

课程名称	操作系统	课程编号	A2130330
实验地点	综合实验楼 A511/A512	实验时间	2019-05-13
校外指导教师		校内指导教师	常光辉
实验名称	实验二 并发与调度		
评阅人签字		成绩	

一、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 2016 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 2016 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。

二、工具/准备工作

1. 回顾教材相关内容；
2. 安装有 Windows Server 2016 的计算机或虚拟机；
3. 系统中装有 Visual Studio 或 Visual C++ 6.0 或其他 C++编译器。

三、实验环境

操作系统：Windows Server 2016（虚拟机）

编程语言：C++

集成开发环境：Visual Studio 2019

四、实验步骤与实验过程

1. 事件对象

清单 2-1 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 2016。

步骤 2: 在“开始”菜单中单击“程序”-“Microsoft Visual Studio 2019”，进入 Visual Studio 2019 的窗口。

步骤 3: 在 Visual Studio 2019 窗口的工具栏中单击“文件”按钮，在解决方案 OSLab 中新建一个项目，命名为 Lab2_1，并在源文件中添加新建项，命名为 Lab2_1.cpp，将实验指导书中的代码拷贝至该文件中。

步骤 4: 单击“生成”菜单中的“编译”命令，系统对 Lab2_1.cpp 进行编译。

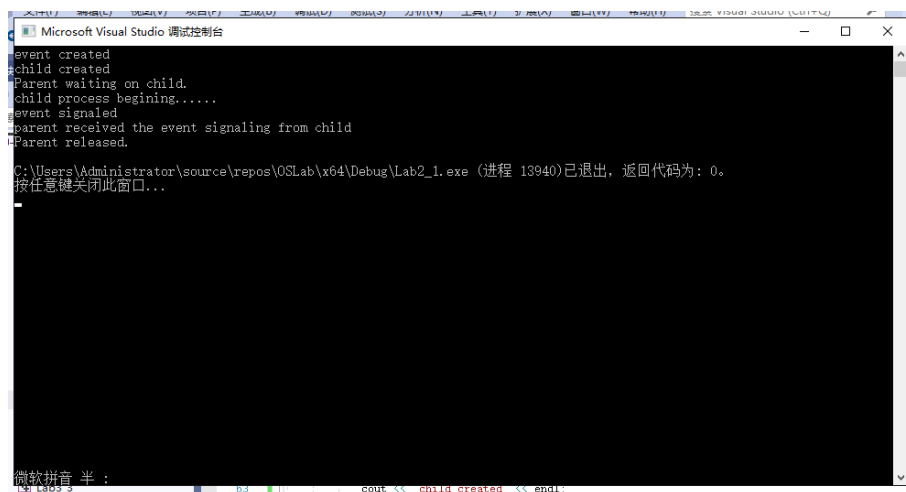
步骤 5: 编译完成后，单击“生成”菜单中的“生成”命令，建立 Lab2_1.exe 可执行文件。

操作能否正常进行？如果不行，则可能的原因是什么？

由于程序代码是由 pdf 文件复制出来的，调整缩进与分号等规范后，进一步调整了安全性，如使用 sprintf 等，并使用了命名空间 std，减少代码冗余，调整过后，程序代码可以调通。

步骤 6: 在解决方案资源管理器中右键当前项目，并设置为启动项目。在工具栏单击“调试”-“开始执行（不调试）”按钮，执行 Lab2_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？）：



```
Microsoft Visual Studio 调试控制台
event created
child created
Parent waiting on child.
child process begining.....
event signaled
parent received the event signaling from child
Parent released.
C:\Users\Administrator\source\repos\OSLab\x64\Debug\Lab2_1.exe (进程 13940) 已退出，返回代码为：0。
按任意键关闭此窗口...

cout << "child created" << endl;
```

图 2-1 程序 Lab2_1 运行结果

1) event created

2) child created

3) Parent waiting on child.

4) child process begining.....

5) event signaled

6) parent received the event signaling from child

7) Parent released.

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）

结果与期望一致。

子进程向事件发出信号，父进程终止。两个进程在发出信号之后几乎立即终止。

阅读和分析程序 Lab2_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. 创建一个事件使用了 CreateEvent() 函数；

b. 创建时设置的初始信号状态为 FALSE。

2) 创建一个进程（子进程）使用了哪一个系统函数？

创建一个（子）进程使用了 CreateProcess() 函数。

3) 从步骤 6 的输出结果，对照分析 Lab2_1 程序，可以看出程序运行的流程吗？请简单描述：

如图 2-2：

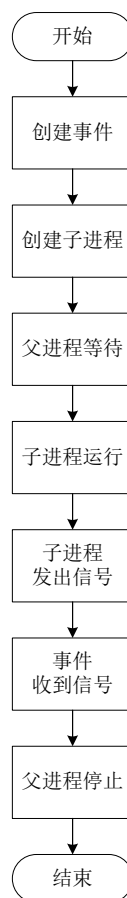


图 2-2 流程图

2. 互斥体对象

清单 2-2 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 Studio 2019 窗口的工具栏中单击“文件”按钮，在解决方案 OSLab 中新建一个项目，命名为 Lab2_2，并在源文件中添加新建项，命名为 Lab2_2.cpp，将实验指导书中的代码拷贝至该文件中。

步骤 8: 单击“生成”菜单中的“编译”命令，系统对 Lab2_2.cpp 进行编译。

步骤 9: 编译完成后，单击“生成”菜单中的“生成”命令，建立 Lab2_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？

由于程序代码是由 pdf 文件复制出来的，调整缩进与分号等规范后，使用了命名空间 std，减少代码冗余，调整过后，程序代码可以调通。

步骤 10: 在解决方案资源管理器中右键当前项目，并设置为启动项目。在工具栏单击“调试” - “开始执行（不调试）”按钮，执行 Lab2_2.exe 程序。

分析程序 Lab2_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?)：

thread: 13524 value: 1 access: 50

thread: 15952 value: 0 access: 49

thread: 13524 value: 1 access: 48

thread: 15952 value: 0 access: 47

thread: value: ... access: ...

thread: value: ... access: ...

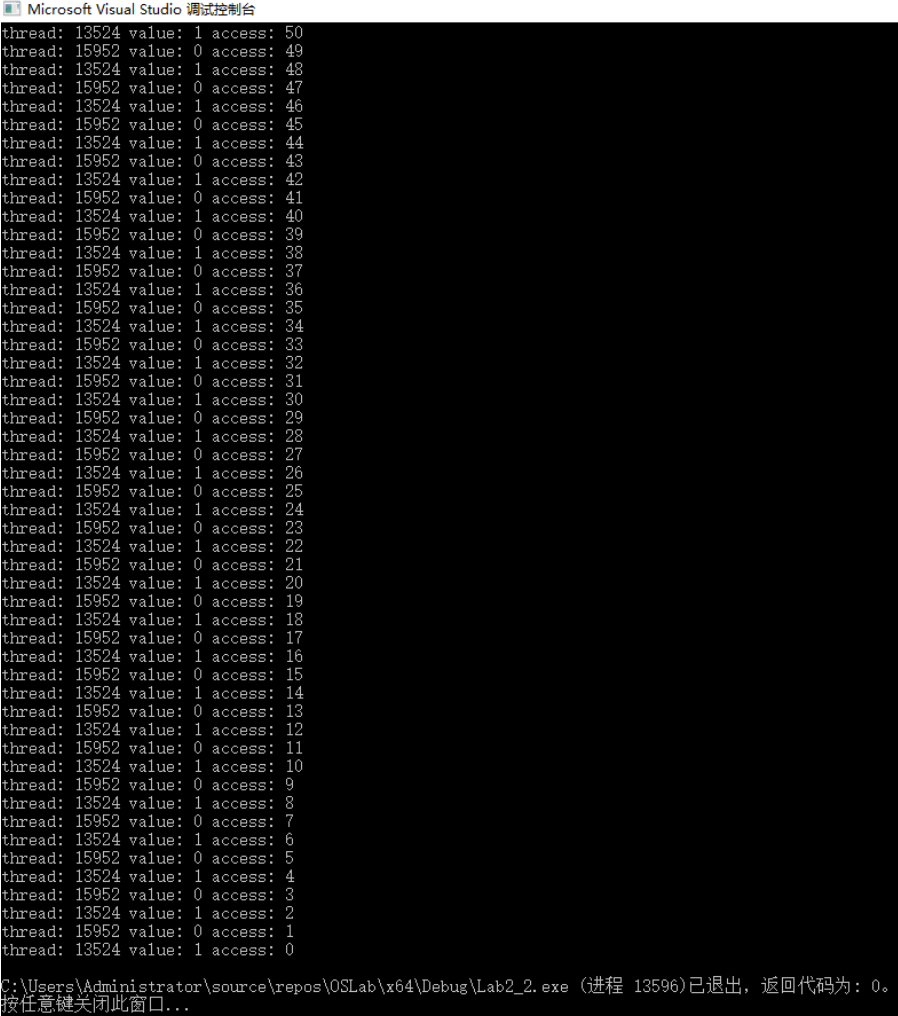
thread: 15952 value: 0 access: 3

thread: 13524 value: 1 access: 2

thread: 15952 value: 0 access: 1

thread: 13524 value: 1 access: 0

具体结果如图 2-3 所示：



```
Microsoft Visual Studio 调试控制台
thread: 13524 value: 1 access: 50
thread: 15952 value: 0 access: 49
thread: 13524 value: 1 access: 48
thread: 15952 value: 0 access: 47
thread: 13524 value: 1 access: 46
thread: 15952 value: 0 access: 45
thread: 13524 value: 1 access: 44
thread: 15952 value: 0 access: 43
thread: 13524 value: 1 access: 42
thread: 15952 value: 0 access: 41
thread: 13524 value: 1 access: 40
thread: 15952 value: 0 access: 39
thread: 13524 value: 1 access: 38
thread: 15952 value: 0 access: 37
thread: 13524 value: 1 access: 36
thread: 15952 value: 0 access: 35
thread: 13524 value: 1 access: 34
thread: 15952 value: 0 access: 33
thread: 13524 value: 1 access: 32
thread: 15952 value: 0 access: 31
thread: 13524 value: 1 access: 30
thread: 15952 value: 0 access: 29
thread: 13524 value: 1 access: 28
thread: 15952 value: 0 access: 27
thread: 13524 value: 1 access: 26
thread: 15952 value: 0 access: 25
thread: 13524 value: 1 access: 24
thread: 15952 value: 0 access: 23
thread: 13524 value: 1 access: 22
thread: 15952 value: 0 access: 21
thread: 13524 value: 1 access: 20
thread: 15952 value: 0 access: 19
thread: 13524 value: 1 access: 18
thread: 15952 value: 0 access: 17
thread: 13524 value: 1 access: 16
thread: 15952 value: 0 access: 15
thread: 13524 value: 1 access: 14
thread: 15952 value: 0 access: 13
thread: 13524 value: 1 access: 12
thread: 15952 value: 0 access: 11
thread: 13524 value: 1 access: 10
thread: 15952 value: 0 access: 9
thread: 13524 value: 1 access: 8
thread: 15952 value: 0 access: 7
thread: 13524 value: 1 access: 6
thread: 15952 value: 0 access: 5
thread: 13524 value: 1 access: 4
thread: 15952 value: 0 access: 3
thread: 13524 value: 1 access: 2
thread: 15952 value: 0 access: 1
thread: 13524 value: 1 access: 0
C:\Users\Administrator\source\repos\OSLab\x64\Debug\Lab2_2.exe (进程 13596)已退出，返回代码为：0。
按任意键关闭此窗口...
```

图 2-3 程序 Lab3_2 运行结果

2) 根据运行输出结果，对照分析 Lab2_2 程序，可以看出程序运行的流程吗？请简单描述：

该程序使用了互斥体 CCountUpDown 来保护能被同时访问的共享资源。

该程序首先创建了两个进程来访问共享值，并创建了互斥体来访问数值，随后两个进程依次轮流访问数值（访问后改变并释放）。

五、实验结果与分析

实验结果与分析见实验步骤与实验过程。

六、实验心得体会

通过本次实验，我加深了对计算机线程的理解，并深入理解了互斥体等相关概念。通过分析实验程序，了解管理事件对象的 API；通过对事件和互斥体对象的编程，加深

了对 Windows Server 2016 线程同步的理解。

附录 程序清单

清单 2-1

```
1.  // event 项目
2.  #include <windows.h>
3.  #include <iostream>
4.  using namespace std;
5.
6.  // 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
7.  static LPCTSTR g_szContinueEvent = "w2kdg.EventDemo.event.Continue";
8.
9.  // 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
10. BOOL CreateChild()
11. {
12.     // 提取当前可执行文件的文件名
13.     TCHAR szFilename[MAX_PATH];
14.     ::GetModuleFileName(NULL, szFilename, MAX_PATH);
15.
16.     // 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
17.     TCHAR szCmdLine[MAX_PATH];
18.     ::sprintf_s(szCmdLine, "\\\"%s\\\" child", szFilename);
19.     // 子进程的启动信息结构
20.     STARTUPINFO si;
21.     ::ZeroMemory(&si, sizeof(si));
22.     si.cb = sizeof(si); // 必须是本结构的大小
23.     // 返回的子进程的进程信息结构
24.     PROCESS_INFORMATION pi;
25.     // 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
26.     BOOL bCreateOK = ::CreateProcess(
27.         szFilename, // 生成的可执行文件名
28.         szCmdLine, // 指示其行为与子进程一样的标志
29.         NULL, // 子进程句柄的安全性
30.         NULL, // 子线程句柄的安全性
31.         FALSE, // 不继承句柄
32.         0, // 特殊的创建标志
33.         NULL, // 新环境
34.         NULL, // 当前目录
35.         &si, // 启动信息结构
36.         &pi); // 返回的进程信息结构
37.
38.     // 释放对子进程的引用
39.     if (bCreateOK) {
40.         ::CloseHandle(pi.hProcess);
```

```

41.         ::CloseHandle(pi.hThread);
42.     }
43.     return (bCreateOK);
44. }
45.
46. // 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
47. void WaitForChild()
48. {
49.     // create a new event object for the child process
50.     // to use when releasing this process
51.     HANDLE hEventContinue = ::CreateEvent(
52.         NULL,    // 缺省的安全性，子进程将具有访问权限
53.         TRUE,    // 手工重置事件
54.         FALSE,   // 初始时是非接受信号状态
55.         g_szContinueEvent); // 事件名称
56.
57.     if (hEventContinue != NULL)
58.     {
59.         cout << "event created " << endl;
60.         // 创建子进程
61.         if (::CreateChild())
62.         {
63.             cout << "child created" << endl;
64.             // 等待，直到子进程发出信号
65.             cout << "Parent waiting on child." << endl;
66.             ::WaitForSingleObject(hEventContinue, INFINITE);
67.             ::Sleep(1500); // 删去这句试试
68.             cout << "parent received the event signaling from child" << endl;
69.         }
70.
71.         // 清除句柄
72.         ::CloseHandle(hEventContinue);
73.         hEventContinue = INVALID_HANDLE_VALUE;
74.     }
75. }
76.
77. // 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
78. void SignalParent()
79. {
80.     // 尝试打开句柄
81.     cout << "child process begining....." << endl;
82.     HANDLE hEventContinue = ::OpenEvent(
83.         EVENT_MODIFY_STATE, // 所要求的最小访问权限
84.         FALSE,             // 不是可继承的句柄
85.         g_szContinueEvent); // 事件名称
86.

```

```

87.     if (hEventContinue != NULL)
88.     {
89.         ::SetEvent(hEventContinue);
90.         cout << "event signaled" << endl;
91.     }
92.
93.     // 清除句柄
94.     ::CloseHandle(hEventContinue);
95.     hEventContinue = INVALID_HANDLE_VALUE;
96. }
97.
98. int main(int argc, char* argv[])
99. {
100.     // 检查父进程或是子进程是否启动
101.     if (argc > 1 && ::strcmp(argv[1], "child") == 0)
102.     {
103.         // 向父进程创建的事件发出信号
104.         ::SignalParent();
105.     }else{
106.         // 创建一个事件并等待子进程发出信号
107.         ::WaitForChild();
108.         ::Sleep(1500);
109.         cout << "Parent released." << endl;
110.     }
111.     return 0;
112.}

```

清单 2-2

```

1.  // mutex 项目
2.  #include <windows.h>
3.  #include <iostream>
4.  using namespace std;
5.
6.  // 利用互斥体来保护同时访问的共享资源
7.  class CCountUpDown
8.  {
9.  public:
10.     // 创建者创建两个线程来访问共享值
11.     CCountUpDown(int nAccesses) : m_hThreadInc(INVALID_HANDLE_VALUE), m_hThreadDec(I
        NVALID_HANDLE_VALUE),
12.         m_hMutexValue(INVALID_HANDLE_VALUE), m_nValue(0), m_nAccess(nAccesses)
13.     {
14.         // 创建互斥体用于访问数值
15.         m_hMutexValue = ::CreateMutex(
16.             NULL, // 缺省的安全性

```



```

17.         TRUE, // 初始时拥有，在所有的初始化结束时将释放
18.         NULL); // 匿名的
19.
20.         m_hThreadInc = ::CreateThread(
21.             NULL, // 缺省的安全性
22.             0, // 缺省堆栈
23.             IncThreadProc, // 类线程进程
24.             reinterpret_cast<LPVOID>(this), // 线程参数
25.             0, // 无特殊的标志
26.             NULL); // 忽略返回的 id
27.
28.         m_hThreadDec = ::CreateThread(
29.             NULL, // 缺省的安全性
30.             0, // 缺省堆栈
31.             DecThreadProc, // 类线程进程
32.             reinterpret_cast<LPVOID>(this), // 线程参数
33.             0, // 无特殊的标志
34.             NULL); // 忽略返回的 id
35.
36.         // 允许另一线程获得互斥体
37.         ::ReleaseMutex(m_hMutexValue);
38.     }
39.
40.     // 解除程序释放对对象的引用
41.     virtual ~CCountUpDown()
42.     {
43.         ::CloseHandle(m_hThreadInc);
44.         ::CloseHandle(m_hThreadDec);
45.         ::CloseHandle(m_hMutexValue);
46.     }
47.
48.     // 简单的等待方法，在两个线程终止之前可暂停主调者
49.     virtual void WaitForCompletion()
50.     {
51.         // 确保所有对象都已准备好
52.         if (m_hThreadInc != INVALID_HANDLE_VALUE &&
53.             m_hThreadDec != INVALID_HANDLE_VALUE)
54.         {
55.             // 等待两者完成（顺序并不重要）
56.             ::WaitForSingleObject(m_hThreadInc, INFINITE);
57.             ::WaitForSingleObject(m_hThreadDec, INFINITE);
58.         }
59.     }
60.
61. protected:
62.

```

```

63.    // 改变共享资源的简单的方法
64.    virtual void DoCount(int nStep)
65.    {
66.        // 循环, 直到所有的访问都结束为止
67.        while (m_nAccess > 0)
68.        {
69.            // 等待访问数值
70.            ::WaitForSingleObject(m_hMutexValue, INFINITE);
71.            // 改变并显示该值
72.            m_nValue += nStep;
73.            cout << "thread: " << ::GetCurrentThreadId()
74.                << " value: " << m_nValue
75.                << " access: " << m_nAccess << endl;
76.            // 发出访问信号并允许线程切换
77.            --m_nAccess;
78.            ::Sleep(1000); // 使显示速度放慢
79.            // 释放对数值的访问
80.            ::ReleaseMutex(m_hMutexValue);
81.        }
82.    }
83.
84.    static DWORD WINAPI IncThreadProc(LPVOID lpParam)
85.    {
86.        // 将参数解释为 'this' 指针
87.        CCountUpDown* pThis =
88.            reinterpret_cast<CCountUpDown*>(lpParam);
89.        // 调用对象的增加方法并返回一个值
90.        pThis->DoCount(+1);
91.        return (0);
92.    }
93.
94.    static DWORD WINAPI DecThreadProc(LPVOID lpParam)
95.    {
96.        // 将参数解释为 'this' 指针
97.        CCountUpDown* pThis =
98.            reinterpret_cast<CCountUpDown*>(lpParam);
99.        // 调用对象的减少方法并返回一个值
100.        pThis->DoCount(-1);
101.        return (0);
102.    }
103.
104.protected:
105.    HANDLE m_hThreadInc;
106.    HANDLE m_hThreadDec;
107.    HANDLE m_hMutexValue;
108.    int m_nValue;

```

```
109.     int m_nAccess;  
110.};  
111.  
112.int main()  
113.{  
114.    CCountUpDown ud(50);  
115.    ud.WaitForCompletion();  
116.    return 0;  
117.}
```