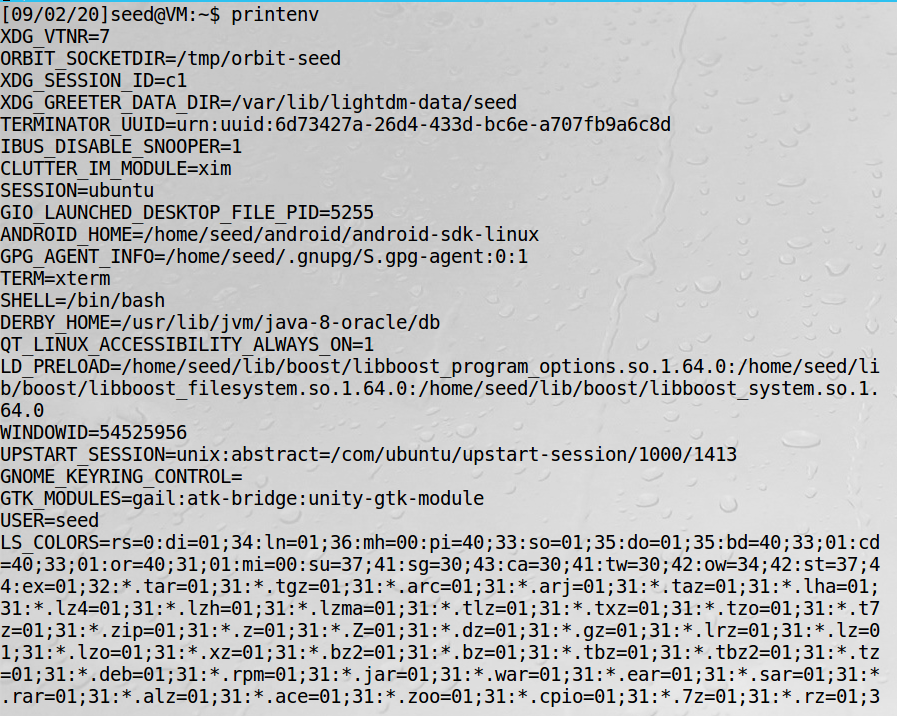
**Lab1-report**

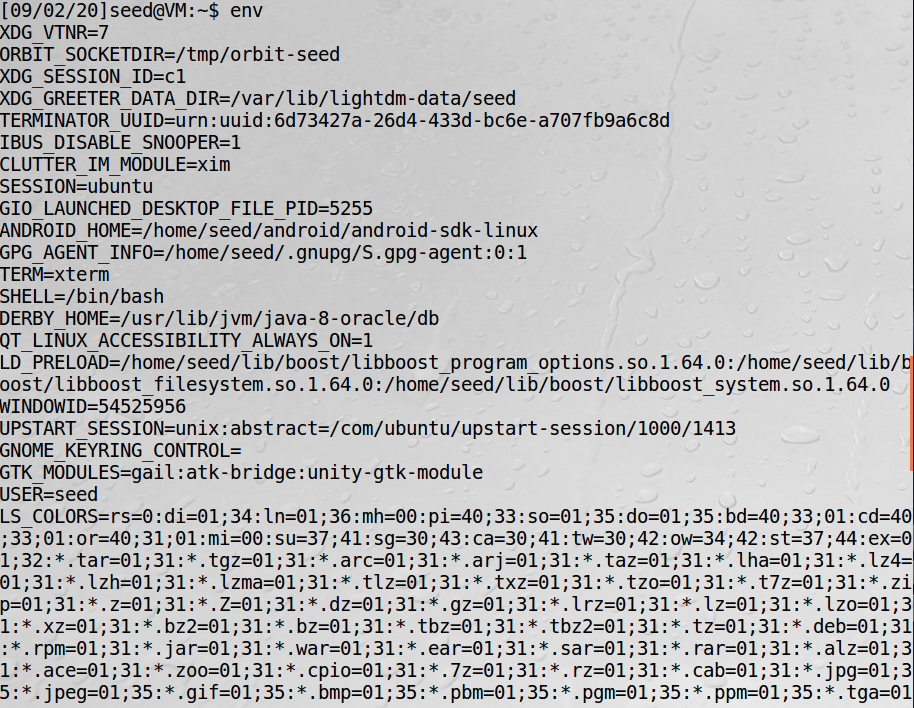
**Task 1：Manipulating Environment Variables**

**（1）用printenv和env打印出环境变量**

**Use printenv**

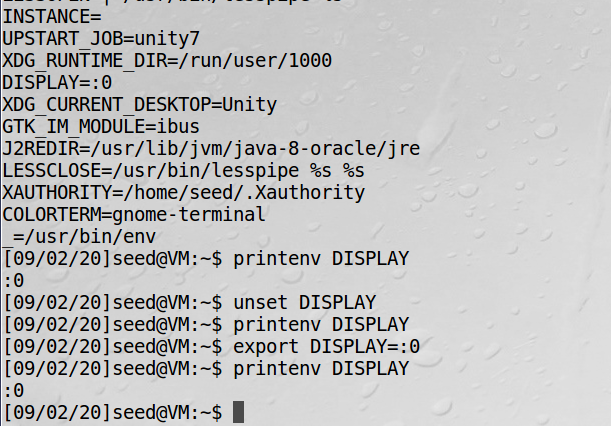


**Use env**

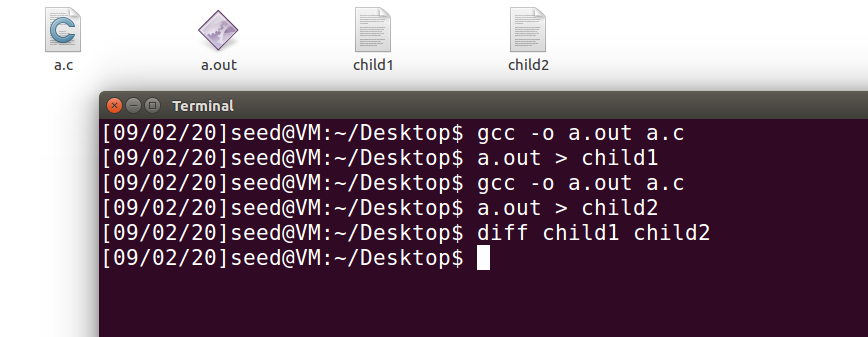


printenv与env都是打印系统当前的环境变量，不同的是printenv不加参数和env相同，printenv可以打印指定名称的环境变量。

**（2）使用export和unset命令和删除环境变量**



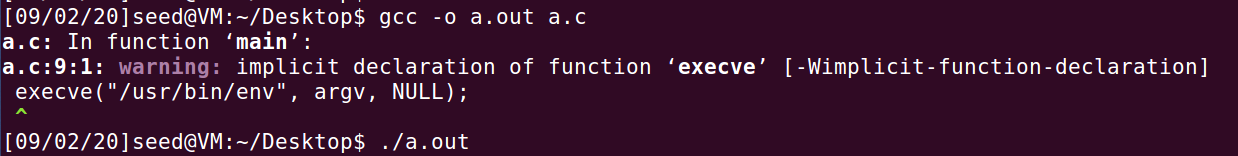
**Task 2：Passing Environment Variables from Parent Process to Child Process**



通过比较这两个文件，可以发现，这两个文件输出的环境变量完全相同。说明原环境变量被子进程完全继承。考虑进程的虚拟地址空间，命令行参数和环境变量在栈之上，子进程以父进程为模板，拷贝PCB、虚拟地址空间。所谓子进程，就是在我目前的这个shell下，去打开另一个新的shell，新的那个shell就是子进程，一般的状态下，父进程的自定义变量是无法在子进程内使用的，但是通过export将变量变成环境变量后，就能在子进程下面用了。

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

**（1）**重新保存和编译文件，发现执行结果为空



查询函数execve()的作用，其调用格式如下：  
int execve(const char \* filename，char \* const argv[]，char \* const envp[])  
第一个参数为一个可执行的有效的路径名。第二个参数系利用数组指针来传递给执行文件，argv是要调用的程序执行的参数序列，也就是我们要调用的程序需要传入的参数。  
envp则为传递给执行文件的新环境变量数。  
所以在此处，我们赋予新进程的环境变量为空，自然印出环境变量结果为空。

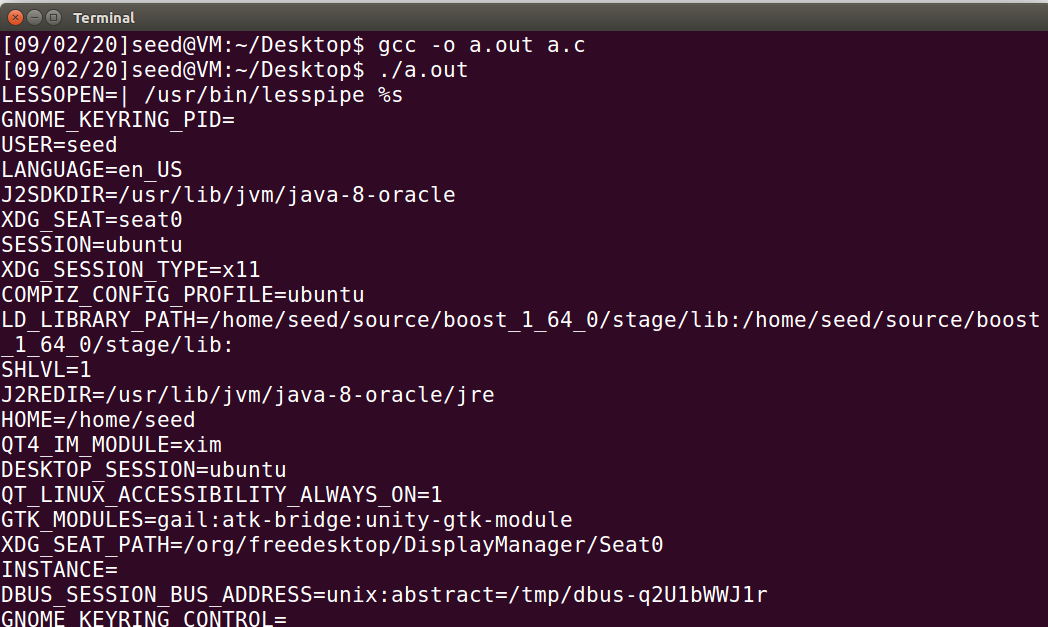
**（2）**参数加上环境变量后，重新保存和编译文件，得到如下结果



**（3）**

从以上实验可以看出，execve()产生的新进程的环境变量又调用时重新赋予，而fork()则是直接继承父进程环境变量。

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



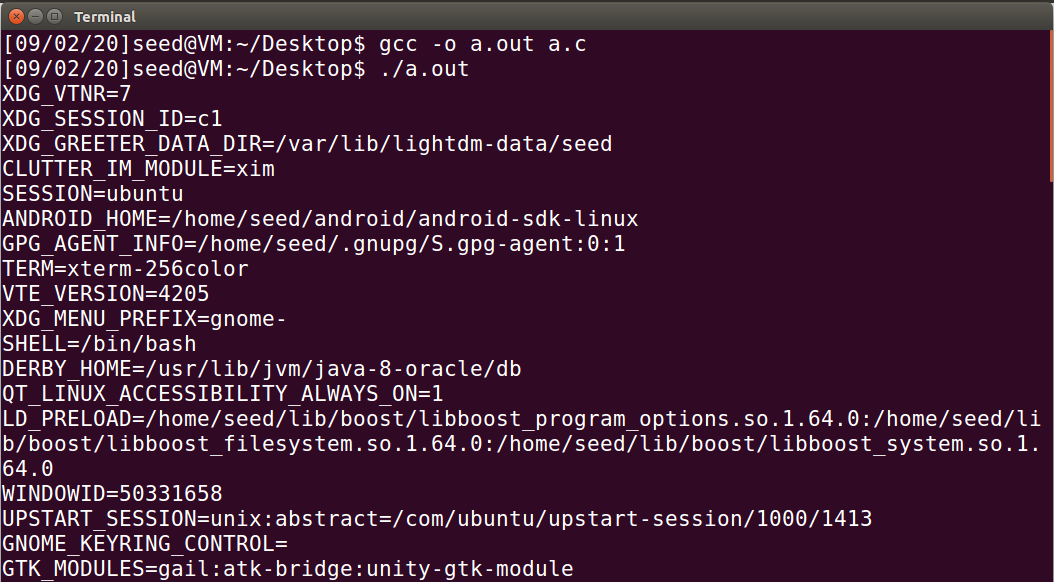
先看一下system（）函数的简单介绍，int system（const char \* string）。

system函数定义为 int system（const char \* string），该函数调用/bin/sh来执行参数指定的命令,/bin/sh一般是一个软连接，指向某个具体的shell，比如bash,-c 选项是告诉shell从字符串command中读取命令。system（）函数的函数返回值有些复杂。为了更好地理解system（）函数的返回值，需要了解其执行过程，实际上system（）函数执行了三步操作：

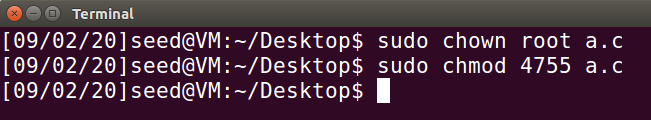
1.fork一个子进程；  
2.在子进程中调用exec函数去执行command；  
3.在父进程中调用wait去等待子进程结束。

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

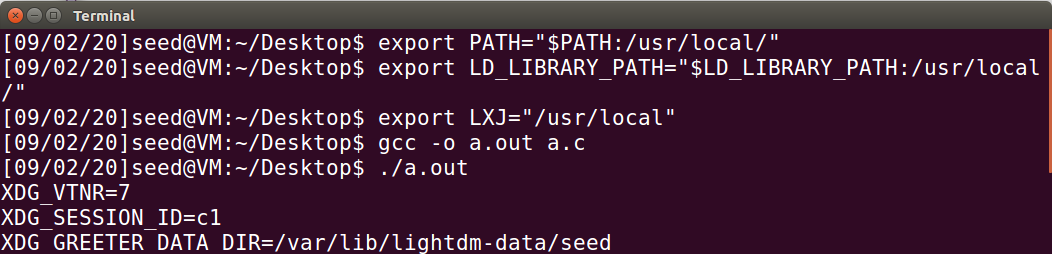
Step1



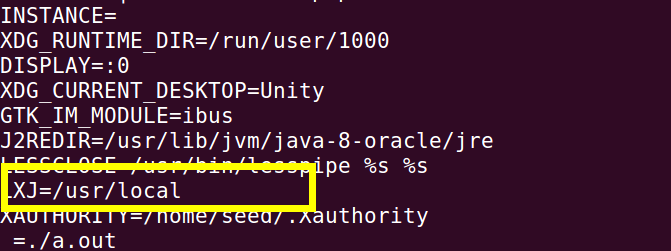
Step2



Step3

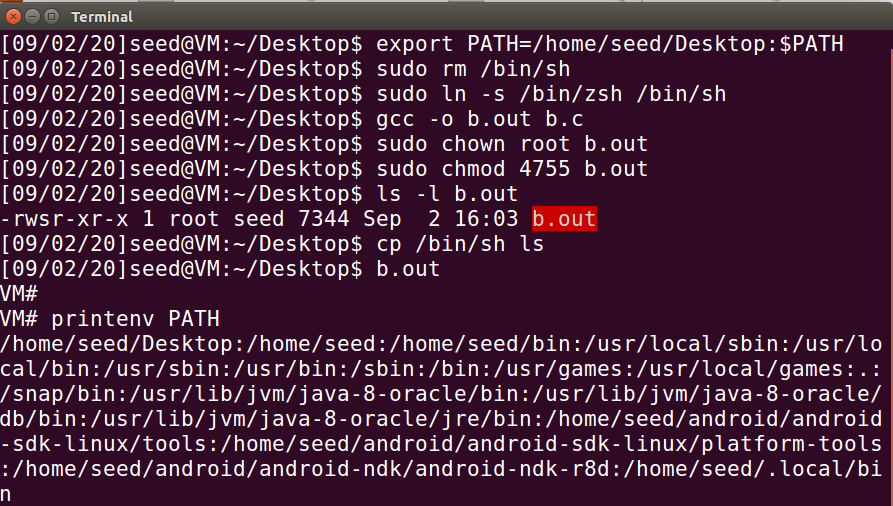






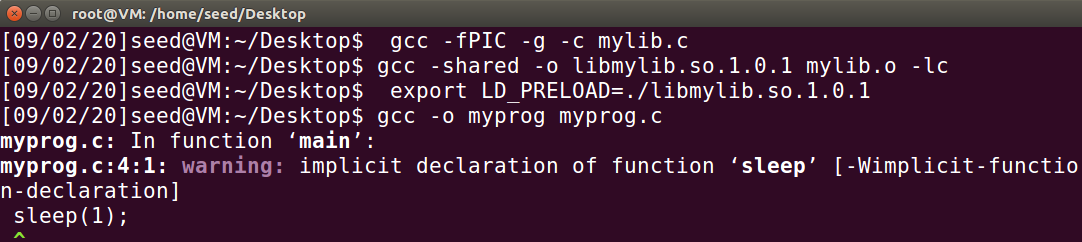
可以看到，以上三个被定义的环境变量全部被包括在shell中。

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



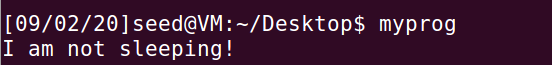
将bin/sh复制到当前目录并命名为ls，执行demo6就会获得root权限。详细分析，先看一下PATH环境变量，它的命令找寻顺序是先找寻当前目录，而当前目录我们自己编造了一个ls，所以程序就会直接执行伪造的ls。sh原本的作用是创建一个新shell，在执行此命令后我们就会一直停留在子进程中，知道我们主动退出这个程序，我们才会回到原来的权限。

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

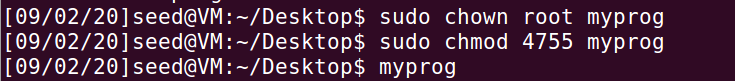


gcc -fPIC -g -c mylib.c #fPIC表示编译生成代码与位置无关

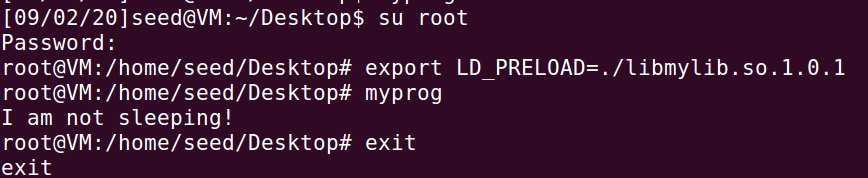
gcc -shared -o libmylib.so.1.0.1 mylib.o -lc #让编译器知道是要编译一个共享库



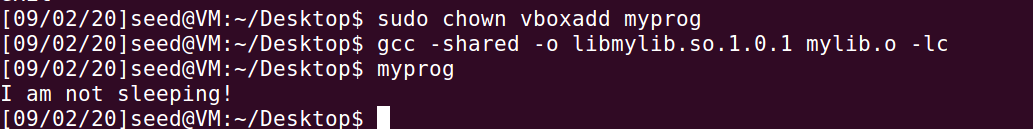
Make myprog a regular program, and run it as a normal user



Make myprog a Set-UID root program, and run it as a normal user



Make myprog a Set-UID root program, export the LD PRELOAD environment variable again in the root account and run it



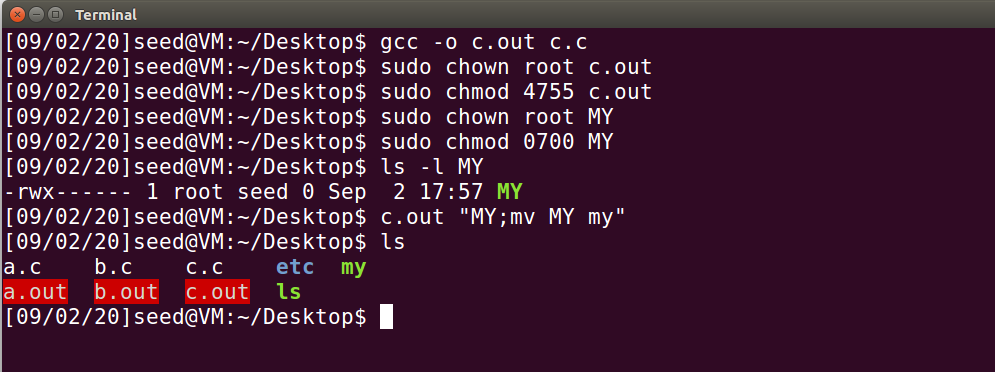
Make myprog a Set-UID user1 program (i.e., the owner is user1, which is another user account), export the LD PRELOAD environment variable again in a different user’s account (not-root user) and run it

1、3、4输出“I am not sleeping!”，2无输出

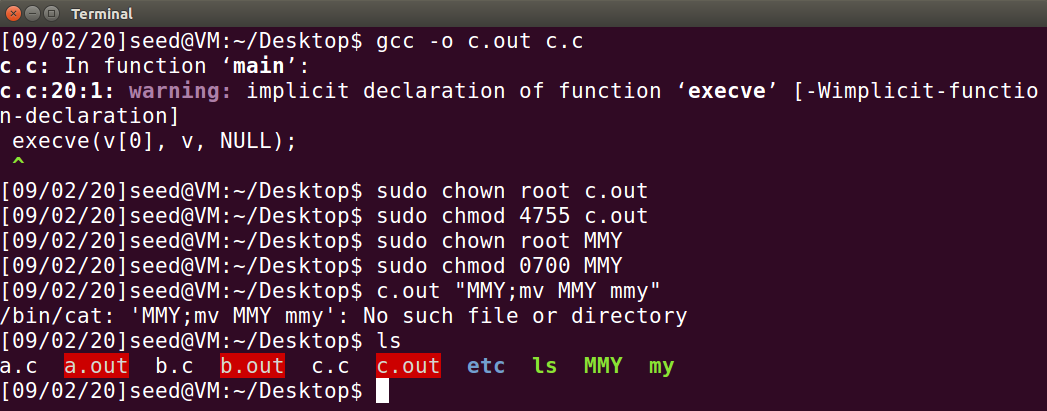
导致他们不同的原因就在于LD\_PRELOAD环境变量。LD\_PRELOAD环境变量是Unix动态链接库的世界中的一个环境变量，它可以影响程序的运行时的链接，它允许你定义在程序运行前优先加载的动态链接库。这个功能主要是用来有选择性的载入不同动态链接库中的相同函数。在该实验中，mylib.c通过sleep函数，生成了一个libmylib.so.1.0.1链接库。然后将该链接库添加到LD\_PRELOAD环境变量上。

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

**(1)使用system()函数下可以更改删除写入的文件**

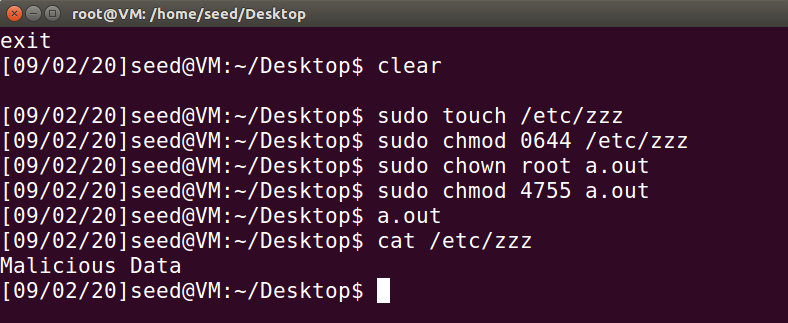


**(2) 使用execve()函数下不可以更改删除写入的文件**



分析其原因，是因为execve()函数会把file; mv file file\_new 看成是一个文件名，系统会提示不存在这个文件，system()则不会。

**Task 9：Capability Leaking**



分析其原因是因为其在取消权限前并没有关闭文件，导致seed用户任然可以进行root用户才可以执行的写入操作。