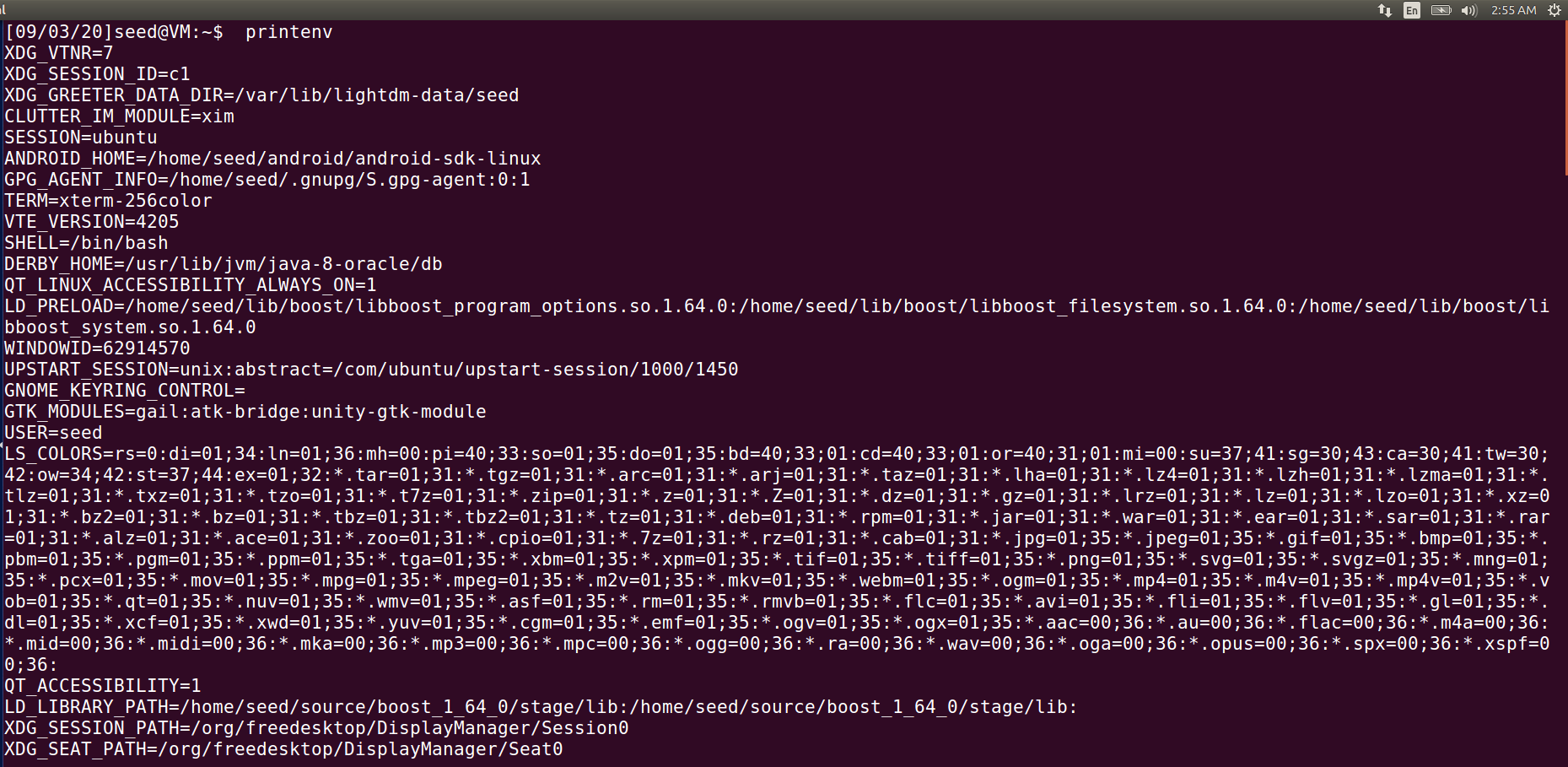
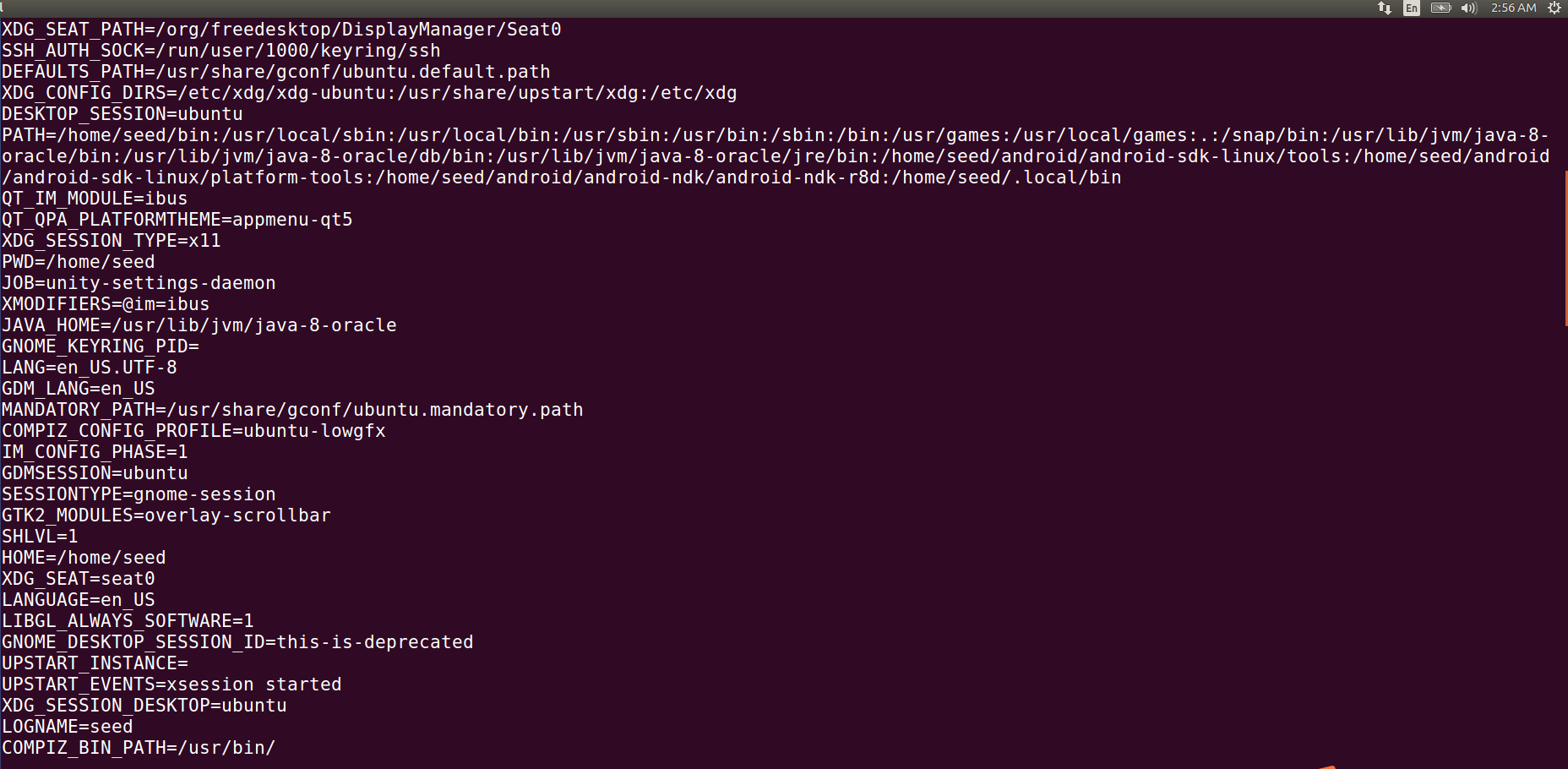
**LAB1-Report**

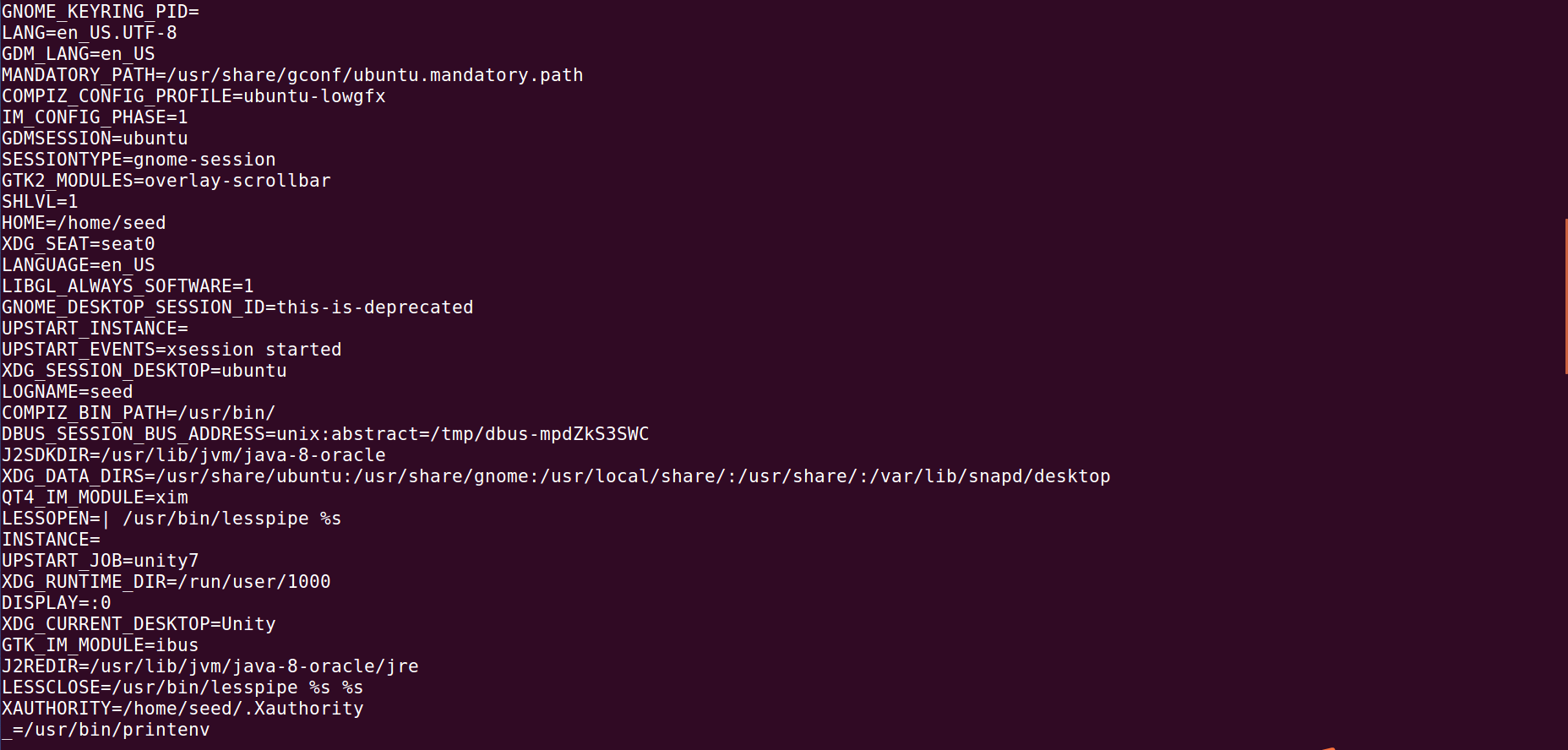
57118103 郭欣然

Task1：Manipulating Environment Variables

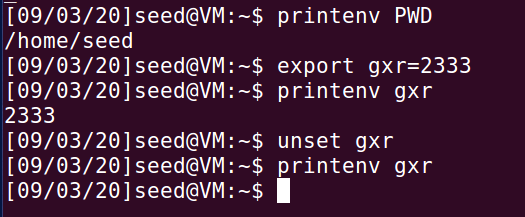
1、使用prinitenv打印环境变量







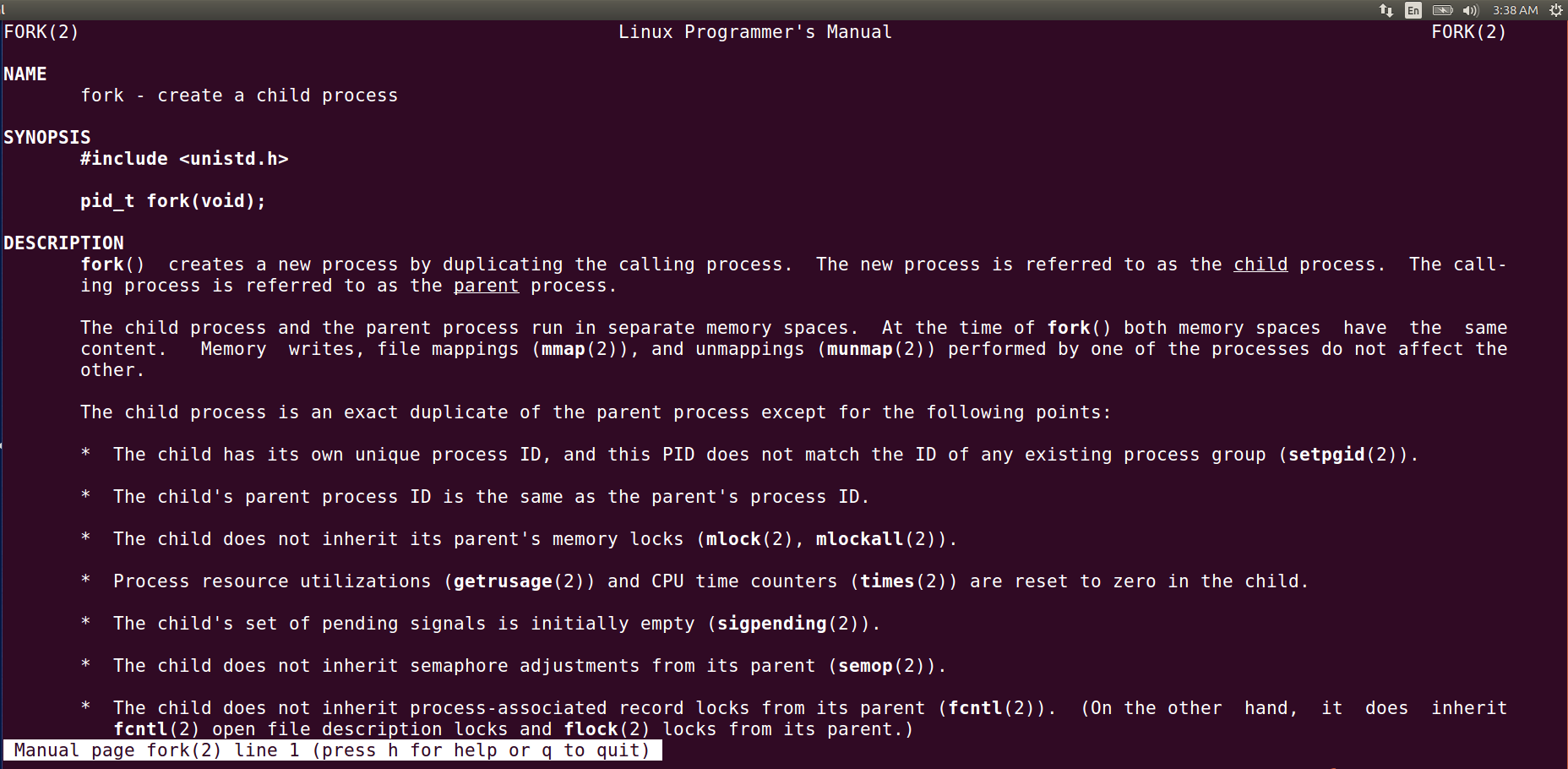
2、使用printenv打印特定环境变量PWD、设定并删除新的环境变量



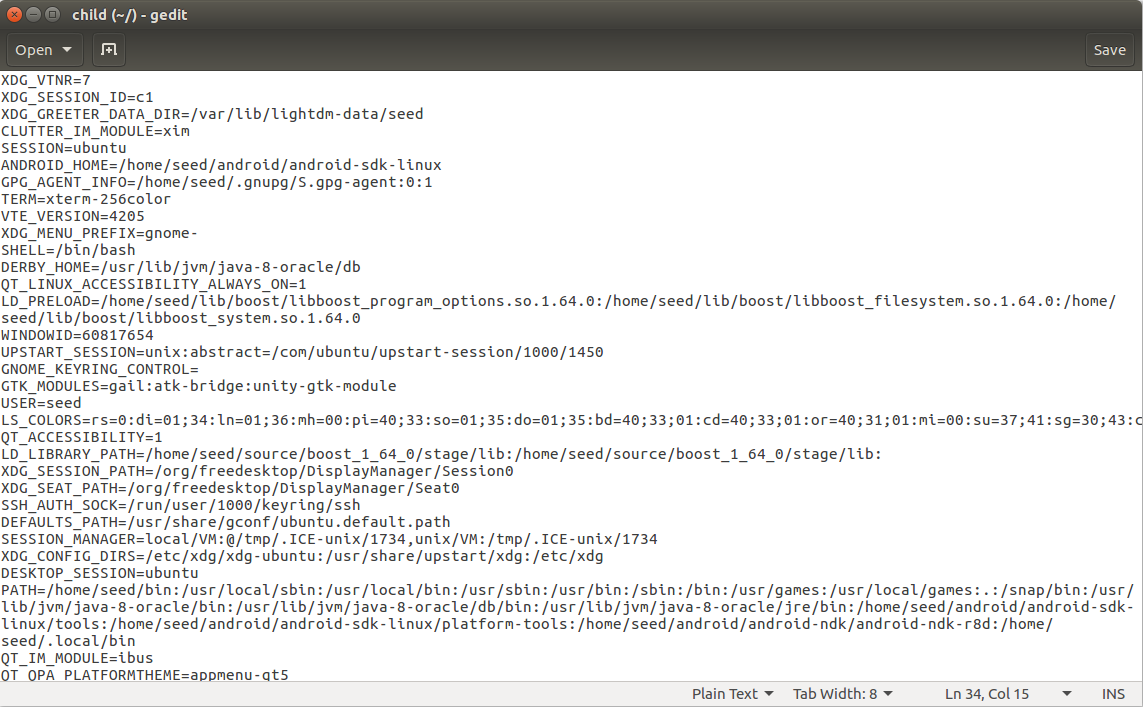
设定新的环境变量gxr=2333，打印出，最终删除变量

Task2：Passing Environment Variables from Parent Process to Child Process

1、Man fork查看fork manual

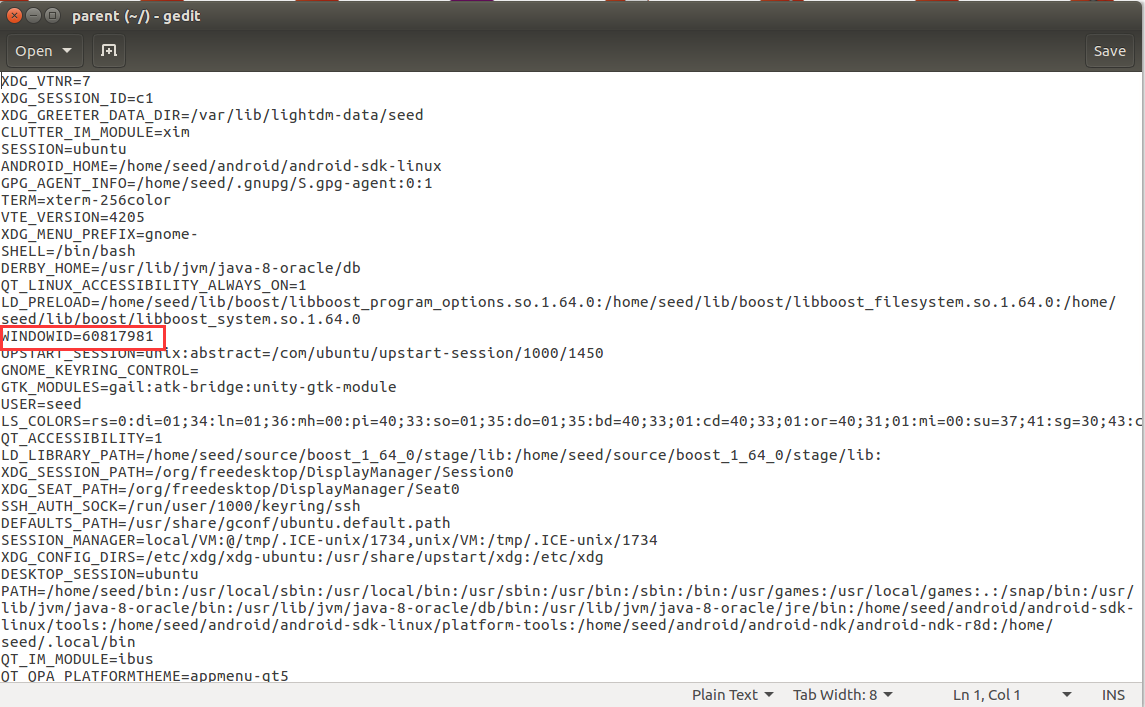


2、编译，设定可执行文件为a.out，运行程序并将结果输出到child文件中



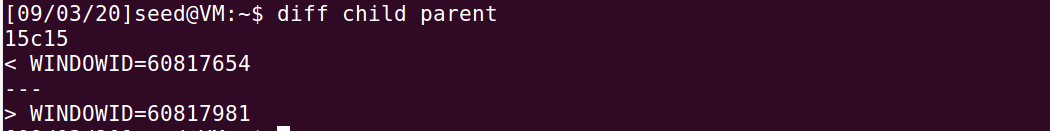
可以看到只有子进程执行了printenv（），输出了环境变量

3、将子进程中printenv（）注释，将父进程的printenv（）取消注释，进行编译与运行，并将结果输出到parent文件中



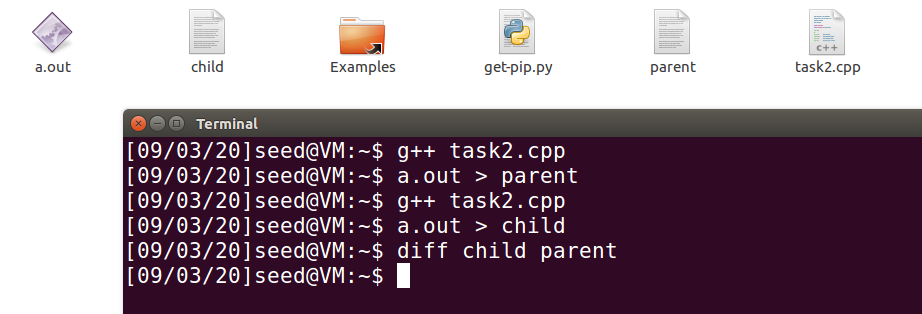
发现父进程执行了printenv（），输出了环境变量。

4、用diff命令比较两次的输出结果



发现只有WINDOWID不同，说明父进程在向子进程传递环境变量使除了WINDOWID其他全部相同。WINDOWID不同是因为两次运行处于两个终端中。下面将两次运行放在一个终端中进行。

5、



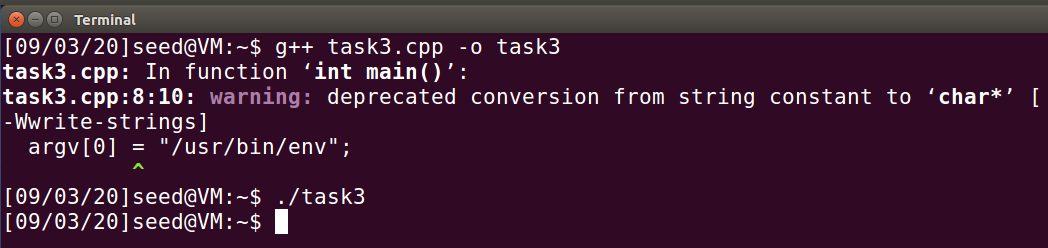
经过比较，父进程与子进程的环境变量完全相同，说明fork（）出的子进程继承了父进程的所有环境变量。

父进程调用fork()创建子进程，子进程成为父进程的副本，复制父进程的所有进程内存到其内存地址空间中。父、子进程的“数据段”，“堆栈段”和“代码段”完全相同，即子进程中的每一个字节都和父进程一样。fork之后，从 fork 处开始父子进程并行执行，公有变量的值也在 fork 处被复制（当时的值），只不过父子进程分别使用各自的复本（而非共享）。

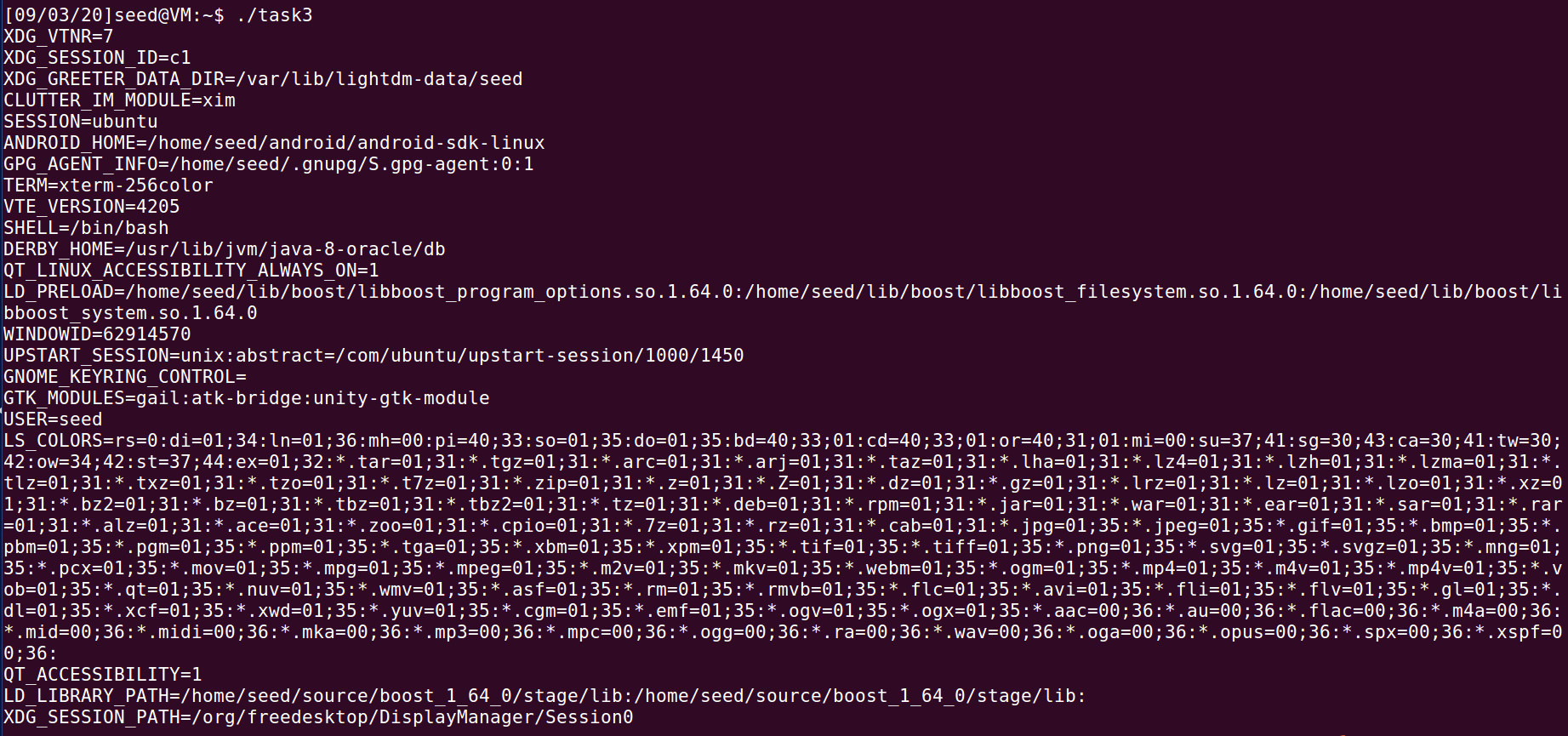
在一般的状态下，父程序的自定义变量是无法在子程序内使用的。但是透过export将变量变成环境变量后，就能够在子程序底下应用了。

Task3：Environment Variables and execve()

1、原文件没有结果输出



2、将NULL改为environ后，程序打印出了当前所有的环境变量



对比可以看出第三个参数为空时没有结果输出，即没有传环境变量。因此环境变量通过第三个参数传递。经过查阅：

**函数定义：**

int execve(const char \*filename, char \*const argv[ ], char \*const envp[ ]);

**返回值：**

函数执行成功时没有返回值，执行失败时的返回值为-1.

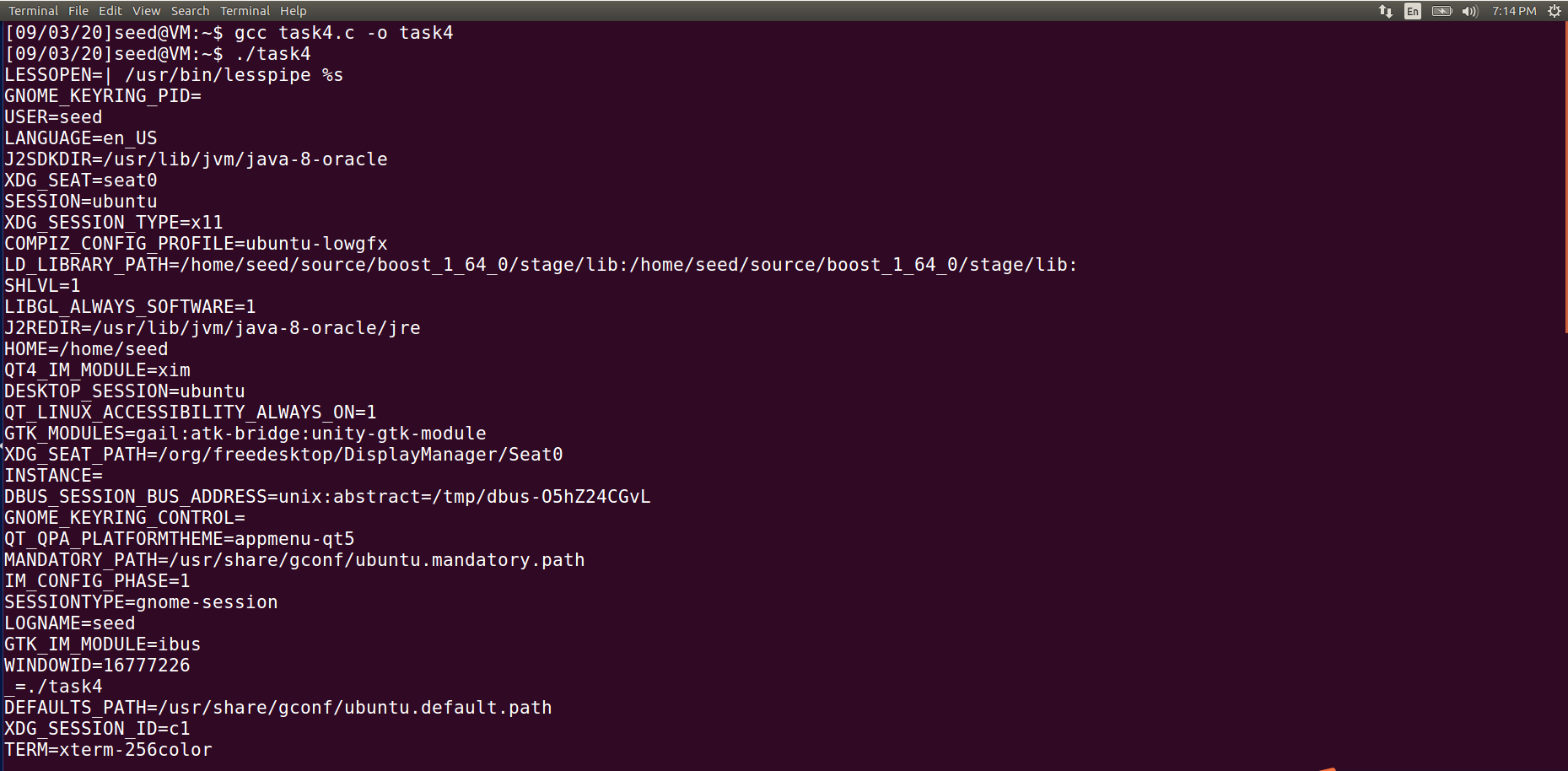
**函数说明：**

execve()用来执行参数filename字符串所代表的文件路径，第二个参数是利用数组指针来传递给执行文件，并且需要以空指针(NULL)结束，最后一个参数则为传递给执行文件的新环境变量数组。

最后一个参数为传递给执行文件的新环境变量数组，原文件中将其设置为空因此没有结果输出。

通过两个实验对比发现，execve()产生的新进程的环境变量会在该函数中定义，fork()则是直接继承父进程的环境变量。

Task4：Environment Variables and system()

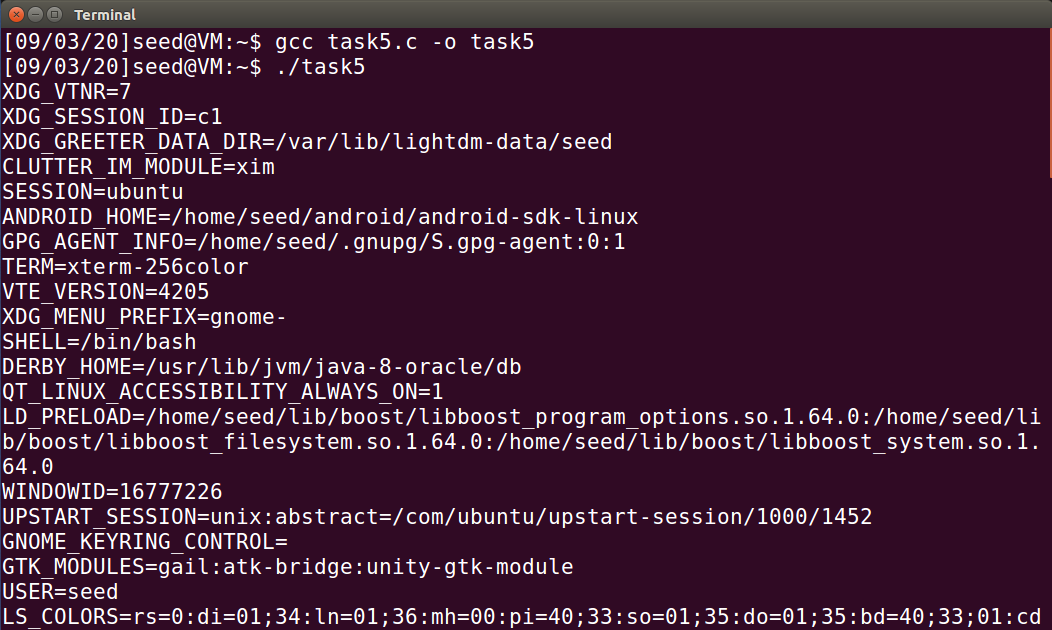


**定义函数：**int system(const char \* string);   
**函数说明：**  
执行shell命令，参数字符串command为命令名。在Linux/Unix系统中，system函数会调用fork函数产生子进程，子进程中调用exec函数来执行command命令，命令执行完后随即返回原调用的进程。

因此子进程会继承父进程的环境变量，程序执行结果将打印出环境变量。

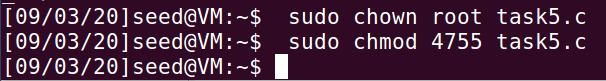
Task 5：Environment Variable and Set-UID Programs

1、运行原程序

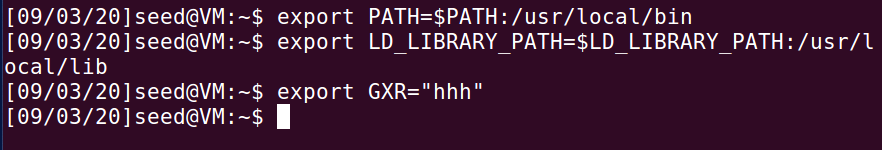


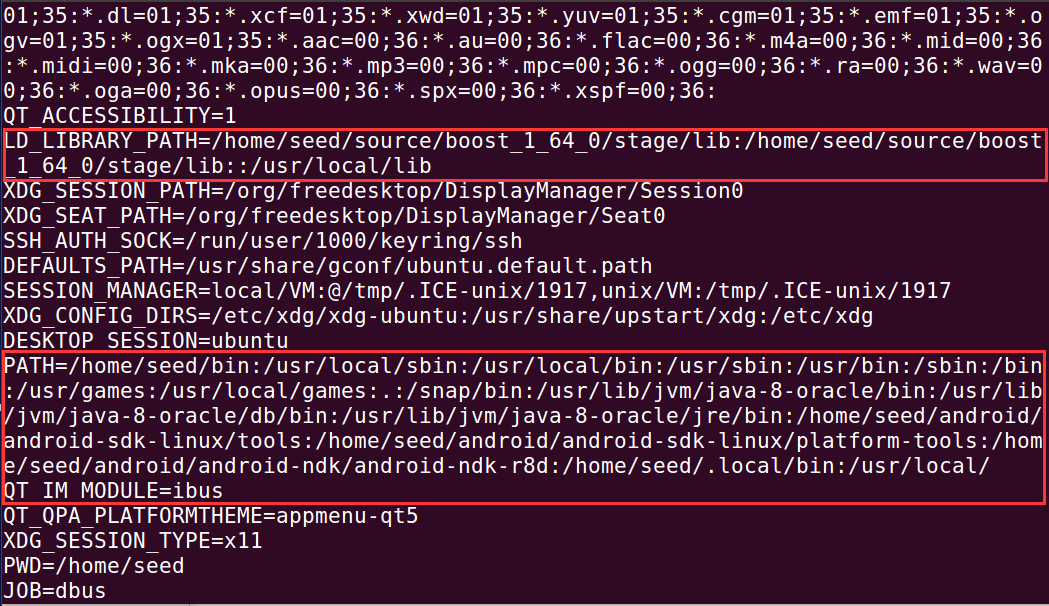
打印出了环境变量

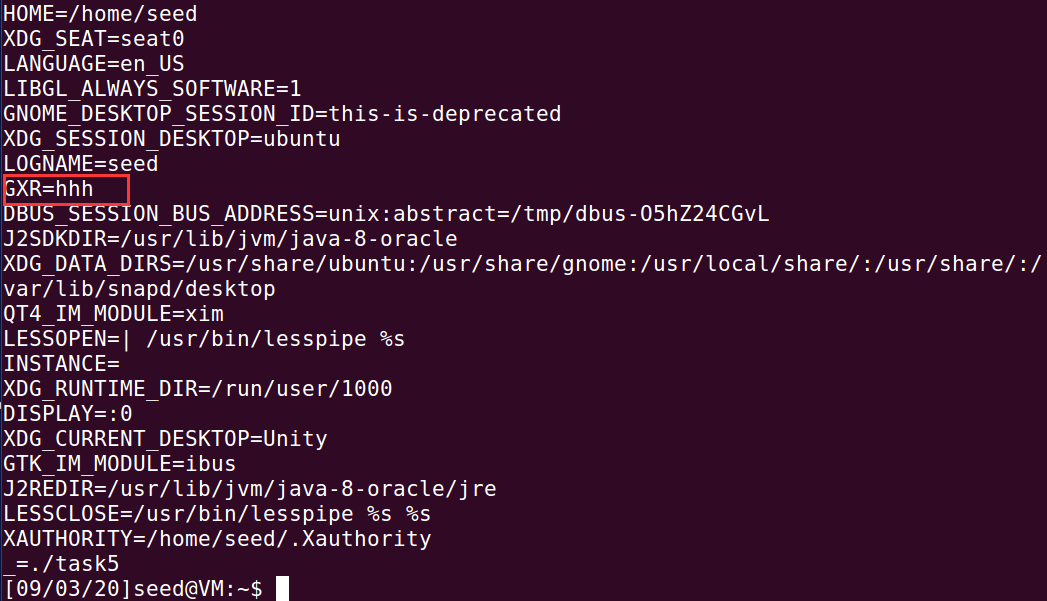
2、修改为set UID程序



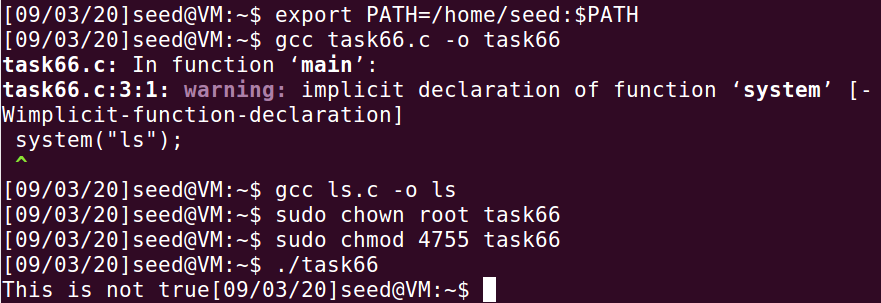
3、







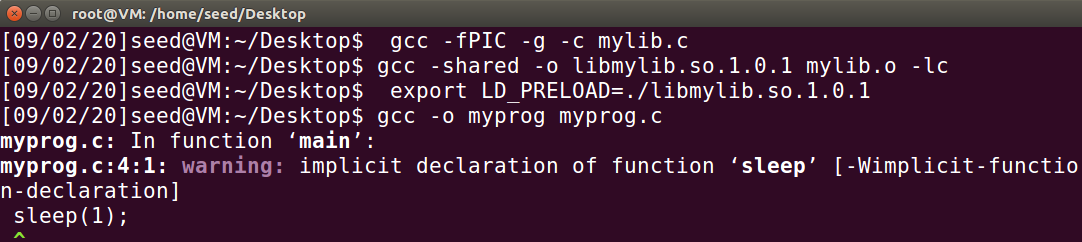
Task 6：The PATH Environment Variable and Set-UID Programs



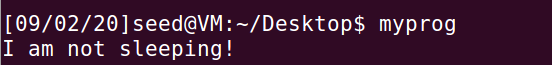
在当前目录创建ls文件，其中输出“This is not true”，程序改为set-UID程序获得root权限。PATH环境变量寻找顺序是先找寻当前目录，在当前目录我们自己编造了一个ls，所以程序就会直接执行伪造的ls。

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

1、编译一个共享库

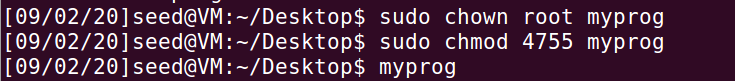


2、Make myprog a regular program, and run it as a normal user



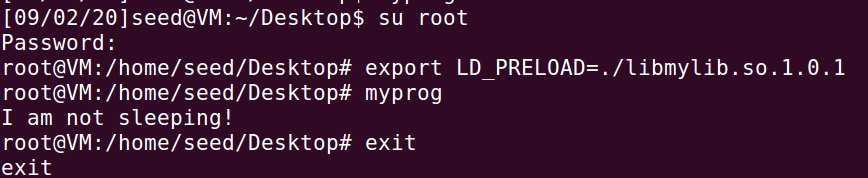
以普通用户的身份运行myprog程序输出I am not sleeping!

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



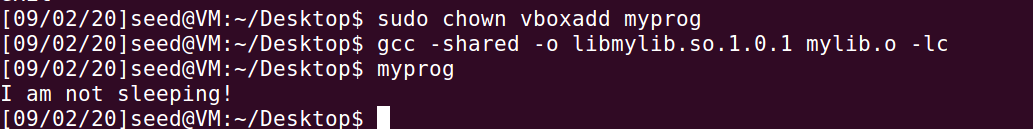
以普通用户运行拥有root权限的myprog程序没有输出。

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



以root权限运行程序并将LD PRELOAD环境变量添加到root中，输出I am not sleeping!

5、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



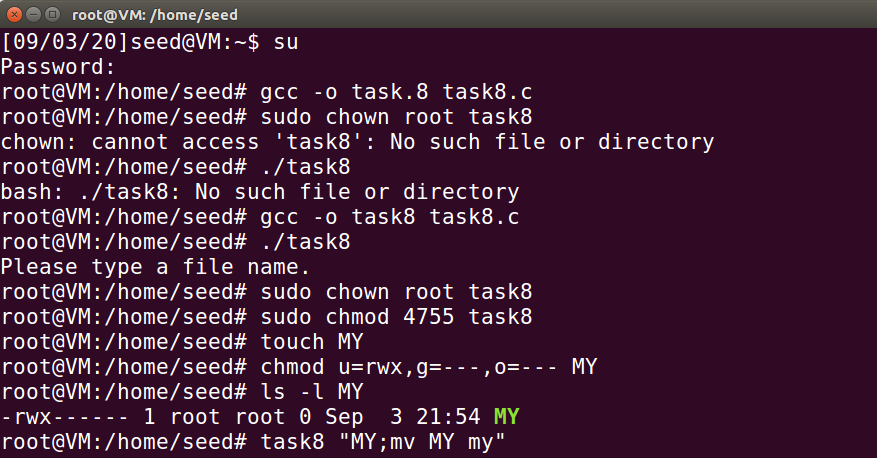
在user中再次设置LD\_PRELOAD环境变量并运行myprog程序输出I am not sleeping!

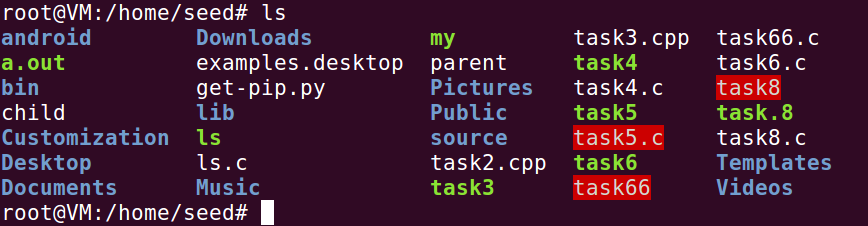
6、分析：

起决定作用的因素为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，允许定义在程序运行前优先加载的动态链接库。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。因此当以seed用户权限运行时（即1、5），有结果输出。以root权限运行时（即3），没有结果输出。在root中导入LD\_PRELOAD环境变量（即4），并以root权限运行，有结果输出。

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

1、使用system()函数可以更改删除写入的文件

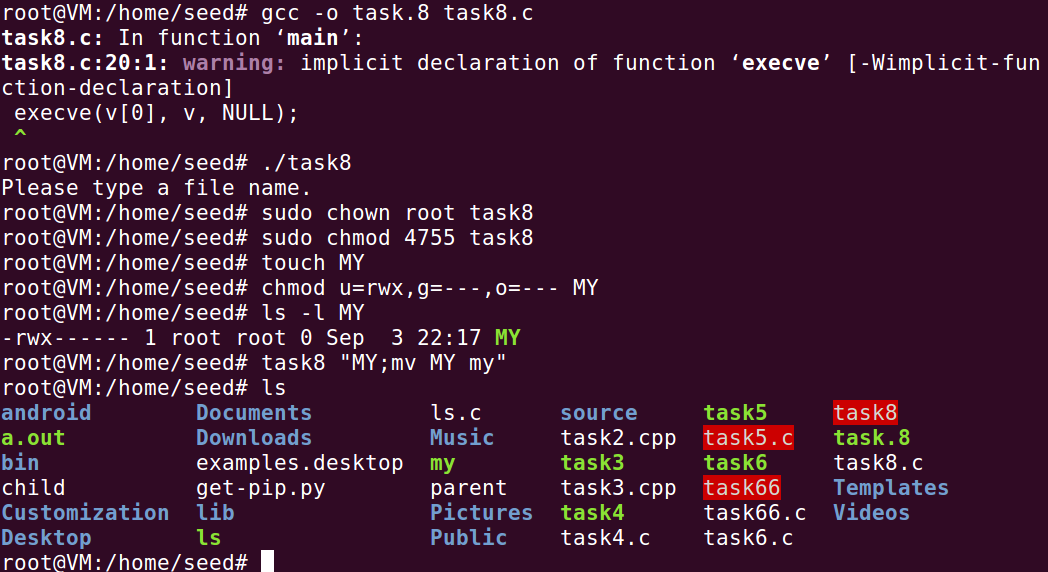


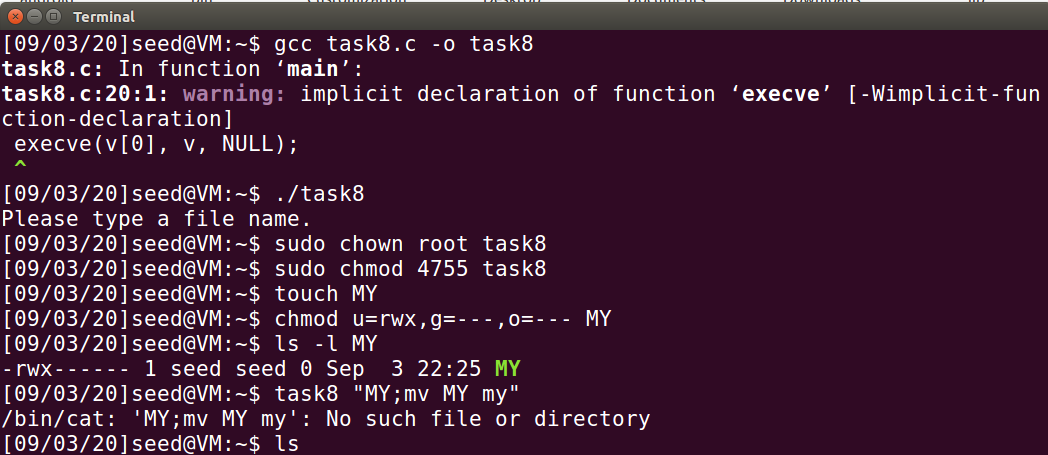


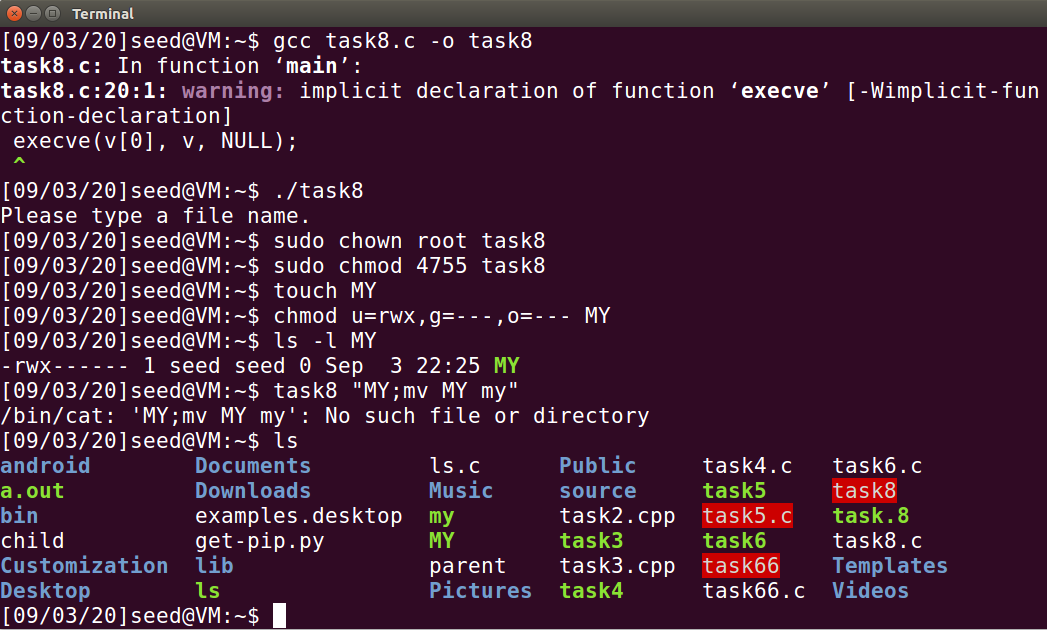
保存代码为task.c文件，切换为root用户，将其编译为task8，并设置其所有者为root，赋予set-UID特殊权限。

新建一个名为MY的文件，并设置其权限为仅root用户可读、写、执行。执行task8，发现原本只有root用户才具有读、写、执行的MY文件，已经更名为my

2、使用execve()函数不可以更改删除写入的文件



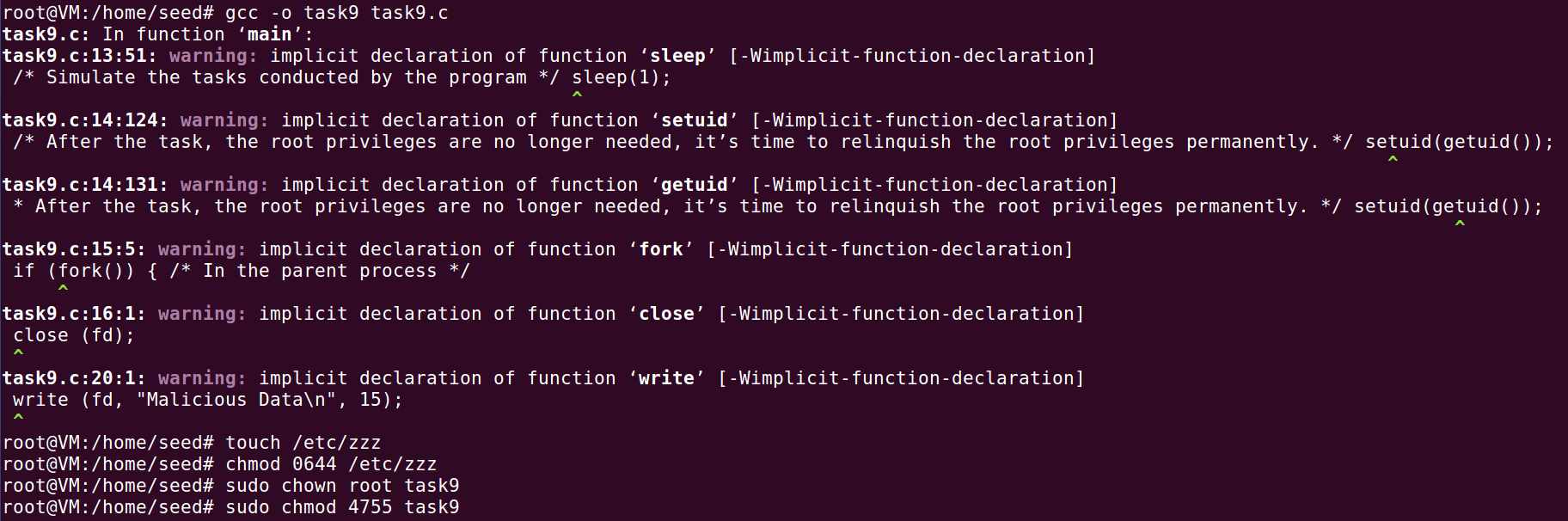




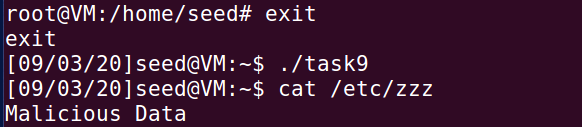
分析原因，是因为execve()函数会把file; mv file file\_new 看成是一个文件名，系统会提示不存在这个文件，system()则不会。

Task 9：Capability Leaking

切换为root用户，创建/etc/zzz，设置其为只有root用户可以写入。保存代码为task9.c文件，将其编译为task9，并设置为set-UID程序。



执行demo9，查看内容，发现被写入信息。



这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。