

## 实验四

### *Hollow Man*

#### 实验名称：

进程管理（二）

#### 实验目的：

1. 进一步学习进程的属性
2. 学习进程管理的系统调用
3. 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
4. 掌握进程管理的基本原理

#### 实验时间

6 学时

#### 实验要求：

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。

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

范例

##### 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） /\*转换到的新用户组的 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） /\*转变管理员身份之后用户的 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） /\*转变之前，原来的组 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） /\*转变管理员身份之前用户的 ID\*/

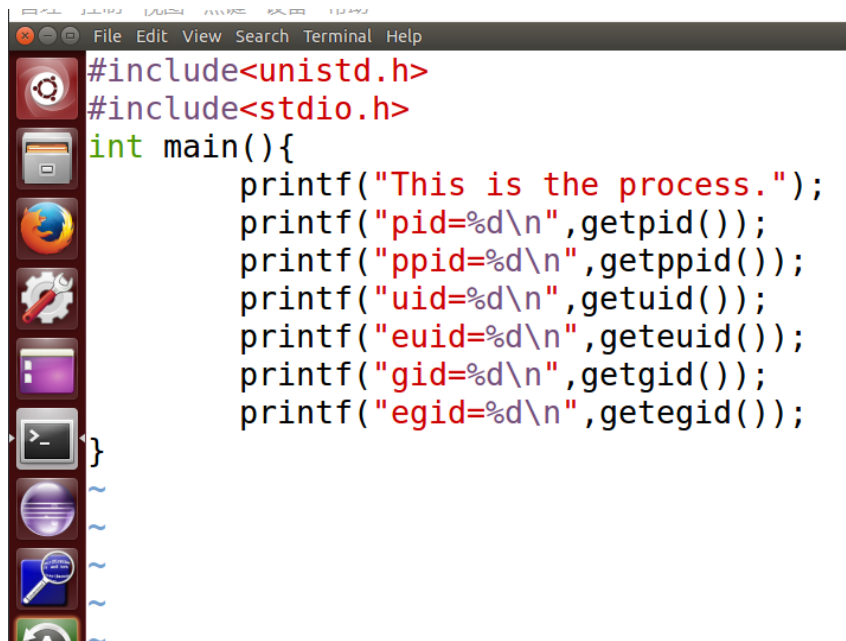
表头文件 `#include<unistd.h>`

`#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



```
File Edit View Search Terminal Help
#include<unistd.h>
#include<stdio.h>
int main(){
    printf("This is the process.");
    printf("pid=%d\n",getpid());
    printf("ppid=%d\n",getppid());
    printf("uid=%d\n",getuid());
    printf("euid=%d\n",geteuid());
    printf("gid=%d\n",getgid());
    printf("egid=%d\n",getegid());
}
```

```
moocos-> ./one
This is the process.pid=2665
ppid=2198
uid=1000
euid=1000
gid=1000
egid=1000
[~]
```

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");
/*找到了输出到 dev//null，寻找错误输出到 dev//null2*/

    end=times(&t_end);

    time_print("elapsed",end-start);

    puts("parent times");

    time_print("\tuser CPU",t_end.tms_utime); /*进程花在执行用户模式代码上的时间*/
    time_print("\tsys CPU",t_end.tms_stime); /*进程花在执行内核代码上的时间*/


    puts("child times");

    time_print("\tuser CPU",t_end.tms_cutime); /*子进程花在执行用户模式代码上的时间*/
    time_print("\tsys CPU",t_end.tms_cstime); /*子进程花在执行内核代码上的时间*/


    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); //计算秒数
}

```

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。

相关函数与变量：

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**。/\*从系统启动到执行 time 经过的滴答数\*/

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);`

程序截图即运行结果：

```

/*      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/nu
ll 2> /dev/null");
    /*找到了输出到dev//null, 寻找错误输出到dev//null2*/
    end=times(&t_end);

    time_print("elapsed",end-start);
    puts("parent times");
    time_print("tuser CPU",t_end.tms_ftime);
;
    time_print("\tsys CPU",t_end.tms_stime)
;

    puts("child times");
}

```

@

"os1.c" [dos] 35L, 848C

1,1

Top

```

    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);
}

```

35,1                  Bot

```

moocos-> ./os1
elapsed:  0.00 secs
parent times
tuser CPU:  0.00 secs
          sys CPU:  0.00 secs
child times
          user CPU:  0.00 secs
          sys CPU:  0.00 secs
[~/Downloads]

```

由于该程序的计算量很小，因此消耗的时间比较少，因不足 10ms 而直接显示为 0s。进程的执行时间等于用户 CPU 时间和系统 CPU 时间加从硬盘读取数据时间之和。接下来增加运算量，将 `grep` 改为计算密集型。

计算密集型截图及运行结果：

```

int main(void)
{
    clock_t start, end;
    struct tms t_start, t_end;
    start = times(&t_start);
    double i=0;
    int j=0,x=0,y=0,z=0;
    while(i<=100000&&j<=100000){
        i++;
        j++;
        z++;
        x=y/z;
        z++;
        x++;
        y++;
        int k;
        for(k=0;k<10000;k++){
            z+=k;
            x=y/z;
        }
    }
    /*找到了输出到dev//null, 寻找错误输出到dev//null2*/
    end=times(&t_end);

```

```

moocos-> ./os1
elapsed:   3.15 secs
parent times
tuser CPU:  2.93 secs
          sys CPU:  0.00 secs
child times
          user CPU:  0.00 secs
          sys CPU:  0.00 secs
[~/Downloads]

```

3. 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){//若为子进程
        puts("in child");
        printf("\tchild pid = %d\n",getpid());//打印 pid
        printf("\tchild ppid = %d\n",getppid());//打印 ppid
        exit(EXIT_SUCCESS);
    } else{//若为父进程
        puts("in parent");
        printf("\tparent pid = %d\n",getpid());//打印 pid
        printf("\tparent ppid = %d\n",getppid());//打印 ppid
    }
    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。

```
/* 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",get
pid());
        printf("\tchild ppid = %d\n",ge
tppid());
        exit(EXIT_SUCCESS);
    }else{
        puts("in parent");
        printf("\tparent pid = %d\n",ge
tpid());
        printf("\tparent ppid = %d\n",g
etppid());
        exit(EXIT_SUCCESS);
    }
}
~
```

1,1 All

Fork 用法:

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的子进程代码(PID), 而在新建的子进程中则返回 0。如果 fork 失败则直接返回-1, 失败原因存于 errno 中。

/\*返回值大小可用于判断检测当前进程运行在父进程还是子进程中\*/

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。

```
moocos-> ./os2
in parent
    parent pid = 3431
    parent ppid = 3354
in child
    child pid = 3432
    child ppid = 1198
[~/Downloads]
moocos-> ./os2
in parent
    parent pid = 3439
    parent ppid = 3354
in child
    child pid = 3440
    child ppid = 1198
[~/Downloads]
moocos-> ./os2
in parent
    parent pid = 3447
    parent ppid = 3354
in child
    child pid = 3448
    child ppid = 1198
[~/Downloads]
-----
```

```

moocos-> ./os2
in parent
    parent pid = 3462
    parent ppid = 3354
in child
    child pid = 3463
    child ppid = 1198
[~/Downloads]
moocos-> ./os2
in parent
    parent pid = 3470
    parent ppid = 3354
in child
    child pid = 3471
    child ppid = 1198
[~/Downloads]
moocos-> ./os2
in parent
    parent pid = 3478
    parent ppid = 3354
in child
    child pid = 3479
    child ppid = 1198
[~/Downloads]

```

创建进程 **id** 开始时一般随机分配，但若多次运行，或创建子进程时，会顺序分配内存。此外，当父进程结束时，子进程尚未结束，则子进程的父进程 **id** 变为 **1**，即 **init** **fork()** 的使用方法：

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

在父进程中用 **fork ()** 创建子进程，通过返回值 **if** 语句判断来进行父子进程代码执行。

注意到子进程的 **ppid** 并非父进程的 **pid**，而是 **1198**，这是由于在子进程结束前，父进程已经结束，子进程的 **ppid** 指向了统一收养孤儿进程的进程，他的 **pid** 号是 **1198**。