实验一 进程控制与描述

姓名：叶倩琳

学号：201706061330

完成日期：2019/10/19

# 一、实验目的

# 利用Windows提供的API函数，编写程序，实现进程的创建和终止（如创建写字板进程及终止该进程），加深对操作系统进程概念的理解，观察操作系统进程运行的动态性能，获得包含多进程的应用程序编程经验。

# 二、实验内容

1、进程的创建和终止。编写一段程序，可以创建一个进程，并终止当前创建的进程。试观察记录程序执行的结果，并分析原因。

2、利用VC++6.0实现上述程序设计和调试操作，对于进程创建的成功与否、终止进程操作的成功与否提供一定的提示框。

3、通过阅读和分析实验程序，学习创建进程、观察进程和终止进程的程序设计方法

4、Linux下进程创建：编写一段程序，使用系统调用fork()创建两个子进程。各进程显示不同的信息，如父进程显示字符“a”，子进程分别显示字符“b”和“c”。多次运行观察显示结果，并分析产生这种执行效果的原因。

5、修改上面编写的程序，将每个进程的输出由单个字符改为循环输出一句话，如父进程显示：“parent：”加上进程ID，子进程分别显示：“Child1：”（或“Child2：”）加上自己的进程ID。再多次运行观察程序执行时屏幕上出现的现象，并分析原因。

# 实验步骤

1. **创建线程**

Windows 所创建的每个进程都从调用CreateProcess() API函数开始，该函数的任务是在对象管理器子系统内初始化进程对象。

(1)函数原型

BOOL CreateProcess(

LPCTSTR lpApplicationName,

LPTSTR lpCommandLine,

LPSECURITY\_ATTRIBUTES lpProcessAttributes,

LPSECURITY\_ATTRIBUTES lpThreadAttributes,

BOOL bInheritHandles,

DWORD dwCreationFlags,

LPVOID lpEnvironment,

LPCTSTR lpCurrentDirectory,

LPSTARTUPINFO lpStartupInfo,

LPPROCESS\_INFORMATION lpProcessInformation

);

(2)函数中各参数的意义

lpApplicationName：指定要执行的模块，包括可执行代码的EXE文件的文件名。

lpCommandLine：指向一个以空结尾的串，该串定义了要执行的命令行。该命令行是可以在Windows提示符下执行的命令行。

lpProcessAttributes：指向一个SECURITY\_ATTRIBUTES结构，该结构决定了返回的句柄是否可被子进程继承。

lpThreadAttributes：指向一个SECURITY\_ATTRIBUTES结构，该结构决定了返回的句柄是否可被子进程继承。

bInheritHandles：表明新进程是否可继承创建者进程的句柄。

dwCreationFlags：定义控制优先类和进程创建的附加标志。

lpEnvironment：指向一个新进程的环境块。

lpCurrentDirectory：指向一个以空结尾的串，该串定义了子进程的当前驱动器和当前目录。

lpStartupInfo：指向一个STARTUPINFO结构，该结构定义了新进程的主窗口将如何显示。

lpProcessInformation：指向进程信息描述结构，该结构接受关于新进程的描述信息。

可执行文件运行时的文件名及其命令行，如 lpCommandLine="c:\\windows\\system32\\cmd.exe”

或lpApplicationName=“c:\windows\system32\cmd.exe”

lpStartupInfor描述新进程的窗口显示情况。定义STARTUPINFO结构，并赋予初始值， 如：STARTUPINFO si;

memset(&si,0,sizeof(si));

si.cb=sizeof(si);

lpProcessInformation指向进程信息描述结构，接受关于新进程的描述信息，如：PROCESS\_INFORMATION pi

1. 一个线程调用该函数首先创建一个进程内核对象来管理新进程，然后系统为新进程创建虚拟地址空间，并将可执行文件的代码和数据加载到这个地址空间，最后系统给新进程的主线程创建一个线程内核对象。使用CreateProcess创建线程的源码如下：

void open()

{

cout<<"Process create successfully!"<<endl;

STARTUPINFO si = { sizeof(si) };

si.dwFlags = STARTF\_USESHOWWINDOW;

si.wShowWindow = TRUE;

BOOL bRet = **CreateProcess**(NULL, proPath, NULL,NULL,FALSE, CREATE\_NEW\_CONSOLE,NULL,NULL,&si,&pi);

// 显示进程位置

cout << "Process ID: " << GetCurrentProcessId() << endl;

return;

}

**2、终止线程**

每一进程都以调用ExitProcess() 或TerminateProcess() API函数终止。TerminateProcess() API函数只要打开带有PROCESS\_TERMINATE访问权的进程对象，就可以终止进程，并向系统返回指定的代码。如果不知道所创建进程中所有线程的状态，最好使用TerminateProcess() 终止进程。

1. 函数原型

BOOL TerminateProcess{

HANDLE hProcess

UINT uExitCode

};

1. 函数中各参数的意义

hProcess:指定要中断进程的句柄。该句柄可由OpenProcess得到。

uExitCode：进程和其所有线程的退出代码。

1. 这个函数可以用来终止一个进程，它不会留给进程及其所有线程清理的时间，系统会马上终止(杀死)这个进程的所有线程。但使用函数前必须要调用OpenProcess函数来获得要终止进程的句柄以及进程的PROCESS\_TERMINATE权限。但本实验由于采用了堆栈式，即每次都将用来接收新进程的识别信息的PROCESS\_INFORMATION结构体（如下所示）存入到一个特定的堆栈中去，可以直接利用里面的参数获取需要终止的进程的句柄，从而避免了使用OpenProcess函数的麻烦。
2. typedef struct \_PROCESS\_INFORMATION {  
    HANDLE hProcess; //Handle to the newly created process.   
    HANDLE hThread; //Handle to the primary thread of the newly created process.   
    DWORD dwProcessId; //Value that can be used to identify a process.   
    DWORD dwThreadId; //Value that can be used to identify a thread. } PROCESS\_INFORMATION,   
    \*LPPROCESS\_INFORMATION;
3. 一个进程终止时，系统会依次执行以下操作：  
   1）终止进程中遗留的任何线程。  
   2）释放进程分配的所有用户对象，关闭所有内核对象。如果它们的使用计数变为0，内核对象将会释放。  
   3）将进程的退出代码从STILL\_ACTIVE变为传给ExitProcess或是TerminateProcess的参数存储在内核对象中。  
   4）进程内核对象变为一触发状态。这也是为什么其他线程可以挂起他们自己直至另一个进程终止运行。  
   5）进程内核对象的使用计数递减1。
4. 终止当前进程的源码如下：

void close()

{

BOOL bEndOK = **TerminateProcess**(pi.hProcess,0) ;

if(bEndOK)

cout<<"Process exit successfully!"<<endl;

//（存放进程信息和调用成员输出进程信息）用来 Process32First指向第一个进程信息，并将进程信息抽取到PROCESSENTRY32中

PROCESSENTRY32 pe;

HANDLE hProcess;

pe.dwSize = sizeof (PROCESSENTRY32);

HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS,0);

Process32First(hSnapshot,& pe);

do

{

/\*

if ( ! \_tcsicmp(pe.szExeFile,\_T("wordpad.exe")))

{

break;

}

\*/

pe.dwSize = sizeof(PROCESSENTRY32);

} while(Process32Next(hSnapshot, & pe));

STARTUPINFO si = { sizeof(si) };

PROCESS\_INFORMATION pi;

si.dwFlags = STARTF\_USESHOWWINDOW;

si.wShowWindow = TRUE;

//根据进程ID返回对象句柄

hProcess = OpenProcess(PROCESS\_TERMINATE,FALSE,pe.th32ProcessID);

//根据对象句柄结束进程

TerminateProcess(hProcess,0);

CloseHandle(hSnapshot);

CloseHandle(hProcess);

return;

}

**3、进程执行顺序**

源码：

#include <iostream>

#include <windows.h>

using namespace std;

DWORD WINAPI Fun(LPVOID lpParamter){

cout << " sub process b " ;

return 0L;

}

DWORD WINAPI Fun\_2(LPVOID lpParamter){

cout << " sub process c " ;

return 0L;

}

int main(){

cout<<"Test the execution sequence of main process a, sub process b and sub process c:"<<endl;

HANDLE hThread = CreateThread(NULL, 0, Fun, NULL, 0, NULL);

HANDLE hThread\_2 = CreateThread(NULL, 0, Fun\_2, NULL, 0, NULL);

CloseHandle(hThread);

CloseHandle(hThread\_2);

cout<<" main process a " ;

Sleep(500);

return 0;

}

1. **Linux系统下进程的创建**

（1）fork

源码：

// Linux下的调用fork()创建进程

#include "stdafx.h"

#include<unisted.h>

#include<stdio.h>

int main()

{

pid\_t pid;

pid = fork(); //当fork调用成功后，子进程开始从fork后开始执行

//创建进程失败

if(-1 == pid){

perror("fork failed");

return -1;

//父进程

}else if(pid > 0){

//pid用于接收当前进程id， ppid用于接收父进程id

printf("parent : pid = %d , ppid = %d\n", getpid(), getppid());

sleep(1);

//子进程

}else if(pid == 0){

printf("child: pid = %d, ppid = %d\n", getpid(), getppid());

sleep(1);

}

return 0;

}

# 实验结果及结论

**1、创建进程、终止进程**



图4-1-1 创建一个记事本进程

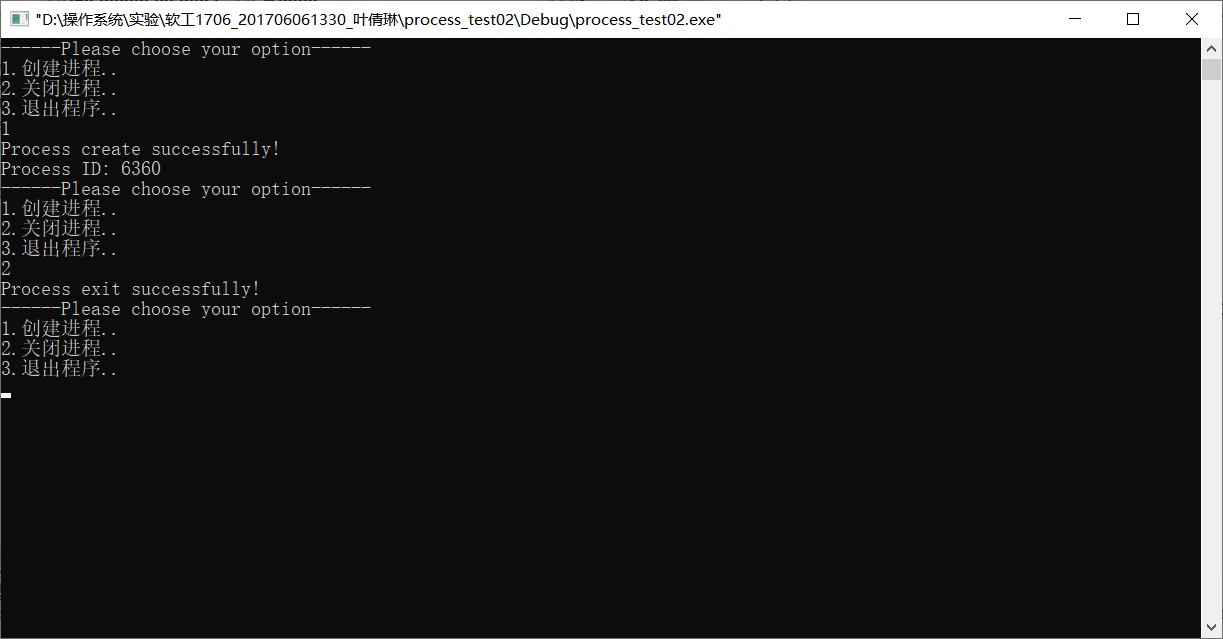


图4-1-2 终止刚刚创建的记事本进程

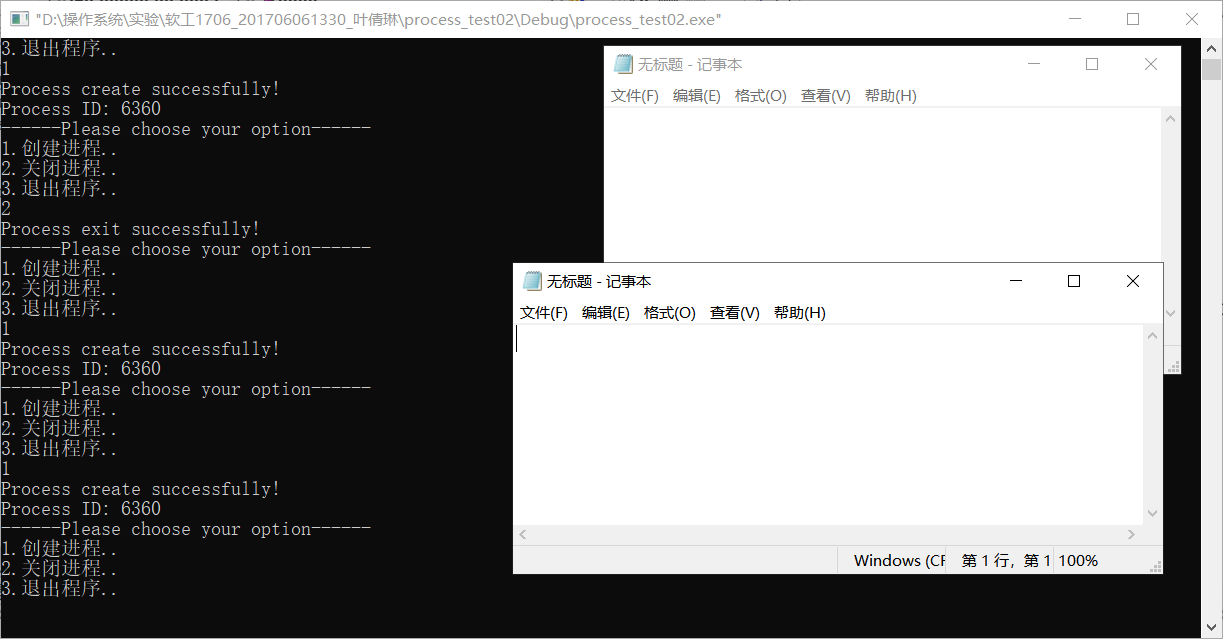


图4-1-3 创建两个个记事本进程



图4-1-4 终止一个记事本进程

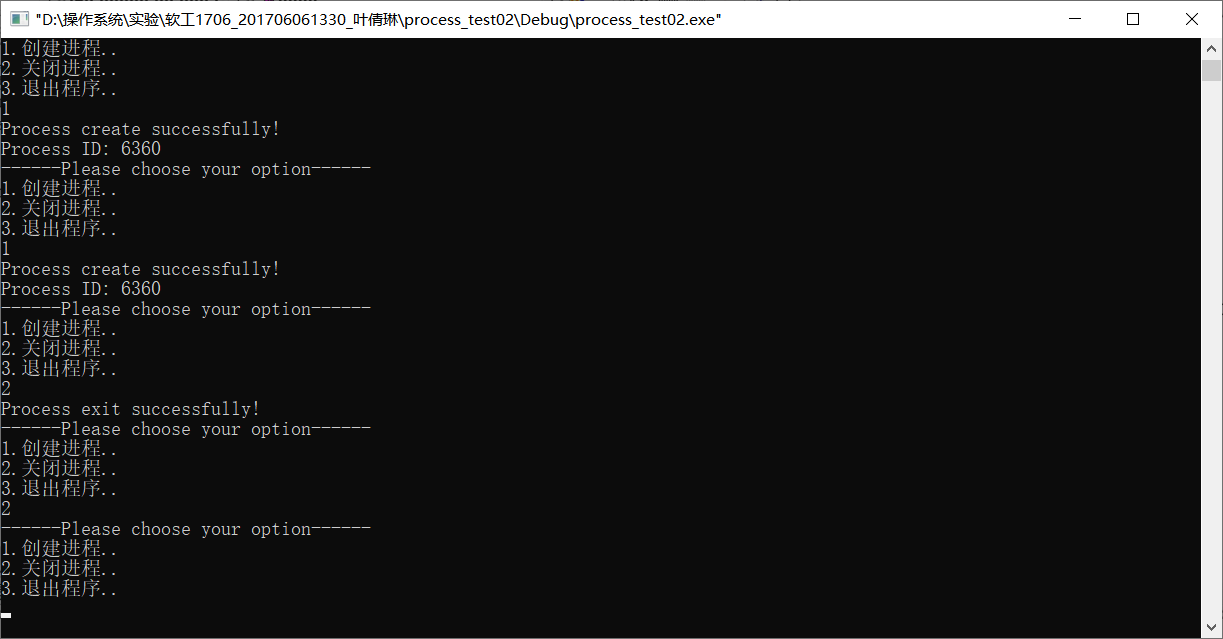


图4-1-5 再终止一个记事本进程

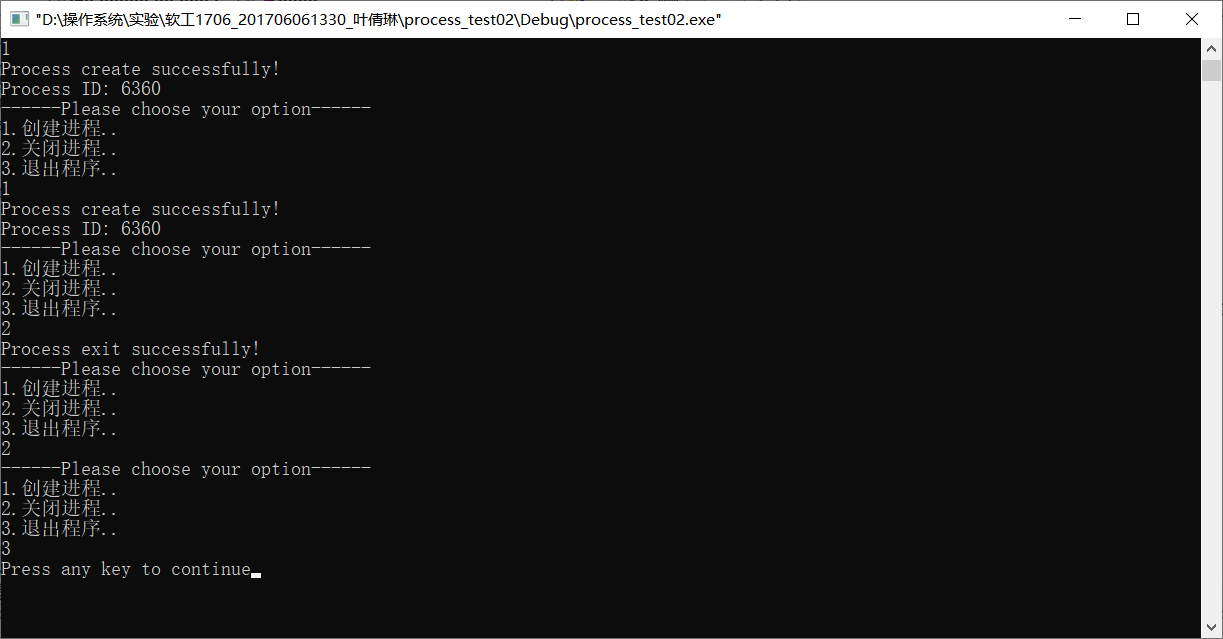


图4-1-6 退出程序

**2、进程执行顺序**

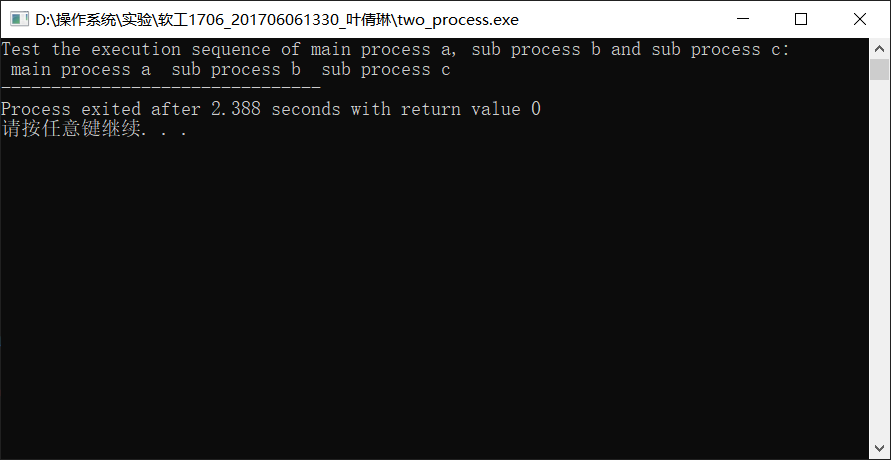


图4-2-1 进程执行顺序结果1

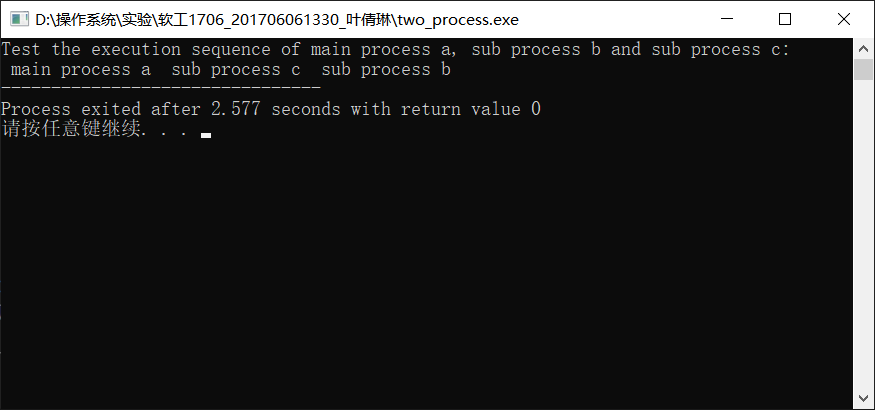


图4-2-2 进程执行顺序结果2

由此可知：1）主进程一定是先执行2）一旦启动子进程，后续的代码就并发，即没有先后顺序。

1. **Linux系统下fork（）创建进程**

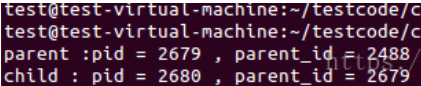


图4-3-1 进程id

结论：父子进程各自打印了自己的pid，同时也打印了父进程的pid。而且，Linux下调用fork创建子进程成功后，父进程和子进程谁先执行是不确定的，这取决于内核的调度算法。

# 实验中遇到的问题及解决方法

1. 之前我采用的都是CreateThread和CloseHandle的方式创建和关闭线程，本次关于进程的实验使我第一次接触到创建进程的Createprocess函数和终止进程的TerminateProcess函数，我通过PPT对这两个函数内部的每个参数的解释，对创建和销毁线程的逻辑有了一个大致的把握，实现了一个能够创建和终止多个记事本线程的C++程序。  
   2、在实验的过程中碰到了不同编译器对有效地址字符串（即Create Process中第二个参数）的处理问题，若采用（char\*）强制转换后出现指针溢出等问题。经过反复思考与去网上所搜答案，我修改原代码为“TCHAR proPath[] = TEXT("NOTEPAD");”后VC++无报错现象。  
   3、本次实验中，我并没有使用OpenProcess函数去获取相应进程的句柄并加以控制，而是用了比较取巧的方法保留了每个进程的状态信息，设置了PROCESS\_INFORMATION pi，直接将其作为TerminateProcess()函数的第二个参数：BOOL bEndOK = TerminateProcess(pi.hProcess,0) ;虽然简化了代码的编写，但实际并不规范。
2. 在进行写字板进程创建和终止时，我发现只能进行创建一个进程再终止一个进程的顺序操作，若一次创建两个写字板进程会导致终止进程时只能终止最近一次创建的进程。经分析，我分析这是因为我只使用了一个全局变量PROCESS\_INFORMATION pi，只能保存一个新建进程状态信息。为了解决这个问题，我采用了BOOL TerminateProcess( HANDLE hProcess, UINT uExitCode )结构体，（存放进程信息和调用成员输出进程信息）用来 Process32First指向第一个进程信息，并将进程[信息抽取](https://baike.baidu.com/item/%E4%BF%A1%E6%81%AF%E6%8A%BD%E5%8F%96" \t "https://baike.baidu.com/item/PROCESSENTRY32/_blank)到PROCESSENTRY32中。要获得可执行文件的完整路径，应调用Module32First函数，再检查其返回的MODULEENTRY32结构的szExePath成员。但是，如果被调用进程是一个64位程序，必须调用QueryFullProcessImageName函数去获取64位进程的可执行文件完整路径名。
3. 一开始我对Linux下父进程调用一次fork会返回2个返回值不是特别理解，经过李波老师在课堂上的详细解释后，我明白了对于父进程来说，父进程调用fork函数成功产生一个子进程，子进程会从父进程“复制”一份数据空间，堆和栈作为副本，这两个进程拥有相同的代码文本，但各自拥有不同的栈段，数据段等，因此，这2个返回值是由父进程和子进程各自返回的。

# 实验思考与体会

1. **系统是怎样创建进程的？**

答：1）给进程分配一个唯一的进程标识号

2）给进程分配空间 （进程要执行的程序代码和数据）

3）初始化进程控制块 （如PC初始化为程序入口地址；进程状态初始化为就绪或就绪/挂起；进程优先级初始化为最低优先级等）

4）将进程控制块加入到正确的队列中 （如加入到就绪队列中）

5）创建或扩充操作系统所需的其他数据结构 （如审计文件）

1. **可执行文件加载时进行了哪些处理？**

答：操作系统先判断该文件是否是一个合法的可执行文件。如果是则操作系统将按照段表中的指示为可执行程序分配地址空间。  
 接着进行了将源代码转换为机器可认识代码的过程。编译程序读取源程序（字符流），对之进行词法和语法的分析，将高级语言指令转换为功能等效的汇编代码，再由汇编程序转换为机器语言，并且按照操作系统对可执行文件格式的要求链接生成可执行程序。具体经过以下几个处理：C源程序－>编译预处理－>编译－>优化程序－>汇编程序－>链接程序－>可执行文件。

1. **当首次调用新创建进程时，其入口在那里？**

答：首次调用新创建进程的入口在指向进程的句柄。

**体会：**

通过本次实验，我基本掌握了进程的概念，并会简单使用CreateProcess()函数和TerminateProcess() 函数来创建、终止进程。我也进一步明确了进程和程序的区别，认识和了解了并发执行的实质。也通过查阅资料对Linux系统下进程创建的原理及方法有了一定了解。以下是我的具体实验体会。  
**（1）进程的概念：**进程是操作系统中可以并发运行和分配系统资源的基本单位。进程是运行中的程序，是程序在某个数据集合上的一次执行过程，具有并发性和动态性。进程是由进程控制块(PCB)、程序段和数据空间等三部分组成的。其中，PCB是进程存在的唯一标志，PCB描述了进程的基本情况，其内容可分为调度信息和执行信息两大部分。

进程的基本状态有就绪、运行和阻塞三种。阻塞态是指一个进程由于等待事件时所处的状态，当事件发生，进程便具备了运行的条件，状态转变为就绪态；就绪态是指一个进程具备了运行的条件，只是没有占有CPU的状态，一旦占有处理，其状态就转变为运行态；运行过程中进程会因时间片已到等非资源请求原因退出运行转变为就绪态，也会因等待事件而变为阻塞态。  
 进程的状态转换都是在操作系统控制下完成的。操作系统提供了对进程的基本操作(系统调用)，也称为原语。这些原语包括创建原语、阻塞原语、终止原语、优先级原语和调度原语。  
**（2）进程和程序的区别：**进程是程序的一次运行活动，是动态的，而程序是是进程运行所对应的运行代码，是静态的，是指令的集合。一个进程对应于一个程序，而一个程序可以同时对应于多个进程。

1. **并发执行的实质：**并发的核心就是如何在代码中协调好并行处理和串行处理。通过课本附录A以及网上资料了学习，我对并发主题中互斥的软件解决方法有了一定的理解，并针对Dekker算法和Peterson算法制作了简易的PPT，以加深印象。

**（4）Linux系统下进程创建的原理及方法：**

父进程调用fork( )函数时会产生一个子进程，且复制了同一份代码给子进程，导致调用一次fork会返回2个返回值，一次是父进程，另一次是子进程，然后分别独自运行，互不干扰，而调用的先后顺序是由操作系统的调度算法决定的。如图6-3-1所示，父进程调用fork函数成功产生一个子进程，子进程会从父进程复制一份数据空间、堆和栈作为副本，这两个进程拥有相同的代码文本，但各自拥有不同的栈段，数据段等，所以一个进程对全局变量的修改不会影响到另一个进程里的全局变量。

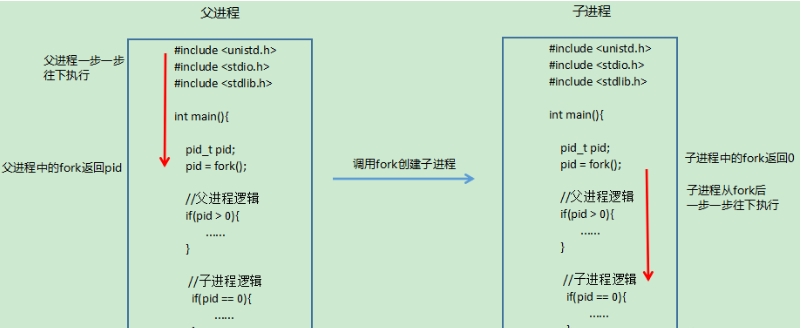


图6-3-1 fork的返回值

父进程中的fork返回值是子进程pid （如果调用fork失败则返回 -1），fork成功调用后，会把子进程中的fork的返回值直接修改为0，即子进程永远返回0，因此我们可以通过fork函数的返回值来区分并且控制父子进程的代码流程。