计算机系统安全课后作业

L170300901 卢兑玩

1、查找资料，列出进程控制相关函数以及系统调用，说明相关函数参数和用法。

：

fork系统调用

函数作用：创建一个子进程

形式：pid_tfork(void);

pid_t vfork(void);

说明：使用vfork创子进程时，不会进程父进程的上下文

返回值：[返回值=-1]子进程创建失败

[返回值=0]子进程创建成功

[返回值>0]对父进程返回子进程PID

```c
#include<stdio.h>
#include<sys/stat.h>
#include<unistd.h>
intmain() {
    pid_t id = fork();
    if (id < 0) {
        perror("子进程创建失败！");
    } else {
        if (id == 0) {
            printf("子进程工作:PID=%d,PPID=%d\n", getpid(), getppid());
        }else
        {
            printf("父进程工作:PID=%d,PPID=%d, 子进程PID=%d\n", getpid(), getppid(),id);
            sleep(5)
        }
    }
}

```

控制台输出

父进程工作:PID=3173,PPID=2432，子进程PID=3176

子进程工作:PID=3176,PPID=3173

exit系统调用

函数作用：终止发出调用的进程

形式：voidexit(int status);

说明

1.exit返回信息可由wait系统函数获得

2.如果父进程先退出子进程的关系被转到init进程下

```c
#include<stdio.h>
#include<sys/stat.h>
#include<unistd.h>
#include<stdlib.h>
intmain() {
    pid_t id = fork();
    if (id < 0) {
        perror("子进程创建失败！");
    } else {
        if (id == 0) {
            printf("子进程工作:PID=%d,PPID=%d\n", getpid(), getppid());
            sleep(20);
            printf("此时子进程:PID=%d,PPID=%d\n", getpid(), getppid());
        }else
        {
            printf("父进程工作:PID=%d,PPID=%d, 子进程PID=%d\n", getpid(), getppid(),id)
            sleep(5);
            exit(3);
        }
    }
    return 0;
}
```

控制台输出

父进程工作:PID=3068,PPID=2432，子进程PID=3071

子进程工作:PID=3071,PPID=3068

此时子进程:PID=3071,PPID=1

wait系统调用

函数作用：父进程与子进程同步，父进程调用后。进入睡眠状态，直到子进程结束或者父进程在被其他进程终止，

形式：pid_twait(int *status)

pid_t waitpid(pid_t pid ,int *status,int option)

参数：status：exit是设置的代码

pid:进程号

option: WNOHANG|WUNTRACED

WNOHANG:,即使没有子进程退出,它也会立即返回,不会像wait那样永远等下去.

WUNTRACED:子进程进入暂停则马上返回,但结束状态不予以理会.

返回值：如果成功等待子进程结束，则返回子进程PID。后者为-1

用来检查子进程返回状态的宏

WIFEXITED这个宏用来指出子进程是否为正常退出的,如果是,它会返回一个非零值.

WEXITSTATUS当WIFEXITED返回非零值时,我们可以用这个宏来提取子进程的返回值

wait函数使用

```c
#include<sys/types.h>
#include<sys/uio.h>
#include<string.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdio.h>
#include<stdlib.h>

intmain() {
    pid_t cid;
    cid = fork();
    if (cid < 0) {
        perror("子进程创建失败！");
    } else {
        if (cid == 0) {
            printf("子进程工作\n");
            printf("子进程PID=%d,PPID=%d\n", getpid(),getppid());
            //sleep(20); //1

        } else {
            //wait(NULL);//2
            //sleep(20);//3
            printf("父进程工作\n");
            printf("父进程PID=%d,PPID=%d\n", getpid(),getppid());
        }
    }
    return 0;
}
```

针对上述代码作以下分析：

1.当子进程退出时，如果父进程没有wait进行回收资源，子进程就会一直变为僵尸进程（Z）直到父进程退出

作法：

打开3处注释后执行程序，查看进程状态，如下

[root@localhostDebug]# ps -C Process -o pid,ppid,stat,cmd

PID PPID STAT CMD

12233 11563S /root/workspace/Process/Debug/Process

12238 12233Z [Process] <defunct>

=>可以看到子进程此时的状态时Z（僵尸进程）

控制台输出如下

子进程工作

子进程PID=12238,PPID=12233

（20S后…..）

父进程工作

父进程PID=12233,PPID=11563

2.使用wait进行进程同步，父进程直到子进程退出，wait才会结束等待

作法：

打开1，2处注释后执行程序，查看进程状态，如下

[root@ Debug8$] ps -C Process -o pid,ppid,stat,cmd

PID PPID STAT CMD

3425 2432 S /root/workspace/Process/Debug/Process

3430 3425 S /root/workspace/Process/Debug/Process

=>父进程与子进程都处于sleep状态

控制台输出如下

子进程工作

子进程PID=3430,PPID=3425

（20S后…..）

父进程工作

父进程PID=3425,PPID=2432

3. 使用wait进行进程同步，子进程退出后，父进程结束wait等待，同时清空子进程信息，此时子进程不再是僵尸进程

作法：

打开2，3处注释后执行程序，查看进程状态，如下

[root@localhostDebug]# ps -C Process -o pid,ppid,stat,cmd

  PID PPID STAT CMD

1250611563 S /root/workspace/Process/Debug/Process

=>可以看到此时只有父进程信息

控制台输出如下

子进程工作

子进程PID=12511,PPID=12506

（20S后…..）

父进程工作

父进程PID=12506,PPID=11563

WEXITSTATUS与WIFEXITED宏的使用

```c
#include<sys/types.h>
#include<sys/uio.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdio.h>
#include<stdlib.h>
intmain() {
    pid_t cid;
    int pr, status;
    cid = fork();
    if (cid < 0) {
        perror("子进程创建失败！");
    } else {
        if (cid == 0) {
            printf("子进程工作PID=%d,父进程PID=%d\n", getpid(),getppid());
            sleep(20);
            exit(3);
        } else {
            pr = wait(&status);
            if (WIFEXITED(status)) {
                printf("父进程工作PID=%d\n", getpid());
                printf("WAIT返回值=%d\n", pr);
                printf("子进程正常退出PID=%d\n", getpid());
                printf("WIFEXITED(status)=%d\n", WIFEXITED(status));
                printf("WEXITSTATUS(status)=%d\n", WEXITSTATUS(status));
            } else {
                printf("子进程异常退出PID=%d,信号=%d\n", getpid(), status);
                printf("WAIT返回值=%d\n", pr);
            }
        }
    }
    return 0;
}
```

基于上面代码做出分析：

1.子进程正常退出

控制台输出信息如下：

子进程工作PID=12070,父进程PID=12069

（20S后…..）

父进程工作PID=12069

WAIT返回值=12070

子进程正常退出PID=12069

WIFEXITED(status)=1

WEXITSTATUS(status)=3

2.子进程异常退出

作法：

运行程序，在子进程SLEEP期间，杀死子进程

[root@localhost Debug]# kill -9 11990

控制台台输出如下

子进程工作PID=11990,父进程PID=11985

（kill -9 PID 杀死子进程）

子进程异常退出PID=11985,信号=9

可以看出子进程正常退出时，status返回值是exit的退出值，子进程异常退出时status返回值信号值

waitpid函数使用

waitpid的参数说明

参数pid的值有以下几种类型:

pid>0时,只等待进程ID等于pid的子进程,不管其它已经有多少子进程运行结束退出了,只要指定的子进程还没有结束,waitpid就会一直等下去.

pid=-1时,等待任何一个子进程退出,没有任何限制,此时waitpid和wait的作用一模一样.

pid=0时,等待同一个进程组中的任何子进程,如果子进程已经加入了别的进程组，waitpid不会对它做任何理睬.

pid<-1时,等待一个指定进程组中的任何子进程,这个进程组的ID等于pid的绝对值.

参数options的值有以下几种类型:

如果使用了WNOHANG参数,即使没有子进程退出,它也会立即返回,不会像wait那样永远等下去.

如果使用了WUNTRACED参数,则子进程进入暂停则马上返回,但结束状态不予以理会.

如果我们不想使用它们,也可以把options设为0,如:ret=waitpid(-1,NULL,0);

WNOHANG使用

```c
#include<sys/types.h>
#include<sys/uio.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdio.h>
#include<stdlib.h>

int main() {
    pid_t cid;
    int pr, status;
    cid = fork();
    if (cid < 0) {
        perror("子进程创建失败！");
    } else {
        if (cid == 0) {
            printf("子进程工作PID=%d\n", getpid());
            sleep(5);
            exit(3);
        } else {

            do{
                pr = waitpid(0,&status,WNOHANG);
                if(pr==0)
                {
                printf("没有子进程退出,继续执行..\n");
                sleep(1);
                }

            }while(pr==0);
            printf("子进程正常退出PID=%d\n", pr);

        }
    }
    return 0;
}
```

控制台输出

没有子进程退出,继续执行..

子进程工作PID=3632

没有子进程退出,继续执行..

没有子进程退出,继续执行..

没有子进程退出,继续执行..

没有子进程退出,继续执行..

子进程正常退出PID=3632

针对某一进程进行等待

```c
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdio.h>
#include<stdlib.h>

intmain() {
    pid_t cid;
    int pr, status;
    cid = fork();
    if (cid < 0) {
        perror("子进程创建失败！");
    } else {
        if (cid == 0) {
            printf("子进程工作PID=%d,PPID=%d\n", getpid(), getppid());
            sleep(20);
            exit(3);
        } else {
            pr = waitpid(cid, &status, 0);
            printf("父进程正常退出PID=%d\n", pr);
        }
    }
    return 0;
}
```

控制台输出

子进程工作PID=4257,PPID=4252

父进程正常退出PID=4257

WUNTRACED 使用

```c
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<stdio.h>
#include<stdlib.h>

intmain() {
    pid_t cid;
    int pr, status;
    cid = fork();
    if (cid < 0) {
        perror("子进程创建失败！");
    } else {
        if (cid == 0) {
            printf("子进程工作PID=%d,PPID=%d\n", getpid(), getppid());
            sleep(30);
            exit(3);
        } else {
            pr = waitpid(cid, &status, WUNTRACED);
            printf("父进程正常退出PID=%d,status=%d\n", pr,status);
        }
    }
    return 0;
}
```

作法：在子进程SLEEP时，通过SHELL命令停止子进程

[root@ ~ 6$] kill -STOP PID

控制台输出

子进程工作PID=4110,PPID=4108

(SLEEP期间，停止子进程)

父进程正常退出PID=4110,status=4991

在查看进程状态，发现此时父进程子进程都已经退出

[root@ Debug 13$] ps -C Process -opid,ppid,stat,cmd

PID PPID STAT CMD

exec系统调用

函数作用：以新进程代替原有进程，但PID保持不变

形式:

```
int execl(const char *path, const char*arg, ...);

int execlp(const char *file, const char*arg, ...);

int execle(const char *path, const char*arg, ..., char * const envp[]);

int execv(const char *path, char *constargv[]);
```

int execvp(const char *file, char *constargv[]);

int execve(const char *path, char *constargv[], char *const envp[]);

举例：

```
exec1.c
#include<stdio.h>
#include<unistd.h>

intmain()
{
printf("这是第一个进程PID=%d\n",getpid());
execv("e2",NULL);
printf("asa");
return 0;
}

exec2.c
#include<stdio.h>
#include<unistd.h>

intmain()
{
printf("这是第二个进程PID=%d\n",getpid());
}
```

运行结果：

[root@ Process 9$] gcc -o e1 exec1.c

[root@ Process 10$] gcc -o e2 exec2.c

[root@ Process 11$] ./e1

这是第一个进程PID=3051

这是第二个进程PID=3051

2、Prctl 、ptrace 的功能和用法。能否在此基础上扩展进行进程状态查看和检查。

：

可查看进行状态并扩张检查

(1) Ptrace:

```
#include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

通过ptrace()这个系统调用,可以让一个进程去观察并且改变另外一个进程的行为,同时监测内存和寄存器.主要被用于断点调试以及系统调用的trace.

黑客也常用用它来做一些其他的工作,如hook

首先,tracee需要被tracer attach.附加和命令序列是针对每一个线程来说的,在一个多线程的进程中,每个线程都可以被独立的附加上一个不同的tracer,或者不被附加.这样的情况下,tracee意味着一个线程,不是一个多线程的进程.Ptrace命令是通过下面个调用发送给tracee

ptrace(PTRACE_foo,pid,...)

这里的pid是相关linux线程的ID.也就是在/proc/%d/task/ ,这个文件下的id号

当tracee被跟踪时,每发送一个信号,tracee都会暂停,即使这个信号会被忽略(除非是SIGKILL,这个会是tracee停止).tracer将会在下次waitpid(2)系统调用的时候被通知.该waitpid调用会返回一个status值,该值中包含让tracee停止的信息.当tracee停止的时候,tracer可以通过多种ptrace调用来修改tracee,然后tracer让tracee继续执行,可以有选择的忽略信号,甚至传递一个不同的信号.

后面看过linux内核才知道，对于ｌｉｎｕｘ来说，没有什么进程之说(不准确)。在linux内核中，只存在一种最小的执行单位，暂且叫做线程吧，如果多个线程有一个共同的进程组，那么他们就组成了一个进程。

在其他系统中,如windows，线程就是跟进程的实线机制不一样，是一种比进程轻量的执行单位。

(2) Prctl:

```
int prctl ( int option,unsigned long arg2,unsigned long arg3,unsigned long arg4,unsigned long arg5 )
```

这个系统调用指令是为进程制定而设计的，明确的选择取决于 option:

PR_GET_PDEATHSIG :返回处理器信号;

PR_SET_PDEATHSIG :arg2作为处理器信号pdeath被输入，正如其名，如果父进程不能再用，进程接受这个信号。

PR_GET_DUMPABLE :返回处理器标志dumpable;

PR_SET_DUMPABLE :arg2作为处理器标志dumpable被输入。

PR_GET_NAME :返回调用进程的进程名字给参数arg2; （Since Linux2.6.9）

PR_SET_NAME :把参数arg2作为调用进程的经常名字。（SinceLinux 2.6.11）

PR_GET_TIMING :

PR_SET_TIMING :判定和修改进程计时模式,用于启用传统进程计时模式的

PR_TIMING_STATISTICAL，或用于启用基于时间戳的进程计时模式的

PR_TIMING_TIMESTAMP。

CAP_CHOWN功能：

在一个_POSIX_CHOWN_RESTRICTED功能定义的系统。这会越过改变系统文件所有者和组所有的权限

CAP_DAC_OVERRIED功能：

如果_POSIX_ACL定义，就会越过所有的DAC访问，包括ACL执行访问，用CAP_LINUX_IMMUTABLE功能来排除

DAC的访问

CAP_DAC_READ_SEARCH功能：

如果_POSIX_ACL定义，就会越过所有的DAC的读限制，

并在所有的文件和目录里搜索，包括ACL限制。用CAP_LINUX_IMMUTABLE来限制DAC访问

CAP_FOWNER功能：

越过文件说有的允许限制，如文件的所有者ID必须和用户ID一样，除了CAP_FSETID可用。它不会越过MAC和DAC限制

CAP_FSETID功能：

越过当设置文件的S_ISUID和S_ISGID位的时候，用户的ID必须和所有者ID匹配的限制，设置S-ISGID位的时候，组ID

必须和所有者ID匹配的限制，用chown来设置S_ISUID和S_ISGID为的功能限制

CAP_FS_MASK功能：

用来回应suser（）或是fsuser（）。

CAP_KILL功能：

一个有有效用户ID的进程发送信号时必须匹配有效用户ID的功能会越过

CAP_SETGID功能：

允许setgid（） 功能， 允许setgroups（）

允许在socket里伪造gid

CAP_SETUID功能：

允许set*uid（）功能 允许伪造pid在socket

CAP_SETPCAP 功能：

把所有的许可给所有的pid。或是把所有的许可删除

CAP_LINUX_IMMUTABLE功能：

允许更改S_IMMUTABLE和S_APPEND文件属性

CAP_NET_BIND_SERVICE功能：

允许绑定1024下的TCP/UDP套接字

CAP_NET_BROADCAST功能：

允许广播，监听多点传送

CAP_NET_ADMIN功能：

允许配置接口

允许管理IP防火墙IP伪装和帐户

允许配置socket调试选项

允许修改路由表

允许配置socket上的进程的组属性

允许绑定所有地址的透明代理

允许配置TOS（服务类型）

允许配置混杂模式

允许清除驱动状态

允许多点传送

允许读或写系统记录

CAP_NET_RAW功能：

允许用RAW套接字

允许用PACKET套接字

CAP_IPC_LOCK功能：

允许琐定共享内存段

允许mlock和mlockall

CAP_IPC_OWNER功能：

越过IPC所有权检查

CAP_SYS_MODULE功能：

插入或删除内核模块

CAP_SYS_RAWIO功能：

允许ioperm/iopl和/dev/prot的访问

允许/dev/mem和/dev/kmem访问

允许块设备访问（/dev/[sh]d? ? ）

CAP_SYS_CHROOT功能：

允许chroot（）

CAP_SYS_PTRACE功能：

允许ptrace（）任何进程

CAP_SYS_PACCT功能：

允许配置进程帐号

CAP_SYS_ADMIN功能：

允许配置安全钥匙

允许管理随机设备

允许设备管理

允许检查和配置磁盘限额

允许配置内核日志

允许配置域名

允许配置主机名

允许调用bdflush（）命令

允许mount（）和umount（）命令

允许配置smb连接

允许root的ioctls

允许nfsservctl

允许VM86_REQUEST_IRQ

允许在alpha上读写pci配置

允许在mips上的irix_prctl

允许刷新所有的m68k缓存

允许删除semaphores

用CAP_CHOWN去代替"chown"IPC消息队列，标志和共享内存

允许锁定或是解锁共享内存段

允许开关swap

允许在socket伪装pids

允许设置块设备的缓存刷新

允许设置软盘驱动器

允许开关DMA开关

允许管理md设备

允许管理ide驱动

允许访问nvram设备

允许管理apm_bios，串口或是bttv电视设备

允许在isdn CAPI的驱动下生成命令

允许读取pci的非标准配置

允许DDI调试ioctl

允许发送qic-117命令

允许启动或禁止SCSI的控制和发送SCSI命令 允许配置加密口令在回路文件系统上

CAP_SYS_BOOT功能：

允许用reboot（） 命令

CAP_SYS_NICE功能：

允许提高或设置其他进程的优先权

允许在自己的进程用FISO和实时的安排和配置

CAP_SYS_RESOURCE功能：

越过资源限制，设置资源限制

越过配额限制

越过保留的ext2文件系统

允许大于64hz的实时时钟中断

越过最大数目的控制终端

越过最大数目的键

CAP_SYS_TIME功能：

允许处理系统时钟

允许_stime

允许设置实时时钟

CAP_SYS_TTY_CONFIG功能：

允许配置终端设备

允许vhangup（）终端

返回值

PR_GET_DUMPABLE 和 PR_GET_KEEPCAPS 成功时返回0或者1。其他的option值都是成功时返回0。

错误时返回 －1，并设置相应的错误号。

EINVAL————option的值不正确，或者当它是PR_SET_PDEATHSIG时，参数arg2的值不是0或者信号数字。

EBADF————无效的描述符

实例：于多线程应用程序，如果能够给每个线程命名，那么调试起来的便利是不言而喻的。

```c
#include<stdio.h>
#include<pthread.h>
#include<sys/prctl.h>
void* tmain(void*arg)
{
    char name[32];
    prctl(PR_SET_NAME,(unsignedlong)"xx");
    prctl(PR_GET_NAME,(unsignedlong)name);
    printf("%s/n", name);
    while(1){
        sleep(1);
    }
}
int main(void)
{
    pthread_t tid;
    pthread_create(&tid,NULL, tmain,NULL);
    pthread_join(tid,NULL);
    return 0;
}

```

编 译并运行：

```
xiaosuo@gentux test $ gcc t_threadname.c -lpthread

xiaosuo@gentux test $ ./a.out

 xx
```

在 另一个终端，通过ps找到a.out的pid：

```
xiaosuo@gentux test $ ps aux | grep a.out

 xiaosuo 29882  0.0  0.0 14144   544 pts/6    Sl+ 16:23   0:00 ./a.out
```

看命名是否奏效：

```
xiaosuo@gentux test $ cd /proc/29882/task/

xiaosuo@gentux task $ ls

29882 29883

xiaosuo@gentux task $ cd 29883/

xiaosuo@gentux 29883 $ cat cmdline

./a.outxiaosuo@gentux 29883 $
```

有点儿郁闷，cmdline显示的竟然还是./a.out。通过 运行时打印的xx和strace检查prctl的返回值确认prctl确实成功运行。怀疑这个名字只能通过prctl获得，有点儿失落，可心仍不甘。查看 ps的man，并实验，终于找到了"xx":

```
xiaosuo@gentux 29883 $ ps -L -p 29882
  PID   LWP TTY          TIME CMD
29882 29882 pts/6    00:00:00 a.out
29882 29883 pts/6    00:00:00 xx
```

Linux下进程重命名的方法：

使用系统函数prctl()，声明如下：

```
#include <sys/prctl.h>
int prctl(int option, unsigned long arg2, unsigned long arg3, unsigned long arg4, unsigned long arg5);
```

具体用法请参考http://www.kernel.org/doc/man-pages/online/pages/man2/prctl.2.html

进程重命名代码：

prctl(PR_SET_NAME, “process_name”, NULL, NULL, NULL);

第一个参数是操作类型，指定PR_SET_NAME，即设置进程名

第二个参数是进程名字符串，长度至多16字节

3、鉴别用户，采用了人脸识别、和手机号关联等新手段。

4、请查阅资料看看还有什么新技术、新方法鉴别用户。鉴别中最常见的问题是什么，是否还有未解决的或需要引入新思路来解决的。

还可以采用红外线传感识别，指纹识别等方式进行用户鉴别。鉴别中最常见的是隐私泄露的问题，可以考虑增设密码安防或者设置独立用户数据库的方式尝试进行解决。