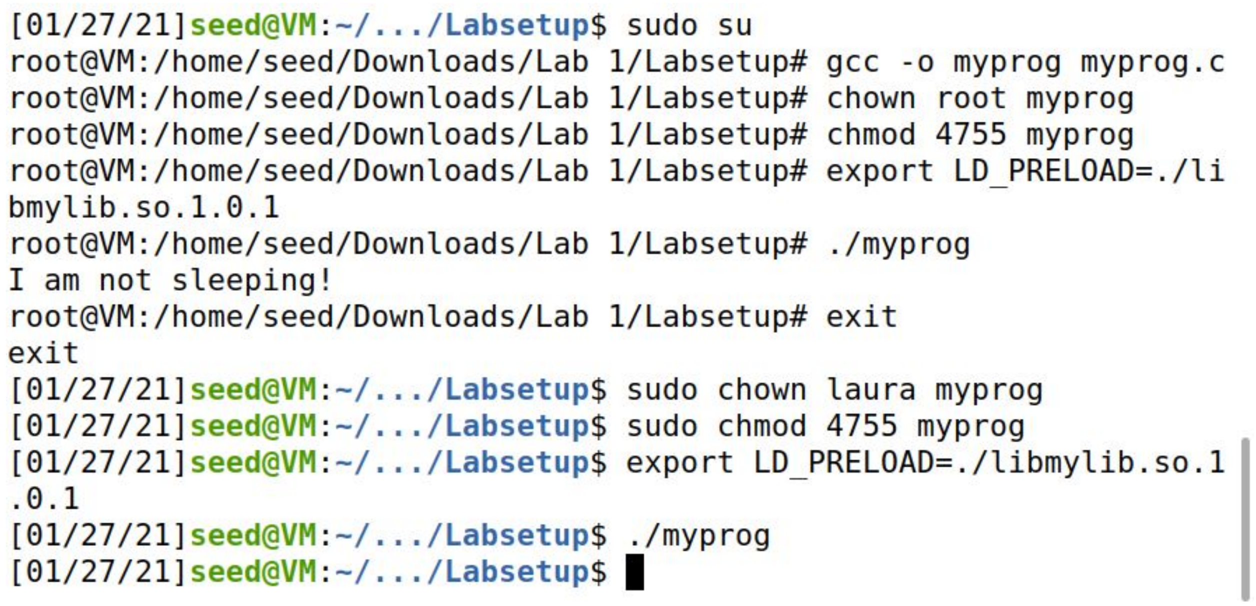
When I ran task6 again, it did whatever was stored in ls.c that I created. The implication is that because we can change the PATH variable, we can tell the program to execute something other than is expected. In this case, system() called ls, and because PATH pointed to a folder containing my file ls, system() ran that file instead of the desired ls command for bash.

**Task 7:**



```
[01/27/21]seed@VM:~/.../Labsetup$ gcc -fPIC -g -c mylib.c
[01/27/21]seed@VM:~/.../Labsetup$ gcc -shared -o libmylib.so.1.0.1
mylib.o -lc
[01/27/21]seed@VM:~/.../Labsetup$ export LD_PRELOAD=./libmylib.so.1
.0.1
[01/27/21]seed@VM:~/.../Labsetup$ gcc -o myprog myprog.c
[01/27/21]seed@VM:~/.../Labsetup$ ls -l myprog
-rwxrwxr-x 1 seed seed 16696 Jan 27 18:46 myprog
[01/27/21]seed@VM:~/.../Labsetup$ ./myprog
I am not sleeping!
[01/27/21]seed@VM:~/.../Labsetup$ sudo chown root myprog
[01/27/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 myprog
[01/27/21]seed@VM:~/.../Labsetup$ ./myprog
[01/27/21]seed@VM:~/.../Labsetup$ su
```

First I compiled mylib.c and libmylib.so.1.0.1 as instructed. Then I exported LD_PRELOAD to be a command to run libmylib.so.1.0.1. Then I compiled myprog and checked that is non-root and not a set-UID program and ran it. The output was the print statement from mylib.c, indicating that mylib.c was run. Then I changed myprog to a root set-UID program and ran it again. This time, myprog ran and the terminal slept for a second with no output.



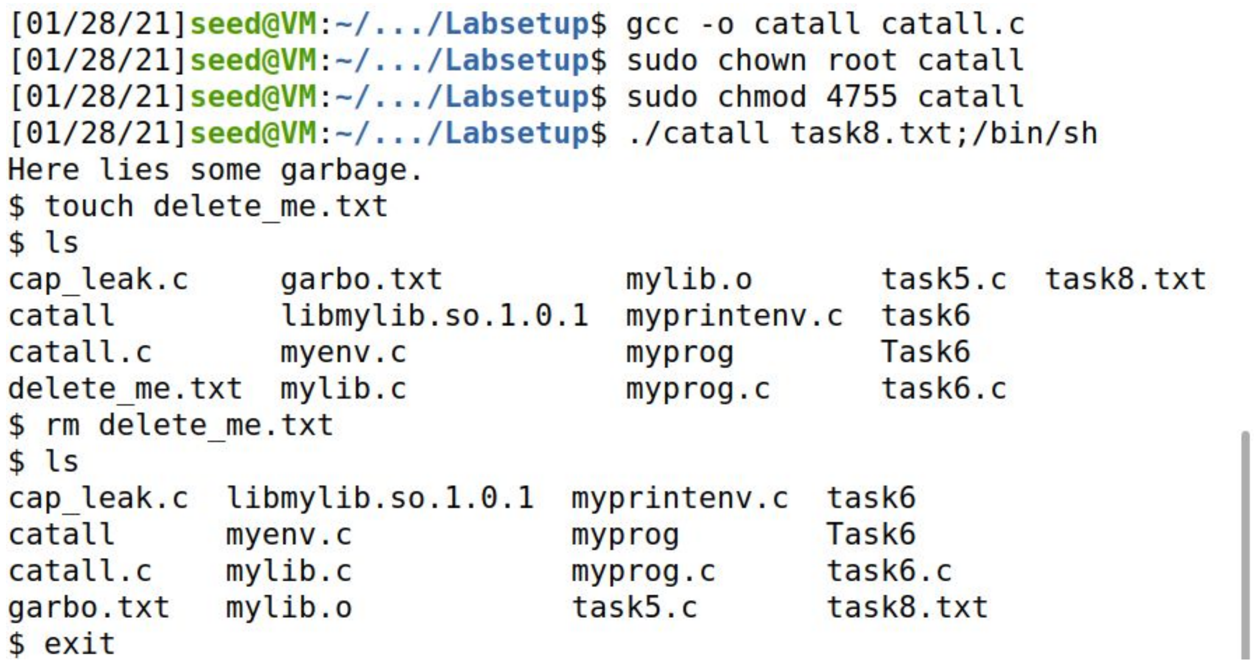
```
[01/27/21]seed@VM:~/.../Labsetup$ sudo su
root@VM:/home/seed/Downloads/Lab 1/Labsetup# gcc -o myprog myprog.c
root@VM:/home/seed/Downloads/Lab 1/Labsetup# chown root myprog
root@VM:/home/seed/Downloads/Lab 1/Labsetup# chmod 4755 myprog
root@VM:/home/seed/Downloads/Lab 1/Labsetup# export LD_PRELOAD=./li
bmylib.so.1.0.1
root@VM:/home/seed/Downloads/Lab 1/Labsetup# ./myprog
I am not sleeping!
root@VM:/home/seed/Downloads/Lab 1/Labsetup# exit
exit
[01/27/21]seed@VM:~/.../Labsetup$ sudo chown laura myprog
[01/27/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 myprog
[01/27/21]seed@VM:~/.../Labsetup$ export LD_PRELOAD=./libmylib.so.1
.0.1
[01/27/21]seed@VM:~/.../Labsetup$ ./myprog
[01/27/21]seed@VM:~/.../Labsetup$ 
```

I entered the root account, compiled myprog.c, and made it a root set-UID program. Then I exported LD_PRELOAD and ran myprog. Since I got the “I am not sleeping!” message I know mylib.c was called. Then I exited the root user account. Previously, I made a non-root user called laura. I made user laura the owner of myprog and set myprog to be a set-UID program, also setting LD_PRELOAD again and running myprog. The terminal slept for a second so I know myprog.c ran.

In order to see what may cause one program to run over the other, I added a system call to myprog.c that executed env | grep LD_PRELOAD. This would tell me which scenarios contained the LD_PRELOAD variable. Then, when I ran all the scenarios again, I noticed that in cases where LD_PRELOAD was a variable, the user and owner were the same. To confirm this, I did one extra test where I logged in as user laura and made laura the owner or myprog; the LD_PRELOAD variable was there. This tells me that in order for the LD_PRELOAD variable to be present, the effective ID and the user ID must be the same.

**Task 8:**

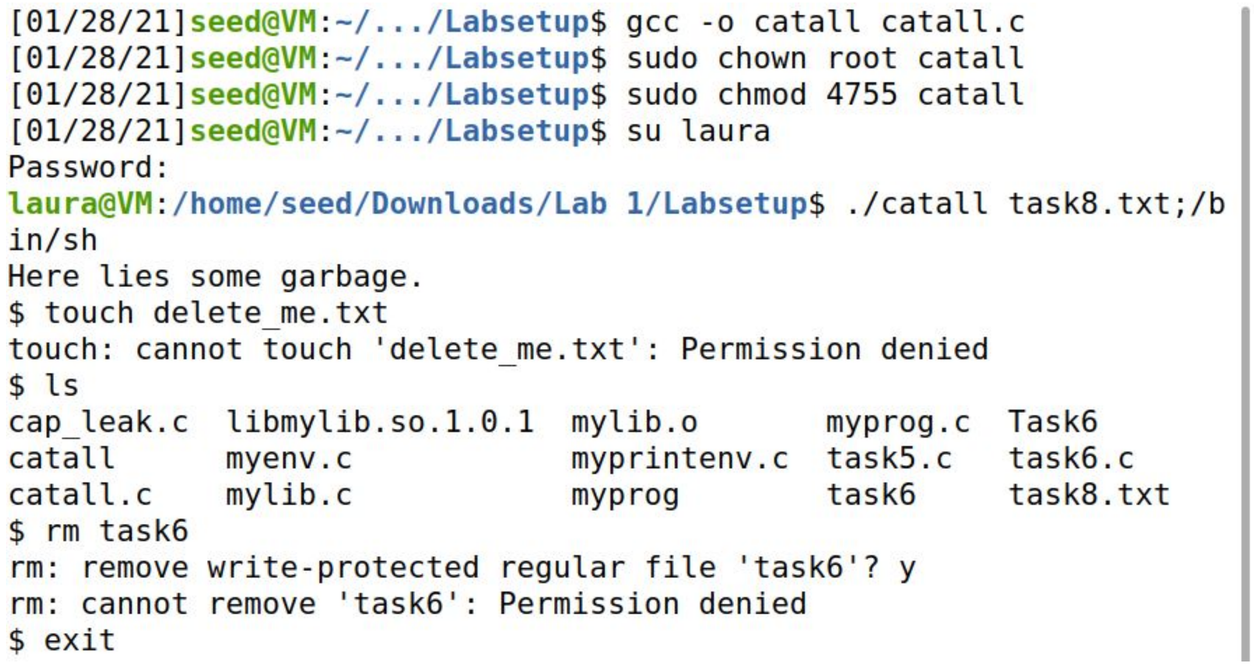
Try to rm file when using system():



```
[01/28/21]seed@VM:~/.../Labsetup$ gcc -o catall catall.c
[01/28/21]seed@VM:~/.../Labsetup$ sudo chown root catall
[01/28/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 catall
[01/28/21]seed@VM:~/.../Labsetup$ ./catall task8.txt;/bin/sh
Here lies some garbage.
$ touch delete_me.txt
$ ls
cap_leak.c     garbo.txt          mylib.o       task5.c  task8.txt
catall         libmylib.so.1.0.1  myprintenv.c  task6
catall.c       myenv.c            myprog        Task6
delete_me.txt  mylib.c            myprog.c      task6.c
$ rm delete_me.txt
$ ls
cap_leak.c  libmylib.so.1.0.1  myprintenv.c  task6
catall      myenv.c            myprog        Task6
catall.c    mylib.c            myprog.c      task6.c
garbo.txt   mylib.o            task5.c       task8.txt
$ exit
```

I compiled catall.c as catall and made it a root set-UID program. I ran catall and passed in a file, task8.txt, to print a message, and /bin/sh to have system() open the shell. From there I created a file delete_me.txt and then deleted it. The screenshot shows that when system() is used there is a vulnerability that Bob, or anyone else, could exploit to mess around with the files when they're not supposed to have the privilege to do so, such as creating and removing files.

Try to rm file when using execve():



```
[01/28/21]seed@VM:~/.../Labsetup$ gcc -o catall catall.c
[01/28/21]seed@VM:~/.../Labsetup$ sudo chown root catall
[01/28/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 catall
[01/28/21]seed@VM:~/.../Labsetup$ su laura
Password:
laura@VM:/home/seed/Downloads/Lab 1/Labsetup$ ./catall task8.txt;/b
in/sh
Here lies some garbage.
$ touch delete_me.txt
touch: cannot touch 'delete_me.txt': Permission denied
$ ls
cap_leak.c  libmylib.so.1.0.1  mylib.o       myprog.c  Task6
catall      myenv.c            myprintenv.c  task5.c   task6.c
catall.c    mylib.c            myprog        task6     task8.txt
$ rm task6
rm: remove write-protected regular file 'task6'? y
rm: cannot remove 'task6': Permission denied
$ exit
```

I commented out the system() call and instead used execve() in catall.c, then recompiled and made it a root set-UID program again. Then, as a random user, laura, I executed catall and passed in the same task8.txt file and /bin/sh. In this case, execve() opened a shell but I could not create or delete any files. This demonstrates how execve() is more secure as we couldn't exploit the same vulnerability as we did when catall.c used system().