Linux 进程管理及其扩展

Author：57119108 吴桐
日期：2021.7.14

内核生成

1、生成内核配置文件

将当前正在运行的内核对应的配置文件作为模板来生成.config 文件，即将/boot 目录下的已有的 config 文件复制到 linux-2.6.21 目录下，重命名为.config。

```
$ make mrproper
```

命令 make mrproper 用来保证内核树是干净的。

更新 config 文件：

```
$ make oldconfig
```

部分新配置项会提示用户选择，都选 N 或者缺省即可，完成后即可生成.config 文件。

2、编译安装内核

在编译内核前，可以定义自己的内核版本号，在内核代码的根目录下有 Makefile 文件，例如将第 4 行改为：

```
EXTRAVERSION = -seu
```

这样新内核版本号就是 2.6.21-seu

```
$ make all
$ su
# make modules_install
# make install
# make headers_install
```

如果三个命令均成功执行，可以观察引导程序 grub 的配置文件/boot/grub/menu.lst 的内容，在 hiddenmenu 之后出现刚刚编译安装的内核版本，将 hiddenmenu 那一行注释或删除，方便直接操作菜单：

```
#hiddenmenu 或者 hiddenmenu
```

然后重启系统：

```
# reboot
```

重启后可以看到 grub 菜单已经包含了新编译的内核。如果新内核启动失败，一般是由于配置或者内核代码修改的有问题，选择原先的内核启动，再进行修改、编译。

新增系统调用 hide

1、设置标识

通过设置一个标记位来控制进程是否隐藏。在 include/linux/sched.h 中修改结构体 task_struct，添加一个成员 cloak，0 表示显示，1 表示隐藏。

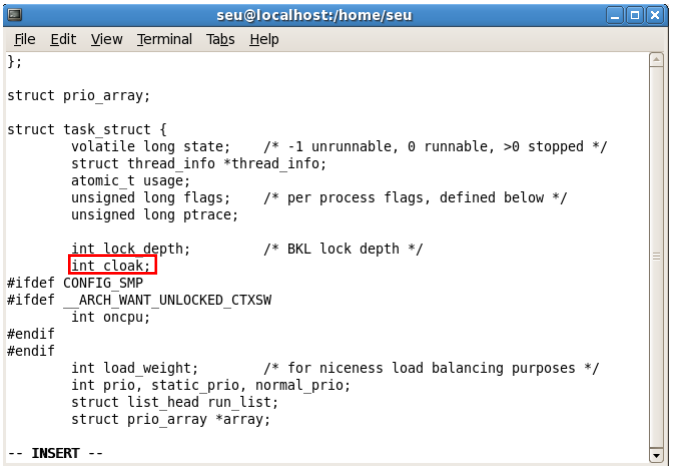

图 1

2、初始化

在进程创建时，将 task_struct 的成员 cloak 初始化为显示（0）。fork 系统调用的实现代码在 kernel/fork.c 中，具体实现的主要函数为 do_fork，do_fork 中调用 copy_process 函数创建子进程，建议将初始化 cloak 的代码添加在 copy_process 函数中。

```
retval = -ENOMEM;
p = dup_task_struct(current);
if (!p)
        goto fork_out;

p->cloak = 0;
/*
 * new content
 */
```

图 2

3、添加 hide 系统调用

在 fs/proc/base.c 中添加 hide 系统调用。通过 pid 获取进程 task_struct 的内核函数为 find_task_by_pid。在隐藏后最好调用函数 proc_flush_task 来清空 VFS 层的缓冲，解除已有的 dentry 项。

```
asmlinkage int sys_hide(pid_t pid, int on) {
        if (current->uid == 0) {
                struct task_struct *task = find_task_by_pid(pid);
                if (on == 0)
                        task->cloak = 0;
                else
                        task->cloak = 1;
                proc_flush_task(task);
        }
        return 0;
}
```

图 3

4、过滤隐藏进程

修改 proc_pid_readdir 函数（在 fs/proc/base.c 文件中）。其中使用 for 循环遍历进程，在遍历过程中添加判断，过滤掉被隐藏的进程。

```
int proc_pid_readdir(struct file * filp, void * dirent, filldir_t filldir)
{
        unsigned int nr = filp->f_pos - FIRST_PROCESS_ENTRY;
        struct task_struct *reaper = get_proc_task(filp->f_path.dentry->d_inode);
        struct task_struct *task;
        int tgid;

        if (!reaper)
                goto out_no_task;

        for (; nr < ARRAY_SIZE(proc_base_stuff); filp->f_pos++, nr++) {
                struct pid_entry *p = &proc_base_stuff[nr];
                if (proc_base_fill_cache(filp, dirent, filldir, reaper, p) < 0)
                        goto out;
        }

        tgid = filp->f_pos - TGID_OFFSET;
        for (task = next_tgid(tgid);
             task;
             put_task_struct(task), task = next_tgid(tgid + 1)) {
                tgid = task->pid;
                filp->f_pos = tgid + TGID_OFFSET;
                if (task->cloak == 0 && proc_pid_fill_cache(filp, dirent, filldir, task, tgid) < 0) {
                        put_task_struct(task);
                        goto out;
                }
        }
        filp->f_pos = PID_MAX_LIMIT + TGID_OFFSET;
out:
        put_task_struct(reaper);
out_no_task:
        return 0;
}
```

图 4

修改 proc_pid_lookup 函数，在进程查找完成前过滤掉被隐藏的进程。

```
struct dentry *proc_pid_lookup(struct inode *dir, struct dentry * dentry, struct nameidata *nd)
{
        struct dentry *result = ERR_PTR(-ENOENT);
        struct task_struct *task;
        unsigned tgid;

        result = proc_base_lookup(dir, dentry);
        if (!IS_ERR(result) || PTR_ERR(result) != -ENOENT)
                goto out;

        tgid = name_to_int(dentry);
        if (tgid == ~0U)
                goto out;

        rcu_read_lock();
        task = find_task_by_pid(tgid);
        if (task)
                get_task_struct(task);
        rcu_read_unlock();
        if (!task)
                goto out;

        if (task->cloak == 1)
                goto out;

        result = proc_pid_instantiate(dir, dentry, task, NULL);
        put_task_struct(task);
out:
        return result;
}
```

图 5

新增系统调用 hide_user_processes

1、添加 hide_user_processes 系统调用

在 fs/proc/base.c 中添加 hide_user_processes 系统调用

```
asmlinkage int sys_hide_user_processes(uid_t uid, char *comm, int on) {
        if (current->uid == 0) {
                int t;
                if (on == 0)
                        t = 0;
                else
                        t = 1;
                struct task_struct *task = NULL;
                for_each_process(task) {
                        if (comm == NULL) {
                                if (task->uid == uid) {task->cloak = t;}
                        }
                        else {
                                if (task->uid == uid && task->comm == comm) {task->cloak = t;}
                        }
                }
                proc_flush_task(task);
        }
}
```

图 6

编译内核及测试

在文件 arch/i386/kernel/syscall_table.S 的尾部加上要新增的系统调用函数的名称。

```
        .long sys_hide                  /* 321 */
        .long sys_hide_user_processes                   /* 322 */
```

图 7

在 include/asm-i386/unistd.h 里加上系统调用号的宏定义。

```
#define __NR_hide               321
#define __NR_hide_user_processes 322

#ifdef __KERNEL__

#define NR_syscalls 323
```

图 8

修改 include/linux/syscalls.h，加上函数 sys_psta 的声明。

```
asmlinkage int sys_psta(struct pinfo *buf);
asmlinkage int sys_hide(pid_t pid, int on);
asmlinkage int sys_hide_user_processes(uid_t uid, char *comm, int on);
```

图 9

根据第一部分“内核生成”重新编译安装内核。进入新版本内核，使用以下命令编译测试文件：

```
gcc -o testX testX.c -I/home/seu/Desktop/linux-2.6.21/usr/include
```

使用 root 权限运行可执行文件：

```
sudo ./testX
```

设置 pid=1，on=1，即隐藏 pid=1 的进程，测试程序如图 10 所示，测试结果如图 11 所示。

test2.c

```c
#include <sys/syscall.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    int syscallNum=321;
    pid_t pid=1;
    int on=1;
    syscall(syscallNum,pid,on);
    return 0;
}
```

图 10

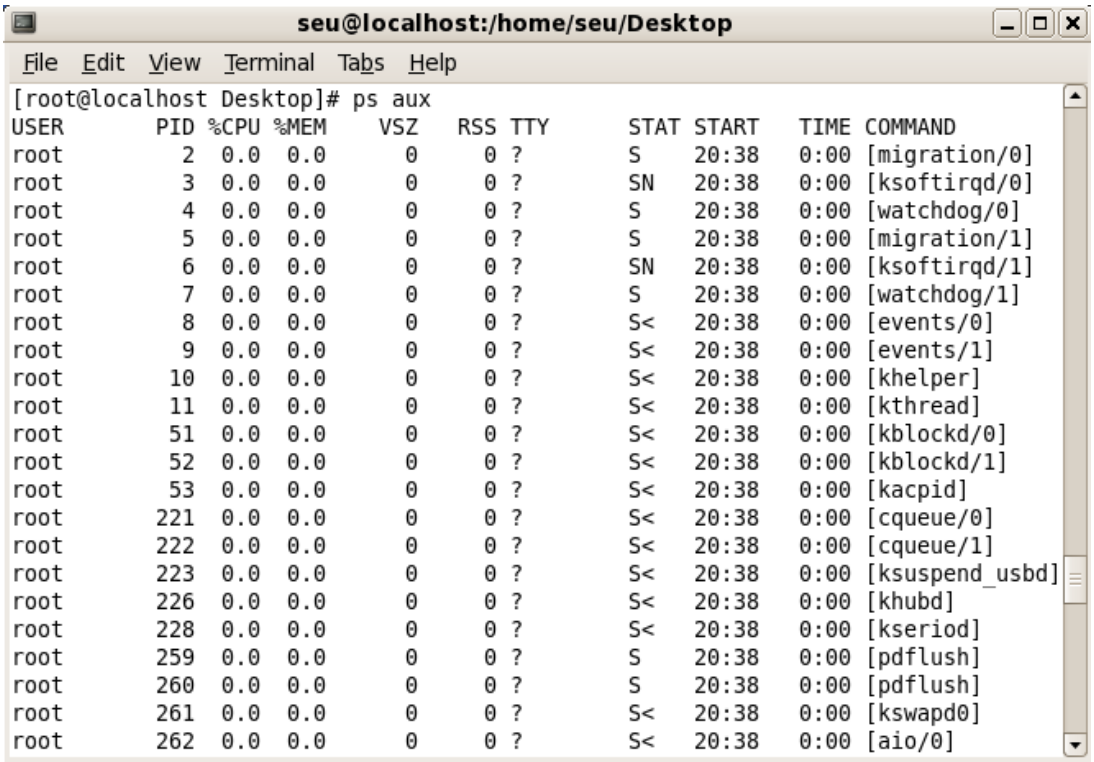

图 11

设置 pid=1，on=0，即显示 pid=1 的进程，测试程序如图 12 所示，测试结果如图 13 所示。

test2.c

```
#include <sys/syscall.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    int syscallNum=321;
    pid_t pid=1;
    int on=0;
    syscall(syscallNum,pid,on);
    return 0;
}
```

图 12

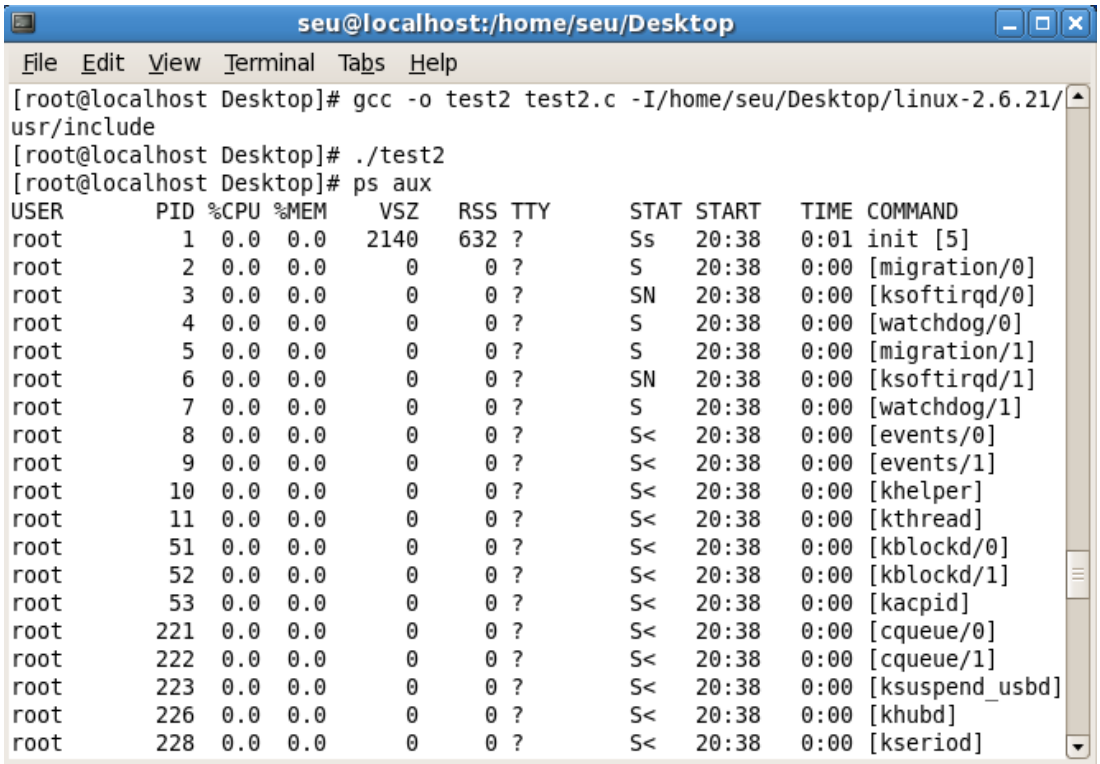

图 13

设置 uid=500，on=1，即隐藏 uid=500（seed 用户）的进程，测试程序如图 15 所示，测试结果对比如图 14 和图 16 所示。

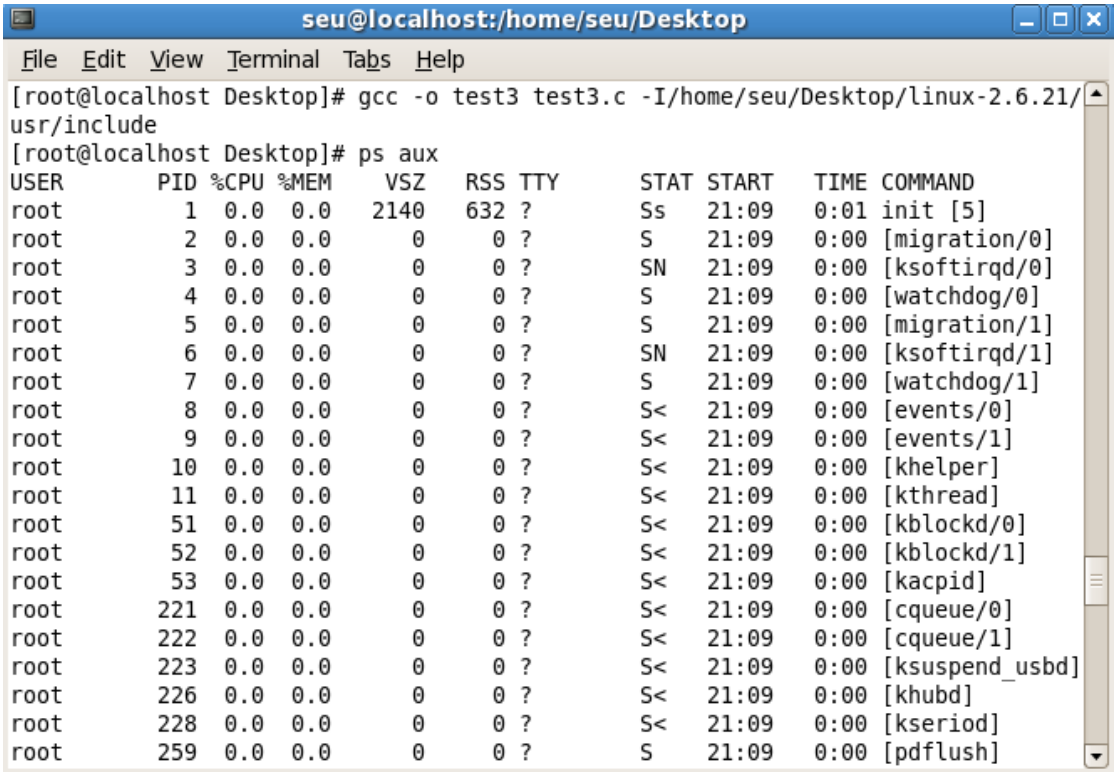

图 14

test3.c

```c
#include <sys/syscall.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    int syscallNum=322;
    uid_t uid=500;
    char *binname=NULL;
    int on=1;
    syscall(syscallNum,uid,binname,on);
    return 0;
}
```

图 15

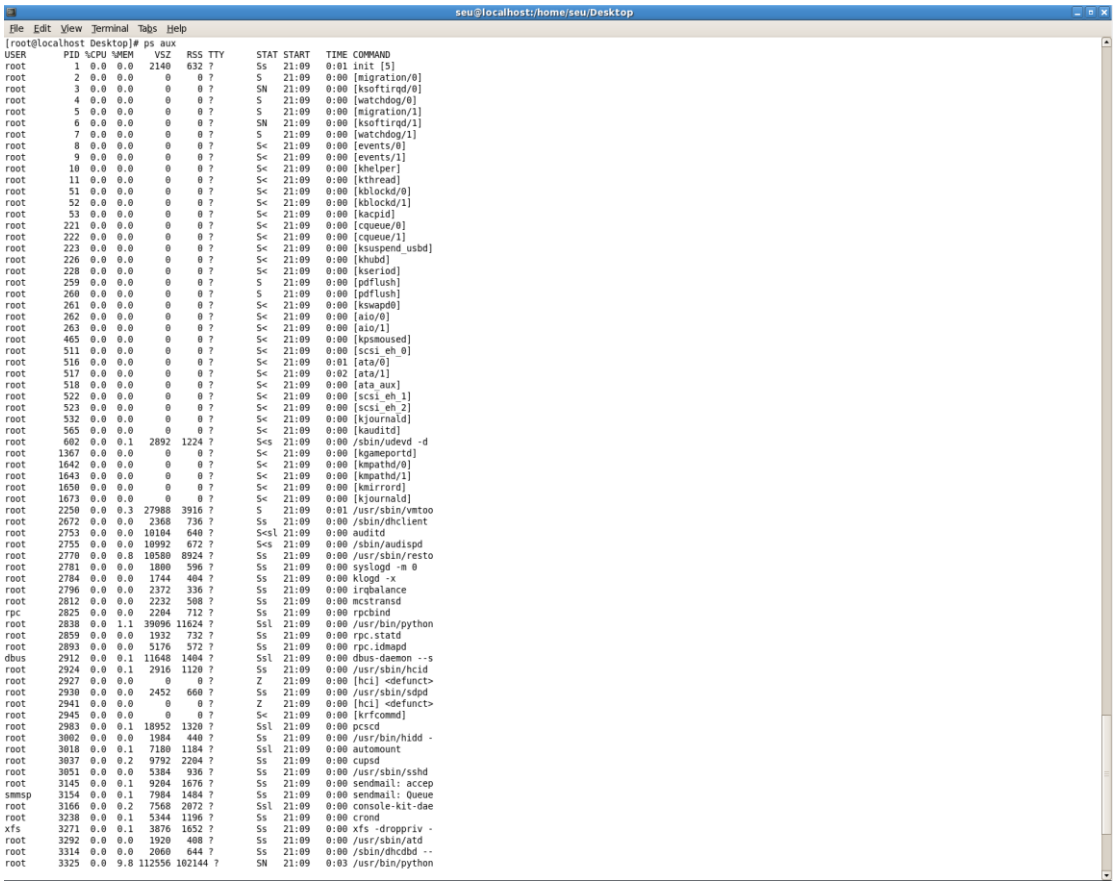

图 16

设置 uid=500，on=0，即显示 uid=500（seed 用户）的进程，测试程序如图 17 所示，测试结果如图 18 所示。

test3.c

```c
#include <sys/syscall.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    int syscallNum=322;
    uid_t uid=500;
    char *binname=NULL;
    int on=0;
    syscall(syscallNum,uid,binname,on);
    return 0;
}
```

图 17

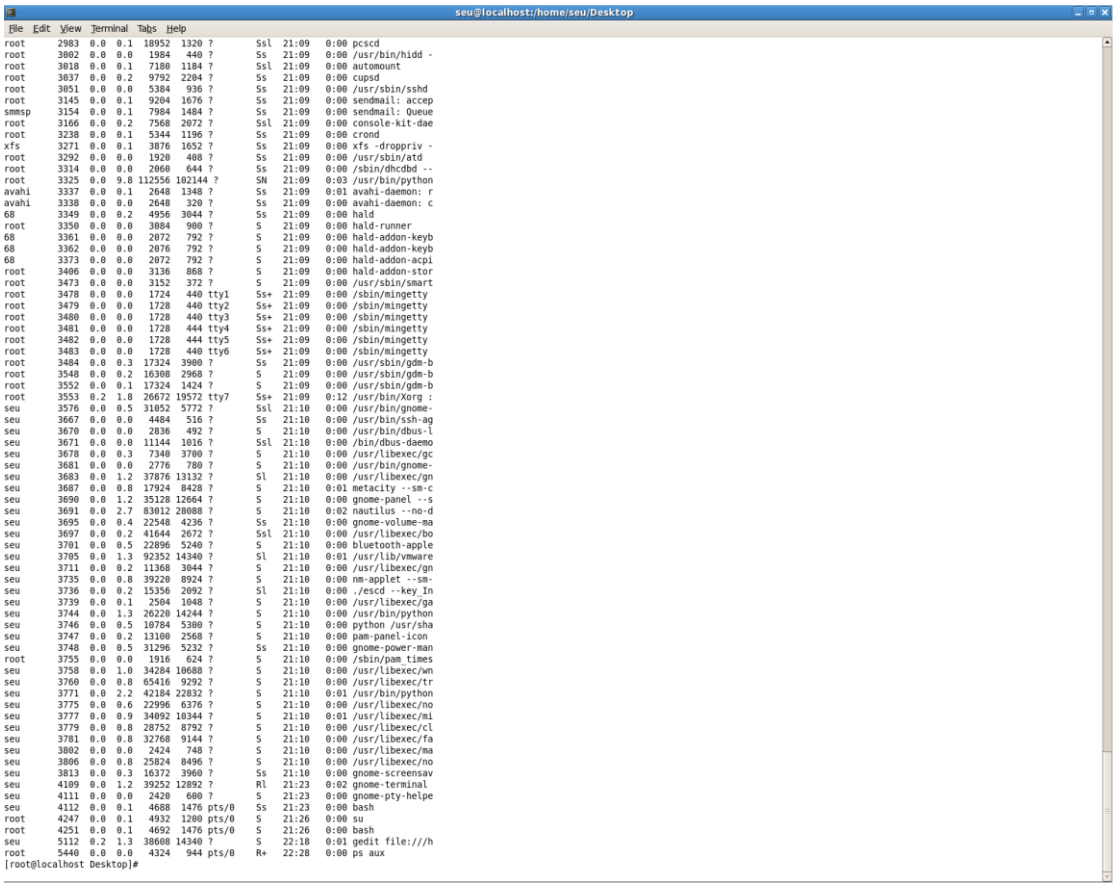

图 18

在/proc 目录下创建一个文件/proc/hidden

1、全局变量 hidden_flag

首先在 fs/proc/proc_misc.c 文件中声明全局变量，EXPORT_SYMBOL()函数可以使该变量在整个内核中可见。使用时只要 extern int hidden_flag;即可访问同一变量。

```
int hidden_flag=0;
EXPORT_SYMBOL(hidden_flag);
```

图 19

2、hidden 文件的创建及读写

proc 文件系统在初始化函数 proc_root_init 中会调用 proc_misc_init 函数，此函数用于创建/proc 根目录下的文件，那么将创建 hidden 文件的代码插入到此函数中就可以在 proc 初始化时得到执行。在/fs/proc/proc_misc.c 中添加回调函数，在/fs/proc/proc_misc.c 中 proc_misc_init 函数的最后添加创建 hidden 文件的代码，并指定其回调函数。

```
static int proc_read_hidden(char *page,char **start,off_t off,int count,int *eof,void *data)
{
        int len = 0;
        len=sprintf(page,"%d",hidden_flag);
        return len;
}

static int proc_write_hidden(struct file *file,const char *buffer,unsigned long count,void *data)
{
        int len = 0;
        int BUF_LEN = 128;
        char temp[BUF_LEN];
        if(count >= BUF_LEN)
                len = BUF_LEN - 1;
        else
                len = count;
        copy_from_user(temp,buffer,len);
        temp[len] = '\0';
        hidden_flag = temp[0] - '0';
        return len;
}
```

图 20

```
void __init proc_misc_init(void)
{
        struct proc_dir_entry *ptr = create_proc_entry("hidden",0644,NULL);

         static struct {
                 char *name;
                 int (*read_proc)(char*,char**,off_t,int,int*,void*);
         } *p, simple_ones[] = {
                 {"loadavg",     loadavg_read_proc},
                 {"uptime",      uptime_read_proc},
                 {"meminfo",     meminfo_read_proc},
                 {"version",     version_read_proc},
#ifdef CONFIG_PROC_HARDWARE
                 {"hardware",    hardware_read_proc},
#endif
```

图 21

```
#ifdef CONFIG_MAGIC_SYSRQ
         {
                 struct proc_dir_entry *entry;
                 entry = create_proc_entry("sysrq-trigger", S_IWUSR, NULL);
                 if (entry)
                         entry->proc_fops = &proc_sysrq_trigger_operations;
         }
#endif

        ptr->read_proc = proc_read_hidden;
        ptr->write_proc = proc_write_hidden;
}
```

图 22

3、根据 hidden_flag 显示/隐藏进程

结合上面根据 cloak 判断进程，这个实验与之类似，只需在 fs/proc/base.c 文件中，修改 proc_pid_readdir 函数以及 proc_pid_lookup 函数，在 cloak 判断之前，增加 hidden_flag 对进程的约束。

```
extern int hidden_flag;
```

图 23

```
/* for the /proc/ directory itself, after non-process stuff has been done */
int proc_pid_readdir(struct file * filp, void * dirent, filldir_t filldir)
{
        unsigned int nr = filp->f_pos - FIRST_PROCESS_ENTRY;
        struct task_struct *reaper = get_proc_task(filp->f_path.dentry->d_inode);
        struct task_struct *task;
        int tgid;

        if (!reaper)
                goto out_no_task;

        for (; nr < ARRAY_SIZE(proc_base_stuff); filp->f_pos++, nr++) {
                struct pid_entry *p = &proc_base_stuff[nr];
                if (proc_base_fill_cache(filp, dirent, filldir, reaper, p) < 0)
                        goto out;
        }

        tgid = filp->f_pos - TGID_OFFSET;
        for (task = next_tgid(tgid);
             task;
             put_task_struct(task), task = next_tgid(tgid + 1)) {
                tgid = task->pid;
                filp->f_pos = tgid + TGID_OFFSET;
                if (hidden_flag==1 && task->cloak == 0 && proc_pid_fill_cache(filp, dirent, filldir, task, tgid) < 0) {
                        put_task_struct(task);
                        goto out;
                }

                if (hidden_flag==0 && proc_pid_fill_cache(filp, dirent, filldir, task, tgid) < 0) {
                        put_task_struct(task);
                        goto out;
                }
        }
        filp->f_pos = PID_MAX_LIMIT + TGID_OFFSET;
out:
        put_task_struct(reaper);
out_no_task:
        return 0;
}
```

图 24

```
struct dentry *proc_pid_lookup(struct inode *dir, struct dentry * dentry, struct nameidata *nd)
{
        struct dentry *result = ERR_PTR(-ENOENT);
        struct task_struct *task;
        unsigned tgid;

        result = proc_base_lookup(dir, dentry);
        if (!IS_ERR(result) || PTR_ERR(result) != -ENOENT)
                goto out;

        tgid = name_to_int(dentry);
        if (tgid == ~0U)
                goto out;

        rcu_read_lock();
        task = find_task_by_pid(tgid);
        if (task)
                get_task_struct(task);
        rcu_read_unlock();
        if (!task)
                goto out;

        if (hidden_flag==1 && task->cloak == 1)
                goto out;

        result = proc_pid_instantiate(dir, dentry, task, NULL);
        put_task_struct(task);
out:
        return result;
}
```

图 25

编译内核及测试

根据第一部分“内核生成”重新编译安装内核。

首先默认设置 hidden_flag=1，使用 hide_user_processes 隐藏 uid=500，即 seu 用户的所有进程。

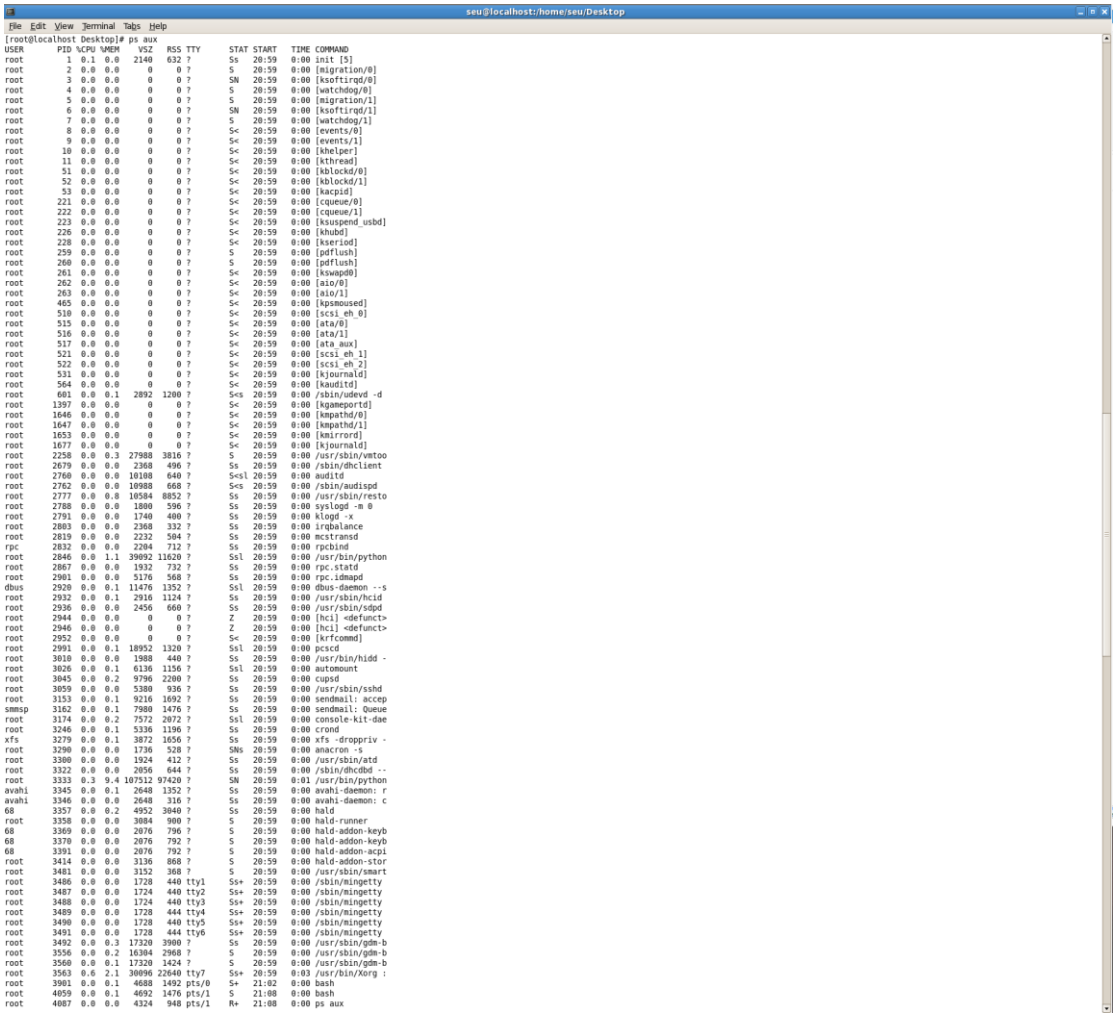

图 26

```
[root@localhost seu]# cd '/home/seu/Desktop/linux-2.6.21'
[root@localhost linux-2.6.21]# cd /proc
```

图 27

```
[root@localhost proc]# echo "0">hidden
```

图 28

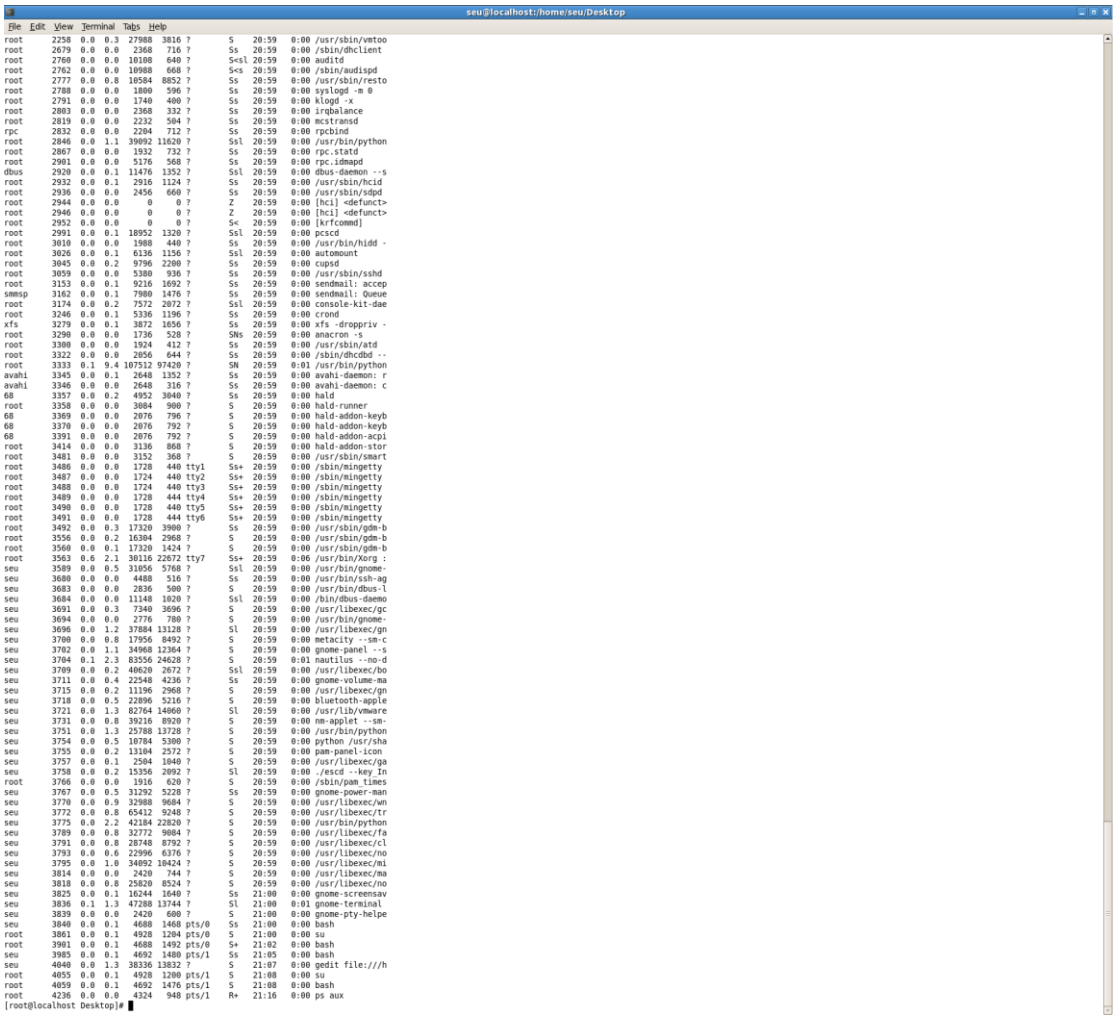
seu@localhost:/home/seu/Desktop
File Edit View Terminal Tabs Help
root 2258 0.0 0.3 27988 3816 ? S 20:59 0:00 /usr/sbin/vmtoo
root 2679 0.0 0.0 2368 716 ? Ss 20:59 0:00 /sbin/dhclient
root 2760 0.0 0.0 10108 640 ? S<sl 20:59 0:00 auditd
root 2762 0.0 0.0 10988 668 ? S<s 20:59 0:00 /sbin/audispd
root 2777 0.0 0.8 10584 8852 ? Ss 20:59 0:00 /usr/sbin/resto
root 2788 0.0 0.0 1800 596 ? Ss 20:59 0:00 syslogd -m 0
root 2791 0.0 0.0 1740 400 ? Ss 20:59 0:00 klogd -x
root 2803 0.0 0.0 2368 332 ? Ss 20:59 0:00 irqbalance
root 2819 0.0 0.0 2232 504 ? Ss 20:59 0:00 mcstransd
rpc 2832 0.0 0.0 2204 712 ? Ss 20:59 0:00 rpcbind
root 2846 0.0 1.1 39092 11620 ? Ssl 20:59 0:00 /usr/bin/python
root 2867 0.0 0.0 1932 732 ? Ss 20:59 0:00 rpc.statd
root 2901 0.0 0.0 5176 568 ? Ss 20:59 0:00 rpc.idmapd
dbus 2920 0.0 0.1 11476 1352 ? Ssl 20:59 0:00 dbus-daemon --s
root 2932 0.0 0.1 2916 1124 ? Ss 20:59 0:00 /usr/sbin/hcid
root 2936 0.0 0.0 2456 660 ? Ss 20:59 0:00 /usr/sbin/sdpd
root 2944 0.0 0.0 0 0 ? Z 20:59 0:00 [hci] <defunct>
root 2946 0.0 0.0 0 0 ? Z 20:59 0:00 [hci] <defunct>
root 2952 0.0 0.0 0 0 ? S< 20:59 0:00 [krfcommd]
root 2991 0.0 0.1 18952 1320 ? Ssl 20:59 0:00 pcscd
root 3010 0.0 0.0 1988 440 ? Ss 20:59 0:00 /usr/bin/hidd -
root 3026 0.0 0.1 6136 1156 ? Ssl 20:59 0:00 automount
root 3045 0.0 0.2 9796 2200 ? Ss 20:59 0:00 cupsd
root 3059 0.0 0.0 5380 936 ? Ss 20:59 0:00 /usr/sbin/sshd
root 3153 0.0 0.1 9216 1692 ? Ss 20:59 0:00 sendmail: accep
smmsp 3162 0.0 0.1 7980 1476 ? Ss 20:59 0:00 sendmail: Queue
root 3174 0.0 0.2 7572 2072 ? Ssl 20:59 0:00 console-kit-dae
root 3246 0.0 0.1 5336 1196 ? Ss 20:59 0:00 crond
xfs 3279 0.0 0.1 3872 1656 ? Ss 20:59 0:00 xfs -droppriv -
root 3290 0.0 0.0 1736 528 ? SNs 20:59 0:00 anacron -s
root 3300 0.0 0.0 1924 412 ? Ss 20:59 0:00 /usr/sbin/atd
root 3322 0.0 0.0 2056 644 ? Ss 20:59 0:00 /sbin/dhcdbd --
root 3333 0.1 9.4 107512 97420 ? SN 20:59 0:01 /usr/bin/python
avahi 3345 0.0 0.1 2648 1352 ? Ss 20:59 0:00 avahi-daemon: r
avahi 3346 0.0 0.0 2648 316 ? Ss 20:59 0:00 avahi-daemon: c
68 3357 0.0 0.2 4952 3040 ? Ss 20:59 0:00 hald
root 3358 0.0 0.0 3084 900 ? S 20:59 0:00 hald-runner
68 3369 0.0 0.0 2076 796 ? S 20:59 0:00 hald-addon-keyb
68 3370 0.0 0.0 2076 792 ? S 20:59 0:00 hald-addon-keyb
68 3391 0.0 0.0 2076 792 ? S 20:59 0:00 hald-addon-acpi
root 3414 0.0 0.0 3136 868 ? S 20:59 0:00 hald-addon-stor
root 3481 0.0 0.0 3152 368 ? S 20:59 0:00 /usr/sbin/smart
root 3486 0.0 0.0 1728 440 tty1 Ss+ 20:59 0:00 /sbin/mingetty
root 3487 0.0 0.0 1724 440 tty2 Ss+ 20:59 0:00 /sbin/mingetty
root 3488 0.0 0.0 1724 440 tty3 Ss+ 20:59 0:00 /sbin/mingetty
root 3489 0.0 0.0 1728 444 tty4 Ss+ 20:59 0:00 /sbin/mingetty
root 3490 0.0 0.0 1728 440 tty5 Ss+ 20:59 0:00 /sbin/mingetty
root 3491 0.0 0.0 1728 444 tty6 Ss+ 20:59 0:00 /sbin/mingetty
root 3492 0.0 0.3 17320 3900 ? Ss 20:59 0:00 /usr/sbin/gdm-b
root 3556 0.0 0.2 16304 2968 ? S 20:59 0:00 /usr/sbin/gdm-b
root 3560 0.0 0.1 17320 1424 ? S 20:59 0:00 /usr/sbin/gdm-b
root 3563 0.6 2.1 30116 22672 tty7 Ss+ 20:59 0:06 /usr/bin/Xorg :
seu 3589 0.0 0.5 31056 5768 ? Ssl 20:59 0:00 /usr/bin/gnome-
seu 3680 0.0 0.0 4488 516 ? Ss 20:59 0:00 /usr/bin/ssh-ag
seu 3683 0.0 0.0 2836 500 ? S 20:59 0:00 /usr/bin/dbus-l
seu 3684 0.0 0.0 11148 1020 ? Ssl 20:59 0:00 /bin/dbus-daemo
seu 3691 0.0 0.3 7340 3696 ? S 20:59 0:00 /usr/libexec/gc
seu 3694 0.0 0.0 2776 780 ? S 20:59 0:00 /usr/bin/gnome-
seu 3696 0.0 1.2 37884 13128 ? Sl 20:59 0:00 /usr/libexec/gn
seu 3700 0.0 0.8 17956 8492 ? S 20:59 0:00 metacity --sm-c
seu 3702 0.0 1.1 34968 12364 ? S 20:59 0:00 gnome-panel --s
seu 3704 0.1 2.3 83556 24628 ? S 20:59 0:01 nautilus --no-d
seu 3709 0.0 0.2 40620 2672 ? Ssl 20:59 0:00 /usr/libexec/bo
seu 3711 0.0 0.4 22548 4236 ? Ss 20:59 0:00 gnome-volume-ma
seu 3715 0.0 0.2 11196 2968 ? S 20:59 0:00 /usr/libexec/gn
seu 3718 0.0 0.5 22896 5216 ? S 20:59 0:00 bluetooth-apple
seu 3721 0.0 1.3 82764 14060 ? Sl 20:59 0:00 /usr/lib/vmware
seu 3731 0.0 0.8 39216 8920 ? S 20:59 0:00 nm-applet --sm-
seu 3751 0.0 1.3 25788 13728 ? S 20:59 0:00 /usr/bin/python
seu 3754 0.0 0.5 10784 5300 ? S 20:59 0:00 python /usr/sha
seu 3755 0.0 0.2 13104 2572 ? S 20:59 0:00 pam-panel-icon
seu 3757 0.0 0.1 2504 1040 ? S 20:59 0:00 /usr/libexec/ga
seu 3758 0.0 0.2 15356 2092 ? Sl 20:59 0:00 ./escd --key_In
root 3766 0.0 0.0 1916 620 ? S 20:59 0:00 /sbin/pam_times
seu 3767 0.0 0.5 31292 5228 ? Ss 20:59 0:00 gnome-power-man
seu 3770 0.0 0.9 32988 9684 ? S 20:59 0:00 /usr/libexec/wn
seu 3772 0.0 0.8 65412 9248 ? S 20:59 0:00 /usr/libexec/tr
seu 3775 0.0 2.2 42184 22820 ? S 20:59 0:00 /usr/bin/python
seu 3789 0.0 0.8 32772 9084 ? S 20:59 0:00 /usr/libexec/fa
seu 3791 0.0 0.8 28748 8792 ? S 20:59 0:00 /usr/libexec/cl
seu 3793 0.0 0.6 22996 6376 ? S 20:59 0:00 /usr/libexec/no
seu 3795 0.0 1.0 34092 10424 ? S 20:59 0:00 /usr/libexec/mi
seu 3814 0.0 0.0 2420 744 ? S 20:59 0:00 /usr/libexec/ma
seu 3818 0.0 0.8 25820 8524 ? S 20:59 0:00 /usr/libexec/no
seu 3825 0.0 0.1 16244 1640 ? Ss 21:00 0:00 gnome-screensav
seu 3836 0.1 1.3 47288 13744 ? Sl 21:00 0:01 gnome-terminal
seu 3839 0.0 0.0 2420 600 ? S 21:00 0:00 gnome-pty-helpe
seu 3840 0.0 0.1 4688 1468 pts/0 Ss 21:00 0:00 bash
root 3861 0.0 0.1 4928 1204 pts/0 S 21:00 0:00 su
root 3901 0.0 0.1 4688 1492 pts/0 S+ 21:02 0:00 bash
seu 3985 0.0 0.1 4692 1480 pts/1 Ss 21:05 0:00 bash
seu 4040 0.0 1.3 38336 13832 ? S 21:07 0:00 gedit file:///h
root 4055 0.0 0.1 4928 1200 pts/1 S 21:08 0:00 su
root 4059 0.0 0.1 4692 1476 pts/1 S 21:08 0:00 bash
root 4236 0.0 0.0 4324 948 pts/1 R+ 21:16 0:00 ps aux
[root@localhost Desktop]#

图 29

```
[root@localhost proc]# echo "1">hidden
```

图 30

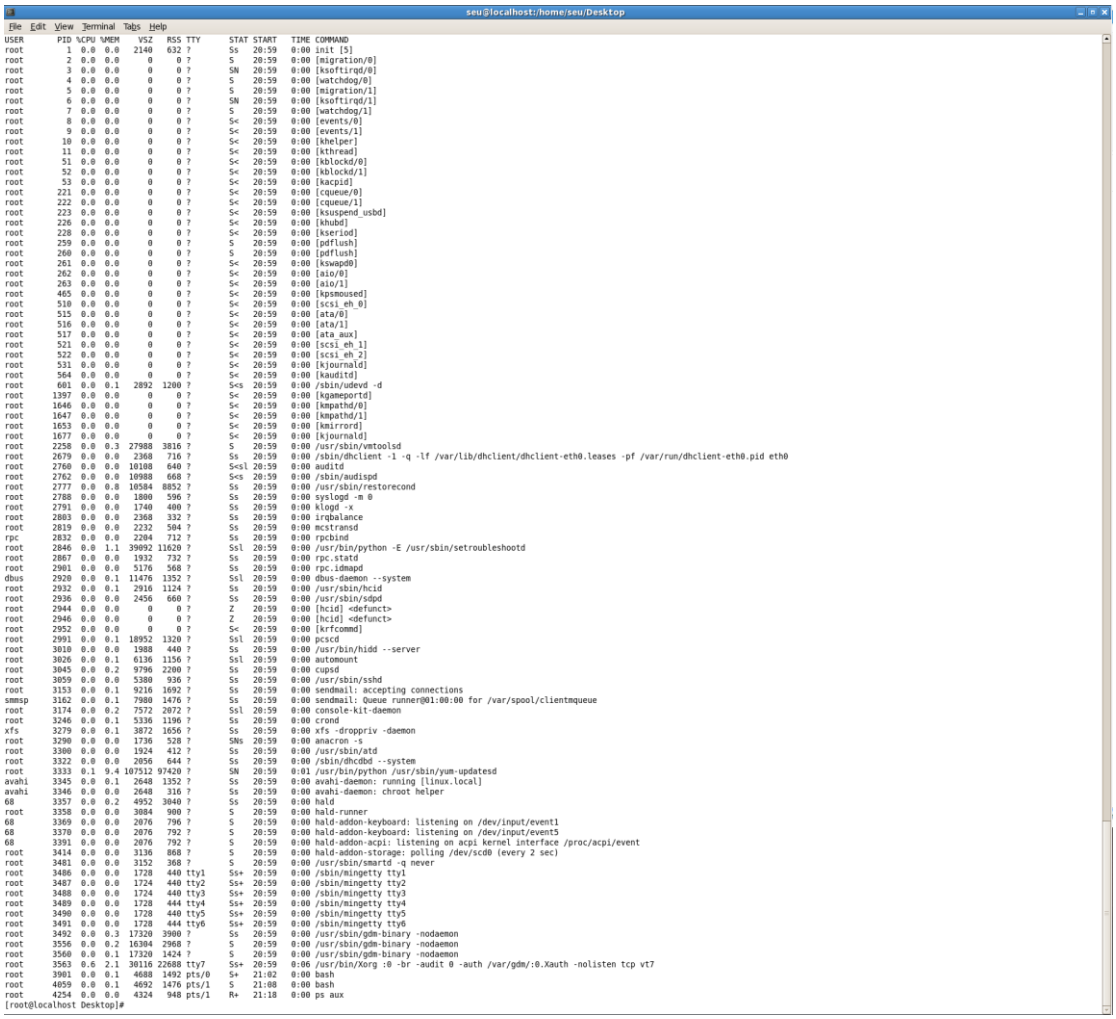

图 31

在/proc 目录下创建一个文件/proc/hidden_process

1、hidden_process 文件的读回调函数

在/fs/proc/proc_misc.c 中添加回调函数，在/fs/proc/proc_misc.c 中 proc_misc_init 函数的最后添加创建 hidden 文件的代码，并指定其回调函数。

```
static int proc_read_hidden_processes(char *page,char **start,off_t off,int count,int *eof,void *data)
{
        static char buf[1024*8]="";
        char tmp[128];
        struct task_struct *p;
        if (off>0) return 0;
        sprintf(buf,"%s","");
        for_each_process(p)
        {
                if (p->cloak==1)
                {
                        sprintf(tmp,"%d ",p->pid);
                        strcat(buf,tmp);
                }
        }
        sprintf(page,"%s",buf);
        return strlen(buf);
}
```

图 32

```
void __init proc_misc_init(void)
{
        struct proc_dir_entry *ptr = create_proc_entry("hidden",0644,NULL);
        struct proc_dir_entry *myptr = create_proc_entry("hidden_process",0644,NULL);

        static struct {
                char *name;
                int (*read_proc)(char*,char**,off_t,int,int*,void*);
        } *p, simple_ones[] = {
                {"loadavg",     loadavg_read_proc},
                {"uptime",      uptime_read_proc},
                {"meminfo",     meminfo_read_proc},
                {"version",     version_read_proc},
#ifdef CONFIG_PROC_HARDWARE
                {"hardware",    hardware_read_proc},
#endif
```

图 33

```
    #ifdef CONFIG_MAGIC_SYSRQ
            {
                    struct proc_dir_entry *entry;
                    entry = create_proc_entry("sysrq-trigger", S_IWUSR, NULL);
                    if (entry)
                            entry->proc_fops = &proc_sysrq_trigger_operations;
            }
    #endif

            ptr->read_proc = proc_read_hidden;
            ptr->write_proc = proc_write_hidden;
            myptr->read_proc = proc_read_hidden_processes;
    }
```

图 34

编译内核及测试

根据第一部分“内核生成”重新编译安装内核。

首先默认设置 hidden_flag=1，使用 hide_user_processes 隐藏 uid=500，即 seu 用户的所有进程。

```
[root@localhost proc]# echo "1">hidden
[root@localhost proc]# cat hidden_process
3587 3678 3681 3682 3689 3692 3694 3698 3701 3702 3706 3708 3710 3715 3739 3743
3745 3746 3747 3748 3749 3757 3761 3768 3773 3780 3788 3790 3792 3794 3813 3817
3828 3864 3867 3868 3888 3982 4013 [root@localhost proc]#
```

图 35

实验总结

本次实验的难度本身不是很大，但由于实验步骤比较复杂，需要极大的耐心和毅力。一开始进行实验的时候，进展比较慢，过程中也遇到了一些小问题。问题虽小，但是致命且浪费时间。不过随着实验的推进，我逐渐熟悉了实验操作，也理解了实验背后的原理，最后实验如期完成了。感谢助教和同学们的帮助，指导我完成实验，让我对 Linux 进程管理有了更深的理解。