操作系统实验1 Readme

一、进程相关编程实验

0)基本程序

完成操作系统原理课程教材P103作业 3.7 的运行验证，多运行程序几次观察结果；

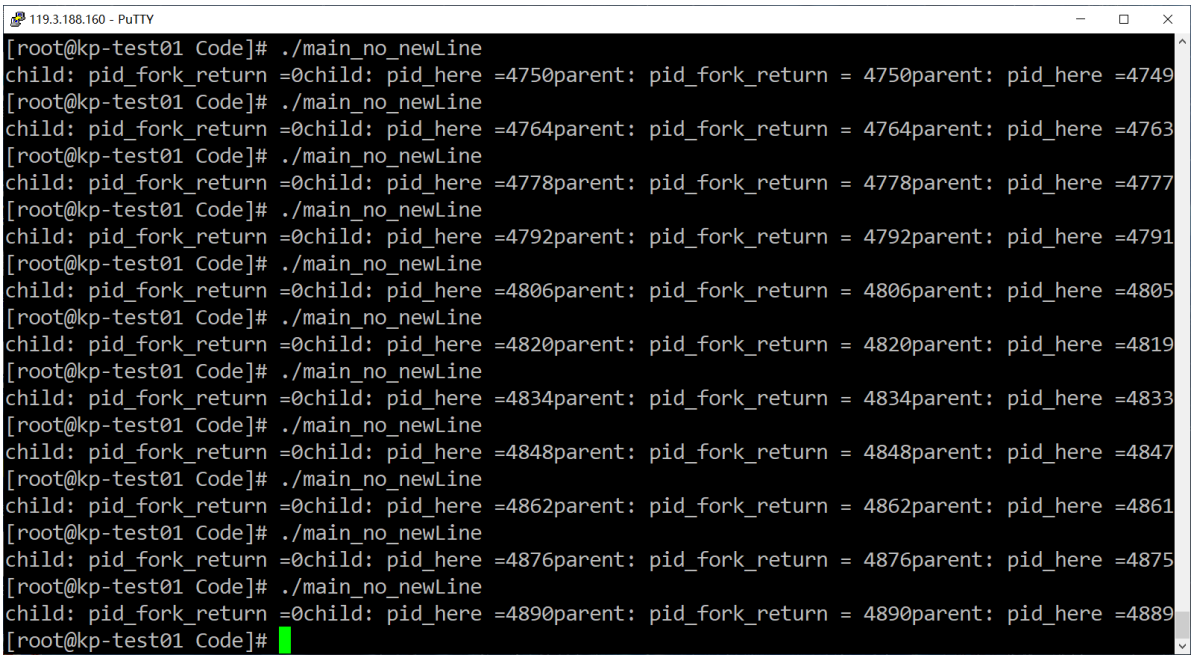
在Linux中，使用fork创建一个新进程，在父进程和子进程中分别执行，观察执行结果。

源程序如下：

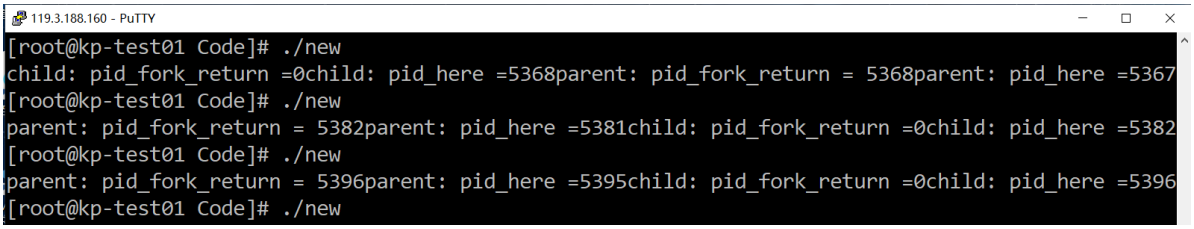
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    pid_t pid_fork_return, pid_here;
    pid_fork_return = 10;
    /* fork a child process */
    pid_fork_return = fork();

    if (pid_fork_return < 0) {
        /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }

    else if (pid_fork_return == 0) { /* child process */

        pid_here = getpid();

        printf("child: pid_fork_return =%d\n", pid_fork_return);   /* A */
        printf("child: pid_here =%d\n", pid_here); /* B */
    }

    else { /* parent process */
        pid_here = getpid();
        printf("parent: pid_fork_return = %d\n", pid_fork_return);  /* C */
        printf("parent: pid_here =%d\n", pid_here); /* D */
        int a =20;
        wait(NULL);
    }

    return 0;
}
```

创建新进程成功后，系统中出现两个基本完全相同的进程，这两个进程执行没有固定的先后顺序，哪个进程先执行要看系统的进程调度策略。

多次运行，发现每次运行的结果中，父进程和子进程执行顺序相同。



去除wait后，发现父进程和子进程执行顺序可能不相同。



其中父进程优先执行，可能是因为linux中2.6.32版本后的默认设置

不安全 | http://lxr.linux.no/linux+v2.6.32/kernel/sched_fair.c#L50

```
23  #include <linux/latencytop.h>
24
25  /*
26   * Targeted preemption latency for CPU-bound tasks:
27   * (default: 5ms * (1 + ilog(ncpus)), units: nanoseconds)
28   *
29   * NOTE: this latency value is not the same as the concept of
30   * 'timeslice length' - timeslices in CFS are of variable length
31   * and have no persistent notion like in traditional, time-slice
32   * based scheduling concepts.
33   *
34   * (to see the precise effective timeslice length of your workload,
35   *  run vmstat and monitor the context-switches (cs) field)
36   */
37  unsigned int sysctl_sched_latency = 5000000ULL;
38
39  /*
40   * Minimal preemption granularity for CPU-bound tasks:
41   * (default: 1 msec * (1 + ilog(ncpus)), units: nanoseconds)
42   */
43  unsigned int sysctl_sched_min_granularity = 1000000ULL;
44
45  /*
46   * is kept at sysctl_sched_latency / sysctl_sched_min_granularity
47   */
48  static unsigned int sched_nr_latency = 5;
49
50  /*
51   * After fork, child runs first. If set to 0 (default) then
52   * parent will (try to) run first.
53   */
```

原因可能是在没有 “\n” 的情况下，printf 时只是将内容写入了缓冲区中；在调用exit()时，会将此进程中的缓冲区全部刷新并输出。故两者可能会有区别。

a) 观察进程调度中的全局变量改变

添加一个全局变量并在父进程和子进程中对这个变量做不同操作，输出操作结果和两种变量的地址：

```
// global variable
int gla = 99;
```

```
else if (pid_fork_return == 0) { /* child process */

    pid_here = getpid();

    printf("child: pid_fork_return =%d  ", pid_fork_return);
    printf("child: pid_here =%d  ", pid_here);
    gla++;
    printf("child: global here =%d  ", gla);
    printf("child: global location =%p  ", &gla);
}

else { /* parent process */
    pid_here = getpid();
    printf("parent: pid_fork_return = %d  ", pid_fork_return);
    printf("parent: pid_here =%d  ", pid_here);
    gla--;
    printf("parent: global here =%d  ", gla);
    printf("parent: global location =%p  ", &gla);

//  wait(NULL);
}
```

```
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ ./main_global
parent: pid_fork_return = 6038  parent: pid_here =6037  parent: global here =98
parent: global location =0x55cd6d0b9010  child: pid_fork_return =0  child: pid_he
re =6038  child: global here =100  child: global location =0x55cd6d0b9010  xjtuhp
c@ubuntu:~/Code/OS_exp/source_1$
```

原因是在fork时全局变量也一并复制，全局变量的初始值相同，后面由于不同的进程中进行了不同的操作而导致全局变量的最终值不同。两者的全局变量地址同。

b) 在return前增加对全局变量的操作并输出结果，观察并解释

```
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ ./main_global
parent: pid_fork_return = 6163  parent: pid_here =6162  parent: global here =98
parent: global location =0x56073f64e010  pid_here =6162  global here =196  child:
 pid_fork_return =0  child: pid_here =6163  child: global here =100  child: globa
l location =0x56073f64e010  pid_here =6163  global here =200  xjtuhpc@ubuntu:~/Co
de/OS_exp/source_1$
```

能够看到，两个进程都对全局变量进行了操作。两个进程在return前对全局变量的操作，是对自己的那个全局变量进行的。

c) 在子进程中调用system函数

system()函数可以启动一个新的进程。

它会建立一个独立的进程，这个进程拥有独立的代码空间，内存空间。

它必须用一个shell来启动需要的程序。这样对shell的安装情况，以及shell的版本依赖性很大。

system()函数实际上就是先执行了fork函数，然后新产生的子进程立刻执行了exec函数。这就意味着新进程的PID、PPID等与原进程不同。system也会产生新的进程空间，而且新的进程空间是为新的程序准备的，所以和原进程的进程空间没有任何关系（不像fork新进程空间是对原进程空间的一个复制）。

system函数代码中else部分执行了wait函数，这就意味着，原进程会等待新的进程执行完毕，system才返回。也就是说，它是阻塞的。

```
int system (const char *string )
```

这个函数的效果就相当于执行

```
sh -c string。
```

子进程中，使用system函数来执行指定命令，运行程序Hello：

```
else if (pid_fork_return == 0) { /* child process */

    pid_here = getpid();

    printf("child: pid_fork_return =%d  ", pid_fork_return);
    printf("child: pid_here =%d  ", pid_here);

    int status;
    status = system("./Hello");
    if (-1 == status) {
        printf("system error!");
    } else {
        printf("exit status value = [0x%x]\n", status);
    }
}

else { /* parent process */
    pid_here = getpid();
    printf("parent: pid_fork_return = %d  ", pid_fork_return);
    printf("parent: pid_here =%d  ", pid_here);
    wait(NULL);
}
```

程序Hello：

source_1 > C Hello.c > ...

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

int main() {
    printf("Hello World from another executable!\n");
    pid_t pid_here = getpid();
    printf("child programme: pid_here =%d  ", pid_here);
    return 0;
}

```

运行结果：

```
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ ./main_system
Hello World from another executable!
child programme: pid_here =6795  child: pid_fork_return =0  child: pid_here =6793  exit status value = [0x0]
parent: pid_fork_return = 6793  parent: pid_here =6792  xjtuhpc@ubuntu:~/Code/OS_exp/source_1$
```

d) 在子进程中调用exec族函数

exec函数可以用来替换进程映像。当进程调用某种exec函数时，原来的进程将不再执行，源进程完全由新程序代换，而新程序则从其main函数开始执行。

因为调用exec并不创建新进程，所以前后的进程ID并未改变。新的进程的PID、PPID和nice值与原先的完全一样。

子进程中，使用exec组函数execl来覆盖生成的子进程。

```c
    else if (pid_fork_return == 0) { /* child process */

        pid_here = getpid();

        printf("child: pid_fork_return =%d  ", pid_fork_return);
        printf("child: pid_here =%d  ", pid_here);

        if (execl("./Hello","attribute1",NULL)<0){
        // if (execl("/bin/ls", "ls","-a", NULL)<0){
            printf("execl error  ");
        }
    }

    else { /* parent process */
        pid_here = getpid();
        printf("parent: pid_fork_return = %d  ", pid_fork_return);  /* C */
        printf("parent: pid_here =%d  ", pid_here); /* D */
        int a =20;
    //  wait(NULL);
    }
```

运行结果：

```
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ ./main_exec
parent: pid_fork_return = 6808  parent: pid_here =6807  xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ Hello World f
rom another executable!
child programme: pid_here =6808
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$
```

exec族函数信息如下

所要头文件	#include<unistd.h>
函数原型	int execl(const char *path, const char *arg,...)
	int execv(const char *path, char *const argv[])
	int execle(const char *path, const char *arg,...,char *const envp[])
	int execve(const char *path, char *const argv[], char *const envp[])
	int execlp(const char *file, const char *arg,...)
	int execvp(const char *file, char *const argv[])
函数返回值	-1:错误

l : 使用参数列表
p：使用文件名，并从PATH环境进行寻找可执行文件
v：应先构造一个指向各参数的指针数组，然后将该数组的地址作为这些函数的参数。
e：多了envp[]数组，使用新的环境变量代替调用进程的环境变量

exec系列函数区别：

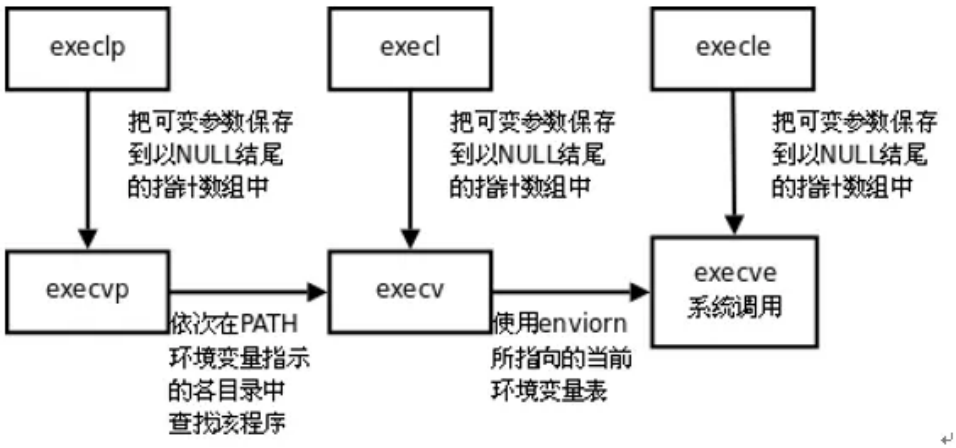
1，带l的一类exac函数（l表示list），包括execl、execlp、execle，要求将新程序的每个命令行参数都说明为 一个单独的参数。这种参数表以空指针结尾。

2，带p的一类exac函数，包括execlp、execvp、execvpe，如果参数file中包含/，则就将其视为路径名，否则就按 PATH环境变量，在它所指定的各目录中搜寻可执行文件。

3，带v不带l的一类exac函数，包括execv、execvp、execve，应先构造一个指向各参数的指针数组，然后将该数组的地址作为这些函数的参数。
如char *arg[]这种形式，且arg最后一个元素必须是NULL，例如char *arg[] = {“ls”,”-l”,NULL};

4，带e的一类exac函数，包括execle、execvpe，可以传递一个指向环境字符串指针数组的指针。 参数例如char *env_init[] = {“AA=aa”,”BB=bb”,NULL}; 带e表示该函数取envp[]数组，而不使用当前环境。

- 查找方式：上表中前4个函数的查找方式都是完整的文件目录路径(即绝对路径)，而最后两个函数(也就是以p结尾的两个函数)可以只给出文件名，系统就会自动从环境变量“$PATH”所指出的路径中进行查找。
- 参数传递方式：有两种方式，一种是逐个列举的方式，另一种是将所有参数整体构造成一个指针数组进行传递。（在这里，字母“l”表示逐个列举的方式，字母“v”表示将所有参数整体构造成指针数组进行传递，然后将该数组的首地址当做参数传递给它，数组中的最后一个指针要求时NULL）
- 环境变量：exec函数族使用了系统默认的环境变量，也可以传入指定的环境变量。这里以“e”结尾的两个函数就可以在envp[]中指定当前进程所使用的环境变量替换掉该进程继承的所有环境变量。



exec 函数族

二、线程相关编程实验

1、在进程中给一变量赋初值并创建两个线程；

2、在两个线程中分别对此变量循环五千次以上做不同的操作并输出结果；

```
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ ./thread0
At first var = 5000
In thread 1 var =4974
In thread 2 var =4992
after pthread_create, var = 4992
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ gcc thread0.c -lpthread -o thread0
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ ./thread0
At first var = 5000
In thread 2 var =5050
In thread 1 var =5010
after pthread_create, var = 5010
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ gcc thread0.c -lpthread -o thread0
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ ./thread0
At first var = 5000
In thread 1 var =4994
In thread 2 var =4996
after pthread_create, var = 4996
```

3、多运行几遍程序观察运行结果，如果发现每次运行结果不同，请解释原因并修改程序解决，考虑如何控制互斥和同步；

加上互斥锁：

```c
//初始化锁
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int var = 5000;

void *thr1(void *arg){
    int i;
    for (i = 0; i < 5000; i++)    {
        pthread_mutex_lock(&mutex);//加锁，当有线程已经加锁的时候会阻塞
        var--;
        sleep(0.00001);
        pthread_mutex_unlock(&mutex);//释放锁
    }
    printf("In thread 1 var =%d \n",var);

}

void *thr2(void *arg){
    int j;
    for (j = 0; j < 5000; j++)    {
        pthread_mutex_lock(&mutex);//加锁，当有线程已经加锁的时候会阻塞
        var++;
        sleep(0.00001);
        pthread_mutex_unlock(&mutex);//释放锁
    }
    printf("In thread 2 var =%d \n",var);
}
```

结果：

```
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ gcc thread1.c -lpthread -o thread1
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ ./thread1
At first var = 5000
In thread 1 var =625
In thread 2 var =5000
after pthread_create, var = 5000
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$
```

4、 将任务一中第一个实验调用system函数和调用exec族函数改成在线程中实现，观察运行结果输出进程PID与线程TID进行比较并说明原因。

exec：

```c
void *fun(void *arg)
{
    printf("child, the tid=%lu, pid=%ld\n",pthread_self(),syscall(SYS_gettid));
    printf("child, getpid()=%d\n",getpid());

    if (execl("./Hello2","attribute1",NULL)<0){
        printf("execl error  ");
    }


    // int status;
    // status = system("./Hello2");
    // if (-1 == status) {
    //     printf("system error!");
    // } else {
    //     printf("exit status value = [0x%x]\n", status);
    // }

    return NULL;
}
```

```
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ gcc thread2.c -lpthread -o thread2
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ ./thread2
child, the tid=140615623141120, pid=9174
child, getpid()=9173
Hello World from another executable!
child, the tid=1387373888, pid=9173
child, getpid()=9173
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$
```

system：

```c
void *fun(void *arg)
{
    printf("child, the tid=%lu, pid=%ld\n",pthread_self(),syscall(SYS_gettid));
    printf("child, getpid()=%d\n",getpid());

    // if (execl("./Hello2","attribute1",NULL)<0){
    //     printf("execl error  ");
    // }


    int status;
    status = system("./Hello2");
    if (-1 == status) {
        printf("system error!");
    } else {
        printf("exit status value = [0x%x]\n", status);
    }

    return NULL;
}
```

```
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ gcc thread2.c -lpthread -o thread2
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$ ./thread2
child, the tid=140345171203840, pid=9205
child, getpid()=9204
Hello World from another executable!
child, the tid=783570240, pid=9207
child, getpid()=9207
exit status value = [0x0]
I am main, my pid = 9204
xjtuhpc@ubuntu:~/Code/OS_exp/source_1$
```

getpid()得到的是进程的pid，在内核中，每个线程都有自己的PID，要得到线程的PID,必须用syscall(SYS_gettid);

pthread_self函数获取的是线程ID，线程ID在某进程中是唯一的，在不同的进程中创建的线程可能出现ID值相同的情况。

实验中的问题与解决过程

1.问题描述

主要是在ubuntu虚拟机中，主函数不使用wait时，多次运行程序，进程执行顺序较为固定。在使用wait时先执行完child，不使用wait时总是先执行完parent函数再执行child。

2.解决过程：（网址，参考资料），具体解决方法

最终判断可能是系统调度策略不同引起的结果。改为使用openeuler系统后执行结果较为符合预期。