

实验2：进程及线程创建

一、实验目的

理解创建子进程函数的fork()的用法，通过观察运行结果理解进程的基本特征；通过代码及运行结果理解线程的概念，能够理解进程与线程之间的关联。

二、实验方法

本次实验属于验证型实验，按照实验内容的指导完成所有实验步骤，并记录下实验结果。利用C语言创建子进程与线程等函数，在deepin的terminal中运行，以实现探究目的。

三、实验内容

1.使用编辑器gedit新建一个helloProcess.c源文件，并输入后面的范例代码。

实验过程：

(1)利用gedit创建文本以编写如下C语言代码。

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main()
6 {
7     //pid_t是数据类型，实际上是一个整型，通过typedef重新定义了一个名字，用于存储进程id
8     pid_t pid,cid;
9     //getpid()函数返回当前进程的id号
10    printf("Before fork Process id :%d\n", getpid());
11
12    /*
13     fork()函数用于创建一个新的进程，该进程为当前进程的子进程，创建的方法是：将当前进程的内存内容完整拷贝一份到内存的另一个区域，两个进程为父子关系，他们会同时（并发）执行fork()语句后面的所有语句。
14     fork()的返回值：
15     ... 如果成功创建子进程，对于父子进程fork会返回不同的值，对于父进程它的返回值是子进程的进程id值，对于子进程它的返回值是0。
16     ... 如果创建失败，返回值为-1。
17     */
18    cid = fork();
19
20    printf("After fork, Process id :%d\n", getpid());
21
22    return 0;
23 }
```

(2) 保存退出gedit，使用gcc对源文件进行编译，然后运行，结果如下图所示：

```
navy@navy-PC: ~/Desktop$ ./helloProcess
Before fork Process id:44094
After fork,Process id:44094
After fork,Process id:44095
navy@navy-PC: ~/Desktop$
```

使用gcc对源文件进行编译，然后运行，观察结果并解释原因。

现象：

第一条printf显示出当前进程ID号为“44094”；

第二条printf则显示出两次，并且为不同的进程ID号

分析：

由于fork（）使得创建一个子进程，第二个printf运行了两次，分别显示了父进程与子进程的进程ID。

2.练习ps命令，该命令可以列出系统中当前运行的进程状态，我们在上面代码的21行处加入下面两行语句，目的是让父子进程暂停下来，否则我们无法观测到他们运行时的状态

实验过程：

(1) 键入如下代码并保存，在Terminal中重新连接，并运行。

```
int i;
scanf("%d",&i);
```

```
navy@navy-PC: ~/Desktop$ ./helloProcess
Before fork Process id:48570
After fork,Process id:48570
After fork,Process id:48571
```

3. 通过判断fork的返回值让父子进程执行不同的语句。

如上图所示，运行后并未退出，需键入ctrl+C退出。这也从侧面说明父子进程暂停了下来,并未结束。也可用pause()达到同样的效果。

(2) 由于此时进程已中止，故应新打开一个Terminal页面，输入ps -al查看当前进程。

```
navy@navy-PC: ~/Desktop$ ps -al
F S  UID  PID  PPID  C  PRI  NI  ADDR  SZ  WCHAN  (tty) 实验2：进程TIME CMD ht
0 S  1000  51263  51197  0  80   0  -  1071  wait_w  pts/2    00:00:00  hello
1 S  1000  51264  51263  0  80   0  -  1071  n_tty_  pts/2    00:00:00  hello
4 R  1000  51272  51267  0  80   0  -  7304  -      pts/3    00:00:00  ps
navy@navy-PC: ~/Desktop$
```

返回值让父子进程执行不同的语句。

现象：如图所示，第二行的PPID与第一行的PID相同，说明第二行的这个进程为第一行进程的子进程。

3.通过判断fork的返回值让父子进程执行不同的语句。

实验过程：

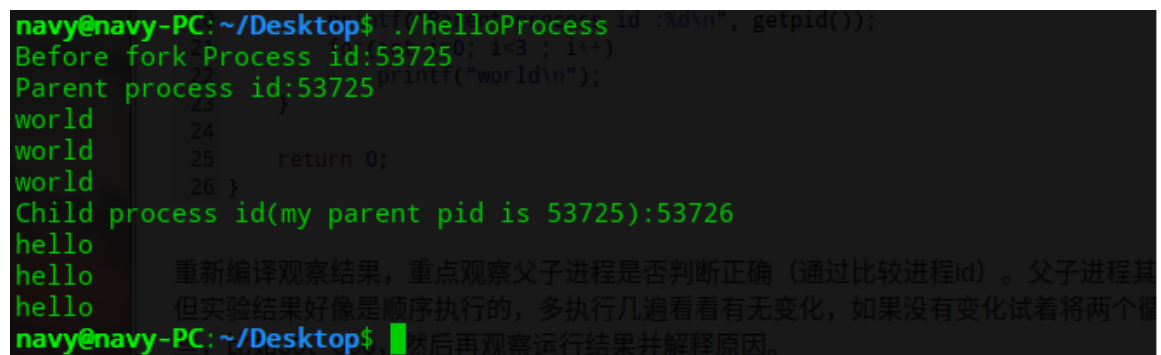
(1) 键入如下代码并保存，在Terminal中重新连接，并运行。

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main()
6 {
7     pid_t cid;
8     printf("Before fork process id :%d\n", getpid());
9
10    cid = fork();
11
12    if(cid == 0){ //该分支是子进程执行的代码
13
14        printf("Child process id (my parent pid is %d):%d\n", getppid(),getpid());
15        for(int i=0; i<3 ; i++)
16            printf("hello\n");
17    }else{ //该分支是父进程执行的代码
18
19        printf("Parent process id :%d\n", getpid());
20        for(int i=0; i<3 ; i++)
21            printf("world\n");
22    }
23
24    return 0;
25 }
26 }

```

结果如下图所示：



```

navy@navy-PC: ~/Desktop$ ./helloProcess
Before fork process id :53725
Parent process id:53725
world
world
world
Child process id(my parent pid is 53725):53726
hello
hello
hello

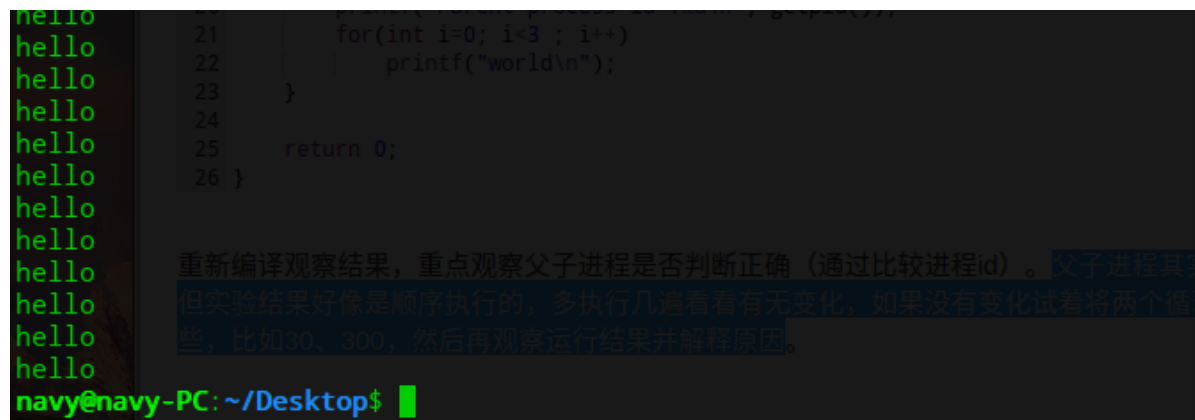
```

重新编译观察结果，重点观察父子进程是否判断正确（通过比较进程id）。父子进程其
但实验结果好像是顺序执行的，多执行几遍看看有无变化，如果没有变化试着将两个循
navy@navy-PC: ~/Desktop\$

现象：第一行输出fork()之前的进程ID；接下来的条件语句分别执行如上图所示，说明先是父进程，然后是子进程。Parent process id为“53725”，Child process id为“53726”，判断正确。

(2) 父子进程其实是并发执行的，但实验结果好像是顺序执行的，多执行几遍看看有无变化，如果没有变化试着将两个循环的次数调整高一些，分别尝试300、3000，然后再观察运行结果并解释原因。

当i=300时：（下图为部分截图）



```

21     for(int i=0; i<3 ; i++)
22         printf("world\n");
23 }
24
25 return 0;
26 }

```

重新编译观察结果，重点观察父子进程是否判断正确（通过比较进程id）。父子进程其
但实验结果好像是顺序执行的，多执行几遍看看有无变化，如果没有变化试着将两个循
些，比如30、300，然后再观察运行结果并解释原因。

navy@navy-PC: ~/Desktop\$

现象：一次性执行完条件语句各分支内的所有循环（由于计算机性能较好，暂时表现不出父子进程的并发）

当i=3000时：（下图为部分截图）

```
hello
world
hello
world
hello
world
hello
19
20     printf("Parent process id :%d\n", getpid());
21     for(int i=0; i<3 ; i++)
22         printf("world\n");
23     }
24
25     return 0;
```

现象：“world”与“hello”交替出现，表现出父子进程的并发。

4.验证父子进程间的内存空间是相互独立的。在终端中进入自己的主目录，使用gedit命令新建一文件helloProcess2.c，输入下面的代码，然后编译运行，解释其原因。

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <unistd.h>
4
5 int main()
6 {
7     pid_t cid;
8     int x = 100;
9
10    cid = fork();
11
12    if(cid == 0){ //该分支是子进程执行的代码
13        x++;
14        printf("In child: x=%d\n",x);
15    }else{ //该分支是父进程执行的代码
16        x++;
17        printf("In parent: x=%d\n",x);
18    }
19
20    return 0;
21 }
22
23
24 }
```

结果如下图所示：

```
navy@navy-PC: ~$ ./helloProcess2
In parent:x=101
In child:x=101
```

现象：父进程和子进程先后执行，由x均为101，说明子进程的初始内容由父进程克隆而来。

5.在上一步的代码的20行添加如下语句，同时代码最顶端要包含一个新的头文件。

实验过程：

（1）键入如下代码。wait函数会让调用者陷入等待，直到子进程的状态变为可用（即子进程结束前父进程一直处于等待状态）；sleep该函数可以让调用进程睡上指定的时间长度（单位是second）。重新编译代码运行，我们特意让子进程输出完毕后睡了3秒，在这期间父进程什么事也没有做一直在wait，直到子进程结束后父进程才执行printf语句

```
#include<sys/wait.h>
sleep(3);
wait(NULL);
```

```
打开(O)  helloProcess2.c 保存(S)  ~/  
*无标题文档 1  helloProcess2.c  
#include<stdio.h>  
#include<sys/types.h>  
#include<unistd.h>  
#include<sys/wait.h>  
int main()  
{  
    pid_t cid;  
    int x=100;  
    cid =fork();  
    if(cid==0){  
        x++;  
        printf("In child:x=%d\n",x);  
        sleep(3);  
    }else{  
        x++;  
        wait(NULL);  
        printf("In parent:x=%d\n",x);  
    }  
    return 0;  
}
```

结果如下图所示：相隔三秒输出

```
navy@navy-PC:~$ ./helloProcess2  
In child:x=101  
In parent:x=101
```

6.创建线程。先关闭先前的文件，gedit helloThread.c以创建一个新的C语言源文件，将下面的代码拷贝进编辑器

实验过程：

(1) 创建新的C语言源文件，键入下列代码：

```
1 #include <sys/types.h>  
2 #include <unistd.h>  
3 #include <stdio.h>  
4 #include <pthread.h>  
5  
6 void* threadFunc(void* arg){ //线程函数  
7  
8     printf("In NEW thread\n");  
9  
10 }  
11  
12 int main()  
13 {  
14     pthread_t tid;  
15     pthread_create(&tid, NULL, threadFunc, NULL);  
16  
17     //pthread_join(tid, NULL);  
18  
19     printf("In main thread\n");  
20  
21     return 0;  
22 }  
23 }
```

并在terminal中 输入下列命令并执行。

```
gcc helloThread.c -o helloThread -l pthread
```

结果如下图所示：

```
navy@navy-PC:~/Desktop$ gcc helloThread.c -o helloThread -l pthread
navy@navy-PC:~/Desktop$ ./helloThread
In main thread
```

现象：只输出“In main thread”。由于main()中线程执行后没有等待子线程“threadFunc”就返回值，导致子线程不返回“In New thread”。

(2)将第18行代码的注释去掉再次运行。

结果如下图所示：

```
navy@navy-PC:~/Desktop$ gcc helloThread.c -o helloThread -l pthread
navy@navy-PC:~/Desktop$ ./helloThread
In NEW thread
In main thread
```

现象：子/父线程的结果依次输出。说明父线程有等待子线程的运行结束（第18行代码类似于上文中的wait()作用，可在子线程中使用sleep()做极端条件下的验证）

(3) 试着在主线程和新线程里加入循环输出，观察一下输出的效果和并发父子进程的执行效果是否相似。

执行下列代码：

```

打开(O)  helloThread.c  保存(S)  ~/Desktop
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<pthread.h>
#include<time.h>
#include<stdlib.h>

void* Func1(void* arg){
    for(int i=0;i<5;++i)
    { printf("英明神武 (%d) \n",rand()%100);
      sleep(1);
    }
}
void* Func2(void* arg){
    for(int i=0;i<5;++i)
    {
        printf("杨老师(%d)\n",rand()%100);
        sleep(2);
    }
}
int main()
{
    srand(time(NULL));
    pthread_t tid,tid2;
    pthread_create(&tid,NULL,Func1,NULL);
    pthread_create(&tid2,NULL,Func2,NULL);
    pthread_join(tid,NULL);
    printf("In main thread\n");
    return 0;
}

```

如下图所示：

```

navy@navy-PC:~/Desktop$ gcc helloThread.c -o helloThread
-pthread
navy@navy-PC:~/Desktop$ ./helloThread
杨老师 (46)
英明神武 (58)
英明神武 (6)
杨老师 (16)
英明神武 (13)
英明神武 (12)
杨老师 (98)
英明神武 (92)
杨老师 (94)
杨老师 (60)
In main thread
navy@navy-PC:~/Desktop$

```

现象：“杨老师”和“英明神武”交替出现，说明父子线程并发执行。

四、总结

无

