操作系统 -- 实验一实验报告

实验一：Windows进程管理

一、实验题目

> （1）编写基本的 Win32 Consol Application 步骤 1：登录进入 Windows 系统，启动 VC++ 6.0。
>
> 步骤 2：在“FILE”菜单中单击“NEW”子菜单，在“projects”选项卡中选择“Win32 Consol Application”,然后在“Project name”处输入工程名，在“Location” 处输入工程目录。创建一个新的控制台应用程序工程。
>
> 步骤 3：在“FILE”菜单中单击“NEW”子菜单，在“Files”选项卡中选择“C++ Source File”, 然后在“File” 处输入 C/C++源程序的文件名。
>
> 步骤 4：将清单 1-1 所示的程序清单复制到新创建的 C/C++源程序中。编译成可执行文件。步骤 5：在“开始”菜单中单击“程序”-“附件”-“命令提示符”命令，进入 Windows“命令提示符”窗口，然后进入工程目录中的 debug 子目录，执行编译好的可执行程序，列出运行结果(如果运行不成功，则可能的原因是什么？)

> （2） 创建进程 本实验显示了创建子进程的基本框架。该程序只是再一次地启动自身，显示它的系统进程 ID 和它在进程列表中的位置。
>
> 步骤 1：创建一个“Win32 Consol Application”工程，然后拷贝清单 1-2 中的程序，编译成可执行文件。
>
> 步骤 2：在“命令提示符”窗口运行步骤 1 中生成的可执行文件，列出运行结果。按下ctrl+alt+del，调用 windows 的任务管理器，记录进程相关的行为属性。
>
> 步骤 3：在“命令提示符”窗口加入参数重新运行生成的可执行文件，列出运行结果。按下ctrl+alt+del，调用 windows 的任务管理器，记录进程相关的行为属性。
>
> 步骤 4：修改清单 1-2 中的程序，将 nClone 的定义和初始化方法按程序注释中的修改方法进行修改，编译成可执行文件（执行前请先保存已经完成的工作）。再按步骤 2 中的方式运行，看看结果会有什么不一样。列出行结果。从中你可以得出什么结论？说明 nClone 的作用。 变量的定义和初始化方法（位置）对程序的执行结果有影响吗？为什么？

> （3） 父子进程的简单通信及终止进程 步骤 1：创建一个“Win32 Consol Application”工程，然后拷贝清单 1-3 中的程序，编译成可执行文件。
>
> 步骤 2：在 VC 的工具栏单击“Execute Program”(执行程序) 按钮，或者按 Ctrl + F5 键，或者在“命令提示符”窗口运行步骤 1 中生成的可执行文件，列出运行结果。
>
> 步骤 3：按源程序中注释中的提示，修改源程序 1-3，编译执行（执行前请先保存已经完成的工作），列出运行结果。在程序中加入跟踪语句，或调试运行程序，同时参考 MSDN 中的帮助文件CreateProcess()的使用方法，理解父子进程如何传递参数。给出程序执行过程的大概描述。
>
> 步骤 4：按源程序中注释中的提示，修改源程序 1-3，编译执行，列出运行结果。

> 步骤 5： 参考 MSDN 中的帮助文件 CreateMutex() 、 OpenMutex() 、 ReleaseMutex() 和 WaitForSingleObject()的使用方法，理解父子进程如何利用互斥体进行同步的。给出父子进程同步过程的一个大概描述。

二、实验目的

> （1）学会使用 VC 编写基本的 Win32 Consol Application（控制台应用程序)。
>
> （2）通过创建进程、观察正在运行的进程和终止进程的程序设计和调试操作，进一步熟悉操作系统的进程概念，理解 Windows 进程的“一生”。
>
> （3）通过阅读和分析实验程序，学习创建进程、观察进程、终止进程以及父子进程同步的基本程序设计方法。

三、总体设计

- 主函数 `main()` 中判断当前处于父进程还是子进程，如果处于子进程则调用 `Parent()` 函数创建子进程并传入参数，实现**进程间通信**。
- `Parent()` 函数中主要包括创建子进程并调用 `ReleaseMutex()` 函数释放互斥体的所有权，以及调用 `CloseHandle()` 消除句柄。
- `Child()` 子进程中则主要包括打开自杀互斥体并进入**阻塞状态**，等待父进程通过互斥体发来的信号。收到信号后准备终止并清除句柄。

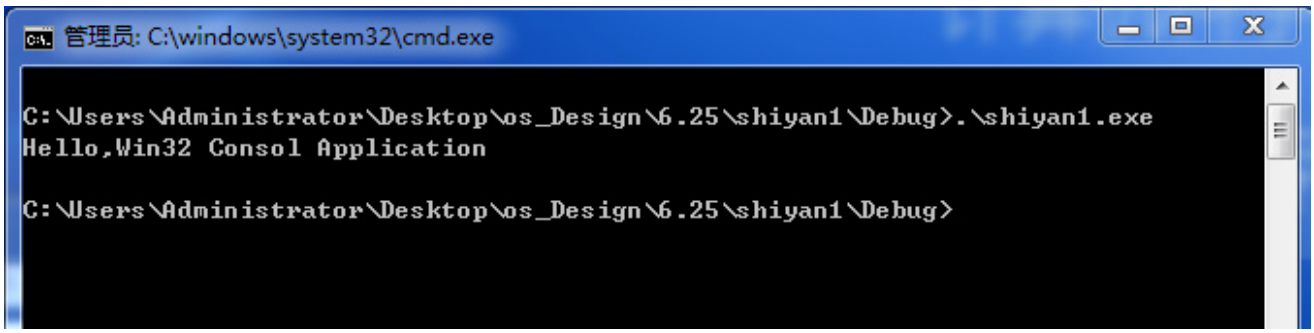
四、详细设计

1. 编写基本的Win32 Console Application

实验代码：

```
#include "stdafx.h"
#include<iostream>
void main()
{
    std::cout<<"Hello,Win32 Consol Application"<<std::endl;
}
```

运行截图



2. 创建进程

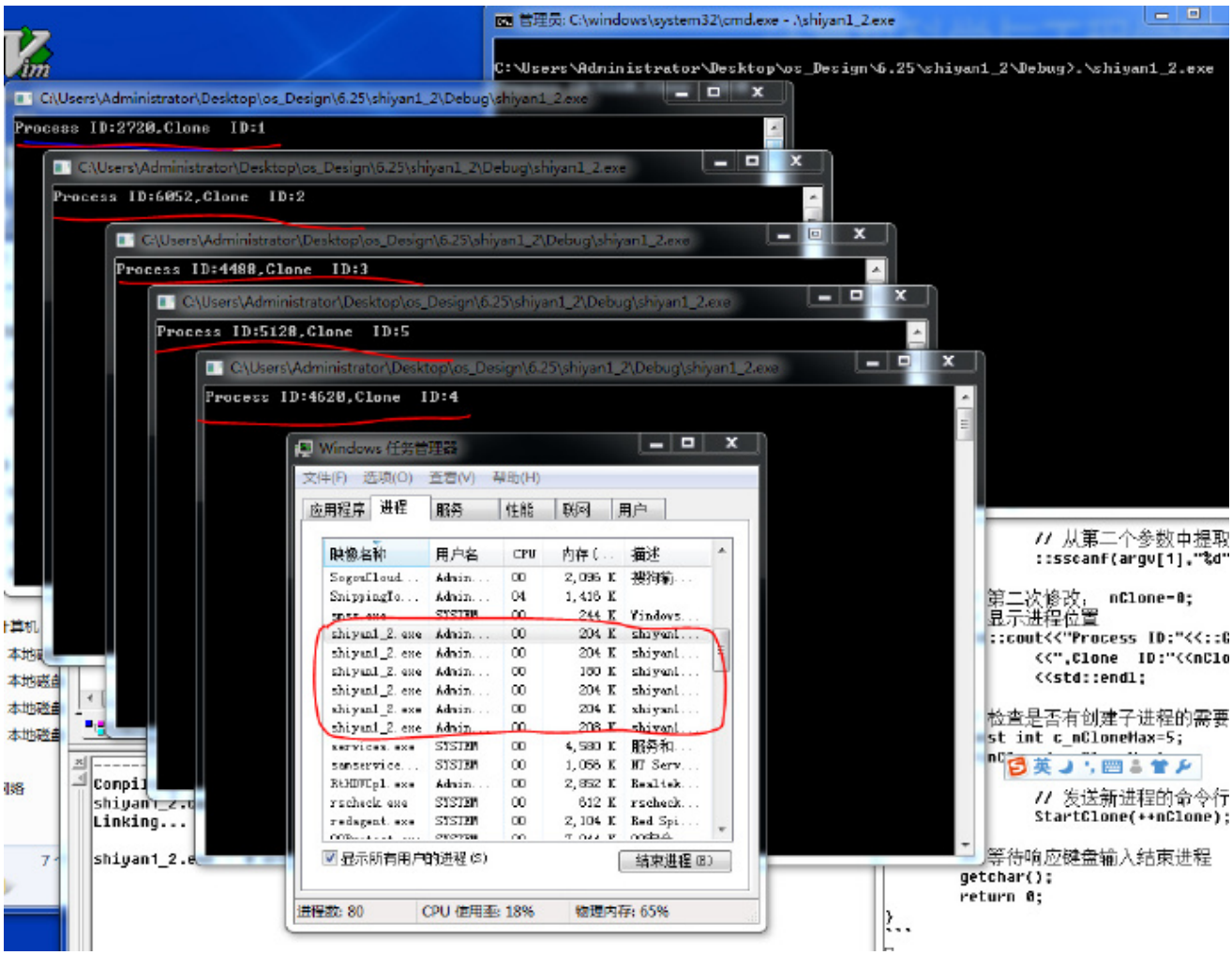
实验代码：

```
// shiyan1_2.cpp : Defines the entry point for the console application.
```

```
//

#include "stdafx.h"
#include<windows.h>
#include<iostream>
#include<stdio.h>

// 创建传递过来的进程的克隆过程并赋予其ID值
void StartClone(int nCloneID)
{
    // 提取用于当前可执行文件的文件名
    // szFilename:[D:\testProject\Debug\testProject.exe]
    // MAX_PATH:[MAX_PATH是C语言运行时库中通过#define指令定义的一个宏常量，它定义了编译器所支持的最长全路径名的长度]
    // Windows的MAX_PATH:[MAX_PATH的解释：  文件名最长256（ANSI），加上盘符（X:\）3字节，259字节，再加上结束符1字节，共260]
    TCHAR szFilename[MAX_PATH];
    GetModuleFileName(NULL,szFilename,MAX_PATH);

    // 格式化用于子进程的命令行并通知其EXE文件名和克隆ID
    // szCmdLine:["D:\testProject\Debug\testProject.exe"5]
    TCHAR szCmdLine[MAX_PATH];
    sprintf(szCmdLine,"\"%s\"%d",szFilename,nCloneID);


    // 用于子进程的STARTUPINFO结构
    STARTUPINFO si;
    ZeroMemory(&si,sizeof(si));
    si.cb=sizeof(si); // 必须是本结构的大小

    // 返回的用于子进程的进程信息
    PROCESS_INFORMATION pi;

    // 利用同样的可执行文件和命令行创建进程，并赋予其子进程的性质
    BOOL bCreateOK=::CreateProcess(
        szFilename,  // 产生这个EXE文件的应用程序的名称
        szCmdLine,   // 告诉其行为像一个子进程的标志
        NULL,        // 缺省的进程安全性
        NULL,        // 缺省的线程安全性
        FALSE,       // 不继承句柄
        CREATE_NEW_CONSOLE, // 使用新的控制台
        NULL,        // 新的环境
        NULL,        // 当前目录
        &si,         // 启动信息
        &pi);        // 返回的进程信息

    // 对子进程释放引用
    if(bCreateOK)
    {
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
    }

}
```

```
int main(int argc, char* argv[])
{
    // 确定派生出几个进程，及派生进程在进程列表中的位置
    int nClone=0;
    // 修改语句: int nClone;
    // 第一次修改:  nClone=0;
    if(argc>1)
    {
        // 从第二个参数中提取克隆 ID
        // 之所以按照注释修改代码就会死循环，原因在于父进程在创建子进程时会把nClone当作命令行参数传入子进程，nClone=0的位置就显得尤为重要!!（参考line 20注释，命令行后附带参数[cClone]）
        ::sscanf(argv[1],"%d",&nClone);
        printf("子进程读取到命令行参数:%d\n",nClone);
    }
    // 第二次修改:  nClone=0;
    //nClone=0;
    // 显示进程位置
    std::cout<<"Process ID:"<<::GetCurrentProcessId()
        <<",Clone  ID:"<<nClone
        <<std::endl;

    // 检查是否有创建子进程的需要
    const int c_nCloneMax=5;
    if(nClone<c_nCloneMax)
    {
        // 发送新进程的命令行和克隆号
        StartClone(++nClone);
    }
    // 等待响应键盘输入结束进程
    getchar();
    return 0;
}
```

进程相关的行为属性截图（任务管理器）



3. 父进程的简单通信及终止进程

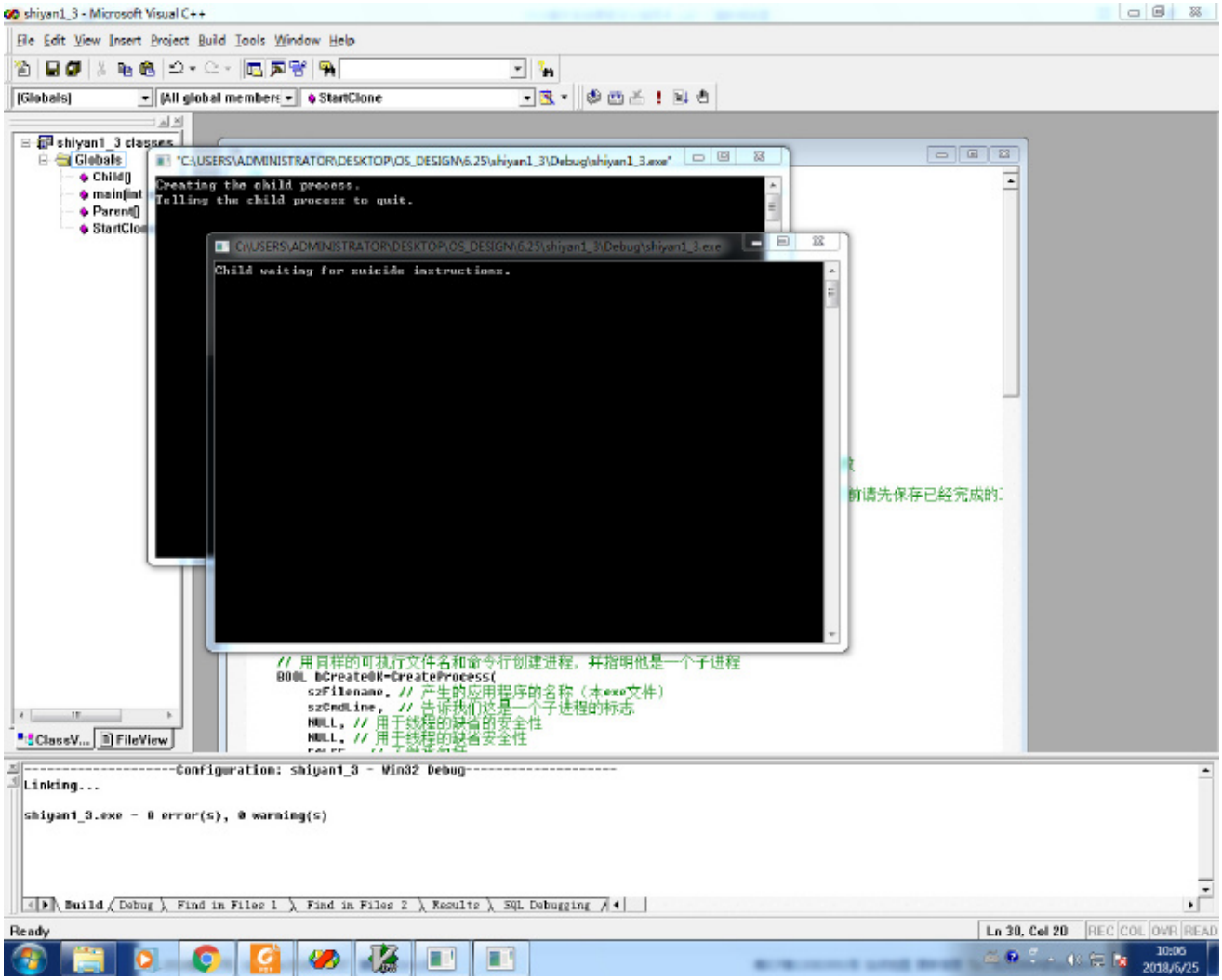
实验代码：

```
/*
 * shiyan1_3.cpp : Defines the entry point for the console application.
 *
 * procterm 项目
 */

#include "stdafx.h"
#include<windows.h>
#include<iostream>
#include<stdio.h>
static LPCTSTR g_szMutexName="w2kdg.ProcTerm.mutex.Suicide";

// 创建当前进程的克隆进程的简单方法
void StartClone()
{
    // 提取当前可执行文件的文件名
    TCHAR szFilename[MAX_PATH];
    GetModuleFileName(NULL,szFilename,MAX_PATH);
```

```
    // 格式化用于子进程的命令行，字符串”child“将作为形参传递给子进程的main函数
    TCHAR szCmdLine[MAX_PATH];
    //// 实验1-3 步骤3：将下句中的字符创child改为别的字符创，重新编译执行，执行前请先保存已经完成的工
作
    sprintf(szCmdLine,"\"%s\"child",szFilename);

    // 子进程的启动信息结构
    STARTUPINFO si;
    ZeroMemory(&si,sizeof(si));
    si.cb=sizeof(si);  // 应当是此结构的大小

    // 返回的用于子进程的进程信息
    PROCESS_INFORMATION pi;

    // 用同样的可执行文件名和命令行创建进程，并指明他是一个子进程
    BOOL bCreateOK=CreateProcess(
        szFilename, // 产生的应用程序的名称（本exe文件）
        szCmdLine,  // 告诉我们这是一个子进程的标志
        NULL, // 用于线程的缺省的安全性
        NULL, // 用于线程的缺省安全性
        FALSE,  // 不继承句柄
        CREATE_NEW_CONSOLE, // 创建新窗口
        NULL, // 新环境
        NULL, // 当前目录
        &si, // 启动信息结构
        &pi); // 启动的进程信息

    // 释放指向子进程的引用
    if(bCreateOK)
    {
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
    }
}

void Parent()
{
    // 创建”自杀“互斥程序体
    HANDLE hMutexSuicide=CreateMutex(
        NULL, // 缺省的安全性
        TRUE, // 最初拥有的
        g_szMutexName); // 互斥体名称
    if(hMutexSuicide!=NULL)
    {
        // 创建子进程
        std::cout<<"Creating the child process."<<std::endl;
        StartClone();
        // 指令子进程”杀“掉自身
        std::cout<<"Telling the child process to quit."<<std::endl;
        // 等待父进程的键盘响应
        getchar();
        // 释放互斥体的所有权，这个信号会送给子进程的WaitForSingleObject过程

        ReleaseMutex(hMutexSuicide);
```

```
        // 消除句柄
        CloseHandle(hMutexSuicide);
    }
}

void Child()
{
    // 打开”自杀“互斥体
    HANDLE hMutexSuicide=OpenMutex(
        SYNCHRONIZE, // 打开用于同步
        FALSE, // 不需要向下传递
        g_szMutexName); // 名称
    if(hMutexSuicide!=NULL)
    {
        // 报告我们正在等待指令
        std::cout<<"Child waiting for suicide instructions."<<std::endl;

        // 子进程进入阻塞状态，等待父进程通过互斥体发来的信号
        WaitForSingleObject(hMutexSuicide,INFINITE);
        // 实验1-3步骤4：将上句改为 WaitForSingleObject(hMutexSuicide,0),重新百衲衣执行

        // 准备好终止，清除句柄
        std::cout<<"Child quiting."<<std::endl;
        CloseHandle(hMutexSuicide);
    }
}

int main(int argc,char* argv[])
{
    // 决定其行为是父进程还是子进程
    if(argc>1 && ::strcmp(argv[1],"child")==0)
    {
        Child();
    }
    else
    {
        Parent();
    }
    return 0;
}
```

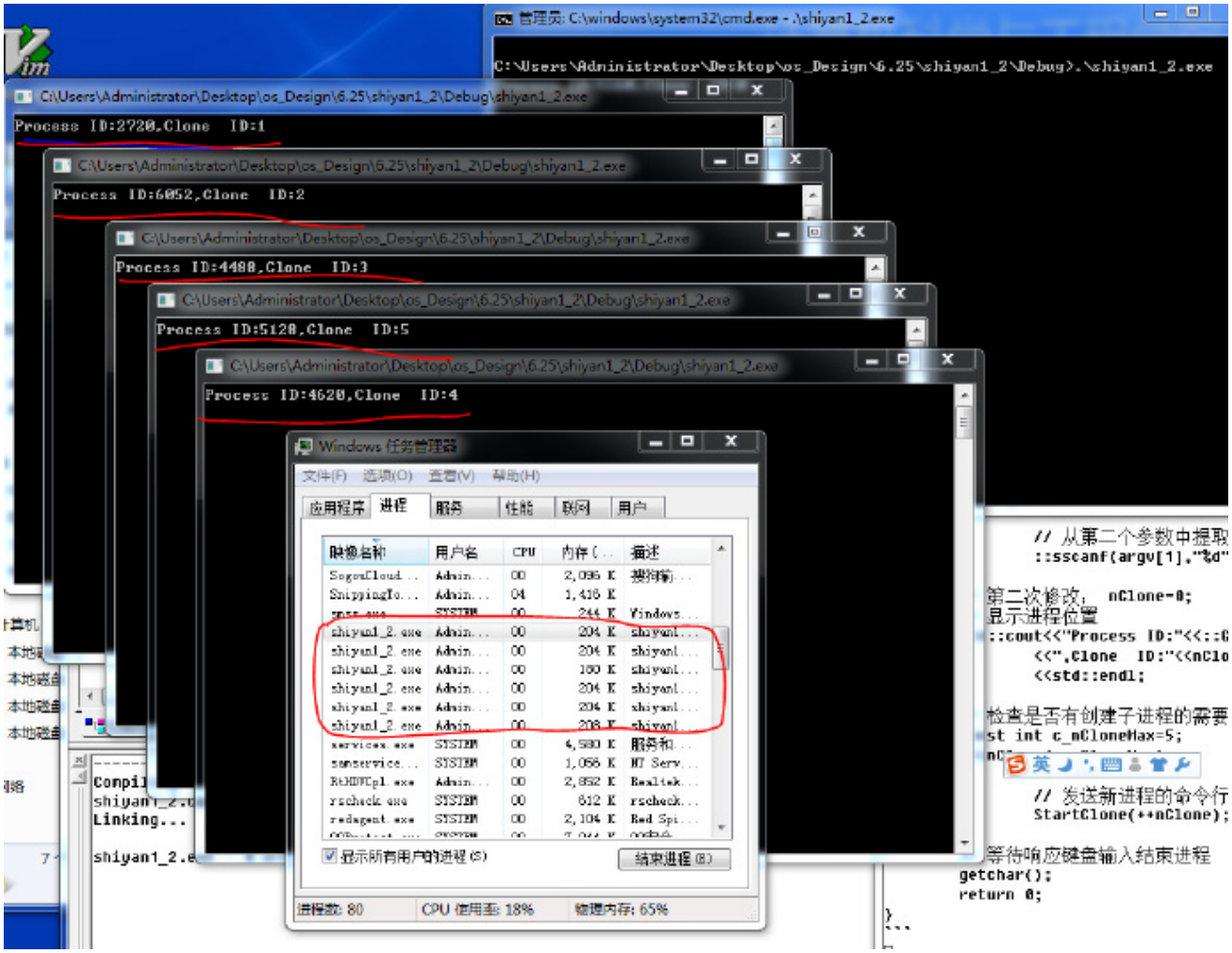
实验截图（程序运行截图）：



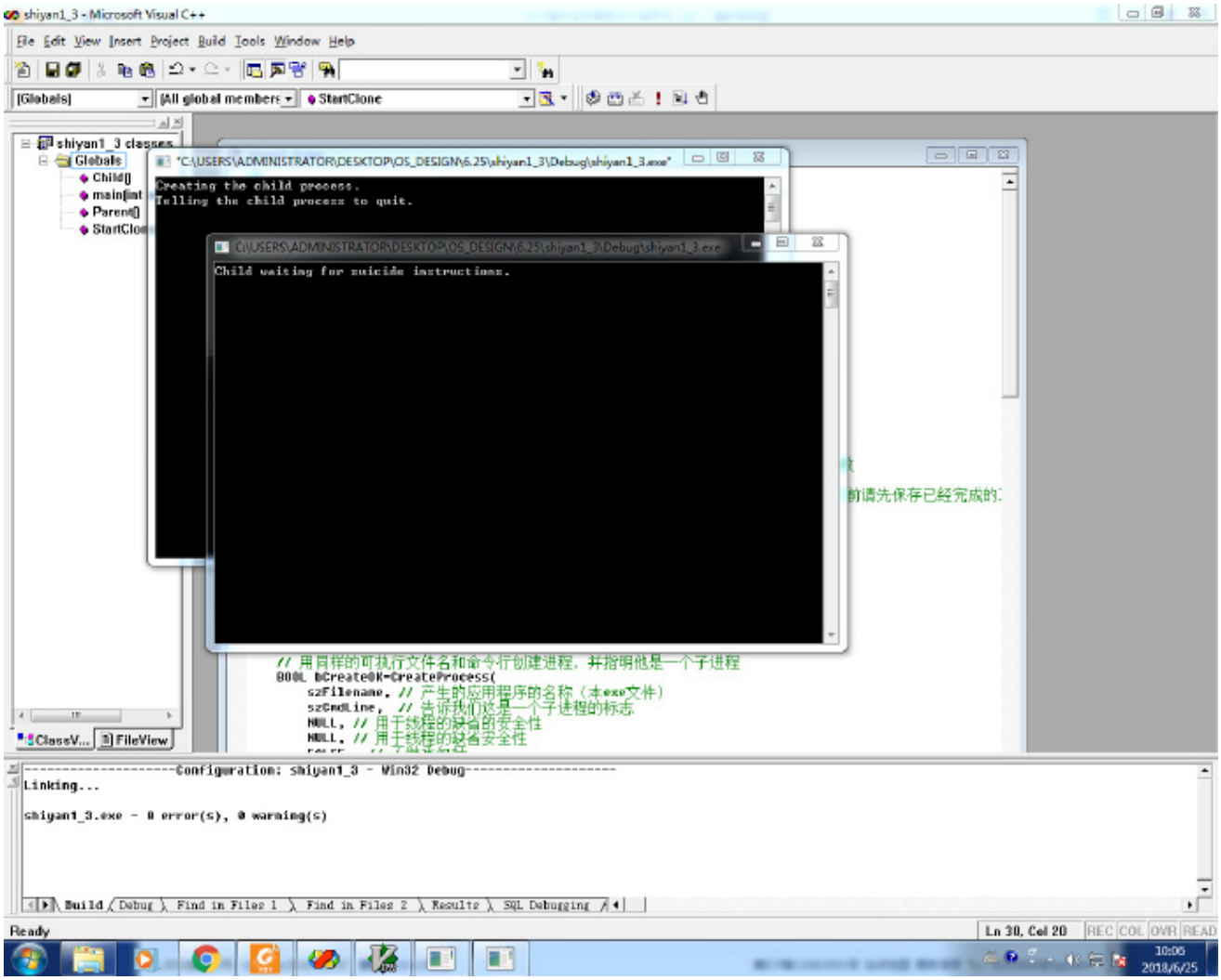
五、实验结果与分析

实验结果：

(1)创建进程运行结果：



(2)父子进程间的简单通信及进程终止运行结果：



实验分析:

- nClone 的作用为确定创建子进程的数量；
- 之所以按照注释修改代码就会死循环，原因在于父进程在创建子进程时会把nClone当作命令行参数传入子进程，nClone=0的位置就显得尤为重要!
- 将代码`WaitForSingleObject(hMutexSuicide,INFINITE);`改为`WaitForSingleObject(hMutexSuicide,INFINITE);`后，子进程不再等待父进程通过互斥体发来的信号（等待时间为0），因此直接往下执行代码。清除句柄，放弃进程。

六、小结与心得体会

- MAX_PATH:[MAX_PATH是C语言运行时库中通过#define指令定义的一个宏常量，它定义了编译器所支持的最长全路径名的长度]
- Windows的MAX_PATH:[MAX_PATH的解释： 文件名最长256（ANSI），加上盘符（X:\）3字节，259字节，再加上结束符1字节，共260]
- 为了获得互斥体，首先，想要访问调用的线程可使用 OpenMutex() API 来获得指向对象的句柄；