东南大学计算机科学与工程学院

操作系统专题实践

实验报告1：Linux进程管理及其扩展

|  |  |
| --- | --- |
| 学号： | 09019106 |
| 姓名： | 牟倪 |
| 完成日期： | 2021年10月8日 |

目录

[一、实验内容与目的 3](#_Toc85054847)

[1.1 实验内容 3](#_Toc85054848)

[1.2 实验目的 3](#_Toc85054849)

[二、设计思路与流程图 3](#_Toc85054850)

[2.1 得到Linux内核源码 3](#_Toc85054851)

[2.2 添加hide系统调用 4](#_Toc85054852)

[2.3 添加hide\_user\_process系统调用 4](#_Toc85054853)

[2.4 实现用hidden\_flag控制进程是否隐藏 5](#_Toc85054854)

[2.5 实现在hidden\_process文件记录被隐藏进程的pid 5](#_Toc85054855)

[2.6 配置内核 5](#_Toc85054856)

[三、主要数据结构及说明 6](#_Toc85054857)

[四、源程序 6](#_Toc85054858)

[五、程序运行结果 13](#_Toc85054859)

[5.1 测试hide系统调用 13](#_Toc85054860)

[5.2 测试hide\_user\_process系统调用 14](#_Toc85054861)

[5.3 测试用hidden\_flag控制进程是否隐藏 15](#_Toc85054862)

[5.4 测试在hidden\_process文件记录被隐藏进程的pid 15](#_Toc85054863)

[六、实验体会 15](#_Toc85054864)

# 一、实验内容与目的

## 1.1 实验内容

实现系统调用hide，该系统调用可以根据指定的参数隐藏进程，使用户无法使用ps或top观察到进程状态。

* 实现系统调用int hide(pid\_t pid, int on)。在进程pid有效的前提下，如果on置1，进程被隐藏，用户无法通过ps或top观察到进程状态；如果on置0且此前为隐藏状态，则恢复正常状态。
* 设计一个新的系统调用int hide\_user\_processes(uid\_t uid, char \*binname)， 参数uid为用户ID号，当binname参数为NULL时，隐藏该用户的所有进程；否则，隐藏二进制映像名为binname的用户进程。该系统调用应与hide系统调用共存。
* 考虑权限问题，只有root用户才能隐藏进程。
* 在/proc目录下创建一个文件/proc/hidden，该文件可读可写，对应一个全局变量hidden\_flag，当 hidden\_flag为0时，所有进程都无法隐藏，即便此前进程被hide系统调用要求隐藏。只有当hidden\_flag为 1时，此前通过hide调用要求被屏蔽的进程才隐藏起来。
* 在/proc目录下创建一个文件/proc/hidden\_process， 该文件的内容包含所有被隐藏进程的pid，各pid之间用空格分开。

## 1.2 实验目的

加深对进程控制块（PCB）、进程队列等概念的理解，了解Linux对进程管理的具体实施方法。

# 二、设计思路与流程图

## 2.1 得到Linux内核源码

* 访问https://mirrors.edge.kernel.org/pub/linux/kernel，下载一份内核源码。我下载的版本是2.6.21。
* 将源码解压到usr/src。

## 2.2 添加hide系统调用

* 在PCB中添加一个标记位，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。
* 进程刚创建时处于显示状态，所以需要将cloak初始化为0。fork系统调用的实现位于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，在该函数中初始化cloak。
* 添加hide系统调用。
  + 在kernel/sys.C中，书写hide系统调用的逻辑。首先判断是否为root用户，然后根据pid找到进程，修改相应进程的cloak位。
  + 修改include/asm-i386/unistd.h，添加新系统调用的id宏定义。
  + 修改arch/i386/kernel/syscall\_table.S、include/linux/syscalls.h，添加新系统调用的声明。
* 修改fs/proc/base.c中的proc\_pid\_readdir函数以及proc\_pid\_lookup函数，过滤掉被隐藏的进程。

## 2.3 添加hide\_user\_process系统调用

* 考虑到隐藏某一用户所有进程后，很难通过hide系统调用一个一个恢复显示状态，我们为hide\_user\_process系统调用添加第三个参数recover，recover为0时执行进程的隐藏，recover为1时恢复所有进程为显示状态。这样，就可以批量恢复进程的显示状态了。
* 在kernel/sys.C中，书写hide\_user\_process系统调用的逻辑。
  + 首先，判断是否为root用户。
  + 使用函数for\_each\_process遍历所有进程，每个进程的task\_struct中有成员变量uid和comm，uid为该进程的用户id，comm为进程名。
  + 若recover==1，置所有进程的cloak为0。
  + 若参数binname为NULL，隐藏该用户所有进程。
  + 否则，使用strcmp精确查找进程名为binname的进程，隐藏该进程。
* 修改include/asm-i386/unistd.h，添加新系统调用的id宏定义。
* 修改arch/i386/kernel/syscall\_table.S、include/linux/syscalls.h，添加新系统调用的声明。

## 2.4 实现用hidden\_flag控制进程是否隐藏

* 创建fs/proc/hidden文件。
  + fs/proc部分的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fs/proc根目录下的文件。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。首先初始化hidden\_flag为1；读操作中sprintf将hidden\_flag的值传给page，然后显示出来；写操作中使用copy\_from\_user先将用户输入的值传给temp，然后从temp中得到值传给hidden\_flag。
  + 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。使用create\_proc\_entry函数创建hidden文件。
* 设置全局变量hidden\_flag。在fs/proc目录下新建头文件var.h，里面包含全局变量hidden\_flag。其它文件中需要用到hidden\_flag时，需使用include包含这个头文件。
* 根据hidden\_flag显示/隐藏进程。再次修改fs/proc/base.c文件，包含头文件var.h；并修改函数proc\_pid\_readdir和proc\_pid\_lookup，在判断cloak之前先判断hidden\_flag。

## 2.5 实现在hidden\_process文件记录被隐藏进程的pid

* 创建fs/proc/hidden\_process文件。
  + 将创建hidden文件的代码插入到proc\_misc\_init函数中。
  + 在fs/proc/proc\_misc.c中添加回调函数。由于该文件不会被用户写入，所以只需考虑读回调函数。
  + 在fs/proc/proc\_misc.c中proc\_misc\_init函数的最后添加创建hidden\_process文件的代码，并指定其回调函数。

## 2.6 配置内核

执行以下命令：

* cd /usr/src/linux-2.6.21
* sudo make menuconfig

# 三、主要数据结构及说明

* 结构体task\_struct：添加cloak变量，控制进程是否隐藏。

# 四、源程序

include/linux/sched.h，结构体task\_struct：

1. struct task\_struct {
2. volatile long state;    */\* -1 unrunnable, 0 runnable, >0 stopped \*/*
3. struct thread\_info \*thread\_info;
4. atomic\_t usage;
5. unsigned long flags;    */\* per process flags, defined below \*/*
6. unsigned long ptrace;
7. int lock\_depth;         */\* BKL lock depth \*/*
8. int cloak;

kernel/fork.c，函数copy\_process（片段）：

1. static struct task\_struct \*copy\_process(unsigned long clone\_flags,
2. unsigned long stack\_start,
3. struct pt\_regs \*regs,
4. unsigned long stack\_size,
5. int \_\_user \*parent\_tidptr,
6. int \_\_user \*child\_tidptr,
7. int pid)
8. {
9. int retval;
10. struct task\_struct \*p = NULL;
11. if ((clone\_flags & (CLONE\_NEWNS|CLONE\_FS)) == (CLONE\_NEWNS|CLONE\_FS))
12. return ERR\_PTR(-EINVAL);
13. */\**
14. \* Thread groups must share signals as well, and detached threads
15. \* can only be started up within the thread group.
16. \*/
17. if ((clone\_flags & CLONE\_THREAD) && !(clone\_flags & CLONE\_SIGHAND))
18. return ERR\_PTR(-EINVAL);
19. */\**
20. \* Shared signal handlers imply shared VM. By way of the above,
21. \* thread groups also imply shared VM. Blocking this case allows
22. \* for various simplifications in other code.
23. \*/
24. if ((clone\_flags & CLONE\_SIGHAND) && !(clone\_flags & CLONE\_VM))
25. return ERR\_PTR(-EINVAL);
26. retval = security\_task\_create(clone\_flags);
27. if (retval)
28. goto fork\_out;
29. retval = -ENOMEM;
30. p = dup\_task\_struct(current);
31. if (!p)
32. goto fork\_out;
33. p->cloak=0;

fs/proc/base.c，函数proc\_pid\_readdir：

1. int proc\_pid\_readdir(struct file \* filp, void \* dirent, filldir\_t filldir)
2. {
3. unsigned int nr = filp->f\_pos - FIRST\_PROCESS\_ENTRY;
4. struct task\_struct \*reaper = get\_proc\_task(filp->f\_path.dentry->d\_inode);
5. struct task\_struct \*task;
6. int tgid;
7. if (!reaper)
8. goto out\_no\_task;
9. for (; nr < ARRAY\_SIZE(proc\_base\_stuff); filp->f\_pos++, nr++) {
10. struct pid\_entry \*p = &proc\_base\_stuff[nr];
11. if (proc\_base\_fill\_cache(filp, dirent, filldir, reaper, p) < 0)
12. goto out;
13. }
14. tgid = filp->f\_pos - TGID\_OFFSET;
15. for (task = next\_tgid(tgid);
16. task;
17. put\_task\_struct(task), task = next\_tgid(tgid + 1)) {
18. tgid = task->pid;
19. filp->f\_pos = tgid + TGID\_OFFSET;
20. if (hidden\_flag==1 && task->cloak==0 && proc\_pid\_fill\_cache(filp, dirent, filldir, task, tgid) < 0) {
21. put\_task\_struct(task);
22. goto out;
23. }
24. }
25. filp->f\_pos = PID\_MAX\_LIMIT + TGID\_OFFSET;
26. out:
27. put\_task\_struct(reaper);
28. out\_no\_task:
29. return 0;
30. }

fs/proc/base.c，函数proc\_pid\_lookup：

1. struct dentry \*proc\_pid\_lookup(struct inode \*dir, struct dentry \* dentry, struct nameidata \*nd)
2. {
3. struct dentry \*result = ERR\_PTR(-ENOENT);
4. struct task\_struct \*task;
5. unsigned tgid;
6. result = proc\_base\_lookup(dir, dentry);
7. if (!IS\_ERR(result) || PTR\_ERR(result) != -ENOENT)
8. goto out;
9. tgid = name\_to\_int(dentry);
10. if (tgid == ~0U)
11. goto out;
12. rcu\_read\_lock();
13. task = find\_task\_by\_pid(tgid);
14. if (task)
15. get\_task\_struct(task);
16. rcu\_read\_unlock();
17. if (!task)
18. goto out;
19. if(hidden\_flag==1 && task->cloak==1)
20. goto out;
21. result = proc\_pid\_instantiate(dir, dentry, task, NULL);
22. put\_task\_struct(task);
23. out:
24. return result;
25. }

arch/i386/kernel/syscall\_table.S（片段）：

1. .long sys\_tee   */\* 315 \*/*
2. .long sys\_vmsplice
3. .long sys\_move\_pages
4. .long sys\_getcpu
5. .long sys\_epoll\_pwait
6. .long sys\_hide   */\* 320 \*/*
7. .long sys\_hide\_user\_processes */\* 321 \*/*

include/asm-i386/unistd.h（片段）：

1. #define \_\_NR\_hide  320
2. #define \_\_NR\_hide\_user\_processes 321

include/linux/syscalls.h（片段）：

1. asmlinkage long sys\_getcpu(unsigned \_\_user \*cpu, unsigned \_\_user \*node, struct getcpu\_cache \_\_user \*cache);
2. asmlinkage long sys\_hide(pid\_t pid, int on);
3. asmlinkage long sys\_hide\_user\_processes(uid\_t uid, char \*binname, int recover);
4. int kernel\_execve(const char \*filename, char \*const argv[], char \*const envp[]);

kernel/sys.c，函数sys\_hide与函数sys\_hide\_user\_process：

1. asmlinkage long sys\_hide(pid\_t pid, int on)
2. {
3. struct task\_struct \*p=NULL;
4. if(pid>0 && (current->uid)==0) *// are you root?*
5. {
6. p=find\_task\_by\_pid(pid);
7. p->cloak=on; *// set the cloak state of the process*
8. if(on==1)
9. {
10. printk("process %d is hidden~\n",pid);
11. }
12. if(on==0)
13. {
14. printk("process %d is unhidden~\n",pid);
15. }
16. proc\_flush\_task(p);
17. }
18. else
19. printk("sorry you are not root.\n");
20. return 0;
21. }
22. asmlinkage long sys\_hide\_user\_processes(uid\_t uid,char \*binname,int recover){
23. struct task\_struct \*p=NULL;
24. if(current->uid!=0)
25. {
26. printk("sorry you are not root.\n");
27. return 0;
28. }
29. if(recover==0)
30. {
31. if(binname==NULL) *// hide all processes of the corresponding user*
32. {
33. for\_each\_process(p)
34. {
35. if((p->uid)==uid)
36. {
37. p->cloak=1;
38. proc\_flush\_task(p);
39. }
40. }
41. printk("all of the processes of uid %d are hidden~\n",uid);
42. }
43. else *// otherwise, to find the exact name*
44. {
45. for\_each\_process(p)
46. {
47. char \*s=p->comm;
48. if(strcmp(s,binname)==0 && p->uid==uid)
49. {
50. p->cloak=1;
51. printk("process %s of uid %d is hidden~\n",binname,uid);
52. proc\_flush\_task(p);
53. }
54. }
55. }
56. }
57. else if(recover != 0) *// unhidden all the processes*
58. {
59. for\_each\_process(p)
60. {
61. p->cloak=0;
62. }
63. printk("all the processes are unhidden~\n");
64. }
65. return 0;
66. }

fs/proc/proc\_misc.c，函数proc\_misc\_init（片段）：

1. void \_\_init proc\_misc\_init(void)
2. {
3. struct proc\_dir\_entry \*ptr=create\_proc\_entry("hidden",0644,NULL);
4. struct proc\_dir\_entry \*myptr=create\_proc\_entry("hidden\_process",0644,NULL);
5. static struct {
6. char \*name;
7. int (\*read\_proc)(char\*,char\*\*,off\_t,int,int\*,void\*);
8. ......
9. ......
10. ......
11. if (entry)
12. entry->proc\_fops = &proc\_sysrq\_trigger\_operations;
13. }
14. #endif
15. ptr->read\_proc=proc\_read\_hidden;
16. ptr->write\_proc=proc\_write\_hidden;
17. myptr->read\_proc=proc\_read\_hidden\_processes;
18. }

fs/proc/proc\_misc.c，hidden\_flag声明、函数proc\_read\_hidden、proc\_write\_hidden和proc\_read\_hidden\_processes：

1. int hidden\_flag=1;
3. static int proc\_read\_hidden(char \*page,char \*\*start,off\_t off,int count,int \*eof,void \*data)
4. {
5. int len=0;
6. len=sprintf(page,"%d",hidden\_flag);
7. return len;
8. }
10. static int proc\_write\_hidden(struct file \*file,const char \*buffer,unsigned long count,void \*data)
11. {
12. int len=0;
13. char temp[BUF\_LEN];
14. if(count > BUF\_LEN)
15. len = BUF\_LEN;
16. else
17. len = count;
19. copy\_from\_user(temp, buffer, len); *// convert user's input to temp*
20. temp[len]='\0';
21. hidden\_flag=temp[0]-'0'; *// set the value of hidden\_flag*
22. return len;
23. }
24. static int proc\_read\_hidden\_processes(char \*page,char \*\*start,off\_t off,int count,int \*eof,void \*data)
25. {
26. static char buf[1024\*8]="";
27. char tmp[128];
28. struct task\_struct \*p;
29. if(off>0)
30. return 0;
31. sprintf(buf,"%s","");
32. for\_each\_process(p)
33. {
34. if(p->cloak==1)
35. {
36. sprintf(tmp,"%d ",p->pid); *// it's called important*
37. strcat(buf,tmp);
38. }
39. }
40. sprintf(page,"%s",buf); *// convert buf to page to display on the terminal*
41. return strlen(buf);
42. }

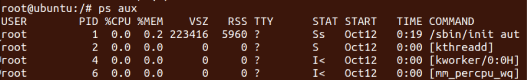
# 五、程序运行结果

## 5.1 测试hide系统调用

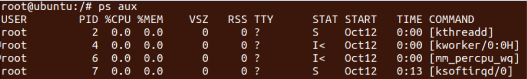
代码如下：

1. #include<stdio.h>
2. #include<sys/syscall.h>
3. #include<unistd.h>
4. int main()
5. {
6. int syscallNum=321;
7. int on=1;
8. pid\_t pid=1;
9. syscall(syscallNum,pid,on);
10. return 0;
11. }

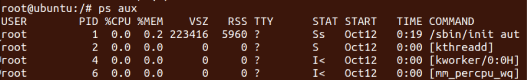
我们希望隐藏1号进程，这是执行前的状态：



执行后：



更改参数on=0，1号进程再次出现：

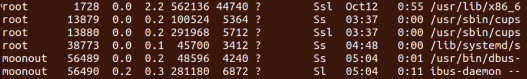


## 5.2 测试hide\_user\_process系统调用

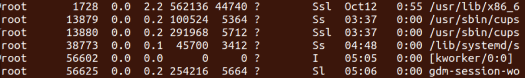
代码如下：

1. #include<stdio.h>
2. #include<sys/syscall.h>
3. #include<unistd.h>
4. int main()
5. {
6. int syscallNum=322;
7. uid\_t uid=500;
8. char \*binname=NULL;
9. int recover=0;
10. syscall(syscallNum,uid,binname,recover);
11. return 0;
12. }

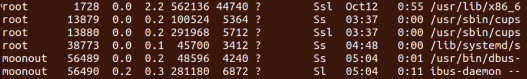
我们希望隐藏用户“moonout”的进程，这是执行前的状态：



执行后：

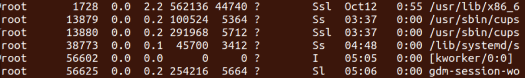


将recover改为1，“moonout”的进程再次出现：

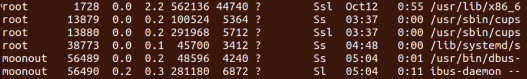


## 5.3 测试用hidden\_flag控制进程是否隐藏

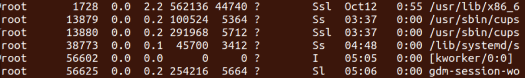
首先，默认设置hidden\_flag=1，使用hide\_user\_processes隐藏用户“moonout”的所有进程。



将hidden\_flag改为0，发现所有被隐藏的进程又出现了。

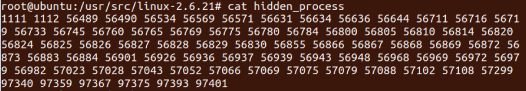


将hidden\_flag改回为1，发现 “moonout”的进程又被隐藏了。



## 5.4 测试在hidden\_process文件记录被隐藏进程的pid

输出hidden\_process文件的内容：



# 六、实验体会

本次实验对我而言，如同打开了新的大门。我愈发感觉操作系统真是博大精深，深深震撼于前人的智慧。然而，理解概念很简单，上手修改内核却是个艰难的过程。我本以为，只要做好足够的调研，把基本逻辑落实到代码，就可以一步一步操作成功。但是，整个实验过程与我的想象大不相同，各种丰富的技术细节超乎了我的预设。对着满屏错误一个一个寻找解决方案，自己试图按照提示信息魔改内核代码，对于一个对Linux实现细节一无所知的人，整个实验过程凌乱而令人自闭，这是在真正去做之前难以想象的。非常幸运的是，我看到了很多相关且高质量的资料与博客，它们指明了实验的方向，给我提供了莫大的帮助。