Windows操作系统

C/C++ 程序实验

首都师范大学 信息工程学院

系统分析与管理实验室

## 实验二 Windows 进程控制

### 一、背景知识

  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。每一进程都以调用ExitProcess() 或TerminateProcess() API函数终止。通常应用程序的框架负责调用 ExitProcess() 函数。对于C++ 运行库来说，这一调用发生在应用程序的main() 函数返回之后。

#### 1. 创建进程

CreateProcess() 调用的核心参数是可执行文件运行时的文件名及其命令行。表 2-1详细地列出了每个参数的类型和名称。

**表2-1 CreateProcess() 函数的参数**

|  |  |
| --- | --- |
| 参数名称 | 使用目的 |
| LPCTSTR lpApplivationName | 全部或部分地指明包括可执行代码的EXE文件的文件名 |
| LPCTSTR lpCommandLine | 向可执行文件发送的参数 |
| LPSECURIITY\_ATTRIBUTES lpProcessAttributes | 返回进程句柄的安全属性。主要指明这一句柄是否应该由其他子进程所继承 |
| LPSECURIITY\_ATTRIBUTES lpThreadAttributes | 返回进程的主线程的句柄的安全属性 |
| BOOL bInheritHandle | 一种标志，告诉系统允许新进程继承创建者进程的句柄 |
| DWORD dwCreationFlage | 特殊的创建标志 (如CREATE\_SUSPENDED) 的位标记 |
| LPVOID lpEnvironment | 向新进程发送的一套环境变量；如为null值则发送调用者环境 |
| LPCTSTR lpCurrentDirectory | 新进程的启动目录 |
| STARTUPINFO lpStartupInfo | STARTUPINFO结构，包括新进程的输入和输出配置的详情 |
| LPPROCESS\_INFORMATION lpProcessInformation | 调用的结果块；发送新应用程序的进程和主线程的句柄和ID |

可以指定第一个参数，即应用程序的名称，其中包括相对于当前进程的当前目录的全路径或者利用搜索方法找到的路径；lpCommandLine参数允许调用者向新应用程序发送数据；接下来的三个参数与进程和它的主线程以及返回的指向该对象的句柄的安全性有关。

然后是标志参数，用以在dwCreationFlags参数中指明系统应该给予新进程什么行为。经常使用的标志是CREATE\_SUSPNDED，告诉主线程立刻暂停。当准备好时，应该使用ResumeThread() API来启动进程。另一个常用的标志是CREATE\_NEW\_CONSOLE，告诉新进程启动自己的控制台窗口，而不是利用父窗口。这一参数还允许设置进程的优先级，用以向系统指明，相对于系统中所有其他的活动进程来说，给此进程多少CPU时间。

接着是CreateProcess() 函数调用所需要的三个通常使用缺省值的参数。第一个参数是lpEnvironment参数，指明为新进程提供的环境；第二个参数是lpCurrentDirectory，可用于向主创进程发送与缺省目录不同的新进程使用的特殊的当前目录；第三个参数是STARTUPINFO数据结构所必需的，用于在必要时指明新应用程序的主窗口的外观。

CreateProcess() 的最后一个参数是用于新进程对象及其主线程的句柄和ID的返回值缓冲区。以PROCESS\_INFORMATION结构中返回的句柄调用CloseHandle() API函数是重要的，因为如果不将这些句柄关闭的话，有可能危及主创进程终止之前的任何未释放的资源。

#### 2. 正在运行的进程

如果一个进程拥有至少一个执行线程，则为正在系统中运行的进程。通常，这种进程使用主线程来指示它的存在。当主线程结束时，调用ExitProcess() API函数，通知系统终止它所拥有的所有正在运行、准备运行或正在挂起的其他线程。当进程正在运行时，可以查看它的许多特性，其中少数特性也允许加以修改。

首先可查看的进程特性是系统进程标识符 (PID) ，可利用GetCurrentProcessId() API函数来查看，与GetCurrentProcess() 相似，对该函数的调用不能失败，但返回的PID在整个系统中都可使用。其他的可显示当前进程信息的API函数还有GetStartupInfo()和GetProcessShutdownParameters() ，可给出进程存活期内的配置详情。

通常，一个进程需要它的运行期环境的信息。例如API函数GetModuleFileName() 和GetCommandLine() ，可以给出用在CreateProcess() 中的参数以启动应用程序。在创建应用程序时可使用的另一个API函数是IsDebuggerPresent() 。

可利用API函数GetGuiResources() 来查看进程的GUI资源。此函数既可返回指定进程中的打开的GUI对象的数目，也可返回指定进程中打开的USER对象的数目。进程的其他性能信息可通过GetProcessIoCounters()、GetProcessPriorityBoost() 、GetProcessTimes() 和GetProcessWorkingSetSize() API得到。以上这几个API函数都只需要具有PROCESS\_QUERY\_INFORMATION访问权限的指向所感兴趣进程的句柄。

另一个可用于进程信息查询的API函数是GetProcessVersion() 。此函数只需感兴趣进程的PID (进程标识号) 。本实验程序清单3-6中列出了这一API函数与GetVersionEx() 的共同作用，可确定运行进程的系统的版本号。

#### 3. 终止进程

所有进程都是以调用ExitProcess() 或者TerminateProcess() 函数结束的。但最好使用前者而不要使用后者，因为进程是在完成了它的所有的关闭“职责”之后以正常的终止方式来调用前者的。而外部进程通常调用后者即突然终止进程的进行，由于关闭时的途径不太正常，有可能引起错误的行为。

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。这是一种“野蛮”的终止进程的方式，但是有时却是需要的。

如果开发人员确实有机会来设计“谋杀”(终止别的进程的进程) 和“受害”进程 (被终止的进程) 时，应该创建一个进程间通讯的内核对象——如一个互斥程序——这样一来，“受害”进程只在等待或周期性地测试它是否应该终止。

### 二、实验目的

1) 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”。

2) 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。

### 三、工具/准备工作

在开始本实验之前，请回顾教科书的相关内容。

需要做以下准备：

1) 一台运行Windows 操作系统的计算机。

2) 计算机中需安装Microsoft Visual Studio Code。

### 四、实验内容与步骤

请回答：

Windows所创建的每个进程都是以调用\_\_\_CreateProcess()\_\_\_ API函数开始和以调用\_\_\_\_\_ExitProcess()\_\_\_\_ 或 \_\_\_TerminateProcess()\_\_API函数终止。

#### 1. 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**：登录进入Windows 。

**步骤2**：在“开始”菜单中单击“程序”-“Microsoft Visual Studio Code”，进入VSCode窗口。

**步骤3：**新建项目名为“2-1”，并且新建项“2-1.cpp”。

**清单2-1 创建子进程**

// proccreate项目

# include <windows.h>

# include <iostream>

# include <stdio.h>

// 创建传递过来的进程的克隆过程并赋于其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(szCmdLine, “\”%s\” %d”, szFilename, nCloneID) ;

  // 用于子进程的STARTUPINFO结构

STARTUPINFO si;

:: ZeroMemory(reinterpret\_cast <void\*> (&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，并赋于其子进程的性质

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[] )

{

// 确定进程在列表中的位置

int nClone(0) ;

if (argc > 1)

{

// 从第二个参数中提取克隆ID

:: sscanf(argv[1] , “%d” , &nClone) ;

}

  // 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 = 25;

if (nClone < c\_nCloneMax)

{

// 发送新进程的命令行和克隆号

StartClone(++nClone) ;

}

// 在终止之前暂停一下 (l/2秒)

:: Sleep(500) ;

system(“pause“);

  return 0;

}

**步骤4：**在代码宏定义里添加 #define \_CRT\_SECURE\_NO\_WARNINGS

**步骤5：**按“F5”开始调试，注意路径里不要含有中文。

操作能否正常进行？如果不行，则可能的原因是什么？

需要将中文字符进行修改才能成功编译。

**步骤6：**按暂停按钮可暂停程序的执行，按终止按钮可终止程序的执行。



清单2-1展示的是一个简单的使用CreateProcess() API函数的例子。首先形成简单的命令行，提供当前的EXE文件的指定文件名和代表生成克隆进程的号码。大多数参数都可取缺省值，但是创建标志参数使用了：CREATE\_NEW\_CONSOLE

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

标志，指示新进程分配它自己的控制台，这使得运行示例程序时，在任务栏上产生许多活动标记。然后该克隆进程的创建方法关闭传递过来的句柄并返回main() 函数。在关闭程序之前，每一进程的执行主线程暂停一下，以便让用户看到其中的至少一个窗口。

CreateProcess() 函数有\_10\_个核心参数？本实验程序中设置的各个参数的值是：

a. LPCTSTR lpApplivationName szFilename；

b. LPCTSTR lpCommandLine szCmdLine；

c. LPSECURIITY\_ATTRIBUTES lpProcessAttributes NULL；

d. LPSECURIITY\_ATTRIBUTES lpThreadAttributes NULL；

e. BOOL bInheritHandle FALSE；

f. DWORD dwCreationFlage CREATE\_NEW\_CONSOLE；

g. LPVOID lpEnvironment NULL；

h. LPCTSTR lpCurrentDirectory NULL;

i. STARTUPINFO lpStartupInfo &si；

j. LPPROCESS\_INFORMATION lpProcessInformation &pi。

|  |
| --- |
| LPCTSTR lpApplivationName |
| LPCTSTR lpCommandLine |
| LPSECURIITY\_ATTRIBUTES lpProcessAttributes |
| LPSECURIITY\_ATTRIBUTES lpThreadAttributes |
| BOOL bInheritHandle |
| DWORD dwCreationFlage |
| LPVOID lpEnvironment |
| LPCTSTR lpCurrentDirectory |
| STARTUPINFO lpStartupInfo |
| LPPROCESS\_INFORMATION lpProcessInformation |

程序运行时屏幕显示的信息是：

程序运行时屏幕显示的信息类似于Process ID: 144, Clone ID: 25

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### 2. 正在运行的进程

本实验的程序中列出了用于进程信息查询的API函数GetProcessVersion() 与GetVersionEx() 的共同作用，可确定运行进程的操作系统的版本号。

**步骤7**：新建项目名为“2-2”，并且新建项“2-2.cpp”。

**清单2-2 使用进程和操作系统的版本信息**

// version项目

# include <windows.h>

# include <iostream>

// 利用进程和操作系统的版本信息的简单示例

void main()

{

// 提取这个进程的ID号

DWORD dwIdThis = :: GetCurrentProcessId() ;

// 获得这一进程和报告所需的版本，也可以发送0以便指明这一进程

DWORD dwVerReq = :: GetProcessVersion(dwIdThis) ;

WORD wMajorReq = (WORD) (dwVerReq >> 16) ;

WORD wMinorReq = (WORD) (dwVerReq & 0xffff );

std :: cout << "Process ID: " << dwIdThis

<< ", requires OS: " << wMajorReq << wMinorReq << std :: endl ;

// 设置版本信息的数据结构，以便保存操作系统的版本信息

OSVERSIONINFOEX osvix;

:: ZeroMemory(&osvix, sizeof(osvix) ) ;

osvix.dwOSVersionInfoSize = sizeof(osvix) ;

// 提取版本信息和报告

#pragma warning(disable:4996)

:: GetVersionEx(reinterpret\_cast < LPOSVERSIONINFO > (&osvix) ) ;

std :: cout << "Running on OS: " << osvix.dwMajorVersion << "."

<< osvix.dwMinorVersion << std :: endl;

// 根据系统版本，判断是否提高优先权

if ( osvix.dwPlatformId == VER\_PLATFORM\_WIN32\_NT && osvix.dwMajorVersion >= 5)

{

// 改变优先级

:: SetPriorityClass(

:: GetCurrentProcess() , // 利用这一进程

HIGH\_PRIORITY\_CLASS) ; // 改变为high

// 报告给用户

std :: cout << "Task Manager should now indicate this"

"process is high priority." << std :: endl;

}

system("pause");

}

**步骤8：**按“F5”开始调试，注意路径里不要含有中文。

操作能否正常进行？如果不行，则可能的原因是什么？

能够正常运行

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

运行结果：

当前PID信息：22444\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

当前操作系统版本：6.2\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

系统提示信息：Task Manager should now indicate thisprocess is high priority.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

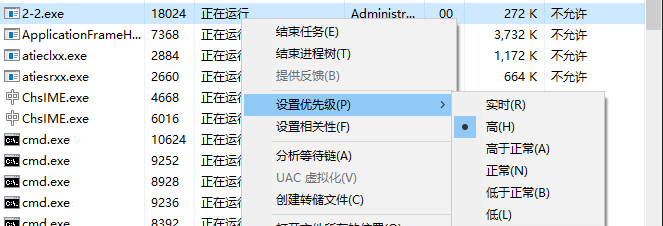
清单2-2中的程序向读者表明了如何获得当前的PID和所需的进程版本信息。为了运行这一程序，系统处理了所有的版本不兼容问题。

接着，程序演示了如何使用GetVersionEx() API函数来提取OSVERSIONINFOEX结构。这一数据块中包括了操作系统的版本信息。其中，“OS : \*.\*”表示当前运行的操作系统是：6.2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

清单2-2的最后一段程序利用了优先级显示以确认运行的操作系统版本信息。

**步骤9：**分析程序，当前进程优先级是否被修改，修改程序显示进程优先级是否有变化。

当前进程优先级被修改为HIGH\_PRIORITY  将程序修改为

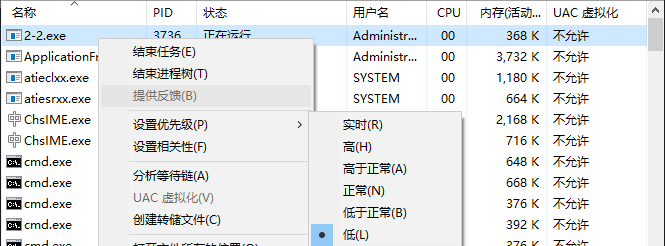
  // 改变优先级

              :: SetPriorityClass(

                  :: GetCurrentProcess() ,                  // 利用这一进程

                  IDLE\_PRIORITY\_CLASS) ;            // 改变为idle

呈现效果为低优先级，可以确认该方法有效更改程序优先级。

 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

除了改变进程的优先级以外，还可以对正在运行的进程执行几项其他的操作，只要获得其进程句柄即可。SetProcessAffinityMask() API函数允许开发人员将线程映射到处理器上；SetProcessPriorityBoost() API可关闭前台应用程序优先级的提升；而 SetProcessWorkingSet() API可调节进程可用的非页面RAM的容量；还有一个只对当前进程可用的API函数，即SetProcessShutdownParameters() ，可告诉系统如何终止该进程。

#### 3. 终止进程

在清单2-3列出的程序中，先创建一个子进程，然后指令它发出“自杀弹”互斥体去终止自身的运行。

**步骤10**：新建项目名为“2-3”，并且新建项“2-3.cpp”。

**清单2-3 指令其子进程来“杀掉”自己的父进程**

// 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，指明它是一个EXE文件和子进程

TCHAR szCmdLine[MAX\_PATH] ;

:: sprintf(szCmdLine, "\"%s\" child" , szFilename) ;

// 子进程的启动信息结构

STARTUPINFO si;

:: ZeroMemory(reinterpret\_cast < void\* > (&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，使输出更直观

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) ; // 为其命名

if (hMutexSuicide != NULL)

{

// 创建子进程

std :: cout << "Creating the child process." << std :: endl;

:: StartClone() ;

// 暂停

::Sleep(5000) ;

// 指令子进程"杀"掉自身

std :: cout << "Telling the child process to quit. " << std :: endl;

:: 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)

{

// 报告正在等待指令

std :: cout << "Child waiting for suicide instructions. " << std :: endl;

:: WaitForSingleObject(hMutexSuicide, INFINITE) ;

// 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() ;

}

system("pause");

return 0;

}

清单2-3中的程序说明了一个进程从“生”到“死”的整个一生。第一次执行时，它创建一个子进程，其行为如同“父亲”。在创建子进程之前，先创建一个互斥的内核对象，其行为对于子进程来说，如同一个“自杀弹”。当创建子进程时，就打开了互斥体并在其他线程中进行别的处理工作，同时等待着父进程使用ReleaseMutex() API发出“死亡”信号。然后用Sleep() API调用来模拟父进程处理其他工作，等完成时，指令子进程终止。

当调用ExitProcess() 时要小心，进程中的所有线程都被立刻通知停止。在设计应用程序时，必须让主线程在正常的C++ 运行期关闭 (这是由编译器提供的缺省行为) 之后来调用这一函数。当它转向受信状态时，通常可创建一个每个活动线程都可等待和停止的终止事件。

在正常的终止操作中，进程的每个工作线程都要终止，由主线程调用ExitProcess()。接着，管理层对进程增加的所有对象释放引用，并将用 GetExitCodeProcess() 建立的退出代码从STILL\_ACTIVE改变为在ExitProcess() 调用中返回的值。最后，主线程对象也如同进程对象一样转变为受信状态。

等到所有打开的句柄都关闭之后，管理层的对象管理器才销毁进程对象本身。还没有一种函数可取得终止后的进程对象为其参数，从而使其“复活”。当进程对象引用一个终止了的对象时，有好几个API函数仍然是有用的。进程可使用退出代码将终止方式通知给调用GetExitCodeProcess() 的其他进程。同时，GetProcessTimes() API函数可向主调者显示进程的终止时间。

**步骤11：** 在代码宏定义里添加 #define \_CRT\_SECURE\_NO\_WARNINGS

**步骤12：**按“F5”开始调试，注意路径里不要含有中文。

操作能否正常进行？如果不行，则可能的原因是什么？

能够正常运行，如果不行应该是文件格式问题或者是键入内容含有中文字符。

运行结果：

1) Creating the child process.

表示：父进程创建子进程。

2) Child waiting for suicide instructions.

表示：子进程报告正在等待父进程下达“自杀”指令。

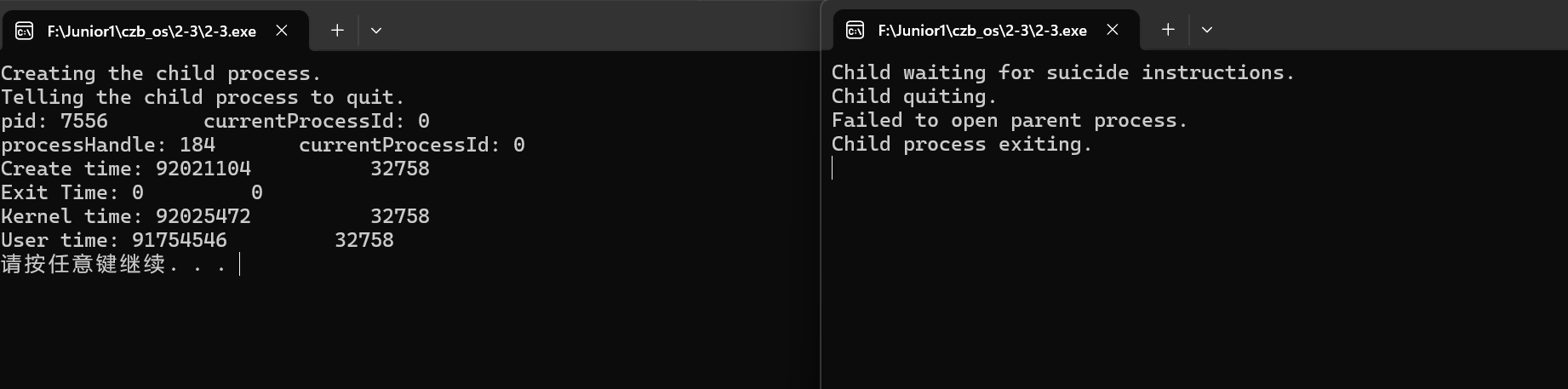
3) Telling the child process to quit.

表示：父进程指令子进程"杀"掉自身。

4) Child quiting.

表示：子进程准备好终止，清除句柄。

**步骤13（选作）：**在熟悉清单2-3源代码的基础上，利用本实验介绍的API函数(如ExitProcess()、GetExitCodeProcess()、GetProcessTimes() 等)来尝试改进本程序并运行。请描述你所做的工作：



修改后通过ExitProcess(0); 来确保子进程正确退出，

GetProcessTimes函数来查看父进程的运行时间，

尝试用GetExitCodeProcess和TerminateProcess来终止父进程。

 // procterm项目

  #define \_CRT\_SECURE\_NO\_WARNINGS

      # include <windows.h>

      # include <iostream>

      # include <stdio.h>

      static LPCTSTR g\_szMutexName = "w2kdg.ProcTerm.mutex.Suicide" ;

      HANDLE hParentProcess = NULL;

      DWORD P\_Pid;

       // 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，指明它是一个EXE文件和子进程

          TCHAR szCmdLine[MAX\_PATH] ;

          :: sprintf(reinterpret\_cast < char\* > (szCmdLine), "\"%s\" child" , szFilename) ;

           // 子进程的启动信息结构

          STARTUPINFO si;

          :: ZeroMemory(reinterpret\_cast < void\* > (&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，使输出更直观

              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)

          {

                hParentProcess = GetCurrentProcess();

              :: CloseHandle(pi.hProcess) ;

              :: CloseHandle(pi.hThread) ;

          }

      }

void PrintProcessTimes(HANDLE process) {

    FILETIME creationTime, exitTime, kernelTime, userTime;

    if (GetProcessTimes(process, &creationTime, &exitTime, &kernelTime, &userTime)) {

        ULONGLONG cTime = (((ULONGLONG)creationTime.dwHighDateTime) << 32) | creationTime.dwLowDateTime;

        ULONGLONG eTime = (((ULONGLONG)exitTime.dwHighDateTime) << 32) | exitTime.dwLowDateTime;

        double elapsedSeconds = (eTime - cTime) / 10000000.0;

        std::cout << "Process lifetime: " << elapsedSeconds << " seconds" << std::endl;

    } else {

        std::cerr << "Failed to get process times." << std::endl;

    }

}

      void Parent()

      {

          // 创建"自杀"互斥程序体

          P\_Pid = GetCurrentProcessId();

            // hParentProcess = GetCurrentProcess();

          HANDLE hMutexSuicide = :: CreateMutex(

              NULL,                 // 缺省的安全性

              TRUE,                 // 最初拥有的

              g\_szMutexName) ;          // 为其命名

          if (hMutexSuicide != NULL)

          {

              // 创建子进程

              std :: cout << "Creating the child process." << std :: endl;

              :: StartClone() ;

               // 暂停

              ::Sleep(2500) ;

               // 指令子进程"杀"掉自身

      std :: cout << "Telling the child process to quit. " << std :: endl;

                    // ::Sleep(5000) ;

              :: ReleaseMutex(hMutexSuicide) ;

    HANDLE     processHandle = hMutexSuicide;

    DWORD      currentProcessId = GetProcessId(processHandle);

    FILETIME createTime, exitTime, kernelTime, userTime;

    // 获取当前进程的PID

    DWORD pid = GetCurrentProcessId();

    printf("pid: %d\t currentProcessId: %d\n", pid, currentProcessId);

    GetProcessTimes(processHandle, &createTime, &exitTime, &kernelTime, &userTime);

    printf("processHandle: %lu\t currentProcessId: %d\n", HandleToULong(processHandle), currentProcessId);

    printf("Create time: %lu\t       %lu\nExit Time: %lu\t     %lu\nKernel time: %lu\t       %lu\nUser time: %lu\t    %lu\n",

        createTime.dwLowDateTime, createTime.dwHighDateTime,

        exitTime.dwLowDateTime, exitTime.dwHighDateTime,

        kernelTime.dwLowDateTime, kernelTime.dwHighDateTime,

        userTime.dwLowDateTime, userTime.dwHighDateTime);

               // 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)

          {

              // 报告正在等待指令

              std :: cout << "Child waiting for suicide instructions. " << std :: endl;

              :: WaitForSingleObject(hMutexSuicide, INFINITE) ;

              // 准备好终止，清除句柄

              std :: cout << "Child quiting. " << std :: endl;

                 // 获取父进程的句柄（假设父进程ID已知）

        DWORD parentPid = P\_Pid;

 if (hParentProcess != NULL) {

            DWORD exitCode;

            if (GetExitCodeProcess(hParentProcess, &exitCode)) {

                std::cout << "Parent process exit code: " << exitCode << std::endl;

                if (exitCode == STILL\_ACTIVE) {

                    std::cout << "Parent process is active. Terminating..." << std::endl;

                    TerminateProcess(hParentProcess, 0);  // 终止父进程

                    std::cout << "Parent process terminated." << std::endl;

                }

            } else {

                std::cerr << "Failed to get parent process exit code." << std::endl;

            }

            PrintProcessTimes(hParentProcess);  // 打印父进程时间信息

            ::CloseHandle(hParentProcess);

        } else {

            std::cerr << "Failed to open parent process." << std::endl;

        }

        ::CloseHandle(hMutexSuicide);

    std::cout << "Child process exiting." << std::endl;

    Sleep(2500);

    ExitProcess(0);  // 确保子进程正确退出

}

}

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() ;

          }

  system("pause");

          return 0;

}