操作系统实验说明

目录

[一、实验前准备 编译内核 1](#_Toc492026874)

[1.1 VirtualBox安装 1](#_Toc492026875)

[1.2 Linux安装 1](#_Toc492026876)

[1.3编译内核 17](#_Toc492026877)

[二、实验一 Linux进程管理及其扩展 20](#_Toc492026878)

[2.1 隐藏系统调用简介 20](#_Toc492026879)

[2.2隐藏系统调用实现 20](#_Toc492026880)

[三、实验二 shell的实现 29](#_Toc492026881)

[3.1 shell简介 29](#_Toc492026882)

[3.2 shell实现 30](#_Toc492026883)

[1.管道的实现 31](#_Toc492026884)

[2.重定向的实现 31](#_Toc492026885)

[参考文献： 33](#_Toc492026886)

# 一、实验前准备 编译内核

## 1.1 VirtualBox安装

VirtualBox是免费的虚拟机平台，可以搭载操作系统。本次试验采用的版本为4.3.12，安装过程非常简单，中间只需要选定一个安装目录即可，其他直接next，安装完成。

## 1.2 Linux安装

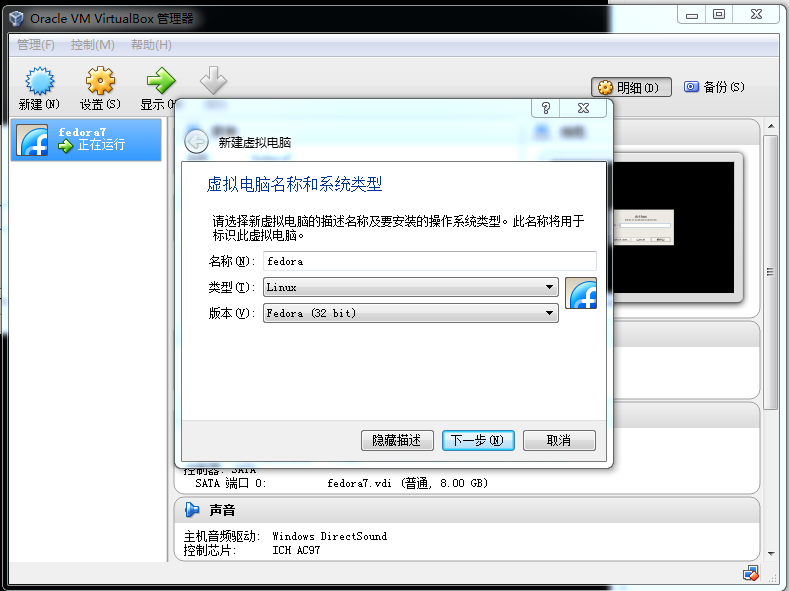
**注意：1.之后的所有路径建议都采用英文路径，不要出现中文**

**2.确保在连接到互联网环境下安装，中间需要下载安装软件**

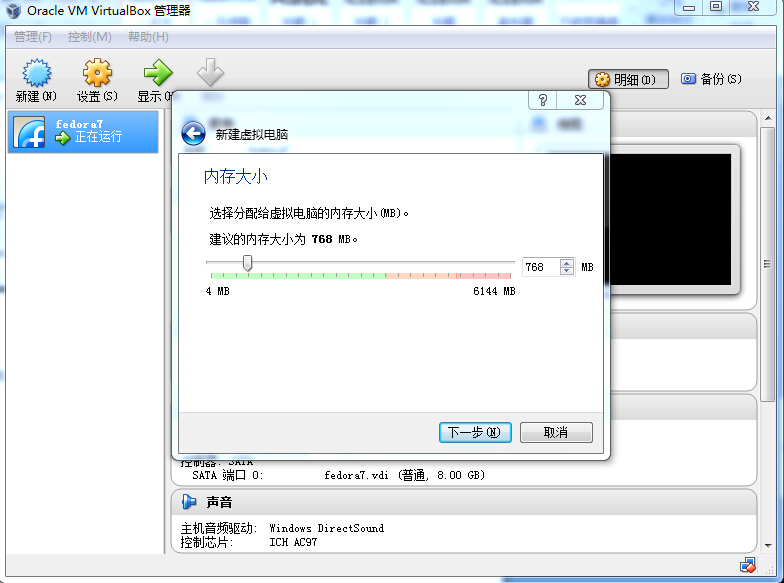
linux系统采用的是Fedora 7发行版，内核版本为2.6.21.7

安装步骤如下：

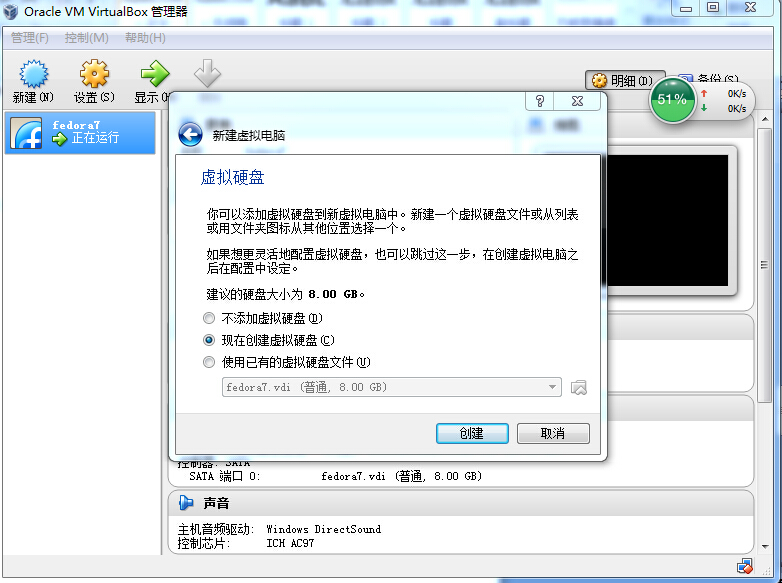
新建虚拟机，名称自定义，类型选择Linux，版本选择Fedora（32bit）



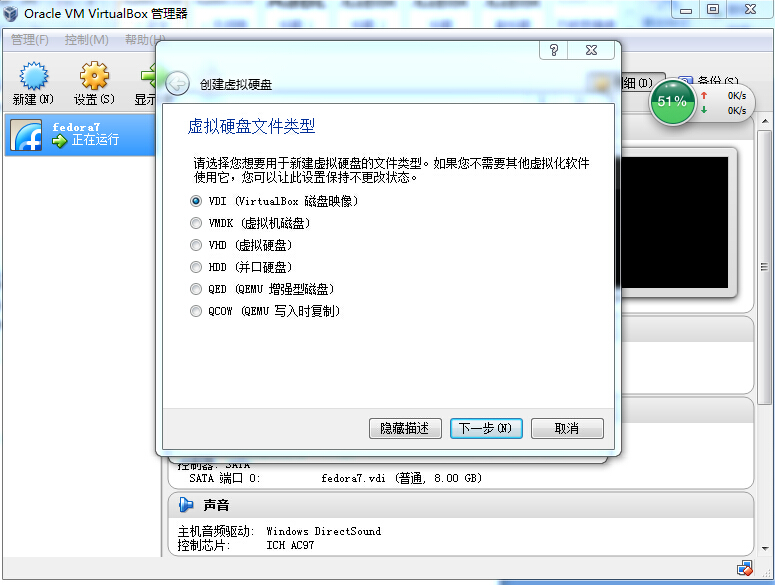
配置内存大小，默认就好，基本够用



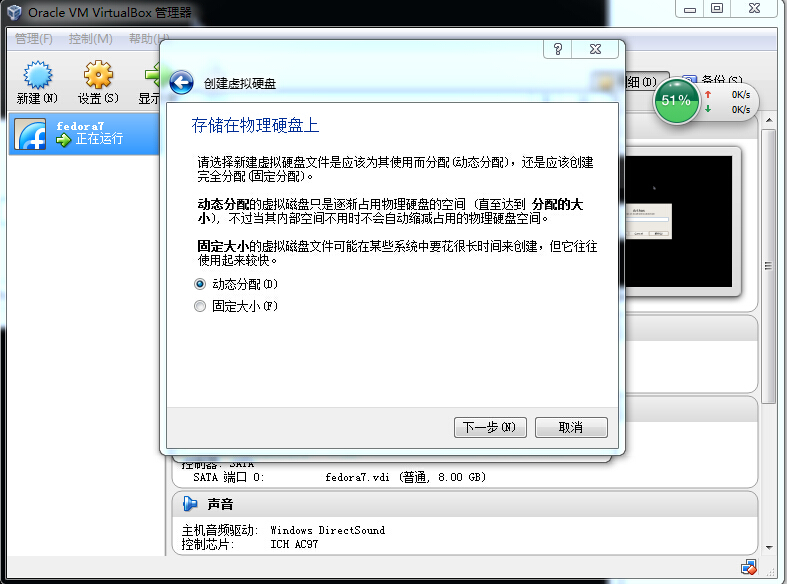
创建虚拟硬盘，选择现在创建虚拟硬盘，点击创建



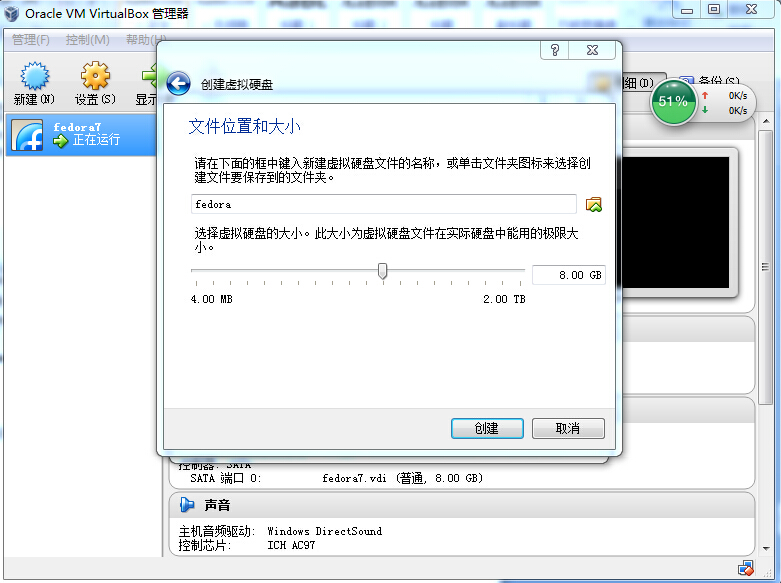
选择VDI，点击下一步



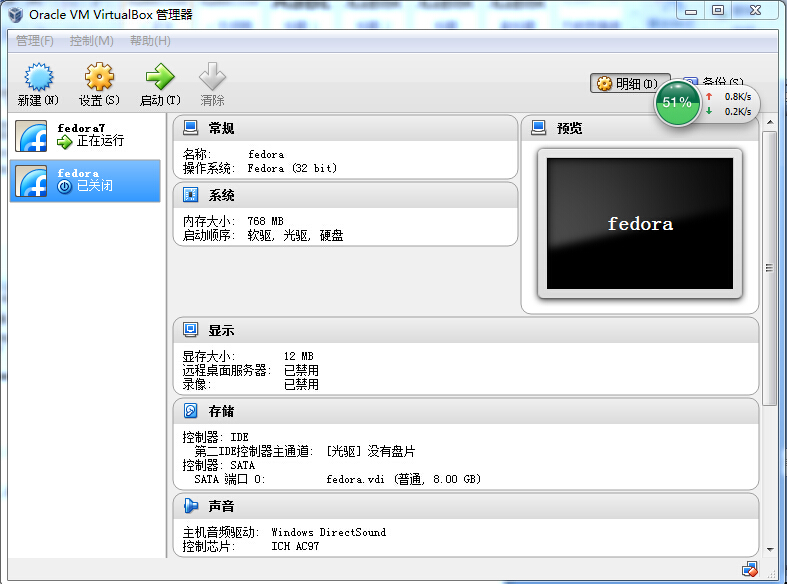
选择动态分配，点击下一步



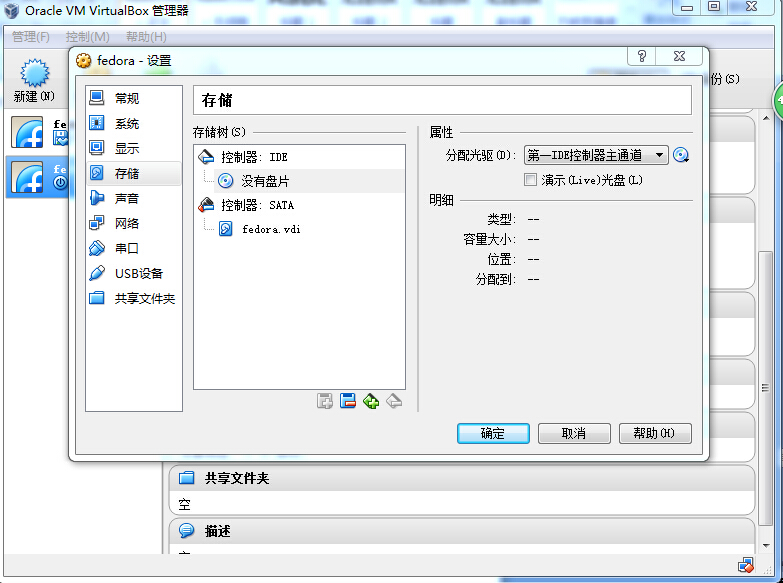
设置文件位置和大小，大小默认，位置要设置在非系统盘，建议新建自己的文件夹，把文件都放里面，完成后点击创建



新创建的fedora（下图第二个）



右键点击fedora->设置->存储->没有盘片,点击最右边的光盘标志，选择一个虚拟光盘，即你的操作系统镜像（.iso）文件

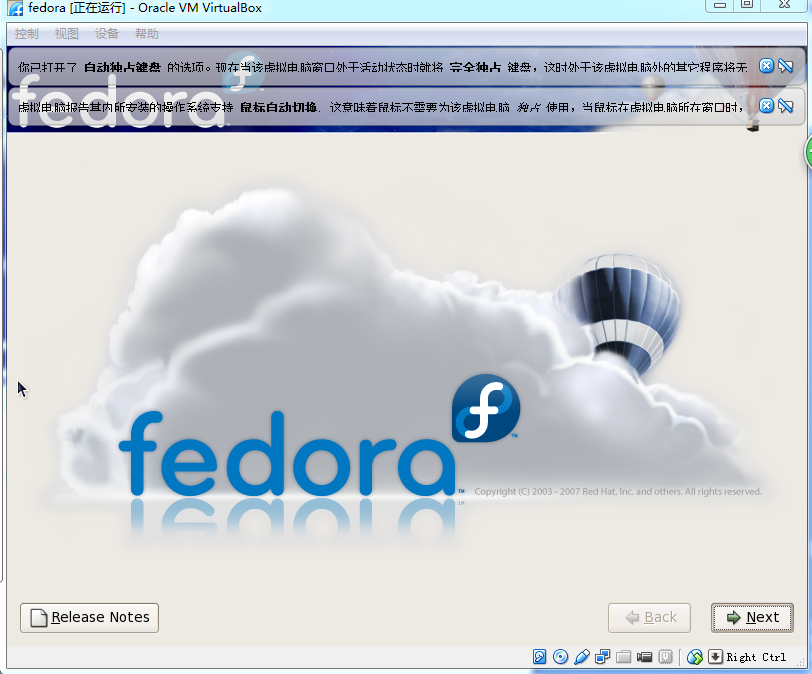


完成后右键fedora->启动，出现下面界面，选择第一项

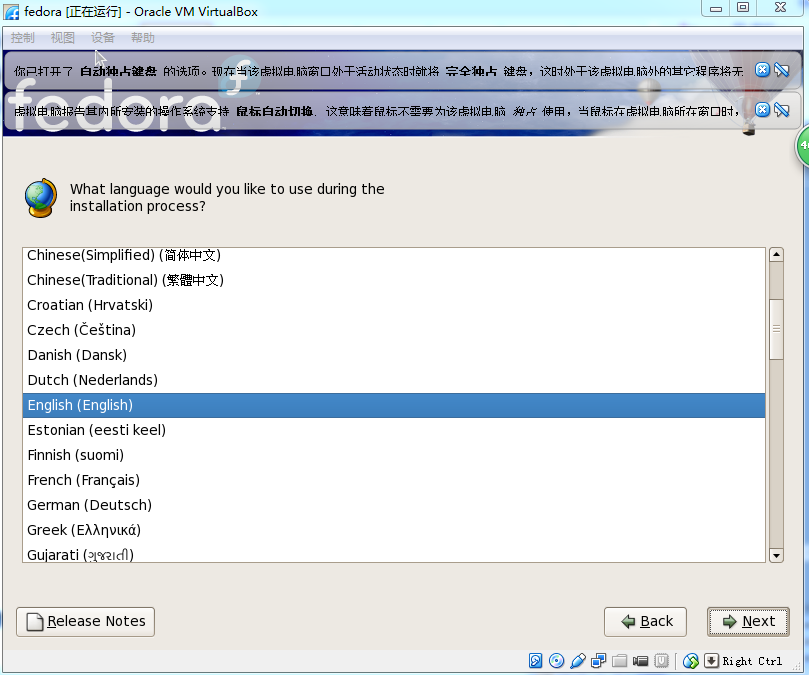


下一个界面直接选择skip，进入下面的界面，选择next

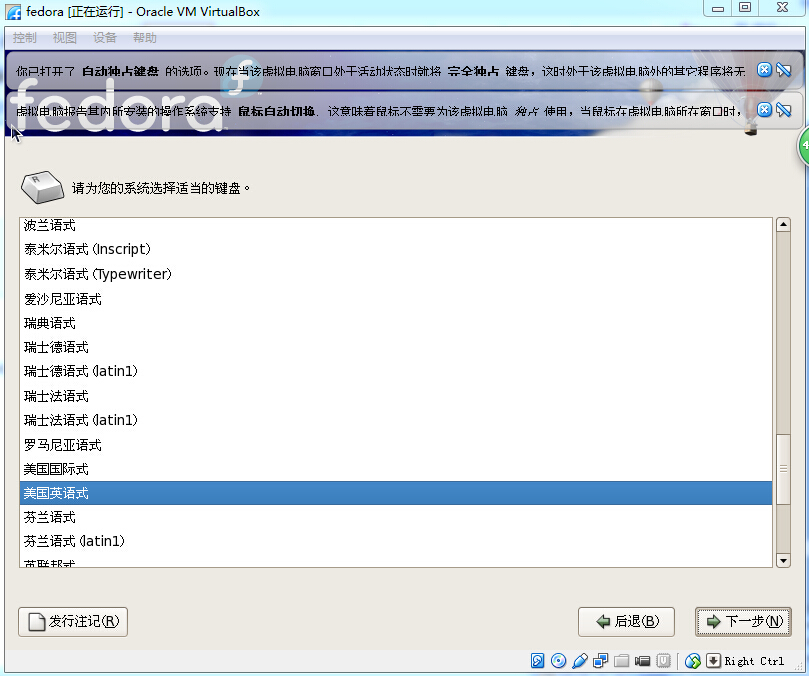
注：这时候出现鼠标共享的情况，按键盘右边的ctrl键进行鼠标切换



选择语言:简体中文，next



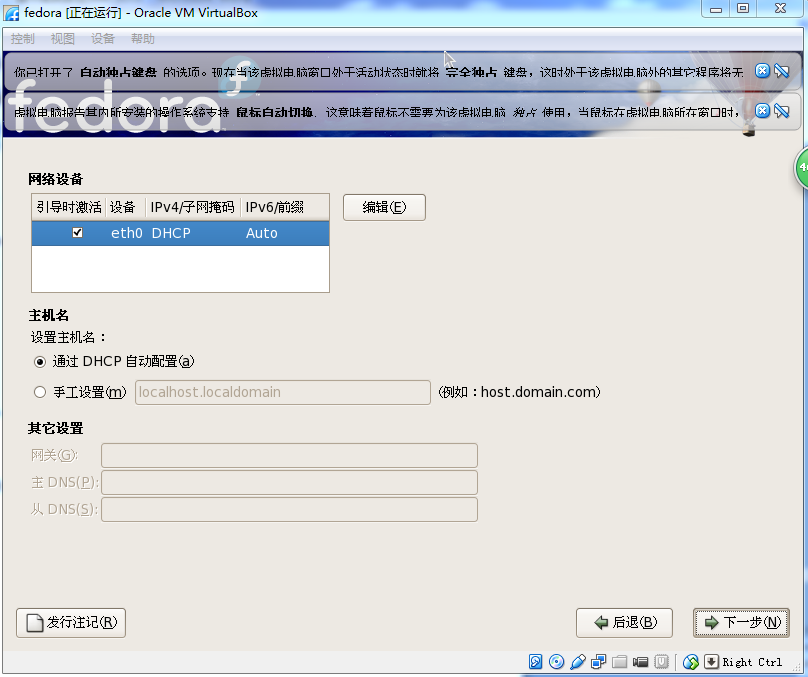
选择键盘：美国英语式，next



出现警告，选择是，出现下面的界面，直接选择下一步



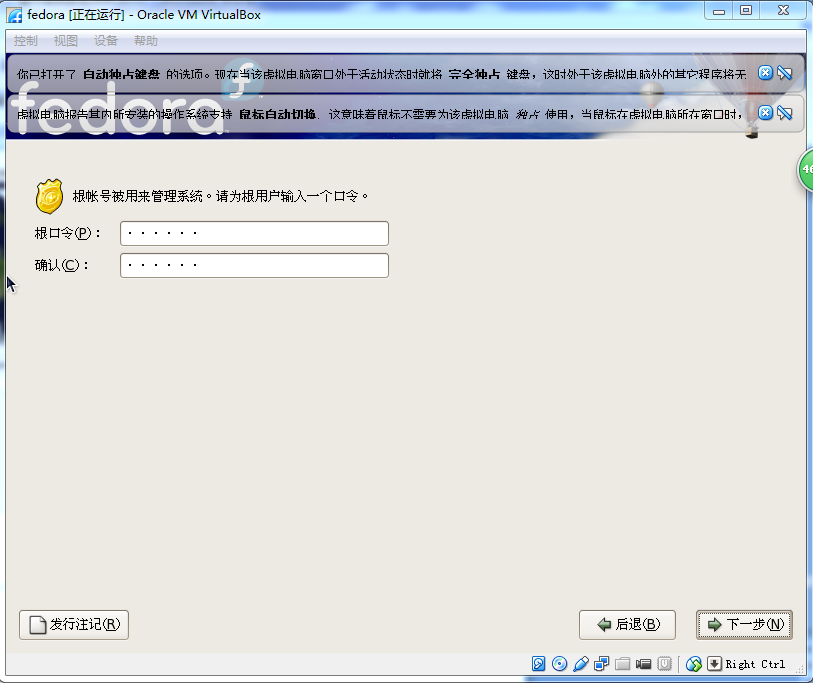
再次出现警告，选择是，出现下面界面，直接下一步



选择时区，任意，不重要（处女座可选），直接next



设置root用户密码,下一步



定制软件，勾选软件开发下一步



后面的直接下一步，到达下面界面，需要较长时间，耐心等待安装结束，下一步



在最后一个界面选择reboot即可启动，启动后设置一下用户名和密码即可，重启进入系统

## 1.3编译内核

启动fedora，输入用户名密码，进入到下面界面



由于无法进行文件共享，故从网上下载linux-2.6.21.7.tar.bz2.（应用程序里有firefox浏览器）把该文件移动到计算机->文件系统->usr->src目录下。

打开应用程序->终端，（最好把终端拖到桌面上，以后经常使用），

输入su root //输入口令，取得root权限

在实验前需要安装一些必要的软件包：

yum install gcc

yum install gcc-c++

yum -y install ncurces

yum install ncurces-devel

cd /usr/src //进入src目录

tar -xvf linux-2.6.21.7.tar.bz2

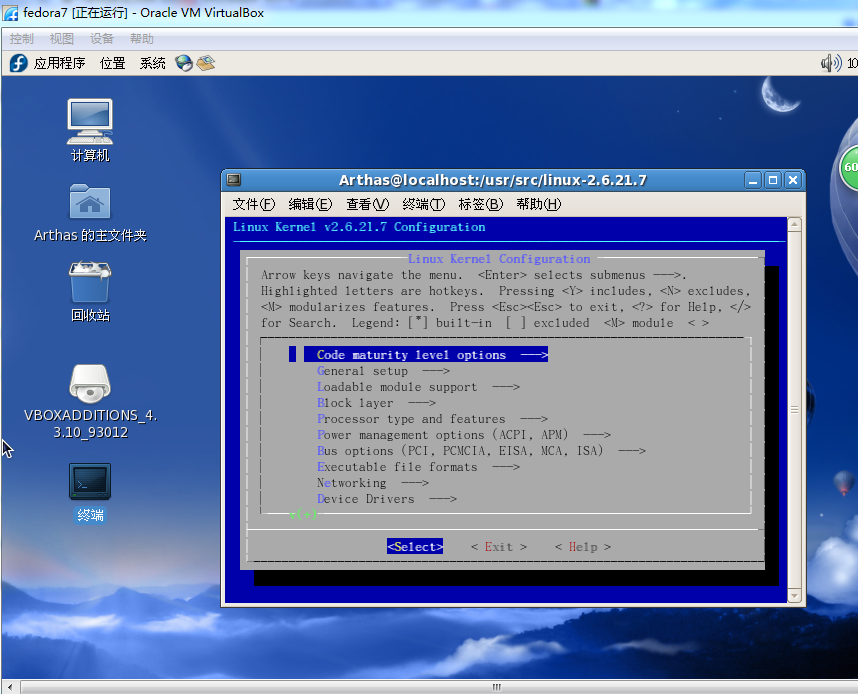
cd linux-2.6.21.7

make mrproper //删除以前垃圾文件

**用make menuconfig启动配置的图形界面。因为配置选项很多，短时间根本搞不懂哪些选项是必须的，哪些是可以不要的，所以，最佳的办法是在一个已经存在的配置文件上面修改。很显然，Fedora 7 就有现成的一个。把 /boot(文件系统里)目录里面 config-2.6.21-1.3194.fc7复制到/usr/src/linux-2.6.21.7里面并命名为.config。当执行make menuconfig时，首先寻找.config,如果找到就读取。否则进入arch目录里面寻找相应体系结构默认的配置文件。**

**cp /boot/ config-2.6.21-1.3194.fc7 .config //复制.config**

**make menuconfig //出现下面界面**



直接exit即可。

make dep //建立相依的属性关系

make clean //去除旧资料

make bzImage //编译内核

make modules //编译模块

make modules\_install //安装模块

make install //安装内核

reboot //重启

重启之后，按F12，选择linux-2.6.21.7（新编译生成的内核），成功启动。

# 二、实验一 Linux进程管理及其扩展

## 2.1 隐藏系统调用简介

隐藏进程一般有两种方法：

方法一：Linux 系统中用来查询文件信息的系统调用是sys\_getdents6。当查询文件或者目录的相关信息时，Linux系统用 sys\_getedents64来执行相应的查询操作，并把得到的信息传递给用户空间运行的程序，所以如果修改该系统调用，去掉结果中与某些特定文件的相关信息，那么所有利用该系统调用的程序将看不见该文件，从而达到了隐藏的目的。首先介绍一下原来的系统调用，其原型为：

|  |
| --- |
| int sys\_getdents64(unsigned int fd, struct linux\_dirent64 \*dirp,unsigned int count) |

其中fd为指向目录文件的文件描述符，该函数根据fd所指向的目录文件读取相应dirent结构，并放入dirp中，其中count为dirp中返回的数据量，正确时该函数返回值为填充到dirp的字节数。

因此，只需要把上述的sys\_getdents64替换成自己写的hacked\_getdents函数，对隐藏的进程进行过滤，从而实现进程的隐藏。

方法二:投机取巧的方法，但是效果一样。把task->pid变成0，就成了0号进程。而在ps，top命令中，是不显示0号进程的相关信息。这么一来，在/proc/文件夹下就不会有该进程的相关信息了。

## 2.2隐藏系统调用实现

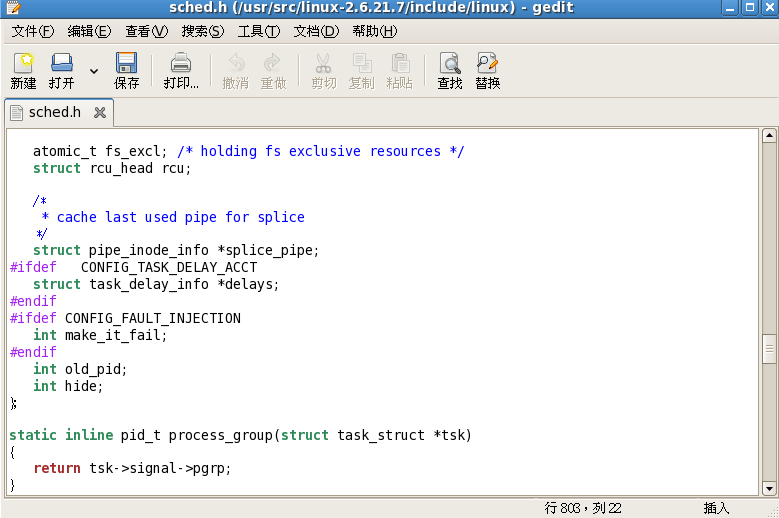
**修改/usr/src/linux-2.6.21.7/include/linux/sched.h。**

在task\_structd的末尾添加变量hide和old\_pid。

**注意：一定要在末尾添加**

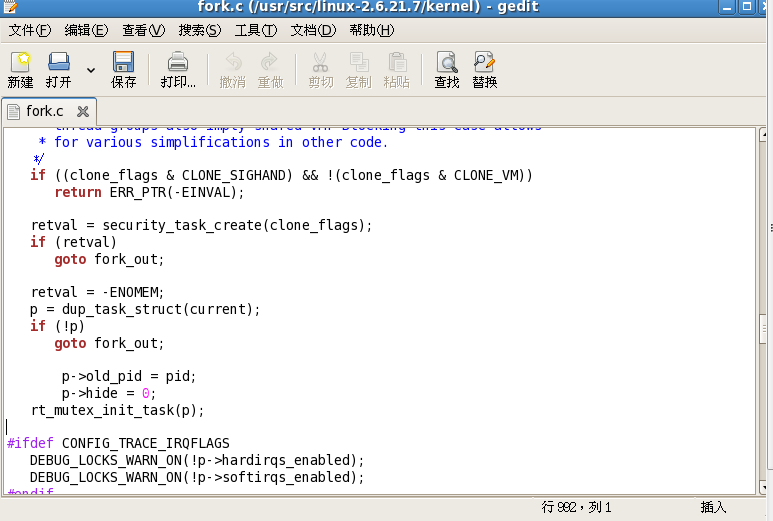
hide用于表示是否隐藏，old\_pid用于保存进程原本的pid，用于进程的恢复。

如下图：



**修改/usr/src/linux-2.6.21.7/kernel/fork.c。**

在copy\_process函数如下位置加上p->old\_pid = pid; p->hide = 0; 进行初始化。如下图：



**修改系统调用sys.c,在文件结尾添加新的系统调用hide和hide\_user\_process**

asmlinkage int sys\_hide( pid\_t pid, int on )

{

struct task\_struct \*p;

struct task\_struct \* me = NULL;

p = &init\_task;

do

{

if( pid == p->old\_pid )

{

me = p;

break;

}

}while( ( p = next\_task(p) ) && ( p != &init\_task ) );

if( current->uid != 0 || me == NULL ) //判断是否为root用户或者pid是否有效

return 0;

if( on == 1 )

{

me->pid = 0; //置pid为0，隐藏

me->hide = 1; //置隐藏标志为1

}

else

{

if( me->hide == 1 )

{

me->pid = me->old\_pid; //取消隐藏

me->hide = 0;

}

}

return 0;

}

asmlinkage int sys\_hide\_user\_process( uid\_t uid, char\* binname )

{

if( current->uid != 0 )

return 0;

struct task\_struct \*p;

p = &init\_task;

if( binname == NULL ) //隐藏该用户所有进程

{

do

{

if( p->uid == uid )

{

p->pid = 0;

p->hide = 1;

}

}while( ( p = next\_task(p) ) && ( p != &init\_task ) );

}

else //隐藏特定进程

{

do

{

if( p->uid == uid )

{

int flag = 1;

int i = 0;

for( i = 0; ( binname[ i ] != NULL && p->comm[ i ] != NULL ); i++ )

{

if( binname[ i ] != p->comm[ i ] )

{

flag = 0;

break;

}

}

if( flag == 1 && binname[ i ] == NULL && p->comm[ i ] == NULL ) //判断名字是否相同

{

p->pid = 0;

p->hide = 1;

}

}

}while( ( p = next\_task(p) ) && ( p != &init\_task ) );

}

return 0;

}

修改完之后，再别覆盖，接下来还得修改sys.c

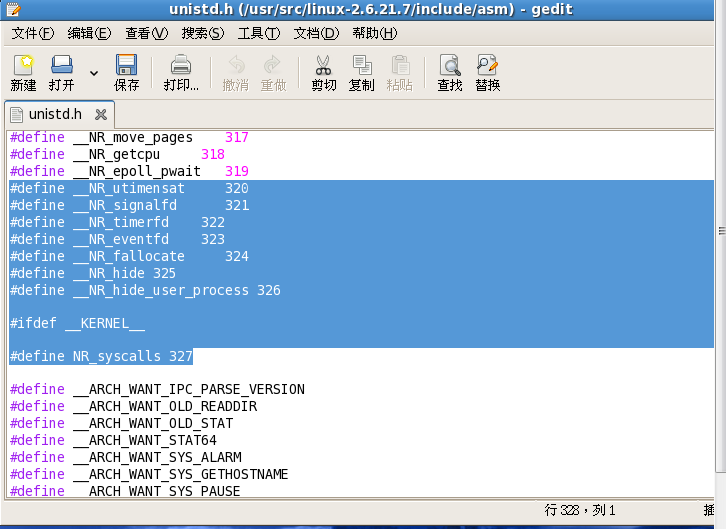
**修改头文件unistd.h**

/usr/src/linux-2.6.21.7/include/asm/unistd.h 这个是内核代码头文件

/usr/include/asm/unistd.h 标准C库的头文件

发现两个头文件定义的系统调用个数不同，为使这个一致，在/usr/src/linux-2.6.21.7/include/asm/unistd.h中加入/usr/include/asm/unistd.h

中多出来的几个系统调用，并且加上自定义的系统调用，如下图：



**不要忘记修改 #define NR\_syscall 327. 327代表总系统调用个数**

同样修改/usr/include/asm下的unistd.h，使两者系统调用相同。

重新修改sys.c，增加新系统调用，donothing就行

asmlinkage int sys\_utimensat( void )

{

return 0;

}

asmlinkage int sys\_signalfd( void )

{

return 0;

}

asmlinkage int sys\_timerfd( void )

{

return 0;

}

asmlinkage int sys\_eventfd( void )

{

return 0;

}

asmlinkage int sys\_fallocate( void )

{

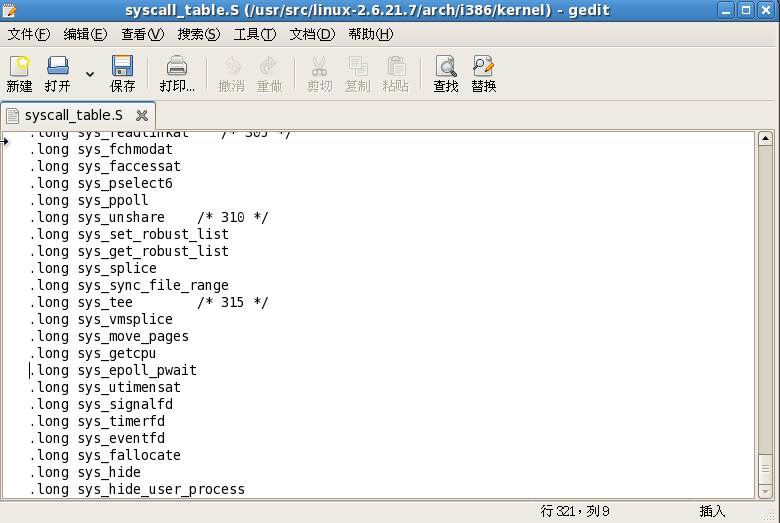
return 0;

}

**最后修改syscall\_table**

直接打开 /usr/src/linux-2.6.21.7/arch/i386/kernel/syscall\_table.S

在末尾添加新系统调用，如下图所示。



按实验二步骤重新编译内核，漫长等待，完成后重启。

**编写测试函数test1.c测试326号系统调用**

#include<linux/unistd.h>

#include<sys/syscall.h>

#include<stdio.h>

#define \_\_NR\_hide\_user\_process 326

main()

{

printf( "%d", 6 );

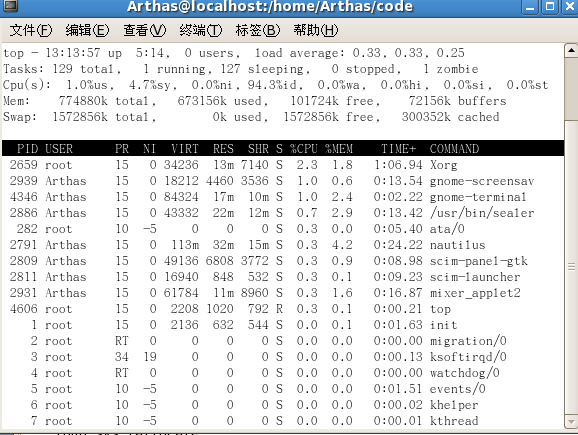
syscall(\_\_NR\_hide\_user\_process, 500, 0 ); //uid 500是本机用户，uid 0 是//root用户

printf( "%d", 6);

}

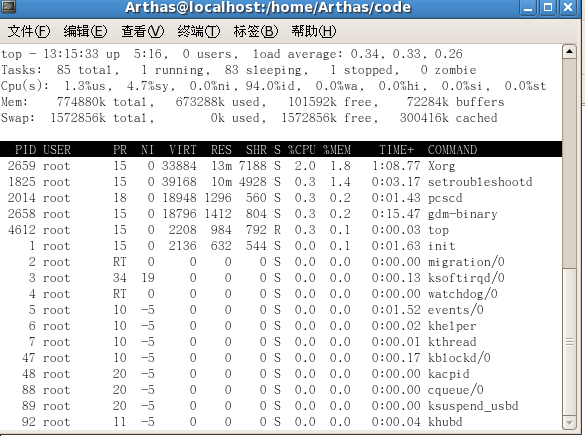
**编译，**

**先不执行，用top查看所有进程，如下图：**



Arthas即为uid为500的用户，再执行测试程序：

结果如下图：



用户Arthas的进程已全部隐藏

**编写测试函数test2.c测试325号系统调用**

#include<linux/unistd.h>

#include<sys/syscall.h>

#include<stdio.h>

#define \_\_NR\_hide 325

main()

{

printf( "%d", 5 );

syscall(\_\_NR\_hide, 2940, 1 ); //2940为su程序pid

printf( "%d", 5);

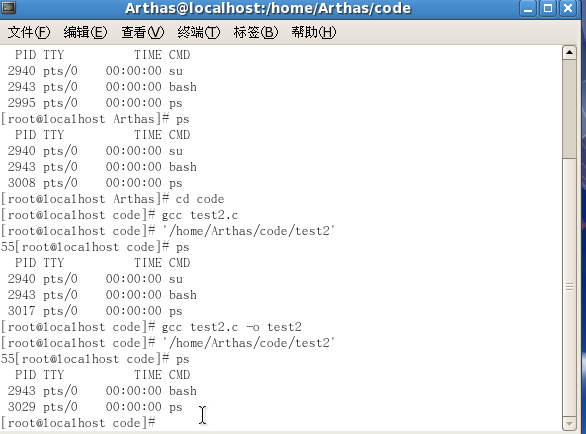
}

**编译，先用ps看进程，如下图**



再执行测试程序，结果如下图：

pid 2940号进程已被隐藏。



**注意：执行上面两个程序，退出终端时有可能导致死机。**

# 三、实验二 shell的实现

## 3.1 shell简介

**在shell中有管道和输入输出重定向两个比较重要的概念。**

**管道：**

管道是Linux中很重要的一种通信方式,是把一个程序的输出直接连接到另一个程序的输入

在Linux中，管道是一种使用非常频繁的通信机制。从本质上说，管道也是一种文件，但它又和一般的文件有所不同，管道可以克服使用文件进行通信的两个问题，具体表现为：

 限制管道的大小。实际上，管道是一个固定大小的缓冲区。在Linux中，该缓冲区的大小为1页，即4K字节，使得它的大小不象文件那样不加检验地增长。使用单个固定缓冲区也会带来问题，比如在写管道时可能变满，当这种情况发生时，随后对管道的write()调用将默认地被阻塞，等待某些数据被读取，以便腾出足够的空间供write()调用写。

 读取进程也可能工作得比写进程快。当所有当前进程数据已被读取时，管道变空。当这种情况发生时，一个随后的read()调用将默认地被阻塞，等待某些数据被写入，这解决了read()调用返回文件结束的问题。

注意：从管道读数据是一次性操作，数据一旦被读，它就从管道中被抛弃，释放空间以便写更多的数据。

**输入输出重定向：**

linux文件描述符：可以理解为linux跟踪打开文件，而分配的一个数字，这个数字有点类似c语言操作文件时候的句柄，通过句柄就可以实现文件的读写操作。 用户可以自定义文件描述符范围是：3-num,这个最大数字，跟用户的：ulimit –n 定义数字有关系，不能超过最大值。   
  
linux启动后，会默认打开3个文件描述符，分别是：标准输入standard input 0,正确输出standard output 1,错误输出：error output 2   
  
以后打开文件后。新增文件绑定描述符 可以依次增加。 一条shell命令执行，都会继承父进程的文件描述符。因此，所有运行的shell命令，都会有默认3个文件描述符。   
  
对于任何一条linux 命令执行，它会是这样一个过程：



一个命令执行了：   
先有一个输入：输入可以从键盘，也可以从文件得到   
命令执行完成：成功了，会把成功结果输出到屏幕：standard output默认是屏幕   
命令执行有错误：会把错误也输出到屏幕上面：standard error默认也是指的屏幕   
  
文件输入输出由追踪为一个给定的进程所有打开文件的整数句柄来完成。这些数字值就是文件描述符。最为人们所知的文件米描述符是 stdin, stdout 和 stderr，文件描述符的数字分别是0，1和2。这些数字和各自的设备是保留的。一个命令执行前，先会准备好所有输入输出，默认分别绑定（stdin,stdout,stderr)，如果这个时候出现错误，命令将终止，不会执行。命令解析过程，可以参考：Linux Shell 通配符、元字符、转义符使用实例介绍   
  
这些默认的输出，输入都是linux系统内定的，我们在使用过程中，有时候并不希望执行结果输出到屏幕。我想输出到文件或其它设备。这个时候我们就需要进行输出重定向了，即不再使用默认的输入输出方向，自定义输入输出的方向。

## 3.2 shell实现

相比于前几个实验，shell的实现已经回归到我们熟悉的编程中来，只要是对系统接口的使用，下面的代码是参考网上的。

### 1.管道的实现

管道输出核心代码为：

order = trim(strtok(cmd, "|")); //取得第一个命令

other = trim(strtok(NULL, "")); //取剩下的命令

if (!other) //如果只有一个命令，直接重定向

redirect(order);

else {

pipe(&fd[0]);

if ((pid = fork()) == 0) { //子进程，执行order并重定向

close(fd[0]); //关闭管道输入

close(STD\_OUT); //关闭标注输出

dup(fd[1]); //把输出连接到fd[1]上，即输出到fd[1]所指向的文件上

close(fd[1]); //关闭fd[1]

redirect(order); //

} else { //父进程，执行剩下的order，进行输入重定向

close(fd[1]);

close(STD\_IN);

dup(fd[0]);

close(fd[0]);

waitpid(pid, &status, 0);

pipel(other);

}}

### 2.重定向的实现

redirectcommand := command ">" outfile | // type = 4

command "<" infile | //type =3

command "<" infile ">" outfile | //type = 6

command ">" outfile "<" infile | //type =5

command //type =2

输入输出重定向核心代码：

if (type == 4 || type == 5 || type == 6) {

if ((fd\_out = creat(outfile, 0755)) == -1) {

fprintf(stderr, "#error: redirect standard out error\n");

return -1;

}

close(STD\_OUT);

dup(fd\_out);

close(fd\_out);

}

if (type == 3 || type == 5 || type == 6) {

if ((fd\_in = open(infile, O\_RDONLY, S\_IRUSR | S\_IWUSR)) == -1) {

fprintf(stderr, "#error: can't open inputfile '%s'\n", infile);

return -1;

}

close(STD\_IN);

dup(fd\_in);

close(fd\_in);

}

程序命名为myShell.c

编译命令 gcc myShell.c -o myShell -lreadline -ltermcap

**正确编译下面的代码需要手动引入readline库和termcap库**

引入步骤：

下载这两个压缩文件至/usr/src

安装readline库

su root

cd /usr/src

tar -zxvf readline-5.1.tar.gz

cd readline-5.1

./configure

make

make install

安装termcap库

su root

cd /usr/src

tar -zxvf termcap-1.3.1.tar.gz

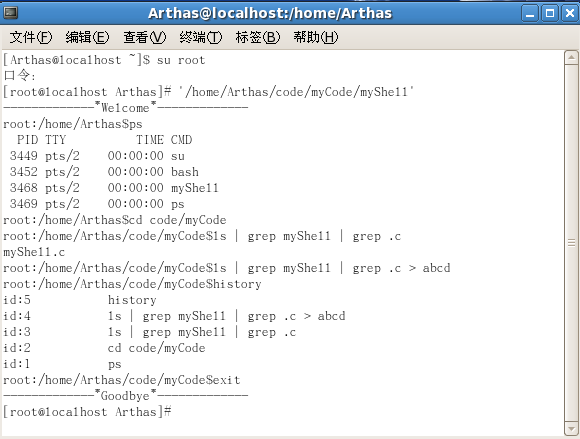
cd termcap-1.3.1

./configure

make

make install

执行myShell



友情提示：做这些实验一定要细心，确认无误再编译。等待编译要耐心，一次大概1.5小时左右。出现问题先百度，基本都有，祝实验顺利！

# 参考文献：

<http://blog.sina.com.cn/s/blog_6240b5980100ldiy.html>

<http://blog.chinaunix.net/uid-14735472-id-2930265.html>

http://soft.chinabyte.com/os/368/11655868.shtml

<http://blog.csdn.net/sanbailiushiliuye/article/details/7561869>

<http://blog.163.com/tianhail@126/blog/static/140395915201022344656633/>

<http://www.cppblog.com/csjiaxin/articles/136378.html>

<http://wenku.baidu.com/link?url=fd7H4VmJwxjf2MsnA1dVRAEWZD3oD2GRMtcn5s3WzOMzji_oNPYQ63_KgivTa7DifIDa0IU3JmOILqe2zavQ-UlIRB9jKRijQt56KI_Zi-q>

<http://blog.csdn.net/mtv0312/article/details/6183583>