

1 多核同步

我们在使用Inline Hook技术时，如果你的系统环境是多核时就可能会出现一些情况导致系统蓝屏，如果要了解这个原因，我们就需要了解多核同步。

1.1 临界区

1.1.1 并发与同步

在了解临界区之前我们需要补充一些前置知识，以此作为铺垫再进行学习。我们需要了解并发与同步的概念：并发是指多个线程同时执行，在单核的情况下并不是真正的同时执行而是分时执行，多核的情况下就可以在某一个时刻同时多个线程执行。同步则是保证在并发执行的环境中各个线程可以有序的执行，无论是单核或多核的环境。

1.1.2 单行指令同步

如下所示代码，有一个全局变量dwVal，在线程中执行的代码是对dwVal的自增：

```
1  DWORD dwVal = 0; // 全局变量
2
3  // 线程执行的代码
4  dwVal++;
```

以C++的角度来看，上面的代码dwVal的自增只有一行指令，所以多个线程去调用不会存在问题，但实际并不是这样的，因为程序的执行是以汇编指令出发的，自增的汇编代码如下：

```
1  mov eax, [0x12345678]
2  add eax, 1
3  mov [0x12345678], eax
```

所以我们试想一下当A线程去执行自增代码时，走到了ADD指令，这时候发生了线程的切换，B线程执行完自增代码，此时dwVal的值就是1，线程在切换回A，A线程将值填过去，dwVal的值仍然是1，但从代码实现的本意上来看，自增执行了两次那dwVal的值就应该是2。

既然单行代码会存在问题，那么我们尝试将其变成单行汇编指令是否就可以避免这个问题呢？如下汇编所示：

```
1  INC DWORD PTR DS:[0x12345678]
```

这条汇编指令只有一行，即使出现线程切换也不会造成上述的情况，但也不能完全避免。因为在单核的情况下是分时执行，所以这条指令是没问题的，但是在多核的情况下可以在某一个时刻同时多个线程执行，因此也会出现不同线程同时执行这条指令的情况，也会产生上述的问题。

我们想要真正的解决这类问题，就需要使用LOCK指令，如下所示：

```
1  LOCK INC DWORD PTR DS:[0x12345678]
```

从LOCK指令字面意思来看，我们就知道它是一个锁，它锁的不是CPU而是内存，可以锁定当前指令执行时线程所访问的内存。如上所示代码执行时，会对0x12345678地址处的值进行修改，在LOCK指令的限制下，其它线程是不能访问或修改0x12345678地址处的值的，**只有在这条指令执行完后，其他线程才可以对此地址的值进行访问或者修改。**

这类不可被中断的操作叫做原子操作，Windows页提供了一部分API供用户使用来保证在多核情况下的线程同步。（主要位于Kernel32.dll和Ntdll.dll）

原子操作相关的API:

InterlockedIncrement

InterlockedDecrement

InterlockedExchange

InterlockedCompareExchange

InterlockedExchangeAdd

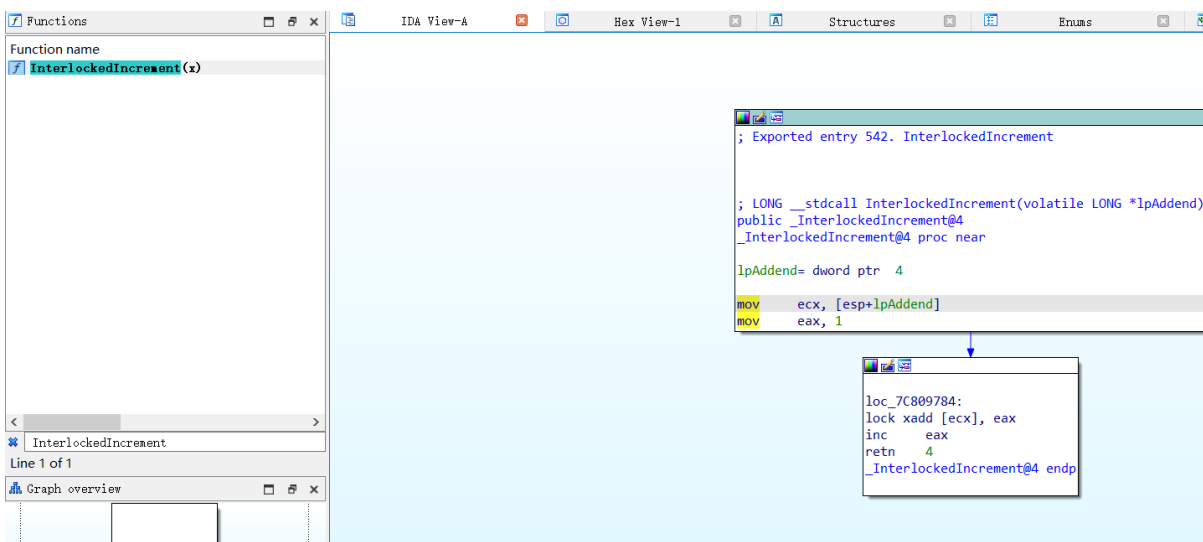
InterlockedFlushSList

InterlockedPopEntrySList

InterlockedPushEntrySList

.....

接下来我们分析Windows提供的原子函数InterlockedIncrement，来理解一下Windows是如何实现多核同步的，通过IDA打开Kernel32.dll，直接在函数列表搜索InterlockedIncrement即可找到：



如上图所示我们可以看见该函数的实现很简单，一共只有5行汇编指令，我们逐行来分析一下：

1. 把要操作的全局变量的地址取出，给到ECX，后续就可以通过[ECX]来访问全局变量的值；
2. 将EAX的值设为1；
3. 除了LOCK指令的原子操作外，第三行指令还用了XADD，该指令的意思就是交换相加，也就表示这里会将EAX的值存入到[ECX]中，然后将[ECX]的值存入到EAX中，接着再执行加法；
4. EAX的值加1，此时EAX的值就与[ECX]的值一致了，它的作用就是用于函数返回值；
5. 平衡堆栈返回。

从本质上来看就是用LOCK指令完成一个原子操作以保证多核状态下的线程同步。

1.1.3 多行指令同步

我们上述所学习的是单行指令下通过LOCK来锁定内存，**但是在多行指令的情况下并不能每行都使用这个指令**，我们来看一下如下的多行指令：

```

1  LOCK INC DWORD PTR DS:[0x12345678]
2  LOCK INC DWORD PTR DS:[0x23456789]
3  LOCK INC DWORD PTR DS:[0x34567890]

```

执行这个多行指令，线程A在对0x12345678地址上的值进行修改时，发生了线程切换，切换到了线程B，虽然线程B不能对0x12345678上的值进行修改，但是线程B可以对0x23456789地址上的值进行修改，如果修改时、又发生线程切换，切回线程A，线程B就锁住了0x23456789地址，不让线程A去修改0x23456789地址处的值，那么这样就还是没有办法保证多行指令的同步，因此我们就需要学习一个新的概念：临界区。

临界区是一个概念，即这个区域一次只允许一个线程进入，直至线程离开才允许下一个线程进入，虽然Windows也提供了临界区，但也不影响我们自己按照这个概念去实现自己的临界区。

临界区实现起来很简单，本质上就是加一个锁，例如我们可以设置一个全局变量，当线程进入时设为1，当线程离开时设为0，当该全局变量为0时才允许进入，在临界区内的指令就是原子操作，否则不允许。

```

DWORD dwFlag = 0;    //实现临界区的方式就是加锁
                      //锁：全局变量 进去加一 出去减一

```

```

if(dwFlag == 0)      //进入临界区
{
    dwFlag = 1
    .....
    .....
    .....

    dwFlag = 0      //离开临界区
}

```

但这样看似解决了多行指令的同步问题，但实际又出了一个新的问题，即在dwFlag的值设为1之前发生了线程切换，其他的线程仍然可以通过判断进入临界区，这样临界区的作用就失效了。那么这里的问题就很明显是发生在上了锁的顺序上，所以我们可以换一种方式，先修改锁的值，然后再判断进入临界区。

如下图所示的伪代码，首先赋予EAX为1，接着通过交换相加的方式，将全局变量的值加1，将原全局变量的值0给到EAX，而后判断EAX的值是否为0：如果是0则可以进入临界区，执行完原子操作之后，离开临界区将全局变量的值减1；如果不是则将全局变量的值减1，调用sleep函数进入线程等待，一段时间后，在跳回进入临界区的地方重新判断。

全局变量：Flag = 0

进入临界区：

Lab:

```

mov eax,1
lock xadd [Flag],eax
cmp eax,0
jz endLab
dec [Flag]
//线程等待Sleep..
jmp lab
endLab:
ret

```

离开临界区：

```
lock dec [Flag]
```

1.2 自旋锁

在上一章节中，我们了解了多行指令通过临界区的方式在多核环境下进行线程同步，但那只是简单了解，本章我们需要了解在Windows下实现多核同步的机制：自旋锁。

我们要了解自旋锁需要根据内核文件去分析，在Windows操作系统下不同版本有着不同的内核文件。

单核：

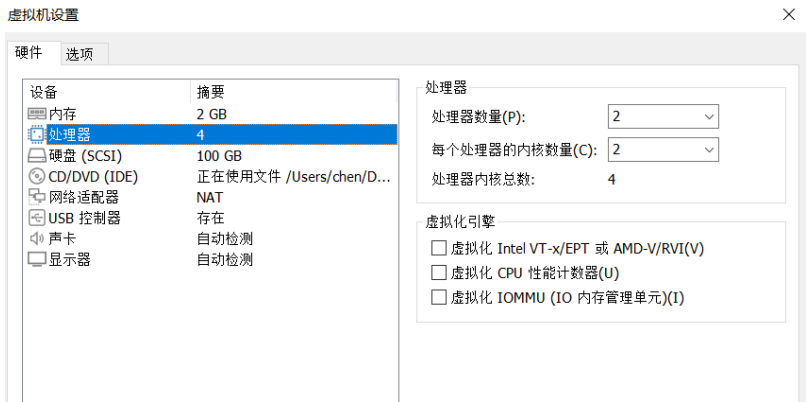
ntkrnlpa.exe	2-9-9-12分页
ntoskrnl.exe	10-10-12分页

多核：

ntkrnlpa.exe	2-9-9-12分页
ntoskrnl.exe	10-10-12分页

系统在安装时会根据你的硬件环境，如Intel的多核CPU，系统就会将ntkrnlmp.exe拷贝成ntoskrnl.exe，虽然与单核时的名字是一样的，但真正的源文件不同。

我们目前虚拟机是单核配置，想要获取多核情况下内核文件，就需要重新设置虚拟机的处理器核的数量，通过VMWare进行配置即可。



然后进入虚拟机，拷贝出来ntoskrnl.exe重命名为ntkrnlmp.exe，再通过IDA打开，此时IDA就会去调用ntkrnlmp.exe的符号文件来解析。

通过IDA我们可以看见在线程切换的关键函数SwapContext中，多核模式下在函数开头多了两个函数的使用，即：KeAcquireQueuedSpinLockAtDpcLevel与KeReleaseQueuedSpinLockFromDpcLevel。

```

.text:00405E77 ; char __usercall SwapContext@<al>(int *@<ebx>, int@<edi>, int@<esi>)
.text:00405E77 SwapContext proc near ; CODE XREF: KiUnlockDispatcherDatabase(x)+99↑p
.text:00405E77 ; KiSwapContext(x)+2A↑p
.text:00405E77 ; KiDispatchInterrupt()+A2↑p
.text:00405E77
.text:00405E77 ; FUNCTION CHUNK AT .text:00405FFE SIZE 00000033 BYTES
.text:00405E77
.text:00405E77 or     cl, cl
.text:00405E79 mov     byte ptr es:[esi+20h], 2
.text:00405E7E pushf
.text:00405E7F lea     ecx, [ebx+540h]
.text:00405E85 call    @KeAcquireQueuedSpinLockAtDpcLevel@4 ; KeAcquireQueuedSpinLockAtDpcLevel(x)
.text:00405E85
.text:00405E8A lea     ecx, [ebx+538h]
.text:00405E90 call    @KeReleaseQueuedSpinLockFromDpcLevel@4 ; KeReleaseQueuedSpinLockFromDpcLevel(x)

```

了解Windows自旋锁，我们可以来看这个函数：KeAcquire**SpinLock**AtDpcLevel，在函数名中带有Spinlock的都
与自旋锁有关。

我们可以通过IDA看见，在KeAcquire**SpinLock**AtDpcLevel函数内的指令很简单：

1. 获取传递过来的参数，先进行LOCK原子操作，然后BTS指令会先判断[ecx]值是否为0，为0则将CF位设为1，为1则CF位设为0，接着将[ecx]中下标为0的位置值设1；
2. 当CF位为0 ([ecx]为1) 则跳转，当CF位为1 ([ecx]为0) 则返回；
3. 判断[ecx]值是否为1，不为1则ZF位为1，为1则ZF位为0；
4. 当ZF位为1 ([ecx]为0) 则跳转，ZF位为0 ([ecx]为1) 则使用降温指令（PAUSE），短时间内让CPU功率降低。

```

mov     ecx, [esp+arg_0] ; 获取传递过来的参数

loc_40BA4F: ; CODE XREF: KeAcquireSpinLockAtDpcLevel(x)+14↑j
lock bts dword ptr [ecx], 0 ; LOCK原子操作，BTS指令会先判断[ecx]值是否为0，为0则将CF位设为1，为1则CF位设为0，接着将[ecx]中下标为0的位置值设1
jb      short loc_40BA59 ; CF位为0 ([ecx]为1) 则跳转

ret     4 ; CF位为1 ([ecx]为0) 则返回

; -----
loc_40BA59: ; CODE XREF: KeAcquireSpinLockAtDpcLevel(x)+9↑j
; KeAcquireSpinLockAtDpcLevel(x)+18↑j
test     dword ptr [ecx], 1 ; 判断[ecx]值是否为1，不为1则ZF位为1，为1则ZF位为0
jz       short loc_40BA4F ; ZF位为1 ([ecx]为0) 则跳转

pause ; ZF位为0 ([ecx]为1) 则使用降温指令，短时间内让CPU功率降低
jmp      short loc_40BA59

_KeAcquireSpinLockAtDpcLevel@4 endp

```

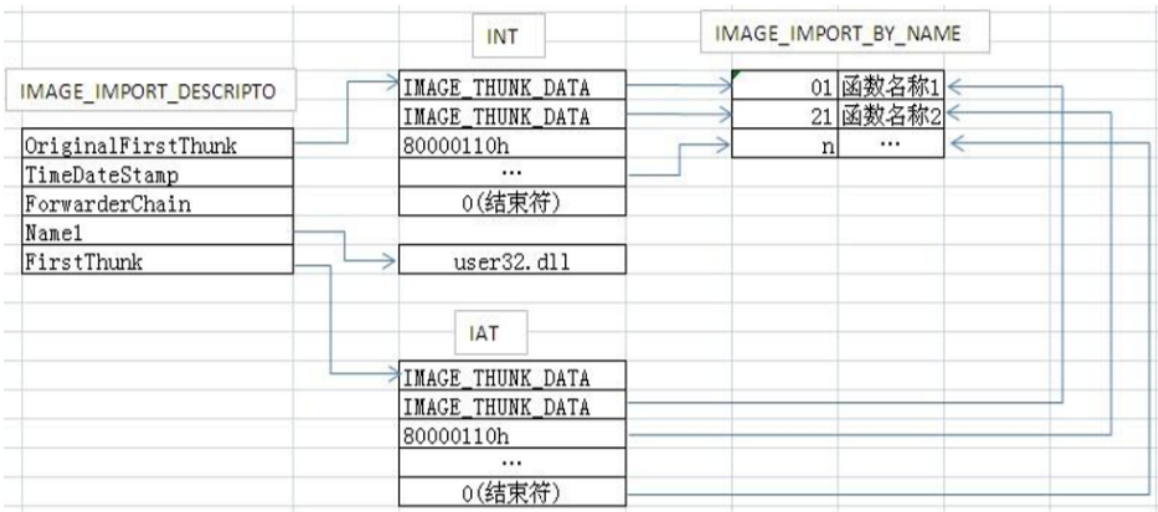
我们可以看见当多核模式下，A线程（1核）执行该函数之后获得了自旋锁，此时B线程（2核）加入执行，那么B线程就需要原地等待在那转一会，等待自旋锁被释放，也就是为空闲状态。因此，自旋锁只对多核有意义，自旋锁与临界区、事件、互斥体一样，都是一种同步机制，都可以让当前线程处于等待状态，**区别在于自旋锁不用切换线程，更加轻量。**

1.3 重载内核

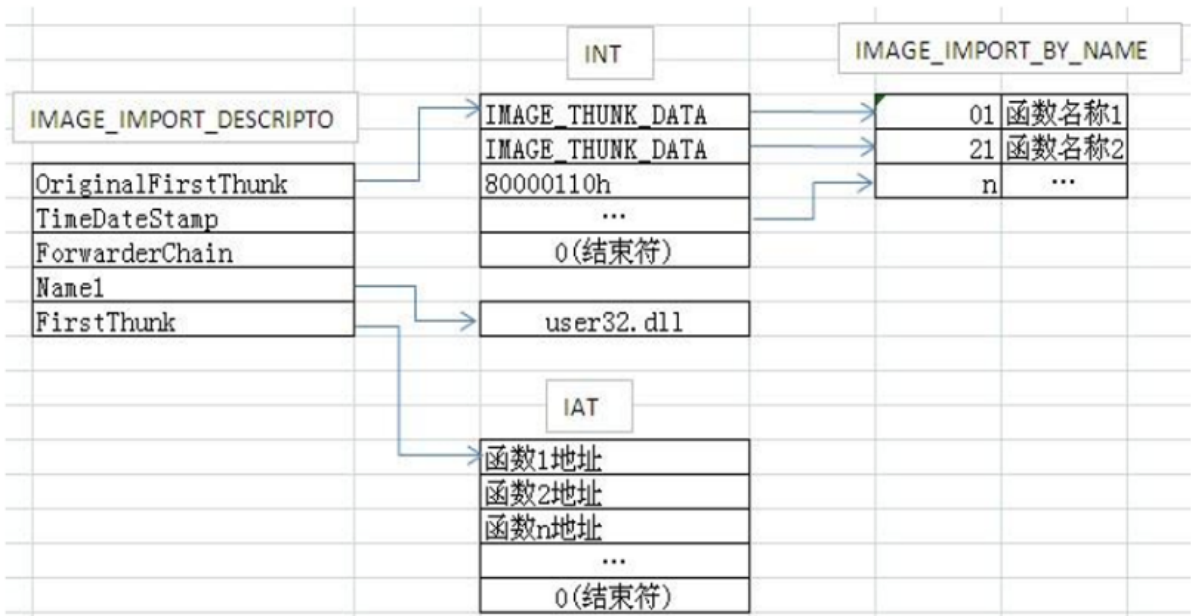
内核中会有很多的函数被层层Hook，要想绕过Hook我们就可以使用重载内核的方式。重载内核跟权限（R3、R0）和内核没关系，重载内核的本质就是将内核文件（PE文件）自己拿过来在按内存格式要求（**按PE头成员SectionAlignment的值在内存中进行对齐**）放入内存，然后自己使用。

但实际操作并没有这么简单，我们需要考虑当内核文件加载进来之后它申请到的内存基址是否与ImageBase一致，如果不一致的情况下，我们就需要去修改重定位表，这样才可以正确的使用全局变量；除此以外，我们还需要去修复IAT表，如下两张图所示，是Windows在PE文件加载前后IAT表存储内容的变化，**如果我们不去修复IAT表使得其存储的内容执向真正的函数地址，那么当我们使用某个函数时就会出错**，因为它指向的是一个IMAGE_THUNK_DATA结构体。

PE文件加载前:



PE文件加载后:



重载完成之后我们是否就可以使用了呢，实际上并不能，我们在系统调用章节学习时了解到3环API本质上就是系统调用，进入0环之后执行的是KiFastCallEntry，该函数是需要根据系统调用号在系统服务表中找到函数地址，因此我们想要能够真正的使用重载后的内存，也需要自己去山寨一份系统服务表，将里面的函数地址指向我们新的内核对应地址。

接着我们还需要去Hook KiFastCallEntry，使得执行该函数时它所寻找的系统服务表是我们山寨的那一份，这样我们才可以使用重载后的内核。

1.3.1 总结

从本质来说并不需要所谓的“重载内核”，只要了解PE基础，对于我们来说一切都是“重载模块”，我们需要就是通过自己的代码来实现Windows加载模块的过程。

虽然通过重载内核我们可以绕过内核上的Hook，但改动太大，即使抹去PE指纹也无法完全隐形，最好的办法不是重载内核，而是需要什么函数自己来实现。