操作系统实验指导书

软件学院

2019 年 9 月

目录

《操作系统》实验教学大纲

课程编号：A2130330

课程名称：操作系统

实验学时：8

一、本实验课的性质、任务与目的

操作系统作为软件工程专业的一门专业基础课，是软件工程专业的核心课程之一，学好与否直接关系到学生是否能更好地学习后续课程。通过本实验课程的学习，使学生理解与掌握操作系统设计所遵循的基本原理，基本方法，建立多道程序设计环境下的并行程序设计的思维方式。此外，操作系统用到的各种算法也是学生加强算法锻炼的好机会，对日后从事系统开发方面的工作有直接的借鉴作用。

本实验课程在操作系统原理课程教学中占有重要地位，目的是让学生及时掌握和巩固所学的基本原理和基础理论，加深理解。提高学生自适应能力，为将来设计和开发系统级程序打下良好的基础。

二、本实验课所依据的课程基本理论

计算机操作系统及其介绍的相关系统设计算法。

三、实验类型与要求

序号	实验内容	内容提要	实验时数	实验类型
一	进程控制描述与控制	操作系统界面、进程管理	2	验证
二	并发与调度	进程并发、进程状态转换	2	验证
三	存储管理	内存空间分配及虚拟存储器	2	验证
四	设备管理	磁盘的物理组织	2	验证

四、考核方式与评分办法

综合每次实验课堂表现以及实验完成情况（包含程序的质量、完成及时性及实验报告）。

五、实验报告要求

实验题目、问题描述、算法说明、算法框图、数据结构及符号说明、程序清单及运行结果。

六、实验考核方式与基本要求

1） 按要求设计相应的模拟系统并上机调试运行

2） 写出详细的实验报告，实验报告要求如下：

（1）实验题目。

（2）程序中使用的数据结构及符号说明。

（3）流程图。

（4）源程序（含注释）。

（5）程序运行时的初值和运行结果。

基本要求：4-6 人为一小组，采取课内上机和业余上机相结合的方式进行，在规定时间内上交实验（设计）结果并上机演示说明。

实验一 进程控制描述与控制

[1] Windows“任务管理器”的进程管理

背景知识

Windows Server 2003 的任务管理器提供了用户计算机上正在运行的程序和进程的相关信息，也显示了最常用的度量进程性能的单位。使用任务管理器，可以打开监视计算机性能的关键指示器，快速查看正在运行的程序的状态，或者终止已停止响应的程序。也可以使用多个参数评估正在运行的进程的活动，以及查看 CPU 和内存使用情况的图形和数据。其中：

1) “应用程序”选项卡显示正在运行程序的状态，用户能够结束、切换或者启动程序。

2) “进程”选项卡显示正在运行的进程信息。例如，可以显示关于 CPU 和内存使用情况、页面错误、句柄计数以及许多其他参数的信息。

3) “性能”选项卡显示计算机动态性能，包括 CPU 和内存使用情况的图表，正在运行的句柄、线程和进程的总数，物理、核心和认可的内存总数 (KB) 等。

实验目的

通过在 Windows 任务管理器中对程序进程进行响应的管理操作，熟悉操作系统进程管理的概念，学习观察操作系统运行的动态性能。

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。

需要准备一台运行 Windows Server 2003 操作系统的计算机。

实验内容与步骤

1. 使用任务管理器终止进程
2. 显示其他进程计数器
3. 更改正在运行的程序的优先级

启动并进入 Windows 环境，单击 Ctrl + Alt + Del 键，或者右键单击任务栏，在快捷菜单中单击“任务管理器”命令，打开“任务管理器”窗口。

在本次实验中，你使用的操作系统版本是：

_____

当前机器中由你打开，正在运行的应用程序有：

1) _____
2) _____
3) _____
4) _____
5) _____

Windows“任务管理器”的窗口由_____个选项卡组成，分别是：

1) _____
2) _____
3) _____

当前“进程”选项卡显示的栏目分别是 (可移动窗口下方的游标/箭头，或使窗口最大化进行观察) ：

1) _____
2) _____
3) _____

4) _____

1. 使用任务管理器终止进程

步骤 1：单击“进程”选项卡，一共显示了_____个进程。请试着区分一下，其中：系统 (SYSTEM) 进程有_____个，填入表 3-1 中。

表 3-1　实验记录

映像名称	用户名	CPU	内存使用

服务 (SERVICE) 进程有_____个，填入表 3-2 中。

表 3-2　实验记录

映像名称	用户名	CPU	内存使用

用户进程有_____个，填入表 3-3 中。

表 3-3　实验记录

映像名称	用户名	CPU	内存使用

步骤 2：单击要终止的进程，然后单击“结束进程”按钮。

> **注意：**终止进程时要小心。终止进程有可能导致不希望发生的结果，包括数据丢失和系统不稳定等。因为在被终止前，进程将没有机会保存其状态和数据。如果结束应用程序，您将丢失未保存的数据。如果结束系统服务，系统的某些部分可能无法正常工作。

终止进程，将结束它直接或间接创建的所有子进程。例如，如果终止了电子邮件程序 (如 Outlook 98) 的进程树，那么同时也终止了相关的进程，如 MAPI 后台处理程序 mapisp32.exe。

请将终止某进程后的操作结果与原记录数据对比，发生了什么：

_____
_____
_____
_____

2. 显示其他进程属性

在“进程”选项卡上单击“查看”菜单，然后单击“选择列”命令。单击要增加显示为列标题的项目，然后单击“确定”。

为对进程列表进行排序，可在“进程”选项卡上单击要根据其进行排序的列标题。而为了要反转排序顺序，可再次单击列标题。

经过调整，“进程”选项卡现在显示的项目分别是：

_____
_____
_____

通过对“查看”菜单的选择操作，可以在“任务管理器”中更改显示选项：

- 在“应用程序”选项卡上，可以按详细信息、大图标或小图标查看。
- 在“性能”选项卡上，可以更改 CPU 记录图，并显示内核时间。“显示内核时间”选项在“CPU 使用”和“CPU 使用记录”图表上添加红线。红线指示内核操作占用的 CPU 资源数量。

3. 更改正在运行的程序的优先级

要查看正在运行的程序的优先级，可单击“进程”选项卡，单击“查看”菜单，单击“选择列”-“基本优先级”命令，然后单击“确定”按钮。

为更改正在运行的程序的优先级，可在“进程”选项卡上右键单击您要更改的程序，指向“设置优先级”，然后单击所需的选项。

更改进程的优先级可以使其运行更快或更慢 (取决于是提升还是降低了优先级) ，但也可能对其他进程的性能有相反的影响。

记录操作后所体会的结果：

_____
_____
_____
_____
_____

[2]Windows　Server 2003 进程的“一生”

背景知识

1. 创建进程
2. 正在运行的进程
3. 终止进程

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。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 3-4 详细地列出了每个参数的类型和名称。

表 3-4 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。

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 Server 2003 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。

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 Server 2003 操作系统的计算机。

2) 计算机中需安装 Visual C++ 6.0 专业版或企业版。

实验内容与步骤

1. 创建进程
2. 正在运行的进程
3. 终止进程

请回答：

Windows 所创建的每个进程都是以调用_____ API 函数开始和以调用_____ 或 _____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　Server 2003　。

步骤 2：在“开始”菜单中单击“程序”-“Microsoft Visual Studio 6.0”–“Microsoft Visual C++ 6.0”命令，进入 Visual C++窗口。

步骤 3：在工具栏单击“打开”按钮，在“打开”对话框中找到并打开实验源程序 3-5.cpp。

清单 3-5　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与其 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) ;

        return 0;
    }
```

步骤 4：单击“Build”菜单中的“Compile 3-5.cpp”命令，系统显示：

This build command requires an active project workspace. Would you like to create a default project workspace ?

(build 命令需要一个活动的项目工作空间。你是否希望建立一个缺省的项目工作空间？)

单击“是”按钮确认。系统对 3-5.cpp 进行编译。

步骤 5：编译完成后，单击“Build”菜单中的“Build 3-5.exe”命令，建立 3-5.exe 可执行文件。

操作能否正常进行？如果不行，则可能的原因是什么？

_____

_____

步骤 6：在工具栏单击“Execute Program”(执行程序) 按钮，或者按 Ctrl + F5 键，或者单击“Build”菜单中的“Execute 3-5.exe”命令，执行 3-5.exe 程序。

步骤 7：按 Ctrl + S 键可暂停程序的执行，按 Ctrl + Pause (Break) 键可终止程序的执行。

清单 3-5 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：

_____

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_____个核心参数？本实验程序中设置的各个参数的值是：

a. _____；
b. _____；
c. _____；
d. _____；
e. _____；
f. _____；
g. _____；
h. _____。

程序运行时屏幕显示的信息是：

_____

_____

提示：部分程序在 Visual C++环境完成编译、链接之后，还可以在 Windows Server 2003 的“命令提示符”状态下尝试执行该程序，看看与在可视化界面下运行的结果有没有不同？为什么？

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 8：在 Visual C++ 窗口的工具栏中单击“打开”按钮，在“打开”对话框中找到并打开实验源程序 3-6.cpp。

清单 3-6 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
        :: GetVersionEx(reinterpret_cast < LPOSVERSIONINFO > (&osvix) ) ;
        std :: cout << "Running on OS: " << osvix.dwMajorVersion << "."
            << osvix.dwMinorVersion << std :: endl;

        // 如果是 NTS (Windows  Server 2003) 系统，则提高其优先权
        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 now indicate this"
                "process is high priority." << std :: endl;
        }
    }
```

步骤 9：单击“Build”菜单中的“Compile 3-6.cpp”命令，再单击“是”按钮确认。系统对 3-6.cpp 进行编译。

步骤 10：编译完成后，单击“Build”菜单中的“Build 3-6.exe”命令，建立 3-6.exe 可执行文件。

操作能否正常进行？如果不行，则可能的原因是什么？

_____

_____

步骤 11：在工具栏单击“Execute Program” (执行程序) 按钮，执行 3-6.exe 程序。

运行结果：

当前 PID 信息：_____

当前操作系统版本：_____

系统提示信息：_____

_____

清单 3-6 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 : 5.0”表示当前运行的操作系统是：

_____

清单 3-6 的最后一段程序利用了操作系统的版本信息，以确认运行的是 Windows Server 2003。代码接着将当前进程的优先级提高到比正常级别高。

步骤 12：单击 Ctrl + Alt + Del 键，进入“Windows 任务管理器”，在“应用程序”选项卡中右键单击“3-6”任务，在快捷菜单中选择“转到进程”命令。

在“Windows 任务管理器”的“进程”选项卡中，与“3-6”任务对应的进程映像名称是(为什么？) ：

_____

右键单击该进程名，在快捷菜单中选择“设置优先级”命令，可以调整该进程的优先级，如设置为“高”后重新运行 3-6.exe 程序，屏幕显示有变化吗？为什么？

_____

_____

_____

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 3-7 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 13：在 Visual C++ 窗口的工具栏中单击“打开”按钮，在“打开”对话框中找到并打开实验源程序 3-7.cpp。

清单 3-7　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() ;
    }
```

```
    return 0;
}
```

清单 3-7 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 14：单击“Build”菜单中的“Compile 3-7.cpp”命令，再单击“是”按钮确认。系统对 3-7.cpp 进行编译。

步骤 15：编译完成后，单击“Build”菜单中的“Build 3-7.exe”命令，建立 3-7.exe 可执行文件。

操作能否正常进行？如果不行，则可能的原因是什么？

_____

_____

步骤 16：在工具栏单击“Execute Program”按钮，执行 3-7.exe 程序。

运行结果：

1) _____

表示：_____

2) _____

表示：_____

步骤 17：在熟悉清单 3-7 源代码的基础上，利用本实验介绍的 API 函数来尝试改进本程序 (例如使用 GetProcessTimes() API 函数) 并运行。

实验二 并发与调度

[1]Windows Server 2003 线程同步

背景知识

Windows Server 2003 提供的常用对象可分成三类：核心应用服务、线程同步和线程间通讯。其中，开发人员可以使用线程同步对象来协调线程和进程的工作，以使其共享信息并执行任务。此类对象包括互锁数据、临界段、事件、互斥体和信号等。

多线程编程中关键的一步是保护所有的共享资源，工具主要有互锁函数、临界段和互斥体等；另一个实质性部分是协调线程使其完成应用程序的任务，为此，可利用内核中的事件对象和信号。

在进程内或进程间实现线程同步的最方便的方法是使用事件对象，这一组内核对象允许一个线程对其受信状态进行直接控制 (见表 4-1) 。

而互斥体则是另一个可命名且安全的内核对象，其主要目的是引导对共享资源的访问。拥有单一访问资源的线程创建互斥体，所有想要访问该资源的线程应该在实际执行操作之前获得互斥体，而在访问结束时立即释放互斥体，以允许下一个等待线程获得互斥体，然后接着进行下去。

与事件对象类似，互斥体容易创建、打开、使用并清除。利用 CreateMutex() API 可创建互斥体，创建时还可以指定一个初始的拥有权标志，通过使用这个标志，只有当线程完成了资源的所有的初始化工作时，才允许创建线程释放互斥体。

表 4-1 用于管理事件对象的 API

API 名称	描述
CreateEvent()	在内核中创建一个新的事件对象。此函数允许有安全性设置、手工还是自动重置的标志以及初始时已接受还是未接受信号状态的标志
OpenEvent()	创建对已经存在的事件对象的引用。此 API 函数需要名称、继承标志和所需的访问级别
SetEvent()	将手工重置事件转化为已接受信号状态
ResetEvent()	将手工重置事件转化为非接受信号状态
PulseEvent()	将自动重置事件对象转化为已接受信号状态。当系统释放所有的等待它的线程时此种转化立即发生

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；然后，线程将这个句柄提供给一个等待函数。当内核将互斥体对象发送给等待线程时，就表明该线程获得了互斥体的拥有权。当线程获得拥有权时，线程控制了对共享资源的访问——必须设法尽快地放弃互斥体。放弃共享资源时需要在该对象上调用 ReleaseMute() API。然后系统负责将互斥体拥有权传递给下一个等待着的线程 (由到达时间决定顺序) 。

实验目的

在本实验中，通过对事件和互斥体对象的了解，来加深对 Windows Server 2003 线程同步的理解。

1) 回顾系统进程、线程的有关概念，加深对 Windows Server 2003 线程的理解。
2) 了解事件和互斥体对象。
3) 通过分析实验程序，了解管理事件对象的 API。
4) 了解在进程中如何使用事件对象。
5) 了解在进程中如何使用互斥体对象。

6) 了解父进程创建子进程的程序设计方法。

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。
您需要做以下准备：
1) 一台运行 Windows Server 2003 操作系统的计算机。
2) 计算机中需安装 Visual C++ 6.0 专业版或企业版。

实验内容与步骤

1. 事件对象

清单 4-1 程序展示了如何在进程间使用事件。父进程启动时，利用 CreateEvent() API 创建一个命名的、可共享的事件和子进程，然后等待子进程向事件发出信号并终止父进程。在创建时，子进程通过 OpenEvent() API 打开事件对象，调用 SetEvent() API 使其转化为已接受信号状态。两个进程在发出信号之后几乎立即终止。

步骤 1：登录进入 Windows Server 2003 。

步骤 2：在“开始”菜单中单击“程序”-“Microsoft Visual Studio 6.0”–“Microsoft Visual C++ 6.0”命令，进入 Visual C++窗口。

步骤 3：在工具栏单击“打开”按钮，在“打开”对话框中找到并打开实验源程序 4-1.cpp。

清单 4-1 创建和打开事件对象在进程间传送信号

```
// event 项目
# include <windows.h>
# include <iostream>

// 以下是句柄事件。实际中很可能使用共享的包含文件来进行通讯
static LPCTSTR g_szContinueEvent = "w2kdg.EventDemo.event.Continue";

// 本方法只是创建了一个进程的副本，以子进程模式 (由命令行指定) 工作
BOOL CreateChild()
{
    // 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);             // 必须是本结构的大小

    // 返回的子进程的进程信息结构
    PROCESS_INFORMATION pi;

    // 使用同一可执行文件和告诉它是一个子进程的命令行创建进程
    BOOL bCreateOK = :: CreateProcess(
        szFilename,             // 生成的可执行文件名
        szCmdLine,                  // 指示其行为与子进程一样的标志
```

```
        NULL,                   // 子进程句柄的安全性
        NULL,                   // 子线程句柄的安全性
        FALSE,                      // 不继承句柄
        0,                      // 特殊的创建标志
        NULL,                   // 新环境
        NULL,                   // 当前目录
        &si,                        // 启动信息结构
        &pi ) ;                 // 返回的进程信息结构

        // 释放对子进程的引用
        if (bCreateOK)
            {
             :: CloseHandle(pi.hProcess);
             :: CloseHandle(pi.hThread);
        }
        return(bCreateOK) ;
    }

    // 下面的方法创建一个事件和一个子进程，然后等待子进程在返回前向事件发出信号
    void WaitForChild()
    {
        // create a new event object for the child process
        // to use when releasing this process
        HANDLE hEventContinue = :: CreateEvent(
            NULL,                   // 缺省的安全性，子进程将具有访问权限
            TRUE,                   // 手工重置事件
            FALSE,                      // 初始时是非接受信号状态
            g_szContinueEvent);         // 事件名称
        if (hEventContinue!=NULL)
        {
                std :: cout << "event created " << std :: endl;

                // 创建子进程
            if (:: CreateChild())
            {
                 std :: cout << " chlid created" << std :: endl;

                 // 等待，直到子进程发出信号
                 std :: cout << "Parent waiting on child." << std :: endl;
                 :: WaitForSingleObject(hEventContinue, INFINITE);

                 :: Sleep(1500);                    // 删去这句试试
                    std :: cout << "parent received the envent signaling from child" << std ::
endl;
            }

            // 清除句柄
            :: CloseHandle(hEventContinue);
            hEventContinue = INVALID_HANDLE_VALUE;
        }
    }

    // 以下方法在子进程模式下被调用，其功能只是向父进程发出终止信号
    void SignalParent()
    {
```

```
        // 尝试打开句柄
            std :: cout << "child process begining....." << std :: endl;
        HANDLE hEventContinue = :: OpenEvent(
              EVENT_MODIFY_STATE,          // 所要求的最小访问权限
              FALSE,                       // 不是可继承的句柄
        g_szContinueEvent);                // 事件名称
        if (hEventContinue != NULL)
            {
             :: SetEvent(hEventContinue);
                std :: cout << "event signaled" << std :: endl;
        }

        // 清除句柄
        :: CloseHandle(hEventContinue) ;
        hEventContinue = INVALID_HANDLE_VALUE;
    }

    int main(int argc, char* argv[] )
    {
        // 检查父进程或是子进程是否启动
        if (argc>1 && :: strcmp(argv[1] , "child" )== 0)
        {
            // 向父进程创建的事件发出信号
             :: SignalParent() ;
        }
        else
        {
                  // 创建一个事件并等待子进程发出信号
             :: WaitForChild() ;
                 :: Sleep(1500) ;
             std :: cout << "Parent released." << std :: endl ;
        }
        return 0;
    }
```

步骤 4：单击“Build”菜单中的“Compile 4-1.cpp”命令，并单击“是”按钮确认。系统对 4-1.cpp 进行编译。

步骤 5：编译完成后，单击“Build”菜单中的“Build 4-1.exe”命令，建立 4-1.exe 可执行文件。

操作能否正常进行？如果不行，则可能的原因是什么？

_____

_____

步骤 6：在工具栏单击“Execute Program” (执行程序) 按钮，执行 4-1.exe 程序。

运行结果 (分行书写。如果运行不成功，则可能的原因是什么？) ：

1) _____

2) _____

3) _____

4) _____

5) _____

6) _____

这个结果与你期望的一致吗？(从进程并发的角度对结果进行分析)

阅读和分析程序 4-1，请回答：

1) 程序中，创建一个事件使用了哪一个系统函数？创建时设置的初始信号状态是什么？

   a. _____

   b. _____

2) 创建一个进程 (子进程) 使用了哪一个系统函数？

_____

3) 从步骤 6 的输出结果，对照分析 4-1 程序，可以看出程序运行的流程吗？请简单描述：

_____
_____
_____
_____
_____
_____

2. 互斥体对象

清单 4-2 的程序中显示的类 CCountUpDown 使用了一个互斥体来保证对两个线程间单一数值的访问。每个线程都企图获得控制权来改变该数值，然后将该数值写入输出流中。创建者实际上创建的是互斥体对象，计数方法执行等待并释放，为的是共同使用互斥体所需的资源 (因而也就是共享资源) 。

步骤 7：在 Visual C++ 窗口的工具栏中单击“打开”按钮，在“打开”对话框中找到并打开实验源程序 4-2.cpp。

清单 4-2　利用互斥体保护共享资源

```
// mutex 项目
# include <windows.h>
# include <iostream>
// 利用互斥体来保护同时访问的共享资源
class CCountUpDown
{
  public:
    // 创建者创建两个线程来访问共享值
    CCountUpDown(int nAccesses) :
      m_hThreadInc(INVALID_HANDLE_VALUE) ,
      m_hThreadDec(INVALID_HANDLE_VALUE) ,
      m_hMutexValue(INVALID_HANDLE_VALUE) ,
      m_nValue(0) ,
      m_nAccess(nAccesses)
    {
      // 创建互斥体用于访问数值
      m_hMutexValue = :: CreateMutex(
          NULL,              // 缺省的安全性
          TRUE,              // 初始时拥有，在所有的初始化结束时将释放
          NULL) ;            // 匿名的
      m_hThreadInc = :: CreateThread(
          NULL,              // 缺省的安全性
          0,                 // 缺省堆栈
          IncThreadProc,     // 类线程进程
          reinterpret_cast <LPVOID> (this) ,          // 线程参数
          0,                 // 无特殊的标志
          NULL) ;            // 忽略返回的 id
      m_hThreadDec = :: CreateThread(
          NULL,              // 缺省的安全性
          0,                 // 缺省堆栈
          DecThreadProc,     // 类线程进程
          reinterpret_cast <LPVOID> (this) ,          // 线程参数
          0,                 // 无特殊的标志
          NULL) ;            // 忽略返回的 id
```

```
        // 允许另一线程获得互斥体
        :: ReleaseMutex(m_hMutexValue) ;
    }

    // 解除程序释放对对象的引用
virtual ~CCountUpDown()
    {
        :: CloseHandle(m_hThreadInc) ;
        :: CloseHandle(m_hThreadDec) ;
        :: CloseHandle(m_hMutexValue) ;
    }

    // 简单的等待方法，在两个线程终止之前可暂停主调者
    virtual void WaitForCompletion()
    {
        // 确保所有对象都已准备好
        if (m_hThreadInc != INVALID_HANDLE_VALUE &&
            m_hThreadDec != INVALID_HANDLE_VALUE)
        {
            // 等待两者完成 (顺序并不重要)
            :: WaitForSingleObject(m_hThreadInc, INFINITE) ;
            :: WaitForSingleObject(m_hThreadDec, INFINITE) ;
        }
    }

protected:
    // 改变共享资源的简单的方法
    virtual void DoCount(int nStep)
    {
        // 循环，直到所有的访问都结束为止
        while (m_nAccess > 0)
        {
            // 等待访问数值
            :: WaitForSingleObject(m_hMutexValue, INFINITE) ;

            // 改变并显示该值
            m_nValue += nStep;

            std :: cout << "thread: " << :: GetCurrentThreadId()
                        << "value: " << m_nValue
                        << "access: " << m_nAccess << std :: endl;

            // 发出访问信号并允许线程切换
            --m_nAccess;
            :: Sleep(1000) ;                // 使显示速度放慢

            // 释放对数值的访问
            :: ReleaseMutex(m_hMutexValue) ;
        }
    }
    static DWORD WINAPI IncThreadProc(LPVOID lpParam)
    {
        // 将参数解释为 'this' 指针
        CCountUpDown* pThis =
            reinterpret_cast < CCountUpDown* > (lpParam) ;
```

```
 // 调用对象的增加方法并返回一个值
                pThis -> DoCount(+1) ;
                return(0) ;
            }

            static DWORD WINAPI DecThreadProc(LPVOID lpParam)
            {
                // 将参数解释为 'this' 指针
                CCountUpDown* pThis =
                   reinterpret_cast <CCountUpDown* > (lpParam) ;
                // 调用对象的减少方法并返回一个值
                pThis -> DoCount(-1) ;
                return(0) ;
            }

protected:
            HANDLE m_hThreadInc;
            HANDLE m_hThreadDec;

            HANDLE m_hMutexValue;
            int m_nValue;
            int m_nAccess ;
} ;

        void main()
        {
            CCountUpDown ud(50) ;
            ud.WaitForCompletion() ;
        }
```

步骤 8：单击“Build”菜单中的“Compile 4-2.cpp”命令，并单击“是”按钮确认。系统对 4-2.cpp 进行编译。

步骤 9：编译完成后，单击“Build”菜单中的“Build 4-2.exe”命令，建立 4-2.exe 可执行文件。

操作能否正常进行？如果不行，则可能的原因是什么？

_____

_____

步骤 10：在工具栏单击“Execute Program”按钮，执行 4-2.exe 程序。

分析程序 4-2 的运行结果，可以看到线程 (加和减线程) 的交替执行 (因为 Sleep() API 允许 Windows 切换线程) 。在每次运行之后，数值应该返回初始值 (0) ，因为在每次运行之后写入线程在等待队列中变成最后一个，内核保证它在其他线程工作时不会再运行。

1) 请描述运行结果 (如果运行不成功，则可能的原因是什么？) ：

_____

_____

_____

_____

_____

2) 根据运行输出结果，对照分析 4-2 程序，可以看出程序运行的流程吗？请简单描述：

_____

_____

_____

_____

_____

_____

实验三 存储管理

[1]Windows Server 2003 内存结构

背景知识

Windows Server 2003 是 32 位的操作系统，它使计算机 CPU 可以用 32 位地址对 32 位内存块进行操作。内存中的每一个字节都可以用一个 32 位的指针来寻址。这样，最大的存储空间就是 232 字节或 4000 兆字节 (4GB) 。这样，在 Windows 下运行的每一个应用程序都认为能独占可能的 4GB 大小的空间

而另一方面，尽管现在的内存容量已经很大，但历史上实际上没有几台机器的 RAM 能达到 4GB，更不必说让每个进程都独享 4GB 内存。Windows 在幕后将虚拟内存 (virtual memory，VM) 地址映射到了各进程的物理内存地址上。而这里所谓的物理内存是指计算机的 RAM 和由 Windows 分配到用户驱动器根目录上的换页文件。

实验目的

1） 通过实验了解 windows Server 2003 内存的使用，学习如何在应用程序中管理内存、体会 Windows 应用程序内存的基本原理以及使用方法。
2） 了解 windows Server 2003 的内存结构和虚拟内存的管理，进而了解 Windows 为使用内存而提供的一些扩展功能。

工具/准备工作

您需要做以下准备：
一台运行 Windows Server 2003 操作系统的计算机
计算机中需安装 Visual C++ 6.0 专业版或企业版

实验内容与步骤

Windows 提供了一个 API 即 GetSystemInfo() ，以便用户能检查系统中虚拟内存的一些特性。程序 5-1 显示了如何调用该函数以及显示系统中当前内存的参数。

步骤 1：登录进入 Windows Server 2003 。

步骤 2：在“开始”菜单中单击“程序”-“Microsoft Visual Studio 6.0”–“Microsoft Visual C++ 6.0”命令，进入 Visual C++窗口。

步骤 3：在工具栏单击“打开”按钮，在“打开”对话框中找到并打开实验源程序 5-1.cpp。

程序 5-1：获取有关系统的内存设置的信息

```
// 工程 vmeminfo
#include <windows.h>
#include <iostream>
#include <shlwapi.h>
#include <iomanip>
#pragma comment(lib, "shlwapi.lib")

void main( )
{
    // 首先，让我们获得系统信息
    SYSTEM_INFO si;
    :: ZeroMemory(&si, sizeof(si) ) ;
    :: GetSystemInfo(&si) ;
```

```
        //格式化
        TCHAR szPageSize[MAX_PATH];
        ::StrFormatByteSize(si.dwPageSize, szPageSize, MAX_PATH) ;

         TCHAR dwAllocationGranularity[MAX_PATH];
        ::StrFormatByteSize(si.dwAllocationGranularity,          dwAllocationGranularity,
MAX_PATH) ;

       DWORD dwMemSize = (DWORD)si.lpMaximumApplicationAddress -
              (DWORD) si.lpMinimumApplicationAddress+1;
        TCHAR szMemSize [MAX_PATH] ;
        :: StrFormatByteSize(dwMemSize, szMemSize, MAX_PATH) ;

        // 将内存信息显示出来
        std :: cout << "Virtual memory page size: " << szPageSize << std :: endl;
        std :: cout << "虚拟内存粒度: " << dwAllocationGranularity << std :: endl;

        std :: cout.fill ('0') ;
        std :: cout << "Minimum application address: 0x"
                << std :: hex << std :: setw(8)
                << (DWORD) si.lpMinimumApplicationAddress
                << std :: endl;
        std :: cout << "Maximum application address: 0x"
                << std :: hex << std :: setw(8)
                << (DWORD) si.lpMaximumApplicationAddress
                << std :: endl;

        std :: cout << "Total available virtual memory: "
                << szMemSize << std :: endl ;
    }
```

步骤 4：单击“Build”菜单中的“Compile 5-1.cpp”命令，并单击“是”按钮确认。系统对 4-1.cpp 进行编译。

步骤 5：编译完成后，单击“Build”菜单中的“Build 5-1.exe”命令，建立 5-1.exe 可执行文件。

操作能否正常进行？如果不行，则可能的原因是什么？

_____

_____

步骤 6：在工具栏单击“Execute Program” (执行程序) 按钮，执行 5-1.exe 程序。

运行结果 (分行书写。如果运行不成功，则可能的原因是什么？) ：

1) 虚拟内存每页容量为：_____

2) 最小应用地址：_____

3) 最大应用地址为：_____

4) 当前可供应用程序使用的内存空间为：_____

5) 当前计算机的实际内存大小为：_____

阅读和分析程序 5-1，请回答问题：

1) 理论上每个 windows 应用程序可以独占的最大存储空间是：_____

2) 在程序 5-1 中，用于检索系统中虚拟内存特性的 API 函数是：_____

提示：可供应用程序使用的内存空间实际上已经减去了开头与结尾两个 64KB 的保护区。虚拟内存空间中的 64KB 保护区是防止编程错误的一种 Windows 方式。任何对内存中这一区域的访问（读、写、执行）都将引发一个错误陷井，从而导致错误并终止程序的执行。也就是说，假如用户有一个 NULL 指针（地址为 0），但仍试图在此之前很近的地址处使用另一个指针，这将因为试图从更低的保留区域读写数据，从而产生意外错误并终止程序的执行。

[2]Windows Server 2003 虚拟内存

背景知识

在 Windows Server 2003 环境下，4GB 的虚拟地址空间被划分成两个部分：低端 2GB 提供给进程使用，高端 2GB 提供给系统使用。这意味着用户的应用程序代码，包括 DLL 以及进程使用的各种数据等，都装在用户进程地址空间内 (低端 2GB) 。用户过程的虚拟地址空间也被分成三部分：

1）虚拟内存的已调配区 (committed) ：具有备用的物理内存，根据该区域设定的访问权限，用户可以进行写、读或在其中执行程序等操作。

2）虚拟内存的保留区 (reserved) ：没有备用的物理内存，但有一定的访问权限。

3）虚拟内存的自由区 (free) ：不限定其用途，有相应的 PAGE_NOACCESS 权限。

与虚拟内存区相关的访问权限告知系统进程可在内存中进行何种类型的操作。例如，用户不能在只有 PAGE_READONLY 权限的区域上进行写操作或执行程序；也不能在只有 PAGE_EXECUTE 权限的区域里进行读、写操作。而具有 PAGE_ NOACCESS 权限的特殊区域，则意味着不允许进程对其地址进行任何操作。

在进程装入之前，整个虚拟内存的地址空间都被设置为只有 PAGE_NOACCESS 权限的自由区域。当系统装入进程代码和数据后，才将内存地址的空间标记为已调配区或保留区，并将诸如 EXECUTE、READWRITE 和 READONLY 的权限与这些区域相关联。

如表 3-2 所示，给出了 MEMORY_BASIC_INFORMAITON 的结构，此数据描述了进程虚拟内存空间中的一组虚拟内存页面的当前状态，其中 State 项表明这些区域是否为自由区、已调配区或保留区；Protect 项则包含了 windows 系统为这些区域添加了何种访问保护；type 项则表明这些区域是可执行镜像、内存映射文件还是简单的私有内存。VirsualQueryEX() API 能让用户在指定的进程中，对虚拟内存地址的大小和属性进行检测。

Windows 还提供了一整套能使用户精确控制应用程序的虚拟地址空间的虚拟内存 API。一些用于虚拟内存操作及检测的 API 如表 3-2 所示。

表 3-1 MEMORY_BASIC_INFORMAITON 结构的成员

成员名称	目的
PVOID BaseAddress	虚拟内存区域开始处的指针
PVOID AllocationBase	如果这个特定的区域为子分配区的话，则为虚拟内存外面区域的指针；否则此值与 BaseAddress 相同
DWORD AllocationProtect	虚拟内存最初分配区域的保护属性。其可能值包括：PAGE_NOACCESS,PAGE_READONLY,PAGE_READWRITE 和 PAGE_EXECUTE_READ
DWORD RegionSize	虚拟内存区域的字节数
DWORD State	区域的当前分配状态。其可能值为 MEM_COMMIT,MEM_PREE 和 MEM_RESERVE
DWORD Protect	虚拟内存当前的保护属性。可能值与 AllocationProtect 成员的相同
DWORD Type	虚拟内存区域中出现的页面类型。可能值为 MEM_IMAGE,MEM_MAPPED 和 MEM_PRIVATE

表 3-2 虚拟内存的 API

API 名称	描述
VisualQueryEX()	通过填充 MEMORY_BASIC_INFORMATION 结构检测进程内虚拟内存的区域
VisualAlloc()	保留或调配进程的部分虚拟内存，设置分配和保护标志

VirsualFree()	释放或收回应用程序使用的部分虚拟地址
VirsualProtect()	改变虚拟内存区域保护规范
VirsualLock()	防止系统将虚拟内存区域通过系统交换到页面文件中
VirsualUnlock()	释放虚拟内存的锁定区域，必要时，允许系统将其交换到页面文件中

提供虚拟内存分配功能的是 VirtualAlloc( ) API。该 API 支持用户向系统要求新的虚拟内存或改变已分配内存的当前状态。用户若想通过 VirtualAlloc() 函数使用虚拟内存，可以采用两种方式通知系统：

1)简单地将内存内容保存在地址空间内

2)请求系统返回带有物理存储区 (RAM 的空间或换页文件) 的部分地址空间

用户可以用 flAllocation Type 参数 (commit 和 reserve) 来定义这些方式，用户可以通知 Windows 按只读、读写、不可读写、执行或特殊方式来处理新的虚拟内存。

与 VirtualAlloc() 函数对应的是 VirtualFree() 函数，其作用是释放虚拟内存中的已调配页或保留页。用户可利用 dwFree Type 参数将已调配页修改成保留页属性。

VirtualProtect() 是 VirtualAlloc() 的一个辅助函数，利用它可以改变虚拟内存区的保护规范。

实验目的

1）通过实验了解 Windows Server 2003 内存的使用，学习如何在应用程序中管理内存，体会 Windows 应用程序内存的简单性和自我防护能力。

2）学习检查虚拟内存空间或对其进行操作。

3)了解 Windows Server 2003 的内存结构和虚拟内存的管理,进而了解进程堆和 Windows 为使用内存而提供的一些扩展功能。

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。

您需要做以下准备：

1) 一台运行 Windows Server 2003 操作系统的计算机。

2) 计算机中需安装 Visual C++ 6.0 专业版或企业版。

实验内容与步骤

1. 虚拟内存的检测

清单 5-2 所示的程序使用 VirtualQueryEX()函数来检查虚拟内存空间。

步骤 1：登录进入 Windows Server 2003。

步骤 2：在“开始”菜单中单击“程序”-“Microsoft Visual Studio 6.0”–“Microsoft Visual C++ 6.0”命令，进入 Visual C++窗口。

步骤 3：在工具栏单击“打开”按钮，在“打开”对话框中找到并打开实验源程序 5-2.cpp。

清单 5-2 检测进程的虚拟地址空间

```
// 工程 vmwalker
#include <windows.h>
#include <iostream>
#include <shlwapi.h>
#include <iomanip>
#pragma comment(lib, "Shlwapi.lib")
```

```
// 以可读方式对用户显示保护的辅助方法。
// 保护标记表示允许应用程序对内存进行访问的类型
// 以及操作系统强制访问的类型
inline bool TestSet(DWORD dwTarget, DWORD dwMask)
{
    return ((dwTarget &dwMask) == dwMask) ;
}
# define SHOWMASK(dwTarget, type) \
if (TestSet(dwTarget, PAGE_##type) ) \
    {std :: cout << ", " << #type; }
void ShowProtection(DWORD dwTarget)
{
    SHOWMASK(dwTarget, READONLY) ;
    SHOWMASK(dwTarget, GUARD) ;
    SHOWMASK(dwTarget, NOCACHE) ;
    SHOWMASK(dwTarget, READWRITE) ;
    SHOWMASK(dwTarget, WRITECOPY) ;
    SHOWMASK(dwTarget, EXECUTE) ;
    SHOWMASK(dwTarget, EXECUTE_READ) ;
    SHOWMASK(dwTarget, EXECUTE_READWRITE) ;
    SHOWMASK(dwTarget, EXECUTE_WRITECOPY) ;
    SHOWMASK(dwTarget, NOACCESS) ;
}

// 遍历整个虚拟内存并对用户显示其属性的工作程序的方法
void WalkVM(HANDLE hProcess)
{
    // 首先，获得系统信息
    SYSTEM_INFO si;
    :: ZeroMemory(&si, sizeof(si) ) ;
    :: GetSystemInfo(&si) ;

    // 分配要存放信息的缓冲区
    MEMORY_BASIC_INFORMATION mbi;
    :: ZeroMemory(&mbi, sizeof(mbi) ) ;

    // 循环整个应用程序地址空间
    LPCVOID pBlock = (LPVOID) si.lpMinimumApplicationAddress;
    while (pBlock < si.lpMaximumApplicationAddress)
    {
        // 获得下一个虚拟内存块的信息
        if (:: VirtualQueryEx(
            hProcess,                       // 相关的进程
            pBlock,                           // 开始位置
            &mbi,                             // 缓冲区
            sizeof(mbi))==sizeof(mbi) )     // 大小的确认
        {
            // 计算块的结尾及其大小
            LPCVOID pEnd = (PBYTE) pBlock + mbi.RegionSize;
            TCHAR szSize[MAX_PATH];
            :: StrFormatByteSize(mbi.RegionSize, szSize, MAX_PATH) ;

            // 显示块地址和大小
            std :: cout.fill ('0') ;
            std :: cout
                << std :: hex << std :: setw(8) << (DWORD) pBlock
```

```
            << "-"
            << std :: hex << std :: setw(8) << (DWORD) pEnd
            //<< " (" << szSize
            << (:: strlen(szSize)==7? " ("    : " ( ")<< szSize
            << ") " ;

        // 显示块的状态
        switch(mbi.State)
        {
            case MEM_COMMIT :
                std :: cout << "Committed" ;
                break;
            case MEM_FREE :
                std :: cout << "Free" ;
                break;
            case MEM_RESERVE :
                std :: cout << "Reserved" ;
                break;
        }

        // 显示保护
        if(mbi.Protect==0 && mbi.State!=MEM_FREE)
        {
            mbi.Protect=PAGE_READONLY;
        }
        ShowProtection(mbi.Protect);

        // 显示类型
        switch(mbi.Type){
            case MEM_IMAGE :
                std :: cout << ", Image" ;
                break;
            case MEM_MAPPED:
                std :: cout << ", Mapped";
                break;
            case MEM_PRIVATE :
                std :: cout << ", Private" ;
                break;
        }

        // 检验可执行的影像
        TCHAR szFilename [MAX_PATH] ;
        if (:: GetModuleFileName (
            (HMODULE) pBlock,          // 实际虚拟内存的模块句柄
            szFilename,                //完全指定的文件名称
            MAX_PATH)>0)                  //实际使用的缓冲区大小
        {
            // 除去路径并显示
            :: PathStripPath(szFilename) ;
            std :: cout << ", Module: " << szFilename;
        }

        std :: cout << std :: endl;
        // 移动块指针以获得下一下个块
        pBlock = pEnd;
    }
```

```
        }
    }

    void main()
    {
        // 遍历当前进程的虚拟内存
        ::WalkVM(::GetCurrentProcess());
    }
```

清单 5-2 中显示一个 walkVM()函数开始于某个进程可访问的最低端虚拟地址处，并在其中显示各块虚拟内存的特性。虚拟内存中的块由 VirsualQueryEX()API 定义成连续快或具有相同状态（自由区，已调配区等）的内存，并分配以一组统一的保护标志（只读、可执行等）。

步骤 4：单击“Build”菜单中的“Compile 5-2.cpp”命令，并单击“是”按钮确认。系统对 5-2.cpp 进行编译。

步骤 5：编译完成后，单击“Build”菜单中的“Build 5-2.exe”命令，建立 5-2.exe 可执行文件。

操作能否正常进行？如果不行，则可能的原因是什么？

_____

_____

步骤 6：在工具栏单击“Execute Program” (执行程序) 按钮，执行 5-2.exe 程序。

1）分析运行结果（如果运行不成功，则可能的原因是什么）

按 committed,reserved,free 等三种虚拟地址空间分别记录实验数据，其中“描述”是对该组数据的简单描述，例如，对下列一组数据：

00010000-00012000<8.00KB>Committed,READWRITE,Private 可描述为：具有 READWRITE 权限的已调配私有内存区。

将系统当前的自由区（Free）虚拟地址空间填入表 3-3 中。

表 3-3 实验记录

地址	大小	虚拟空间类型	访问权限	描述
		free		
		free		
		free		
		free		
		free		
		free		
		free		

将系统当前的已调配区（Committed）虚拟地址空间填入表 3-4 中。

表 3-4 实验记录

地址	大小	虚拟空间类型	访问权限	描述
		Committed		
		Committed		
		Committed		
		Committed		
		Committed		
		Committed		
		Committed		

将系统当前的保留区（Reserved）虚拟地址空间填入表 3-5 中。

表 3-5 实验记录

地址	大小	虚拟空间类型	访问权限	描述
		Reserved		
		Reserved		
		Reserved		
		Reserved		
		Reserved		
		Reserved		
		Reserved		

2)从上述输出结果，对照分析清单 5-2 的程序，请简单描述程序运行的流程：

_____
_____
_____
_____
_____
_____

2. 虚拟内存的分配与释放

能正确使用系统函数 GetMeoryStatus()和数据结构 MEMORY_STATUS 了解系统内存和虚拟存储空间使用情况，会使用 VirsualAlloc（）函数和 VirsualFree（）函数分配和释放虚拟内存空间。

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

#include "stdafx.h"
#include "GetMemoryStatus.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

 void GetMemSta(void);
//The one and only application object
CWinApp theApp;
using namespace std;

int _tmain(int argc,TCHAR * argv[],TCHAR * envp[])
{
   int nRetCode=0;
   LPVOID BaseAddr;
   char *str;

   GetMemSta();
   printf("Now Allocate 32M Virsual Memory and 2M Physical Memory\n\n");

  BaseAddr=::VirtualAlloc(NULL,1024*1024*32,MEM_RESERVE|MEM_COMMIT,PAG
  E_READWRITE);//分配虚拟内存
  if (BaseAddr==NULL) printf("Virsual Allocate Fail.\n");
  str=(char *)malloc(1024*1024*2);                    //分配内存
```

```
    GetMemSta();
    printf("Now Release 32M Virsual Memory and 2M Physical Memory\n\n");
    if (::VirtualFree(BaseAddr,0,MEM_RELEASE)==0)                    //释放虚拟内存
        printf("Release Allocate Fail.\n");
    free(str);                                                       //释放内存
    GetMemSta();
    return nRetCode;
}

void GetMemSta(void)
{
    MEMORYSTATUS MemInfo;
    GlobalMemoryStatus(&MemInfo);

    printf("Current Memory Status is :\n");
    printf("\t Total Physical Memory is %d MB\n",MemInfo.dwTotalPhys/(1024*1024));
    printf("\t Available Physical Memory is %d MB\n",MemInfo.dwAvailPhys/(1024*1024));
    printf("\t Total Page File is %d MB\n",MemInfo.dwTotalPageFile/(1024*1024));
    printf("\t Available Page File is %d MB\n",MemInfo.dwAvailPageFile/(1024*1024));
    printf("\t Total Virtual Memory is %d MB\n",MemInfo.dwTotalVirtual/(1024*1024));
    printf("\t Available Virsual memory is %d MB\n",MemInfo.dwAvailVirtual/(1024*1024));
    printf("\t Memory Load is %d %%\n\n",MemInfo.dwMemoryLoad);

}
```

步骤 1：在 VC 6.0 环境下选择 Win32 Console Application 建立一个控制台工程文件，选择 An application that Supports MFC。

步骤 2：编辑并编译完成后，单击“Build”菜单中的“Build GetMemoryStatus.exe”命令，建立 GetMemoryStatus.exe 可执行文件。

操作能否正常进行？如果不行，则可能的原因是什么？

_____

步骤 3：在工具栏单击“Execute Program”按钮，执行 GetMemoryStatus.cpp.exe 程序。

分析程序 GetMemoryStatus.cpp 的运行结果

1) 请描述运行结果 (如果运行不成功，则可能的原因是什么？)：

_____

_____

_____

_____

_____

2) 根据运行输出结果，若要改变分配和回收的虚拟内存和物理内存的大小，要改变程序代码的语句，分别为：

_____

_____

3) 根据运行输出结果，对照分析 4-2 程序，可以看出程序运行的流程吗？请简单描述：

_____

_____

_____

_____

_____

实验四 设备管理

[1] 磁盘 I/O API 函数应用

背景知识

相关的 API 介绍

1.获取磁盘的基本信息的磁盘 I/O API 函数 DeviceIoControl 格式如下：

```
BOOL DeviceIoControl( HANDLE hDevice,          DWORD dwioControlCode,
                      LPVOID lplnBuffer,       DWORD nlnBufferSize,
                      LPVOID lpOutBuffer,      DWORD nOutBufferSize,
                      LPDWORD lpBytesReturned, LPOVERLAPPED lpOverlapped  );
```

. hDevice：所要进行操作的设备的句柄，它通过调用 CreateFile 函数来获得。

. dwIoControlCode：指定操作的控制代码。这个值用来辨别将要执行的指定的操作，以及对哪一种设备进行操作。对磁盘应设置为 IOCTL_DISK_GET_DRIVE_GEOMETRY。

. lpInBuffer：操作所要的输入数据缓冲区指针，NULL 表示不需要输入数据。

. nInBufferSize：指定 lpInBuffer 所指向的缓冲区的大小（以字节为单位）。

. lpOutBuffer：接收操作输出的数据缓冲区指针，NULL 表示操作没有产生输出数据。输出数据的缓冲区要足够大，对磁盘它采用固定的数据结构 DISK_GEOMETRY，格式如下：

```
struct DISK_GEOMETRY {
  unsigned  bytesPerSector;    unsigned  sectorsPerTrack;
  unsigned  heads;             unsigned  cylinders;          }
```

. nOutBufferSize：指定 lpOutBuffer 所指向的缓冲区的大小（以字节为单位）。

. lpBytesReturned：指向一个变量，它接收 lpOutBuffer 所指的缓冲区储存的数据个数。

. lpOverlapped：指向一个 OVERLAPPED 结构。

返回值：如果函数调用成功，返回值是一个非 0 值。如果函数调用失败 GetLastError 函数来获得相关的错误信息。

2.建立文件或打开一个已存在文件 API 函数 CreateFile

该函数用来创建或打开下列对象（文件、管道、目录、邮件插口、控制台、通信资源、磁盘设备等）并返回一个用于读取该对象的句柄。

```
HANDLE CreateFile ( LPCTSTR lpFilename , DWORD  dwDesiredAccess ,
        DWORD  dwShareMode , LPSECURITY_ATTRIBUTES  lpSecurityAttributes ,
        DWORD  dwCreationDisposition ,  DWORD  dwFlagsAndAttributes ,
        HANDLE  hTemplateFile  );
```

. lpFileName：指向一个以 NULL 结束的字符串的指针，该字符串用于创建或打开对象、指定对象名。

. dwDesiredAccess：指定对对象的访问类型，一个应用程序可以得到读、写、读写或设备查询访问等类型，此参数可以为下列值的任意一个组合值：

. 0：指定对象的查询访问权限，一个应用程序可以不通过访问设备来查询设备属性。

. GENERIC_READ：指定对象的读访问，可以读文件的数据且可移动文件中的指针。

. GENERIC_WRITE：指定对象的写访问，可以写文件的数据且可以移动文件指针，写访问 GENERIC_WRITE 要与 GENERIC_READ 联合使用。

. dwShoreMode：设成 NULL 即可。

. lpSecurityAttributes：设成 NULL 即可。

. dwCreationDisposition：指定对存在的文件采取哪种措施，且当文件不存在时采用哪种措施，此函数必须是下列值中的一个：

. CREAT_NEW：创建一个新文件，如果文件存在，则函数调用失败。

. CREAT_ALWAYS：创建一个新文件，如果文件存在，函数重写文件且清空现有属性。

. OPEN_EXISTING：打开文件，如果文件不存在，则函数调用失败。

. OPEN_ALWAYS：如果文件存在，则打开文件。如果文件不存在，则创建一个新文件。

. TRUNCATE_EXISTING：打开文件，一旦文件打开，就被删截掉，从而使文件的大小为 0 字节，调用函数必须用 GENERIC_WRITE 访问来打开文件，如果文件不存在，则函数调用失败。

. dwFlagsAndAttributes：指定文件属性和标志，该参数可取很多种组合，以下示三种：

. FILE_FLAG_OVERLAPPED：指导系统对对象进行初始化，以便操作有足够的时间来处理返回 ERROR_IO_PENDING，当完成操作时，指定事件被设置为发信号状态。

. FILE_FLAG_NO_BUFFERING：引导系统打开没有瞬间缓冲或缓存的文件，当与 FILE_FLAG_OVERLAPPED 结合时，标志给出最大的按时间顺序的操作，因为 I／O 不依靠内存管理器的时间顺序的操作，但是，因为数据没有在缓存中，一些 I/O 操作将长一些。

. FILE_FLAG_SEQUENTIAL_SCAN：表明文件从开头到结尾按顺序被访问。使用它，系统可优化文件缓存。访问方式读大文件的应用程序，指定此标志可以增加它的性能。

. hTemplateFile：设成 NULL 即可。

返回值：如果函数调用成功，返回值为指向指定文件的打开句柄；如果函数调用失败，返回值为 INVALID_HANDLE_VALUE。

实验目的

本实验着重于了解磁盘的物理组织，以及如何通过用户态的程序直接调用磁盘 I/O API 函数（DeviceIoControl）根据输入的驱动器号读取驱动器中磁盘的基本信息，在 Windows Server 2003 环境进行。

实验内容与参考源代码：

SoftDiskIo-1.cpp

```
#include <windows.h>
#include <iostream.h>
#include <winioctl.h>
#include <string.h>
struct Disk        //关于 Disk 结构的定义
{
 HANDLE handle;
 DISK_GEOMETRY disk_info;
};
Disk disk;
HANDLE Floppy;
static _int64 sector;
bool flag;
Disk physicDisk(char driverLetter);

void main(void)
```

```
{
   char DriverLetter;
   cout<< "请输入磁盘号：a/c" <<endl;
   cin>>DriverLetter;                          //选择要查看的磁盘
   disk = physicDisk(DriverLetter);
}

Disk physicDisk(char driverLetter) //
{
  flag = true;
  DISK_GEOMETRY*    temp = new DISK_GEOMETRY;
  char device[9]    =    "\\\\.\\c:";
  device[4]    = driverLetter;
  Floppy = CreateFile( device,              //将要打开的驱动器名
                       GENERIC_READ,    //存取的权限
                       FILE_SHARE_READ | FILE_SHARE_WRITE,    // 共享的权限
                       NULL,                 //默认属性位
                       OPEN_EXISTING, //创建驱动器的方式
                       0,                    //所创建的驱动器的属性
                       NULL );             //指向模板文件的句柄
  if ( GetLastError() == ERROR_ALREADY_EXISTS )   //如打开失败，返回错误代码
{
   cout<<"不能打开磁盘"<<endl;
   cout<<GetLastError()<<endl;
   flag = false;
   return disk;
  }
  DWORD    bytereturned;
  BOOL Result;
  disk.handle = Floppy;
  Result = DeviceIoControl ( Floppy,
                             IOCTL_DISK_GET_DRIVE_GEOMETRY,
                             NULL,
                             0,
                             temp,
                             sizeof(*temp),
                             &bytereturned,
                             (LPOVERLAPPED)NULL );
  if (!Result)              //如果失败，返回错误代码
  {
   cout<<"打开失败”<<endl;
   cout<<"错误代码为："<<GetLastError()<<endl;
   flag = false;
   return disk;
```

```
  }
  disk.disk_info = *temp;  //输出整个物理磁盘的信息
  cout<<driverLetter<<"盘有: "<<endl;
  cout<<"柱面数为："<<(unsigned long)disk.disk_info.Cylinders.QuadPart<<endl;
  cout<< "每柱面的磁道数为："<<disk.disk_info.TracksPerCylinder<<endl;
  cout<< "每磁道的扇区数为："<<disk.disk_info.SectorsPerTrack<<endl;
  cout<< "每扇区的字节数为："<<disk.disk_info.BytesPerSector<<endl;
  sector = disk.disk_info.Cylinders.QuadPart* (disk.disk_info.TracksPerCylinder)*
(disk.disk_info.SectorsPerTrack);
  double DiskSize =(double)disk.disk_info.Cylinders.QuadPart * //
                                    (disk.disk_info.TracksPerCylinder) *
                                    (disk.disk_info.SectorsPerTrack) *
                                    (disk.disk_info.BytesPerSector);
  cout<<driverLetter<<"盘所在磁盘总共有"<<(long)sector<<"个扇区"<<endl;
  cout<<"磁盘大为:"<<DiskSize/(1024*1024)<<"MB "<<endl;
  delete temp;
  return disk;
}
```

程序的结果

```
请输入磁盘号：a/c
a
a 盘有:
柱面数为：80
每柱面的磁道数为：2
每磁道的扇区数为：18
每扇区的字节数为：512
a 盘所在磁盘总共有 2880 个扇区
磁盘大为:1.40625MB
```

讨论

如输入磁盘号为 c，显示的磁盘信息是整个硬盘信息，而不是 c 盘分区的信息。如输入磁盘号为 d，显示的磁盘信息与如输入磁盘号为 c 显示的磁盘信息相同。用磁盘 I/O API 函数读出的磁盘信息是从硬盘的主引导区得到。

附录 实验报告参考规范

使用学院统一的实验报告封面并正确给出课程名称、课程号、专业、班级、学号、姓名和完成日期。

报告内容及格式如下：

1．实验目的

给出本实验要求达到的目的。

2．实验内容

给出本实验要求完成的实验任务。

3　实验步骤

（1） 任务分析：以无歧义的陈述说明实验任务，强调的是要做什么？并明确规定：

(1) 输入的形式和输入值的范围；

(2) 输出的形式；

(3) 程序所能达到的功能；

(4) 测试数据：包括正确的输入及其输出结果和含有错误的输入及其输出结果。

（2） 概要设计：说明本程序中用到的所有抽象数据类型的定义、主程序的流程以及各程序模块之间的层次(调用)关系。

（3） 详细设计

实现概要设计中定义的所有数据类型，对每个操作只需要写出伪码算法；对主程序和其他模块也都需要写出伪码算法(伪码算法达到的详细程度建议为：按照伪码算法可以在计算机键盘直接输入高级程序设计语言程序)；画出函数和过程的调用关系图。

（4） 调试分析：

a．调试过程中遇到的问题是如何解决的以及对设计与实现的回顾讨论和分析；

b．算法的时空分析(包括基本操作和其他算法的时间复杂度和空间复杂度的分析)和改进设想；

c．经验和体会等。

（5） 测试结果：列出你的测试结果，包括输入和输出。这里的测试数据应该完整和严格，最好多于需求分析中所列。

（6） 使用说明：说明如何使用你编写的程序，详细列出每一步的操作步骤。

4．实验总结

5．附录

带注释的程序清单。