**《Windows核心编程》读书笔记**

# 0概述

作者：keenjin（金才）

时间：2017/11/1

计划阅读：**2**个月

截止日期：2017/1/1

# 1内核对象

# 2进程

# 3线程、调度及同步

## 3.1线程使用注意事项

划重点：

1. 伪线程句柄永远表示当前线程所在句柄，有歧义，在传递句柄时候，尽可能使用DuplicateHandle将GetCurrentThread转换成真正句柄。这个只有在父子线程传递的时候才需要用到。
2. \_beginthread一定不要用，而用\_beginthreadex，因为\_endthread内部会调用CloseHandle，外界如果再次调用CloseHandle会导致句柄重复释放
3. 线程栈默认占用虚拟内存1M，占用物理内存1K。如果物理内存1K用完，请求会触发页面异常中断；但是如果线程栈占用空间超过1M，则会导致栈溢出。

## 3.2线程上下文

|  |
| --- |
| typedef struct \_CONTEXT {  //  // The flags values within this flag control the contents of  // a CONTEXT record.  //  // If the context record is used as an input parameter, then  // for each portion of the context record controlled by a flag  // whose value is set, it is assumed that that portion of the  // context record contains valid context. If the context record  // is being used to modify a threads context, then only that  // portion of the threads context will be modified.  //  // If the context record is used as an IN OUT parameter to capture  // the context of a thread, then only those portions of the thread's  // context corresponding to set flags will be returned.  //  // The context record is never used as an OUT only parameter.  //  DWORD ContextFlags;  //  // This section is specified/returned if CONTEXT\_DEBUG\_REGISTERS is  // set in ContextFlags. Note that CONTEXT\_DEBUG\_REGISTERS is NOT  // included in CONTEXT\_FULL.  //  DWORD Dr0;  DWORD Dr1;  DWORD Dr2;  DWORD Dr3;  DWORD Dr6;  DWORD Dr7;  //  // This section is specified/returned if the  // ContextFlags word contians the flag CONTEXT\_FLOATING\_POINT.  //  FLOATING\_SAVE\_AREA FloatSave;  //  // This section is specified/returned if the  // ContextFlags word contians the flag CONTEXT\_SEGMENTS.  //  DWORD SegGs;  DWORD SegFs;  DWORD SegEs;  DWORD SegDs;  //  // This section is specified/returned if the  // ContextFlags word contians the flag CONTEXT\_INTEGER.  //  DWORD Edi;  DWORD Esi;  DWORD Ebx;  DWORD Edx;  DWORD Ecx;  DWORD Eax;  //  // This section is specified/returned if the  // ContextFlags word contians the flag CONTEXT\_CONTROL.  //  DWORD Ebp;  DWORD Eip;  DWORD SegCs; // MUST BE SANITIZED  DWORD EFlags; // MUST BE SANITIZED  DWORD Esp;  DWORD SegSs;  //  // This section is specified/returned if the ContextFlags word  // contains the flag CONTEXT\_EXTENDED\_REGISTERS.  // The format and contexts are processor specific  //  BYTE ExtendedRegisters[MAXIMUM\_SUPPORTED\_EXTENSION];  } CONTEXT; |

通过BOOL WINAPI GetThreadContext(\_\_in HANDLE hThread, \_\_in\_out LPCONTEXT lpContext);

可以获取到线程句柄对应的线程上下文

其中，FS寄存器，记录了TEB结构地址

## 3.3线程调度

（1）大约每隔20ms（GetSystemTimeAdjustment第二个参数的返回），Windows会查看当前所有存在的线程内核对象，查找可调度的（未被挂起的线程），载入CPU寄存器

（2）CreateThread时，线程挂起计数为1，直到初始化结束，会检查是否传入CREATE\_SUSPENDED，没有，挂起计数递减为0

（3）SuspendThread要非常小心，因为线程如果正在分配堆中的内存，线程将锁定堆，将它挂起，当前进程的其他线程就会一直等待这个线程恢复，才能使用堆

（4）Sleep传入非0，会导致当前线程在这个时间段内不被调度，不存在线程上下文切换。传入0，只是主动放弃主调线程剩余的时间片，下次任意时间依旧可以被调度。

（5）Sleep（0）和SwitchToThread功能类似，只不过前者重新回到调用Sleep的线程，后者可以选择到饥饿线程

（6）调用GetThreadContext前，应该先调用SuspendThread

## 3.4线程优先级

（1）每个线程都被赋予0（最低）~31（最高）的优先级数，高优先级线程会优先被执行，再逐步检查低优先级的

（2）20ms检查线程状态的时候，较高优先级的线程总是会抢占较低优先级的线程（抢占式操作系统），即使较低优先级的线程的时间片依旧存在。

（3）Windows提供了6个大的进程优先级类，如下：

idle（IDLE\_PRIORITY\_CLASS）：系统空闲时运行，例如屏保程序、后台使用程序和统计数据收集软件

below normal（BELOW\_NORMAL\_PRIORITY\_CLASS）

normal（NORMAL\_PRIORITY\_CLASS）：99%的应用程序都在这一层

above normal（ABOVE\_NORMAL\_PRIORITY\_CLASS）

high（HIGH\_PRIORITY\_CLASS）：此进程中的线程必须立即响应事件，例如任务管理器

real-time（REALTIME\_PRIORITY\_CLASS）：会抢占操作系统组件的CPU时间

设置进程优先级的方法有两种

方法一：由父进程创建子进程时，传递参数，如下：

BOOL WINAPI CreateProcess(

……

// NORMAL\_PRIORITY\_CLASS、IDLE\_PRIORITY\_CLASS、BELOW\_NORMAL\_PRIORITY\_CLASS

**DWORD dwCreationFlags,**

……

);

方法二：调用SetPriorityClass提权

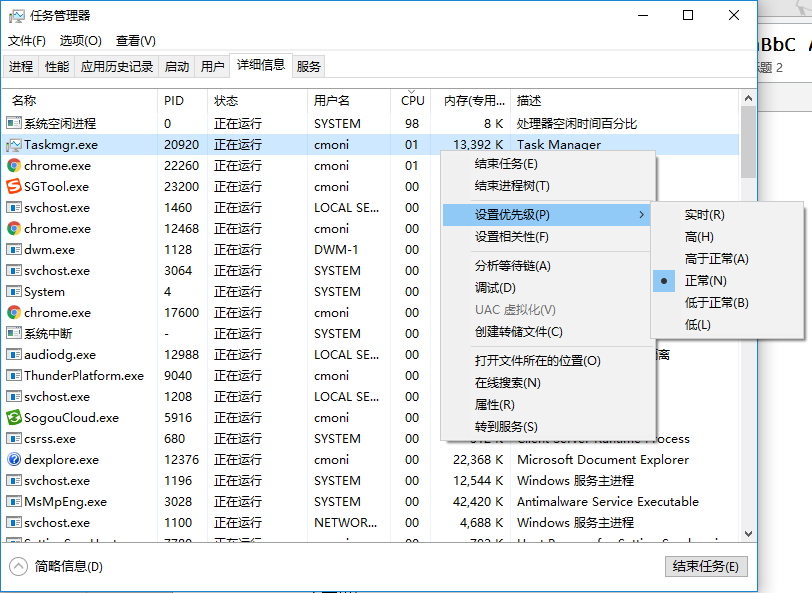
BOOL WINAPI SetPriorityClass(

\_\_in HANDLE hProcess,

\_\_in DWORD dwPriorityClass

);

DWORD WINAPI GetPriorityClass(HANDLE hProcess);



（4）Windows提供了7个大的相对线程优先级类，如下：

idle（THREAD\_PRIORITY\_IDLE）

lowest（THREAD\_PRIORITY\_LOWEST）

below normal（THREAD\_PRIORITY\_BELOW\_NORMAL）

normal（THREAD\_PRIORITY\_NORMAL）

above normal（THREAD\_PRIORITY\_ABOVE\_NORMAL）

highest（THREAD\_PRIORITY\_HIGHEST）

time-critical（THREAD\_PRIORITY\_TIME\_CRITICAL）

线程优先级只有一种方式可以改变，如下：

BOOL WINAPI SetThreadPriority(

\_\_in HANDLE hThread,

\_\_in int nPriority

);

int WINAPI GetThreadPriority(HANDLE hThread);

（5）CreateThread总是创建相对线程优先级为normal的线程，要创建idle的线程，需要传入CREATE\_SUSPENDED标记，然后设置线程优先级为idle，然后再ResumeThread

（6）从Vista开始，可以进行I/O请求时设置优先级SetThreadPriority（THREAD\_MODE\_BACKGROUND\_BEGIN），提升性能，同时还能针对文件维度，设置优先级（SetFileInformationByHandle）

## 3.5用户模式线程同步方式

（1）旋转锁：

|  |
| --- |
| // 旋转锁范例  BOOL g\_fResourceInUse = FALSE;  void Func1()  {  // 等待可以访问资源  while (InterlockedExchange(&g\_fResourceInUse, TRUE) == TRUE) {  Sleep(0);  }  // 访问资源  ...  InterlockedExchange(&g\_fResourceInUse, FALSE);  } |

缺点：

1. 所有使用旋转锁的线程需要以相同优先级运行，否则容易卡死（禁用线程优先级提升）
2. 锁变量需要与保护的资源处于不同高速缓冲行，否则CPU之间可能发生争夺
3. 单CPU机器上尽量避免旋转锁，主要是浪费CPU时间（线程切换），而且会阻止其他线程改变锁的值（Sleep很短，锁范围很窄，其他CPU很难命中修改锁的时刻，增加Sleep时间则会引起CPU更多的浪费）

（2）关键段：

使用旋转锁，只不过会设定循环次数，比如4000，届时仍无法访问资源，则会切换到内核模式等待

（3）原子栈（Interlocked单向链表的栈，所有的入栈出栈，都是原子操作）

|  |
| --- |
| #include <windows.h>  #include <malloc.h>  #include <stdio.h>  // Structure to be used for a list item; the first member is the  // SLIST\_ENTRY structure, and additional members are used for data.  // Here, the data is simply a signature for testing purposes.  typedef struct \_PROGRAM\_ITEM {  SLIST\_ENTRY ItemEntry;  ULONG Signature;  } PROGRAM\_ITEM, \*PPROGRAM\_ITEM;  int main()  {  ULONG Count;  PSLIST\_ENTRY pFirstEntry, pListEntry;  PSLIST\_HEADER pListHead;  PPROGRAM\_ITEM pProgramItem;  // Initialize the list header.  pListHead = (PSLIST\_HEADER)\_aligned\_malloc(sizeof(SLIST\_HEADER),  MEMORY\_ALLOCATION\_ALIGNMENT);  if (NULL == pListHead)  {  printf("Memory allocation failed.\n");  return -1;  }  InitializeSListHead(pListHead);  // Insert 10 items into the list.  for (Count = 1; Count <= 10; Count += 1)  {  pProgramItem = (PPROGRAM\_ITEM)\_aligned\_malloc(sizeof(PROGRAM\_ITEM),  MEMORY\_ALLOCATION\_ALIGNMENT);  if (NULL == pProgramItem)  {  printf("Memory allocation failed.\n");  return -1;  }  pProgramItem->Signature = Count;  pFirstEntry = InterlockedPushEntrySList(pListHead,  &(pProgramItem->ItemEntry));  }  // Remove 10 items from the list and display the signature.  for (Count = 10; Count >= 1; Count -= 1)  {  pListEntry = InterlockedPopEntrySList(pListHead);  if (NULL == pListEntry)  {  printf("List is empty.\n");  return -1;  }  pProgramItem = (PPROGRAM\_ITEM)pListEntry;  printf("Signature is %d\n", pProgramItem->Signature);  // This example assumes that the SLIST\_ENTRY structure is the  // first member of the structure. If your structure does not  // follow this convention, you must compute the starting address  // of the structure before calling the free function.  \_aligned\_free(pListEntry);  }  // Flush the list and verify that the items are gone.  pListEntry = InterlockedFlushSList(pListHead);  pFirstEntry = InterlockedPopEntrySList(pListHead);  if (pFirstEntry != NULL)  {  printf("Error: List is not empty.\n");  }  return 1;  } |

（4）Slim读/写锁：

读写锁的具体做法，就是让写线程独占资源（AcquireSRWLockExclusive），任何读线程或写线程，都无法访问资源；而读线程，仅仅只是共享资源（AcquireSRWLockShared）。

缺点：

1. 如果锁已经被占用，那么AcquireSRWLock（Exclusive/Shared）调用线程会被阻塞
2. 不能递归获得SRWLOCK，一个线程不能为了多次写入资源而多次锁定资源。

（5）性能优化总结：为了得到最佳性能，

1）首先尝试不要共享数据

2）volatile读取

3）volatile写入

4）Interlocked API

5）SRWLock

6）关键段

（6）条件变量：释放锁并且立马等待某个条件成立的原子操作，比如队列满的时候，等待队列被另一个线程清除掉一点空间

## 3.6内核模式线程同步方式

4设备IO及线程池

5内存、内存映射文件及堆

6动态链接库

# 7异常

## 7.1 EXCEPTION\_EXECUTE\_HANDLER

1. 将会处理接下来的\_\_except内的异常处理代码
2. 会进行全局展开，在执行\_\_except之前，会先从栈的底部（当前函数内），开始寻找\_\_finally，没有找到，会继续上层寻找（外层函数），找到\_\_finally后，会执行\_\_finally内的代码，执行完后，继续向上找到所有其他\_\_finally块继续执行，直到所有的未完成的\_\_try块，对应的\_\_finally都被找到。至此，全局展开结束。
3. 执行完\_\_finally，又会继续回到最内层（当前栈底）函数，执行\_\_except块代码
4. 如果执行的任意一个\_\_finally中包含return语句，会导致全局展开结束，系统回到栈底（最内层函数）继续执行，\_\_except块将没有机会执行

## EXCEPTION\_CONTINUE\_EXECUTION

1. 将会回到异常发生的位置，继续执行（如果异常不被处理掉，会循环抛出异常）
2. 可以借用异常过滤程序，先处理掉异常，然后再让异常回到发生的位置
3. 非常危险，不建议使用

## EXCEPTION\_CONTINUE\_SEARCH

1. 当前不处理异常，操作系统请继续向上寻找异常处理函数（如果所有的异常处理标记，都是CONTINUE\_SEARCH，那么异常就完全无法处理，最终会被SetUnhandledExceptionFilter设置的未处理异常函数所处理）
2. c/c++运行时库，会再进程入口点，安装一个全局异常过滤程序\_\_CxxUnhandledExceptionFilter，它会检查是否是c++异常，如果是，则执行abort函数，这个函数会调用Kernel32.dll中的UnhandledExceptionFilter函数，如果检查不是c++异常，会返回EXCEPTION\_CONTINUE\_SEARCH，来让Windows处理这个未处理异常，最后Windows会接盘，接盘原理如下：

VOID BaseThreadStart(PTHREAD\_START\_ROUTINE pfnStartAddr, PVOID pvParam){

\_\_try{

ExitThread((pfnStartAddr)(pvParam));

}

\_\_except(**UnhandledExceptionFilter**(GetExceptioninformation())){

ExitProcess(GetExceptionCode());

}

}

这里有必要了解到一个问题，c++运行时库，在初始化的时候，安装了异常过滤，但是假设我们的异常过滤程序，是通过全局变量设置（比如类的构造函数中设置），会不会比c++的早呢？这个不会早，是因为程序初始化顺序如下：

1. 进程命令行、环境变量等指针获取
2. C运行时库全局及静态变量初始化
3. 初始化C运行时库内存分配函数（malloc和calloc）及使用的堆（heap）
4. 调用所有全局及静态C++构造函数
5. 进入入口点主函数

## SEH和VEH的区别

（1）SEH既可以用在用户态，也可以用在内核态驱动程序中；VEH只能用在用户态

（2）异常执行的顺序： VEH、SEH、调试器

（3）SEH的注册信息是以固定的结构存储在线程栈中的，不同层次的各个SEH的注册信息依次被压入到栈中，分布在**栈**的不同位置上，依靠结构体内的指针相联系

（4）VEH的注册信息是存储在进程的内存**堆**中

（5）VEH相对于整个进程有效，具有全局性；SEH是动态建立在所在函数的栈帧上，会随着函数的返回而被注销，只对当前函数和其子函数有效

（6）SEH的注册和注销是依赖编译器编译时生成的数据结构和代码；VEH的注册和注销是通过显示调用API完成的，不依赖编译器

（7）最核心的区别就是，SEH设置顶层异常过滤程序，但是顶层异常过滤程序只有一个，任意一个人设置了，都会把别人的顶掉；VEH就不会如此，是链表结构。

（8）VEH不能返回EXCEPTION\_EXECUTER\_HANDLER。当异常可以被处理时，需要返回EXCEPTION\_CONTINUE\_EXECUTION，一旦返回这个，SEH就没有执行的机会；只有所有的VEH过滤程序都返回EXCEPTION\_CONTINUE\_SEARCH，SEH才有机会执行

（9）VEH是不管3721，所有的异常都会被监控，包括代码中自己定义的\_\_try{}\_\_exception(){}；SEH的SetUnhandledExceptionFilter只会关注没有被处理的异常

## 7.5必看博客

<https://www.codeproject.com/Articles/207464/Exception-Handling-in-Visual-Cplusplus>

<https://www.codeproject.com/Articles/2126/How-a-C-compiler-implements-exception-handling>

<https://www.codeproject.com/Articles/5259/XCrashReport-Exception-Handling-and-Crash-Report>

## 7.6引起异常的可能

（1）访问无效内存（如空指针）

（2）栈溢出（无穷递归调用）

（3）大块数据写入小块buffer

（4）纯虚函数调用

（5）内存分配失败（内存不够用）

（6）无效参数传递到C++系统函数

（7）C运行时库检测到错误请求终止程序