2021/04/27_Windows32位内核_第5课_内核中缓冲区三种通讯方式以及控制回调的编写

笔记本：	Windows32位内核
创建时间：	2021/4/27 星期二 15:04
作者：	ileemi

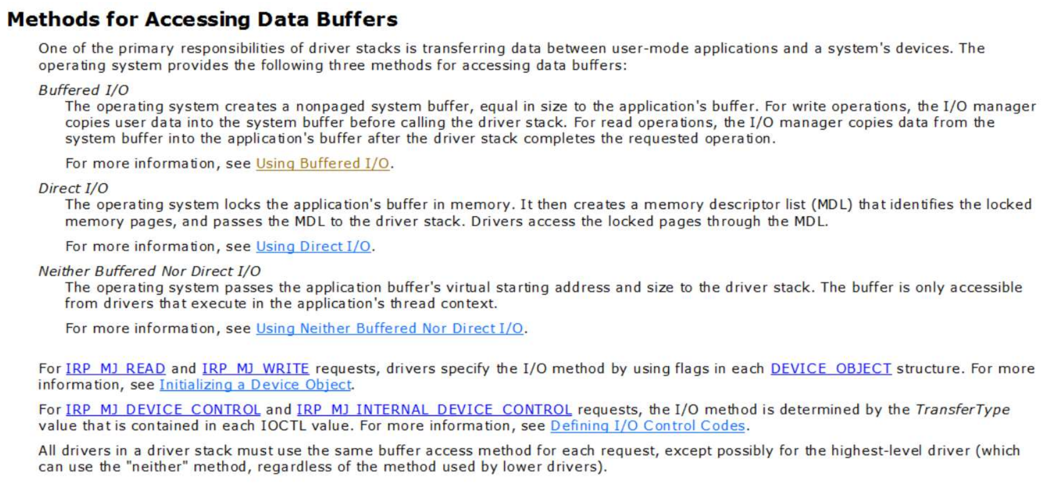
- 访问数据缓冲区的方法
    - 缓冲区方式
    - 直接方式
- 控制回调的编写

IoAllocateIrp

访问数据缓冲区的方法

应用程序和驱动的缓冲区通讯方式（打开、关闭、读、写）：

1. 缓冲区方式：创建设备时会产生设备对象，设备对象的Flags成员用来指定其通讯方式。示例：pDevObj->Flags |= DO_BUFFERED_IO;
2. 直接方式：不需要额外的缓冲区进行操作。
3. 其它方式：(1) 驱动直接访问RING3的地址（进程隔离），需要保证进程不会被切换。
   (2) 保证不会传递内核空间地址
   (3) 保证内存属性一定要正确


Methods for Accessing Data Buffers

One of the primary responsibilities of driver stacks is transferring data between user-mode applications and a system's devices. The operating system provides the following three methods for accessing data buffers:

Buffered I/O
The operating system creates a nonpaged system buffer, equal in size to the application's buffer. For write operations, the I/O manager copies user data into the system buffer before calling the driver stack. For read operations, the I/O manager copies data from the system buffer into the application's buffer after the driver stack completes the requested operation.

For more information, see Using Buffered I/O.

Direct I/O
The operating system locks the application's buffer in memory. It then creates a memory descriptor list (MDL) that identifies the locked memory pages, and passes the MDL to the driver stack. Drivers access the locked pages through the MDL.

For more information, see Using Direct I/O.

Neither Buffered Nor Direct I/O
The operating system passes the application buffer's virtual starting address and size to the driver stack. The buffer is only accessible from drivers that execute in the application's thread context.

For more information, see Using Neither Buffered Nor Direct I/O.

For IRP_MJ_READ and IRP_MJ_WRITE requests, drivers specify the I/O method by using flags in each DEVICE_OBJECT structure. For more information, see Initializing a Device Object.

For IRP_MJ_DEVICE_CONTROL and IRP_MJ_INTERNAL_DEVICE_CONTROL requests, the I/O method is determined by the *TransferType* value that is contained in each IOCTL value. For more information, see Defining I/O Control Codes.

All drivers in a driver stack must use the same buffer access method for each request, except possibly for the highest-level driver (which can use the "neither" method, regardless of the method used by lower drivers).

在内核中访问RING3的地址是有风险的（当进程切换时，会导致访问的内存地址不在当前进程，可能会导致操作系统出现蓝屏）。访问内核地址是没有风险的。高2G的内存是共享的，切换进程访问内存不受影响。

操作系统有一个内存分配表，每个进程都有属于自己的分配表（在自己的分配表中查询地址）。

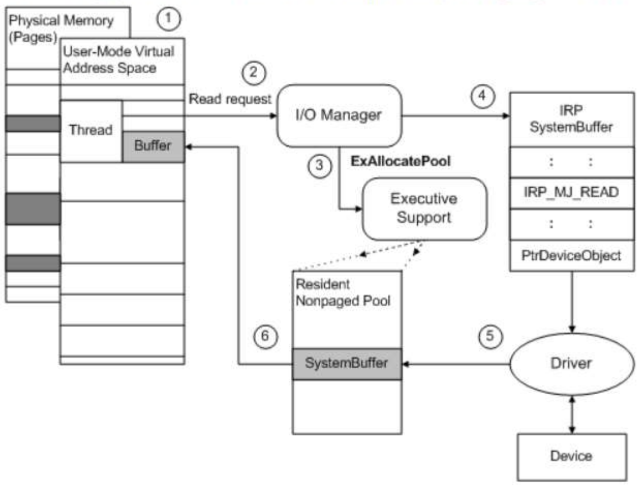
编写驱动时，在驱动中访问RING3的地址，需要在对应的回调函数中使用 "__try{} __except{}"，防止驱动出现异常导致操作系统崩溃。

ProbeForWrite 函数检查用户模式缓冲区是否实际位于地址空间的用户模式部分、是否可写以及是否正确对齐（地址不能访问，在内核中抛异常）。该函数存在一个漏洞，当参数2（Length）为0时，函数不进行检查。

缓冲区方式

需要设置通讯方式：创建设备时会产生设备对象，设备对象的Flags成员用来指定其通讯方式。
pDevObj->Flags |= DO_BUFFERED_IO; // 在DriverEntry中指定通讯方式为缓冲区方式，设置该通讯方式后，在驱动的Irp中看不到Ring3的地址。

The following figure illustrates how the I/O manager sets up an IRP_MJ_READ request for a transfer operation that uses buffered I/O.



Buffered I/O for User Buffers

The figure shows an overview of how drivers can use the **SystemBuffer** pointer in the IRP to transfer data for a read request, when a driver has ORed the device object's **Flags** with DO_BUFFERED_IO:

1. Some range of user-space virtual addresses represents the current thread's buffer, and that buffer's contents might be stored somewhere within a range of page-based physical addresses (dark shading in the previous figure).
2. The I/O manager services the current thread's read request, for which the thread passes a range of user-space virtual addresses representing a buffer.
3. The I/O manager checks the user-supplied buffer for accessibility and calls **ExAllocatePoolWithTag** to create a resident system-space buffer (**SystemBuffer**) the size of the user-supplied buffer.
4. The I/O manager provides access to the newly allocated **SystemBuffer** in the IRP it sends to the driver.

   If the figure showed a write request, the I/O manager would copy data from the user buffer into the system buffer before it sent the IRP to the driver.
5. For the read request shown in the previous figure, the driver reads data from the device into the system-space buffer. When the read request has been satisfied, the driver calls **IoCompleteRequest** with the IRP.
6. When the original thread is again active, the I/O manager copies the read-in data from the system buffer into the user buffer. It also calls **ExFreePool** to release the system buffer.

After the I/O manager has created a system-space buffer for the driver, the requesting user-mode thread can be swapped out and its physical memory can be reused by another thread, possibly by a thread belonging to another process. However, the system-space virtual address range supplied in the IRP remains valid until the driver calls **IoCompleteRequest** with the IRP.

Drivers that transfer large amounts of data at a time, in particular, drivers that do multipage transfers, should not attempt to use buffered I/O. As the system runs, nonpaged pool can become fragmented so that the I/O manager cannot allocate large, contiguous system-space buffers to send in IRPs for such a driver.

Typically, a driver uses buffered I/O for some types of IRPs, such as IRP_MJ_DEVICE_CONTROL requests, even if it also uses direct I/O. Drivers that use direct I/O typically only do so for IRP_MJ_READ and IRP_MJ_WRITE requests, and possibly driver-defined IRP_MJ_INTERNAL_DEVICE_CONTROL requests that require large data transfers.

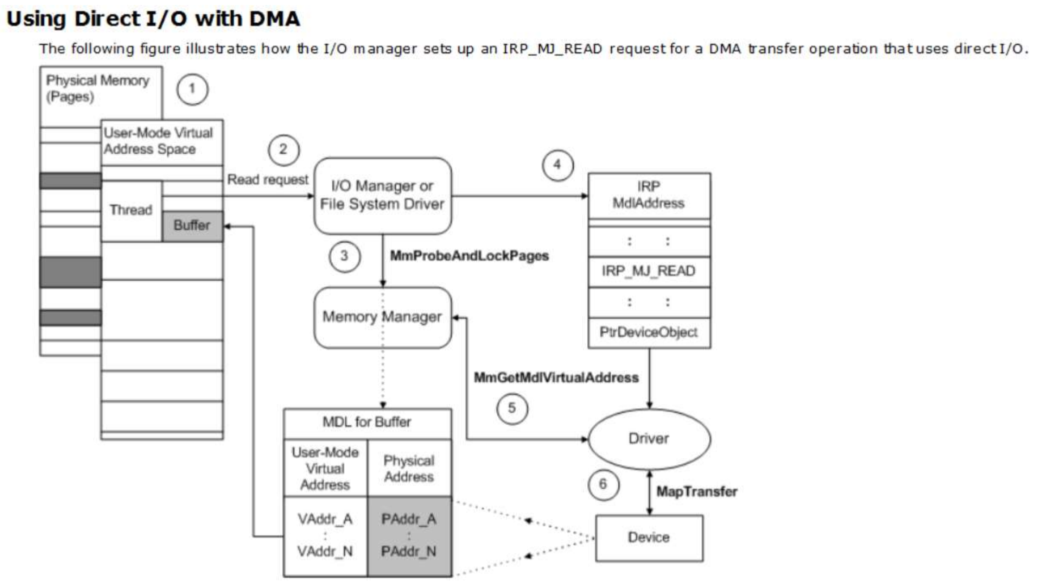
Every IRP_MJ_DEVICE_CONTROL and IRP_MJ_INTERNAL_DEVICE_CONTROL request includes an I/O control code. If the I/O control code indicates that the IRP must be supported by using buffered I/O, the I/O manager uses a single system buffer to represent the user application's input and output buffers. A driver that supports such an I/O control code must read input data (if any) from the buffer and then supply output data (if any) by overwriting the input data. For more information, see Defining I/O Control Codes.

缓冲区通讯方式不包含 "控制操作" 是因为在Ring3中使用 "DeviceIoControl" 可以传递两个缓冲区，但是在Ring0中 "Irp->AssociatedIrp.SystemBuffer" 不知道映射哪个缓冲区的数据。

> 这种方式存在数据拷贝操作，当有大量数据操作时，这种方式效率就不是很高，注重效率推荐使用直接方式。

直接方式

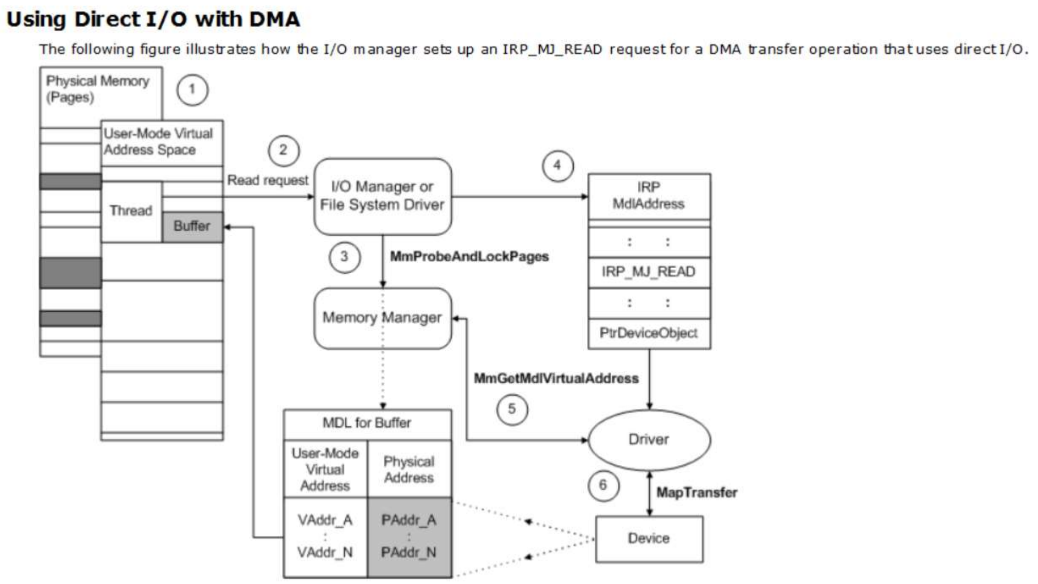
> pDevObj->Flags |= DO_DIRECT_IO; // 指定通讯方式为直接方式


Using Direct I/O with DMA

The following figure illustrates how the I/O manager sets up an IRP_MJ_READ request for a DMA transfer operation that uses direct I/O.

在第三步需要 通过内存管理器计算出缓冲区对应的物理地址。将保存缓冲区信息的 MDl 结构保存到 IRP 中。通过 "MmGetMdlVirtualAddress" 获取内核中映射该缓冲区物理地址的虚拟地址，访问内核中这个虚拟地址就相当于访问这个缓冲器对应的物理内存地址，这样就不需要拷贝数据。推荐使 "MmGetSystemAddressForMdlSafe" 来替代 "MmGetMdlVirtualAddress"。

```
// MdlAddress 保存了内存的映射信息
PVOID lpBuff   = MmGetSystemAddressForMdlSafe(Irp->MdlAddress,
NormalPagePriority);
```


Using Direct I/O with DMA

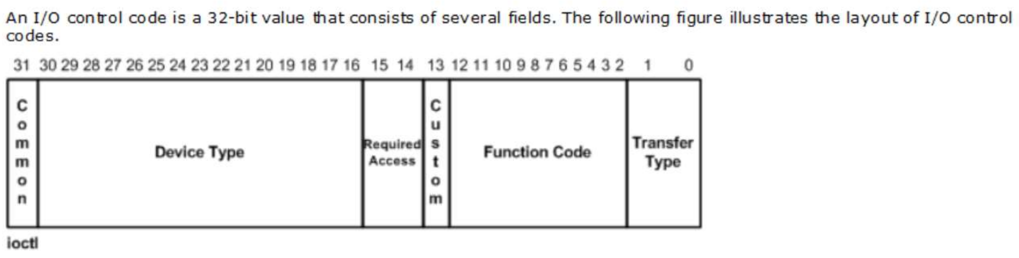
The following figure illustrates how the I/O manager sets up an IRP_MJ_READ request for a DMA transfer operation that uses direct I/O.

MmMapIoSpace：将给定的物理地址范围映射到非分页系统空间中，也就是映射出一块虚拟地址。

当在Ring3、Ring0互相传递数据时，就需要用到控制。使用 "DeviceIoControl" 函数可以传递两个缓冲区。所以在内核驱动中一般给三个回调函数就够使用（打开、关闭、控制）。

控制回调的编写

控制码定义格式：


An I/O control code is a 32-bit value that consists of several fields. The following figure illustrates the layout of I/O control codes.

缓冲区通讯方式需要写到控制码的解析中。控制码在 IRP 堆栈中。通过控制回调控制设备需要编写控制码。驱动控制码的编写，由驱动开发者定义。

```
// 使用 CTL_CODE 需要包含 WinIoCtl.h
CTL_CODE(FILE_DEVICE_UNKNOWN, (CODE_BASE+(code)), METHOD_BUFFERED,
FILE_ANY_ACCESS)

// 自定义控制码
#define MY_CTL_CODE(code) CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800 + code,
METHOD_OUT_DIRECT, FILE_ANY_ACCESS)
#define MYCTL_SHOW_MSG      MY_CTL_CODE(1)
#define MYCTL_GET_MSG        MY_CTL_CODE(2)
#define MYCTL_LOOKUP_MSG MY_CTL_CODE(3)
```