

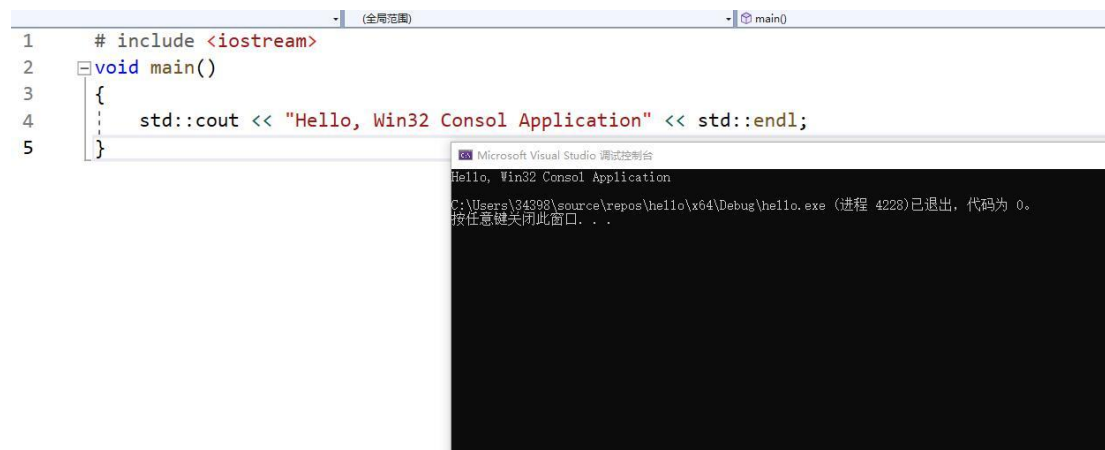
(1) 编写基本的 Win32 Consol Application

实验代码：

```
#include <iostream>

void main()
{
    std::cout << "Hello, Win32 Consol Application" << std::endl;
}
```

实验结果：



The screenshot shows the Microsoft Visual Studio IDE. The top pane displays the source code for a Win32 Console Application. The code is as follows:

```
1 #include <iostream>
2 void main()
3 {
4     std::cout << "Hello, Win32 Consol Application" << std::endl;
5 }
```

The bottom pane shows the Output window with the following text:

```
Hello, Win32 Consol Application
C:\Users\34398\source\repos\hello\x64\Debug\hello.exe (进程 4228) 已退出，代码为 0。
按任意键关闭此窗口。...
```

(2) 创建进程

实验代码：

```
#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
    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
    TCHAR szCmdLine[MAX_PATH];
    sprintf_s(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 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;
    //int nClone;
    //nClone=0;
    if (argc > 1)
    {
        // 从第二个参数中提取克隆 ID
        ::scanf_s(argv[1], "%d", &nClone);
    }
    //第二次修改
    //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

```

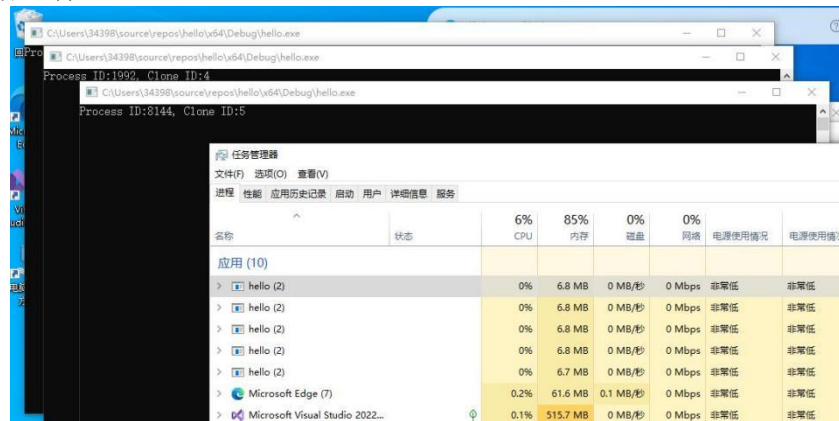
```

getchar();
return 0;
}

```

实验结果：

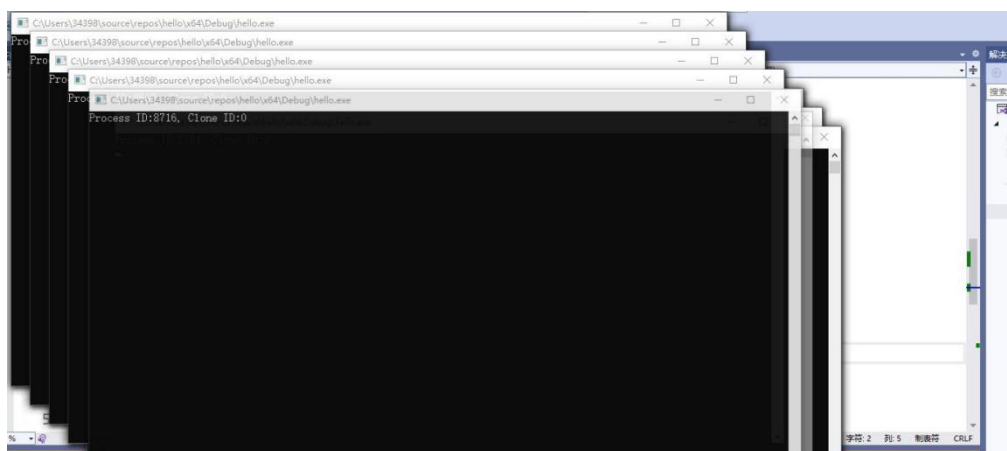
(1) 初始运行



(2) 第一次修改



(3) 第二次修改



(3) 父子进程的简单通信及终止进程

实验代码：

```
// procterm 项目
#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 2-3 步骤 3：将下句中的字符串 child 改为别的字符串，重新编译执行，执行前
    // 请先保存已经完成的工作
    sprintf_s(szCmdLine, "\\\"%s\\\" hello", 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，并指明它是一个子进程
    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, // 用于进程的缺省的安全性
        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); // 返回的进程信息
    // 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，这个信号会发送给子进程的 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 2-3 步骤 4：将上句改为 WaitForSingleObject(hMutexSuicide, 0)，重新编译执行
        // 准备好终止，清除句柄
        std::cout << "Child quitting." << std::endl;
        CloseHandle(hMutexSuicide);
    }
}

int main(int argc, char* argv[])

```

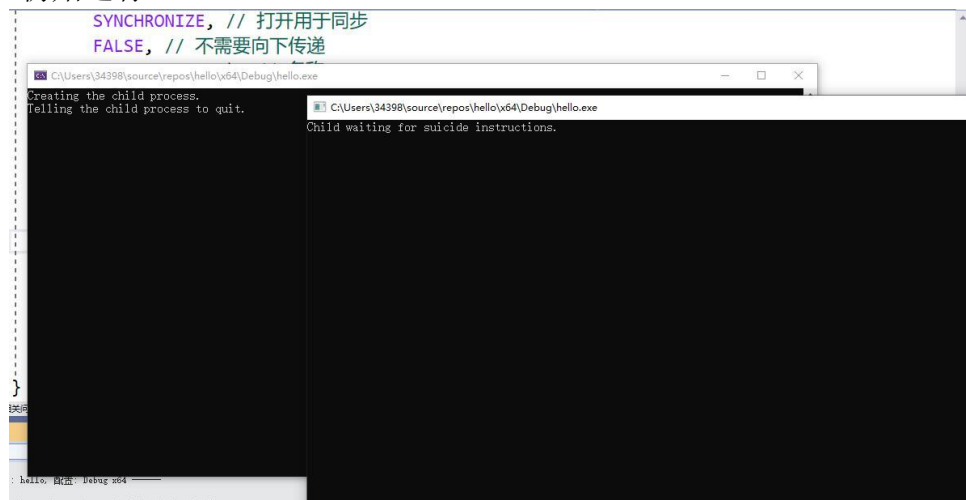
```

{
    // 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;
}

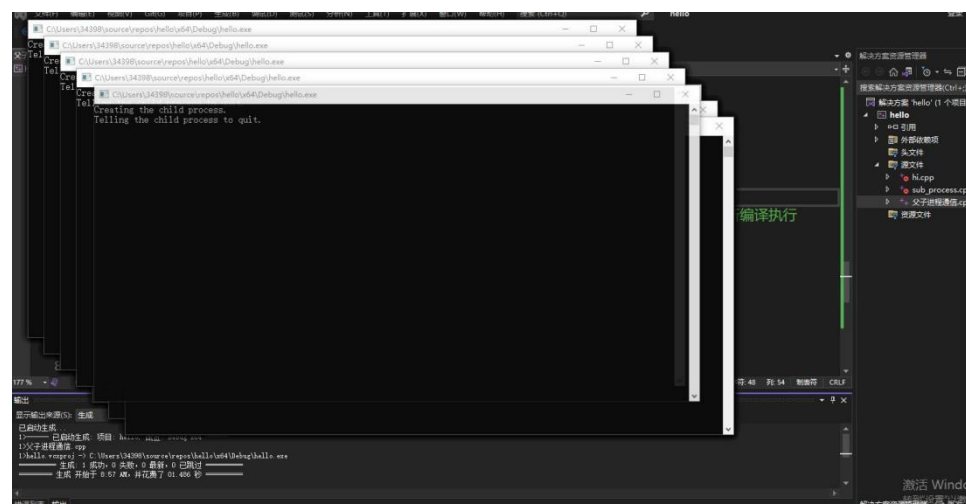
```

实验结果：

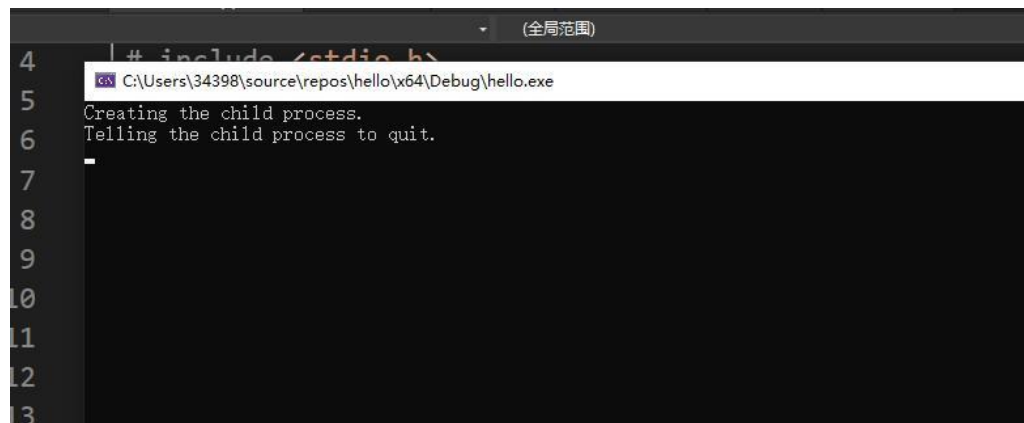
(1) 初始运行



(2) 将字符串 child 改为别的字符串



(3) 将 WaitForSingleObject(hMutexSuicide, INFINITE);
改为 WaitForSingleObject(hMutexSuicide, 0)



```
4 | #include <stdio.h>
5 |
6 | Creating the child process.
7 | Telling the child process to quit.
8 |
9 |
10 |
11 |
12 |
13 |
```