## 实验四：进程管理（二）

### 实验目的：

1. 学习进程管理的系统调用
2. 掌握使用系统调用获取进程的属性、创建进程、实现进程控制等

### 预备知识：

1. 进程属性

1.1 getpid（取得进程ID）

表头文件 #include<unistd.h>

定义函数 pid\_t getpid(void);

函数说明 getpid（）用来取得目前进程的进程ID，许多程序利用取到的此值来建立临时文件，以避免临时文件相同带来的问题。

返回值 目前进程的进程ID

1.2 getppid（取得父进程的进程ID）

表头文件 #include<unistd.h>

定义函数 pid\_t getppid(void);

函数说明 getppid()用来取得目前进程的父进程ID。

返回值 目前进程的父进程ID。

1.3 getegid（取得有效的组ID）

表头文件 #include<unistd.h>

#include<sys/types.h>

定义函数 gid\_t getegid(void);

函数说明 getegid()用来取得执行目前进程有效组ID。有效的组ID用来决定进程执行时组的权限。返回值返回有效的组ID。

1.4 geteuid（取得有效的用户ID）

表头文件 #include<unistd.h>

#include<sys/types.h>

定义函数 uid\_t geteuid(void)

函数说明 geteuid()用来取得执行目前进程有效的用户ID。有效的用户ID用来决定进程执行的权限，借由此改变此值，进程可以获得额外的权限。倘若执行文件的setID位已被设置，该文件执行时，其进程的euid值便会设成该文件所有者的uid。

返回值 返回有效的用户ID。

1.5 getgid（取得真实的组ID）

表头文件 #include<unistd.h>

#include<sys/types.h>

定义函数 gid\_t getgid(void);

函数说明 getgid()用来取得执行目前进程的组ID。

返回值 返回组ID

1.6 getuid（取得真实的用户ID）

表头文件 #include<unistd.h>

#include<sys/types.h>

定义函数 uid\_t getuid(void);

函数说明 getuid()用来取得执行目前进程的用户ID。

返回值 用户ID

1.7 times （取得进程相关的时间）

表头文件 #include<sys/times.h>

定义函数 clock\_t times(struct tms \*buf);

函数说明 取得进程运行相关的时间。

参数说明

/\*sys/times.h\*/

struct tms{

clock\_t tms\_utime; /\*进程花在执行用户模式代码上的时间\*/

clock\_t tms\_stime; /\*进程花在执行内核代码上的时间\*/

clock\_t tms\_cutime; /\*子进程花在执行用户模式代码上的时间\*/

clock\_t tms\_cstime; /\*子进程花在执行内核代码上的时间\*/

}

返回值 自系统自举后经过的时钟嘀嗒数。

注意 时钟嘀嗒数time转换为用户可读的方式，即多少秒，需通过如下方式：

(float)time/sysconf(\_SC\_CLK\_TCK);

1. 进程创建

2.1 system（执行shell 命令）

表头文件 #include<stdlib.h>

定义函数 int system(const char \* string);

函数说明 system()会调用fork()产生子进程，由子进程来调用/bin/sh -c string来执行参数string字符串所代表的命令，此命令执行完后随即返回原调用的进程。

返回值 如果system()在调用/bin/sh时失败则返回127，其他失败原因返回-1。若参数string为空指针(NULL)，则返回非零值。

2.2 fork（建立一个新的进程）

表头文件 #include<unistd.h>

定义函数 pid\_t fork(void);

函数说明 fork()会产生一个新的子进程，其子进程会复制父进程的数据与堆栈空间，并继承父进程的用户代码，组代码，环境变量、已打开的文件代码、工作目录和资源限制等。Linux 使用copy-on-write(COW)技术，只有当其中一进程试图修改欲复制的空间时才会做真正的复制动作，由于这些继承的信息是复制而来，并非指相同的内存空间，因此子进程对这些变量的修改和父进程并不会同步。此外，子进程不会继承父进程的文件锁定和未处理的信号。

注意 Linux不保证子进程会比父进程先执行或晚执行，因此编写程序时要留意死锁或竞争条件的发生。

返回值 如果fork()成功则在父进程会返回新建立的子进程代码(PID)，而在新建立的子进程中则返回0。如果fork 失败则直接返回-1，失败原因存于errno中。

错误代码 EAGAIN 内存不足。ENOMEM 内存不足，无法配置核心所需的数据结构空间。

* 1. exec函数簇

表头文件 #include<unistd.h>

2.3.1 execl（执行文件）

定义函数 int execl(const char \* path,const char \* arg,....);

函数说明 execl()用来执行参数path字符串所代表的文件路径，接下来的参数代表执行该文件时传递过去的argv(0)、argv[1]……，最后一个参数必须用空指针(NULL)作结束。

返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno中。

范例 #include<unistd.h>

main()

{

execl(“/bin/ls”,”ls”,”-al”,”/etc/passwd”,(char \* )0);

}

2.3.2 execlp（从PATH 环境变量中查找文件并执行）

定义函数 int execlp(const char \* file,const char \* arg,……)；

函数说明 execlp()会从PATH 环境变量所指的目录中查找符合参数file的文件名，找到后便执行该文件，然后将第二个以后的参数当做该文件的argv[0]、argv[1]……，最后一个参数必须用空指针(NULL)作结束。

返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno 中。

错误代码 参考execve()。

范例 /\* 执行ls -al /etc/passwd execlp()会依PATH 变量中的/bin找到/bin/ls \*/

#include<unistd.h>

main()

{

execlp(“ls”,”ls”,”-al”,”/etc/passwd”,(char \*)0);

}

2.3.3 execv（执行文件）

定义函数 int execv (const char \* path, char \* const argv[ ]);

函数说明 execv()用来执行参数path字符串所代表的文件路径，与execl()不同的地方在于execve()只需两个参数，第二个参数利用数组指针来传递给执行文件。

返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno 中。

范例 /\* 执行/bin/ls -al /etc/passwd \*/

#include<unistd.h>

main()

{

char \* argv[ ]={“ls”,”-al”,”/etc/passwd”,(char\*) 0}};

execv(“/bin/ls”,argv);

}

2.3.4 execve（执行文件）

定义函数 int execve(const char \* filename,char \* const argv[ ],char \* const envp[ ]);

函数说明 execve()用来执行参数filename字符串所代表的文件路径，第二个参数系利用数组指针来传递给执行文件，最后一个参数则为传递给执行文件的新环境变量数组。

返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno 中。

错误代码 EACCES

1. 欲执行的文件不具有用户可执行的权限。

2. 欲执行的文件所属的文件系统是以noexec 方式挂上。

3.欲执行的文件或script翻译器非一般文件。

EPERM

1.进程处于被追踪模式，执行者并不具有root权限，欲执行的文件具有SUID 或SGID 位。

2.欲执行的文件所属的文件系统是以nosuid方式挂上，欲执行的文件具有SUID 或SGID 位元，但执行者并不具有root权限。

E2BIG 参数数组过大

ENOEXEC 无法判断欲执行文件的执行文件格式，有可能是格式错误或无法在此平台执行。

EFAULT 参数filename所指的字符串地址超出可存取空间范围。

ENAMETOOLONG 参数filename所指的字符串太长。

ENOENT 参数filename字符串所指定的文件不存在。

ENOMEM 核心内存不足

ENOTDIR 参数filename字符串所包含的目录路径并非有效目录

EACCES 参数filename字符串所包含的目录路径无法存取，权限不足

ELOOP 过多的符号连接

ETXTBUSY 欲执行的文件已被其他进程打开而且正把数据写入该文件中

EIO I/O 存取错误

ENFILE 已达到系统所允许的打开文件总数。

EMFILE 已达到系统所允许单一进程所能打开的文件总数。

EINVAL 欲执行文件的ELF执行格式不只一个PT\_INTERP节区

EISDIR ELF翻译器为一目录

ELIBBAD ELF翻译器有问题。

范例 #include<unistd.h>

main()

{

char \* argv[ ]={“ls”,”-al”,”/etc/passwd”,(char \*)0};

char \* envp[ ]={“PATH=/bin”,0}

execve(“/bin/ls”,argv,envp);

}

2.3.5 execvp（执行文件）

定义函数 int execvp(const char \*file ,char \* const argv []);

函数说明 execvp()会从PATH 环境变量所指的目录中查找符合参数file 的文件名，找到后便执行该文件，然后将第二个参数argv传给该欲执行的文件。

返回值 如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno中。

范例 /\*请与execlp（）范例对照\*/

#include<unistd.h>

main()

{

char \* argv[ ] ={ “ls”,”-al”,”/etc/passwd”,0};

execvp(“ls”,argv);

}

1. 进程等待

3.1 wait（等待子进程中断或结束）

表头文件 #include<sys/types.h>

#include<sys/wait.h>

定义函数 pid\_t wait (int \* status);

函数说明 wait()会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用wait()时子进程已经结束，则wait()会立即返回子进程结束状态值。子进程的结束状态值会由参数status 返回，而子进程的进程ID也会一快返回。如果不在意结束状态值，则参数 status可以设成NULL。子进程的结束状态值请参考waitpid()。

返回值 如果执行成功则返回子进程ID(PID)，如果有错误发生则返回-1。失败原因存于errno中。

附加说明

范例 #include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/wait.h>

main()

{

pid\_t pid;

int status,i;

if(fork()= =0){

printf(“This is the child process .pid =%d\n”,getpid());

exit(5);

}else{

sleep(1);

printf(“This is the parent process ,wait for child...\n”;

pid=wait(&status);

i=WEXITSTATUS(status);

printf(“child’s pid =%d .exit status=%d\n”,pid,i);

}

}

3.2 waitpid（等待子进程中断或结束）

表头文件 #include<sys/types.h>

#include<sys/wait.h>

定义函数 pid\_t waitpid(pid\_t pid,int \* status,int options);

函数说明 waitpid()会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用wait()时子进程已经结束，则wait()会立即返回子进程结束状态值。子进程的结束状态值会由参数status返回，而子进程的进程ID也会一快返回。如果不在意结束状态值，则参数status可以设成NULL。参数pid为欲等待的子进程ID，其他数值意义如下:

pid<-1 等待进程组ID为pid绝对值的任何子进程。

pid=-1 等待任何子进程，相当于wait()。

pid=0 等待进程组ID与目前进程相同的任何子进程。

pid>0 等待任何子进程ID为pid的子进程。

参数option可以为0 或下面的OR 组合：

WNOHANG 如果没有任何已经结束的子进程则马上返回，不予以等待。

WUNTRACED 如果子进程进入暂停执行情况则马上返回，但结束状态不予以理会。

子进程的结束状态返回后存于status，底下有几个宏可判别结束情况

WIFEXITED(status)如果子进程正常结束则为非0值。

WEXITSTATUS(status)取得子进程exit()返回的结束代码，一般会先用WIFEXITED 来判断是否正常结束才能使用此宏。

WIFSIGNALED(status)如果子进程是因为信号而结束则此宏值为真

WTERMSIG(status)取得子进程因信号而中止的信号代码，一般会先用WIFSIGNALED 来判断后才使用此宏。

WIFSTOPPED(status)如果子进程处于暂停执行情况则此宏值为真。一般只有使用WUNTRACED 时才会有此情况。

WSTOPSIG(status)取得引发子进程暂停的信号代码，一般会先用WIFSTOPPED 来判断后才使用此宏。

返回值 如果执行成功则返回子进程ID(PID)，如果有错误发生则返回-1。失败原因存于errno中。

1. 信号

4.1 kill（传送信号给指定的进程）

表头文件 #include<sys/types.h>

#include<signal.h>

定义函数 int kill(pid\_t pid,int sig);

函数说明 kill()可以用来送参数sig指定的信号给参数pid指定的进程。参数pid有几种情况:

pid>0 将信号传给进程ID为pid 的进程。

pid=0 将信号传给和目前进程相同进程组的所有进程

pid=-1 将信号广播传送给系统内所有的进程

pid<0 将信号传给进程组ID为pid绝对值的所有进程

返回值 执行成功则返回0，如果有错误则返回-1。

错误代码 EINVAL 参数sig 不合法

ESRCH 参数pid 所指定的进程或进程组不存在

EPERM 权限不够无法传送信号给指定进程

范例 #include<unistd.h>

#include<signal.h>

#include<sys/types.h>

#include<sys/wait.h>

main()

{

pid\_t pid;

int status;

if(!(pid= fork())){

printf(“Hi I am child process!\n”);

sleep(10);

return;

}

else{

printf(“send signal to child process (%d) \n”,pid);

sleep(1);

kill(pid ,SIGABRT);

wait(&status);

if(WIFSIGNALED(status))

printf(“chile process receive signal %d\n”,WTERMSIG(status));

}

}

4.2 signal（设置信号处理方式）

表头文件 #include<signal.h>

定义函数 void (\*signal(int signum,void(\* handler)(int)));

函数说明 signal()会依参数signum 指定的信号编号来设置该信号的处理函数。当指定的信号到达时就会跳转到参数handler指定的函数执行。如果参数handler不是函数指针，则必须是下列两个常数之一:

SIG\_IGN 忽略参数signum指定的信号。

SIG\_DFL 将参数signum 指定的信号重设为核心预设的信号处理方式。

返回值 返回先前的信号处理函数指针，如果有错误则返回SIG\_ERR(-1)。

附加说明 在信号发生跳转到自定的handler处理函数执行后，系统会自动将此处理函数换回原来系统预设的处理方式，如果要改变此操作请改用sigaction()。

1. 进程调度

5.1 getpriority（取得程序进程执行优先权）

表头文件 #include<sys/time.h>

#include<sys/resource.h>

定义函数 int getpriority(int which,int who);

函数说明 getpriority()可用来取得进程、进程组和用户的进程执行优先权。

参数 which有三种数值，参数who 则依which值有不同定义

which who 代表的意义

PRIO\_PROCESS who 为进程ID

PRIO\_PGRP who 为进程的组ID

PRIO\_USER who 为用户ID

此函数返回的数值介于-20 至20之间，代表进程执行优先权，数值越低代表有较高的优先次序，执行会较频繁。

返回值 返回进程执行优先权，如有错误发生返回值则为-1 且错误原因存于errno。

附加说明 由于返回值有可能是-1，因此要同时检查errno是否存有错误原因。最好在调用次函数前先清除errno变量。

错误代码 ESRCH 参数which或who 可能有错，而找不到符合的进程。EINVAL 参数which 值错误。

5.2 setpriority（设置程序进程执行优先权）

表头文件 #include<sys/time.h>

#include<sys/resource.h>

定义函数 int setpriority(int which,int who, int prio);

函数说明 setpriority()可用来设置进程、进程组和用户的进程执行优先权。参数which有三种数值，参数who 则依which值有不同定义

参数prio介于-20 至20 之间。代表进程执行优先权，数值越低代表有较高的优先次序，执行会较频繁。此优先权默认是0，而只有超级用户（root）允许降低此值。

返回值 执行成功则返回0，如果有错误发生返回值则为-1，错误原因存于errno。

ESRCH 参数which或who 可能有错，而找不到符合的进程

EINVAL 参数which值错误。

EPERM 权限不够，无法完成设置

EACCES 一般用户无法降低优先权

5.3 nice（改变进程优先顺序）

表头文件 #include<unistd.h>

定义函数 int nice(int inc);

函数说明 nice()用来改变进程的进程执行优先顺序。参数inc数值越大则优先顺序排在越后面，即表示进程执行会越慢。只有超级用户才能使用负的inc 值，代表优先顺序排在前面，进程执行会较快。

返回值 如果执行成功则返回0，否则返回-1，失败原因存于errno中。

错误代码 EPERM 一般用户企图转用负的参数inc值改变进程优先顺序。

1. 其他

6.1 atexit（设置程序正常结束前调用的函数）

表头文件 #include<stdlib.h>

定义函数 int atexit (void (\*function)(void));

函数说明 atexit()用来设置一个程序正常结束前调用的函数。当程序通过调用exit()或从main中返回时，参数function所指定的函数会先被调用，然后才真正由exit()结束程序。

返回值 如果执行成功则返回0，否则返回-1，失败原因存于errno中。

范例 #include<stdlib.h>

void my\_exit(void)

{

printf(“before exit () !\n”);

}

main()

{

atexit (my\_exit);

exit(0);

}

6.2 exit（正常结束进程）

表头文件 #include<stdlib.h>

定义函数 void exit(int status);

函数说明 exit()用来正常终结目前进程的执行，并把参数status返回给父进程，而进程所有的缓冲区数据会自动写回并关闭未关闭的文件。

### 实验内容:

1. 编写一个程序，打印进程的如下信息：进程标识符，父进程标识符，真实用户ID，有效用户ID，真实用户组ID，有效用户组ID。并分析真实用户ID和有效用户ID的区别。
2. 阅读如下程序：

/\* process using time \*/

#include<stdio.h>

#include<stdlib.h>

#include<sys/times.h>

#include<time.h>

#include<unistd.h>

void time\_print(char \*,clock\_t);

int main(void)

{

clock\_t start,end;

struct tms t\_start,t\_end;

start = times(&t\_start);

system(“grep the /usr/doc/\*/\* > /dev/null 2> /dev/null”);

end=times(&t\_end);

time\_print(“elapsed”,end-start);

puts(“parent times”);

time\_print(“\tuser CPU”,t\_end.tms\_utime);

time\_print(“\tsys CPU”,t\_end.tms\_stime);

puts(“child times”);

time\_print(“\tuser CPU”,t\_end.tms\_cutime);

time\_print(“\tsys CPU”,t\_end.tms\_cstime);

exit(EXIT\_SUCCESS);

}

void time\_print(char \*str, clock\_t time)

{

long tps = sysconf(\_SC\_CLK\_TCK);

printf(“%s: %6.2f secs\n”,str,(float)time/tps);

}

编译并运行，分析进程执行过程的时间消耗（总共消耗的时间和CPU消耗的时间），并解释执行结果。再编写一个计算密集型的程序替代grep，比较两次时间的花销。注释程序主要语句。

1. 阅读下列程序：

/\* fork usage \*/

#include<unistd.h>

#include<stdio.h>

#include<stdlib.h>

int main(void)

{

pid\_t child;

if((child=fork())==-1{

perror(“fork”);

exit(EXIT\_FAILURE);

}else if(child==0){

puts(“in child”);

printf(“\tchild pid = %d\n”,getpid());

printf(“\tchild ppid = %d\n”,getppid());

exit(EXIT\_SUCCESS);

}else{

puts(“in parent”);

printf(“\tparent pid = %d\n”,getpid());

printf(“\tparent ppid = %d\n”,getppid());

}

exit(EXIT\_SUCCESS);

}

编译并多次运行，观察执行输出次序，说明次序相同（或不同）的原因；观察进程ID，分析进程ID的分配规律。总结fork()的使用方法。注释程序主要语句。

1. 阅读下列程序：

/\* usage of kill,signal,wait \*/

#include<unistd.h>

#include<stdio.h>

#include<sys/types.h>

#include<signal.h>

int flag;

void stop();

int main(void)

{

int pid1,pid2;

signal(3,stop);

while((pid1=fork()) ==-1);

if(pid1>0){

while((pid2=fork()) ==-1);

if(pid2>0){

flag=1;

sleep(5);

kill(pid1,16);

kill(pid2,17);

wait(0);

wait(0);

printf(“\n parent is killed\n”);

exit(EXIT\_SUCCESS);

}else{

flag=1;

signal(17,stop);

printf(“\n child2 is killed by parent\n”);

exit(EXIT\_SUCCESS);

}

}else{

flag=1;

signal(16,stop);

printf(“\n child1 is killed by parent\n”);

exit(EXIT\_SUCCESS);

}

}

void stop(){

flag = 0;

}

编译并运行，等待或者按^C，分别观察执行结果并分析，注释程序主要语句。

flag有什么作用？通过实验说明。

1. 编写程序，要求父进程创建一个子进程，使父进程和个子进程各自在屏幕上输出一些信息，但父进程的信息总在子进程的信息之后出现。
2. 编写程序，要求父进程创建一个子进程，子进程执行shell命令find / -name hda\* 的功能，子进程结束时由父进程打印子进程结束的信息。执行中父进程改变子进程的优先级。
3. 编写程序，要求父进程创建一个子进程，子进程对一个50\*50的字符数组赋值，由父进程改变子进程的优先级，观察不同优先级进程使用CPU的时间。
4. 查阅Linux系统中struct task\_struct 的定义，说明每项成员的作用。  
   注：search in /usr/src/linux-2.6/include/linux/sched.h

### 实验报告：

完成上述要求中的第2,4,5,6,8题并提交报告。