

# 操作系统实验1

徐中伟  
计算机001  
2206515211

# 任务1：华为云上openEuler操作系统环境

## 1.1题目1:教材p103&&打印地址

## 1.2题目分析：

首先，这道题主要考察fork（）函数的使用以及相关的特性，其次，还要来观察父子进程之间的进程号的关系，最后，是对wait（）函数的分析。（因为相关代码比较简单，只以注释形式给出）。

在进行实验前：我的猜测是父进程会fork一个子进程，fork在父进程中返回子进程pid，子进程中pid的值为0。wait（）是父进程来等待子进程结束的函数，否则，父进程可能会提前结束。

问题：为什么子进程的pid值为0呢？猜测：为了区别父子进程，便于程序员编程。

## 1.3源代码&&实验结果：

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

int main()
{
    pid_t pid, pid1;
    pid = fork();
    if(pid<0)
    {
        fprintf(stderr,"fork failure");
    }else if(pid == 0){
        pid1 = getpid();
    }
```

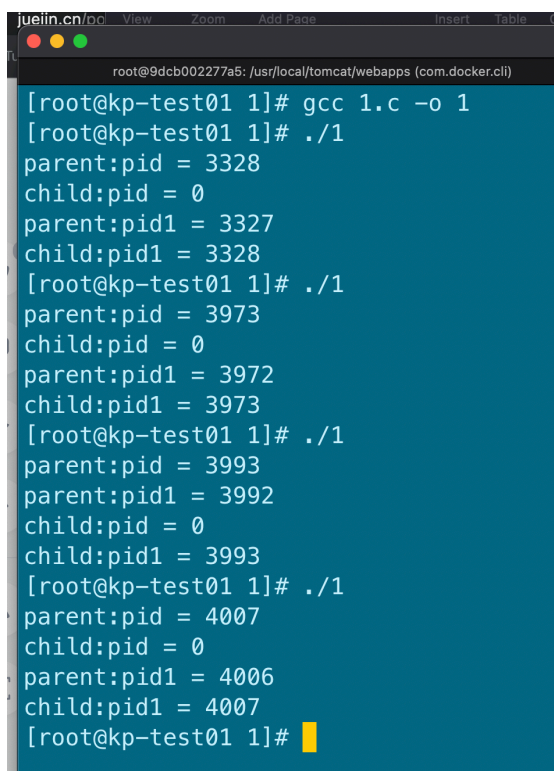
```

        printf("child:pid = %d\n",pid);
        printf("child:pid1 = %d\n",pid1);
    }else{
        pid1 = getpid();
        printf("parent:pid = %d\n",pid);
        printf("parent:pid1 = %d\n",pid1);
        wait(NULL);
    }

    return 0;
}

```

### 1.3.1原函数运行结果：



```

juejin.cn/doc View Zoom Add Page Insert Table C
root@9dcb002277a5: /usr/local/tomcat/webapps (com.docker.cli)

[root@kp-test01 1]# gcc 1.c -o 1
[root@kp-test01 1]# ./1
parent:pid = 3328
child:pid = 0
parent:pid1 = 3327
child:pid1 = 3328
[root@kp-test01 1]# ./1
parent:pid = 3973
child:pid = 0
parent:pid1 = 3972
child:pid1 = 3973
[root@kp-test01 1]# ./1
parent:pid = 3993
parent:pid1 = 3992
child:pid = 0
child:pid1 = 3993
[root@kp-test01 1]# ./1
parent:pid = 4007
child:pid = 0
parent:pid1 = 4006
child:pid1 = 4007
[root@kp-test01 1]#

```

分析：从图中不难看出，进程号随着运行的次数加大不断加大，猜测应该是诸如%mod的方式来分配pid。

但都具有一个规律：即child pid为0, child中显示getpid()  
( ) 得到的值和父进程一样。

### 1.3.2验证去除wait () 后的实验

```
[root@kp-test01 1]# gcc 1.c -o 1
[root@kp-test01 1]# ./1
parent:pid = 4458
child:pid = 0
parent:pid1 = 4457
child:pid1 = 4458
[root@kp-test01 1]# ./1
parent:pid = 4472
child:pid = 0
parent:pid1 = 4471
child:pid1 = 4472
[root@kp-test01 1]# ./1
parent:pid = 4492
parent:pid1 = 4491
child:pid = 0
child:pid1 = 4492
[root@kp-test01 1]#
```

分析：从实验结果上不难看出其实前后的差别并不是很大，无法体现出wait的作用，因为子进程结束的太快了,如果子进程。

查阅网上资料可知，父进程结束不一定会杀死子进程。（每个进程都会属于一个进程组，每个进程组有个组长，组长的进程ID即是该进程组的ID。然后每个会话会拥有多个进程组，但是只会有一个前台进程组，其他的都是后台进程组。该前台进程组的ID即是这个会话的ID。也就是说如果一个进程是会话首进程，那么他的进程ID等于所在进程组的ID，也等于所在会话的ID。）

### 1.2.1打印地址：

### 1.2.2源代码&&实验结果：

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

int a;

int main()
{
    pid_t pid, pid1;
    pid = fork();
    int b;
    if(pid<0)
    {
        fprintf(stderr,"fork failure");
    }else if(pid == 0){
        a+=1;
        b+=1;
        printf("parent:a = %d\n",a);
        printf("parent:b = %d\n",b);
        printf("addr of a in child = %d\n",&a);
        printf("addr of b in child = %d\n",&b);
    }else{
        a*=-1;
        b*=-1;
        printf("parent:a = %d\n",a);
        printf("parent:b = %d\n",b);
        printf("addr of a in parent = %d\n",&a);
        printf("addr of b in parent = %d\n",&b);
        wait(NULL);
    }
}
```

```

        a*=2;

        printf("final value of a = %d\n",a);

    return 0;

}

```

```

[root@kp-test01 1]# gcc sup.c -o sup
[root@kp-test01 1]# ./sup
parent:a = 0
parent:b = 0
child:a = 1
addr of a in parent = 4325460
child:b = 1
addr of b in parent = -517909928
addr of a in child = 4325460
addr of b in child = -517909928
final value of a = 2
final value of a = 0
[root@kp-test01 1]# ./sup
parent:a = 0
parent:b = 0
addr of a in parent = 4325460
addr of b in parent = -826326136
child:a = 1
child:b = 1
addr of a in child = 4325460
addr of b in child = -826326136
final value of a = 2
final value of a = 0
[root@kp-test01 1]#

```

### 1.2.3分析:

父进程创建子进程时，代码、数据、地址空间等进行复制，因此父子进程各自独享n，各自的操作互不影响，因而对于初始值为0的a，子进程加1后输出1，父进程乘-1后输出0；由于子进程对父进程的地址空间进行拷贝，父子进程的n的虚拟地址是一样的（物理地址不同），而打印出来的地址是虚拟地址，因此父子进程输出结果相同。

### 1.3.1关于exec的实验

分析：父进程在给予子进程分配了相关的寄存器，代码，数据后，默认自己导入自己的代码，如果使用exec函数，则可以将指定的函数代码导入到子进程的代码段中，然后执行相关的代码。

### 1.3.2源代码&&实验结果

```

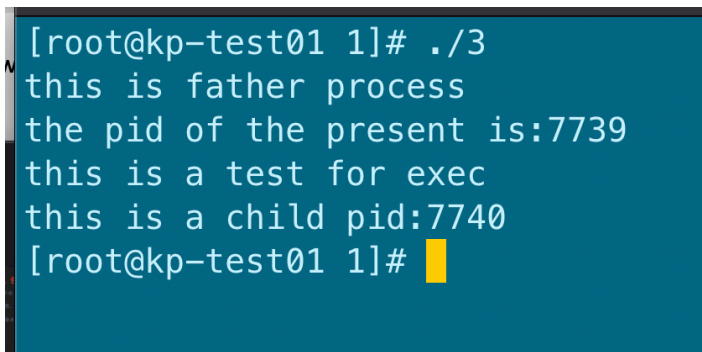
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>

```

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

int main()
{
    pid_t pid;
    pid = fork();
    if(pid<0)
    {
        fprintf(stderr,"fork failure");
    }else if(pid == 0){
        char *arg[] = {NULL};
        execvp("./4", arg);
        printf("error exec\n");
    }else{
        printf("this is father process\n");
        printf("the pid of the present is:%d\n",getpid());
    }

    return 0;
}
```



```
[root@kp-test01 1]# ./3
this is father process
the pid of the present is:7739
this is a test for exec
this is a child pid:7740
[root@kp-test01 1]#
```

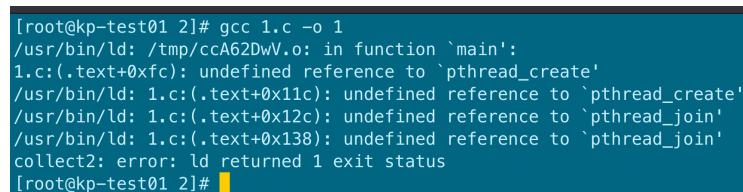
实验结果和预期一样，其中，`getpid()` 所打印出来的也刚好是子进程的pid。

## 二、线程相关编程实验

### 2.1.1 创建两个线程运行后体会线程共享进程信息、线程对共享变量操作中同步与互斥的知识。

分析：进程往往会创建不同的线程以达到并行的目的，所以线程的用处很大，但是同时也要考虑到相关的诸如临界资源的访问等问题。

### 2.1.2 源代码&&运行结果：



```
[root@kp-test01 2]# gcc 1.c -o 1
/usr/bin/ld: /tmp/ccA62DwV.o: in function `main':
1.c:(.text+0xfc): undefined reference to `pthread_create'
/usr/bin/ld: 1.c:(.text+0x11c): undefined reference to `pthread_create'
/usr/bin/ld: 1.c:(.text+0x12c): undefined reference to `pthread_join'
/usr/bin/ld: 1.c:(.text+0x138): undefined reference to `pthread_join'
collect2: error: ld returned 1 exit status
[root@kp-test01 2]#
```

遇到问题：编译错误，解决，使用-pthread来动态链接

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<pthread.h>
#include<stdlib.h>
int n=1;
void *count1(){
    int i=0;
    for(i=0;i<5000;i++)
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<pthread.h>
#include<stdlib.h>
int n=1;
void *count1(){
    int i=0;
    for(i=0;i<5000;i++)
    {
        printf("%d\n",n++);
    }
    return (void*)0;
}
void *count2(){
    int i=0;
    for(i=0;i<5000;i++)
    {
        printf("%d\n",n--);
    }
    return (void*)0;
}
int main()
{
    pthread_t tid1,tid2;
    void *ptr = NULL;
    int ret1 = pthread_create(&tid1,NULL,count1,NULL);
    int ret2 = pthread_create(&tid2,NULL,count2,NULL);
    pthread_join(tid1,ptr);
    pthread_join(tid2,ptr);
    if(ret1!=0 || ret2!=0){
        printf("error\n");
        exit(1);
    }

    return 0;
}
```



```
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
```

```
Inse -27
-26
-25
-24
-23
-22
-21
-20
-19
-18
-17
-16
-15
-14
-13
-12
-11
-10
-9
-8
-7
-6
-5
-4
-3
-2
-1
```

```
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
```

## 线程id

```
-1
0
the thread id = 3133796832
```

分析：明显发现几次运行程序的结果不一样，且结果也不为0（根据实验的初衷，分别加减5000次将会导致结果最终为0，这是因为没有做好线程的互斥，于是设计用信号量的方式实现相关的同步与互斥）

## 源代码&&分析

```

[root@kp-test01 1]# ./sup
parent:a = 0
parent:a = 1
parent:b = 0
parent:b = 1
addr of a in parent = 4325460
addr of a in child = 4325460
addr of b in parent = -337443096
addr of b in child = -337443096
final value of a = 2
final value of a = 0
[root@kp-test01 1]# ./sup
parent:a = 0
parent:a = 1
parent:b = 0
parent:b = 1
addr of a in parent = 4325460
addr of a in child = 4325460
addr of b in parent = -615921976
addr of b in child = -615921976
final value of a = 2
final value of a = 0

```

分析：初始两个信号量分别为0和1，然后在进入访问临界变量的时候，锁住另一个线程，同时，在结束对某一临界变量的访问时将唤醒另一临界变量的访问。

```

#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>
#include<pthread.h>
#include<stdlib.h>
#include<semaphore.h>

```

```
sem_t single1,single2;
```

```
int n=1;
```

```

void *count1(){
    int i=0;
    for(i=0;i<5000;i++)
    {
        sem_wait(&single1);
        printf("%d\n",n++);
        sem_post(&single2);

    }
    return (void*)0;
}
void *count2(){
    int i=0;
    for(i=0;i<5000;i++)
    {
        sem_wait(&single2);
        printf("%d\n",n--);
        sem_post(&single1);
    }
    return (void*)0;
}
int main()
{

    sem_init(&single1,0,1);
    sem_init(&single2,0,0);

    pthread_t tid1,tid2;
    void *ptr = NULL;
    int ret1 = pthread_create(&tid1,NULL,count1,NULL);
    int ret2 = pthread_create(&tid2,NULL,count2,NULL);
    pthread_join(tid1,ptr);
    pthread_join(tid2,ptr);
    if(ret1!=0 || ret2!=0){
        printf("error\n");
        exit(1);
    }

    return 0;
}

```

[illegible]