

《Linux操作系统》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，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gedit新建一个helloProcess.c源文件，并输入后面的范例代码。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。

```
(permit@permit)-[~/文档/yunxu]
$ cat fork.c
#include"stdio.h"
#include"sys/types.h"
#include"unistd.h"
int main()
{
    pid_t pid,cid;
    printf("this time task id:%d\n",getpid());
    cid=fork();
    printf("After fork process id:%d\n",getpid());
    int i;
    scanf("%d",&i);
    return 0;
}

(permit@permit)-[~/文档/yunxu]
$ ./a.out
this time task id:1825
After fork process id:1825
After fork process id:1826
```

原因：Fork()将当前进程完整拷贝了一份，子进程从fork()语句开始执行

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。

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

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
```

```
(permit@permit)-[~/文档/yunxu]
$ ./a.out
this time task id:2005
After fork process id:2005
After fork process id:2006
1

(permit@permit)-[~/文档/yunxu]
$ ps -all
F S  UID      PID     PPID  C  PRI  NI ADDR  SZ  WCHAN  TTY          TIME CMD
0 S   1000     1754     1663  0   80   0  -   3431  -      pts/0        00:00:01 zsh
0 R   1000     2011     1754  0   80   0  -   2425  -      pts/0        00:00:00 ps

(permit@permit)-[~/文档/yunxu]
$
```

原因：helloProcess进程因为scanf（）函数调用，helloProcess等待输入设备输入，进程进入等待状态

3. 通过判断fork的返回值让父子进程执行不同的语句。

```
\#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;
}
```

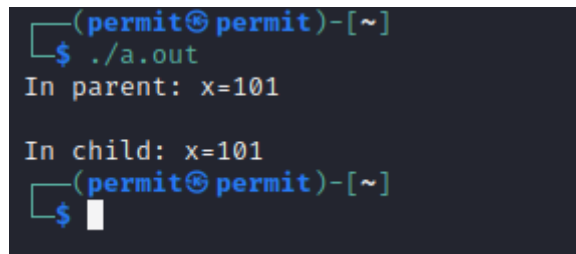
重新编译观察结果，重点观察父子进程是否判断正确（通过比较进程id）。父子进程其实是**并发**执行的，但实验结果好像是顺序执行的，多执行几遍看看有无变化，如果没有变化试着将两个循环的次数调整高一些，比如30、300，然后再观察运行结果并解释原因。

```
(permit@permit)-[~]
$ ./a.out
Before fork process id :1204
Parent process id :1204
world
world
world

Child process id (my parent pid is 1):1205
hello
hello
hello
(permit@permit)-[~]
```

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。

```
\#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;
}
```



```
(permit@permit)-[~]
$ ./a.out
In parent: x=101

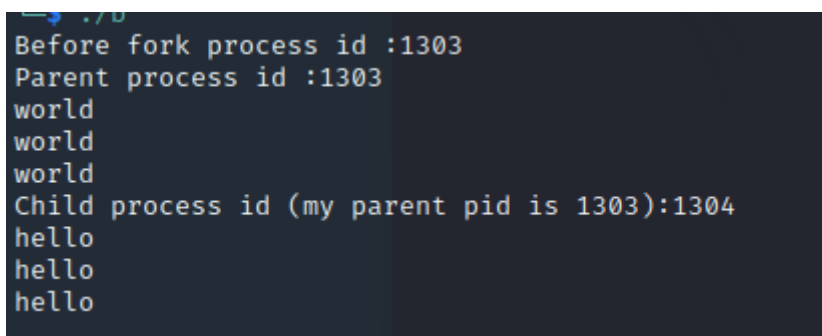
In child: x=101
(permit@permit)-[~]
$
```

原因：fork()将当前进程完整的拷贝给了子进程，子进程被分配了一个完整的新的内存空间。父进程和子进程不共享一块内存空间。

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

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

```
wait(NULL);
```



```
Before fork process id :1303
Parent process id :1303
world
world
world
Child process id (my parent pid is 1303):1304
hello
hello
hello
```

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);
```

```

$ ./c
Before fork process id :1341
Parent process id :1341
Child process id (my parent pid is 1341):1342
hello
hello
hello
world
world
world

```

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。

1. 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;
}

```

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
```

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？

```

(permit@permit)-[~]
$ gcc helloworld.c -o d -l pthread

(permit@permit)-[~]
$ ./d
In main thread

(permit@permit)-[~]
$ mousepad helloworld.c

(permit@permit)-[~]
$ gcc helloworld.c -o d -l pthread

(permit@permit)-[~]
$ ./d
In NEW thread
In main thread

```

原因：父子进程相互独立，父进程执行完毕，子进程终止。

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。

```
$ mousepad helloworld.c

(permit⊗permit)-[~]
$ gcc helloworld.c -o e -l pthread

(permit⊗permit)-[~]
$ ./e
In NEW thread2(84)
In NEW thread1(68)
In NEW thread1(9)
In NEW thread2(62)
In NEW thread1(9)
In NEW thread1(72)
In NEW thread2(21)
In NEW thread1(19)
In main thread
```

四、实验总结

进程并发执行