Windows操作系统

C/C++ 程序实验

姓名：\_\_\_\_\_\_\_陈展博\_\_\_\_\_\_

学号：\_\_\_\_\_1221001003\_\_\_\_

班级：\_\_\_\_\_\_计科1班\_\_\_\_\_

院系：\_\_\_\_\_\_\_信工\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_2024\_\_\_\_\_\_\_年\_\_\_10\_\_月\_\_16\_\_\_日

## 实验二 Windows 进程控制

### 一、背景知识

### 二、实验目的

### 三、工具/准备工作

### 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

**步骤5：**编译完成后，单击“调试”菜单中的“开始执行”命令。

操作能否正常进行？如果不行，则可能的原因是什么？

不能正常进行，需要将中文字符进行修改才能成功编译。

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；指明包括可执行代码的EXE文件的文件名

b. LPCTSTR lpCommandLine szCmdLine；向可执行文件发送的命令行参数

c. LPSECURIITY\_ATTRIBUTES lpProcessAttributes NULL；返回进程句柄的安全属性

d. LPSECURIITY\_ATTRIBUTES lpThreadAttributes NULL；返回进程的主线程的句柄的安全属性

e. BOOL bInheritHandle FALSE；一种标志，告诉系统允许新进程继承创建者进程的句柄

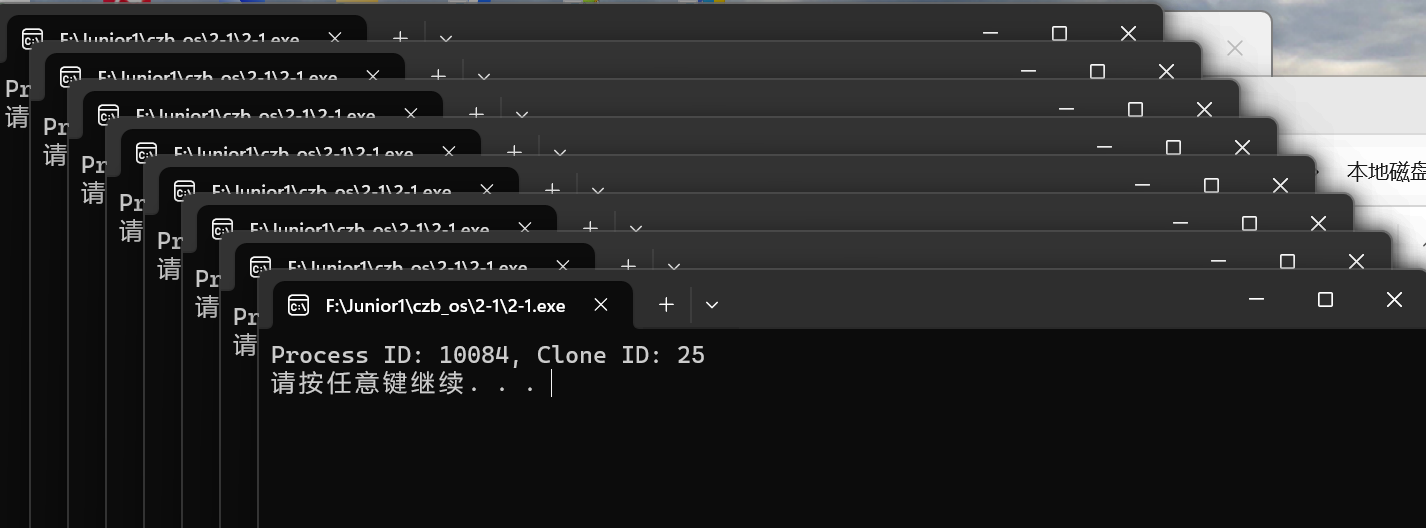
f. DWORD dwCreationFlage CREATE\_NEW\_CONSOLE；特殊的创建标志 (如CREATE\_SUSPENDED) 的位标记，这里用的是CREATE\_NEW\_CONSOLE

g. LPVOID lpEnvironment NULL；发送的一套环境变量，NULL值则发送调用者环境

h. LPCTSTR lpCurrentDirectory NULL; 新进程的启动目录

i. STARTUPINFO lpStartupInfo &si；STARTUPINFO结构，包括新进程的输入和输出配置的详情

j. LPPROCESS\_INFORMATION lpProcessInformation &pi。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

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: 10084, Clone ID: 25。

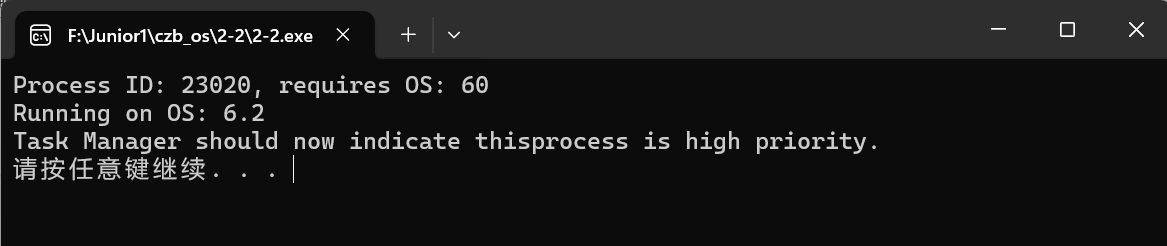
#### 2. 正在运行的进程

**步骤8：**编译完成后，单击“调试”菜单中的“开始执行”命令。

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：



当前PID信息：23020

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：OS : 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 图形用户界面, 应用程序

描述已自动生成 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。

图形用户界面, 应用程序

描述已自动生成

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

**步骤12：**

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：

图片包含 图形用户界面

描述已自动生成

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;

}