实验2：进程及线程创建

网安1901 201904080104 曹颖

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，遇到不懂的问题或是在某一步骤上卡壳，先尝试在搜索引擎上寻找解决方法，积极与老师、同学沟通，务必亲自将实验完成。

三、实验内容

1. 使用编辑器sublime新建一个2.c源文件，并输入后面的范例代码。

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

int main()
{
    //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
    pid_t pid,cid;
    //getpid()函数返回当前进程的id号
    printf("Before fork Process id :%d\n", getpid());

    /*
    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。
    fork()的返回值：
      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.
      如果创建失败，返回值为-1.
    */
    cid = fork();

    printf("After fork, Process id :%d\n", getpid());

    return 0;
}
```

2.c

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    //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
    pid_t pid,cid;
    //getpid()函数返回当前进程的id号
    printf("Before fork Process id :%d\n", getpid());
    /*
    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。
    fork()的返回值：
        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.
        如果创建失败，返回值为-1.
    */
    cid = fork();
    printf("After fork, Process id :%d\n", getpid());
    return 0;
}
```

保存退出gedit，使用gcc对源文件进行编译，然后运行，观察结果并解释原因。

```
cyhhhh@cyhhhh-PC:~$ cd Desktop
cyhhhh@cyhhhh-PC:~/Desktop$ gcc 2.c -o 2
cyhhhh@cyhhhh-PC:~/Desktop$ ls
2  2.c  code  com.sublimetext.2.desktop  cy1  cy1.c  deepin-terminal.desktop
cyhhhh@cyhhhh-PC:~/Desktop$ ./2
Before fork Process id :5175
After fork, Process id :5175
After fork, Process id :5176
cyhhhh@cyhhhh-PC:~/Desktop$
```

> 如图上结果，在fork()函数前打印出来pid为5175，而在fork()函数后，有两条结果输出，因为fork()函数是创建一个子进程，而父进程与子进程的pid不一样，且fork()后的代码会被父进程和子进程并发的各执行一遍，所以会有两条pid不同的输出。

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。

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

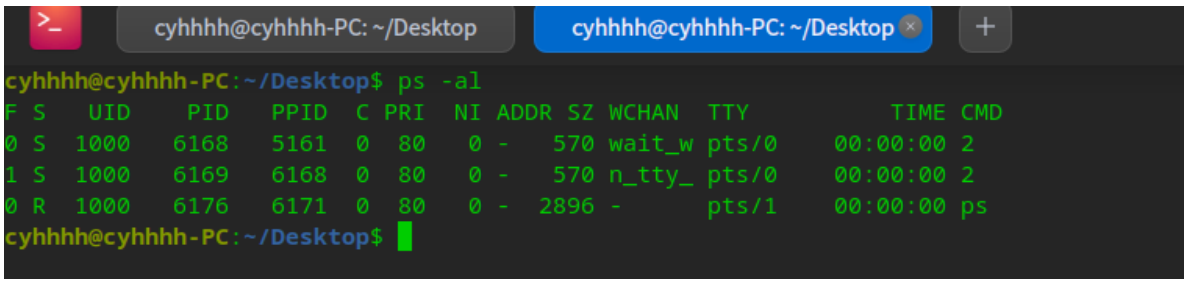
2.c

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    //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
    pid_t pid,cid;
    //getpid()函数返回当前进程的id号
    printf("Before fork Process id :%d\n", getpid());
    /*
    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。
    fork()的返回值：
        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.
        如果创建失败，返回值为-1.
    */
    cid = fork();
    printf("After fork, Process id :%d\n", getpid());
    int i;
    scanf("%d",&i);
    return 0;
}
```

重新编译运行程序，开启一个新的终端窗口输入下面的命令并观察运行结果。

```
ps -al
```

```
cyhhhh@cyhhhh-PC:~/Desktop$ gcc 2.c -o 2
cyhhhh@cyhhhh-PC:~/Desktop$ ./2
Before fork Process id :6168
After fork, Process id :6168
After fork, Process id :6169
```



3. 通过判断fork的返回值让父子进程执行不同的语句。

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

int main()
{
    pid_t cid;
    printf("Before fork process id :%d\n", getpid());

    cid = fork();

   if(cid == 0){ //该分支是子进程执行的代码

       printf("Child process id (my parent pid is %d):%d\n",
getppid(),getpid());
       for(int i=0; i<3 ; i++)
            printf("hello\n");

    }else{ //该分支是父进程执行的代码

        printf("Parent process id :%d\n", getpid());
        for(int i=0; i<3 ; i++)
            printf("world\n");
    }

    return 0;
}
```

```c
    pid_t cid;
    printf("Before fork process id :%d\n", getpid());

    cid = fork();

   if(cid == 0){ //该分支是子进程执行的代码

       printf("Child process id (my parent pid is %d):%d\n", getppid(),getpid());
       for(int i=0; i<3 ; i++)
           printf("hello\n");

    }else{ //该分支是父进程执行的代码

        printf("Parent process id :%d\n", getpid());
        for(int i=0; i<3 ; i++)
           printf("world\n");
    }

    return 0;
}
```

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，比如30、300，然后再观察运行结果并解释原因。

> 循环3次

```
cyhhhh@cyhhhh-PC:~/Desktop$ ./2
Before fork process id :3514
Parent process id :3514
world
world
world
Child process id (my parent pid is 3514):3515
hello
hello
hello
cyhhhh@cyhhhh-PC:~/Desktop$ a
```

> 循环30次

```
Before fork process id :3646
Parent process id :3646
world
world
world
world
world
world
world
world
world
world
world
world
world
world
world
Child process id (my parent pid is 3646):3647
world
hello
world
hello
world
```

> 原因：父子进程是并发执行的，两个进程交替使用CPU。

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。

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

```

```c
int main()
{
    pid_t cid;
    int x = 100;

    cid = fork();

   if(cid == 0){ //该分支是子进程执行的代码
       x++;
       printf("In child: x=%d\n",x);

    }else{ //该分支是父进程执行的代码
       x++;

       printf("In parent: x=%d\n",x);

    }

    return 0;
}
```

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

int main()
{
    pid_t cid;
    int x = 100;

    cid = fork();

   if(cid == 0){ //该分支是子进程执行的代码
       x++;
       printf("In child: x=%d\n",x);

    }else{ //该分支是父进程执行的代码
       x++;

       printf("In parent: x=%d\n",x);

    }

    return 0;
}
```

> 运行结果

```
cyhhhh@cyhhhh-PC:~/Desktop/code$ ./2y
In parent: x=101
In child: x=101
cyhhhh@cyhhhh-PC:~/Desktop/code$
```

> 原因：fork()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，所以x的初始值都是100，而父子进程占用的内存空间相互独立，所以最终值都是101。

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

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

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）。
为了让效果更清楚，请将wait语句从20行移到18行，并在15行加上如下语句：

```
sleep(3);
```

sleep该函数可以让调用进程睡上指定的时间长度（单位是second）。

2y.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){ //该分支是子进程执行的代码
       x++;
       printf("In child: x=%d\n",x);
       sleep(3);
    }else{ //该分支是父进程执行的代码
       x++;
       wait(NULL);
       printf("In parent: x=%d\n",x);

    }

    return 0;
}
```

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。

```
cyhhhh@cyhhhh-PC:~/Desktop/code$ ./2y
In child: 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。

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

void* threadFunc(void* arg){ //线程函数

        printf("In NEW thread\n");

}

int main()
{
    pthread_t tid;

    pthread_create(&tid, NULL, threadFunc, NULL);

    //pthread_join(tid, NULL);

    printf("In main thread\n");

    return 0;
}
```

2y.c 2cy.c

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

void* threadFunc(void* arg){ //线程函数

        printf("In NEW thread\n");

}

int main()
{
    pthread_t tid;

    pthread_create(&tid, NULL, threadFunc, NULL);

    //pthread_join(tid, NULL);

    printf("In main thread\n");

    return 0;
}
```

编译该段代码时，请注意gcc要加入新的参数，命令如下：

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

```
cyhhhh@cyhhhh-PC:~/Desktop/code$ gcc 2cy.c -o 2cy -l pthread
cyhhhh@cyhhhh-PC:~/Desktop/code$ ls
2  2.c  2cy  2cy.c  2y  2y.c  code  cy1  cy1.c
cyhhhh@cyhhhh-PC:~/Desktop/code$ ./2cy
In main thread
cyhhhh@cyhhhh-PC:~/Desktop/code$
```

运行一下观察到什么现象了？将上面第18行代码的注释去掉又观察到了什么现象？为什么？

```
cyhhhh@cyhhhh-PC:~/Desktop/code$ gcc 2cy.c -o 2cy -l pthread
cyhhhh@cyhhhh-PC:~/Desktop/code$ ./2cy
In NEW thread
In main thread
cyhhhh@cyhhhh-PC:~/Desktop/code$
```

> 原因：因为第二次执行了 pthread_join(tid, NULL)；所以调用这个函数的父进程会等子进程结束后再一起结束，子进程printf后，父进程再进行printf，子进程里的“In NEW thread”因此输出。

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。

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

void* threadFunc(void* arg){ //线程函数
    for (int i = 0; i <30; ++i)
    {
        printf("In NEW thread\n");
    }

}

int main()
{
    pthread_t tid;

    pthread_create(&tid, NULL, threadFunc, NULL);

    //pthread_join(tid, NULL);
    for (int i = 0; i < 30; ++i)
    {
        printf("In main thread\n");
    }

}
```

> 结果

```
cyhhhh@cyhhhh-PC:~$ cd Desktop
cyhhhh@cyhhhh-PC:~/Desktop$ cd code
cyhhhh@cyhhhh-PC:~/Desktop/code$ gcc 2cy.c -o 2cy -l pthread
cyhhhh@cyhhhh-PC:~/Desktop/code$ ./2cy
In main thread
In main thread
In main thread
In main thread
In main thread
In main thread
In main thread
In main thread
In main thread
In main thread
In main thread
In main thread
In main thread
In main thread
In main thread
In main thread
In NEW thread
In NEW thread
In NEW thread
In NEW thread
In NEW thread
In NEW thread
In NEW thread
```

> 父子进程的执行和父子线程的执行相同，都为并发执行。