Environment Variable and Set-UID Program Lab

57119133 张明璇

Task1

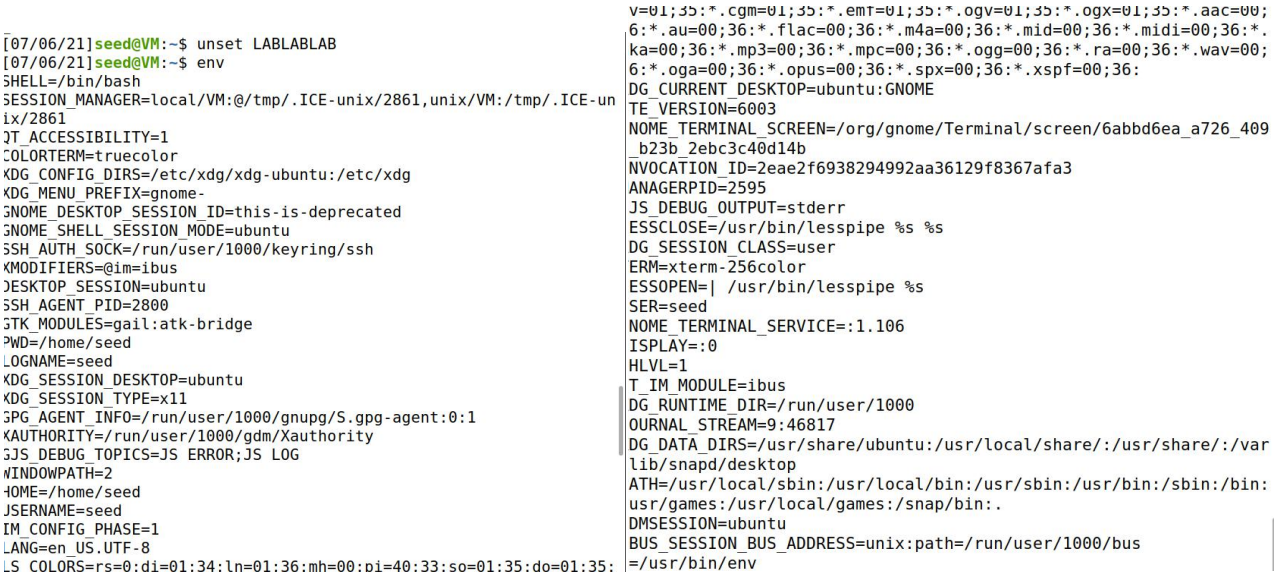
实验内容：Manipulating Environment Variables

In this task, we study the commands that can be used to set and unset environment variables. We are using Bash in the seed account. The default shell that a user uses is set in the /etc/passwd file (the last field of each entry). You can change this to another shell program using the command chsh (please do not do it for this lab). Please do the following tasks：

Use printenv or env command to print out the environment variables. If you are interested in some particular environment variables, such as PWD, you can use "printenv PWD" or "env grep PWD".

Use export and unset to set or unset environment variables. It should be noted that these two commands are not seperate programs; they are two of the Bash’s internal commands (you will not be able to find them outside of Bash).

实验结果及分析：



由此看见，这里LABLABLAB被删除

经过实验发现，printenv 和 env 均可输出当前系统的环境变量。不同的是 printenv 不加参数和 env 一样，而 printenv 可以打印指定名称的环境变量。

```
[07/06/21]seed@VM:~$ printenv | grep Myname
Myname=jack
[07/06/21]seed@VM:~$ unset Myname
[07/06/21]seed@VM:~$ printenv | grep Myname
[07/06/21]seed@VM:~$
```

使用 export 或者 unset 命令设置或去掉环境变量

Task2

实验内容：Passing Environment Variables from Parent Process to Child Process

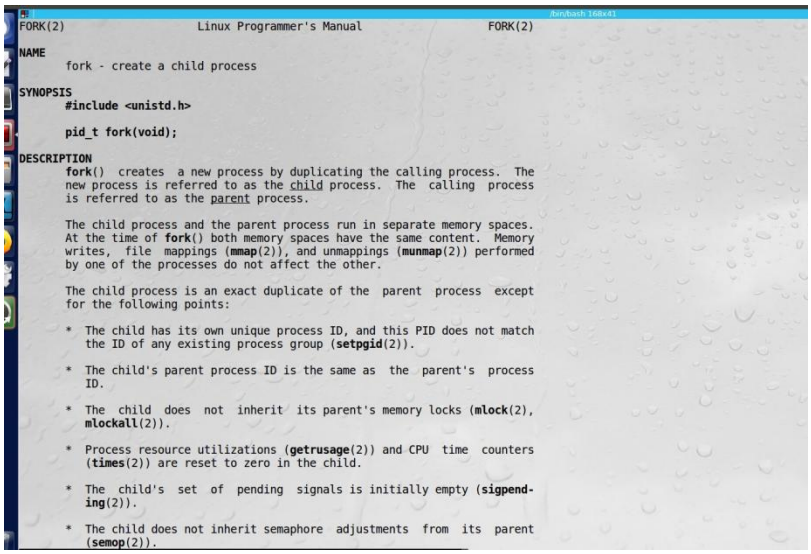
In this task, we study how a child process gets its environment variables from its parent. In Unix, fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent; however, several things are not inherited by the child (please see the manual of fork() by typing the following command: man fork). In this task, we would like to know whether the parent’s environment variables are inherited by the child process or not.

实验结果及分析：

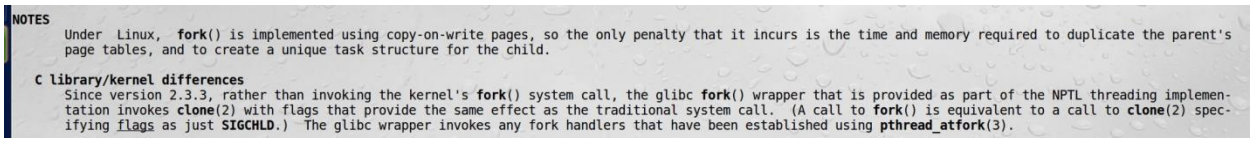
我们想知道父环境变量是否由子进程继承，

```
#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();          ①
      exit(0);
    default:  /* parent process */
      //printenv();        ②
      exit(0);
  }
}
```

首先对 fork 函数做一定了解：

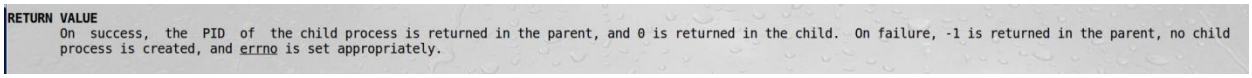


```
FORK(2)                    Linux Programmer's Manual                   FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates  a new process by duplicating the calling process.  The
       new process is referred to as the child process.  The  calling  process
       is referred to as the parent process.

       The child process and the parent process run in separate memory spaces.
       At the time of fork() both memory spaces have the same content.  Memory
       writes,  file  mappings (mmap(2)), and unmappings (munmap(2)) performed
       by one of the processes do not affect the other.

       The child process is an exact duplicate of the  parent  process  except
       for the following points:

       *  The child has its own unique process ID, and this PID does not match
          the ID of any existing process group (setpgid(2)).

       *  The child's parent process ID is the same as  the  parent's  process
          ID.

       *  The  child  does  not  inherit  its parent's memory locks (mlock(2),
          mlockall(2)).

       *  Process resource utilizations (getrusage(2)) and CPU  time  counters
          (times(2)) are reset to zero in the child.

       *  The  child's  set  of  pending  signals is initially empty (sigpend-
          ing(2)).

       *  The child does not inherit semaphore  adjustments  from  its  parent
          (semop(2)).
```



```
NOTES
       Under  Linux,  fork() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's
       page tables, and to create a unique task structure for the child.

   C library/kernel differences
       Since version 2.3.3, rather than invoking the kernel's fork() system call, the glibc fork() wrapper that is provided as part of the NPTL threading implemen-
       tation invokes clone(2) with flags that provide the same effect as the traditional system call.  (A call to fork() is equivalent to a call to clone(2) spec-
       ifying flags as just SIGCHLD.)  The glibc wrapper invokes any fork handlers that have been established using pthread_atfork(3).
```

使用 man fork 命令,我们发现 fork 函数创建一个父进程的副本,即子进程,除了一些进程 ID,memory lock 等不同外,其余均相同,当然也包括父进程的环境变量值,全部复制给子进程



```
RETURN VALUE
       On  success,  the  PID  of  the child process is returned in the parent, and 0 is returned in the child.  On failure, -1 is returned in the parent, no child
       process is created, and errno is set appropriately.
```

观察返回值信息，即 fork 函数运行后,程序会出现两个进程,一个父进程(本身),一个子进程,如果成功父进程返回子进程的 ID,子进程返回 0,如果失败,父进程返回-1,没有子进程产生,则产生相应的错误码。

输入命令，将程序运行结果写到两个文件之中child1 child2，第一个是父进程的环境变量，第二个是子进程

```
07/06/21]seed@VM:~$ gcc task2.c -o task2.out
07/06/21]seed@VM:~$ task2.out>child_1
07/06/21]seed@VM:~$ gedit child_1
07/06/21]seed@VM:~$ gedit task2.c
07/06/21]seed@VM:~$ gcc task2.c -o task2.out
07/06/21]seed@VM:~$ task2.out>child_2
07/06/21]seed@VM:~$ gedit child 2
```

```
1 SHELL=/bin/bash
2 SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2861,unix/VM:/
  tmp/.ICE-unix/2861
3 QT_ACCESSIBILITY=1
4 COLORTERM=truecolor
5 XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
6 XDG_MENU_PREFIX=gnome-
7 GNOME_DESKTOP_SESSION_ID=this-is-deprecated
8 GNOME_SHELL_SESSION_MODE=ubuntu
9 SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
0 XMODIFIERS=@im=ibus
1 DESKTOP_SESSION=ubuntu
2 SSH_AGENT_PID=2800
3 GTK_MODULES=gail:atk-bridge
4 PWD=/home/seed
5 LOGNAME=seed
6 XDG_SESSION_DESKTOP=ubuntu
7 XDG_SESSION_TYPE=x11
8 GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
9 XAUTHORITY=/run/user/1000/gdm/Xauthority
0 GJS_DEBUG_TOPICS=JS ERROR;JS LOG
1 WINDOWPATH=2
2 HOME=/home/seed
3 USERNAME=seed
4 IM_CONFIG_PHASE=1
5 LANG=en_US.UTF-8
6 LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=0
  7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=
7 XDG_CURRENT_DESKTOP=ubuntu:GNOME
8 VTE_VERSION=6003
9 GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/
  1ce5ba2f_5169_45e4_8d6d_b35904a7b985
```



```
1 SHELL=/bin/
2 SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2861,unix/VM:/
  tmp/.ICE-unix/2861
3 QT_ACCESSIBILITY=1
4 COLORTERM=truecolor
5 XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
6 XDG_MENU_PREFIX=gnome-
7 GNOME_DESKTOP_SESSION_ID=this-is-deprecated
8 GNOME_SHELL_SESSION_MODE=ubuntu
9 SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
10 XMODIFIERS=@im=ibus
11 DESKTOP_SESSION=ubuntu
12 SSH_AGENT_PID=2800
13 GTK_MODULES=gail:atk-bridge
14 PWD=/home/seed
15 LOGNAME=seed
16 XDG_SESSION_DESKTOP=ubuntu
17 XDG_SESSION_TYPE=x11
18 GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
19 XAUTHORITY=/run/user/1000/gdm/Xauthority
20 GJS_DEBUG_TOPICS=JS ERROR;JS LOG
21 WINDOWPATH=2
22 HOME=/home/seed
23 USERNAME=seed
24 IM_CONFIG_PHASE=1
25 LANG=en_US.UTF-8
26 LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=0
  7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=
27 XDG_CURRENT_DESKTOP=ubuntu:GNOME
28 VTE_VERSION=6003
29 GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/
  1ce5ba2f_5169_45e4_8d6d_b35904a7b985
```

左为父进程，右为子进程（把程序修改一下即可得到父进程子进程，注意看源码）可以发现，产生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。

Task3

实验内容：Environment Variables and execve()

In this task, we study how environment variables are affected when a new program is executed via execve(). The function execve() calls a system call to load a new command and execute it; this function never return. No new process is created; instead, the calling process's text, data, bss, and stack are overwritten by that of the program loaded. Essentially, execve() runs the new program inside the calling process. We are interested in what happens to the environment variables; are they automatically inherited by the new program?

实验结果及分析：

```
#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);     ①

  return 0 ;
}
```

我们可以比较将execve函数里面的null改为environ进行比较，

```
07/06/21]seed@VM:~$ gcc task_null.c -o task_null.out
07/06/21]seed@VM:~$ ./task_null.out
07/06/21]seed@VM:~$
```

execve第三参数为NULL版本没有输出

```
[07/06/21]seed@VM:~$ gcc task_env.c -o task_env.out
[07/06/21]seed@VM:~$ ./task_env.out
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2861,unix/VM:/tmp/.ICE-
unix/2861
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
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=2800
GTK_MODULES=gail:atk-bridge
PWD=/home/seed
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/1000/gdm/Xauthority
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
WINDOWPATH=2
```

execve第三参数为environ版本有输出。可以发现后者成功输出了当前进程的环境变量，而前者失败了。

分析：对于 execve() 函数，第一个参数用来执行参数filename字符串所代表的文件路径，第二个参数是利用 指针数组来传递给执行文件，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。成功不返回，失败返回-1。 前者参数三传递的是NULL,所以打印为空， 后者传递了外部环境变量数据给新程序，从而打印出了当前环境变量。

Task4

实验内容：Environment Variables and system()In this task, we study how environment variables are affected when a new program is executed via the system() function. This function is used to execute a command, but unlike execve(), which directly executes a command, system() actually executes "/bin/sh -c command", i.e., it executes/bin/sh, and asks the shell to execute the command.If you look at the implementation of the system() function, you will see that it uses execl() to execute /bin/sh; execl() calls execve(), passing to it the environment variables array. Therefore,using system(), the environment variables of the calling process is passed to the new program /bin/sh. Please compile and run the following program to verify this.

实验结果及分析：

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

int main()
{
  system("/usr/bin/env");

  return 0 ;
}
```

分析分别由system和execve两个函数分别调用的结果

```
07/06/21]seed@VM:~$ gedit task4_system.c
07/06/21]seed@VM:~$ gedit task4_execve.c
07/06/21]seed@VM:~$ gcc task4_system.c -o task4_system.out
07/06/21]seed@VM:~$ ./task4_system.out
JS_DEBUG_TOPICS=JS ERROR;JS LOG
ESSOPEN=| /usr/bin/lesspipe %s
SER=seed
SH_AGENT_PID=2800
DG_SESSION_TYPE=x11
HLVL=1
OME=/home/seed
ESKTOP_SESSION=ubuntu
NOME_SHELL_SESSION_MODE=ubuntu
TK_MODULES=gail:atk-bridge
ANAGERPID=2595
BUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
OLORTERM=truecolor
M_CONFIG_PHASE=1
OGNAME=seed
OURNAL_STREAM=9:46817
=./task4_system.out
DG_SESSION_CLASS=user
SERNAME=seed
ERM=xterm-256color
NOME_DESKTOP_SESSION_ID=this-is-deprecated
INDOWPATH=2
ATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:.
ESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2861,unix/VM:/tmp/.ICE-unix/2861
NVOCATION_ID=2eae2f6938294992aa36129f8367afa3
DG_MENU_PREFIX=gnome-
NOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/1ce5ba2f_5169_45e4_8d6d_b35904a7b985
DG_RUNTIME_DIR=/run/user/1000
ISPLAY=:0
ANG=en_US.UTF-8
DG_CURRENT_DESKTOP=ubuntu:GNOME
MODIFIERS=@im=ibus
DG_SESSION_DESKTOP=ubuntu
AUTHORITY=/run/user/1000/gdm/Xauthority
S_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=3
2:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lz
lz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=
01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.e
ar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*
.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=
01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*
*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01
.;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=0
1;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.m
pc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
NOME_TERMINAL_SERVICE=:1.115
SH_AUTH_SOCK=/run/user/1000/keyring/ssh
HELL=/bin/bash
T_ACCESSIBILITY=1
DMSESSION=ubuntu
.ESSCLOSE=/usr/bin/lesspipe %s %s
PG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
JS_DEBUG_OUTPUT=stderr
T_IM_MODULE=ibus
WD=/home/seed
DG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
DG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share/:/usr/share/:/var/lib/snapd/desktop
TE_VERSION=6003
07/06/21]seed@VM:~$
```

这个是system调用

```
[07/06/21]seed@VM:~$ ./task4_execve.out
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2861,unix/VM:/tmp/.ICE-unix/2861
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
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=2800
GTK_MODULES=gail:atk-bridge
PWD=/home/seed
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/1000/gdm/Xauthority
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
WINDOWPATH=2
HOME=/home/seed
USERNAME=seed
IM_CONFIG_PHASE=1
LANG=en_US.UTF-8
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43
42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31
tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:
=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;3
sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01
*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*
=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01
:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.w
1;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.
01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;3
mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
XDG_CURRENT_DESKTOP=ubuntu:GNOME
VTE_VERSION=6003
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/1ce5ba2f_5169_45e4_8d6d_b35904a7b985
INVOCATION_ID=2eae2f6938294992aa36129f8367afa3
MANAGERPID=2595
GJS_DEBUG_OUTPUT=stderr
LESSCLOSE=/usr/bin/lesspipe %s %s
XDG_SESSION_CLASS=user
TERM=xterm-256color
LESSOPEN=| /usr/bin/lesspipe %s
USER=seed
GNOME_TERMINAL_SERVICE=:1.115
DISPLAY=:0
SHLVL=1
QT_IM_MODULE=ibus
XDG_RUNTIME_DIR=/run/user/1000
JOURNAL_STREAM=9:46817
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share/:/usr/share/:/var/lib/snapd/desktop
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:.
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
_=./task4_execve.out
[07/06/21]seed@VM:~$
```

这个是execve调用

分析实验结果：

除开运行的文件命以外，环境变量一样，即输出的环境变量为当前程序的环境变量

参考教程里所述， system 通过 execl() 不仅调用了 /bin/sh ,还将当前的环境变量数组传递给了 execve() 。执行流程即 fork child_process ， child_process:exec commandString ， father_process:wait child_process exit 。因此输出的环境变量为当前程序实际运行的环境变量。执行成功返回子shell终止状态， fork() 失败返 回-1，类比 execve ,即shell执行exit命令。

Task5

实验内容：Environment Variable and Set-UID Programs

Set-UID is an important security mechanism in Unix operating systems. When a Set-UID program runs, it assumes the owner's privileges. For example, if the program's owner is root, then when anyone runs this program, the program gains the root's privileges during its execution. Set-UID allows us to do many interesting things, but it escalates the user's privilege when executed, making it quite risky. Although the behaviors of Set-UID programs are decided by their program logic, not by users, users can indeed affect the behaviors via environment variables. To understand how Set-UID programs are affected, let us figure out whether environment variables are inherited by the Set-UID program's process from the user's process.

实验结果及分析：

第一步

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

extern char **environ;

void main()
{
  int i = 0;
  while (environ[i] != NULL) {
     printf("%s\n", environ[i]);
     i++;
  }
}
```

第二步

```
// Asssume the program's name is foo
$ sudo chown root foo
$ sudo chmod 4755 foo
```

编译得到可执行程序Task5.out 并完成Step2,给程序手动提权

```
[07/06/21]seed@VM:~$ sudo chown root task5.out
[07/06/21]seed@VM:~$ sudo chmod 4775 task5.out
[07/06/21]seed@VM:~$ export PATH=$PATH:/zmx
[07/06/21]seed@VM:~$ export LD_LIBRRARY_PATH=$LD_LIBRRARY_PATH:/
zmx
[07/06/21]seed@VM:~$ export LEMONOIL=$LEMONOIL:/zmx
[07/06/21]seed@VM:~$ ./task5.out
SHELL=/bin/bash
SESSION_MANAGER=local/VM:@/tmp/.ICE-unix/2861,unix/VM:/tmp/.ICE-
unix/2861
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
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
SSH_AGENT_PID=2800
GTK_MODULES=gail:atk-bridge
PWD=/home/seed
LOGNAME=seed
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/1000/gdm/Xauthority
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
WINDOWPATH=2
HOME=/home/seed
USERNAME=seed
IM_CONFIG_PHASE=1
LANG=en_US.UTF-8
LEMONOIL=:/zmx
1;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:
*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf
=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;
36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:
*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opu
s=00;36:*.spx=00;36:*.xspf=00;36:
XDG_CURRENT_DESKTOP=ubuntu:GNOME
VTE_VERSION=6003
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/1ce5ba2f_5169_4
5e4_8d6d_b35904a7b985
INVOCATION_ID=2eae2f6938294992aa36129f8367afa3
MANAGERPID=2595
GJS_DEBUG_OUTPUT=stderr
LESSCLOSE=/usr/bin/lesspipe %s %s
XDG_SESSION_CLASS=user
TERM=xterm-256color
LESSOPEN=| /usr/bin/lesspipe %s
USER=seed
GNOME_TERMINAL_SERVICE=:1.115
DISPLAY=:0
SHLVL=1
QT_IM_MODULE=ibus
XDG_RUNTIME_DIR=/run/user/1000
JOURNAL_STREAM=9:46817
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share/:/usr/share/:/v
ar/lib/snapd/desktop
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bi
n:/usr/games:/usr/local/games:/snap/bin:.:/zmx
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
LD_LIBRRARY_PATH=:/zmx
_=./task5.out
[07/06/21]seed@VM:~$
```

可以明显看出来PATH等后面留有export进去的/zmx,发现设置 LD_LIBRARY_PATH 环境变量(动态库的查找路径不会出现在子进程的环境变量中,而剩下的可以被包含。

```c
int main()
{
    system("ls");
    return 0;
}
```

Task6

实验内容 The PATH Environment Variable and Set-UID Programs

Because of the shell program invoked, calling system() within a Set-UID program is quite dangerous.

This is because the actual behavior of the shell program can be affected by environment variables, such as PATH; these environment variables are provided by the user, who may be malicious. By changing these variables, malicious users can control the behavior of the Set-UID program.

实验结果及分析：

```c
int main()
{
    system("ls");
    return 0;
}
```

```
[07/06/21]seed@VM:~$ gedit task6.c
[07/06/21]seed@VM:~$ sudo gcc task6.c -o task6.out
[07/06/21]seed@VM:~$ sudo chown root task6.out
[07/06/21]seed@VM:~$ sudo chmod 4775 task6.out
[07/06/21]seed@VM:~$ ./task6.out
child_1    Public            task5.c       task_null.out
child_2    task2.c           task5.out     Templates
Desktop    task2.out         task6.c       test1.c
Documents  task4_execve.c    task6.out     Videos
Downloads  task4_execve.out  task_env.c
Music      task4_system.c    task_env.out
Pictures   task4_system.out  task_null.c
[07/06/21]seed@VM:~$
```

此时运行Task6.out，程序实际上调用了 /bin/ls ，显示正常,就相当于把命令ls用了一次，接下来我们构造 Trick 程序 首先将当前目录添加至PATH之首

```
[07/06/21]seed@VM:~$ export PATH=/home/operationsystem/lab1/task
6:$PATH
[07/06/21]seed@VM:~$ env | grep PATH
WINDOWPATH=2
PATH=/home/operationsystem/lab1/task6:/usr/local/sbin:/usr/local
/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/
snap/bin:.:/zmx
LD_LIBRARY_PATH=:/zmx
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.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;
}
```

（trick程序）

```
[07/06/21]seed@VM:~$ gedit trick_ls.c
[07/06/21]seed@VM:~$ sudo gcc trick_ls.c -o ls
[07/06/21]seed@VM:~$ ./task6.out
```

现在让我们再次运行之前的Task6.out

此时程序并没有像之前一样运行 /bin/ls ,而是转而运行当前目录下所构造的 ls ,该程序被定义为调用 execve 执行 /usr/bin/env ,导致输出的结果不同。 这个trick分为两步，首先Task6.out中 system 函数运行的命令没有提供绝对路径，此时链接器会在环境 变量中逐个寻找含有 ls 程序的目录，找到第一个后就会链接并运行。所以第二步我们通过在环境变量 PATH之前插队，加入当前目录，使得当前目录下的 ls 程序在 PATH 中运行优先级比在其之后 的 /bin/ls 更高，使得trick成立。

Task8

实验内容：Invoking External Programs Using system() versus execve()

Although system() and execve() can both be used to run new programs, system() is quite danger ous if used in a privileged program, such as Set-UID programs. We have seen how the PATH environment variable affect the behavior of system(), because the variable affects how the shell works. execve() does not have the problem, because it does not invoke shell. Invoking shell has another dangerous consequence, and this time, it has nothing to do with environment variables. Let us look at the following scenario. Bob works for an auditing agency, and he needs to investigate a company for a suspected fraud. For the investigation purpose, Bob needs to be able to read all the files in the company's Unix system; on the other hand, to protect the integrity of the system, Bob should not be able to modify any file. To achieve this goal, Vince, the superuser of the system, wrote a special set-root-uid program (see below), and then gave the executable permission to Bob. This program requires Bob to type a fifile name at the command line, and then it will run /bin/cat to display the specifified fifile. Since the program is running as a root, it can display any file Bob specififies. However, since the program has no write operations, Vince is very sure that Bob cannot use this special program to modify any fifile.

实验结果及分析：

```
#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 followings.
  system(command);
  // execve(v[0], v, NULL);

  return 0 ;
}
```

新建一个文件 try_to_delete ,修改其权限，此时我们发现我们没有权限可以删除这样一个我们 自己创建的文件

```
try_to_delete: Permission denied
```

暂时换回 777 权限，向文件中加入内容

```
[07/06/21]seed@VM:~$ sudo chmod 000 try_to_delete
[07/06/21]seed@VM:~$ rm try_to_delete
rm: remove write-protected regular empty file 'try_to_delete'?
[07/06/21]seed@VM:~$ echo "123456">> try_to_delete
[07/06/21]seed@VM:~$ cat try_to_delete
123456
```

利用子shell的方式，通过管道运行了新的命令 rm try_to_delete ，可以观察到文件已被删除

```
[07/06/21]seed@VM:~$ ls
child_1     Public            task5.c       task_null.out
child_2     set-root-uid      task5.out     Templates
Desktop     task2.c           task6.c       test1.c
Documents   task2.out         task6.out     trick_ls.c
Downloads   task4_execve.c    task8_sru.c   try_to_delete
ls          task4_execve.out  task_env.c    Videos
Music       task4_system.c    task_env.out
Pictures    task4_system.out  task_null.c
[07/06/21]seed@VM:~$ rm try_to_delete
rm: remove write-protected regular file 'try_to_delete'? y
[07/06/21]seed@VM:~$ ls
child_1     Public            task5.c       task_null.out
child_2     set-root-uid      task5.out     Templates
Desktop     task2.c           task6.c       test1.c
Documents   task2.out         task6.out     trick_ls.c
Downloads   task4_execve.c    task8_sru.c   Videos
ls          task4_execve.out  task_env.c
Music       task4_system.c    task_env.out
Pictures    task4_system.out  task_null.c
[07/06/21]seed@VM:~$
```

可发现 文件被删除，本来 对 seed 用户是不可写的，但因为 task8 是 SET-UID 程序，且时 root 权限，因此可以删除 文件。由此可得出结论：set-UID 程序是非常危险的。