通俗解析IRP和I/O设备栈在内核程序中的作用（转自看雪）

转载

z6470975 于 2012-01-09 13:08:09 发布 1624 收藏 7
分类专栏： windows内核 文章标签： struct object alignment io integer ddk


windows内核 专栏收录该内容
21 篇文章 0 订阅
订阅专栏

正文：
言归正传，所有的I/O请求都是以IRP（I/O请求包）的形式来提交的，同时内核程序的所有分发函数（Dispatch Function）的第二个参数都是

PIRP（也即是指向IRP的指针）。为了说明问题，防止跳跃性太大，先解释几个名词（都按自己理解的），可以帮助会的人复习，不会的人学习

1：DRIVER_OBJECT：
驱动对象，由即插即用管理创建和传递到DriverEntry，作为DriverEntry的第一个参数,函数原型如下：

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,
         IN PUNICODE_STRING pRegistryPath);
```

代表了一个内核驱动程序，一个驱动程序有且只有一个DRIVER_OBJECT，在wdm.h里有定义，也可以使用WinDbg查看其结构（保留最重要的

，并作了注释）：

```
DRIVER_OBJECT//
typedef struct _DRIVER_OBJECT {
   。。。。
    PDEVICE_OBJECT DeviceObject;//指向驱动程序中创建的第一个设备对象
。。。
    PDRIVER_EXTENSION DriverExtension;//驱动扩展，里面有一个AddDevice，对于WDM程序很重要
。。。
   PDRIVER_STARTIO DriverStartIo;//很重要，IRP串行化
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];//0x1E个分发函数，是内核驱动程序的核心，可以理解

只需要告诉驱动程序的分发函数名，并且编写函数实体，由系统I/O管理器完成调用）。若有机会，我会专门写一篇关于回调函数的文章

} DRIVER_OBJECT;
typedef struct _DRIVER_OBJECT *PDRIVER_OBJECT;
```

2：DEVICE_OBJECT：
设备对象，和IRP一样重要，一开始说道的每个分发函数有两个参数，第二个是PIRP，那么第一个就是PDEVICE_OBJECT，如下所示：

```
NTSTATUS WdmDefaultDispachRoutine(IN PDEVICE_OBJECT pFdo,
          IN PIRP pIrp)
```

设备对象的重要在于：一个驱动程序光有驱动对象是玩不转的，为什么？若是编写过Win32应用程序都知道，消息的重要性，但是光有消息而没

有窗口也是玩不转的，窗口是用来接收消息的。同样的道理，设备对象是唯一能够接受IRP的实体（若是理解了这个，那么你就将迈进了内核编

程的门槛）。不干他如何如何的重要，先看一下结构，也只列出重要的（其他的也很重要，感兴趣的可以自己查看），而且都有注释：

代码:

```
///DEVICE_OBJECT//
typedef struct _DEVICE_OBJECT {
........
    struct _DRIVER_OBJECT *DriverObject;//指出设备对象归属于哪个驱动程序（驱动对象）
    struct _DEVICE_OBJECT *NextDevice;//下一个设备对象（一个驱动程序可以创建多个设备对象），这个设备对象是同一层

设备栈）
    struct _DEVICE_OBJECT *AttachedDevice;//指向下一层驱动程序的设备对象（可以理解被挂载的设备对象）
    struct _IRP *CurrentIrp;//使用IRP串行化时很重要，用来决策当前IRP完成还是挂起等

    CCHAR StackSize;//设备栈的个数

     PVOID DeviceExtension;// 设备扩展，是一个自定义的大小的结构，很重要。因为驱动程序中的每个分发例程都有可能工

程和进程中，所以全局变量很危险，为了防止这一点，把全局变量都定义到设备扩展里，而设备对象又作为每个分发例程都带有设备对

以都可以通过设备对象得到设备扩展里面的信息（在这一点上，个人认为有点像C++的封装性
} DEVICE_OBJECT;
typedef struct _DEVICE_OBJECT *PDEVICE_OBJECT;
```

3、IRP：
I/O请求包，很重要，若把设备对象理解为窗口，那么IRP就可以理解为消息。I/O请求包一般由I/O管理器构建，但是也可以用户在程序中使用DDK函数来

构建（Io开头的函数），后面重点讲解这个，也是这篇文章的核心。IRP是一个复杂的结构体（比之MSG复杂多了），看部分结构（本来想放图

，不过需要超链接，很麻烦）。

代码:

```
```

```
/IRP
typedef struct _IRP {

    PMDL MdlAddress;//MDL地址，内存描述符表。用来建立一块虚拟地址空间与物理地址页面之间的映射（具体有机会我会专门

拟内存和实际物理内存在内核程序中的使用）。实际应用中，在使用DO_DIRECT_IO方式和应用程序读写数据时采取的一种内存映射的

度快，一般设备驱动程序都采用这种方式通信，尤其是WriteFile和ReadFile中

/*下面是一个共用体，很重要，联合的IRP，里面的SystemBuffer是指向应用层传递来的数据，采用的是DO_BUFFER_IO缓冲区拷贝

速度慢，一般在DeviceIoControl小数据使用*/
    union {
        struct _IRP *MasterIrp;
        LONG IrpCount;
        PVOID SystemBuffer;
    } AssociatedIrp;

 /*里面有两个结构，一个Status是IRP完成的状态，一个是Infotmation存放数据传输的个数*/
    IO_STATUS_BLOCK IoStatus;

    CHAR StackCount;//栈的个数，可以由设备对象中StackSize的值决定

    CHAR CurrentLocation;//当前的设备栈位置，很重要，过滤器驱动需要判断是否大于0，否则直接蓝屏处理

    PKEVENT UserEvent;//构建IRP时很重要，同步事件，后面会讲到。


       } Overlay;

    PVOID UserBuffer;//用户缓冲区，第三种方式和应用程序共享数据。这种速度最快，但也是最不安全，内核程序直接读取用

须保证在相同设备上下文中访问才不会出错。

    union {

        struct {

            struct {

                union {


                    struct _IO_STACK_LOCATION *CurrentStackLocation;//IO设备栈指针，他是一个设备栈数组。说

为了讲他。后面会详细讲解。

                };
            };

        } Overlay;


    } Tail;

} IRP, *PIRP;
```

由上面的结构可得知，创建一个IRP时，会同时带着创建一个和他关联的IO_STACK_LOCATION数组，他们形成一个栈的形式（满足先入后出的原

则），具体关于IO_STACK_LOCATION 看下一个。
4、IO_STACK_LOCATION ：
I/O设备栈。结构体很复杂，但是里面的东西包罗万象，有些字段和IRP的结构重复。那么IO设备栈有什么作用
先看一下他的结构：

代码:

```
IO_STACK_LOCATION/
typedef struct _IO_STACK_LOCATION {
    UCHAR MajorFunction;//IRP主功能码
    UCHAR MinorFunction;//IRP次功能码，尤其是Pnp的IRP尤为重要
    UCHAR Flags;
    UCHAR Control;//DeviceControl的控制码


/*以下是一个联合体，非常重要，几乎所有的用户API的请求都在这里面体现出来，记录了所有的用户请求信息，例如读写的长度信息

只保留了几个*/
    union {


        struct {
            ULONG Length;
            ULONG POINTER_ALIGNMENT Key;
            LARGE_INTEGER ByteOffset;
        } Read;//NtReadFile（也即是ReadFile的实现的）

        struct {
            ULONG Length;
            ULONG POINTER_ALIGNMENT Key;
            LARGE_INTEGER ByteOffset;
        } Write;//NtWriteFile


        struct {
            ULONG OutputBufferLength;
            ULONG POINTER_ALIGNMENT InputBufferLength;
            ULONG POINTER_ALIGNMENT IoControlCode;
            PVOID Type3InputBuffer;
        } DeviceIoControl;//NtDeviceIoControlFile


        struct {
            PCM_RESOURCE_LIST AllocatedResources;
            PCM_RESOURCE_LIST AllocatedResourcesTranslated;
        } StartDevice;//为什么保留这个，原因在于我自己是做硬件设备驱动的，而这个是启动设备的PnP能够获取到硬件的设

是纯内核开发不需要关心。



        struct {
            PVOID Argument1;
            PVOID Argument2;
            PVOID Argument3;
            PVOID Argument4;
        } Others;//这个的重要性在于，若没有列举的结构都可以用强类型转换这几个字段。很灵活
```

```
    } Parameters;

    PDEVICE_OBJECT DeviceObject;//指向的设备对象，很重要，从设备对象中可以获得驱动对象，然后再得到相应的分发函数

    PFILE_OBJECT FileObject;//文件对象，文件系统之类的信息安全内核编程很重要。

    PIO_COMPLETION_ROUTINE CompletionRoutine;

    PVOID Context;

} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

IO_STACK_LOCATION里面存放着需要处理的IRP信息，联合IRP就可以找到下一个设备驱动处理IRP的分发函数（具体可以参考IoCallDriver()的

反汇编代码，有机会专门讲述）

/

其实说了这么多，上面的都不是本帖子要讲的，但是又不得不说的。下面回到重点：
这里要讲到的其实是驱动程序调用驱动程序以及WDM分层驱动的东西。
不管是I/O 管理器构建的IRP还是自己手动构建的IRP，对IRP的操作都有以下4种情况：

1、直接完成IRP，返回。
2、放入IRP队列中，调用StartIO完成IRP串行化。
3、传递IRP到下一层，由下一层（或者再下一层完成）并且不需要获得IRP完成的情况
4、传递IRP到下一层，同时需要得到获得下一层处理完IRP的信息（键盘过滤器等很多FilterDriver就这么干）

其实可以把第1点和第2点归为一起，都是本层处理，把第4点归结到第3点上，都是下一层处理，但是这两者之间的IO设备栈的处理有点差异。

接下来说明一个驱动程序的创建一般过程：
DriverEntry和AddDevice两个函数由即插即用管理器调用，DriverEntry里面完成分发函数的注册，AddDevice用来创建一个设备对象并且挂载到下一层设备对象上。一般的WDM驱动程序都有一个总线设备驱动程序，负责枚举和分配设备资源。为了说明问题，假设存在设备A和设备B，设备B挂载到设备A上，现在你要创建设备C也要挂载到A上，那么使用IoAttatchDeviceToDeviceStack函数挂载到A时，返回的是B的设备对象。也就是从上往下看IRP的传递是C->B->A的流向。到这里就可以讨论IO设备栈了！！！
回到一开始说的当收到一个IRP时，本层驱动需要做什么处理，4种情况处理IRP不同,如下：

1、直接完成IRP：IoCompleteRequest()
2、放入IRP队列：IoMarkIrpPending()和IoStartPacket()
3、传递IRP到下一层：IoSkipCurrentIrpStackLocation()和IoCallDriver()
4、需要获取完成信息：IoCopyCurrentIrpStackLocationToNext()和IoCallDriver()

到这里也许很多驱动的资料都会这么讲，但为什么需要这么做？？
每一层设备对象都需要一个设备栈，用来保存上一层创建或者传递IRP的信息，使用IoGetCurrentIrpStackLocation()可以获得当前设备栈，也

即获得里面的IRP信息。当前设备首先获得IRP处理的权利，可以完成，可以放入队列，也可以传递到下一层。而是用IoCallDriver完成IRP的传递，可以使用反汇编查看IoCallDriver()或者看DDK文档就会知道，IoCallDriver里面首先会将CurrentLocation减1（汇编指令为dec），然后再去和0进行判断比较跳转，当当前值比0大，那么会去将当前设备栈指针移动到前一个设备栈（结合stack结构的概念，先入后出）。
那么如果当前设备不关心IRP的完成，那么也就不需要当前设备栈，那么为了让底层驱动处理这个IRP，就必须应该把当前设备栈信息告诉底层设备，可以采取两种方式，使用IoSkipCurrentIrpStackLocation()把指针拨回去（和IoCallDriver()方向相反），也就是下一层的设备栈其实为本层的设备栈（速度快，另外一种是拷贝的方式，见下面）。一般不是很关心的IRP都这么做，例如PnP的，电源管理的（记住，电源管理的传递IRP使用的是 PoCallDriver()）。但是作为过滤器驱动，都不会放过

Read和Write的，Rootkit不会，HIPS也不会（现在的攻防问题，也许不再是这么简单的Hook了，但是思想都是一样，看谁最先获得IRP的主控权）。那么IoCopyCurrentIrpStackLocationToNext()的好处就在于，把本层的设备栈拷贝到下一层，那么拷贝完之后呢？？如何第一时间得到下一层完成了IRP，也就是数据已经有效了？？

可以使用同步或者异步方式，同步方式是等待IoCallDriver()返回，异步方式采用的是完成例程和事件通知的方式。一般都是采用完成例程+事件通知方式（具体实现可以参考一些DDK例子），若是在IRP总设置了完成例程，那么下一层驱动程序完成IRP后会直接去调用完成例程（回调函数概念），Hook就达到，你就可以在这个例程中为所欲为了（例如我直接从IRP中获得读缓冲区地址和长度，然后加密数据，想搞破坏的人可能会原封不动的把数据上传，但是他会备份一份，□）。。。。。

前面一直提到了IRP的创建，创建IRP的方式有以下几种：
1、IoAllocateIrp 常用，创建人一类型的IRP。（其实下面的方式都封装了这个函数），但一定记住了，需要IoFreeIrp（一般在完成函数里调用）,否则内存泄露。
2、IoBuildAsynchronousFsdRequest，IoBuildSynchronousFsdRequest。创建 IRP_MJ_PNP，IRP_MJ_READ，IRP_MJ_WRITE，IRP_MJ_FLUSH_BUFFERS，IRP_MJ_SHUTDOWN这些类型的IRP，分同步和异步，异步IRP可以工作在任意设备上下文中。
3、IoBuildDeviceIoControlRequest 编写USB设备驱动时用过。创建 IRP_MJ_INTERNAL_DEVICE_CONTROL和IRP_MJ_DEVICE_CONTROL

原文链接：http://bbs.pediy.com/showthread.php?t=111559