1 临界区&自旋锁

这两个章节**在”多核同步“篇已经学习过了**，需要了解的可以自行查看对应章节。

2 线程等待与唤醒

我们在之前的课程里面了解了如何**自己实现临界区**以及**什么是Windows自旋锁**，这两种同步方案在线程无法进入临界区时都会让当前线程进入等待状态。

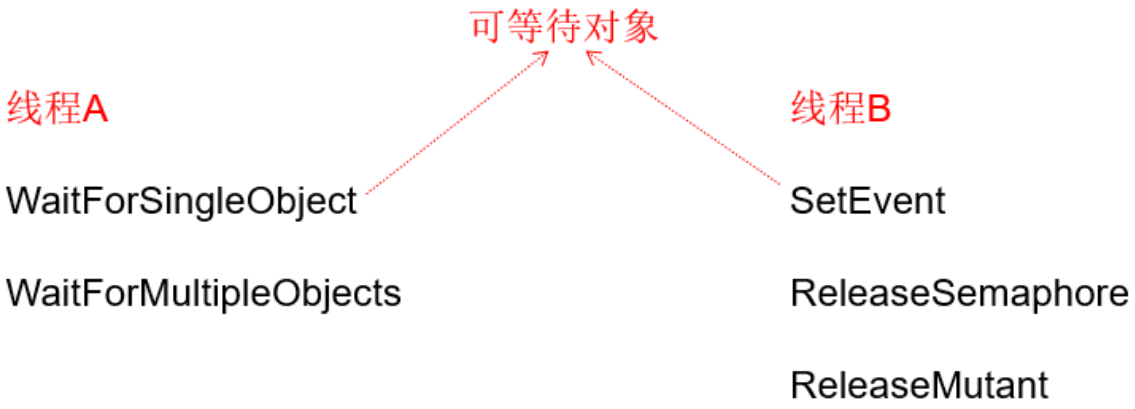
一种是通过Sleep函数实现的，一种是通过让当前的CPU”空转“实现的，但这两种等待方式都有局限性：

1. 通过Sleep函数进行等待，并没有办法确定具体等待的时间，有可能出现**等待过长或过短**的情况；
2. 通过让CPU”空转“进行等待，只有在等待事件很短的情况下才有意义，否则空转时间过长，**对CPU资源来说就是一种浪费**，并且自旋锁的方式**只在多核的环境下存在**。

我们发现如上所说的两种方案，都是由于等待条件的不成熟，所产生的局限。

2.1 等待与唤醒机制

在Windows中，一个线程可以通过等待一个或者多个可等待对象，从而进入等待状态，另一个线程可以在某些时刻唤醒等待这些对象的其他线程，这就是**Windows的等待与唤醒机制**。



2.2 可等待对象

所谓可等待对象就是结构体，如下是一些结构体，我们可以在Windbg中查看这些结构体。

在Windbg中查看如下结构体：

dt _KPROCESS	进程
dt _KTHREAD	线程
dt _KTIMER	定时器
dt _KSEMAPHORE	信号量
dt _KEVENT	事件
dt _KMUTANT	互斥体
dt _FILE_OBJECT	文件

在Windbg中查看结构体，我们会发现这些**结构体的第一个成员都是_DISPATCHER_HEADER**，也就表示第一个成员为_DISPATCHER_HEADER的结构体就是可等待对象。

```
kd> dt _KPROCESS
nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
kd> dt _KTHREAD
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
```

除了这个以外，在Windows中还有一些特殊的结构体，也称之为可等待对象，如_FILE_OBJECT，我们可以看见该结构体的第一个成员就不是_DISPATCHER_HEADER，**但是在它的0x5C偏移位成员有一个_KEVENT结构体，这个结构体是一个可等待对象**，因此_FILE_OBJECT也是一个可等待对象。

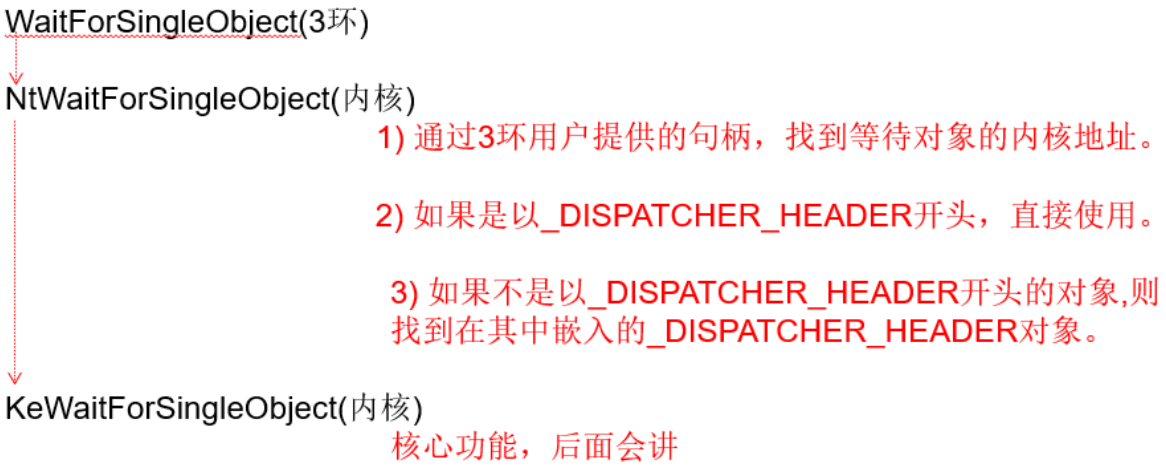
```
kd> dt _FILE_OBJECT
nt!_FILE_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x008 Vpb              : Ptr32 _VPB
   +0x00c FsContext        : Ptr32 Void
   +0x010 FsContext2       : Ptr32 Void
   +0x014 SectionObjectPointer : Ptr32 _SECTION_OBJECT_POINTERS
   +0x018 PrivateCacheMap  : Ptr32 Void
   +0x01c FinalStatus      : Int4B
   +0x020 RelatedFileObject : Ptr32 _FILE_OBJECT
   +0x024 LockOperation    : UChar
   +0x025 DeletePending    : UChar
   +0x026 ReadAccess       : UChar
   +0x027 WriteAccess      : UChar
   +0x028 DeleteAccess     : UChar
   +0x029 SharedRead       : UChar
   +0x02a SharedWrite      : UChar
   +0x02b SharedDelete     : UChar
   +0x02c Flags            : Uint4B
   +0x030 FileName         : _UNICODE_STRING
   +0x038 CurrentByteOffset : _LARGE_INTEGER
   +0x040 Waiters          : Uint4B
   +0x044 Busy             : Uint4B
   +0x048 LastLock         : Ptr32 Void
   +0x04c Lock             : _KEVENT
   +0x05c Event            : _KEVENT
   +0x06c CompletionContext : Ptr32 _IO_COMPLETION_CONTEXT
```

因此，综上所述我们可以知道**只要结构体中成员有_DISPATCHER_HEADER，或包含了_DISPATCHER_HEADER结构体的，我们都可以称之为可等待对象**。

2.2.1 差异

虽然以上所述的两种类型结构体都称之为可等待对象，但两者之间也是有差异的。差异在等待函数调用过程中体现出来。

当我们使用WaitForSingleObject函数时，进入内核函数NtWaitForSingleObject，这个内核函数会通过3环用户提供的句柄找到等待对象的内核地址；然后**判断等待对象的第一个成员是否是_DISPATCHER_HEADER，如果是的话则直接使用**；如果不是的话，**则去等待对象中找到嵌入的_DISPATCHER_HEADER对象**。最后**再将找到的对象地址作为参数调用KeWaitForSingleObject函数**，该函数核心功能会在后续章节中学习。



2.3 等待块

一个线程可以等待一个或多个对象，**线程与等待对象建立联系主要通过等待块**，我们可以做个实验来看一下。

2.3.1 一个线程等待一个对象

首先我们可以在XP中编译这样一段代码，来看一下一个线程等待一个对象的情况：

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

HANDLE hEvent[2];

DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
    ::WaitForSingleObject(hEvent[0], -1);

    printf("ThreadProc函数执行...\n");
    return 0;
}

int main(int argc, char* argv[])
{
    hEvent[0] = ::CreateEvent(NULL, TRUE, FALSE, NULL);

    ::CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadProc, NULL, 0, NULL);

    getchar();
    return 0;
}
```

运行程序之后，在Windbg中断一下，然后通过如下图中的指令找到当前进程中正在等待的线程地址：

```
Failed to get VadRoot
PROCESS 8903a8f0  SessionId: 0  Cid: 073c    Peb: 7ffd9000  ParentCid: 03a8
    DirBase: 0a3c0340  ObjectTable: e24e4a48  HandleCount:  16.
    Image: Test1.exe                                          指令：!process 0 0

THREAD 88fce8e8  Cid 073c.0368  Teb: 7ffde000 Win32Thread: 00000000 WAIT: (UserRequest) UserMode Non-Alertable
    89122748  NotificationEvent
Not impersonating
DeviceMap                 e14eee90
Owning Process            00000000       Image:
Attached Process          8903a8f0       Image:         Test1.exe          指令：!process 8903a8f0
Wait Start TickCount      146181173      Ticks: 343 (0:00:00:05.359)
Context Switch Count      4              IdealProcessor: 0
UserTime                  00:00:00.000
KernelTime                00:00:00.000
Win32 Start Address 0x00401005
Stack Init b1ff2000 Current b1ff1ca0 Base b1ff2000 Limit b1fef000 Call 00000000
Priority 12 BasePriority 8 PriorityDecrement 2 IoPriority 0 PagePriority 0
```

接着我们以线程结构体的方式代入线程地址查看，找到0x5C偏移位成员，即等待块：

```
kd> dt _KTHREAD 88fce8e8
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY [ 0x88fce8f8 - 0x88fce8f8 ]
   +0x018 InitialStack     : 0xb1ff2000 Void
   +0x01c StackLimit       : 0xb1fef000 Void
   +0x020 Teb              : 0x7ffde000 Void
   +0x024 TlsArray         : (null)
   +0x028 KernelStack      : 0xb1ff1ca0 Void
   +0x02c DebugActive      : 0 ''
   +0x02d State            : 0x5 ''
   +0x02e Alerted          : [2]  ""
   +0x030 Iopl             : 0 ''
   +0x031 NpxState         : 0xa ''
   +0x032 Saturation       : 0 ''
   +0x033 Priority         : 12 ''
   +0x034 ApcState         : _KAPC_STATE
   +0x04c ContextSwitches  : 4
   +0x050 IdleSwapBlock    : 0 ''
   +0x051 Spare0           : [3]  ""
   +0x054 WaitStatus       : 0n0
   +0x058 WaitIrql         : 0 ''
   +0x059 WaitMode         : 1 ''
   +0x05a WaitNext         : 0 ''
   +0x05b WaitReason       : 0x6 ''
   +0x05c WaitBlockList    : 0x88fce958 _KWAIT_BLOCK
```

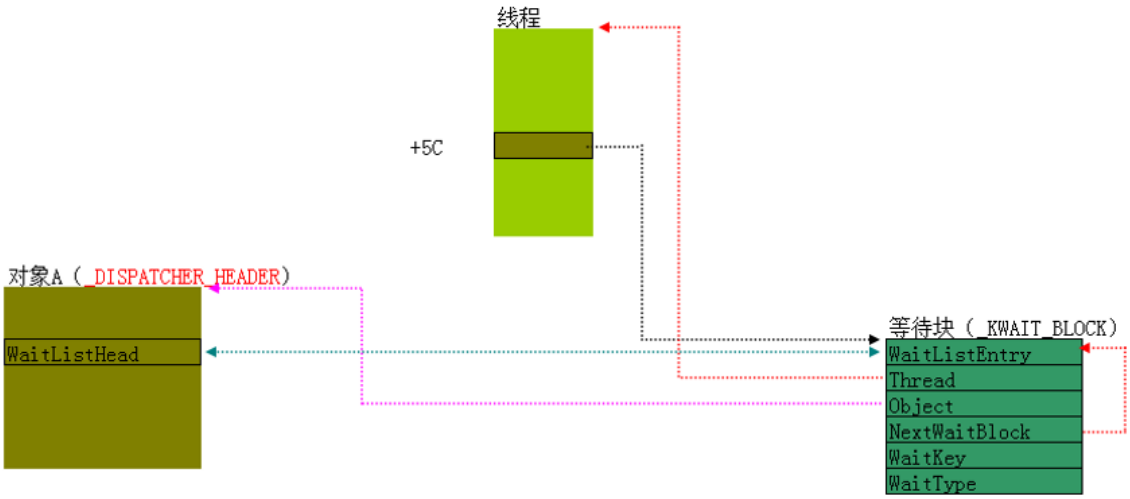
等待块是一个_KWAIT_BLOCK结构体，它将线程与被等待对象联系到了一起，我们接着来看一下它的成员含义：

```
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY         // 稍后了解
   +0x008 Thread           : Ptr32 _KTHREAD      // 当前线程地址
```

```
4     +0x00c Object           : Ptr32 Void           // 等待对象的地址（当前实验中为_KEVENT）
5     +0x010 NextWaitBlock    : Ptr32 _KWAIT_BLOCK   // 下一个等待块地址，这是一个单向循环链表，存储的是与当前线程关联的多个等待块结构体地址，如果只有一个等待块则该地址指向当前等待快地址
6     +0x014 WaitKey          : Uint2B               // 等待块的索引，当前为第一个等待块，因此该值为0
7     +0x016 WaitType         : Uint2B               // 等待类型，若当前只要有一个等待对象符合条件就可以使得线程被唤醒，那么该值就是1；如果你等待多个对象必须全部符合条件才可以使得线程被唤醒，那么该值0
```

```
kd> dt _KWAIT_BLOCK 0x88fce958
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x89122750 - 0x89122750 ]
   +0x008 Thread           : 0x88fce8e8 _KTHREAD
   +0x00c Object           : 0x89122748 Void
   +0x010 NextWaitBlock    : 0x88fce958 _KWAIT_BLOCK
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 1
```

综上所述，我们可以使用如下图来表示一个线程等待一个对象的情况：



2.3.2 一个线程等待多个对象

一个线程等待多个对象的情况，我们需要将代码进行修改，如下代码所示，我们添加两个可等待对象，然后将WaitForSingleObject替换为WaitForMultipleObjects，这样就可以使得一个线程等待多个对象。

值得注意的是，WaitForMultipleObjects函数多出了两个参数，分别是**第一个参数nCount（即等待对象的数量）**和**第二个参数bWaitAll（即是否要所有等待对象符合条件，该值决定了等待块的WaitType成员的值，该值为FALSE，则WaitType值为1，反之为0）**。

```
1   #include <windows.h>
```

```
#include <stdio.h>

HANDLE hEvent[2];

DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
    // ::WaitForSingleObject(hEvent[0], -1);
    ::WaitForMultipleObjects(2, hEvent, FALSE, -1);

    printf("ThreadProc函数执行...\n");
    return 0;
}

int main(int argc, char* argv[])
{
    hEvent[0] = ::CreateEvent(NULL, TRUE, FALSE, NULL);
    hEvent[1] = ::CreateEvent(NULL, TRUE, FALSE, NULL);

    ::CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadProc, NULL, 0,
NULL);

    getchar();
    return 0;
}
```

还是跟之前的步骤一样，找到正在等待的线程地址，然后找到等待块进行查看，我们会发现这时候就有两个等待块了：

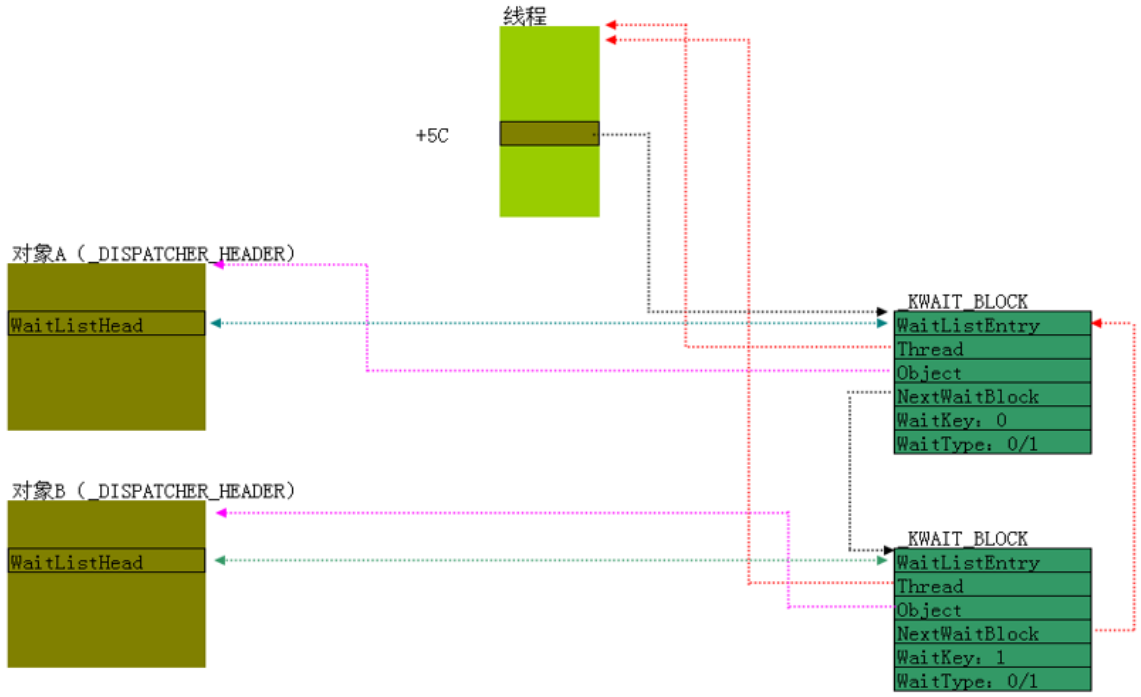
```
kd> dt _KWAIT_BLOCK 0x89052090
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x88ff2570 - 0x88ff2570 ]
   +0x008 Thread           : 0x89052020 _KTHREAD
   +0x00c Object           : 0x88ff2568 Void
   +0x010 NextWaitBlock    : 0x890520a8 _KWAIT_BLOCK
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 1
kd> dt _KWAIT_BLOCK 0x890520a8
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x8975f658 - 0x8975f658 ]
   +0x008 Thread           : 0x89052020 _KTHREAD
   +0x00c Object           : 0x8975f650 Void
   +0x010 NextWaitBlock    : 0x89052090 _KWAIT_BLOCK
   +0x014 WaitKey          : 1
   +0x016 WaitType         : 1
```

那么接着我们来看一下Object成员指向的等待对象，我们知道它是一个_KEVENT结构体，**该结构体只有一个成员，即_DISPATCHER_HEADER**，因此我们可以根据Object的地址代入_DISPATCHER_HEADER结构体进行查看。

如下图所示，我们会发现，在_DISPATCHER_HEADER结构体的0x8偏移位成员是一个WaitListHead，这是一个链表，该链表指向的就是当前等待对象_KEVENT**对应线程的等待块**（若包含多个等待块，也就表示有多个线程在等待一个等待对象，则会将这些等待块依次插入链表中）。接着我们会发现在当前线程等待块的第一个成员WaitListEntry中存储的就是WaitListHead的值，**这样我们就完全明白了等待快与等待对象之间的关联**。

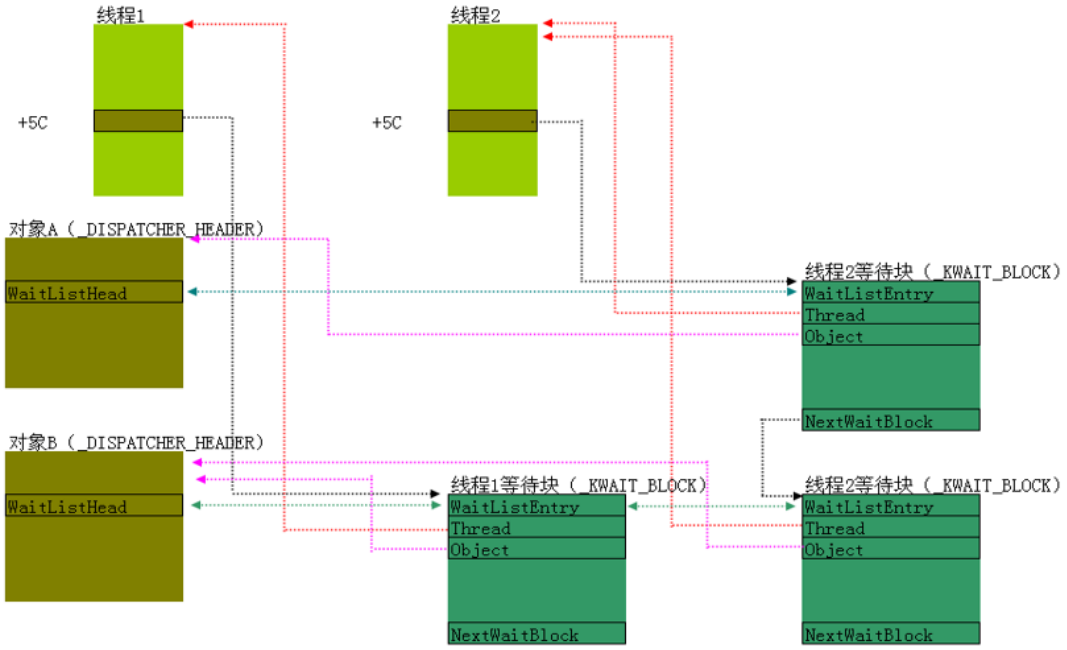
```
kd> dt _KWAIT_BLOCK 0x890520a8
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x8975f658 - 0x8975f658 ]
   +0x008 Thread           : 0x89052020 _KTHREAD
   +0x00c Object           : 0x8975f650 Void
   +0x010 NextWaitBlock    : 0x89052090 _KWAIT_BLOCK
   +0x014 WaitKey          : 1
   +0x016 WaitType         : 1
kd> dt _KEVENT
nt!_KEVENT
   +0x000 Header           : _DISPATCHER_HEADER
kd> dt _DISPATCHER_HEADER 0x8975f650
nt!_DISPATCHER_HEADER
   +0x000 Type             : 0 ''
   +0x001 Absolute         : 0xf3 ''
   +0x002 Size             : 0x4 ''
   +0x003 Inserted         : 0x89 ''
   +0x004 SignalState      : 0n0
   +0x008 WaitListHead     : _LIST_ENTRY [ 0x890520a8 - 0x890520a8 ]
```

综上所述，我们可以使用如下图来表示一个线程等待多个对象的情况：



2.4 等待网

如下图所示，这是海东老师根据等待关系构造的一张等待网（这只是自定义的概念），我们可以看见所有处于等待状态的线程，线程对应的等待块，以及被等待对象，都会位于类似的等待网上。



等待中的线程一定会挂入等待链表中（即KiWaitListHead），同时也一定位于等待网上（即KThread+5C的位置不为空）。

2.5 WaitfFotSingleObject函数分析

无论可等待对象是何种类型，线程都是通过WaitForSingleObject、WaitForMultipleObjects函数进入等待状态的，**这两个函数是理解线程等待与唤醒机制的核心**。本章节，我们以WaitForSingleObject函数为例，进行分析，来了解线程等待与唤醒机制的本质。

WaitForSingleObject函数格式如下，对于我们来说它只是一个3环的API封装，通过系统调用的方式进入0环然后去调用NtWaitForSingleObject函数：

```
DWORD WaitForSingleObject(
    HANDLE hHandle,         // 句柄
    DWORD dwMilliseconds    // 超时时间
);
```

NtWaitForSingleObject函数格式如下，该函数做了两件事情：

1. 调用ObReferenceObjectByHandle函数，通过对象句柄找到等待对象结构体地址；
2. **将等待对象的地址作为传参，调用KeWaitForSingleObject函数**，进入关键循环。

```
NTSTATUS NtWaitForSingleObject(
    [in] HANDLE         Handle,    // 句柄
    [in] BOOLEAN        Alertable, // 对应_KTHREAD结构体的Alertable属性，该值为1时插入用户APC，线程会被吵醒
    [in] PLARGE_INTEGER Timeout    // 超时时间
);
```

因此，实际上**KeWaitForSingleObject函数**才是核心功能的实现。

2.5.1 KeWaitForSingleObject函数

等待块的填充

KeWaitForSingleObject函数可以分为两部分，首先做的是要在线程结构体中加入等待块，微软在设计线程结构体时实际上已经预留了一个空间用于存储等待块，即_KTHREAD的0x70偏移位成员WaitBlock，该成员预留了4个等待块的空间，即0x4 * _KWAIT_BLOCK的大小。

```
kd> dt _KTHREAD
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   ...
   +0x05c WaitBlockList    : _KWAIT_BLOCK
   ...
   +0x070 WaitBlock        : [4] _KWAIT_BLOCK
```

我们可以基于”一个线程等待一个对象“的代码例子来查看下该成员。如下图所示，我们可以看见4个等待块的空间中，第一个和第四个都已经被挂入了等待块，第一个是_KEVENT等待对象，**而第四个实际上是固定的，它是一个定时器**。在当前的环境中，我们可以看见第四个等待块并没有被使用（NextWaitBlock成员值为空），**这是因为在我们使用WaitForSingleObject函数时，超时时间参数设置为了-1**，因此就不会去使用定时器，**如果我们设置为一个正整数值，则定时器就会被启用**。

当定时器启用时，第四个等待块的NextWaitBlock成员就会指向第一个等待块的地址，同样第一个等待块的NextWaitBlock成员就会指向第四个等待块的地址。

```
kd> dt _KWAIT_BLOCK 0x89908020+0x70
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x88fc7b40 - 0x88fc7b40 ]
   +0x008 Thread           : 0x89908020 _KTHREAD
   +0x00c Object           : 0x88fc7b38 Void
   +0x010 NextWaitBlock    : 0x89908090 _KWAIT_BLOCK
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 1
kd> dt _KWAIT_BLOCK 0x89908020+0x70+0x18  _KWAIT_BLOCK结构体大小为0x18
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x008 Thread           : 0x89908020 _KTHREAD
   +0x00c Object           : (null)
   +0x010 NextWaitBlock    : (null)
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 0
kd> dt _KWAIT_BLOCK 0x89908020+0x70+0x18+0x18
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x008 Thread           : 0x89908020 _KTHREAD
   +0x00c Object           : (null)
   +0x010 NextWaitBlock    : (null)
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 0
kd> dt _KWAIT_BLOCK 0x89908020+0x70+0x18+0x18+0x18
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x89908118 - 0x89908118 ]
   +0x008 Thread           : 0x89908020 _KTHREAD
   +0x00c Object           : 0x89908110 Void
   +0x010 NextWaitBlock    : (null)
   +0x014 WaitKey          : 0x102
   +0x016 WaitType         : 1
```

由于在这里微软只预留了4个等待块的空间，并且第四个等待块还是被固定使用的，因此我们在使用WaitForMultipleObjects函数时，当等待对象超过3个，就不会再继续使用这里的空间，而是另辟一块新的空间存放等待块。如下图所示，有4个等待对象时候，**虽然0x70位置处的等待块空间被使用了，但实际上在0x5C位置处的等待块地址并不是它，0x70位置处的等待块地址也并没有出现在0x5C对应等待块的下一等待块地址（单向循环链表）中**。

0x70位置的等待块信息

```
kd> dt _KWAIT_BLOCK 0x8901a620+0x70
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0xb1c99b40 - 0xb1c99b40 ]
   +0x008 Thread           : 0x8901a620 _KTHREAD
   +0x00c Object           : 0x8901a814 Void
   +0x010 NextWaitBlock    : 0x8901a690 _KWAIT_BLOCK
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 1
```

0x5C位置的等待块信息

```
kd> dt _KWAIT_BLOCK 0x89060850
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x88fbd698 - 0x88fbd698 ]
   +0x008 Thread           : 0x8901a620 _KTHREAD
   +0x00c Object           : 0x88fbd690 Void
   +0x010 NextWaitBlock    : 0x89060868 _KWAIT_BLOCK
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 1
kd> dt _KWAIT_BLOCK 0x89060868
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x88fcf5a0 - 0x88fcf5a0 ]
   +0x008 Thread           : 0x8901a620 _KTHREAD
   +0x00c Object           : 0x88fcf598 Void
   +0x010 NextWaitBlock    : 0x89060880 _KWAIT_BLOCK
   +0x014 WaitKey          : 1
   +0x016 WaitType         : 1
kd> dt _KWAIT_BLOCK 0x89060880
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x88ffeb60 - 0x88ffeb60 ]
   +0x008 Thread           : 0x8901a620 _KTHREAD
   +0x00c Object           : 0x88ffeb58 Void
   +0x010 NextWaitBlock    : 0x89060898 _KWAIT_BLOCK
   +0x014 WaitKey          : 2
   +0x016 WaitType         : 1
kd> dt _KWAIT_BLOCK 0x89060898
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x88fc4540 - 0x88fc4540 ]
   +0x008 Thread           : 0x8901a620 _KTHREAD
   +0x00c Object           : 0x88fc4538 Void
   +0x010 NextWaitBlock    : 0x89060850 _KWAIT_BLOCK
   +0x014 WaitKey          : 3
   +0x016 WaitType         : 1
```

但是在”一个线程等待不超过3个对象“的场景下，**_KTHREAD+0x5C所指向的地址就是_KTHREAD+0x70的地址**。

关键循环

在等待块的填充和准备之后，KeWaitForSingleObject就会进入关键循环部分。（在课上，海东老师将该部分以伪代码的形式进行讲解，可以更快的进行理解）

关键循环部分的大致逻辑如下所示，通过代码我们知道在这里，SignalState参数起到了激活条件的判断作用，因此我们要先知道该参数的含义。

```
while(true)                 // 每次线程被其他线程唤醒，都要进入这个循环
{
    if(符合激活条件)        // 激活条件：1.超时；2.等待对象的SignalState成员值大于0
    {
        // 1. 修改SignalState；
        // 2. 退出循环。
    }
    else                    // SignalState不大于0 也没超时
    {
        if(第一次执行)
        {
            // 1. 将当前线程的等待块挂到等待对象的链表(WaitListHead) 中；
            // 2. 将自己挂入等待链表(KiaitListHead)；
            // 3. 切换线程...再次获得CPU时，从这里开始执行。
        }
    }
}

1. 线程将自己0x5C位置清0；
2. 释放_KWAIT_BLOCK所占内存。
```

我们先来看一下_DISPATCHER_HEADER结构体的各个成员含义：

```
kd> dt _DISPATCHER_HEADER
nt!_DISPATCHER_HEADER
   +0x000 Type             : UChar      // 可等待对象的类型
   +0x001 Absolute         : UChar
   +0x002 Size             : UChar
   +0x003 Inserted         : UChar
   +0x004 SignalState      : Int4B      // 是否有信号，与信号量有关
   +0x008 WaitListHead     : _LIST_ENTRY  // 双向链表，所有的等待块都在这里
```

接着我们来理一下这段伪代码的流程：

1. 进入循环，判断当前被等待对象（例如Event）是否有信号（即判断_DISPATCHER_HEADER.SignalState的值）或者是否超时（即是否满足激活条件）；
    a. **满足激活条件时**，修改_DISPATCHER_HEADER.SignalState的值，**这里不一定是清零**，因为每种等待对象修改的方式不一样，**最后退出循环**。
    b. **不满足激活条件时**，判断是否是第一次执行。
        i. **如果是第一次执行**，将当前线程的等待块挂到等待对象的链表（即_DISPATCHER_HEADER.WaitListHead）中，此时进入了等待网。
        ii. **接着将自己挂入等待链表（KiWaitListHead）中**，当线程自己把自己挂入等待链表以后，就相当于交出去CPU控制权，并进行线程切换；此时循环才执行到一半，并未执行完，自己就被切换出去了。
        iii. **当线程将自己挂入等待队列后**，需要等待另一个线程将自己唤醒（**设置等待对象信号SignalState>0**），此时这个线程会根据其设置的等待对象WaitListHead链着的等待块，给所有等待块的所属线程一次**临时复活**的机会；这个临时复活意思是将这些线程**从等待链表上摘除**，但是**依旧挂在等待网上**，所以被称作临时复活。
        iv. 唤醒的线程并不会将等待的线程从等待网上摘除，**这需要等待的线程自己进行摘除，因此当等待的线程再次获得CPU控制权时，会在"自己把自己挂入等待链表（KiWaitListHead）"后的位置，继续执行代码**，并重新进入循环判断的入口，再判断自己是否符合激活条件，如果符合的话则修改SignalState的值，并退出循环。
2. 退出循环之后，线程会将自己结构体0x5C的位置清0，释放_KWAIT_BLOCK所占内存，然后将自己从等待网上摘除，**成功复活（唤醒）**。

这里为什么等待的线程要自己把自己从等待网上进行摘除，而不是通过唤醒的线程来摘除，这是因为会**出现等待的线程不止有一个等待对象，或者一个等待对象有多个线程在等待的场景**，如果通过唤醒的线程去摘除则不符合设计的逻辑。

换句话说，在当前的逻辑里，等待的线程临时复活，会接着执行并重新进入循环，如果它只有一个等待块，那么这个线程会符合激活条件，并退出循环。但又因为它退出循环之前，修改了SignalState的值，**导致后面获得CPU控制权的线程（同样在等待该对象）不符合激活条件**，并将重新挂到KiWaitListHead等待链表上。

如果它有多个等待块，在判断第二个等待块时，也会因为不符合激活条件，也要重新将自己挂到KiWaitListHead上。也就是说，只有第一个获得CPU控制权，且只有一个等待块的线程，**才能完全复活，别的临时复活的线程，要重新挂到等待链表上**。

2.5.2 关于强制唤醒

在APC章节中了解过，当我们插入一个用户APC时（Alertable=1），当前线程是可以被唤醒的，但并不是真正的唤醒。因为，**如果当前的线程在等待网上，执行完用户APC后，仍然要进入等待状态**。

3 等待对象

在之前的章节学习中我们了解到KeWaitForSingleObject函数关键循环部分，会判断线程的激活条件，如果符合条件就**修改等待对象的SignalState**并退出循环。线程的激活条件（等待对象的SignalState）是由等待对象类型对应函数决定的，不同的等待对象有不同的函数**将线程进行临时唤醒**，如_KEVENT事件对象，它对应使用的临时唤醒函数是SetEvent。不同的等待对象，在**临时唤醒的处理**上，以及**修改等待对象的SignalState**的步骤上都会有所差异。

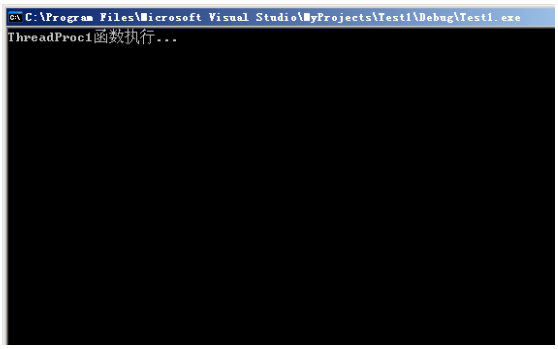
3.1 事件

我们先来看一下事件类型，事件对象可以通过CreateEvent函数进行创建，它的语法如下：

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes, // 决定返回的句柄是否可以被子进程继承,为NULL，则该句柄不能被继承
    BOOL bManualReset,                       // 创建的事件对象类型
    BOOL bInitialState,                      // 初始化的信号状态
    LPCTSTR lpName                           // 指定事件对象的名称
);
```

在这里我们需要关注bManualReset和bInitialState两个参数，bInitialState参数决定了**初始化的信号状态**，如果该值**为FALSE则事件对象的_DISPATCHER_HEADER.SignalState值为0，反之则为1**。如下图所示，我们可以发现确实如此，**初始化信号状态为TRUE时，没有等待直接就执行了下面的代码**，而为FALSE时则一直处于等待状态，没有执行代码。

```
DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
    ::WaitForSingleObject(hEvent[0], -1);

    printf("ThreadProc函数执行...\n");
    return 0;
}

DWORD WINAPI ThreadProc1(LPVOID lpParameter)
{
    ::WaitForSingleObject(hEvent[1], -1);

    printf("ThreadProc1函数执行...\n");
    return 0;
}


int main(int argc, char* argv[])
{
    hEvent[0] = ::CreateEvent(NULL, FALSE, FALSE, NULL);

    hEvent[1] = ::CreateEvent(NULL, FALSE, TRUE, NULL);

    ::CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadProc, NULL, 0, NULL);
    ::CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadProc1, NULL, 0, NULL);
```



bManualReset参数决定了**创建的事件对象类型**，该值为TRUE则表示当前为**通知类型**的对象，**事件对象的_DISPATCHER_HEADER.Type值为0**；该值为FALSE则表示当前为**事件同步类型**的对象，**事件对象的_DISPATCHER_HEADER.Type值为1**。如下图所示，我们也可以成功得到这个结论。

```
DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
    ::WaitForSingleObject(hEvent[0], -1);

    printf("ThreadProc函数执行...\n");
    return 0;
}

DWORD WINAPI ThreadProc1(LPVOID lpParameter)
{
    ::WaitForSingleObject(hEvent[1], -1);

    printf("ThreadProc1函数执行...\n");
    return 0;
}


int main(int argc, char* argv[])
{
    hEvent[0] = ::CreateEvent(NULL, TRUE, FALSE, NULL);

    hEvent[1] = ::CreateEvent(NULL, FALSE, FALSE, NULL);

    ::CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadProc, NULL, 0, NULL);
    ::CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadProc1, NULL, 0, NULL);
```

```
kd> dt _KWAIT_BLOCK 0x88fe86c8
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x88fc98f0 - 0x88fc98f0 ]
   +0x008 Thread           : 0x88fe8658 _KTHREAD
   +0x00c Object           : 0x88fc98e8 Void
   +0x010 NextWaitBlock    : 0x88fe86c8 _KWAIT_BLOCK
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 1
kd> dt _DISPATCHER_HEADER 0x88fc98e8
nt!_DISPATCHER_HEADER
   +0x000 Type             : 0 ''
   +0x001 Absolute         : 0x71 'q'
   +0x002 Size             : 0x4 ''
   +0x003 Inserted         : 0x89 ''
   +0x004 SignalState      : 0n0
   +0x008 WaitListHead     : _LIST_ENTRY [ 0x88fe86c8 - 0x88fe86c8 ]
kd> dt _KWAIT_BLOCK 0x898aa608
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x890493d8 - 0x890493d8 ]
   +0x008 Thread           : 0x898aa598 _KTHREAD
   +0x00c Object           : 0x890493d0 Void
   +0x010 NextWaitBlock    : 0x898aa608 _KWAIT_BLOCK
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 1
kd> dt _DISPATCHER_HEADER 0x890493d0
nt!_DISPATCHER_HEADER
   +0x000 Type             : 0x1 ''
   +0x001 Absolute         : 0x71 'q'
   +0x002 Size             : 0x4 ''
   +0x003 Inserted         : 0x89 ''
   +0x004 SignalState      : 0n0
   +0x008 WaitListHead     : _LIST_ENTRY [ 0x898aa608 - 0x898aa608 ]
```

通知类型与同步类型的事件对象也是有差异的，具体体现在SetEvnet函数使用后，等待通知类型的事件对象线程会**被唤醒然后向下执行代码**，而等待同步类型的事件对象线程**仍然处于等待状态**。

```
DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
    ::WaitForSingleObject(hEvent[0], -1);
    printf("ThreadProc函数执行...\n");
    return 0;
}

DWORD WINAPI ThreadProc1(LPVOID lpParameter)
{
    ::WaitForSingleObject(hEvent[0], -1);
    printf("ThreadProc1函数执行...\n");
    return 0;
}

DWORD WINAPI ThreadProc2(LPVOID lpParameter)
{
    ::WaitForSingleObject(hEvent[1], -1);
    printf("ThreadProc2函数执行...\n");
    return 0;
}

DWORD WINAPI ThreadProc3(LPVOID lpParameter)
{
    ::WaitForSingleObject(hEvent[1], -1);
    printf("ThreadProc3函数执行...\n");
    return 0;
}

int main(int argc, char* argv[])
{
    hEvent[0] = ::CreateEvent(NULL, TRUE, FALSE, NULL);
    hEvent[1] = ::CreateEvent(NULL, FALSE, FALSE, NULL);
    ::CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadProc, NULL, 0, NULL);
    ::CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadProc1, NULL, 0, NULL);
    SetEvent(hEvent[0]);
    SetEvent(hEvent[1]);
```

```
C:\Program Files\Microsoft Visual Studio\MyProjects\Test1\Debug\Test1.exe
ThreadProc函数执行...
ThreadProc1函数执行...
```

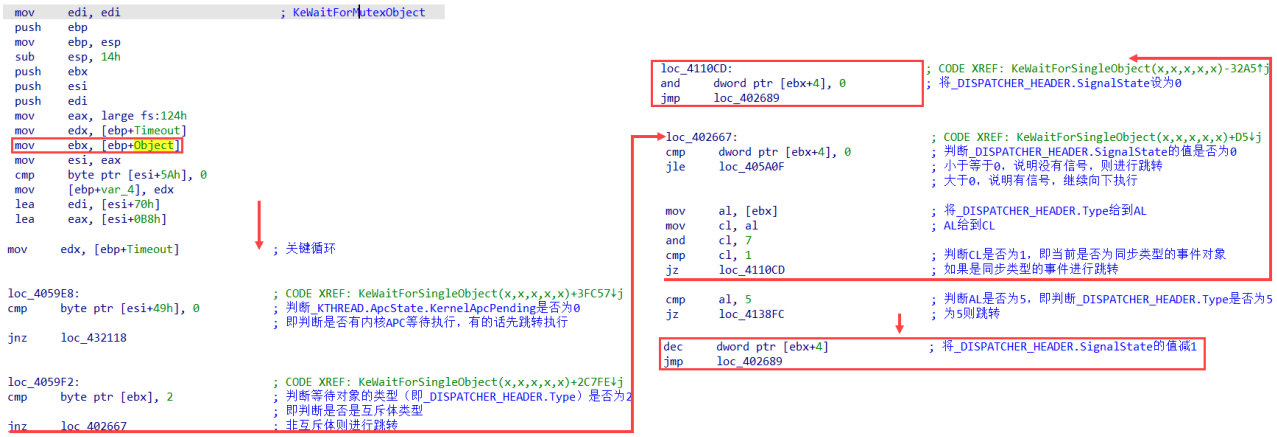
之前我们了解到不同的等待对象，在临时唤醒以及关键循环处修改SignalState会有差异，因此我们先来具体看下SetEvent函数的作用，**该函数对应的内核函数为KeSetEvent**，它做了这些事情：

1. 修改SignalState为1；
2. 判断对象类型：
    a. 通知类型，则唤醒所有等待该状态的线程，也就是**从等待链表中摘除这些线程**；
    b. 同步类型，从等待链表中找到第一个线程的等待块_KWAIT_BLOCK.WaitType值为1的线程，**将其临时唤醒**。
        i. 如下图所示，我们可以验证该结论：

ii.

```
DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
    // ::WaitForSingleObject(hEvent[0], -1);
    ::WaitForMultipleObjects(2, hEvent, TRUE, -1); // 0
    printf("ThreadProc函数执行...\n");
    return 0;
}

DWORD WINAPI ThreadProc1(LPVOID lpParameter)
{
    // ::WaitForSingleObject(hEvent[0], -1);
    ::WaitForMultipleObjects(2, hEvent1, FALSE, -1); // 1
    printf("ThreadProc1函数执行...\n");
    return 0;
}

int main(int argc, char* argv[])
{
    hEvent[0] = ::CreateEvent(NULL, FALSE, FALSE, NULL);
    hEvent[1] = ::CreateEvent(NULL, FALSE, FALSE, NULL);
    hEvent1[0] = ::CreateEvent(NULL, FALSE, FALSE, NULL);
    hEvent1[1] = ::CreateEvent(NULL, FALSE, FALSE, NULL);
    ::CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadProc, NULL, 0, NULL);
    ::CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadProc1, NULL, 0, NULL);
    SetEvent(hEvent[0]);
    SetEvent(hEvent1[0]);
```

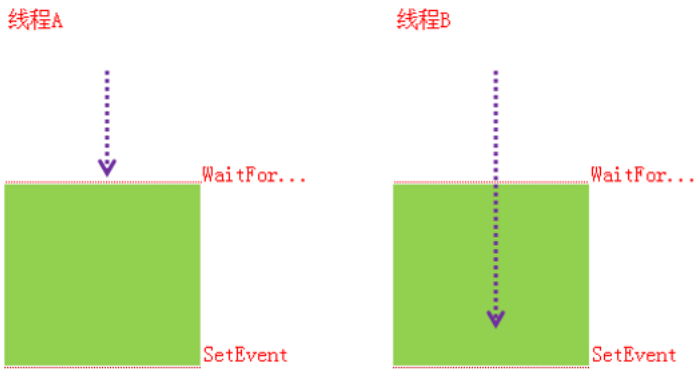
接着我们需要分析一下KeWaitForSingleObject函数，来看下它对不同的事件对象类型做的不同处理，通过IDA打开Ntoskrnl.exe即可。如下图所示，我们可以看见当事件对象的Type为0时（即通知类型事件对象），WaitForSingleObject函数在处理时**并不会修改SignalState**，因此每个等待该对象的进程都可以从等待网中摘除，从而唤醒线程继续执行代码；当事件对象的Type为1时（即同步类型事件对象），**WaitForSingleObject函数在处理时修改SignalState为0**，因此只有一个线程能够得到执行。



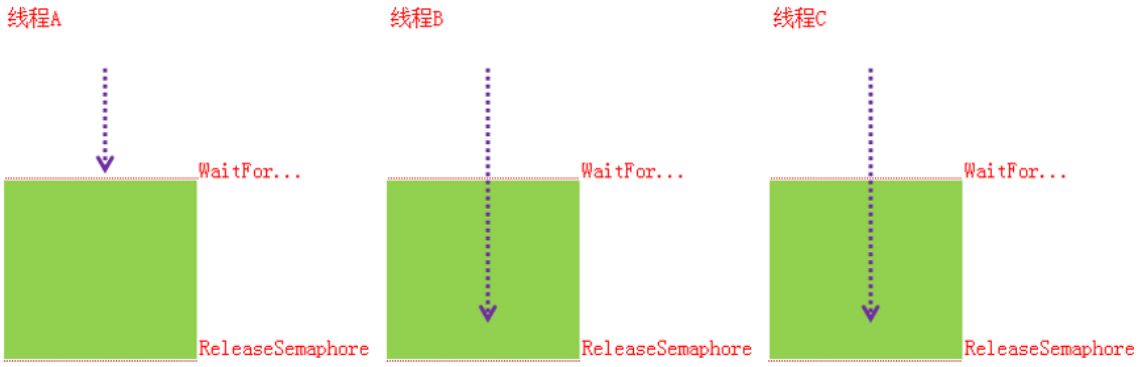
在图中其实我们也可以看见一个判断Type的值为5的情况下会对SignalState的值进行减1，这里Type为5时实际上等待对象的类型就是信号量了，在下一章节的学习中我们就需要来了解信号量。

3.2 信号量

上一章的学习中，我们知道等待事件对象的线程在进入临界区之前会通过调用WaitForSingleObject或者WaitForMultipleObjects来判断当前的事件对象是否有信号（即SignalState的值大于0），只有当事件对象有信号时，才可以进入临界区。 需要说明的是，**这里的临界区指的是广义上的临界区**，即只允许一个线程进入直到退出的一段代码（不是指用EnterCriticalSection和LeaveCriticalSection函数而形成的临界区），这里我们可以认为就是WaitForXX和SetEvent函数所形成的临界区。



信号量与事件的差异在于前者允许多个线程同时进入由WaitForXX和ReleaseSemaphore函数形成的临界区。事件可以在**多个线程想要同时对一个全局变量进行处理时**进行使用，确保只有一个线程会对其进行处理，处理完之后再给到另外一个线程。



信号量用于**解决生产者与消费者问题**，也就是在生产资源与消费者数量不对等的情况下，该如何确保线程同步。如下图所示，有1个生产者线程，每次可以生产出3个资源，此时有5个消费者线程，需要保证不会有2个线程同时消费1个资源。

在这种情况下，如果我们使用事件对象来控制线程的同步就相当的困难，效率也相对较低。例如我们使用同步类型的事件对象，由于**事件对象的SignalState的值只能为0或者1，所以同一时间只有一个消费者线程可以获得资源**，此时效率就很低；如果使用通知类型的事件对象，通知类型对象唤醒的线程，在进入KeWaitForSingleObject的关键循环后，不会修改SignalState的值，**所以5个消费者线程都会被唤醒**，又由于此时仅生产了3个资源，所以会造成性能的浪费。

综上所述，我们可以借助信号量的特点来解决这个问题，而信号量的特点实际上**就是对于信号量对象的SignalState值的修改**。我们首先来看一下如何创建信号量，使用的函数是CreateSemaphore，它的语法格式如下：

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    LONG
lInitialCount,                          // 初始化的SignalState值
    LONG
lMaximumCount,                          // 最大的数量
    LPCTSTR lpName
);
```

信号量对应的结构体如下，该结构体由_DISPATCHER_HEADER和Limit成员组成，**创建信号量对象就是填充这个结构体**，CreateSemaphore函数的lInitialCount填充_DISPATCHER_HEADER.SignalState，lMaximumCount填充Limit。

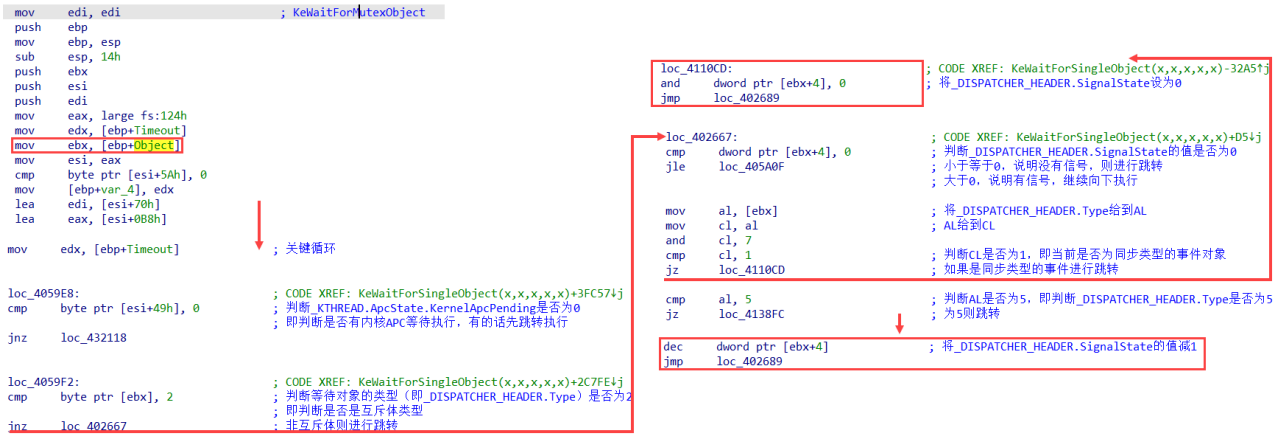
```
kd> dt _KSEMAPHORE
nt!_KSEMAPHORE
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 Limit            : Int4B       // lMaximumCount
kd> dt _DISPATCHER_HEADER
nt!_DISPATCHER_HEADER
   +0x000 Type             : UChar       // 信号量对应的值为5
   +0x001 Absolute         : UChar
   +0x002 Size             : UChar
   +0x003 Inserted         : UChar
   +0x004 SignalState      : Int4B       // lInitialCount
   +0x008 WaitListHead     : _LIST_ENTRY
```

释放信号量对象的函数就是ReleaseSemaphore，它的语法格式如下：

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,        // 信号量的句柄
  LONG lReleaseCount,       // 增加的数量
  LPLONG lpPreviousCount    // 输出增加前的数量
);
```

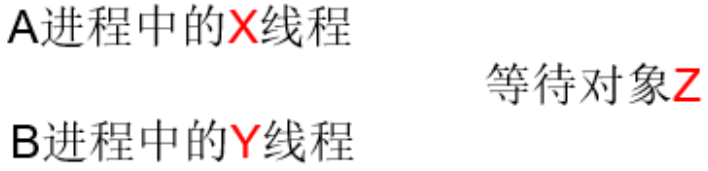
该函数在0环就是KeReleaseSemaphore，如下图所示，我们可以通过IDA分析看见，该函数首先**将原SignalState与lReleaseCount的值相加**，然后重新赋给SignalState，接着通过取线程结构体的方式，将其WaitListEntry链表的Flink、Blink进行位置交换，从而实现从等待链表中摘除当前线程（使用的是我们之前APC章节中提到的KiUnwaitThread函数）。

除了这个操作以外，在WaitForSingleObject函数内，我们也可以看见，之所以信号量可以解决生产消费问题，是因为对SignalState值的修改方式是减1，这样信号量就可以精准控制进入临界区线程的数量，从而实现生产资源与消费者线程的数量对等。



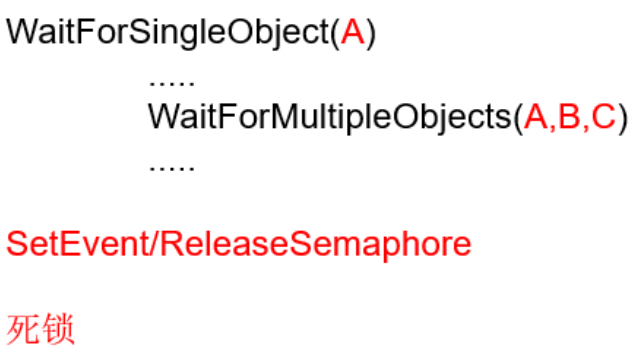
3.3 互斥体

互斥体（Mutant）与事件（Event）和信号量（Semaphore）一样，都可以用来进行线程的同步控制。因为这些都是内核对象，所以我们通过这些对象可以进行跨进程的线程同步控制。



那么可能会有一些极端的情况，如果B进程的Y线程还没有来得及调用修改SignalState的函数（例如SetEvent）就挂掉了，那么**等待对象Z将被遗弃**，这也就意味着X线程将永远等下去，因此为了避免这样的问题，我们可以使用互斥体对象。

除了对象被遗弃问题，互斥体还可以解决重入导致的死锁问题，如下图所示，由于代码设计的问题，在WaitForSingleObject等待A对象后，内部代码又调用了WaitForMultipleObjects等待A对象，**这种情况被称为重入**。当调用一次WaitForSingleObject后，A对象的SignalState变为了0，此时已经没有信号了，当调用WaitForMultipleObjects时，由于被等待对象A是没有信号的，因此代码会永远困在该函数内部，**这种情况叫做死锁**。



互斥体的结构如下所示，我们可以看见它一共有4个成员：

```
kd> dt _KMUTANT
nt!_KMUTANT
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListEntry  : _LIST_ENTRY         // 在_KTHREAD+0x10处有一个MutantListHead字段，指向链表头，链表圈着所有该线程拥有的互斥体对象，该值就表示挂在链表的位置
   +0x018 OwnerThread      : Ptr32 _KTHREAD      // 拥有互斥体的线程
   +0x01c Abandoned        : UChar               // 是否已经被放弃不用
   +0x01d ApcDisable       : UChar               // 是否禁用内核APC
kd> dt _DISPATCHER_HEADER
nt!_DISPATCHER_HEADER
   +0x000 Type             : UChar               // 互斥体的Type值为2
   +0x001 Absolute         : UChar
   +0x002 Size             : UChar
   +0x003 Inserted         : UChar
   +0x004 SignalState      : Int4B
   +0x008 WaitListHead     : _LIST_ENTRY
```

互斥体是通过CreateMutex函数创建的，它的语法格式如下：

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  BOOL bInitialOwner,                          // 表示当前创建的互斥体是否属于当前线程，互斥体结构体成员的OwnerThread就由它决定
  LPCTSTR lpName                               // 对象名称
);
```

CreateMutex到0环的会执行KeInitianlizeMutant函数，该函数的作用就是初始化互斥体，主要有以下几个内容：

```
MUTANT.Header.Type=2;
```

```
MUTANT.Header.SignalState=bInitialOwner ? 0 : 1;
MUTANT.OwnerThread=bInitialOwner ? 当前线程 : NULL;
MUTANT.Abandoned=0;
MUTANT.ApcDisable=0;

if(bInitialOwner==TRUE)
{
    // 将当前互斥体挂入到当前线程的互斥体链表
    // 即_KTHREAD+0x10 -> MutantListHead
}
```

基本的语法和概念了解之后，我们先来看一下为什么互斥体可以重入，**互斥体有一个成员OwnerThread，它就是解决重入的关键**。若一个互斥体被创建时，它的OwnerThread字段不为空，创建它的线程即为互斥体的所属线程。此时，初始化的互斥体SignalState字段被设置为0，也就是没有信号，这个时候别的线程是没法使用这个互斥体的。但是创建它的线程仍然可以使用，**并且可以重复使用0x8000000次**，这也是为什么互斥体可以重入的原因，因为创建它的线程可以在没有信号的情况下使用互斥体，至于为何创建它的线程在互斥体没有信号的情况下也可以使用。

我可以来到WaitForSingleObject看一下，如下图所示，会先**判断等待对象类型**，若是互斥体，继续执行；**接着会判断SignalState的值**，即是否有信号，如果有信号，就跳转；如果没有信号，**判断当前线程与互斥体所属线程是否相同**，如果相同，**就会跳到和有信号时一样的地方**，即使没有信号，互斥体也可以被它的所属线程使用，这样，对于拥有互斥体的线程，就可以重入该互斥体。

我们接着来到跳转后的代码，发现它会先判断SignalState的值是否与0x80000000相等，只要不等于0x80000000，就可以继续执行，执行到蓝色方框的时候，会给SignalState的值减去1。这里要分两种情况：

1. **SignalState有信号**：SignalState值大于0，任何线程可以等待这个互斥体对象；
2. **SignalState无信号**：SignalState值为小于等于0，说明当前线程一定拥有该互斥体，此时SignalState仍会减1，最多可以减少至0x80000000，也就表示**可以重入0x8000000次**。

综上所述，我们知道在互斥体对象中，我们想要激活等待该对象的线程，只需要满足SignalState大于0，或者互斥体所属为当前线程，即可满足激活条件，因此就**解决了重入死锁**问题。

```
loc_4059E8:                             ; CODE XREF: KeWaitForSingleObject(x,x,x,x,x)+3FC57↓j
cmp     byte ptr [esi+49h], 0           ; 判断_KTHREAD.ApcState.KernelApcPending是否为0
                                        ; 即判断是否有内核APC等待执行，有的话先跳转执行
jnz     loc_432118


loc_4059F2:                             ; CODE XREF: KeWaitForSingleObject(x,x,x,x,x)+2C7FE↓j
cmp     byte ptr [ebx], 2               ; 判断等待对象的类型（即_DISPATCHER_HEADER.Type）是否为2
                                        ; 即判断是否是互斥体类型
jnz     loc_402667                      ; 非互斥体则进行跳转

mov     eax, [ebx+4]                    ; 取SignalState
test    eax, eax                        ; 判断SignalState是否为0
jg      loc_402626                      ; 为0则跳转

cmp     esi, [ebx+18h]                  ; 判断互斥体是否属于当前线程
jz      loc_402626                      ; 如果属于，则跳转


        cmp     eax, 80000000h          ; 判断SignalState的值是否为0x80000000
        jz      loc_44B1AE              ; 是的话则进行跳转

        dec     dword ptr [ebx+4]       ; 将SignalState减去1
                                        ; 如果原值为0，则此时就变成了0xFFFFFFFF
        jnz     short loc_40265F
```

接下我们看一下互斥体是如何**解决等待对象被遗弃**的问题，在处理等待对象遗弃的情况时会用到互斥体的两个成员，即MutantListEntry和Abandoned。当一个线程异常“死亡”时，系统会调用内核函数MmUnloadSystemImage处理后事，它会根据“死亡线程”0x10位置指向的链表头，找到它所拥有的所有互斥体，将这些互斥体的Abandoned成员值设置为1，并对它们调用KeReleaseMutant(X, Y, Abandon, Z)函数（ReleaseMutant的内核函数）。

```
_KMUTANT
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListEntry  : _LIST_ENTRY
   +0x018 OwnerThread      : Ptr32 _KTHREAD
   +0x01c Abandoned        : UChar
   +0x01d ApcDisable       : UChar
```

MutantListEntry：
拥有互斥体线程(KTHREAD+0x010 MutantListHead)是个链表头 圈着所有互斥体
Abandoned：
是否已经被放弃不用

KeReleaseMutant函数正常调用时就会**将SignalState自增1**，如果出现互斥体所属线程突然死亡的情况（即Abandon成员值为1），该函数会将SignalState直接设置为1，并且将互斥体所属线程设置为NULL，**同时把自己从死亡线程的互斥体链表中移除**，这样互斥体便可再为其它线程所使用，**从而解决了等待对象被遗弃问题**。

至此，在互斥体中仍有一个成员我们不了解（即ApcDisable），有这个成员的存在是因为互斥体分为两种，即用户、内核下使用的互斥体，它们的区别如下：

```
> 用户互斥体
  + 结构名：Mutant（在3环被创建）
  + 对应内核函数：NtCreateMutant
  + ApcDisable：0

> 内核互斥体
  + 结构名：Mutex（在0环被创建）
  + 对应内核函数：NtCreateMutex
  + ApcDisable：1
```

用户互斥体与内核互斥体结构名不同，但是结构体相同，主要的区别在于ApcDisable成员。用户互斥体是允许内核APC执行的，但是内核互斥体是不允许内核APC进行执行的。在KeWaitForSingleObject的代码中，我们知道它**会根据ApcDisable的值修改_KTHREAD.KernelApcDisable**（即正在使用互斥体的线程），若ApcDisable值为0，则KernelApcDisable的值不会发生改变；若ApcDisable值为1，则KernelApcDisable的值将会减1，此时KernelApcDisable将会是一个不为0的值，内核APC将会被禁用（根据内核APC执行过程，若KernelApcDisable的值不为0，内核APC将会被禁用）。

```
loc_402626:                             ; CODE XREF: KeWaitForSingleObject(x,x,x,x,x)+E0↓j
                                        ; KeWaitForSingleObject(x,x,x,x,x)+E9↓j
cmp     eax, 80000000h                  ; 判断SignalState的值是否为0x80000000
jz      loc_44B1AE                      ; 是的话则进行跳转

dec     dword ptr [ebx+4]               ; 将SignalState减去1
                                        ; 如果原值为0，则此时就变成了0xFFFFFFFF
jnz     short loc_40265F

movzx   eax, byte ptr [ebx+1Dh]         ; _KMUTANT.ApcDisable
sub     [esi+0D4h], eax                 ; _KTHREAD.KernelApcDisable -= _KMUTANT.ApcDisable
cmp     byte ptr [ebx+1Ch], 1           ; 判断_KMUTANTA.Abandoned的是否为1
mov     [ebx+18h], esi                  ; 设置_KMUTANTA.OwnerThread
jz      loc_43209F                      ; 若为1则表示该等待对象被遗弃，即跳转
```