

操作系统实验报告

学 号 : 20 级
姓 名 : Banban
专 业 : 计算机科学与技术
班 级 :
指 导 老 师 :
日 期 : 2022 年 10 月 21 日

目录

实验一 WINDOWS 进程初识	3
1、 实验目的	3
2、 实验内容和步骤	3
3、 实验结论	5
4、 程序清单	错误!未定义书签。
实验二 进程管理	5
背景知识	错误!未定义书签。
1、 实验目的	7
2、 实验内容和步骤	7
3、 实验结论	12
4、 程序清单	错误!未定义书签。
实验三 进程同步的经典算法	13
背景知识	错误!未定义书签。
1、 实验目的	13
2、 实验内容和步骤	13
3、 实验结论	15
4、 程序清单	错误!未定义书签。
实验四 存储管理	18
背景知识	错误!未定义书签。
1、 实验目的	18
2、 实验内容和步骤	19
3、 实验结论	22
4、 程序清单	错误!未定义书签。
实验五 文件系统设计试验	23
1、 试验目的	23
2、 实验内容与步骤	23
3、 实验结论	23
4、 对试验的改进以及效果	错误!未定义书签。
附录 A: 参考程序	23
附录 B: 文件系统模拟程序	27

实验一 WINDOWS 进程初识

1. 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. 掌握 WINDOWS API 的使用方法。
3. 编写测试程序，理解用户态运行和核心态运行。

2. 实验内容和步骤

1. 编写基本的 Win32 Consol Application

步骤1: 登录进入 Windows，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。

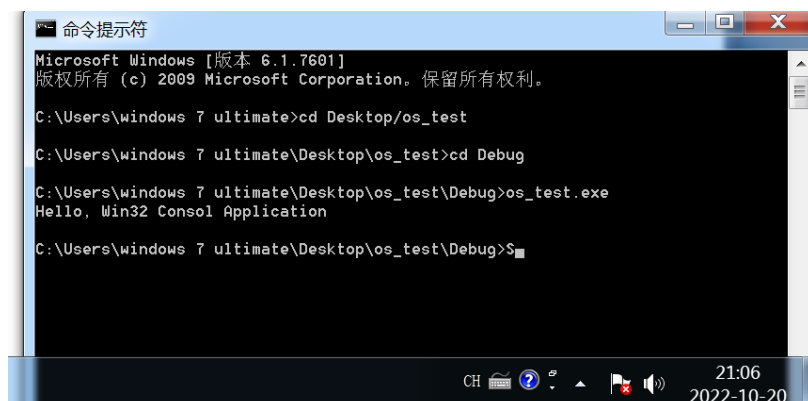
步骤3: 在“FILE”菜单中单击“NEW”子菜单，在“Files”选项卡中选择“C++ Source File”，然后在“File”处输入 C/C++源程序的文件名。

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：E:\课程\os 课\os 实验\程序\os11\debug>hello.exe

运行结果（如果运行不成功，则可能的原因是什么？）：

运行成功，运行结果如下：



```
命令提示符
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\windows 7 ultimate>cd Desktop/os_test
C:\Users\windows 7 ultimate\Desktop\os_test>cd Debug
C:\Users\windows 7 ultimate\Desktop\os_test\Debug>os_test.exe
Hello, Win32 Consol Application
C:\Users\windows 7 ultimate\Desktop\os_test\Debug>S
```

2. 计算进程在核心态运行和用户态运行的时间

步骤1: 按照（1）中的步骤创建一个新的“Win32 Consol Application”工程，然后将清

单 1-2 中的程序拷贝过来，编译成可执行文件。

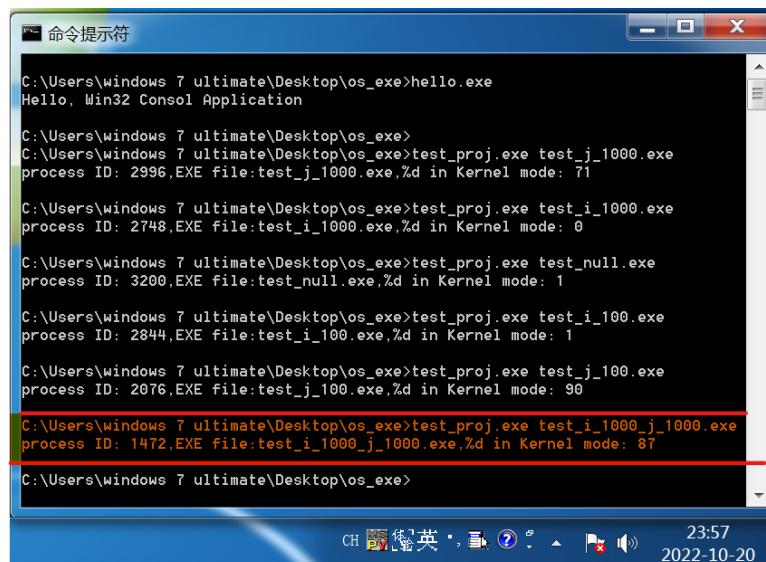
步骤2: 在创建一个新的“Win32 Console Application”工程，程序的参考程序如清单 1-3 所示，编译成可执行文件并执行。

步骤3: 在“命令提示符”窗口中运行步骤 1 中生成的可执行文件，测试步骤 2 中可执行文件在核心态运行和用户态运行的时间。

E:\课程\os 课\os 实验\程序\os12\debug>time TEST.exe

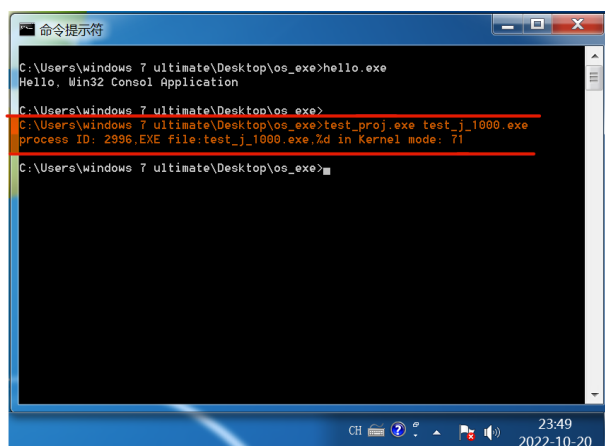
步骤4: 运行结果（如果运行不成功，则可能的原因是什么？）：

运行成功，运行结果如下：

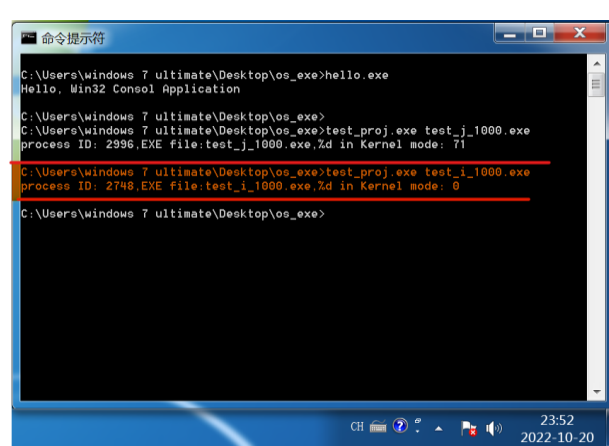


步骤5: 步骤 5：分别屏蔽 While 循环中的两个 for 循环，或调整两个 for 循环的次数，写出运行结果。

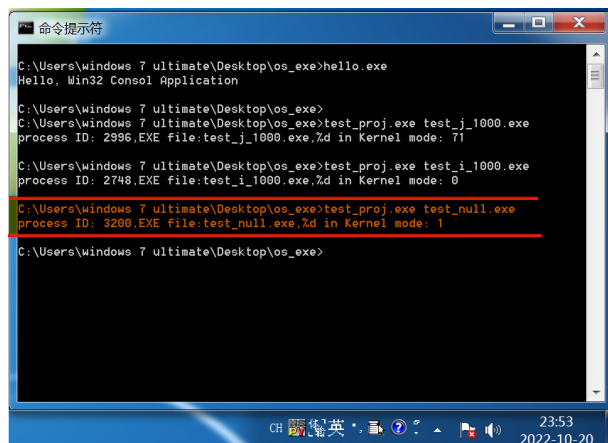
屏蔽 i 循环：



屏蔽 j 循环：



屏蔽 i, j 循环:



```
C:\Users\windows 7 ultimate\Desktop\os_exe>hello.exe
Hello, Win32 Console Application

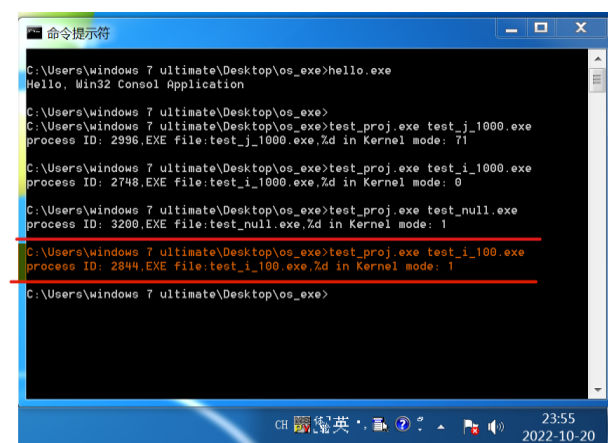
C:\Users\windows 7 ultimate\Desktop\os_exe>
C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_j_1000.exe
process ID: 2996, EXE file: test_j_1000.exe, %d in Kernel mode: 71

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_1000.exe
process ID: 2748, EXE file: test_i_1000.exe, %d in Kernel mode: 0

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_null.exe
process ID: 3200, EXE file: test_null.exe, %d in Kernel mode: 1

C:\Users\windows 7 ultimate\Desktop\os_exe>
```

调整循环变量 i<=100 的循环次数, 屏蔽 j:



```
C:\Users\windows 7 ultimate\Desktop\os_exe>hello.exe
Hello, Win32 Console Application

C:\Users\windows 7 ultimate\Desktop\os_exe>
C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_j_1000.exe
process ID: 2996, EXE file: test_j_1000.exe, %d in Kernel mode: 71

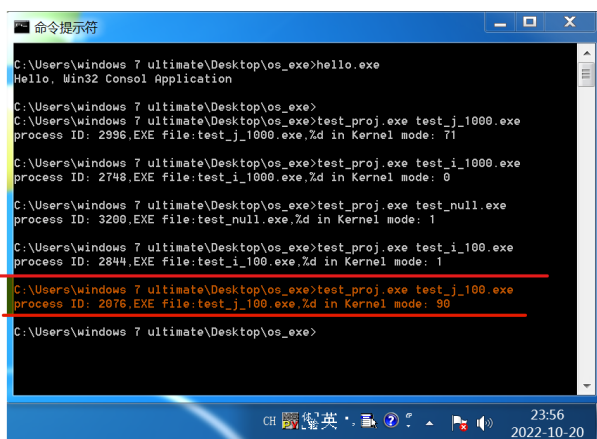
C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_1000.exe
process ID: 2748, EXE file: test_i_1000.exe, %d in Kernel mode: 0

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_null.exe
process ID: 3200, EXE file: test_null.exe, %d in Kernel mode: 1

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_100.exe
process ID: 2844, EXE file: test_i_100.exe, %d in Kernel mode: 1

C:\Users\windows 7 ultimate\Desktop\os_exe>
```

调整循环变量 j<=100 的循环次数, 屏蔽 i:



```
C:\Users\windows 7 ultimate\Desktop\os_exe>hello.exe
Hello, Win32 Console Application

C:\Users\windows 7 ultimate\Desktop\os_exe>
C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_j_1000.exe
process ID: 2996, EXE file: test_j_1000.exe, %d in Kernel mode: 71

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_1000.exe
process ID: 2748, EXE file: test_i_1000.exe, %d in Kernel mode: 0

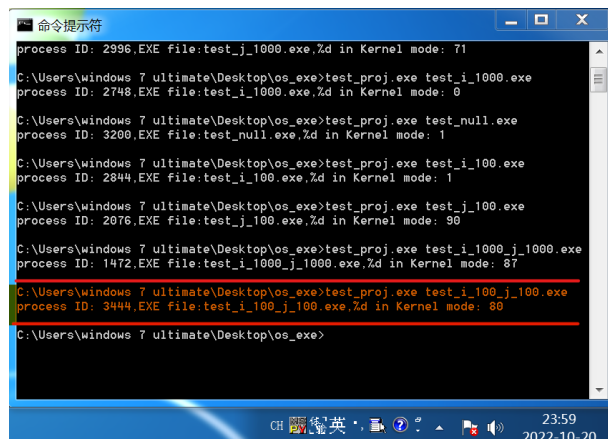
C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_null.exe
process ID: 3200, EXE file: test_null.exe, %d in Kernel mode: 1

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_100.exe
process ID: 2844, EXE file: test_i_100.exe, %d in Kernel mode: 1

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_j_100.exe
process ID: 2076, EXE file: test_j_100.exe, %d in Kernel mode: 90

C:\Users\windows 7 ultimate\Desktop\os_exe>
```

调整循环变量 i<=100, j<=100 的循环次数:



```
process ID: 2996, EXE file: test_j_1000.exe, %d in Kernel mode: 71

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_1000.exe
process ID: 2748, EXE file: test_i_1000.exe, %d in Kernel mode: 0

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_null.exe
process ID: 3200, EXE file: test_null.exe, %d in Kernel mode: 1

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_100.exe
process ID: 2844, EXE file: test_i_100.exe, %d in Kernel mode: 1

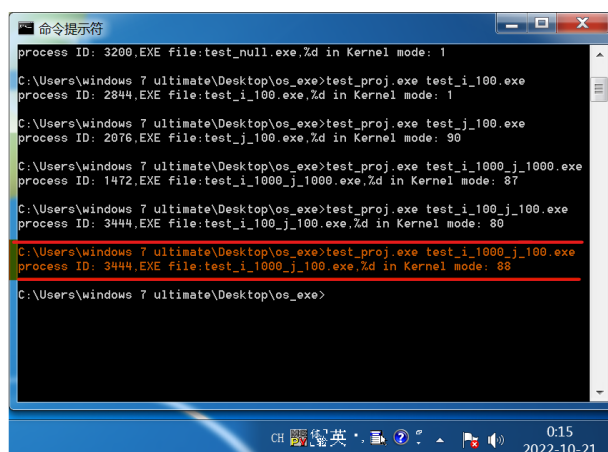
C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_j_100.exe
process ID: 2076, EXE file: test_j_100.exe, %d in Kernel mode: 90

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_1000_j_1000.exe
process ID: 1472, EXE file: test_i_1000_j_1000.exe, %d in Kernel mode: 87

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_100_j_100.exe
process ID: 3444, EXE file: test_i_100_j_100.exe, %d in Kernel mode: 80

C:\Users\windows 7 ultimate\Desktop\os_exe>
```

调整循环变量 i<=1000, j<=100 的循环次数:



```
process ID: 3200, EXE file: test_null.exe, %d in Kernel mode: 1

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_100.exe
process ID: 2844, EXE file: test_i_100.exe, %d in Kernel mode: 1

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_j_100.exe
process ID: 2076, EXE file: test_j_100.exe, %d in Kernel mode: 90

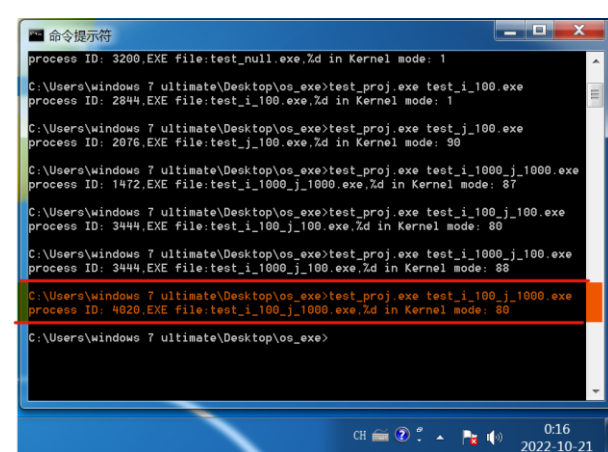
C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_1000_j_1000.exe
process ID: 1472, EXE file: test_i_1000_j_1000.exe, %d in Kernel mode: 87

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_100_j_100.exe
process ID: 3444, EXE file: test_i_100_j_100.exe, %d in Kernel mode: 80

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_1000_j_100.exe
process ID: 3444, EXE file: test_i_1000_j_100.exe, %d in Kernel mode: 88

C:\Users\windows 7 ultimate\Desktop\os_exe>
```

调整循环变量 i<=100, j<=1000 的循环次数:



```
process ID: 3200, EXE file: test_null.exe, %d in Kernel mode: 1

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_100.exe
process ID: 2844, EXE file: test_i_100.exe, %d in Kernel mode: 1

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_j_100.exe
process ID: 2076, EXE file: test_j_100.exe, %d in Kernel mode: 90

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_1000_j_1000.exe
process ID: 1472, EXE file: test_i_1000_j_1000.exe, %d in Kernel mode: 87

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_100_j_100.exe
process ID: 3444, EXE file: test_i_100_j_100.exe, %d in Kernel mode: 80

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_1000_j_100.exe
process ID: 3444, EXE file: test_i_1000_j_100.exe, %d in Kernel mode: 88

C:\Users\windows 7 ultimate\Desktop\os_exe>test_proj.exe test_i_100_j_1000.exe
process ID: 4020, EXE file: test_i_100_j_1000.exe, %d in Kernel mode: 80

C:\Users\windows 7 ultimate\Desktop\os_exe>
```

3. 实验结论

程序 1-2 中不屏蔽 i 和 j, 执行时会调用 printf 函数, 程序要进入核心态, 内核运行

占用的时间多，占用百分比大概 **87%**；而当把 **j** 循环屏蔽掉，即不执行 **printf** 时，**i** 循环只需在用户态运行，不需要进入核心态，因此内核占用时间就为 **0%**。

当改变 **i** 循环的次数，而 **j** 不变时，进入核心态时所占用的时间不变，所以运行时间没有改变；当改变 **j** 循环次数，而 **i** 不变时，**printf** 的次数改变，因此在核心态运行的时间就相对改变，运行时间随 **j** 的增大而增大；随 **j** 的减小而减小，成正比关系；

实验二 进程管理

1. 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、终止进程以及父子进程同步的基本程序设计方法。

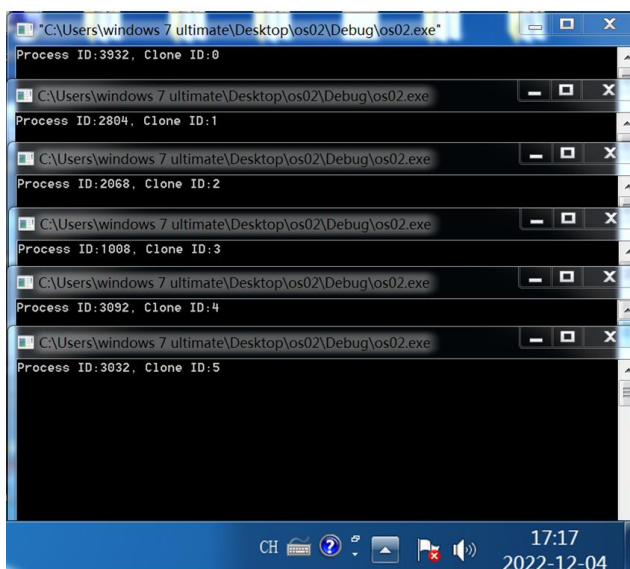
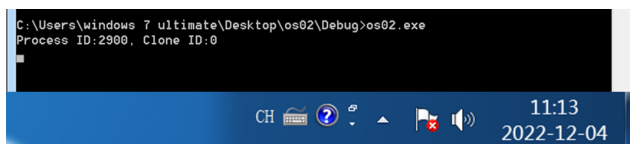
2. 实验内容和步骤

1. 创建进程

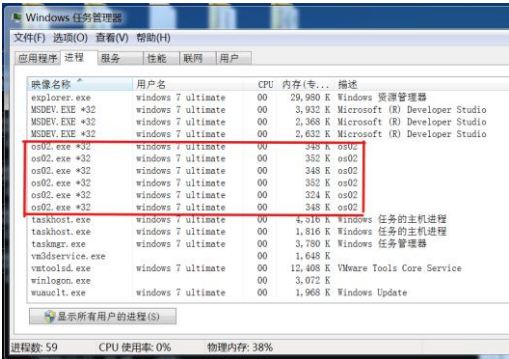
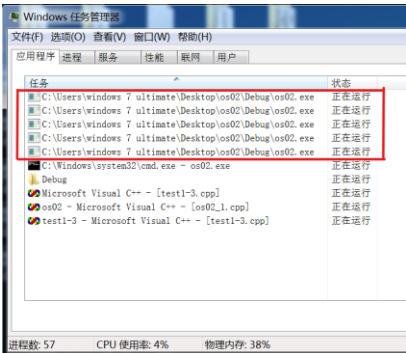
本实验显示了创建子进程的基本框架。该程序只是再一次地启动自身，显示它的系统进程 ID 和它在进程列表中的位置。

步骤1: 创建一个“Win32 Consol Application”工程，然后拷贝清单 2-1 中的程序，编译成可执行文件。

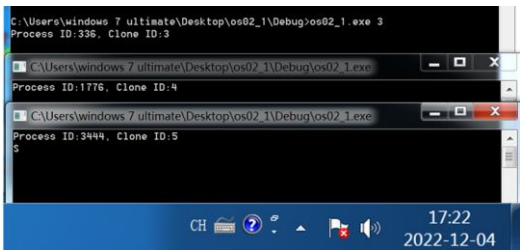
步骤2: 在“命令提示符”窗口运行步骤 1 中生成的可执行文件。运行结果：范例：E:\课程\os 课\os 实验\程序\os11\debug>os21（假设编译生成的可执行文件是 os21.exe）



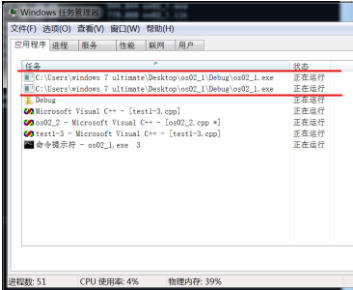
按下 **ctrl+alt+del**，调用 windows 的任务管理器，记录进程相关的行为属性：

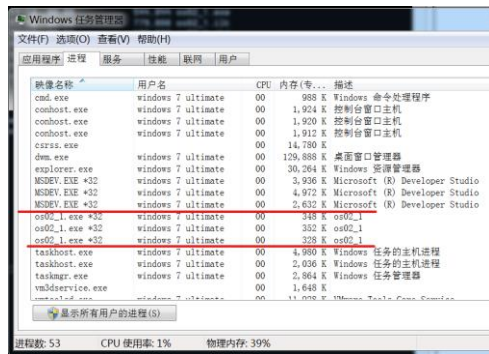


步骤3: 在“命令提示符”窗口加入参数重新运行生成的可执行文件。运行结果：范例：E:\课程\os 课\os 实验\程序\os11\debug>os21 3(假设编译生成的可执行文件是 os21.exe)



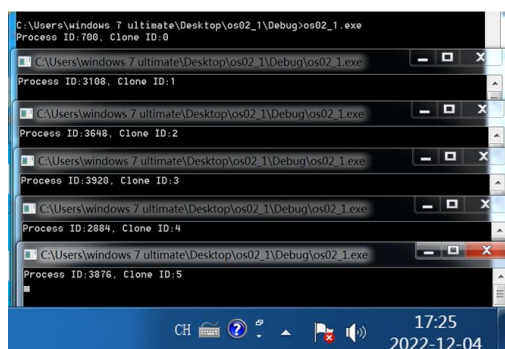
按下 **ctrl+alt+del**，调用 windows 的任务管理器，记录进程相关的行为属性：



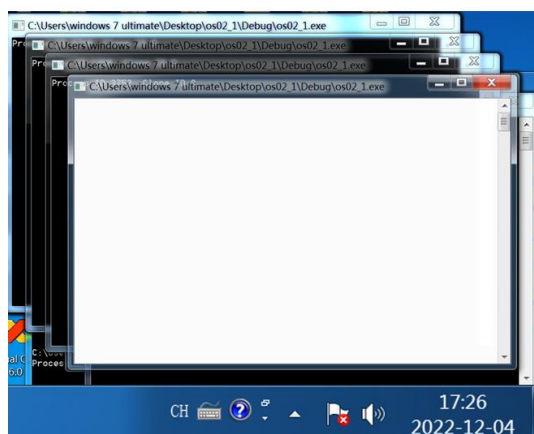


步骤4: 修改清单 2-1 中的程序, 将 nClone 的定义和初始化方法按程序注释中的修改方法进行修 改, 编译成可执行文件 (执行前请先保存已经完成的工作)。再按步骤 2 中的方式运行, 看看结果会有什么不一样。运行结果:

第一次: 正常运行



第二次: 死循环, 不断产生线程。。。



从中你可以得出什么结论:

nClone 的作用: 设置进程的起始 ID 号。

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?

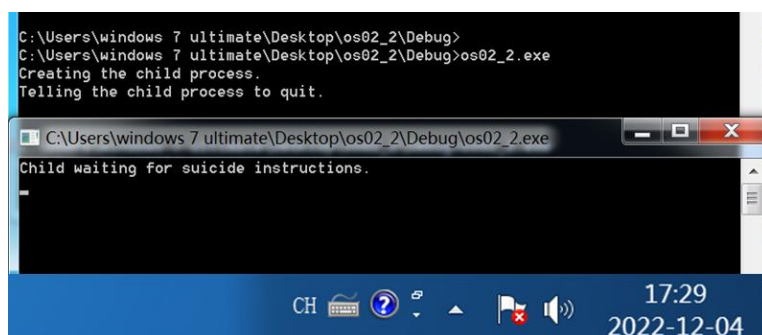
有影响, 因为变量的定义和初始化的不同可能导致进程的克隆的次数及克隆进程的 ID 号不

同

2. 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 2-2 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。运行结果：范例：
E:\课程\os 课\os 实验\程序\os11\debug>os22 (假设编译生成的可执行文件是 os22.exe)

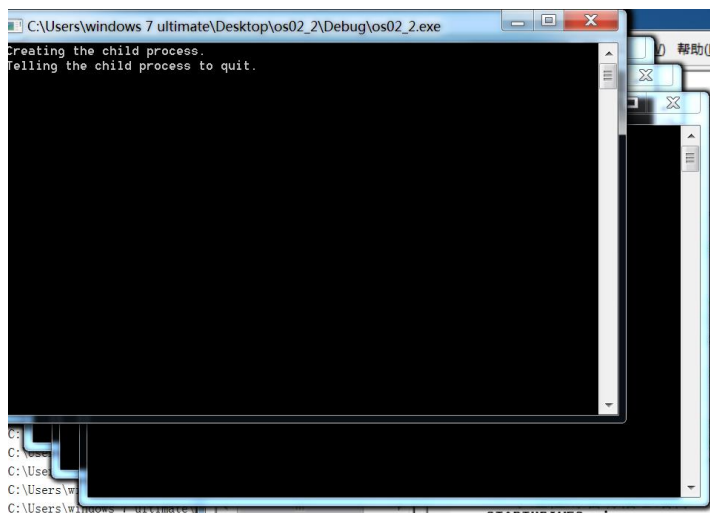


```
C:\Users\windows 7 ultimate\Desktop\os02_2\Debug>
C:\Users\windows 7 ultimate\Desktop\os02_2\Debug>os02_2.exe
Creating the child process.
Telling the child process to quit.
```

```
C:\Users\windows 7 ultimate\Desktop\os02_2\Debug\os02_2.exe
Child waiting for suicide instructions.
```

步骤3: 按源程序中注释中的提示，修改源程序 2-2，编译执行（执行前请先保存已经完成的工作）。运行结果：

现象：死循环，不断产生子进程。。。



步骤4: 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：

通过 **main(int argc, char* argv[])** 传递参数，每次运行时先检测 **argc** 的值，若小于 1，程序运行结束，否则继续往下执行。

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：

```

BOOL CreateProcess(
    LPCTSTR lpApplicationName,           // pointer to name of executable module
    LPTSTR lpCommandLine,               // pointer to command line string
    LPSECURITY_ATTRIBUTES lpProcessAttributes, // pointer to process security attributes
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // pointer to thread security attributes
    BOOL bInheritHandles,               // handle inheritance flag
    DWORD dwCreationFlags,              // creation flags
    LPVOID lpEnvironment,               // pointer to new environment block
    LPCTSTR lpCurrentDirectory,         // pointer to current directory name
    LPSTARTUPINFO lpStartupInfo,        // pointer to STARTUPINFO
    LPPROCESS_INFORMATION lpProcessInformation // pointer to PROCESS_INFORMATION
);

```

1. **lpApplicationName**: 指向一个 NULL 结尾的、用来指定可执行模块的字符串。这个字符串可以是可执行模块的绝对路径，也可以是相对路径，在后一种情况下，函数使用当前驱动器和目录建立可执行模块的路径。这个参数可以被设为 NULL，在这种情况下，可执行模块的名字必须处于 **lpCommandLine** 参数最前面并由空格符与后面的字符分开。
2. **lpCommandLine**: 指向一个以 NULL 结尾的字符串，该字符串指定要执行的命令行。这个参数可以为空，那么函数将使用 **lpApplicationName** 参数指定的字符串当做要运行的程序的命令行。
3. **lpProcessAttributes**: 指向一个 SECURITY_ATTRIBUTES 结构体，这个结构体决定是否返回的句柄可以被子进程继承。如果 **lpProcessAttributes** 参数为空（NULL），那么句柄不能被继承。
4. **lpThreadAttributes**: 同 **lpProcessAttribute**，不过这个参数决定的是线程是否被继承。通常置为 NULL。
5. **bInheritHandles**: 指示新进程是否从调用进程处继承了句柄。如果参数的值为真，调用进程中的每一个可继承的打开句柄都将被子进程继承。被继承的句柄与原进程拥有完全相同的值和访问权限。
6. **dwCreationFlags**: 指定附加的、用来控制优先类和进程的创建的标志。以下的创建标志可以以除下面列出的方式外的任何方式组合后指定。

步骤5： 填空

CreateProcess() 函数有____5____个核心参数？本实验程序中设置的各个参数的值是：

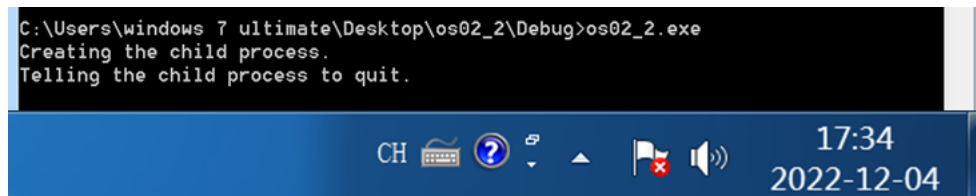
```

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
);

```

按源程序中注释中的提示，修改源程序 2-2，编译执行。运行结果：

一闪而过。。。



```
C:\Users\windows 7 ultimate\Desktop\os02_2\Debug>os02_2.exe
Creating the child process.
Telling the child process to quit.
```

步骤6： 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：

`CreateMutex()`创建互斥体，`OpenMutex()`打开互斥体，`ReleaseMutex()`释放互斥体，`WaitForSingleObject()`检测 `hHandle` 事件的信号状态，通过这些方法可实现当前只有一个进程被创建或使用，实现进程的同步。

3. 实验结论

通过对进程的操作，如创建进程，实现对进程的简单控制。创建互斥体，解决了进程的同步问题，两者相互使用，使进程的运行情况得到了很好的管理。

通过本实验的进行，对 Windows 创建进程，以及父子进程之间的同步互斥有了切身的体会，从中感受到学习操作系统知识的乐趣。另外，通过本次加深了对信号量对于进程间对临界区访问的同步与互斥的理解。

实验三 进程同步的经典算法

1. 实验目的

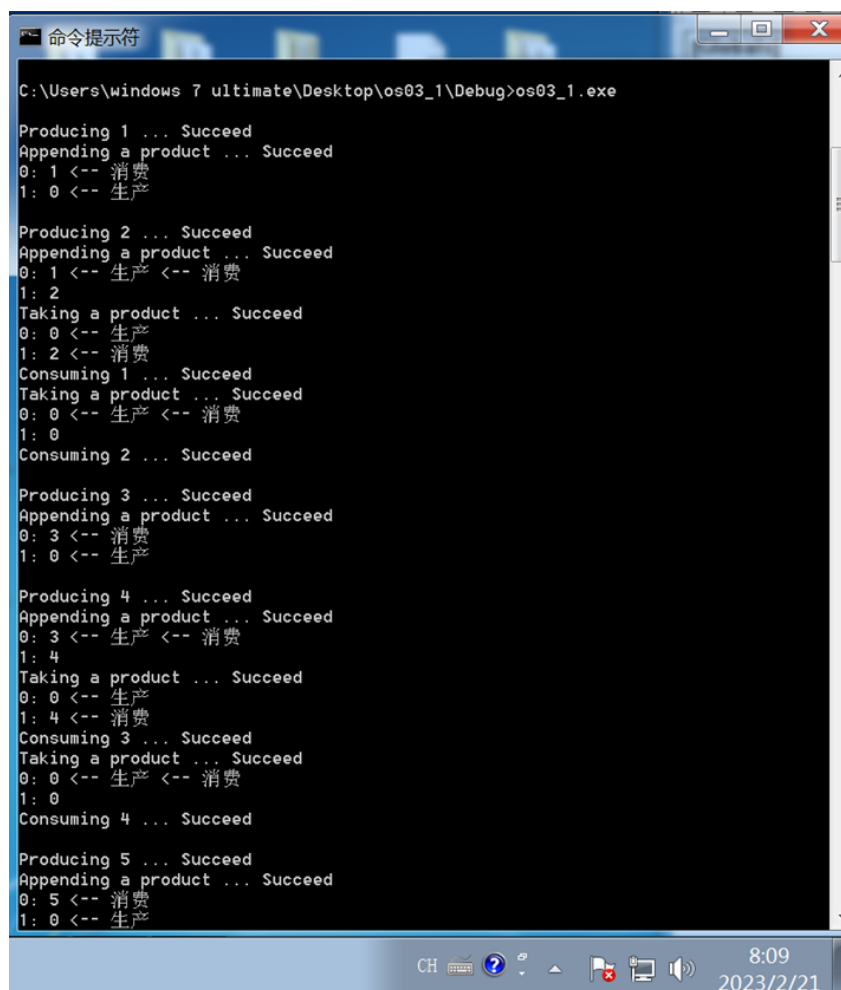
1. 回顾系统进程、线程的有关概念，加深对 Windows 2000 线程的理解。
2. 了解互斥体对象，通过对生产者消费者等进程间同步与互斥经典算法的实现，加深对 P、V 原语以及利用 P、V 原语进行进程间同步与互斥操作的理解。

2. 实验内容和步骤

1. 生产者消费者问题

步骤1: 创建一个“Win32 Consol Application”工程，然后拷贝清单 3-1 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。运行结果：范例：E:\课程\os 课\os 实验\程序\os11\debug>os31（假设编译生成的可执行文件是 os31.exe）



```
C:\Users\windows 7 ultimate\Desktop\os03_1\Debug>os03_1.exe

Producing 1 ... Succeed
Appending a product ... Succeed
0: 1 <-- 消费
1: 0 <-- 生产

Producing 2 ... Succeed
Appending a product ... Succeed
0: 1 <-- 生产 <-- 消费
1: 2

Taking a product ... Succeed
0: 0 <-- 生产
1: 2 <-- 消费
Consuming 1 ... Succeed
Taking a product ... Succeed
0: 0 <-- 生产 <-- 消费
1: 0
Consuming 2 ... Succeed

Producing 3 ... Succeed
Appending a product ... Succeed
0: 3 <-- 消费
1: 0 <-- 生产

Producing 4 ... Succeed
Appending a product ... Succeed
0: 3 <-- 生产 <-- 消费
1: 4
Taking a product ... Succeed
0: 0 <-- 生产
1: 4 <-- 消费
Consuming 3 ... Succeed
Taking a product ... Succeed
0: 0 <-- 生产 <-- 消费
1: 0
Consuming 4 ... Succeed

Producing 5 ... Succeed
Appending a product ... Succeed
0: 5 <-- 消费
1: 0 <-- 生产
```

仔细阅读源程序，找出创建线程的 WINDOWS API 函数，回答下列问题：线程的第一个执行函数是什么（从哪里开始执行）？它位于创建线程的 API 函数的第几个参数中？

答：第一个执行函数是 **Producer**；位于第三/一个参数中。

步骤3： 修改清单 3-1 中的程序，调整生产者线程和消费者线程的个数，使得消费者数目大与生产者，看看结果有何不同。运行结果：



```
C:\Users\windows 7 ultimate\Desktop\os03_1\Debug\os03_1.exe
Producing 1 ... Succeed
Appending a product ... Succeed
0: 1 <-- 消费
1: 0 <-- 生产

Producing 2 ... Succeed
Appending a product ... Succeed
0: 1 <-- 生产 <-- 消费
1: 2
Taking a product ... Succeed
0: 0 <-- 生产
1: 2 <-- 消费
Consuming 1 ... Succeed
Taking a product ... Succeed
0: 0 <-- 生产 <-- 消费
1: 0
Consuming 2 ... Succeed

Producing 3 ... Succeed
Appending a product ... Succeed
0: 3 <-- 消费
1: 0 <-- 生产
Taking a product ... Succeed
0: 0
1: 0 <-- 生产 <-- 消费
Consuming 3 ... Succeed

Producing 4 ... Succeed
Appending a product ... Succeed
0: 0 <-- 生产
1: 4 <-- 消费

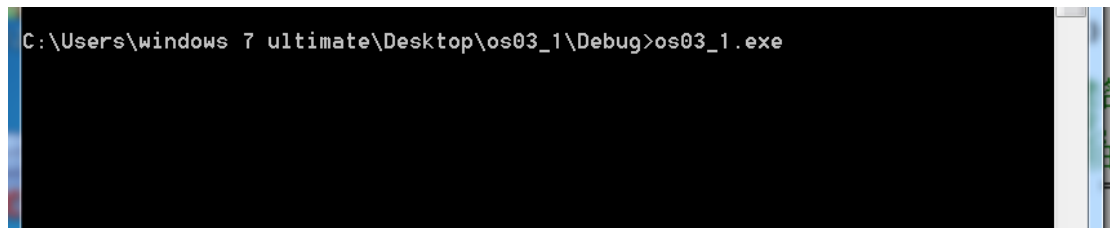
Producing 5 ... Succeed
Appending a product ... Succeed
0: 5
1: 4 <-- 生产 <-- 消费
Taking a product ... Succeed
0: 5 <-- 消费
1: 0 <-- 生产
Consuming 4 ... Succeed
```

从中你可以得出什么结论：

生产速度快，生产者经常等待消费者；反之，消费者经常等待。

生产者多于消费者时,生产者等待时间多一些,消费者多于生产者时,消费者等待时间多一些。

步骤4： 修改清单 3-1 中的程序，按程序注释中的说明修改信号量 EmptySemaphore 的初始化方法，看看结果有何不同。运行结果：



步骤5: 根据步骤 4 的结果, 并查看 MSDN, 回答下列问题

1) CreateMutex 中有几个参数, 各代表什么含义。

```
HANDLE CreateMutex(  
    LPSECURITY_ATTRIBUTES lpMutexAttributes, // 指向安全属性的指针  
    BOOL bInitialOwner, // 初始化互斥对象的所有者  
    LPCTSTR lpName // 指向互斥对象名的指针  
);
```

2) CreateSemaphore 中有几个参数, 各代表什么含义, 信号量的初值在第几个参数中。

```
HANDLE CreateSemaphore (  
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // 信号量属性, 可为 NULL  
    LONG lInitialCount, // 信号量初始值, 必须>=0, 而且<=lpMaximumCount  
    LONG lMaximumCount, // 信号量的最大值, 必须大于 0  
    LPCTSTR lpName // 信号量名字, 长度<MAX_PATH, 可为 NULL, 表示无名信号量  
);
```

3) 程序中 P、V 原语所对应的实际 Windows API 函数是什么, 写出这几条语句。

```
WaitForSingleObject(EmptySemaphore, INFINITE); //p(empty);  
WaitForSingleObject(Mutex, INFINITE); //p(mutex);  
ReleaseMutex(Mutex); //V(mutex);  
ReleaseSemaphore(FullSemaphore, 1, NULL); //V(full);
```

4) CreateMutex 能用 CreateSemaphore 替代吗? 尝试修改程序 3-1, 将信号量 Mutex 完全用 CreateSemaphore 及相关函数实现。写出要修改的语句:

```
HANDLE Mutex; -> HANDLE Semaphore;  
Mutex=CreateMutex(NULL, FALSE, NULL); -> Semaphore=CreateSemaphore(NULL, 1, 1, NULL);  
WaitForSingleObject(Mutex, INFINITE); -> WaitForSingleObject(Semaphore, INFINITE);  
ReleaseMutex(Mutex); -> ReleaseSemaphore(Semaphore, 1, NULL);
```

2. 读者写者问题

根据实验 (1) 中所熟悉的 P、V 原语对应的实际 Windows API 函数, 并参考教材中读者、写者问题的算法原理, 尝试利用 Windows API 函数实现第一类读者写者问题 (读者优先, 写出源码)。

```
#include <windows.h>  
#include <iostream>  
#include <winbase.h>  
#include <stdlib.h>  
  
using namespace std;
```

```

unsigned short ReaderID = 0;    //读者编号
unsigned short WriterID = 0;    //写者编号

int nReaders = 0;              // 当前读者数
bool p_continue = true;        // 控制程序结束

// 定义互斥量和信号量
HANDLE mutex, wMutex;
DWORD WINAPI Reader(LPVOID);    //读者线程
DWORD WINAPI Writer(LPVOID);    //写者线程

// 延时时间
int T_R = 0;
int T_W = 0;
int time_R[5] = {1,2,6,7,8};
float time_W[2] = {2,4.5};

double start;
double end;

int main() {
    srand(time(NULL));

    // 初始化互斥量和读写锁
    mutex = CreateMutex(NULL, FALSE, NULL);
    wMutex = CreateSemaphore(NULL, 1, 1, NULL);

    const unsigned short READER_COUNT = 5; //读者的个数
    const unsigned short WRITER_COUNT = 2; //写者的个数

    //总的线程数
    const unsigned short THREADS_COUNT = READER_COUNT+WRITER_COUNT;

    HANDLE hThreads[THREADS_COUNT]; //各线程的 handle
    DWORD readerID[READER_COUNT]; //读者的标识符
    DWORD writerID[WRITER_COUNT]; //写者线程的标识符

    start = GetTickCount();

    // 创建读者线程
    for (int i = 0; i < READER_COUNT; i++) {
        hThreads[i]=CreateThread(NULL,0,Reader,NULL,0,&readerID[i]);
        if (hThreads[i]==NULL) return -1;
    }

    // 创建写者线程
    for (int j = 0; j < WRITER_COUNT; j++) {
        hThreads[READER_COUNT+j]=CreateThread(NULL,0,Writer,NULL,0,&writerID[j]);
        if (hThreads[j]==NULL) return -1;
    }

    while(p_continue){
        if(getchar()){ //按回车后终止程序运行
            p_continue = false;
        }
    }

    return 0;
}

void Read()

```



```

{
    int a = rand()%500+500; // 0.5-1
    std::cout << "READ..STAR.." << a << ".." << GetTickCount() - start << ".." <<
    ++ReaderID << std::endl;
    Sleep(a);
    std::cout << "READ..END.." << ReaderID << ".." << GetTickCount() - start << std::endl;
}

void Write()
{
    int a = rand()%1000+1000; // 1-2
    std::cout << "WRITER..STAR.." << a << ".." << GetTickCount() - start << ".." <<
    ++WriterID << std::endl;
    Sleep(a);
    std::cout << "WRITER..END.." << WriterID << ".." << GetTickCount() - start <<
    std::endl;
}

// 定义读者线程函数
DWORD WINAPI Reader(LPVOID lpParam) {
    while (p_ccontinue) {

        if(T_R==5) T_R=0;
        Sleep(time_R[T_R++]*1000);

        WaitForSingleObject(mutex, INFINITE);
        if (nReaders == 0) WaitForSingleObject(wMutex, INFINITE);
        nReaders++;
        ReleaseMutex(mutex);

        Read();

        WaitForSingleObject(mutex, INFINITE);
        nReaders--;
        if (nReaders == 0) ReleaseSemaphore(wMutex, 1, NULL);
        ReleaseMutex(mutex);

    }

    return 0;
}

// 定义写者线程函数
DWORD WINAPI Writer(LPVOID lpParam) {
    while (p_ccontinue) {
        if(T_W==2) T_W=0;
        Sleep(time_W[T_W++]*1000);
        WaitForSingleObject(wMutex, INFINITE);
        Write();
        ReleaseSemaphore(wMutex, 1, NULL);
    }

    return 0;
}

```

```
C:\Users\xulon\Desktop\操作
READ..STAR..541..1016..1
READ..END..1..1563
WRITER..STAR..1041..2016..1
WRITER..END..1..3063
READ..STAR..541..3063..2
READ..STAR..967..3063..3
READ..END..3..3610
READ..END..3..4032
WRITER..STAR..1041..4516..2
WRITER..END..2..5578
WRITER..STAR..1467..5578..3
WRITER..END..3..7063
READ..STAR..967..READ..STAR..541..7063..4
READ..STAR..541..7063..5
7063..6
READ..END..6..7625
READ..END..6..7625
READ..STAR..541..8016..7
READ..END..7..8047
READ..END..7..8563
READ..STAR..834..9047..8
READ..END..8..9891
WRITER..STAR..1334..9891..4
WRITER..END..4..11235
READ..STAR..834..11235..9
READ..STAR..967..11235..10
READ..END..10..12078
READ..END..10..12219
```

3. 实验结论

通过本次实验的进行，自己加深了对 **p、v** 操作控制进程访问临界资源的理解。同时，熟悉了 **windows** 系统关于进程同步与互斥的控制。

将信号量看作生产或消费的一个对象，对信号量的生成和销毁操作如同 **P** 操作和 **V** 操作一样，生成者消费者问题模拟的就是对信号量的生成和销毁，其中牵涉了信号量的同步，这也是该问题为何称为同步的经典问题的原因。

实验四 存储管理

1. 实验目的

1. 掌握 OPT、FIFO、LRU 等页面置换算法；
2. 理解分页存储管理的内存分配与布局；
3. 理解页面置换算法对缺页性能的影响；

2. 实验内容和步骤

本实验通过编写 Windows 控制台应用程序的方式来模拟实现页面置换算法。在模拟中，体现了内存的布局与分配，并通过对不同置换算法的对比，加深对页面置换算法影响缺页率的理解。

步骤1: 准备实验，从实验模板获取实验资料。（URL 为 <https://www.codecode.net/engintime/os-lab/Project-Template/page-replace-algorithm.git>）

步骤2: 查看最佳页面置换算法(OPT)和先进先出页面置换算法(FIFO)的执行过程

在新建的项目中，仔细阅读文件中的源代码和注释，查看 OPT 和 FIFO 页面置换算法是如何实现的。

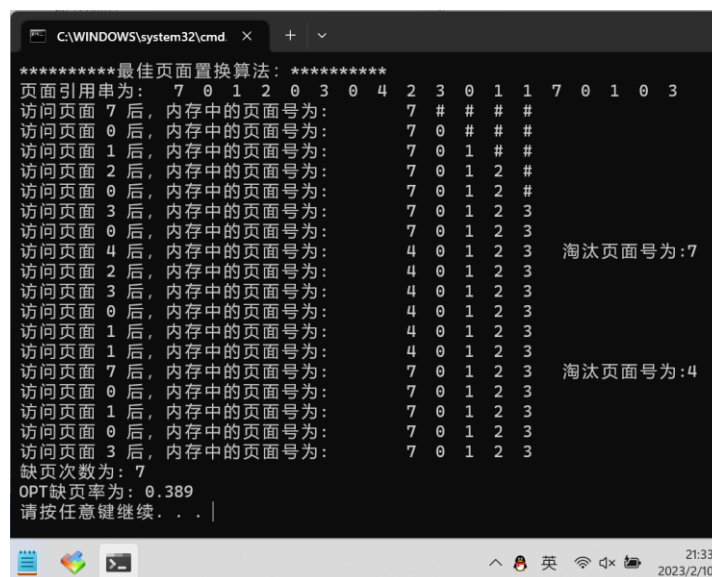
其中，在 main 函数中定义了一个数组 PageNumofR[]来存放页面号引用串，定义了一个指针*BlockofMemory 指向一块存放页面号的内存块。依次将数组 PageNumofR[]中的页面装入内存，根据不同的页面置换算法淘汰内存中的页面。

按照下面的步骤查看 OPT 和 FIFO 的执行过程：

1) 按 F7 键生成修改后的控制台应用程序项目，确保没有语法上的错误。

2) 按 Ctrl+F5 执行此程序，查看 OPT 和 FIFO 的执行过程，其中，“#”表示未引用的内存块。**执行结果如下所示：**

OPT



```
*****最佳页面置换算法：*****
页面引用串为： 7 0 1 2 0 3 0 4 2 3 0 1 1 7 0 1 0 3
访问页面 7 后，内存中的页面号为： 7 # # # #
访问页面 0 后，内存中的页面号为： 7 0 # # #
访问页面 1 后，内存中的页面号为： 7 0 1 # #
访问页面 2 后，内存中的页面号为： 7 0 1 2 #
访问页面 0 后，内存中的页面号为： 7 0 1 2 #
访问页面 3 后，内存中的页面号为： 7 0 1 2 3
访问页面 0 后，内存中的页面号为： 7 0 1 2 3
访问页面 4 后，内存中的页面号为： 4 0 1 2 3 淘汰页面号为:7
访问页面 2 后，内存中的页面号为： 4 0 1 2 3
访问页面 3 后，内存中的页面号为： 4 0 1 2 3
访问页面 0 后，内存中的页面号为： 4 0 1 2 3
访问页面 1 后，内存中的页面号为： 4 0 1 2 3
访问页面 1 后，内存中的页面号为： 4 0 1 2 3
访问页面 7 后，内存中的页面号为： 7 0 1 2 3 淘汰页面号为:4
访问页面 0 后，内存中的页面号为： 7 0 1 2 3
访问页面 1 后，内存中的页面号为： 7 0 1 2 3
访问页面 0 后，内存中的页面号为： 7 0 1 2 3
访问页面 3 后，内存中的页面号为： 7 0 1 2 3
缺页次数为：7
OPT缺页率为：0.389
请按任意键继续...
```

FIFO

```
C:\WINDOWS\system32\cmd. x + v
*****先进先出页面置换算法: *****
页面引用串为: 7 0 1 2 0 3 0 4 2 3 0 1 1 7 0 1 0 3
访问页面 7 后, 内存中的页面号为: 7 # # # #
访问页面 0 后, 内存中的页面号为: 7 0 # # #
访问页面 1 后, 内存中的页面号为: 7 0 1 # #
访问页面 2 后, 内存中的页面号为: 7 0 1 2 #
访问页面 0 后, 内存中的页面号为: 7 0 1 2 #
访问页面 3 后, 内存中的页面号为: 7 0 1 2 3
访问页面 0 后, 内存中的页面号为: 7 0 1 2 3
访问页面 4 后, 内存中的页面号为: 4 0 1 2 3 淘汰页面号为:7
访问页面 2 后, 内存中的页面号为: 4 0 1 2 3
访问页面 3 后, 内存中的页面号为: 4 0 1 2 3
访问页面 0 后, 内存中的页面号为: 4 0 1 2 3
访问页面 1 后, 内存中的页面号为: 4 0 1 2 3
访问页面 1 后, 内存中的页面号为: 4 0 1 2 3
访问页面 7 后, 内存中的页面号为: 4 7 1 2 3 淘汰页面号为:0
访问页面 0 后, 内存中的页面号为: 4 7 0 2 3 淘汰页面号为:1
访问页面 1 后, 内存中的页面号为: 4 7 0 1 3 淘汰页面号为:2
访问页面 0 后, 内存中的页面号为: 4 7 0 1 3
访问页面 3 后, 内存中的页面号为: 4 7 0 1 3
缺页次数为: 9
FIFO缺页率为: 0.500
请按任意键继续. . .
```

3) 通过上图的结果比较, 可以看出哪个算法的缺页率较大, 为什么?

FIFO 的缺页率大。因为存放相同页面号的情况下, FIFO 算法里淘汰页面的次数更多, 所以缺页率也就更大。

步骤3: 完成最近最久未使用页面置换算法(LRU)

当需要换出一个页面时, 淘汰那个在最近一段时间里较久未被访问的页面。它是根据程序执行时所具有的局部性来考虑的, 即那些刚被使用过的页面可能马上还要被使用, 而那些在较长时间里未被使用的页面一般可能不会马上使用。

在新建的项目中, 仔细阅读其中的源代码, 并仿照已经实现的 OPT 和 FIFO 算法来实现最近最久未使用页面置换算法(LRU)。

//最近最久未使用页面置换算法

```
void Lru(int *BlockofMemory, int *PageNumofRef, int BlockCount, int PageNumRefCount) {
    int i;
    int MissCount = 0;
    int EmptyBlockCount = BlockCount;
    printf("*****最近最久未使用页面置换算法: *****\n");

    //输出页面引用串号
    OutputPageNumofRef(PageNumofRef, PageNumRefCount);
    for(i = 0; i < PageNumRefCount; i++) {
        if(!PageInBlockofMemory(PageNumofRef[i], BlockofMemory, BlockCount)) { //页不在内存中
            MissCount++;
            if(EmptyBlockCount > 0) {
                BlockofMemory[BlockCount - EmptyBlockCount] = PageNumofRef[i];
                OutputBlockofMemory(BlockofMemory, BlockCount, -1, PageNumofRef[i]);
                EmptyBlockCount--;
            } else {
```

```

        int k = 0;
        int j = i;
        int q;
        int l;
        int tag = 0;
        int ReplacePage;

        int *T_BlockofMemory;
        T_BlockofMemory = (int*)malloc(BlockCount * sizeof(int));
        ResetBlockofMemory(T_BlockofMemory, BlockCount);

        for(q = 0; q < i; q++){
            j--;
            if(!PageInBlockofMemory(PageNumofRef[j], T_BlockofMemory, BlockCount))
                T_BlockofMemory[k++] = PageNumofRef[j];
            if(k == BlockCount && tag == 0) {
                ReplacePage = PageNumofRef[i];
                for(l = 0; l < BlockCount; l++)
                    if(BlockofMemory[l] == PageNumofRef[j]) {
                        BlockofMemory[l] = ReplacePage;
                        tag = 1;
                        break;
                    }
            }
        }
        free(T_BlockofMemory);
        OutputBlockofMemory(BlockofMemory, BlockCount, ReplacePage, PageNumofRef[i]);
    }
}
else
    OutputBlockofMemory(BlockofMemory, BlockCount, -1, PageNumofRef[i]);
}

printf("缺页次数为: %d\n", MissCount);
printf("LRU 缺页率为: %.3f\n", (float)MissCount / PageNumRefCount);
}

```

步骤4: 正确实现 LRU 算法后的执行结果应该如下所示。

```
C:\WINDOWS\system32\cmd  +  v

*****最近最久未使用页面置换算法：*****
页面引用串为： 7 0 1 2 0 3 0 4 2 3 0 1 1 7 0 1 0 3
访问页面 7 后，内存中的页面号为： 7 # # # #
访问页面 0 后，内存中的页面号为： 7 0 # # #
访问页面 1 后，内存中的页面号为： 7 0 1 # #
访问页面 2 后，内存中的页面号为： 7 0 1 2 #
访问页面 0 后，内存中的页面号为： 7 0 1 2 #
访问页面 3 后，内存中的页面号为： 7 0 1 2 3
访问页面 0 后，内存中的页面号为： 7 0 1 2 3
访问页面 4 后，内存中的页面号为： 4 0 1 2 3 淘汰页面号为：4
访问页面 2 后，内存中的页面号为： 4 0 1 2 3
访问页面 3 后，内存中的页面号为： 4 0 1 2 3
访问页面 0 后，内存中的页面号为： 4 0 1 2 3
访问页面 1 后，内存中的页面号为： 4 0 1 2 3
访问页面 1 后，内存中的页面号为： 4 0 1 2 3
访问页面 7 后，内存中的页面号为： 7 0 1 2 3 淘汰页面号为：7
访问页面 0 后，内存中的页面号为： 7 0 1 2 3
访问页面 1 后，内存中的页面号为： 7 0 1 2 3
访问页面 0 后，内存中的页面号为： 7 0 1 2 3
访问页面 3 后，内存中的页面号为： 7 0 1 2 3
缺页次数为：7
LRU缺页率为：0.389
请按任意键继续. . .
```

步骤5： 结合步骤 2 中 OPT 和 FIFO 模拟结果，说明 LRU 算法的优点，为什么？

LRU 置换将每个页面与它的上次使用的时间关联起来。当需要置换页面时，LRU 选择最长时间没有使用的页面。这种策略可当作在时间上向后看而不是向前看的最优页面置换算法。

LRU 实现简单。当存在热点数据时，LRU 的效率很好，但偶发性的、周期性的批量操作会导致 LRU 命中率急剧下降，缓存污染情况比较严重。

根据实验结果，LRU 的缺页率与 OPT 近似，略小于 FIFO 的缺页率，LRU 算法在某些情况下将接近或等于 OPT 的性能。

3. 实验结论

OPT 算法： 当一个缺页中断发生时，对于保存在内存当中的每一个逻辑页面，计算在它的下一次访问之前，还需等待多长时间，从中选择等待时间最长的那个，作为被置换的页面。这是一种理想情况，在实际系统中是无法实现的，因为操作系统无法知道每一个页面要等待多长时间以后才会再次被访问。可用作其他算法的性能评价的依据。

FIFO 算法选择在内存中驻留时间最长的页面淘汰。从链表的排列顺序来看，链首页面的驻留时间最长，链尾页面的驻留时间最短。当发生一个缺页中断时，把链首页面淘汰出去，并把新的页面添加到链表的末尾。但性能较差，调出的页面有可能是经常要访问的页面。并且有 belady 现象。FIFO 算法很少单独使用。

FIFO 和 OPT 算法的关键区别在于，除了在时间上向后或向前看之外，FIFO 算法使用的是页面调入内存的时间，OPT 算法使用的是页面将来使用的时间。

LRU 算法： 当一个缺页中断发生时，选择最久未使用的那个页面，并淘汰。它是对最优页面置换算法的一个近似，其依据是程序的局部性原理，即在最近一小段时间(最近几条指令)内，如果某些页面被频繁地访问，那么再将来的一小段时间内，他们还可能会再一次被频繁地访问。反过来说，如果过去某些页面长时间未被访问，那么在将来它们还可能会长时间地得不到访问。

实验五 文件系统设计试验

1. 试验目的

通过设计一个基于索引结构的文件系统，加深对文件系统的基本知识理解。了解文件系统设计的基本概念。

1. 熟悉文件系统的物理结构；
2. 熟悉文件系统的目录管理；
3. 掌握文件系统空闲空间管理的基本方法；
4. 进一步理解现代操作系统文件管理知识。

2. 实验内容与步骤

1. 设计一个文件系统的索引结构，描述逻辑结构与物理索引结构之间的关系；
2. 设计文件目录，描述文件名与文件物理结构之中的映射关系；
3. 定义作业；
4. 设计文件建立；
5. 设计文件系统的其它功能；

3. 实验结论

参考程序：

```
#include "stdio.h"
#include "stdlib.h"

//文件索引表的定义
struct index{
    int lr[32];
    int pr[32];
    char st[32];
}*wq;

#define JOBN 20

//文件目录的定义
struct list{
    char names[32];
    int size[32];
    struct index*p[32]; //文件的索引表地址
}*HEAD;
```

```

//作业序列
struct que{
    char name;
    int size;
}job[JOBN];

int i,j,ly,li;
char bb;
int NFile=0;//系统的总文件数，模拟开始时为0;
int N=0;//工作变量，标记当前分配文件
int M=32;//系统中空闲磁盘物理块数，开始为32块
int J[4][8];//用位图表示这些磁盘物理块，J[j][i]为0时标记第j*8+i块空闲，为1标记该块已分配出去
FILE *e;//记录模拟中的相关数据
int jobs=0;//作业数

void run(){
    int x1,y,z;
    //如果当前空闲磁盘物理块数能够满足需要，则进行分配
    if(job[N].size<=M) {
        ly=0;
        N=NFile;//标记当前分配文件
        NFile++;//标记下次分配处理文件
        M-=job[N].size;
        HEAD->names[N]=job[N].name;

        //将文件名以及大小，索引地址填写到文件目录
        HEAD->size[N]=job[N].size;
        wq=(struct index *)malloc(sizeof(struct index));
        HEAD->p[N]=wq;

        //对分配文件的索引表初始化
        for(z=0;z<32;z++){
            wq->lr[z]=z;//逻辑块号
            wq->pr[z]=0;////物理块地址
            wq->st[z]='N';
        }

        //从位图中分配，分配出去的块其状态为1。一直到分配完成
        for(j=0;j<4&&(ly<job[N].size);j++)
            for(i=0;i<8&&(ly<job[N].size);i++) {
                if(J[j][i]==0){li=j*8+i;wq->pr[ly]=li;wq->st[ly]='Y';ly++;J[j][i]=1;}//找到
                一个空闲块，就分配出去
            }
    }
}

```



```

        }
    }
    //如果当前空闲磁盘物理块数不能够满足需要，则出错处理
    else{
        fprintf(e, "\n There are no free blocks in the memory now! \n");
        fprintf(e, "File %c must wait!\n", job[N].name);
    }

    //报告目前为止的分配情况，首先报告文件目录的部分信息
    fprintf(e, "...This time ,the file directory:    ----\n");
    fprintf(e, "NAME          INDEX_ADDRESS\n");
    for(x1=0; x1<N+1; x1++){
        fprintf(e, "          %c          %x\n", HEAD->names[x1], HEAD->p[x1]);

        //其次报告文件索引表的部分信息
        for(x1=0; x1<N+1; x1++){
            fprintf(e, " //////////The index of FILE%c:////////\n", HEAD->names[x1]);
            fprintf(e, " LOGIC_NUMBER  PHYSICAL_NUMBER  FLAG\n");
            for(y=0; y<HEAD->size[x1]; y++){
                fprintf(e, "   %d   %d   %c\n", HEAD->p[x1]->lr[y], HEAD->p[x1]->pr[y], HEAD->p[x1]->st[y]);
            }

            //第三，报告位图信息
            fprintf(e, "          This time the bit mapping graph:          \n");
            for(j=0; j<4; j++){
                fprintf(e, "          ");
                for(i=0; i<8; i++) fprintf(e, "%d", J[j][i]);
                fprintf(e, "\n");
            }
        }
    }
}

int main(){
    int k;
    e=fopen("results.txt", "w");//打开保存结果的文件
    for(j=0; j<4; j++){
        for(i=0; i<8; i++)
            J[j][i]=0;//初始化位图
    }
    HEAD=(struct list*)malloc(sizeof(struct list));

    for(i=0; i<32; i++){
        HEAD->names[i]=' ';
        HEAD->size[i]=0;
        HEAD->p[i]=NULL;
    }//初始化文件目录
}

```

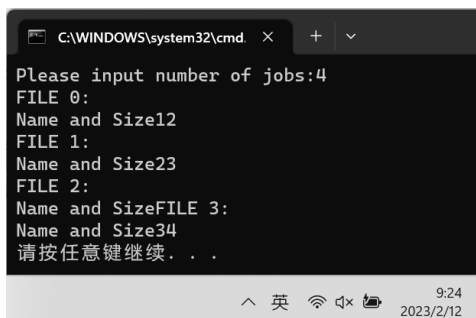
```

printf("Please input number of jobs:");
scanf("%d",&jobs);
scanf("%c",&bb);
int kk=0;

//以下输入建立的文件数以及文件名字，并将这些信息保存在 job 数组之中
while(kk<jobs){
    fprintf(e,"FILE %d: \n",kk);
    printf("FILE %d: \n",kk);
    printf("Name and Size");
    scanf("%c,%d",&(job[kk].name),&(job[kk].size));
    scanf("%c",&bb);
    fprintf(e,"%c,%d",job[kk].name,job[kk].size);
    kk++;
}
for(k=1;k<=jobs;k++)
    run();//每个文件进行一次分配
//回收资源
fclose(e);
for(i=0;i<32;i++){
    free(HEAD->p[i]);
    HEAD->p[i]=NULL;
}
free(HEAD);
HEAD=NULL;
return 0;
}

```

试验运行结果:



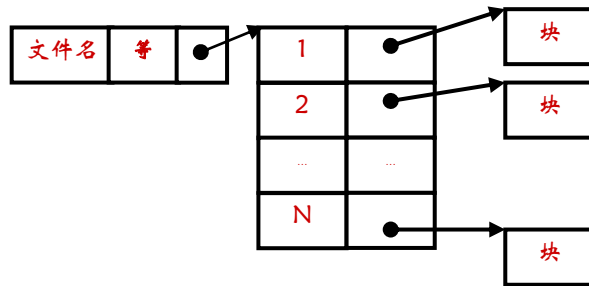
```

C:\WINDOWS\system32\cmd. x + v
Please input number of jobs:4
FILE 0:
Name and Size12
FILE 1:
Name and Size23
FILE 2:
Name and SizeFILE 3:
Name and Size34
请按任意键继续. . .

```

附录 B：文件系统模拟程序

1. 索引结构和文件目录



2. 文件目录模拟

```
struct list{
    char names[32];
    int size[32];
    struct index*p[32]; // 文件的索引表地址
}*HEAD;
```

该模拟文件最多只能模拟 32 个文件,HEAD 指针指向了该模拟目录。

3. 文件索引表模拟

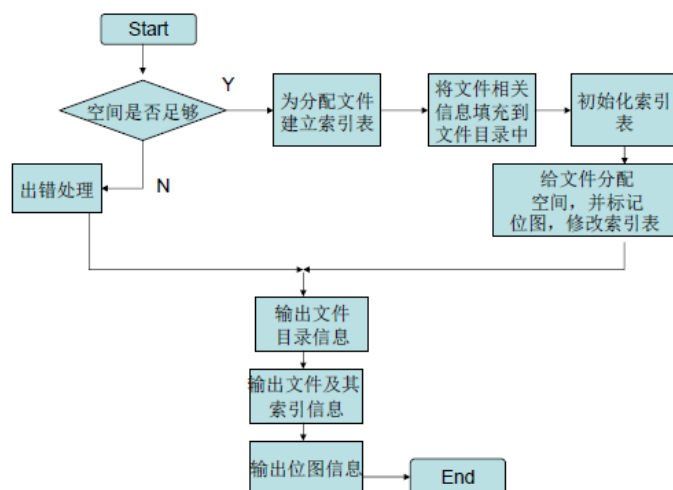
文件索引表的定义

```
struct index{
    int lr[32];
    int pr[32];
    char st[32];
}*wq;
```

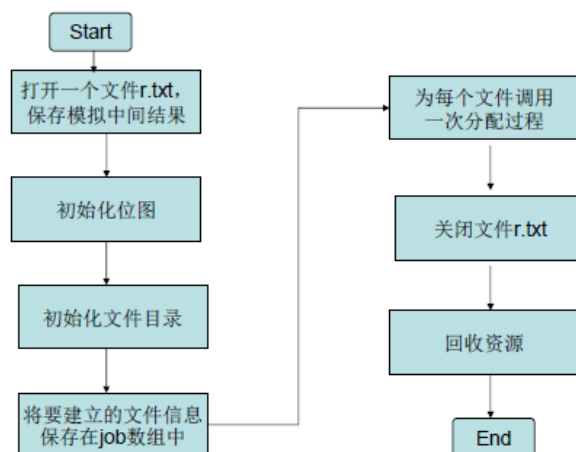
4. 位图

int J[4][8]。位图 J 表示这些磁盘物理块的情况。J[j][i] 为 0 时标记块空闲，为 1 标记该块已分配出去。各块号按行存储。因此，J[j][i] 表示 $j*8+i$ (块号) 块的分配情况

5. run 函数



6. Main 函数



附录 B : Visual studio 2008 中建项目、运行 C++程序的方法

一、建项目

1. 文件->新建->项目->Visual C++->Win32 控制台应用程序->输入项目名称->控制台应用程序, 空项目->完成。
2. 右键单击新建项目名称->从右键菜单中选择“添加”->新建项->C++文件(.cpp)
3. 键入程序并执行即可。

在 Visual studio 2008 中执行 C++程序:

二、运行 C++程序

不管用什么方法,注意 C++中数据类型及其转换。注意 Visual studio 2008 中项目的默认字符编码是否与 VC 的一致

方法一

第 1 步与上相同;

第 2 步键入程序时,用

```
#include "iostream"
using namespace std;
代替#include <iostream.h>
```

方法二

1. 新建一 CLR 控制台应用程序, 步骤与上类似;

2. 在键入源程序时, 用如下代码替换#include <iostream.h>

```
#include "stdafx.h"
#include "iostream"
```

```
using namespace std;
```

3. 例如:

```
#include "stdafx.h"
#include "iostream"
using namespace std;
int main() {
    long l;
    double d;
    int a;
    float b;
    short c;
    cout<<"size of char:"<<sizeof(char)<<endl;
    cout<<"size of long:"<<sizeof(l)<<endl;
    cout<<"size of double:"<<sizeof(d)<<endl;
    cout<<"size of float:"<<sizeof(b)<<endl;
    cout<<"size of int:"<<sizeof(a)<<endl;
    cout<<"size of short:"<<sizeof(c)<<endl;
    return 0;
}
```

4. 调试——>开始执行，程序成功运行;

说明：经查看 Visual studio 的安装目录，发现已无 `iostream.h`，代之以文本文件 `iostream`，如果还有其它库的变化，类似上述处理即可。

4. 思考

该试验中，从功能上讲，根据所学的文件系统管理方面知识，你所设计的（模拟）文件还有那些没有实现：

程序要为每个文件建立一张索引表，索引表中用数组指出文件信息所在的逻辑块号和与之对应的物理块号。程序中 `int J[i][i]` 表示这些磁盘物理块的情况。`J[i][i]` 为 0 时标记块空闲，为 1 标记该块已分配出去。各块号按行存储。

1. 在命令提示符中运行应用程序时要先转到应用程序所在的盘符下，要把应用程序所在目录位置输入正确，才能找到相应的程序运行。
2. 文件的物理结构和组织是指逻辑文件在物理存储空间中存放方法和组织关系；使用多级目录可以解决文件重名问题与缩短搜索时间；现代操作系统文件管理就是对块空间的管理，包括空闲块的分配、回收和组织；索引文件结构中的索引表是用来指示逻辑记录 and 物理块之间对应关系的。
3. 通过本次实验我了解了文件系统的基本概念，物理文件结构有三种结构，连续文件、链接文件、索引文件。用户能直接处理其中的结构与数据的是逻辑

结构，文件在外存上的存储组织形式是物理结构。只有合理的进行存储空间的管理，才能保证多用户共享外存和快速的实现文件的按名存取。

对以后设计的修改建议：

增加一些功能，比如打包的文件系统、远程的文件系统、把某个空间抽象成文件系统、加密的文件系统、带日志的文件系统等等……

目前所做的修改及实际结果如下：