《Windows核心编程》

实验指导书

成都信息大学网络空间安全学院

二零二四年　十一月

实验项目名称：进程和线程编程

实验项目性质：

所属课程名称：Windows核心编程

实验计划学时：4学时

一、 实验目的

1、 完成一个Windows窗口应用程序，熟悉Windows窗口应用结构，消息驱动

2、 掌握WNDCLASSEX结构体、CreateWindows等API函数，图标、光标。了解GDI中DC概念，以及绘图函数

3、 并掌握进程和线程的概念、进程的状态

4、 掌握线程同步技术，包括线程用户模式下同步，例如临界区（关键段）线程同步，

以及采用内核对象模式线程同步，例如事件，互斥量，信号量

5、 熟悉进程和线程API函数编程，要求能够完成进程创建，以及应用线程相关函数完成多线程编程，并完成多线程同步。

6、 熟悉process Explorer软件和spy++软件，使用这两款软件工具对内核对象，进程和线程相关信息进行查看。

二、 实验内容和要求

1、 使用process Explorer软件和Spy++，并学会使用软件，查看内核对象，理解内核对象，进程和线程概念。

2、 1) 创建一个窗口应用程序，完成自定义图标（最小化图标和小图标），以及窗口光标，窗口标题名字;

2) 采用自创画笔和画刷，绘制一个多边形，长方形，椭圆，直线等图形 ；

3）在窗口应用程序添加菜单子项“打开记事本”，然后点击菜单实现，调用CreateToolhelp32Snapshot记事本应用程序是否打开，若没有，则调用CreateProcess创建子进程，打开记事本功能。

4) 窗口应用程序点击鼠标左键，则调用FindWindows找到记事本窗口，分别在记事本和应用程序窗口输出鼠标左键的坐标，并在记事本和窗口应用程序鼠标左键坐标点为圆心，输出圆形，并用自创画笔和画刷填充。

5) 然后菜单项新建菜单子项“退出记事本”，退出记事本，调用结束进程函数TerminateProcess结束记事本进程。

3、 程序内存修改测试，在书本代码“02MemRepair”基础上进行修改，调用ReadProcessMemory查找窗口应用程序圆心的地址，然后通过WriteProcessMemory写入新的值，实现窗口应用程序中原有值的修改。（选做）

4、 程序创建三个线程，一个主线程，主线程创建两个附加线程，采用事件或者其他线程同步机制实现线程同步：

1）第一个附加线程功能为：屏幕输入字符串，写入文件，字符串写入完，通知第二个附加线程。

2）第二个附加线程功能为：读取文件中字符串中，查找一个单词，若找到了该单

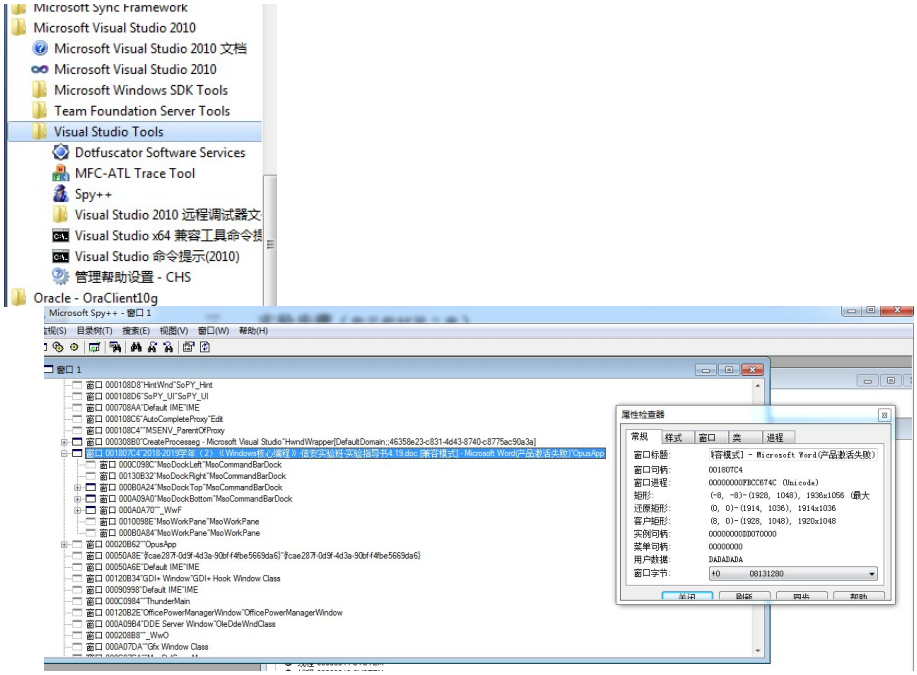
词，通知主线程把找到的单词显示出来；

3）若附加线程退出时，还未查找到此单词，则通知主线程，打印出未找到此单词。

三、 实验步骤（参见教材第2章）

1、用process Explorer软件查看进程相关限制信息。以及process Explorer工具其他作用。具体应用可以参考文档“PROCEXP进程管理器简明使用教程”，下载地址 http://wenku.baidu.com/view/d57b2a97dd88d0d233d46a96.html?re=view###

2、使用Visual studio tools的Spy++查看窗口相关信息，学会使用Spy++软件



3、1) 创建一个窗口应用程序，完成自定义图标（最小化图标和小图标），以及窗口光标，窗口标题名字; （参考书上P86-91的内容）

2)采用自创画笔和画刷，绘制一个多边形，长方形，椭圆，直线，弧线，以及饼状图等图形 ；（参考书上4.4小节）

3）按下键盘上某键，则调用CreateProcess创建子进程，打开记事本程序，则向记事本中输出子进程和线程ID然后显示出来。（参考书上实例代码 02CreateProcess，04 TellToClose 04TimerDemo）

4) 窗口应用程序点击鼠标左键，则分别在记事本和应用程序窗口输出鼠标左键的坐标，并在记事本和窗口应用程序鼠标左键坐标点为圆心，输出半径为100的圆，并用自创画笔和画刷填充。（参考书上实例代码 04TimerDemo）

5) 然后菜单项新建菜单子项“退出记事本”，退出记事本，调用结束进程函数TerminateProcess结束记事本进程。（参考书上实例代码 02 TerminateProcess）

以visual studio 2017为例，看“第一个实验视频录像”

主要参考功能参考代码

调用CreateProcess创建子进程，打开记事本程序，则向记事本中输出子进程和线程ID然后显示出来.

参考代码：

```
WCHAR szCommandLine[] = L"notepad";
 STARTUPINFO si = { sizeof(si) };
 PROCESS_INFORMATION pi;
 si.dwFlags = STARTF_USESHOWWINDOW;    // 指定wShowWindow成员有效
 si.wShowWindow = TRUE;          // 此成员设为TRUE的话则显示新建进程的主窗口,
                         // 为FALSE的话则不显示
 BOOL bRet = ::CreateProcess(
      NULL,              // 不在此指定可执行文件的文件名
      szCommandLine,          // 命令行参数
      NULL,              // 默认进程安全性
      NULL,              // 默认线程安全性
      FALSE,             // 指定当前进程内的句柄不可以被子进程继承
      CREATE_NEW_CONSOLE,     // 为新进程创建一个新的控制台窗口
      NULL,              // 使用本进程的环境变量
      NULL,              // 使用本进程的驱动器和目录
      &si,
      &pi);
 WCHAR szPoint[100];
 wsprintf(szPoint, L"新进程的进程ID号:%d,新进程的主线程ID号:%d", pi.dwProcessId,
pi.dwThreadId);

 if (bRet)
 {
      MessageBox(NULL, L"子进程创建成功", L"提示", MB_OK);
      Sleep(1000);
      HWND hwnd = ::FindWindow(NULL, L"无标题 - 记事本");
      if (hwnd != NULL)
          {
          MessageBox(NULL, L"打开窗口成功", L"提示", MB_OK);
               HDC hdc;
               hdc = GetDC(hwnd);
               HPEN hp;
               hp = CreatePen(PS_DASH, 1, RGB(255, 0, 255));
               SelectObject(hdc, hp);

               HFONT hf;
               hf = CreateFont        //引入新字体
               (10,          //字体高度
                    0,
                    0,
                    0,
                    FW_NORMAL,
                    1,            //定义斜体
                    1,            //定义输出时带下划线
                    0,
                    GB2312_CHARSET,    //所使用的字符集
                    OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS,
                    DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_DONTCARE,
                    L"隶书"
               );
               SelectObject(hdc, hf);
               SetTextColor(hdc, RGB(255, 0, 255));
               SetBkColor(hdc, RGB(0, 0, 0));
               ::TextOut(hdc, 100, 100, szPoint, lstrlen(szPoint));
```

```
                }
                else
                {
                MessageBox(NULL, L"打开窗口失败", L"提示", MB_OK);;
                }

            // 既然我们不使用两个句柄，最好是立刻将它们关闭
            hpr = pi.hProcess;
            ::CloseHandle(pi.hThread);
        //  ::CloseHandle(pi.hProcess);
             }
```

4、 以书上代码02MemRepair基础上进行修改

其中调用CreateToolhelp32Snapshot查找内容2中窗口应用程序，若没有，则创建窗口应用程序子进程参考代码

```
// 启动窗口应用程序进程
    char szFileName[] = "testeg1.exe";
    PROCESSENTRY32 pe32;
    // 在使用这个结构之前，先设置它的大小
    pe32.dwSize = sizeof(pe32);

    // 给系统内的所有进程拍一个快照
    HANDLE hProcessSnap = ::CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hProcessSnap == INVALID_HANDLE_VALUE)
    {
        printf(" CreateToolhelp32Snapshot调用失败！ \n");
        return -1;
    }

    // 遍历进程快照，然后查找窗口应用程序
    BOOL bMore = ::Process32First(hProcessSnap, &pe32);
    char temp[100];
    int flag = 0;//窗口应用程序是否启动标志位
    while (bMore)
    {
        ::wsprintf(temp, "%s", pe32.szExeFile);
         HANDLE hProcess;
         if (!::strcmp(temp, szFileName)) //判断是否为窗口进程
           {
              g_hProcess = ::OpenProcess(PROCESS_ALL_ACCESS, false,
(DWORD)pe32.th32ProcessID);
              flag = 1;
              break;
            }
        bMore = ::Process32Next(hProcessSnap, &pe32);//获得其他进程信息
    }
    // 不要忘记清除掉snapshot对象
    ::CloseHandle(hProcessSnap);
    if (flag == 0)//重新启动进程
    {
        STARTUPINFO si = { sizeof(si) };
        PROCESS_INFORMATION pi;
        ::CreateProcess(NULL, szFileName, NULL, NULL, FALSE,
            CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
        // 关闭线程句柄，既然我们不使用它
        ::CloseHandle(pi.hThread);
```

```
        g_hProcess = pi.hProcess;
    }
```

备注：头文件涉及“tlhelp32.h”，“windows.h”

5. 多线程同步实例

1）采用SDK或mfc或控制台创建两个应用程序。

2）然后采用线程同步事件完成多线程同步

3）然后编写源代码

4）编译运行执行，！若有错修 再编译运行执行，并修改相关特性.

四、备用函数

（1）进程创建

CreateProcess(

LPCSTR lpApplicationName ,//是应用程序的名称。

LPSTR lpCommandLine , //是命令行参数

LPSECURITY_ATTRIBUTES lpProcessAttributes ,//是进程的属性

LPSECURITY_ATTRIBUTES lpThreadAttributes ,//是线程的属性,

BOOL bInheritHandles ,//是否继承父进程的属性

DWORD dwCreationFlags ,//是创建标志

LPVOID lpEnvironment ,//是环境变量,

LPCSTR lpCurrentDirectory ,//是当前目录

LPSTARTUPINFOA lpStartupInfo ,//是传给新进程的信息

LPPROCESS_INFORMATION lpProcessInformation ,//是进程返回的信息

);

（2）进程终止

VOID ExitProcess 终止自身进程

(

UINT uExitCode //线程退出码

)

BOOL TerminateProcess 终止自身进程，还可以终止其他进程

(

HANDLE hProcess,

uExitCode //线程退出码

)

（3）进程通信 管道相关函数

1）PostMessage

2）FindWindow

(4) 读写进程内地址的相关内容函数

ReadProcessMemory（）函数

```
BOOL WINAPI ReadProcessMemory(
  __in   HANDLE hProcess,
```

```
  __in   LPCVOID lpBaseAddress,
  __out  LPVOID lpBuffer,
  __in   SIZE_T nSize,
  __out  SIZE_T *lpNumberOfBytesRead
);
```

第一个参数hProcess为进程的句柄，第二个参数lpBaseAddress为要读取内容的基地址，当然你事先要了解你要读取内容的基地址。第三个参数lpBuffer为接收所读取内容的基地址。第四个参数为要读取内容的大小，以字节为单位。最后一个参数lpNumberOfBytesRead为接收读取内容的buffer中收到的字节数。一般都设为NULL，这个该参数将被忽略掉。

使用ReadProcessMemory（）函数，可以获得该进程内存空间中的信息，或是用于监测进程的执行情况，或是将进程内的数据备份，然后调用writeProcessMemory（）进行修改，必要时再还原该进程的数据。

```
BOOL WINAPI WriteProcessMemory(
  __in   HANDLE hProcess,
  __in   LPVOID lpBaseAddress,
  __in   LPCVOID lpBuffer,
  __in   SIZE_T nSize,
  __out  SIZE_T *lpNumberOfBytesWritten
);
```

该函数与readProcessMemory的参数基本相同，只不过第三个参数lpBuffer为要写入进程的内容

1）创建线程函数

HANDLE CreateThread (
LPSECURITY_ATTRIBUTES lpThreadAttributes,
SIZE_T dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter,
DWORD dwCreationFlags,
LPDWORD lpThreadId
);
unsigned long _beginthreadex(
void *security, //安全属性
unsigned stack_size, //堆栈大小
unsigned (__stdcall *start_address)(void *), //回调函数
void *arglist,unsigned initflag, //创建标志
unsigned* thrdaddr //线程ID
);

2）结束线程函数

VOID ExitThread(DWORD dwExitCode);
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);
void _endthreadex(unsigned status);

3）临界区相关api

初始化临界区

VOID WINAPI InitializeCriticalSection(
LPCRITICAL_SECTION lpCriticalSection
);

删除临界区

VOID WINAPI DeleteCriticalSection(
LPCRITICAL_SECTION lpCriticalSection

);

进入临界区

VOID WINAPI EnterCriticalSection(
LPCRITICAL_SECTION lpCriticalSection
);

离开临界区

VOID WINAPI LeaveCriticalSection(
LPCRITICAL_SECTION lpCriticalSection
);

1）等待相关函数

等待单个内核对象触发

DWORD WatiForSingleObject(
HANDLE hObject,
DWORD dwMilliseconds);

等待多个内核对象

DWORD WaitForMultipleObjects（
DWORD dwCount,
CONST HANDLE*pbObject,
BOOL bWaitAll,
DWORD dwMilliseconds);

dwCount指定等待内核对象的数量。

pbObject指向一个存储所有需要等待内核对象的数组。

bWaitAll指定是否等待所有内核对象都触发。当其为true时，只有当所有内核对象都变成触发态时函数返回。否则，只要有一个内核对象变成触发态等待函数就返回。

dwMillisecond是指定等待时间与WaitForSingleObject相同。

bWaitAll为true和false时函数返回值意义不同。

当为true时，函数返回时所有对象都被触发。这与WaitForSingleObject的返回值意义相同。

当为false时，返回值将会标识那个对象被触发从而导致等待函数返回

5）事件内核对象相关api函数

HANDLE CreateEvent(
LPSECURITY_ATTRIBUTES lpEventAttributes,
BOOL bManualReset,
BOOL bInitialState,
LPTSTR lpName
);

1. 第一个参数同CreateThread类似，也是安全级别相关，通常被被设置为NULL，以获得默认的安全级别。
2. 第二个参数是个布尔值，它能够告诉系统是创建一个人工重置的事件（TRUE）还是创建一个自动重置的事件（ FALSE）。
3. 第三个参数也是布尔值，用于指明该事件是要初始化为已通知状态（TRUE）还是未通知状态（FALSE）。
4. 第四个参数是一个字符串，用于标示这个事件的名字。

一般设置为未通知状态，并由SetEvent设置为已 通知状态。当然也可以反着做，CreateEvent时设置为已通知状态，然后由ResetEvent设置为未通知状态

BOOL SetEven(HANDLE hEvent);
BOOL ResetEven(HANDLE hEvent);

6）互斥量相关api函数

```
HANDLE CreateMutex(
        LPSECURITY_ATTRIBUTES lpMutexAttributes,
        BOOL bInitialOwner,
        LPCTSTR lpName
);
```

如果线程ID为0，那么该互斥量不为任何线程所占用。处于触发状态。

如果线程ID非为0，此时有一个线程已经占有了该互斥量，它处于未触发状态

ReleaseMutex(hMutex); 释放互斥量

7）信号量内核对象相关函数

```
HANDLE CreateSemaphore (
PSECURITY_ATTRIBUTE psa,
LONG lInitialCount,
LONG lMaximumCount,
PCTSTR pszName
);
```

第一个参数表示安全控制，一般直接传入NULL。

第二个参数表示初始资源数量。

第三个参数表示最大并发数量。

第四个参数表示信号量的名称，传入NULL表示匿名信号量。

释放信号量内核对象函数

```
BOOL WINAPI ReleaseSemaphore(
HANDLE hSemaphore,
LONG lReleaseCount,
LPLONG lpPreviousCount
);
```

第一个参数是信号量的句柄。

第二个参数表示增加个数，必须大于0且不超过最大资源数量。

第三个参数可以用来传出先前的资源计数，设为NULL表示不需要传出。

注意：**当前资源数量大于0，表示信号量处于触发，等于0表示资源已经耗尽故信号量处于未触发。**在对信号量调用等待函数时，等待函数会检查信号量的当前资源计数，如果大于0(即信号量处于触发状态)，减1后返回让调用线程继续执行。一个线程可以多次调用等待函数来减小信号量。

实验项目名称：Windows内存、虚拟内存和内存映射文件

实验项目性质：

所属课程名称：Windows核心编程

实验计划学时：4学时

一、 实验目的

1、掌握windows内存体系结构，理解进程虚拟地址，虚拟地址空间分区，地址空间中区域，给区域调拨物理存储器的概念和基本过程

2、学会使用VirtualAlloc，VirtualFree，以及VirtualQuery虚拟内存相关函数使用

3、 掌握内存映射文件基本概念，实现步骤和相关api函数应用，采用内存映射文件进行数据共享和文件分割。

二、实验内容和要求

1、采用虚拟内存函数VirtualAlloc，以及虚拟内存相关函数使用，分配一个60kb地址预订和物理调拨，并在分配地址空间写入数据；采用VirtualQuery进行查询虚拟内存情况，并然后调用VirtualFree释放虚拟内存。

2、在给得课件代码基础上，采用内存映射文件实现两个进程数据共享，写进程，通过键盘输入数据，然后写入数据。然后读进程，则把写进程写入内存的数据读取出来。

3、采用内存文件映射，对一个大文件（大于2G的文件）进行读，并把该文件分割成多个子文件，保存在磁盘中。

三、实验步骤

1、编写虚拟内存分配和调拨，查询和释放程序

1）采用SDK或mfc或控制台创建一个应用程序。

2）然后添加代码，采用虚拟内存函数VirtualAlloc，以及虚拟内存相关函数使用，分配一个64kb地址预订和物理调拨，并将字符串“202200000”**（各自学号）**赋值到虚拟内存；采用VirtualQuery进行查询虚拟内存情况，并然后调用VirtualFree释放虚拟内存。

2、完成两进程的数据共享，

首先写进程端内存映射文件的过程,具体步骤如下：

1）调用CreateFileMapping(INVALID_HANDLE_VALUE, …….)完成文件映射内核对象

2）调用MapViewofFile函数将内存映射文件进程地址空间

3）通过MapViewofFile函数返回值，地址指针值获得对共享内存块的操作，写入需要写入的数据

4）当读进程读取数据之后，然后调用UnmapViewofFile()函数释放进程空间地址，然后调用CloseHandle函数释放文件映射对象和文件对象

然后读进程则完成数据读取和显示，具体步骤如下：

1） 调用OpenFileMapping函数打开写进程的文件映射对象

2） 调用MapViewofFile函数映射进程地址空间，并获得内存的指针

3） 利用MapViewofFile函数获得指针，对共享内存块数据进行读操作，并输出显示

4）当读取完进程数据，然后调用UnmapViewofFile()函数释放进程空间地址，然后调用CloseHandle函数释放文件映射对象和文件对象

3、实现对大文件读取和分割，具体步骤如下：

1） 如下是分割大文件，读取大文件代码

```
//读大文件
SYSTEM_INFO sinf;
GetSystemInfo(&sinf);
HANDLE
hFile=CreateFile(TEXT("d://huge.txt"),GENERIC_WRITE|GENERIC_READ,0,NULL,O
PEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);//打开文件
DWORD dwFileSizeHigh;
_int64 qwFileSize=GetFileSize(hFile,&dwFileSizeHigh);//获得文件大小
qwFileSize+=(((_int64)dwFileSizeHigh)<<32);//将高32位加到文件低32位上
HANDLE
hFileMap=CreateFileMapping(hFile,NULL,PAGE_READWRITE,0,0,NULL);
//创建一个文件内核对象，大小默认为文件大小
CloseHandle(hFile);
_int64 qwFileOffset=0;//每次映射文件大小，初始值为0
while(qwFileSize>0){
DWORD dwBytesInBlock=sinf.dwAllocationGranularity;//预定空间的分配
if(qwFileSize<sinf.dwAllocationGranularity)
dwBytesInBlock=(DWORD)qwFileSize;
PCHAR pbFile=(PCHAR)MapViewOfFile(hFileMap,
FILE_MAP_WRITE,
(DWORD)(qwFileOffset>>32),//高32位
(DWORD)(qwFileOffset&0xFFFFFFFF),//低32位
dwBytesInBlock);
//为文件的数据预定一块地址空间区域并将文件的数据作为物理存储器调拨给区域
cout<<"content: "<<pbFile<<endl;
UnmapViewOfFile(pbFile);//从进程空间撤销对文件数据的关联
qwFileOffset+=dwBytesInBlock;
qwFileSize-=dwBytesInBlock;
}
CloseHandle(hFileMap);
```

2）分割之后，采用内存映射文件写入到子文件，代码类似

四、主要函数

1)虚拟内存相关函数

通过调用VirtualAlloc函数，可以在进程的地址空间中保留一个区域：

```
LPVOID VirtualAlloc(
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD flAllocationType,
    DWORD flProtect );
```

第一个参数pvAddress包含一个内存地址，用于设定想让系统将地址空间保留在什么地方。在大多数情况下，你为该参数传递MULL。

第三个参数是预定地址空间还是分配物理存储器 MEM_RESERVE（预订），MEM_COMMIT(物理）

如果VirtualAlloc函数能够满足你的要求，那么它就返回一个值，指明保留区域的基地址。如果传递一个特定的地址作为VirtualAlloc的pvAddress 参数，那么该返回值与传递给Vi rtualAlloc的值相同，并被圆整为（如果需要的话）64KB边界值。

释放虚拟内存函数

```
BOOL VirtualFree(
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD dwFreeType );
```

查询虚拟内存函数

```
DWORD VirtualQueryEx(
    HANDLE hProcess,
    LPCVOID lpAddress,
    PMEMORY_BASIC_INFORMATION lpBuffer,
    DWORD dwLength
  );
```

参数:

hProcess 进程句柄。

lpAddress 查询内存的地址。

lpBuffer 指向MEMORY_BASIC_INFORMATION结构的指针，用于接收内存信息。

dwLength MEMORY_BASIC_INFORMATION结构的大小。

返回值:

函数写入lpBuffer的字节数，如果不等于sizeof(MEMORY_BASIC_INFORMATION)表示失败

2）内存映射文件相关API函数

a) 打开文件函数CreateFile，该函数可以用来创建文件、邮槽、管道、目录

```
HANDLE CreateFile(
```

LPCTSTR lpFileName, // 文件名、邮槽名、管道名等

DWORD dwDesiredAccess, // 进入模式 （GENER_READ, GENER_WRITE)

DWORD dwShareMode, // 共享模式 设置为0表示不共享

LPSECURITY_ATTRIBUTES lpSecurityAttributes, // SD ,一般为NULL

DWORD dwCreationDisposition, // 怎么去创建 OPEN_EXISTING 已存在的

DWORD dwFlagsAndAttributes, // 文件属性 FILE_ATTRIBUTE_NORMAL

HANDLE hTemplateFile // 模板文件句柄 一般设置为NULL);

第二个参数设置

值	含义
0	不能读取或写入文件的内容。当只想获得文件的属性时，请设定0
GENERIC_READ	可以从文件中读取数据
GENERIC_WRITE	可以将数据写入文件
GENERIC_READ |GENERIC_WRITE	可以从文件中读取数据，也可以将数据写入文件

第三个参数设置

值	含义
0	打开文件的任何尝试均将失败
FILE_SHARE_READ	使用GENERIC_WRITE打开文件的其他尝试将会失败
FILE_SHARE_WRITE	使用GENERIC_READ打开文件的其他尝试将会失败
FILE_SHARE_READ FILE_SHARE_WRITE|	打开文件的其他尝试将会取得成功

b)创建文件映射对象函数

调用CreateFileMapping函数告诉系统，文件映射对象需要多少物理存储器。

```
HANDLE CreateFileMapping(
  HANDLE hFile,
  PSECURITY_ATTRIBUTES psa,
  DWORD fdwProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  PCTSTR pszName);
```

主要参数设置：

第一个参数：hFile用于标识你想要映射到进程地址空间中的文件句柄。该句柄由前面调用的CreateFile函数返回。

第二个参数：psa参数是指向文件映射内核对象的SECURITY_ATTRIBUTES结构的指针，通常传递的值是NULL

第三个参数：fdwProtect参数使你能够设定这些保护属性。

保护属性	含义
PAGE_READONLY	当文件映射对象被映射时，可以读取文件的数据。必须已经将GENERIC_READ传递给CreateFile函数
PAGE_READWRITE	当文件映射对象被映射时，可以读取和写入文件的数据。必须已经将GENERIC_READ | GENERIC_WRITE传递给Creat eFile
PAGE_WRITECOPY	当文件映射对象被映射时，可以读取和写入文件的数据。如果写入数据，会导致页面的私有拷贝得以创建。必须已经将GENERIC_READ或GENERIC_WRITE传递给CreateFile

第四和五个参数：dwMaximumSizeHigh和dwMaximumSizeLow这两个参数将告诉系统该文件的最大字节数

最后一个参数是pszName： 它是个以0结尾的字符串，用于给该文件映射对象赋予一个名字。该名字用于与其他进程共享文件映射对象

c) 打开已经创建文件映射内核对象函数

```
HANDLE OpenFileMapping(
    DWORDdwDesiredAcess, // 指定保护类型
    BOOL bInheritHandle,// 返回的句柄是否可继承
    LPCTSTR lpName     // 创建对象时使用的名字
);
```

d) 将文件映射到进程地址空间

将文件的数据作为映射到该区域的物理存储器进行提交。

```
PVOID MapViewOfFile(
  HANDLE hFileMappingObject,
  DWORD dwDesiredAccess,
  DWORD dwFileOffsetHigh,
  DWORD dwFileOffsetLow,
  SIZE_T dwNumberOfBytesToMap);
```

第一个参数： hFileMappingObject用于标识文件映射对象的句柄，该句柄是前面调用CreateFileMapping或OpenFileMapping函数返回的。

第二个参数：dwDesiredAccess用于标识如何访问该数据。可以设定下表所列的4个值中的一个

值	含义
FILE_MAP_WRITE	可以读取和写入文件数据。CreateFileMapping函数必须通过传递PAGE_READWRITE标志来调用
FILE_MAP_READ	可以读取文件数据。CreateFileMapping函数可以通过传递下列任何一个保护属性来调用：PAGE_READONLY、PAGE_READWRITE或PAGE_WRITECOPY
FILE_MAP_ALL_ACCESS	与FILE_MAP_WRITE相同
FILE_MAP_COPY	可以读取和写入文件数据。如果写入文件数据，可以创建一个页面的私有拷贝。在Windows 2000中，CreateileMapping函数可以用PAGE_READONLY、PAGE_READWRITE或PAGE_WRITECOPY等保护属性中的任何一个来调用。在Windows 98中，CreateFileMapping必须用PAGE_WRITECOPY来调用

第三四个参数：dwFileOfsetHigh和dwFileOfsetLow参数。指定哪个字节应该作为视图中的第一个字节来映射。

第五个参数：dwNumberOfBytesToMap有多少字节要映射到地址空间。如果设定的值是0，那么系统将设法把从文件中的指定位移开始到整个文件的结尾的视图映射到地址空间

e)将进程地址空间撤销文件映射数据

UnmapViewofFile(PVOID pvBaseAddress)

pvBaseAddress 是指定调用的基地，由MapViewOfFile函数返回

f)释放内核对象

CloseHandle（HANDLE handle）

实验项目名称：DLL编程

实验项目性质：

所属课程名称：Windows核心编程

实验计划学时：4学时

一、 实验目的

1、 掌握动态链接的两种不同方法，以及动态链接的不同加载方式基本原理

2、 采用api函数加载动态链接，LoadLibrary（），以及FreeLibrary（），GetProcAddress（）函数的加载动态链接方式

3、 采用MFC和SDK方式编写动态链接，了解动态链接入口函数DllMain

4、 了解Hook编程基本概念，原理。采用动态链接中Hook编程

二、实验内容和要求

（1）编写一个win32动态链接库，实现两个数简单加法和减法，乘法，除法，然后采用显式和隐式两种方式调用。调用界面采用MFC或者wind32窗口程序实现。

（2）编写一个钩子程序KeyBoardProc，在发生键盘事件时，记录用户的按键消息写入到文件中，并且所有窗口不响应键盘事件（除F1键外）。

三、 实验步骤

（1）编写一个win32动态链接库，实现两个数简单加法和减法，乘法，乘法，然后采用显式和隐式两种方式调用。调用界面采用MFC或者wind32窗口程序实现。具体步骤如下：

1） File—New—Project—Win32 Dynamic Link Library

2) 然后添加头文件****.h和源文件****.cpp，头文件.h中添加类似代码：

```
#ifdef DLL1_API
#else
#define DLL1_API extern "C" _declspec(dllimport)
#endif

DLL1_API int _stdcall Add(int a,int b);
DLL1_API int _stdcall Subtract(int a,int b);
DLL1_API int _stdcall Multiply(int a,int b);
DLL1_API int _stdcall Division(int a,int b);
```

在源文件中添加如下代码：

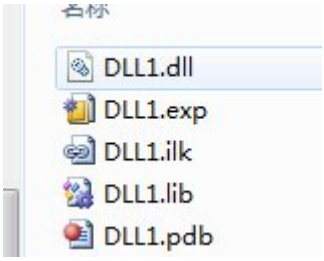
```
#define DLL1_API extern "C" _declspec(dllexport)
#include "Dll1.h"
#include <Windows.h>
#include <stdio.h>

int _stdcall Add(int a,int b)
{
      return a+b;
}

int _stdcall Subtract(int a,int b)
{
```

```
    return a-b;
}
int _stdcall Multiply(int a,int b)
{
    return a*b;
}
int _stdcall Division(int a,int b)
{
    if(a!=0)
        return b/a;
}
```

3) 然后编译链接，产生DLL文件盒Lib文件，如下图所示



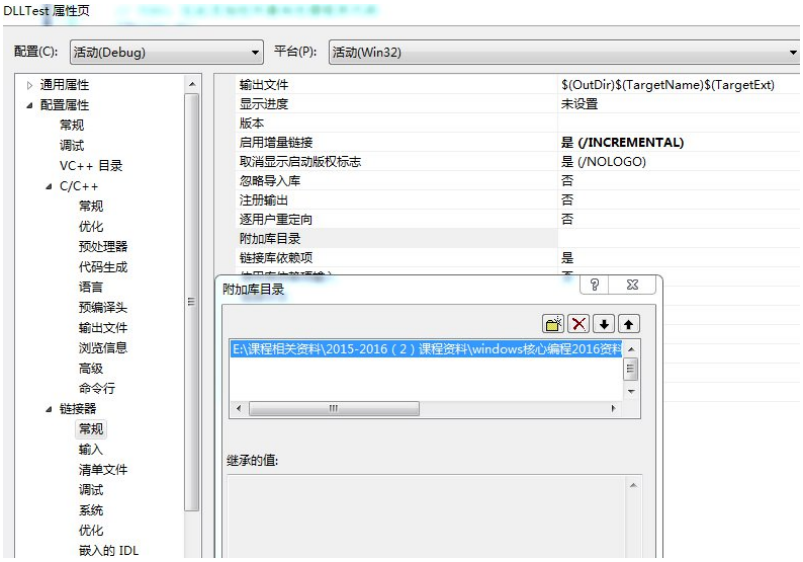
4) 然后采用win32或者MFC程序中，调用动态链接库，显式和隐式两种方式调用，完成调用。

（a）隐式加载动态链接库步骤

第一步 ：首先把动态链接库.lib, .h，.dll拷贝到调用目录下

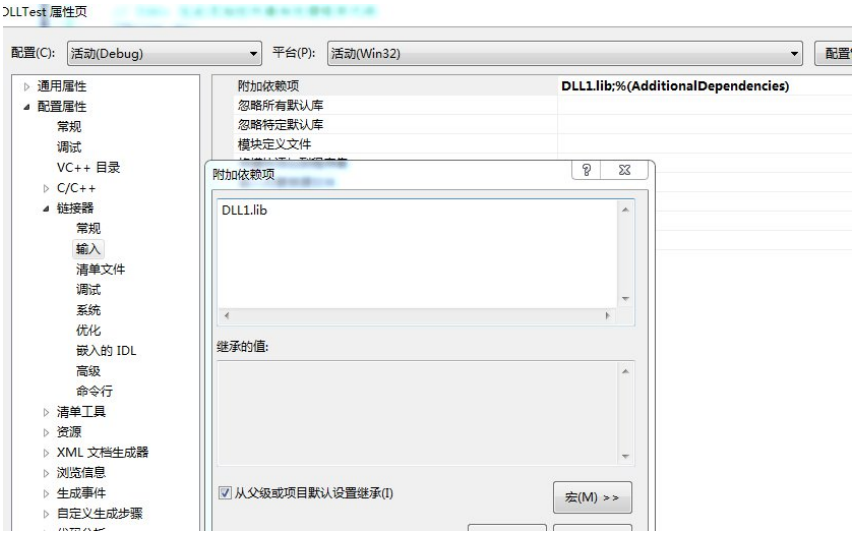
第二步：配置头文件路径：选择项目->属性->C/C++->常规->附加包含目录：设置头文件路径。

第三步：配置DLL和LIB文件路径：选择项目->属性->连接器->常规->附加库目录：设置 DLL和LIB文件路径。



第四步：加载Lib文件：

选择项目->属性->连接器->输入->附加依赖项：输入要加载的Lib文件名（加载多个Lib文件时，以回车为分隔符)。



（b） 显示调用方式，例如调用的动态链接库名字为“MyDLL.dll”，调用函数为sub函数，则动态调用如下所示：

```
HINSTANCE hInst;
hInst=LoadLibrary(_T("MyDLL.dll "));
typedef int (*ADDPROC)(int a,int b);
ADDPROC Sub=(ADDPROC)GetProcAddress(hInst,"sub");
CString str;
str.Format(_T("5-10=%d"),Sub(5,10));
MessageBox(str);
```

（2）编写一个钩子程序KeyBoardProc，在发生键盘事件时自动调用，用于记录用户的按键。（参见11.3 P238页）

四、主要函数

1、动态调用链接库函数

（1）加载动态链接库

```
HMODULE LoadLibrary ( LPCTSTR lpFileName // 链接文件名 );
```

（2）获取函数地址指针

```
FARPROC GetProcAddress(
        HMODULE hModule, // 模块句柄
        LPCSTR lpProcName // 函数名称 );
```

（3）释放动态链接库

```
BOOL FreeLibrary( HMODULE hModule // 链接模块句柄 );
```

钩子编程相关函数

2 钩子编程相关函数

（1）安装钩子

```
SetWindowsHookEx(
  idHook: Integer;    {钩子类型}
  lpfn: TFNHookProc; {函数指针}
  hmod: HINST;        {包含钩子函数的模块(EXE、DLL)句柄; 一般是 HInstance; 如果是当前线程这里可以是 0}
  dwThreadId: DWORD   {关联的线程; 可用 GetCurrentThreadId 获取当前线程; 0 表示是系统级钩子}
): HHOOK;             {返回钩子的句柄; 0 表示失败}
```

```
//钩子类型 idHook 选项:
WH_MSGFILTER       = -1; {线程级；截获用户与控件交互的消息}
WH_JOURNALRECORD   = 0;  {系统级；记录所有消息队列从消息队列送出的输入消息，在消息从队列中清除时发生；可用于宏记录}
WH_JOURNALPLAYBACK = 1;  {系统级；回放由 WH_JOURNALRECORD 记录的消息，也就是将这些消息重新送入消息队列}
WH_KEYBOARD        = 2;  {系统级或线程级；截获键盘消息}
WH_GETMESSAGE      = 3;  {系统级或线程级；截获从消息队列送出的消息}
WH_CALLWNDPROC     = 4;  {系统级或线程级；截获发送到目标窗口的消息，在 SendMessage 调用时发生}
WH_CBT             = 5;  {系统级或线程级；截获系统基本消息，譬如：窗口的创建、激活、关闭、最大最小化、移动等等}
WH_SYSMSGFILTER    = 6;  {系统级；截获系统范围内用户与控件交互的消息}
WH_MOUSE           = 7;  {系统级或线程级；截获鼠标消息}
WH_HARDWARE        = 8;  {系统级或线程级；截获非标准硬件(非鼠标、键盘)的消息}
WH_DEBUG           = 9;  {系统级或线程级；在其他钩子调用前调用，用于调试钩子}
WH_SHELL           = 10; {系统级或线程级；截获发向外壳应用程序的消息}
WH_FOREGROUNDIDLE  = 11; {系统级或线程级；在程序前台线程空闲时调用}
WH_CALLWNDPROCRET  = 12; {系统级或线程级；截获目标窗口处理完毕的消息，在 SendMessage 调用后发生}
```

（2）调用下一个钩子

```
LRESULT CallNextHookEx(

  HHOOK hhk,

  int nCode,

  WPARAM wParam,

  LPARAM lParam

);
```

(3)释放钩子

BOOL UnhookWindowsHookEx(HHOOK hhk);

（4）钩子过程函数，例如鼠标钩子过程函数

LRESULT CALLBACK MouseProc(int nCode, WPARAM wParam, LPARAM lParam);

nCode:指示钩子过程如何处理当前的消息，如果钩子是鼠标消息钩子的话，有两种含义：

nCode 取值	说　明
HC_ACTION	表明参数 wParam 和 lParam 包含了关于鼠标消息的信息
HC_NOREMOVE	表明参数 wParam 和 lParam 包含了关于鼠标消息的信息，而且此鼠标消息尚未从消息队列中删除（应用程序调用 PeekMessage 函数并设置了 PM_NOREMOVE 标志）

wParam:指示鼠标消息标志。

lParam:指向MOUSEHOOKSTRUCT结构体指针。以下是MOUSEHOOKSTRUCT结构体，包含着鼠标消息。

```
typedef struct {
  POINT pt;//光标包含x,y，是屏幕坐标。
  HWND hwnd;//目标窗口句柄
  UINT wHitTestCode;
  ULONG_PTR dwExtraInfo;
} MOUSEHOOKSTRUCT, *PMOUSEHOOKSTRUCT;
```

通过这三个参数，可对消息进行分析处理。

返回值:若nCode的值小于0,则此挂钩处理过程必须返回CallNextHookEx 的返回值.若nCode的值大于或等于0,并且此挂钩处理过程未对该消息进行处理,则调用函数CallNextHookEx 并返回其返回值是被推荐的.否则,其他安装了WH_MOUSE挂钩的应用程

序将无法收到此挂钩通知,并可能由此导致错误的行为.若此挂钩处理过程处理了此消息,它应返回一个非零值以避免系统再将此消息传送给目标窗口处理过程