2020/07/29_Windows编程_第9课_线程的创建、线程的交互

笔记本: Windows编程
创建时间: 2020/7/29 星期三 10:36
作者: ileemi
标签: 线程的创建, 线程的交互

WINMM.dll
WINMMBASE.dll

判断管道内是否有数据API--PeekNamedPipe

进程间的通讯（Inter-Process Communication -- IPC） -- 指至少两个进程或线程间传送数据或信号的一些技术或方法

线程介绍

进程（60年代）
线程（80年代）-- 为每个程序分配单独的空间来运行程序

> 为了解决进程的局限性

每创建一个进程，系统都会为其分配空间，好多软件都是一个主进程外加好多子进程，子进程过多会耗费系统资源。创建进程的开销比较大。进程间的通讯也会占用系统资源。

> 早期Linux解决多进程间资源的耗费：
> 1. 将代码放置在同一个进程中，使其内存（代码）共享，供多个进程使用，可以解决进程间通讯的问题，但是存在不能同时运行的问题。
> 代码区 -- 可以共享
> 数据区（全局数据区） -- 可以共享

> 栈 -- 不可以共享（当一个进程使用了栈中的数据后，另一个进程再使用栈数据会导致数据出错）
> 堆 -- 可以共享
> 为此，操作系统提供一个机制 ，栈不共享，剩下的内存进行共享。由于内存共享，通过全局变量就可以今进行线程间的通讯问题。不在需要进程间的通讯机制。

线程 -- 轻量级的进程

线程进程的差异

CPU 分配时间片以进程为单位，一个进程跑完，在去跑另一个进程。
线程：轻量级的进程（同样需要消耗系统资源，消耗的资源要比进程少）

> **注意**：
> **每个进程可以创建多个线程，创建的线程都属于这个进程，除了栈，其它内存都共享，CPU在调度的时候不在以进程为单位，会以线程为单位（CPU创建的时间片以线程为单位，不用在创建进程消耗更大的资源）进行调度。属于同一个进程的线程其内存是共享的，反之，不属于同一个进程的线程其内存不共享。每个线程单独跑。**

创建线程所消耗的资源要比创建进程所消耗的资源要少。创建一个进程可以使用多线程来解决我们所需要的问题。

创建线程的时候为其提供一个函数地址，这个线程就只负责这个函数地址上代码，该线程单独跑一个模块。

> 多线程同属于一个进程，不需要通讯，可以通过全局变量进程数据的读取，其内部内存是共享的。

多进程的作用：提供**内存隔离**的功能、

耦合性不强的两个模块适合分开来做，放在不同的进程中，方便管理。
耦合性强的两个模块适合放在同一个进程中。

线程的切换

线程的发明不但对操作系统有很大的影响，同时也对硬件的发展有很大的影响（例如：CPU）。
Windows 上 10ms 切换一个线程。

由CPU进程操控，一个线程一个线程的去跑，每个线程跑完后，会保存其**线程上下文**（硬件环境（CPU的寄存器环境））
线程1：1
线程2：2

超线程（一种更快速的切换线程环境）：将一个房间放入两张床，线程1来的时候，住床1，线程2来的时候，住床2，切换线程的时候，不用再保存**线程上下文**，速度上也提高了不少。以很低的成本使CPU运行多个线程，类似现实中的胶囊房。

一种更快速的切换线程环境：线程环境不需要保存和恢复

多核CPU要比超线程还要快。

线程的创建

创建线程API -- CretaeThread

创建一个进程最低有一个线程（主线程）因为现在CPU是以线程为单位的，通过主线程创建的线程称之为其子线程。

程序线程的分工设计：主线程只负责UI，子线程去实现功能。

> 创建线程的参数：
> 参数1：返回的句柄是否可以被子进程继承，一般创建线程不继承
> 参数2：栈的大小（在同一个进程中，每个线程在内存中使用的是同一个栈），当创建一个线程的是否，就要为该线程向系统申请一个指定大小的栈空间，防止和其它栈搞混乱。栈空间大小按需求向系统申请，大小单位为：**页**
> 参数3：函数地址（**线程过程函数**（**工作线程**））-- 创建的线程负责跑的功能模块
> 参数4：**会将该参数传递给线程的过程函数**
> 参数5：线程的标志（是否挂起）线程没有工作任务的时候，可以将其挂起（**CREATE_SUSPENDED**），挂起的线程，系统不为其分配时间片，为线程马上分配时间片填 0 即可（过程函数不会立马执行，线程会排队等待处理）。Widnows系统中的线程有VIP之分（线程优先级）。线程具有线程优先级（动态的，系统会根据情况确定哪个线程先跑，线程调度），优先级高的线程优先执行，优先分配时间片。
> 参数6：返回创建的 线程 ID 类型可以为 DWORD，不需要可以给空

返回值：返回创建线程的句柄。

编程来源于生活，生活中的相关问题都是经过验证的。

创建线程也是有上线的：最大线程数 -- 65535

主线程结束（系统认为进程结束），子线程不管有没有执行完都会停止执行（强制结束）。线程是依赖于进程的，进程结束，线程也就没必要在继续执行。

代码示例：

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

// 线程间数据共享可以通过全局变量
int g_nNum = 0;
bool g_Flag = false;
// 工作线程
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
  //while (true)
  for (int i = 0; i < 10; i++)
```

```
    {
        printf("talk... :%d\r\n", g_nNum);
        Sleep(1000);
    }

    g_Flag = true;
    return 0;
}
int main()
{
    /*
        代码区 -- 共享
        数据区 -- 共享
        栈 -- 不共享
        堆 -- 共享

        线程  超线程
        主线程)   子线程   子线程
        CREATE_SUSPENDED 挂起
        线程优先级  线程调度
        纤程
    */

    // 创建一个线程
    DWORD wdThreadID;
    HANDLE hThread = CreateThread(NULL, 0, WorkThread, NULL, 0,
&wdThreadID);

    if (hThread == NULL)
    {
        printf("CreateThread Error!\r\n");
    }
    printf("CreateThread hThread = %d wdThreadID = %d\r\n", hThread,
wdThreadID);
    //while (true)
    for (int i = 0; i < 5; i++)
    {
        puts("attck...");
        Sleep(100); // 让出时间片
        g_nNum += 1;
    }
    printf("g_nNum = %d\r\n", g_nNum);

    // 线程间需要进程交互
    //system("pause");
    while (!g_Flag);    // 当子线程代码执行完毕，才允许进程结束
```

```
    return 0;//主线程结束  进程结束(强制结束子线程)
}
```

线程间的交互

线程的回调函数的参数

上面的示例，当需求发生改变的时候，比如需要打两只怪，那么就需要再次定义一个线程以及一个回调函数，这样写代码，会造成线程1以和线程2的代码类似，出现代码冗余现象。当创建100个，1000个线程的时候，这种方法显然不是一种好的解决办法。

这个时候我们就需要用到**线程的回调函数的参数**来解决这个问题：参数的类型为指针，当需要传递的参数过多的时候可以传递一个结构体。

> 获取线程句柄：GetCurrentThread
> 获取线程ID：GetCurrentThreadId

代码如下：

```
// 工作线程
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    for (int i = 0; i < 5; i++)
    {
```

```
    printf("ThreadHandle: %p, ThreadID: %04d, attck...master: %d\r\n",
      GetCurrentThread(), // 获取进程句柄
      GetCurrentThreadId(), // 获取进程ID
      lpParameter);
    Sleep(1000);
  }
  g_Flag = true;
  return 0;
}
```

```
int main()
{
  // 创建一个线程
  DWORD wdThreadID[2];
  HANDLE hThread[2];
  for (int i = 0; i < 2; i++)
  {
    hThread[i] = CreateThread(NULL, 0, WorkThread, (LPVOID)(i+1), 0,
&wdThreadID[i]);
    if (hThread[i] == NULL)
    {
      printf("CreateThread %d Error!\r\n", wdThreadID[i]);
    }
    printf("CreateThread hThread = %d wdThreadID = %d\r\n", hThread,
wdThreadID);
  }
  while (!g_Flag);    // 当子线程代码执行完毕，才允许进程结束
  return 0;//主线程结束  进程结束(强制结束子线程)
}
```

等待线程结束

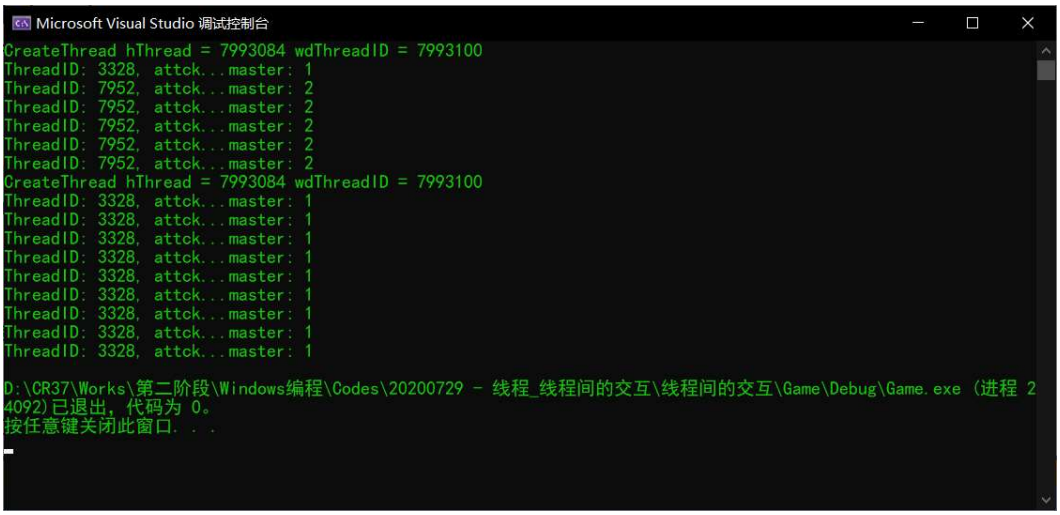
> WaitForSingleObject -- 等待，直到指定的进程等待用户输入而没有输入挂起，或者直到超时间隔结束
> 会阻塞，等待线程结束。（无线等待 **INFINITE**） -- 只等待指定的线程

> WaitForMultipleObjects -- 等待所有线程结束
> 参数1：等待所有的线程数量
> 参数2：存储所有线程的首地址
> 参数3：是否等待所有线程结束，TRUE -- 是，FALSE -- 否
> 参数4：等待的时间，以毫秒为单位

代码示例：

```
// Game.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>
// 线程间数据共享可以通过全局变量
int g_nNum = 0;
int g_Flag = false;
// 工作线程
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    int i = 0;
    for (i = 0; i < 10; i++)
    {
        printf("ThreadID: %04d, attck...master: %d\r\n",
            GetCurrentThreadId(), lpParameter);
        Sleep(500);
    }
    // 线程结束
    return 0;
}
// 测试线程2
DWORD WINAPI WorkThread2(LPVOID lpParameter)
{
    for (int i = 0; i < 5; i++)
    {
        printf("ThreadID: %04d, attck...master: %d\r\n",
            GetCurrentThreadId(), lpParameter);
        //Sleep(500);
    }
    return 0;
}
int main()
{
    /*
    代码区 -- 共享
    数据区 -- 共享
    栈 -- 不共享
    堆 -- 共享
    线程 超线程
    主线程 -- 子线程 -- 子线程
    CREATE_SUSPENDED 挂起
    线程优先级 线程调度
    纤程
    */
    DWORD wdThreadID[2];
    // 创建线程数组
```

```
    HANDLE  hThread[2];
    for  (int  i  =  0;  i  <  2;  i++)
    {
        hThread[i]  =  CreateThread(NULL,  0,
            i  ==  1  ?  WorkThread2  :  WorkThread,
            (LPVOID)(i+1),  0,
            &wdThreadID[i]);
        if  (hThread[i]  ==  NULL)
        {
            //  创建线程失败
            printf("CreateThread  %d  Error!\r\n",  wdThreadID[i]);
        }
        //  线程创建成功，输出对应的线程句柄以及线程ID
        printf("CreateThread  hThread  =  %d  wdThreadID  =  %d\r\n",
        hThread,  wdThreadID);
    }
      //  线程间需要进程交互
      //  等待线程结束     会阻塞
    WaitForSingleObject(hThread[0],  INFINITE);     //  线程不结束，挂起
等待
    WaitForSingleObject(hThread[1],  INFINITE);
    //  等待所有线程结束
    //WaitForMultipleObjects(2,  hThread,  TRUE,  INFINITE);
    return  0; //主线程结束     进程结束(强制结束子线程)
}
```



线程结束

> **TerminateThread** -- 终止一个线程(强制结束)
> 参数1：需要结束的线程句柄
> 参数2：需要结束的线程回调函数的返回值。
> **不推荐使用**，强制结束的线程可能存在还没有完成的事请，比如正在保存文件。

代码示例：
TerminateThread(hThread[0], 0);
TerminateThread(hThread[1], 1);

可以强制结束别的程序的的线程，通过快照遍历得到指定线程的PID。

比较合适的结束线程时机是在线程回调函数返回值之前结束。自己在回调函数中做标记来决定结束线程的时机。

优雅的结束线程，代码示例：

```
// Game.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>

// 线程间数据共享可以通过全局变量
int g_nNum = 0;
int g_Flag = false;

// 工作线程
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
  while (!g_Flag)
  {
    // 打开文件等操作
    printf("ThreadID: %04d, attck...master: %d\r\n",
      GetCurrentThreadId(), lpParameter);
    Sleep(500);
  }
  // 关闭文件操作

  return 0; // 线程结束
}

int main()
{

  DWORD wdThreadID[2];
  // 创建线程数组
  HANDLE hThread[2];
  for (int i = 0; i < 2; i++)
  {
    hThread[i] = CreateThread(NULL, 0,
      WorkThread, (LPVOID)(i+1), 0,
      &wdThreadID[i]);
```

```
        if (hThread[i] == NULL)
        {
            // 创建线程失败
            printf("CreateThread %d Error!\r\n", wdThreadID[i]);
        }
        // 线程创建成功，输出对应的线程句柄以及线程ID
        printf("CreateThread hThread = %d wdThreadID = %d\r\n",
        hThread, wdThreadID);
    }

    g_Flag = true;
    // 等待所有线程结束
    WaitForMultipleObjects(2, hThread, TRUE, INFINITE);

    return 0;//主线程结束  进程结束(强制结束子线程)
}
```

线程自己结束自己

> ExitThread -- 哪个线程调用这个函数，就自动结束自己。参数为：此线程的退出代码
> 主线程使用这个函数就会结束自己

GetExitCodeThread -- 检索指定线程的终止状态，**获取线程的返回值（退出码）。**

线程退出码的使用

GetExitCodeThread 获取线程的返回值（退出码）

代码示例：

```
// Game.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>
// 线程间数据共享可以通过全局变量
int g_nNum = 0;
int g_Flag = false;
```

```
void Fun1()
{
    // TerminateThread(GetCurrentThread(), 0);
    // 自己控制线程结束的位置
    ExitThread(0);
    // xxxx
}
void Fun2()
{
    Fun1();
}
// 工作线程
DWORD WINAPI WorkThread(LPVOID lpParameter)
{
    int i = 0;
    for (i = 0; i < 10 && !g_Flag; i++)
    {
        printf("ThreadID: %04d, attck...master: %d\r\n",
            GetCurrentThreadId(), lpParameter);
        Sleep(500);
    }
    //Fun2();
    // 线程结束
    return i;
}
DWORD WINAPI WorkThread2(LPVOID lpParameter)
{
    for (int i = 0; i < 5; i++)
    {
        printf("ThreadID: %04d, attck...master: %d\r\n",
            GetCurrentThreadId(), lpParameter);
        Sleep(500);
    }
    return 0;
}
int main()
{
    // 创建线程数组
    DWORD wdThreadID[2];
    HANDLE hThread[2];
    for (int i = 0; i < 2; i++)
    {
        hThread[i] = CreateThread(NULL,
        0, WorkThread,
        (LPVOID)(i+1),
        0, &wdThreadID[i]);
        if (hThread[i] == NULL)
```

```
        {
            printf("CreateThread %d Error!\r\n", wdThreadID[i]);
        }
        printf("CreateThread hThread = %d wdThreadID = %d\r\n",
        hThread, wdThreadID);
    }
    // 线程间需要进程交互
    // 等待线程结束    阻塞
    //WaitForSingleObject(hThread[0], 1000);    // 线程不结束，挂起等
待
    //WaitForSingleObject(hThread[1], 1000);
    //WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
    //TerminateThread(hThread[0], 0);
    //TerminateThread(hThread[1], 1);
    // 优雅的结束线程
    Sleep(2000);
    g_Flag = true;
    WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
    // 线程自己想结束
    // 获取线程返回值
    DWORD dwExitID0;
    GetExitCodeThread(hThread[0], &dwExitID0);
    DWORD dwExitID1;
    GetExitCodeThread(hThread[1], &dwExitID1);
    printf("hThread[0] = %d, hThread[1] = %d\r\n",
    dwExitID0, dwExitID1);

    // 设置线程优先级
    SetThreadPriority(hThread[0], THREAD_PRIORITY_HIGHEST);
    // 获取线程优先级
    int nThreadPriority = GetThreadPriority(hThread[0]);
    printf("hThread[0]Priority = %d\r\n", nThreadPriority);

    // 设置线程在哪核CPU上跑
    SetThreadAffinityMask(hThread[0], 1);
    return 0;
}
```

线程相关API

GetThreadPriority -- 获取指定线程的优先级（返回获取到的优先级）
SetThreadPriority -- 设置指定线程的优先级
SetThreadAffinityMask -- 设置线程在哪核CPU上跑
SuspendThread -- 挂起指定的线程（创建线程的时候也可以将线程挂起）
ResumeThread -- 减少线程的挂起计数（将指定挂起的线程恢复）

线程的挂起是用了引用计数，挂起几次就需要恢复几次，对应的线程才会恢复运行。

纤程

> API：CreateFible
> 一个进程可以有多个线程，一个线程可以有多个纤程
> 作用：可以自己控制时间片，自己管理，以及什么时候切换

是为了解决别的操作系统多线程代码可以移植到 Windows 上，能使用线程就不要去使用纤程。

线程存在问题：资源竞争