说完了内核里的三大类基本操作，就说说一些常用的其它操作，这些操作难以归类，但是肯定很有用处。本章节的内容也是会动态更新的。

1.遍历链表。内核里有很多数据结构，但它们并不是孤立的，内核使用双向链表把它们像糖葫芦一样给串了起来。所以遍历双向链表能获得很多重要的内核数据。举个简单的例子，驱动对象 DriverObject 就是使用双向链表给串起来的，遍历这个链表就可以枚举内核里所有的驱动。示例代码如下：

```
//传入驱动自身的 DriverObject
VOID EnumDriver(PDRIVER_OBJECT pDriverObject)
{
	PKLDR_DATA_TABLE_ENTRY
entry=(PKLDR_DATA_TABLE_ENTRY)pDriverObject->DriverSection;
	PKLDR_DATA_TABLE_ENTRY firstentry;
	ULONG64 pDrvBase=0;
	KIRQL OldIrql;
	firstentry = entry;
	//当发现又找到自己时跳出循环，否则成了死循环。
	while((PKLDR_DATA_TABLE_ENTRY)entry->InLoadOrderLinks.Flink != firstentry)
	{
		DbgPrint("BASE=%p\tPATH=%wZ",entry->DllBase,entry->FullDllName);
		entry = (PKLDR_DATA_TABLE_ENTRY)entry->InLoadOrderLinks.Flink;
	}
}
```

2.等待。这个等于 RING3 的 Sleep 函数了。

```
#define DELAY_ONE_MICROSECOND    (-10)
#define DELAY_ONE_MILLISECOND    (DELAY_ONE_MICROSECOND*1000)
VOID MySleep(LONG msec)
{
	LARGE_INTEGER my_interval;
	my_interval.QuadPart = DELAY_ONE_MILLISECOND;
	my_interval.QuadPart *= msec;
	KeDelayExecutionThread(KernelMode,0,&my_interval);
}
```

3.同步。这个可以理解成是“条件等待”。常用的是 KeWaitForSingleObject、KeInitializeEvent、KeSetEvent 这几个函数。为了方便讲解，这个的示例代码与“内核线程”放在一起。先把这几个函数的原型贴出来。

```
NTSTATUS KeWaitForSingleObject
(
  _In_      PVOID Object,
  _In_      KWAIT_REASON WaitReason,
  _In_      KPROCESSOR_MODE WaitMode,
  _In_      BOOLEAN Alertable,
```

```
    _In_opt_    PLARGE_INTEGER Timeout
);
```

```
VOID KeInitializeEvent
(
    _Out_    PRKEVENT Event,
    _In_     EVENT_TYPE Type,
    _In_     BOOLEAN State
);
```

```
LONG KeSetEvent
(
    _Inout_    PRKEVENT Event,
    _In_       KPRIORITY Increment,
    _In_       BOOLEAN Wait
);
```

4.获得系统版本号。内核编程难免使用硬编码，以及使用一些高版本系统才出现的函数。为了使得驱动能在低版本的系统上正常运行，就需要根据不同系统做不同处理了。

```
VOID GetVersion()
{
    ULONG NtBuildNumber;
    RTL_OSVERSIONINFOW osi;
    osi.dwOSVersionInfoSize=sizeof(RTL_OSVERSIONINFOW);
    RtlFillMemory(&osi,sizeof(RTL_OSVERSIONINFOW),0);
    RtlGetVersion(&osi);
    NtBuildNumber=osi.dwBuildNumber;
    DbgPrint("NtBuildNumber: %ld\n",NtBuildNumber);
    return NtBuildNumber;
}
```

5.获得系统时间。在内核里获得系统时间的是标准时间（GMT+0），转换成本地时间还需要进行转换。此功能在发布测试版软件的时候特别有用，限制人们只能在指定时间之前使用。

```
VOID MyGetCurrentTime()
{
    LARGE_INTEGER CurrentTime;
    LARGE_INTEGER LocalTime;
    TIME_FIELDS     TimeFiled;
    // 这里得到的其实是格林威治时间
    KeQuerySystemTime(&CurrentTime);
    // 转换成本地时间
    ExSystemTimeToLocalTime(&CurrentTime, &LocalTime);
    // 把时间转换为容易理解的形式
    RtlTimeToTimeFields(&LocalTime, &TimeFiled);
    DbgPrint("[TimeTest] NowTime : %4d-%2d-%2d %2d:%2d:%2d",
                TimeFiled.Year, TimeFiled.Month, TimeFiled.Day,
```

```
                TimeFiled.Hour, TimeFiled.Minute, TimeFiled.Second);
}
```

6.内核线程。内核线程就是名义上属于 SYSTEM 进程的线程。比如说你要做坏事，却让 SYSTEM 进程背黑锅，是一件很爽的事情。内核线程还有几个特点：1. PreviousMode 是 KernelMode，可以直接调用 Nt 开头的内核函数（Nt 开头的内核函数会检查 PreviousMode，如果 PreviousMode 不是 KernelMode，就会拒绝服务。有些人喜欢直接修改 ETHREAD 里的这个值，但我个人觉得这么改不妥当）。2.内核线程不会自己结束，必须调用 PsTerminateSystemThread 才能被动结束。以下是例子，**同时演示了等待、同步和内核线程的使用**。

```
KEVENT kEvent; //事件
//线程函数
VOID MyThreadFunc(IN PVOID context)
{
	PUNICODE_STRING str = (PUNICODE_STRING)context;
	DbgPrint("Kernel thread running: %wZ\n", str);
	DbgPrint("Wait 3s!\n");
	MySleep(3000);
	DbgPrint("Kernel thread exit!\n");
	KeSetEvent(&kEvent, 0, TRUE);
	PsTerminateSystemThread(STATUS_SUCCESS);
}
//创建线程的函数
VOID CreateThreadTest()
{
	HANDLE		hThread;
	UNICODE_STRING ustrTest = RTL_CONSTANT_STRING(L"This is a string for test!");
	NTSTATUS status;
	// 初始化事件
	KeInitializeEvent(&kEvent, SynchronizationEvent, FALSE);
	status = PsCreateSystemThread(&hThread, 0, NULL, NULL, NULL, MyThreadFunc,
(PVOID)&ustrTest);
	if (!NT_SUCCESS(status))
	{
		DbgPrint("PsCreateSystemThread failed!");
		return;
	}
	ZwClose(hThread);
	// 等待事件
	KeWaitForSingleObject(&kEvent, Executive, KernelMode, FALSE, NULL);
	DbgPrint("CreateThreadTest OVER!\n");
}
```

如果线程创建成功，会依次输出以下字符串：

```
Kernel thread running: This is a string for test!
Wait 3s!
```

```
Kernel thread exit!
CreateThreadTest OVER!
```

7.强制重启计算机。在内核里直接使用 OUT 指令就能强制重启计算机而不可能被任何钩子拦截。此代码可以用在反调试里。

```
VOID ForceReboot()
{
	typedef void (__fastcall *FCRB)(void);
	/*
	mov al, 0FEh
	out 64h, al
	ret
	*/
	FCRB fcrb=NULL;
	UCHAR shellcode[6]="\xB0\xFE\xE6\x64\xC3";
	fcrb=ExAllocatePool(NonPagedPool,5);
	memcpy(fcrb,shellcode,5);
	fcrb();
}
```

8.强制关闭计算机。在内核里直接使用 OUT 指令就能强制关闭计算机而不可能被任何钩子拦截。此代码可以用在反调试里。

```
VOID ForceShutdown()
{
	typedef void (__fastcall *FCRB)(void);
	/*
	mov ax,2001h
	mov dx,1004h
	out dx,ax
	ret
	*/
	FCRB fcrb=NULL;
	UCHAR shellcode[12]="\x66\xB8\x01\x20\x66\xBA\x04\x10\x66\xEF\xC3";
	fcrb=ExAllocatePool(NonPagedPool,11);
	memcpy(fcrb,shellcode,11);
	fcrb();
}
```

课后作业：无。