2020/07/31_Windows编程_第11课_线程间跨进程访问的使用、代码注入

笔记本: Windows编程
创建时间: 2020/7/31 星期五 13:59
作者: ileemi

使用事件对象跨进程访问

> 进程本身没有消息循环的时候，进程间通讯需要写消息循环（比较麻烦）。
> 使用事件对象 `CreateEvent` 手动传递信号，使 `A` 进程 通知 `B` 进程。

哪个进程等待别的进程给自己发送消息就调用 `WaitForSingleObject` 函数进行等待。

使用事件对象可以使两个进程间进行相互通知。

主进程创建的事件对象通过对象名，都可以打开这个事件对象，通过 `SetEvent` 所有打开这个事件对象的进程通过等待消息（`WaitForSingleObject`）都可以收到消息。

将不同的同步对象（事件对象，信号量，互斥体）放入到一个数组中，使用 `WaitForMultipleObjectsEx` 等待指定数量的信号。

以同步的方式可以通过句柄去通讯另外一个进程（不能处理多个进程，只能一个进程通知一个进程）。

代码示例:

```
// A.cpp : 此文件包含 "main" 函数。程序执行将在此处开始并结束。
//
#include <stdio.h>
#include <Windows.h>
int main()
{
    // 创建事件对象
    HANDLE hEvent = CreateEvent(NULL, TRUE, FALSE, "TestProcess");
    if (NULL == hEvent)
    {
        printf("[A]: CreateEvent TestProcess Error\r\n");
    }
```

```
    printf("[A]: CreateEvent TestProcess Ok\r\n");

    // 等待B进程发送消息
    if (WaitForSingleObject(hEvent, INFINITE) == WAIT_FAILED);
    printf("[A]: Data OK\r\n");

    // 接收 B 进程消息后，将指定的事件对象设置为无信号状态
    ResetEvent(hEvent);

    // 关闭句柄
    if (hEvent != NULL)
    {
        CloseHandle(hEvent);
    }
    // 打开事件对象
    HANDLE hEvent2 = OpenEvent(EVENT_ALL_ACCESS,
        FALSE, "TestProcess2");
    if (NULL == hEvent2)
    {
        printf("[A]: OpenEvent TestProcess2 Error\r\n");
    }
    printf("[A]: OpenEvent TestProcess2 Ok\r\n");

    // 将指定的事件对象设置为有信号状态
    SetEvent(hEvent2);
    // 关闭句柄
    if (hEvent2 != NULL)
    {
        CloseHandle(hEvent2);
    }
    return 0;
}
```

B 进程:

```
// B 进程
#include <stdio.h>
#include <Windows.h>

int main()
{
  // 打开事件对象
  HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, "TestProcess");
  if (NULL == hEvent)
  {
    printf("OpenEvent Error\r\n");
  }
```

```
    printf("OpenEvent Ok\r\n");

    // 将指定的事件对象设置为有信号状态
    SetEvent(hEvent);

    // 关闭句柄
    if (hEvent != NULL)
    {
      CloseHandle(hEvent);
    }

    // 向A进程通讯
    // 创建事件对象
    HANDLE hEvent2 = CreateEvent(NULL, TRUE, FALSE, "TestProcess2");
    if (NULL == hEvent2)
    {
      printf("CreateEvent Error\r\n");
    }
    printf("CreateEvent Ok\r\n");

    // 等待B进程发送消息
    if (WaitForSingleObject(hEvent2, INFINITE) == WAIT_FAILED);
    printf("Data OK\r\n");
    // B 进程和接受消息后，将将指定的事件对象设置为无信号状态
    ResetEvent(hEvent2);

    // 关闭句柄
    if (hEvent2 != NULL)
    {
      CloseHandle(hEvent2);
    }
    return 0;
}
```

多线程同步问题

在多线程之前，C语言做了库，其内部也存在同步问题。

死锁

Windows 的库函数也提供了多线程版本在VS项目项目属性中，C/C++ --> 代码生成 --> 运行库（分 Debug/Release 动态、静态库） 中可以查看（高版本已经不分库的所线程（多线程库的效率问题已经解决），低版本的VS可以更改对应的选项）。

解决多线程库的效率问题，代码示例：

```
#include <stdio.h>
#include <Windows.h>

#define MAX_NUM 100000000
int g_nNum[2] = { 0 };
// 工作线程
DWORD WINAPI WorkThread(LPVOID lpParma)
{
    for (int i = 0; i < MAX_NUM; i++)
    {
        g_nNum[lpParma]++;
    }
    return 0;
}
int main()
{
    HANDLE hThread[2];
    for (int i = 0; i < (sizeof(hThread) / sizeof(hThread[0])); i

    {
        hThread[i] = CreateThread(NULL, 0, &WorkThread, (LPVOID)i, 0,

    }

    // 等待线程全部结束
    WaitForMultipleObjects((sizeof(hThread) / sizeof(hThread[0])), hThrea

    printf("g_nNum = %d\r\n", g_nNum[0] + g_nNum[1]);

    return 0;
}
```

C库函数解决性能问题采取的办法就是，每个线程操控一个全局变量，相互独立的。不需要考虑同步问题。

Release 版本的库函数代码会做优化处理，Debug下库函数代码可以跟进去看代码实现。

一个线程同步多个对象的时候会出现死锁（出现的概率较低），两个锁互相在等。锁1等锁2，锁2等锁1。出现的概率较少的时候，是因为现在的CPU处理速度太快了。

出现在两个锁中，定位也比较难。发生的概率比较低。

解决死锁的办法：

1. 将需要同步的对象同时进行同步，但是效率比较低。

2. 两个锁两个线程的时候，调整要操作的同步对象的顺序，第1个线程中在第一个锁中操作1，在第二个锁中操作2，第二个线程中在第二个锁中操作1，在第一个锁中操作2（这样做和一个锁并无差别，做法多余）。
3. 不使用临界区（使用WaitForSingleObject 在指定的时间内没有拿到锁，就说明死锁，可以放弃），逻辑上进行调整。
4. 单独写一个程序，用一个线程来检查锁的状态，或者调用API判断哪个线程被一直挂起，一直挂起就可以判定为死锁（线程一直未分配到时间片，就将其结束，重新开启运行）。

> 当要操作两个以上的同步对象解决一个同步问题的时候就需要注意死锁问题。

代码示例：

```
#include <stdio.h>
#include <windows.h>
int g_num1 = 0;
int g_num2 = 0;
#define MAX_NUM 100000
// 临界区
class CCritical
{
public:
    CCritical()
    {
        InitializeCriticalSection(&m_cs);
    }
    ~CCritical()
    {
        DeleteCriticalSection(&m_cs);
    }
    void lock()
    {
        EnterCriticalSection(&m_cs);
    }
    void un_lock()
    {
        LeaveCriticalSection(&m_cs);
    }
private:
    CRITICAL_SECTION m_cs;
};
CCritical g_Lock1;
CCritical g_Lock2;
//死锁 获取线程等待链 WCT OpenThreadWaitChainSession
DWORD WINAPI WorkThread1(LPVOID lpParam)
{
    for (int i = 0; i < MAX_NUM; i++)
```

```
    {
        g_Lock1.lock();
        g_num1++;
        //Sleep(1);
            g_Lock2.lock();
            g_num2++;
            g_Lock2.un_lock();
        g_Lock1.un_lock();
    }
    return  0;
}
DWORD  WINAPI  WorkThread2(LPVOID  lpParam)
{
    for  (int  i  =  0;  i  <  MAX_NUM;  i++)
    {
        g_Lock2.lock();
        g_num2++;
            g_Lock1.lock();
            g_num1++;
            g_Lock1.un_lock();
        g_Lock2.un_lock();
    }
    return  0;
}
int  main()
{
    HANDLE  hThreads[2];
    hThreads[0]  =  CreateThread(
            NULL,
            0,
            WorkThread1,
            0,
            0,
            NULL);
    hThreads[1]  =  CreateThread(
        NULL,
        0,
        WorkThread2,// 出现死锁
        0,
        0,
        NULL);
    WaitForMultipleObjects(sizeof(hThreads)  /  sizeof(hThreads[0]),  hThrea

    printf("work  ok  g_num1 = %d  g_num2 = %d\r\n",  g_num1,  g_num2);
    return  0;
}
```

> 获取线程等待链，遍历锁的状态

代码注入

> 利用线程在别的进程中运行自己想要的代码（将自己的代码注入到别的程序中并运行）。

将自己的代码注入到别的程序中可以为别的程序添加一些新的功能。

> `CreateRemoteThread`：可以为操作系统上任意的进程创建一个线程。
> 函数说明：创建一个在另一个进程的虚拟地址空间中运行的线程。

为操作系统中的进程创建一个线程的步骤：

使用 `CreateRemoteThread` 创建远程线程，需要提前获取到进程的句柄。

1. 使用 `FindWindow` 通过窗口名获取进程的窗口句柄
2. 使用 `GetWindowThreadProcessId` 通过获取到的窗口句柄获取进程 PID
3. 使用 `OpenProcess` 通过获取到的进程PID以所有权限打开对应的进程后，返回打开后进程的句柄值
4. 使用 `CreateRemoteThread` 通过进程的句柄值为对应的进程创建一个线程，返回创建成功的线程句柄

> 为进程创建线程的时候需要给一个回调函数。
> 可以通过 `WriteProcessMemory` 给指定进程指定地址上添加对应的代码，然后把申请到的内存的地址当作 `CreateRemoteThread` 的参数回调函数使用（注意线程回调函数的调用约定）。但是这个代码必须是二进制的。这样做存在二进制代码不好写的问题，修改指定位置上的代码易将程序给搞崩。
> 微软同时又提供了一个可以给对方进程申请内存的API -- `VirtualAllocEx`（函数说明：在指定进程的虚拟地址空间中保留或提交一个内存区域）

转换一下思路，可以先使用 `VirtualAllocEx` 给指定的进程申请一段内存，`VirtualAllocEx` 函数会返回申请到的内存首地址，然后再通过 `WriteProcessMemory` 再申请到的内存中写自己的代码，最后将申请到的内存地址当作 `CreateRemoteThread` 线程的回调函数来使用。

远程线程注入Dll

将自己的 dll 注入到对方进程中，在注入的时候需要将dll的路径加载到对方进程中。

将 dll 路径加载到对方进程中的时机在向目标进程申请内存的时候，dll的路径就应该加载/写入到目标进程中去。

远程注入代码的效率要比dll劫持的效率高（Dll劫持，dll需要放入到目标进程的路径下，dll隐秘性不高，远程注入代码对dll的路径没有要求，dll隐秘性较高）

给目标进程添加菜单

给添加的菜单响应消息
修改目标进程的过程函数，并保存旧的过程函数
SetWindowsLong

注入思路：先在目标进程的内存空间里开辟一块新地方，往新地方里面写入DLL的路径，再创建远程线程使用 LoadLibrary 函数（其参数个和线程回调函数一样），并在刚才开辟的新地方中读取DLL路径，进而加载我们自己写的DLL。

实现步骤

- 使用进程PID打开进程,获得句柄
- 使用进程句柄申请内存空间
- 把 dll 路径写入到申请的内存中去
- 创建远程线程,调用 LoadLibraryA（注意dll路径的字符格式）
- 释放收尾工作或者卸载dll

代码示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>

int main()
{
  //远程线程注入dll
  // 判断dll是否打开成功，防止多次dll注入(这种操作常用来检查程序多开)
  if (OpenEvent(EVENT_ALL_ACCESS, FALSE, "Text") != NULL)
  {
    printf("Dll Exit\n");
    return 0;
  }

  // 获取窗口句柄
  /*
  参数1：NULL -- 窗口类名
  参数2：目标窗口的标题
  返回值：目标窗口的句柄
  */
  HWND hWnd = FindWindow(NULL, "微信");
  printf("目标窗口的句柄为 %p\r\n", hWnd);

  // 获取目标进程的PID
  /*
  参数1：目标进程的窗口句柄
  参数2：把目标进程的PID存放进去的变量
  返回值：创建窗口的线程的标识符
  */
```

```
  DWORD dwPID = 0;
  DWORD dwRub = GetWindowThreadProcessId(hWnd, &dwPID);
  printf("目标窗口的进程PID为 %d\r\n", dwPID);

  // 为目标进程创建一个线程并注入dll
  //CHAR szDllPath[] = {
"C:\\Users\\ileemi\\Desktop\\Test\\InjectDll.dll" };
  CHAR szDllPath[] = {
"D:\\CR37\\Works\\DayCode\\Inject\\Debug\\InjectDll.dll" };


  // 在目标进程中申请一个空间
/*
获取目标进程句柄
参数1：想要拥有的进程权限（本例为所有能获得的权限）
参数2：表示所得到的进程句柄是否可以被继承
参数3：被打开进程的PID
返回值:指定进程的句柄
*/
  HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwPID);
  printf("OpenProcess hProcess=%p\n", hProcess);
  // 在目标进程的内存里开辟空间
  /*
  参数1：目标进程句柄
  参数2：保留页面的内存地址，一般用NULL自动分配
  参数3：欲分配的内存大小，字节单位
  参数4：MEM_COMMIT：为特定的页面区域分配内存中或磁盘的页面文件中的物理
存储
  参数5：PAGE_EXECUTE_READWRITE 启用页的提交区域的执行、读和写权限
  返回值：执行成功就返回分配内存的首地址，不成功就是NULL
  */
  LPVOID pRemoteAddress = VirtualAllocEx(
    hProcess,
    NULL,
    0x1000,
    MEM_COMMIT,
    PAGE_EXECUTE_READWRITE
  );

  if (pRemoteAddress == NULL)
  {
    return 0;
  }
  printf("VirtualAllocEx lpBuff=%p\n", pRemoteAddress);

  // 把dll的路径写入到目标进程的内存空间中
  DWORD dwWriteSize = 0;
```

```
  // 写一段数据到刚才给指定进程所开辟的内存空间里
  /*
  参数1：OpenProcess返回的进程句柄
  参数2：准备写入的内存首地址
  参数3：指向要写的数据的指针（准备写入的东西）
  参数4：要写入的字节数（东西的长度+\0）
  参数5：返回实际写入的字节
  */
  WriteProcessMemory(hProcess, pRemoteAddress, szDllPath,
sizeof(szDllPath), &dwWriteSize);
  printf("WriteProcessMemory dwBytes=%d\n", dwWriteSize);

  // 创建一个远程线程，让目标进程调用LoadLibrary

  /*
  参数1：该远程线程所属进程的进程句柄
  参数2：一个指向 SECURITY_ATTRIBUTES 结构的指针，该结构指定了线程的安
全属性
  参数3：线程栈初始大小,以字节为单位,如果该值设为0,那么使用系统默认大小
  参数4：在远程进程的地址空间中,该线程的线程函数的起始地址（也就是这个
线程具体要干的活儿）
  参数5：传给线程函数的参数（刚才在内存里开辟的空间里面写入的东西）
  参数6：控制线程创建的标志。0（NULL）表示该线程在创建后立即运行
  参数7：指向接收线程标识符的变量的指针。如果此参数为NULL，则不返回线程
标识符
  返回值：如果函数成功，则返回值是新线程的句柄。如果函数失败，则返回值
为NULL
  */
  HANDLE hThread = CreateRemoteThread(
    hProcess,
    NULL,
    0,
    (LPTHREAD_START_ROUTINE)LoadLibraryA,
    pRemoteAddress,
    NULL,
    NULL
  );
  printf("hThread hThread=%p\n", hThread);

  WaitForSingleObject(hThread, INFINITE); //当句柄所指的线程有信号的时
候，才会返回

  // 释放申请的虚拟内存空间
  /*
  参数1：目标进程的句柄。该句柄必须拥有 PROCESS_VM_OPERATION 权限
  参数2：指向要释放的虚拟内存空间首地址的指针
  参数3：虚拟内存空间的字节数
```

```
    参数4：MEM_DECOMMIT仅标示内存空间不可用，内存页还将存在。
          MEM_RELEASE这种方式很彻底，完全回收。
    */
    CloseHandle(hProcess);
    VirtualFreeEx(hProcess, pRemoteAddress, 0, MEM_DECOMMIT);

    system("pause");
    return 0;
}
```

DLL代码示例：

```
#include <Windows.h>
WNDPROC g_pfnOldWndProc = NULL;
HMODULE g_hModule;

// 对话框过程函数
INT_PTR CALLBACK DialogProc(HWND hwndDlg,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam)
{
    if (uMsg == WM_CLOSE)
    {
        EndDialog(hwndDlg, 0);      // 关闭对话框
        return TRUE;
    }
    return FALSE;
}

// 过程函数
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM
lParam)
{
    // 响应菜单消息
    if (uMsg == WM_COMMAND)
    {
        WORD wID = LOWORD(wParam);
        switch (wID)
        {
        case WM_USER + 1:
            MessageBoxA(NULL, "秒杀", "Test", MB_OK);
            //DialogBox(g_hModule, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
DialogProc);
            return TRUE;
        case WM_USER + 2:
            MessageBoxA(NULL, "帮助：本功能交流使用，禁止使用非法用途",
```

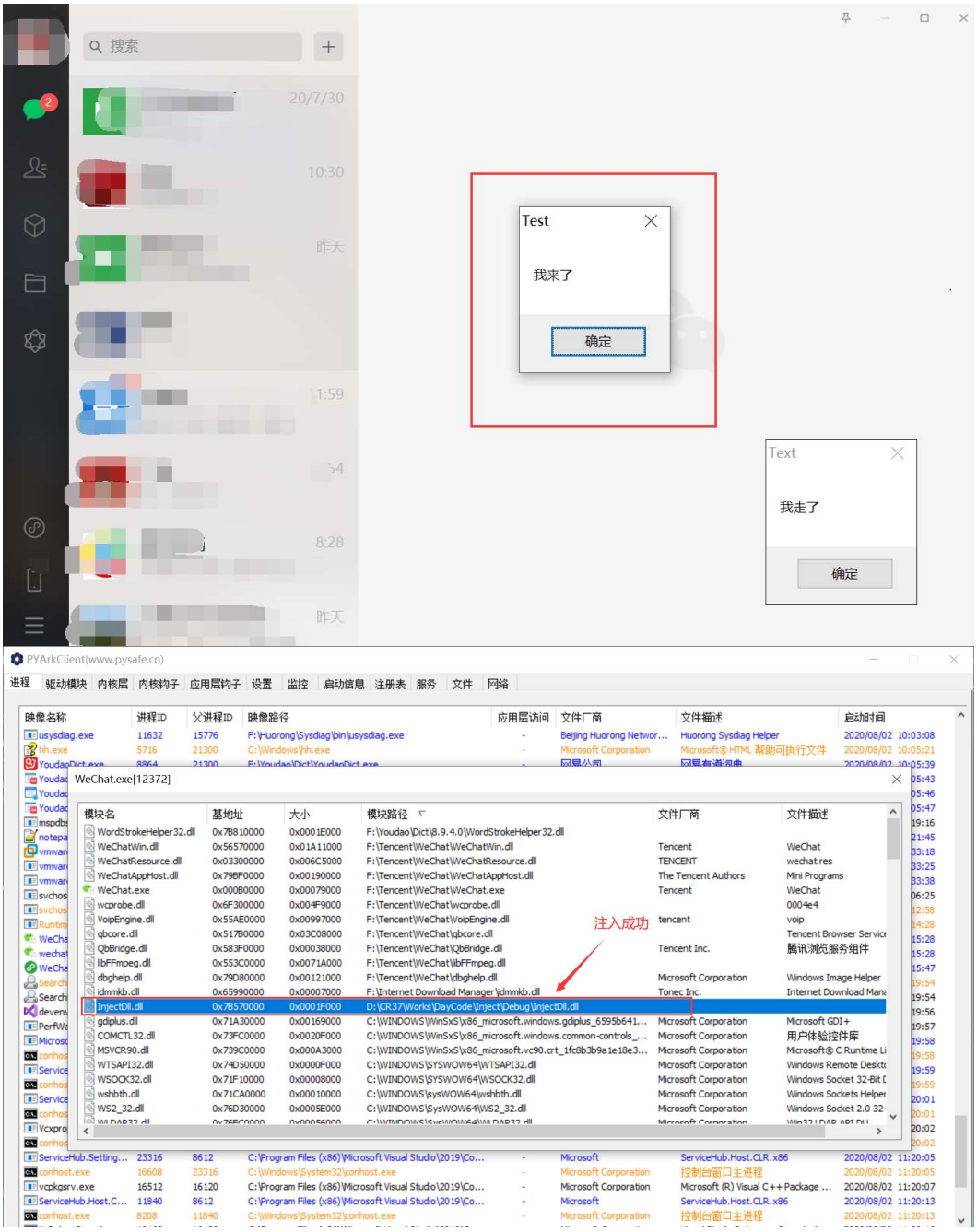
```
"Test", MB_OK);
        return TRUE;
      }
    }

    return g_pfnOldWndProc(hwnd, uMsg, wParam, lParam);
}

BOOL APIENTRY DllMain(HMODULE hModule,
    DWORD  ul_reason_for_call,
    LPVOID lpReserved
)
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
    {
        g_hModule = hModule;
        // 创建事件对象，用来判断dll是否打开成功
        CreateEvent(NULL, TRUE, FALSE, "Test");

        MessageBoxA(NULL, "我来了", "Test", MB_OK);

        // 给目标程序添加菜单，并追加子菜单
        HWND hWnd = FindWindow(NULL, "计算器");
        HMENU hMenu = GetMenu(hWnd);
        AppendMenu(hMenu, MF_POPUP | MF_STRING, (UINT_PTR)hMenu, "功能
(&N)");
        // 添加子菜单
        HMENU hSubMenu = GetSubMenu(hMenu, 3);
        AppendMenu(hSubMenu, MF_STRING, WM_USER + 1, "秒杀(&G)");
        AppendMenu(hSubMenu, MF_STRING, WM_USER + 2, "帮助(&A)");
        // 响应添加的菜单消息，修改过程函数，保存旧的过程函数
        g_pfnOldWndProc = (WNDPROC)SetWindowLongW(hWnd, GWL_WNDPROC,
(LONG)WindowProc);
        break;
    }
    case DLL_THREAD_ATTACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    case DLL_PROCESS_DETACH:
        MessageBoxA(NULL, "我走了", "Text", MB_OK);
        break;
    }
    return TRUE;
}
```


搜索
Test
我来了
确定
Text
我走了
确定
PYArkClient(www.pysafe.cn)
进程
驱动模块
内核层
内核钩子
应用层钩子
设置
监控
启动信息
注册表
服务
文件
网络
WeChat.exe[12372]
注入成功