《Windows 驱动开发技术详解》读书例程-------6~8 章

第 6 章　　Windows 内核函数

本章介绍了 Windows 内核中字符串处理函数，文件读写函数。注册表读写函数。这些函数是 DDK 提供的运行时函数。他们比标准 C 语言的运行时函数功能更丰富。。普通的 C 语言运行时库是不能在内核模式下使用的，必须使用 DDK 提供的运行时函数。

6.1　内核模式下的字符串操作

和应用程序一样，驱动成天徐需要经常和字符串打交道。其中包括 ASCII 字符串，宽字符串，还有 DDK 定义的 ANSI_STRING 数据结构和 UNICODE_STRING 数据结构。

6.1.1　ASCII 字符串和宽字符串

在应用程序中，往往使用两种字符：一种是 char 型的字符串，负责记录 ANSI 字符集，它是指向一个 chr 数组的指针。每个 char 型变量的大小为一个字节。字符串是以 0 标志字符串结束。还有一种是 wchar_t 型的宽字符串，负责描述 unicode 字符集的字符串。它是指向一个 wchaar_t 数组的指针。wchar_t 字符大小为两个字节。字符串是以 0 标志字符串结束

ASCII 字符串对应宽字符串。ANSI 字符集对应 UNICODE 字符集

ANSI 字符集的构造如下。

```
char*　　str1="abc";
```

str1 指针指向的内容是 61626300.

UNOCODE 字符的构造如下

```
wchat_t　*str2=L"abc";
```

str2 指针指向的内容是 6100620063000000.

在构造字符串的时候使用一个关键字"L"...编译器会自动生成所需要的宽字符。

在驱动程序开发中，DDK 将 char 和 wchat_t 类别，替换成 CHAR 和 WCHAR 类别。
对于这两类的字符串，DDK 提供相应的字符串操作函数。例如，strcpy,sprintf,strcat,strlen 等。但 DDK 的帮助文档中，
不会查到这些函数的使用方法。微软公司不鼓励直接使用这些函数。取而代之的是使用同样功能的宏。

NTOSKRSL.EXE 导出的函数

驱动程序可以用 KdPrint 打印 ASCII 字符串和宽字符串。KdPrint 类似于 C 语言的 printf 函数。例如，打印一个 ASCII 字符串。

```
CHAR　*string="Hello";
KdPrint("%s\n",string);//注意是小写%s
```

打印一段宽字符时需要。

```
WCHAR    *string=L"Hello";
KdPrint("%S\n",string);//注意是大写%S..
```

6.1.2　ANSI_STRING 字符串与 UNICODE_STRING 字符串。

DDK 不鼓励程序员使用 C 语言的字符串。主要是因为：标准 C 的字符串处理函数容易导致缓冲区溢出等错误。

如果程序员不对字符串的长度进行检验，很容易导致这个错误。从而导致整个操作系统的崩溃。

DDK 鼓励程序员使用 DDK 自定义的字符串。这种数据格式的定义如下：

```
typedef    struct    _STRING{
USHORT    Length;
USHORT    MaximumLength;
PCHAR     Buffer;
}STRING;
typedef     STRING     ANSI_STRING;
typedef     PSTRING     PANSI_STRING;

typedef     STRING    OEM_STRING;
typedef     STRING    POEM_STRING;
```

这个数据结构对 ASCII 字符串进行了封装。

Length:字符的长度。

MaximumLength:整个字符串缓冲区的最大长度。

Buffer:缓冲区的指针。

和标准的字符串不同，STRING 字符串不是以 0 标志字符串的结束。

字符串长度依靠 Length 字段。在标准 C 中的字符串中，如果缓冲区长度是 N，那么只能容纳 N-1 个字符的字符串，这是因为要留一个字节存储 NULL。而在 STRING 字符串中，缓冲区的大小 MaximumLength, 最大的字符串长度可以是 MaximumLength, 而不是 MaximumLength-1 。

DDK 鼓励程序员使用 DDK 自定义的字符串 STRING..

与 ANSI_STRING 相对应，DDK 将宽字符串封装成 UNICODE_STRING 数据结构。

ANSI_STRING　　UNICODE_STRING

```
typedef　struct　_UNICODE_STRING {
USHORT　Length;
USHORT　MaximumLength;
PWSTR　Buffer;
}UNICODE_STRING;
```

Length:字符的长度，单位是字节。。如果是 N 个字符，那么 Length 等于 N 的 2 倍。
MaximumLength:单位也是字节。
Buffer:缓冲区的指针。

PCHAR　PWSTR.

和 ANSI_STRING 不同，UNICODE_STRING 的缓冲区是记录宽字符的缓冲区。每个元素是宽字符。和 ANSI_STRING 一样，字符串的介绍不是以 NULL 为标志；。
而是依靠字段 Length.

关于 ANSI_STRING 字符串和 UNICODE_STRING 字符串，KdPrint 同样提供了打印 log 的方法。

```
ANSI_STRING　ansiString;
KdPrint("%Z\n",&ansiString);
```

而当打印一段宽字符的时候，需要进行以下操作。

```
UNICODE_STRING　uniString;

KdPrint("%wZ\n",&uniString);
```

6.1.3　字符初始化与销毁
ANSI_STRING 字符串和 UNICODE_STRING 字符串使用前需要进行初始化。
有两种方法构造这个数据结构。

（1）方法一就是使用 DDK 提供的响应 的函数。
初始化 ANSI_STRING 字符串。

```
VOID　RltInitAnsiString(
IN OUT　PANSI_STRING　DestinationString,
```

```
IN    PCSZ      SourceString);
```

SourceString:字符串的内容。

初始化 UNICODE_STRING 字符串

```
VOID    RtlInitUnicodeString(
IN OUT    PUNICODE_STRING    Destinationstring,
IN    PCWSTR      SourceString);
```

SourceString:也是内容而已

RtlInitAnsiString 使用方法。

```
ANSI_STRING    AnsiString1;
CHAR      *      string1="hello";
RtlInitAnsiString(&AnsiString1,string1);
```

```
ANSI_STRING      AnsiString1;
CHAR    *    string="hello";
RtlInitAnsiString(&AnsiString1,string1);
```

这种办法是将 AnsiString1 中的 Buffer 指针等于 string1 指针。这种初始化的优点是操作简单，用完后不用清理内存。但带来另外一个问题，如果
修改 string1,同时会导致 AnsiString1 会发生变化。

```
ANSI_STRING    AnsiString1;
CHAR      *      string1="hello";

RtlInitAnsiString(&AnsiString1,string1);


KdPrint(("AnsiString1：%Z\n",&AnsiString1));


//改变 string1

string1[0]='H';
string[1]='E';
string[2]='L';
```

```
string[3]='L';
string[4]='O';

//改变 string1,AnsiString 同样会导致变化。
KdPrint(("AnsiString1:%Z\n",&AnsiString1));//打印 HELLO
```

(2)方法二就是程序员自己申请内存，并初始化内存，当不用字符串时，需要回收字符串占用的内存。

对于最后一步清理内存，DDK 同样给出了简化函数，分别是 RtlFreeAnsiString 和 RtlFreeUnicodeString...这两个函数内部调用了 ExFreePool 去回收内存还有设置为 0

6.1.4　字符串复制。

DDK 提供针对 ANSI_STRING 字符串和 UNICODE_STRING 字符串的复制字符串命令，分别是

ANSI_STRING 字符串复制函数。

```
VOID    RtlCopyString(
IN OUT  PSTRING     DestinationString,
IN PSTRING  SourceString  OPTIONAL);
```

UNICODE_STRING 字符串复制函数

```
VOID  RltCopyUnicodeString(
IN OUT  PUNICODE_STRING DestinationString,
IN  PUNICODE_STRING  SourceString);
```

6.1.5　字符串比较。

```
LONG    RltCompareString(
IN PSTRING  String1,
IN PSTRING  String2,
BOOLEAN  CaseInSensitive);

LONG    RtlCompareUnicodeString(
IN PUNICODE_STRING  String1,
```

IN PUNICODE_STRING String2,
IN BOOLEAN CaseInSensitive);

返回 0，则表示两个字符串相等。如果小于 0，则表示第一个字符串小于第二个字符串，如果大于 0，则表示第一个字符串大于第二个字符串。

RtlEqualString RtlEqualUnicodeString

6.1.6 字符串转化成大写

DDK 提供了对 ANSI_STRING 字符串和 UNICODE_STRING 字符串的相关字符串大小写转化 的函数

(1)ANSI_STRING 字符串转化成大写。

VOID RtlUpperString(
IN OUT PSTRING DestinationString,
IN PSTRING SourceString);

(2)UNICODE_STRING 字符串转化成大写。

NTSTATUS RtlUpcaseUnicodeString(
IN OUT PUNICODE_STRING DestinationString OPTIONAL,
IN PCUNICODE_STRING SourceString,
IN BOOLEAN AllocateDestinationString);

AllocateDestinationString:是否为目的字符串分配内存
返回值：返回转化是否成功。
目的字符串和源字符串可以是同一个字符串

DDK 虽然提供了转化成大写的函数，但却没有提供转化成小写的函数。

6.1.7 字符串与整形数字相互转换。

(1)将 UNICODE_STRING 字符串转换成整数

NTSTATUS RtlUnicodeStringToInteger(
IN PUNICODE_STRING String,
IN ULONG Base OPTIONAL,
OUT PULONG Value);

String:需要转换的字符串。
Base:转换的数的进制。
Value:需要转换的数字。
返回值：指明是否转换成功。

(2)将整数转换成 UNICODE_STRING 字符串。

NTSTATUS RtlIntegerToUnicodeString(
IN ULONG Value,
IN ULONG Base OPTIONAL,
IN OUT PUNICODE_STRING String);

返回值：指明是否转换成功。

6.1.8 ANSI_STRING 与 UNICODE_STRING 字符串的相互转换

(1)将 UNICODE_STRING 字符串转换成 ANSI_STRING 字符串

```
NTSTATUS RtlUnicodeStringToAnsiString(
IN OUT PANSI_STRING DestinationString,
IN PUNICODE_STRING SourceString,
IN BOOLEAN AllocateDestinationString);
```

AllocateDestinationString:是否需要对呗转换的字符串分配内存
返回值：指明是否转换成功
(2)将 ANSI_STRING 字符串转换成 UNICODE_STRING 字符串。

```
NTSTATUS RtlAnsiStringToUnicodeString(
IN OUT PUNICODE_STRING DestinationString,
IN PANSI_STRING SourceString,
IN BOOLEAN AllocateDestinationString);
```

参数大概一致

转换后的需要销毁，而之前的通过 RtlInitAnsiString 不需要销毁。

6.2 内核模式下的文件操作

在驱动程序开发中。经常会对文件进行操作。与 Win32API 不同，DDK 提供了另外一套对文件的操作函数。

6.2.1 文件的创建。

对文件的创建或者打开都是通过内核函数 ZwCreateFile 实现的。和 Windows API 类似，这个内核函数返回一个文件句柄，
文件的所有操作都是依靠这个句柄进行操作的。
在文件操作完毕后，需要关闭这个句柄。

```
NTSTATUS ZwCreateFile(
OUT PHANDLE FileHandle,
IN ACCESS_MASK DesiredAccess,
IN POBJECT_ATTRIBUTES ObjectAttributes,
OUT PIO_STATUS_BLOCK IoStatusBlock,
IN PLARGE_INTEGER AllocationSize OPTIONAL,
IN ULONG FileAttributes,
IN ULONG ShareAccess,
IN ULONG CreateDisposition,
IN ULONG CreateOptions,
```

IN PVOID EaBuffer OPTIONAL,
IN ULONG EaLength);

DesiredAccess: GENERIC_READ,,GENERIC_WRITE.
ObjectAttributes:结构 OBJECT_ATTRIBUTES 的地址。该结构包含要打开的文件名。
IoStatusBlock:指向 IO_STATUS_BLOCK 结构。该结构接收 ZwCreateFile 操作的结果状态。
AllocationSize:是一个指针。指向一个 64 位整数，该数指定文件初始分配时的大小。该参数仅仅关系到创建或重写文件操作。如果忽略它，那么文件长度将从 0 开始，并随着写入而增长。
FileAttributes:0 或 FILE_ATTRIBUTE_NORMAL,，指定新创建文件的属性。
ShareAccess:FILE_SHARE_READ 或 0.指定文件的共享方式。只有 FILE_SHARE_READ 或 0.
CreateDispostion:FILE_OPEN 或 FILE_OVERWRITE_IF。表明当指定文件存在或不存在时应如何处理
CreateOptions:FILE_SYNCHRONOUS_IO_NONALERT,指定控制打开操作和句柄使用的附加标志位

CreateDispositon: 打开文件 FILE_OPEN ....创建文件： FILE_OVERWRITE_IF.
此时，无论文件是否存在，都会创建新文件

DDK 提供了对 OBJECT_ATTRIBUTES 结构初始化的宏 InitializeObjectAttributes.

VOID InitializeObjectAttributes(
OUT POBJECT_ATTRIBUTESW InitializedAttributes,
IN PUNICODE_STRING ObjectName,
IN ULONG Attributes,
IN HANDLE RootDirectory,
IN PSECURITY_DESCRIPTOR SecurityDescriptor);

Attributes:一般设为 OBJ_CASE_INSENSITIVE,对大小写敏感
RootDirectory:一般设为 NULL
SecurityDescriptor:一般设为 NULL

另外，文件名必须是符号链接或者是设备名。"\??\c:""\??\c:\1.log".

其中，"\??\c:"是符号链接，内核会将它转换成设备名"\Device\HarddiskVolume1" .

(1)创建文件(2)打开文件

6.2.2　文件的打开。。。
除了使用 ZwCreateFile 函数可以打开文件，DDK 还提供了一个内核函数 ZwOpenFile..
ZwOpenFile 内核函数的参数比 ZwCreateFile 的参数简化。方便程序打开文件。

```
NTSTATUS    ZwOpenFile(
OUT    PHANDLE    FileHandle,
IN     ACCESS_MASK    DesiredAccess,
IN     POBJECT_ATTRIBUTES    ObjectAttributes,
OUT    PIO_STATUS_BLOCK    IoStatusBlock,
IN    ULONG    ShareAccess,
IN    ULONG        OpenOptions);
```

FileHandle:返回打开的文件句柄
DesiredAccess:打开的权限，一般设为 GENERIC_ALL.
ObjectAttributes:objectAttributes 结构
IoStatusBolock:指向一个结构体的指针。该结构体指明打开文件的状态
ShareAccess:FILE_SHARE_READ,FILE_SHARE_WRITE.
OpenOptions:一般设为 FILE_SYNCHRONOUS_IO_NONALERT..
返回值：指明文件是否被成打开

6.2.3　获取或修改文件属性
获取和修改文件属性。包括获取文件大小，获取或修改文件指针位置。获取或修改文件名。
获取或修改文属性。
获取或修改文件创建，修改日期等。DDK 提供了内核函数 ZwSetInformationFile 和
ZwQueryInformationFile 函数来进行获取和修改文件属性

```
NTSTATUS    ZwSetInformationFile(
IN    HANDLE    FileHandle,
OUT    PIO_STATUS_BLOCK    IoStatusBlock,
IN    PVOID    FileInformation,
IN    ULONG    Length,
IN    FILE_INFORMATION_CLASS    FileInformationClass);
```

IoStatusBlock:返回设置的状态

```
NTSTATUS    ZwQueryInformationFile(
IN    HANDLE    FileHandle,
OUT    PIO_STATUS_BLOCK    IoStatusBlock,
```

OUT PVOID FileInformation,
IN ULONG Length,
IN FILE_INFORMATION_CLASS FileInformationClass);

(1)当 FileInforamtionClass 是 FileStandardInformation 时，输入和输出的数据是 FILE_STANDARD_INFORMATION 结构体。描述文件的基本信息

typedef struct FILE_STANDARD_INFORMATION{
LARGE_INTEGER AllocationSize;
LARGE_INTEGER EndOfFile;
ULONG NumberOfLinks;
BOOLEAN DeletePending;
BOOLEAN Directory;
}FIEL_STANDARD_INFORMATION,PFILE_STANDARD_INFORMATION;

(2) 当 FileInformationClass 是 FileBasicInformation 时，输入和输出的数据是 FILE_BASIC_INFOMRMATION 结构体。描述文件的基本信息
typedef struct FILE_BASIC_INFORMATION{
LARGE_INTEGER CreationTime;
LARGE_INTEGER LastAccessTime;
LARGE_INTEGER LastWriteTIme;
LARGE_INTEGER ChangeTime;
ULONG FileAttributes;
}FILE_BASIC_INFORMATION,*PFILE_BASIC_INFORMATION;

其中时间参数是从一个 LARGE_INTEGER 的整数，该整数代表从 1601 年经过多少个 100ns（纳秒）。FileAttributes 描述文件属性。
FILE_ATTRIBUTE_NORMAL,FILE_ATTRIBUTE_DIRECTORY,FILE_ATTRIBUTE_READONLY,FILE_ATTRIBUTE_HIDDEN,FILE_ATTRIBUTE_SYSTEM.

(3) 当 FileInformationClass 是 FileNameInformation 时，输入和输出的数据是 FILE_NAME_INFORMATION 结构体，描述文件名信息。

typedef struct _FILE_NAME_INFORMATION{
ULONG FileNameLength;
WCHAR FileName[1];
}FILE_NAME_INFORMATION,*PFILE_NAME_INFORMATION;

(4) 当 FileInformationClass 是 FilePositionInformation 时，输入和输出的数据是

FILE_POSITION_INFORMATION 结构体。描述文件名信息

```
typedef     struct FILE_POSITION_INFORMATION{
LARGE_INTEGER     CurrentByteOffset;
}FILE_POSITION_INFORMATION,*PFILE_POSITION_INFORMATION;
```

6.2.4　文件的写操作。
DDK 提供了文件的写操作的内核函数

```
NTSTATUS    ZwWriteFile(
IN   HANDLE     FileHandle,
IN   HANDLE     Event    OPTIONAL,
IN   PIO_APC_ROUTINE ApcRoutine     OPTIONAL,
IN   PVOID    ApcContext    OPTIONAL,
OUT    PIO_STATUS_BLOCK    IoStatusBlock,
IN   PVOID    Buffer,
IN   ULONG    Length,
IN   PLARGE_INTEGER     ByteOffset    OPTIONAL,
IN   PULONG    Key    OPTIONAL
);
```

6.2.5　文件的读操作
DDK 提供了文件读操作的内核函数。

```
NTSTATUS    ZwReadFile(
IN HANDLE    FileHandle,
IN HANDLE    Event    OPOTIONAL,
IN PIO_APC_ROUTINE    ApcRoutine    OPTIONAL,
IN PVOID    ApcContext    OPTIONAL,
OUT    PIO_STATUS_BLOCK     IoStatusBlock,
IN   PVOID    Buffer,
IN   ULONG    Length,
IN   PLARGE_INTEGER    ByteOffset    OPTIONAL,
IN   PULONG    Key    OPTIONAL
);
```

下面的代码演示了如何在驱动程序中读文件。如果是读取整个文件，需要知道文件的大小。
该功能可以通过内核函数 ZwQueryInformationFile 来实现

6.3　内核模式下的注册表操作。
在驱动程序开发中，经常会用到注册表的操作
与 Win32API 不同，DDK 提供另外一套对注册表操作的相关函数

6.3.1 创建关闭注册表

和文件操作类似，对注册表操作首先要获取一个注册表句柄。。对注册表的操作都需要根据这个句柄进行操作
可以通过 ZwCreateKey 函数获得打开的注册表句柄。。。这个函数打开注册表后，并返回一个操作句柄。

```
NTSTATUS  ZwCreateKey(
OUT  PHANDLE  KeyHandle,
IN   ACCESS_MASK  DesiredAccess,
IN  POBJECT_ATTRIBUTES  ObjectAttributes,
IN  ULONG  TitleIndex,
IN  PUINCODE_STRING  Class  OPTIONAL,
IN  ULONG  CreateOptions,
OUT PULONG  Disposition  OPTIONAL
);
```

OUT Disposition:返回是创建成功。还是打开成功
返回值：返回是否创建成功。

```
REG_CREATED_NEW_KEY    REG_OPENED_EXISTING_KEY
```

6.3.2 打开注册表

ZwCreateKey 函数既可以创建注册表项，也可以打开注册表项。为了简化打开注册表项，DDK 提供了内核函数 ZwOpenKey,以简化打开操作。如果 ZwOpenKey 指定的项不存在，不会创建这个项目。而是返回一个错误状态

```
NTSTATUS  ZwOpenKey(
OUT  PHANDLE  KeyHandle,
IN  ACCESS_MASK    DesiredAccess,
IN  POBJECT_ATTRIBUTES    ObjectAttributes);
```

```
NTSTATUS  ntStatus=ZwOpenKey(&hRegister,KEY_ALL_ACCESS,&objectAttributes);
```

6.3.3 添加，修改注册表键值。。。
打开注册表的句柄后，就可以对该项进行设置和修改了。注册表是以二元形式存储的即“键名”和“键值”存在的。通过键名设置键值

REG_BINARY REG_SZ ,REG_EXPAND_SZ,REG_MULTI_SZ,REG_DWORD,REG_QWORD ..

```
NTSTATUS ZwSetValueKey(
IN HANDLE KeyHandle,
IN PUNICODE_STRING ValueName,
IN ULONG TitleIndex OPTIONAL,
IN ULONG Type,
IN PVOID Data,
IN ULONG DataSize);
```

6.4 查询注册表
驱动程序中有时需要对注册表的项进行查询。从而获取注册表的键值。。DDK 提供的 ZwQueryValueKey 函数可以完成这个任务。

```
NTSTATUS ZwQueryValueKey(
IN HANDLE Key Handle,
IN PUNICODE_STRING ValueName,
IN KEY_VALUE_INFORMATIOIN_CLASS KeyValueInformationClass,
OUT PVOID KeyValueInformation,
IN ULONG Length,
OUT PULONG ResultLength);
```

KeyValueBasicInformation,KeyValueFullInformation,或者 KeyValuePartialInformation 中的一种。这分别代表查询基本信息，查询全部信息和查询部分信息。每种查询类型会有对应的一种数据结构获得查询结果。
一般情况下，选择 KeyValuePartilInformation 就可以查询键值的数据了。他对应的查询数据结构是 KEY_VALUE_PARTIAL_INFORMATION 的数据结构。

```
typedef struct _KEY_VALUE_PARTIAL_INFORMAITON{
ULONG TitleIndex;
ULONG Type;
ULONG DataLength;
UCHAR Data[1];
}KEY_VALUE_PARTIAL_INFORMATION,*PKEY_VALUE_PARTIAL_INFORMATION;
```

6.3.5 枚举子项
在注册表的操作中，还经常有另外两种操作，分别是枚举子项和枚举子键。枚举子项就是事先不知道该项中有多少个
子项目，用某个函数将子项一一列举出来。而枚举子键是事先不知道该项中有多少个子键，用某个函数一一将子键列举出来

DDK 提供了列觉子项的函数，ZwQueryKey 函数和 ZwEnumerateKey..

```
NTSTATUS    ZwQueryKey(
IN HANDLE    KeyHandle,
IN KEY_INFORMATIONc_CLASS    KeyInformationClass,
OUT PVOID    KeyInformation,
IN ULONG    Length,
OUT PULONG    ResultLength);
```

KeyInformation:KeyFullInfomation.
KeyInformation:查询的数据指针。

```
NTSTATUS    ZwEnumerateKey(
IN HANDLE    KeyHandle,
IN ULONG    Index,
IN KEY_INFORMATION_CLASS KeyInformationClass,
OUT    PVOID    KeyInformation,
IN ULONG Length,
OUT PULONG    ResultLength);
```

KeyHandle:注册表项句柄。
Index:很少用到，一般为 0.
KeyInformationClass:该子项的信息。
Length:子项信息的长度。
ResultLength:返回子键信息的长度。
返回值：指明枚举是否成功

ZwQueryKey 的作用主要是获得某注册表项究竟有多少个子项。而 ZwEnumerateKey 的作用主要是针对第几个子项
获取该子项的具体信息。

```
ULONG ulSize;

ZwQueryKey(hRegister,KeyFullInformation,NULL,0,&ulSize);

PKEY_FULL_INFORMATION pfi=(PKEY_FULL_INFORMATION)ExAllocatePool(PagedPool,ulSize);

ZwQueryKey(hRegister,KeyFullInformation,pfi,ulSize,&ulSize);

for(ULONG    i=0;i<pfi->SubKeys;i++)
{
ZwEnumerateKey(hRegister,i,KeyBasicInformation,NULL,0,&ulSize);

PKEY_BASIC_INFORMATION pbi=(PKEY_BASIC_INFORMATION)ExAllocatePool(PagedPool,ulSize);

ZwEnumerateKey(hRegister,i,KeyBasicInformation,pbi,ulSize,&ulSize);

UNICODE_STRING uniKeyName;
uniKeyName.Length=
uniKeyName.MaximumLength=
(USHORT)pbi->NameLength;
uniKeyName.Buffer=pbi->Name;
KdPrint(("The %d sub item name:%wZ\n",i,&uniKeyName));
ExFreePool(pbi);
}
ExFreePool(pfi);

ZwClose(hRegister);
```

6.3.6　枚举子键

需要通过两个函数配合实现。
ZwQueryKey 上节已经介绍过。
ZwEnumerateValueKey　　……

```
ZwEnumerateKey-->ZwEnumerateValueKey...
PKEY_BASIC_INFORMATION-->PKEY_VALUE_BASIC_INFORMATION...
....
```

跟枚举子项就上面这点差别。

6.3.7 删除子项

DDK 同样提供了删除子项的内核函数，ZwDeleteKey……

```
NTSTATUS    ZwDeleteKey(
IN HANDLE    KeyHandle);
```

返回值：指示是否删除成功

6.3.8 其它。

为了简化注册表操作，DDK 还提供了一系列以 Rtl 开头的运行时函数
这些函数吧前面介绍的函数进行了封装。往往一条函数就能实现前面介绍的若干条函数的功能。

RtlCreateRegistryKey
RtlCheckRegistryKey
RtlWriteRegistryValue
RtlDeleteRegistryValue

6.4 小结

第 7 章 派遣函数

派遣函数是 Windows 驱动程序中的重要概念。驱动程序的主要功能是负责处理 I/O 请求，其中大部分 I/O 请求是在派遣函数中处理的
用户模式下所有驱动程序的 I/O 请求，全部由操作系统转化为一个叫做 IRP 的数据结构。。。。。
不同的 IRP 数据会被“派遣”到不同的派遣函数中，这也是派遣函数名字的由来。

7.1 IRP 与派遣函数

IRP 的处理机制类似 Windows 应用程序中的“消息处理”机制，驱动程序接收到不同的类型的 IRP 后，会进入不同的派遣函数，在派遣函数中 IRP 得到处理。

7.1.1 IRP

IRP：输入输出请求包。
上层应用程序与底层驱动程序通信时，应用程序会发出 I/O 请求，操作系统将 I/O 请求转化

为相应的 IRP 数据，不同类型 IRP 会根据类型传递到不同的派遣函数内。

IRP 两个基本的属性。
(1)MajorFunction
(2)MinorFunction.
记录 IRP 的主类型和子类型

HelloDDK 和 HelloWDM 驱动程序，都是在入口函数 DriverEntry 里注册 IRP 的派遣函数函数。

一般来说，NT 式驱动程序和 WDM 驱动程序都是在 DriverEntry 函数中注册派遣函数的。

函数指针数组 MajorFunction.
函数指针数组是个数组，每个元素都记录着一个函数的地址。通过设置这个数组，可以将 IRP 的类型和派遣函数关联起来。

系统默认没有设置的 IRP 类型与_IopInvalidDeviceRequest 函数关联

7.1.2 IRP 类型

文件 I/O 的相关函数
内核中的文件 I/O 处理函数。这两者产生相同的 IRP。
这些都会创建相应的 IRP，并将 IRP 传送到相应的驱动程序的派遣函数中

还有些 IRP 是由系统的某个组件创建的，比如 IRP_MJ_SHUTDOWN 是在 Windows 的即插即用组件在即将关闭系统的时候发出的

IRP_MJ_CREATE 创建设备，由 CreateFile,ZwCreateFile 产生
IRP_MJ_CLEANUP 清楚工作，CloseHandle 会产生此 IRP
IRP_MJ_CLOSE 关闭设备，CloseHande 会产生此 IRP

IRP_MJ_DEVICE_CONTROL DeviceIoControl 函数会产生此 IRP.…….
IRP_MJ_PNP 即插即用消息。NT 驱动不会支持此种 IRP.只有 WDM 驱动才支持此种 IRP.
IRP_MJ_POWER 在操作系统处理电源消息时，产生此 IRP
IRP_MJ_QUERY_INFORMATION,IRP_MJ_READ,IRP_MJ_SET_INFORMATION,IRP_MJ_SHUTDOWN,IRP_MJ_SYSTEM_CONTROL,IRP_MJ_WRITE.

7.1.3 对派遣函数的简单处理

处理这些 IRP 最简单的方法就是在相应的派遣函数中，将 IRP 的状态设置为成功，然后结束 IRP 的请求，并让派遣函数返回成功。

结束 IRP 的请求使用函数 IoCompleteRequest.

下面的代码演示了一种最简单的处理 IRP 请求的派遣函数

```
NTSTATUS    HelloDDKDispatchRoutin(IN PDEVICE_OBJECT    pDevObj,IN PIRP    pIrp)
{
KdPrint(("Enter    HelloDDKDispatchRoutin\n"));

NTSTATUS    status=STATUS_SUCCESS;

pIrp->IoStatus.Status=status;

pIrp->IoStatus.Information=0;
//设置 IRP 操作了多少字节。

IoCompleteRuquest(pIrp,IO_NO_INCREMENT);
KdPrint(("Leave    HelloDDKDispatchRoutin.\n"));
return status;
}
```

所得的错误代码会和 IRP 设置的状态相一致。

```
VOID    IoCompleteRuquest(
IN PIRP    Irp,
IN CCHAR PriorityBoost);
```

PriorityBoost:代表线程恢复时的优先级别

Win32API 的内部操作过程（ReadFile 为例）
ReadFile 调用 ntdll 中的 NtReadFile。其中 ReadFile 函数是 Win32API,而 NtReadFile 函数是 NativeAPI
ntdll 中的 NtReadFile 进入到内核模式，并调用系统服务中的 NtReadFile 函数。
系统服务函数 NtReadFile 创建 IRP_MJ_WRITE 类型的 IRP。然后将它这个 IRP 发送到某个驱动程序的派遣函数中
在派遣函数中一般会将 IRP 请求结束，结束 IRP 是通过 IoCompleteRequest 函数
在 IoCompleteRequest 函数内部会设置刚才等待的时间，“睡眠”的线程被恢复运行

对某些特殊情况，需要将阻塞的线程以“优先”的身份恢复运行。如键盘，鼠标等输入设备，他们需要更快的反应

IO_NO_INCREMENT
IO_CD_ROM_INCREMENT
IO_DISK_INCREMENT
IO_KEYBOARD_INCREMENT
IO_MOUSE_INCREMENT
…….

本节只讨论了派遣函数中最简单的处理 IRP 的方法

7.1.4　通过设备链接打开设备

7.1.5　编写一个更通用的派遣函数
IO_STACK_LOCATION.即 I/O 堆栈。这个数据结构与 IRP 紧密相连。

IRP 会被操作系统发送到设备栈的顶层，如果顶层的设备对象的派遣函数结束了 IRP 的请求，则这次 I/O 请求结束。
如果没有将 IRP 的请求结束，那么操作系统将 IRP 转发到设备栈的下一层设备处理。如果这个设备的派遣函数依然不能结束 IRP 请求，则会继续向下层设备转发

因此，一个 IRP 可能会被转发多次。为了记录 IRP 在每层设备中做的操作，IRP 会有一个 IO_STACK_LOCATION 数组。
数组的元素数应该大于 IRP 穿越过的设备数。每个 IO_STACK_LOCATION 元素记录着对应设备中做的操作
对于本层设备对应的 IO_STACK_LOCATION，可以通过 IoGetCurrentIrpStackLocation 函数得到

```
PIO_STACK_LOCATION stack=IoGetCurrentIrpStackLocation(pIrp);

pDriverObject->MajorFuction[IRP_MJ_CREATE]=HelloDDKDispatchRoutin;
…….
```

然后在派遣函数中分辨出 IRP 的类型，并打出相应的 log 信息

7.1.6　跟踪 IRP 的利器 IRPTrace
最后我们还能查看 IRP 请求是如何被结束的，本例中 IRP 是在 HelloDDK 驱动程序中被结束的。
位置是 HelloDDK.sys!+40EEh.
IRP 请求是在 HelloDDK.sys!+40EEh 的位置被结束的。

7.2 缓冲区方式读写操作

7.2.1 缓冲区设备

7.2.2 缓冲区设备读写

7.2.3 缓冲区设备模拟文件读写

为了实现这个目的，需要编写三个 IRP 的派遣函数。他们分别对应着写操作，读操作，获取文件长度操作。

(1)写操作

```
UCHAR buffer[10];
memset(buffer,0xBB,10);
ULONG ulRead;
ULONG ulWrite;
BOOL bRet;

bRet=WriteFile(hDevice,buffer,10,&ulWrite,NULL);
if(bRet)
{
printf("Write %d bytes\n",ulWrite);
}
```

WriteFile 内部会创建 IRP_MJ_WRITE 类型的 IRP，操作系统会将这个 IRP 传递给驱动程序。IRP_MJ_WRITE 的派遣函数需要将传送进来的数据保存起来。以便读取该设备的时候读取。。在本例中，这个数据存储在一个缓冲区中，缓冲区的地址记录在设备扩展中。
在设备启动的时候，驱动程序负责分配这个缓冲区，在设备被卸载的时候，驱动程序回收该缓冲区。。

(2)读操作

在应用程序中，通过 ReadFile 从设备读取数据

```
bRet=ReadFile(hDevice,buffer,10,&ulRead,NULL);
if(bRet)
{
printf("Read %d bytes:",ulRead);

for(int i=0;i<(int)ulRead;i++)
{
printf("%02X ",buffer[i]);
}
printf("\n");
}
```

IRP_MJ_READ 的派遣函数的主要任务是把记录的数据复制到 AssociatedIrp.SystemBuffer 中。

缓冲区是内核下的缓冲区。也就是说 AssociatedIrp.SystemBuffer...

(3)读取文件长度
读取文件长度靠 GetFileSize Win32 获得。GetFileSize 内部会创建 IRP_MJ_IQUERY_INFORMATION 类型的 IRP。
这个 IRP 请求的作用是向设备查询一些信息设备一些信息。这包括查询文件长度，设备创建时间。设备属性等。
在本例中，IRP_MJ_QUERY_INFORMATION 派遣函数的主要任务是告诉应用程序这个设备的长度。
FILE_INFORMATION_CLASS.
查询用 WIN32API. GetFileSize.

```
typedef struct _FILE_STATNDARD_INFORMATION{
LARGE_INTEGER AllocationSize;
LARGE_INTEGER EndOfFile;
ULONG NumberOfLinks;
BOOLEAN DeletePending;
BOOLEAN Directory;
}FILE_STANDARD_INFORMATION,*PFILE_STANDARD_INFORMATION;
```

其中，EndOfFile 子域指明设备长度，修改这个子域会在 GetFileSize 的返回值中得到体现，IRP_MJ_QUERY_INFORMATIN 派遣函数。

读取的都是 stack 里的东西。
都是从 IRP 里获取东西的。

```
(PFILE_STANDARD_INFORMATION)pIrp->AssociatedIrp.SystemBuffer;
```

SystemBuffer 到处都是代表缓冲区。

7.3 直接方式读写操作

本节将介绍对设备的直接方式读写。

7.3.1 直接读取设备

这种方式需要创建完设备对象后，在设置设备属性的时候，设置为 DO_DIRECT_IO,而不是设置 DO_BUFFERED_IO 属性

pDevObj->Flags|=DO_DIRECT_IO;

在应用程序进行读写的时候，例如，用 WriteFIle 写设备时，会要求用户提供一段缓冲区，这段缓冲区里面是要写入设备的数据。操作系统在创建 IRP_MJ_WRITE 的时候，将需要写入的数据复制到IRP数据结构中的AssociatedIrp.SystemBuffer中。对于读设备操作也是类似的。

和缓冲区方式读写设备不同，直接方式读写设备，操作系统会将用户模式下的缓冲区锁住。然后操作系统将这段缓冲区在内核地址再次映射一遍
这样，用户模式的缓冲区和内核模式的缓冲区指向的是同一区域的物理内存物理内存。无论操作系统如何切换进程，内核模式地址都保持不变。

操作系统先将用户模式的地址锁定后，操作系统用内存描述符表(MDL 数据结构）记录这段内存。

用户模式的这段缓冲区在虚拟内存上是连续的，但是在物理内存上可能是离散的。
MDL 记录这段虚拟内存，这段虚拟内存的大小存储在 mdl->ByteCount 里，这段虚拟内存的第一个页地址是 mdl->StartVa,这段虚拟内存的首地址对于第一个页地址的偏移量是 mdl->ByteOffset.
因此，这段虚拟内存的首地址应该是 mdl->StartVa,+mdl->ByteOffset.

DDK 提供了几个宏方便程序员得到这几个数值

```
#define MmGetMdlByteCount(Mdl)   ((Mdl)->ByteCount)
#define MmGetMdlByteOffset(Mdl) ((Mdl)->ByteOffset)
#define MmGetMdlVirtualAddress(Mdl)
((PVOID)((PCHAR)((Mdl)->StartVa)+(Mdl)->ByteOffset))
```

7.3.2 直接读取设备的读写

stack->Parameters.Read.Length 希望读取字节数。
pIrp->IoStatus.Information 实际读取的字节数。

通过 IRP 的 pIrp->MdlAddress 得到 MDL 数据结构。这个结构描述了被锁定的缓冲区内存。
通过DDK的三个宏MmGetMdlByteCount,MmGetMdlVirtualAddress,MmGetMdlByteOffset 可以得到锁定缓冲区的长度，虚拟内存地址，偏移量

如果派遣函数的返回值是 STATUS_SUCCESS,则 ReadFile 返回 TRUE,表明读操作成功。

```
memset(kernel_address,0xaa,ulReadLength);
```

pIrp->IoStatus.Information=0;

映射以后，驱动程序读写 0XF7D59F70 就相当于读写 0X0012FF70 地址了

7.4 其他方式读写操作

7.4.1 其他方式设备

对 pDevObj->Flags 既不设置 DO_BUFFERED_IO,也不设置 DO_DIRECT_IO，此时采用的读写方式就是其他读写方式

在使用其他方式读写设备时,派遣函数直接读写应用程序提供的缓冲区地址。在驱动程序中，直接操作应用程序的缓冲区地址是很危险的。
只有驱动程序与应用程序运行在相同线程上下文的情况下，才能使用这种方式

用其他方式读写时，ReadFile 或者 WriteFile 提供的缓冲区内存地址，可以再派遣函数中通过 IRP->UserBuffer 字段得到。读取的字节数可以从 IO 堆栈中的
stack->Parameters.Read.Length 字段中得到。

驱动程序使用用户地址前，需要探测这段内存是否可读或可写。探测可读或者可写，应用使用 ProbeForWrite 函数和 try 块。

7.5 IO 设备控制操作

应用程序还可以通过另外一个 Win 32 API DeviceIoControl 操作设备。
DeviceIoControl 内部会使操作系统创建一个 IRP_MJ_DEVICE_CONTROL 类型的 IRP。然后系统会将这个 IRP 转发到派遣函数中。
程序员可以用 DeviceIoControl 定义除读写之外的其他操作，它可以让应用程序和驱动程序进行通信。
例如，要对一个设备进行初始化操作，程序员自定义一种 I/O 控制码，然后用 DeviceIoControl 将这个控制码和请求一起传递给驱动程序。
在派遣函数中，分别对不同的 I/O 控制码进行处理 I/O 控制码 I/O 控制码进行处理。

7.5.1 DeviceIoControl 与驱动交互

除了 ReadFile 和 WriteFile 两个 WIN32 API 可以与驱动程序进行通信。还有一个 WIN32 APIDeviceIoControl 可以与驱动程序相互通信。

BOOL DeviceIoControl(
HANDLE hDevice,
DWROD dwIoControlCode,

LPVOID lpInBuffer,
DWROD nInBufferSize,
LPVOID lpOutBuffer,
DWORD nOutBufferSize,
LPDWORD lpBytesReturned,
LPOVERLAPPED lpOverlapped);

其中，lpBytesReturned 对应派遣函数中的 IRP 结构中的 pIrp->IoStatus.Information.
控制码也称 IOCTL 值，是一个 32 位的无符号整形。IOCTL 需要符合 DDK 的规定。

DDK 提供了一个宏 CTO_CODE.
CTL_CODE(DeviceType,Function,Method,Access)

DeviceType:设备对象的类型。这个类型应和创建设备（IoCreateDevice)时的类型想匹配。一般是形如 FILE_DEVICE_XX 的宏。

7.5.2 缓存内存模式 IOCTL
缓存内存模式：应该指定 Method 参数为 METHOD_BUFFERED.

在 Win32 API DeviceIoControl 的内部，用户提供的输入缓冲区的内容被复制到 IRP 中的 pIrp->AssociatedIrp.SystemBuffer 内存地址。复制的字节数由 DeviceIoControl 指定的输入字节数。
派遣函数可以读取 pIrp->AssociatedIrp.SystemBuffer 的内存地址，从而获得应用程序提供的输入缓冲区数据。另外，派遣函数还可以写入 pIrp->AssociatedIrp.SystemBuffer 的内存地址，这被当做设备输出的数据。操作系统会将
这个地址的数据再次复制到 DeviceIoControl 提供的输出缓冲区.复制的字节数由 pIrp->IoStatus.Information 指定。
DeviceIoControl 也可以通过它的第七个参数得到这个操作字节数。。

IogetCurrentIrpStackLocation.
stack->Parameters.DeviceIoControl.InputBufferLength.输入缓冲区大小
stack->Parameters.DeviceIoControl.OutputBufferLength 得到输出缓冲区大小。
最后，stack->Parameters.DeviceIoControl.IoControlCode 得到 IOCTL。
在派遣函数中通过 C 语言的 switch 语句分别处理不同的 IOCTL。。

首先用 CTL_CODE 宏定义 IOCTL 码。
#define IOCTL_TEST1
CTL_CODE(FILE_DEVICE_UNKOWN,0X800,METHOD_BUFFERED,FILE_ANY_ACCESS)

7.5.3 直接内存模式 IOCTL

指定 Method 参数为 METHOD_OUT_DIRECT 或者 METHOD_IN_DIRECT.。
输入缓冲区的方式和缓存内存模式 IOCTL 的处理是一样的。
输出缓冲区的方式和缓冲内存模式 IOCTL 的处理是不一样的。

pIrp->MdlAddress
MmGetSystemAddressForMdlSafe 将这段内存映射到内核模式下的内存地址。

METHOD_IN_DIRECT 和 METHOD_OUT_DIRECT 的差别。其细微差别仅仅体现在打开设备的权限上。

```
#define IOCTL_TEST2
CTL_CODE(FILE_DEVICE_UNKOWN,0x801,METHOD_IN_DIRECT,FILE_ANY_ACCESS)
```

7.5.4 其他内存模式 IOCTL

METHOD_NEITHER.

7.6 小结

本章重点介绍了驱动程序中的处理IRP请求的派遣函数。所有对设备的操作最终将转化为IRP请求，这些 IRP 请求会被传送到派遣函数处理。本章主要介绍了 IRP_MJ_READ, IRP_MJ_WRITE,IRP_MJ_CONTROL 的派遣函数。这些 IRP 请求分别有缓冲区方式，直接方式和其他方式的操作。其中，缓冲区方式和直接方式是在驱动程序开发中经常用到的。

第 2 篇

进阶篇

第 8 章 驱动程序的同步处理

Windows 是一个多任务的操作系统，每个任务对应一个运行的进程

如果没有同步机制的控制，所有的线程会任意运行。然而，多个线程可能会要求操作同一个资源，这时就需要同步处理
如果驱动程序没有很好的处理同步问题，操作系统的性能就会下降，甚至会出现死锁等现象

8.1 基本概念

8.1.1 问题的引出

在支持多线程的操作系统下，有些函数会出现不可重入的现象。
所谓“可重入”，是指函数的执行结果不和执行顺序有关。反之，如果执行结果和执行顺序有关，则称这个函数是“不可重入的”的。
不可重入的函数会对多线程操作系统下的程序带来错误，不可重入的根本原因是由于各个线程之间的切换导致的。

```
int number=0;
void Foo()
{
number++;
//做一些事情。
number--;
}
```

然而，在多线程的环境下，却不能保证 number 数值前后不变
导致汇编指令交错在一起，从而导致函数的不可冲入性

为了结果函数的不可重入性，使之变成可重入的函数，需要对函数进行同步控制。

8.1.2 同步与异步

各个线程的关系是异步的，就是每个线程的运行不受其他线程的影响

8.2 中断请求级

8.2.1 中断请求（IRQ)与可编程中断控制器（PIC）

中断请求（IRQ)一般有两种，一种是外部中断，也就是硬件产生的中断，另一种由软件指令 int n 产生的中断

在传统 PC 中，一般可以接收 16 个中断信号，每个中断信号对应一个中断号
外部中断分为不可屏蔽(NMI)和可屏蔽中断。分别由 CPU 的两个引脚 NUI 和 INTR 莱接收

可屏蔽中断是可编程中断控制器(PIC)8259A 芯片向 CPU 发出请求的

8.2.2 高级可编程控制器(APIC)

8.2.3 中断请求级(IRQL)

用户模式的代码是运行在最低优先级的 PASSIVE_LEVEL 级别。驱动程序的 DriverEntry 函数，派遣函数，AddDevice 等函数一般都运行在 PASSIVE_LEVEL 级别上。他们在必要时可以申请进入 DISPATCH_LEVEL 级别

Windows 负责线程调度的组件是运行在 DISPATCH_LEVEL 级别，当前的线程运行完时间片后，系统自动从 PASSIVE_LEVEL 级别提升到 DISPATCH_LEVEL 级别。当线程切换完毕后，操作系统

又从 DISPATCH_LEVEL 级别降到 PASSIVE_LEVEL 级别

驱动程序的 StartIO 函数和 DPC 函数也运行在 DISPATCH_LEVEL 级别

在内核模式下，可以通过调用 KeGetCurrentIrql 内核函数来得到当前的 IRQL 级别

8.2.4 线程调度与线程优先级

线程优先级和 IRQL 是两个容易混淆的概念。所有应用程序都运行在 PASSIVE_LEVEL 级别上，它的优先级别最低，可以被其他 IRQL 级别的程序打断。线程优先级只针对应用程序而言，只有程序运行在 PASSIVE_LEVEL 级别才有意义。

线程优先级是指某县城是否有更过的机会运行在 CPU 上，线程优先级高的线程有更多的机会被内核调度。负责调度线程的内核组件运行在 DISPATHCH_LEVEL 级别的 IRQL 上，这时候所有应用程序都已停止，等待着被调度。

ReadFile 内部创建 IRP_MJ_READ，然后这个 IRP 被传递到驱动程序的派遣函数中，这时候派遣函数运行在 ReadFile 所在的线程中。或者说 ReadFile 和派遣函数位于同一个线程上下文中。

8.2.5 IRQL 的变化

如果提升 IRQL 到 DISPATCH_LEVEL 级别，这时候不会出现线程的切换。这是一种很常用的同步处理机制。但只能适用于单 CPU 的系统。

8.2.6 IRQL 与内存分页

在使用内存分页的时候，可能会导致页故障。因为分页内存随时可能从屋里内存交换到磁盘文件。读取不在物理内存中的分页内存时，会引发一个页故障，从而执行这个异常的处理函数。

异常处理函数会重新将磁盘文件的内容交换到物理内存中。

页故障允许出现在 PASSIVE_LEVEL 级别的程序中，但如果在 DISPATCH_LEVEL 或者更高级别 IRQL 的程序中会带来系统崩溃。

对于等于或高于 DISPATCH_LEVEL 级别的程序不能使用分页内存，必须使用非分页内存。驱动程序的 StartIO 例程，DPC 例程，中断服务例程都运行在 DISPATCH_LEVEL 或者更高的 IRQL. 因此，在这些例程中不能使用分页内存，否则会导致系统崩溃。

8.2.7 控制 IRQL 提升与降低

这一般是基于同步处理的需要。

VOID RaiseIRQL_Test()

```
{
KIRQL     oldirql;
ASSERT(KeGetCurrentIrql()<=DISPATCH_LEVEL)
KeRaiseIrql(DISPATCH_LEVEL,&oldirql);
...
KeLowerIrql(oldirql);
}
```

8.3 自旋锁

自旋锁也是一种同步处理机制，它能保证某个资源只能被一个线程所拥有。

8.3.1 原理

在 Windows 内核中，有一种被称为自旋锁(Spin Lock)的锁，它可以用于驱动程序中的同步处理。初始化自旋锁时，处于解锁状态，这时它可以被程序“获取”。获取后的自旋锁处于锁住状态。锁住的自旋锁必须“释放”后，才能再次被获取。

如果自旋锁已经被锁住，这时有程序申请获取这个这个自旋锁，程序则处于自旋状态。所谓自旋状态，就是不停地询问是否可以获取自旋锁。自旋锁也因此得名。

自旋锁不同于线程中的等待事件。在线程中如果等待某个事件(Event),操作系统会使这个线程进入休眠状态，CPU 会运行其他线程。而自旋锁是一直让这个线程自旋。因此，对自旋锁占用时间不宜过长。否则会导致申请自旋锁的其他线程处于自旋，这会浪费 CPU 宝贵的时间。

在单 CPU 的系统中，获取自旋锁仅仅是将当前的 IRQL 从 PASSIVE_LEVEL 级别提升到 DISPATCH_LEVEL 级别。但是在多 CPU 的系统中，自旋锁的实现方法会发杂得多。

需要注意的是，驱动程序必须在低于或者等于 DISPATCH_LEVEL 的 IRQL 级别中使用自旋锁。

8.3.2 使用方法

自旋锁的作用一般是为使各派遣函数之间同步。尽量不要将自旋锁放在全局变量中。而应该将自旋锁放在设备扩展里。自旋锁用 KSPIN_LOCK 数据结构表示。

```
typedef struct    _DEVICE_EXTENSION
{
….
KSPIN_LOCK     My_SpinLock;
}DEVICE_EXTENSION,*PDEVICE_EXTENSION;
```

使用自旋锁前，需要先对其进行初始化，可以使用 KeInitializeSpinLock 内核函数。一般在驱动程序的 DriverEntry 或者 AddDevice 函数中初始化自旋锁。

申请自旋锁可以使用内核函数 KeAcquireSpinLock,它

```
PDEVICE_EXTENSION pdx=(PDEVICE_EXTENSION)pDevObj->DeviceExtension;
KIRQL    olcirql;
KeAcquireSpinLock(&pdx->My_SpinLock,oldirql);
```

如果在 DISPATCH_LEVEL 级别申请自旋锁，不会改变 IRQL 级别。这时，申请自旋锁可以简单的使用 KeAcquireSpinLockAtDpcLevel 内核函数，而释放自旋锁使用 KeReleaseSpinLockFromDpcLevel 内核函数。

8.4 用户模式下的同步对象

在内核模式下可以使用很多内核同步对象，这些内核对象和同步模式下的同步对象非常类似。同步对象包括事件（EVENT),互斥体(Mutex),信号灯(Semaphore)等。

用户膜下的同步对象都是借助内核模式的同步对象实现的。用户模式下的同步对象其实是内核模式下的同步对象的再次封装。

8.4.1 用户模式的等待

在应用程序中，可以使用 WaitForSingleObject 和 WaitForMultipleObjects 等待同步对象。

```
DWORD    WaitForSingleObject(
HANDLE    hHandle,
DWORD     dwMilliseconds);
```

如果同步有两种状态。一种是激发状态，一种是未激发状态。如果同步对象处于未激发状态，WaitForSingleObject 则进入休眠，等待同步对象被激发。如果同步对象在指定的等待时间内，还没有处于激发状态，则自动停止休眠。
dwMilliseconds 也可以为 0，其作用是强迫操作系统将当前线程切换到其他线程。

```
DWORD    WaitForMultipleObjects(
DWORD nCount,
CONST HANDLE *lpHandles,
BOOL    bWaitAll,
DWORD dwMilliseconds);
```

如果 bWaitAll 为假，则只要有一个同步对象被激发，则线程恢复运行。

8.4.2 用户模式开启多线程

等待同步对象一般出现在多线程的编程中，因此这里介绍一下应用程序如何创建新线程。

```
HANDLE    CreateThread(
LPSECURITY_ATTRIBUTES    lpThreadAttributes,
SIZE_T      dwStackSize,
LLPTHREAD_START_ROUTINE    lpStartAddress,
LPVOID      lpParameter,
DWORD    dwCreationFlags,
LPDWORD    lpThreadId);
```

CreateThread 用 lpStartAddress 参数指定新线程的运行地址。用 dwStackSize 参数指定新线程的堆栈大小。用 lpParameter 指定新线程的参数等信息。

一些旧的运行时函数不支持多线程。运行时函数使用了大量的全局变量和静态变量。

8.4.3 用户模式的事件

事件是一种典型的同步对象。用户模式下的事件和内核模式下的事件对象紧密相连。
在使用事件之前，应该先对它进行初始化。

```
HANDLE    CreateEvent(
LPSECURITY_ATTRIBUTES lpEventAtrributes,
BOOL    bManualReset,
BOOL    bInitialState,
LPCTSTR      lpName);
```

所有形如 CreateXXX 的 Win32 API 函数，如果它的第一个函数是 LPSECURITY_ATTRIBUTESl 类型，那么这种 API 内部都会创建一个相应的内核对象。这种 API 返回一个句柄，操作系统可以通过这个句柄找到具体的内核对象。

CreateEvent 内部会使操作系统创建一个内核事件对象。

应用程序无法获得 CreateEvent 创建的内核事件对象的指针，而用一个句柄代表事件对象。

在这个例子的主线程中，开启新的辅助线程。主线程把一个事件的句柄传递给子线程。同时，主线程等待该时间激发。
辅助线程所作的事情就是显示一些信息，并设置该事件。

8.4.4 用户模式的信号灯

信号灯也是一种常用的同步对象，信号灯也有两种状态：激发状态，未激发状态。
信号灯内部有个计数器，可以理解信号灯内部有 N 个灯泡。如果有一个灯泡亮着，就代表

着信号灯处于激发状态。

```
HANDLE CreateSemaphore(
LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
LONG lInitialCount,
LONG lMaximumCount,
LPCTSTR    lpName);
```

增加信号灯的计数器

```
BOOL    ReleaseSemaphore(
HANDLE    hSemaphore,
LONG lReleaseCount,
LPLONG    lpPreviousCount);
```

对信号灯执行一次等待操作，就会减少一个计数。相当于熄灭一个灯泡。当计数器为 0 时，也就是所有灯泡都熄灭时，当前进程进入睡眠状态。直到信号灯变为激发状态。

8.4.5 用户模式的互斥体

互斥体也是一种常用的同步对象。互斥体可以避免多个线程争夺同一个资源。
例如，多线程环境中，只能有一个线程占优互斥体。
互斥体的概念类似于同步事件，所不同的是同一个线程可以递归获得互斥体。
递归获得互斥体就是得到互斥体的线程还可以再次获得这个互斥体，或者说互斥体对于已经获得互斥体的线程不产生互斥关系。而同步事件不能递归获取。
互斥体也有两种状态。激发态和未激发态。如果线程获得互斥体时，此时的状态是为激发状态，当释放互斥体时，互斥体的状态为激发状态。

```
HANDLE    CreateMutex(
LPSECURITY_ATTRIBUTES lpMutexAttributes,
BOOL bInitialOwner,
LPCTSTR    lpName);
```

8.4.6 等待线程完成

还有一种同步对象，就是线程对象。每个线程同样有两个状态，激发状态和未激发状态。当线程处于运行之中的时候，是未激发状态。当线程终止后，线程处于激发状态。可以用 WaitThread 函数对线程进行等待。
下面的例子演示了如何等待线程的结束。例子中，主线程创建两个线程，主线程用

WaitForMultipleObjects 等待两个子线程全部运行结束。

8.5 内核模式下的同步对象

在内核模式下，有一系列的同步对象与用户模式下的同步对象相对应。在用户模式下，各个函数都是以句柄操作同步对象的。而在用户模式下，程序员无法获得真实同步对象的指针。而用一个句柄（其实是一个 32 位整数）代表这个对象。在内核模式下，程序员可以获得真实同步对象的指针。

每种同步对象在内核中都会对应一种数据结构，但是在内核模式下，程序员可以很自由的操作这些对象。但要注意仔细使用这些同步对象。否则会引起系统的死锁。

8.5.1 内核模式下的等待

在内核模式下，同样有两个函数负责等待内核同步对象。

```
NTSTATUS    KeWaitForSingleObject(
IN PVOID Object,
IN KWAIT_REASON WaitReason,
IN KPROCESSOR_MODE WaitMode,
IN BOOLEAN Alertable,
IN PLARGE_INTEGER Timeout    OPTIONAL);
```

Object:是一个同步对象的指针，不是句柄。
WaitReason:Executive.
WaitMode:KernelMode.
Alertable:FALSE;
Timeout:NULL.无限期的等待。直到激发态。对于线程是停止了。

如果等待的同步对象变为激发态，这个函数会退出睡眠状态。并返回 STATUS_SUCCESS.如果这个函数(KeWaitForSingleObject)是因为超时而退出，则返回 STATUS_TIMEOUT.

```
NTSTATUS    KeWaitForMultipleObjects(
IN ULONG Count,
IN PVOID Object[],
IN WAIT_TYPE WaitType,
IN KWAIT_REASON WaitReason,
IN KPROCESSOR_MODE WaitMode,
IN BOOLEAN Alertable,
IN PLARGE_INTEGER Timeout OPTIONAL,
IN PKWAIT_BLOCK WaitBlockArray    OPTIONAL);
```

如何使用 KeWaitForMultipleObjects 函数。

```
NTSTATUS status=KeWaitForMultipleObjects(...);
if(NT_SUCCESS(status))
{
ULONG index=status-STATUS_WAIT_0;
}
```

8.5.2 内核模式下开启多线程

内核函数 PsCreateSystemThread 负责创建新线程。该函数可以创建两种线程，一种是用户线程，一种是系统线程。

用户线程是属于当前进程中的线程。当前进程指的是当前 I/O 操作的发起者。

如果在 IRP_MJ_READ 的派遣函数中调用 PsCreateSystemThread 函数创建用户线程，新线程就属于 ReadFile 的进程。

系统线程不属于当前用户进程。而属于系统进程。

驱动程序中的 DriverEntry 和 AddDevice 等函数都是被某个系统线程调用的。

```
NTSTATUS    PsCreateSystemThread(
OUT PHANDLE ThreadHandle,
IN ULONG DesiredAccess,
IN POBJECT_ATTRIBUTES ObjectAttributes    OPTIONAL,
IN HANDLE    ProcessHandle OPTIONAL,
OUT PCLIENT_ID    ClientId OPTIONAL,
IN PKSTART_ROUTINE    StartRoutine,
IN PVOID StartContext);
```

ThreadHandle:用于输出，这个参数得到新创建的线程句柄。

DesiredAccess:创建的权限。

ObjectAttributes:线程的属性，一般设为 NULL。

ProcessHandle:创建用户线程还是系统线程。为 NULL,则创建系统线程，如果该值是一个进程句柄，则新创建的线程属于这个指定的进程。DDK 提供的宏 NtCurrentProcess 可以得到当前进程的句柄。

StartRoutine:为新线程的运行地址。

StartContext:新线程接收的参数。

在内核模式下，创建的线程必须用函数 PsTerminateSystemThread 强制线程结束。否则，该线程是无法自动退出的。

IoGetCurrentProcess 函数得到 PEPROCESS 数据结构。

查看当前线程的进程。

```
PEPROCESS pEProcess=IoGetCurrentProcess();
PTSTR    ProcessName=(PTSTR)((ULONG)pEProcess+0x174);
KdPrint(("This thread run in %s process!\n",ProcessName));
```

用户线程在 Text.exe 进程。
系统线程在系统进程。。

8.5.3 内核模式下的事件对象

在应用程序中，程序员只能操作事件句柄，无法得到事件对象的指针。在与事件相关的 Win32API 函数的内部，会调用内核模式事件的相关函数。

在内核中，用 KEVENT 数据结构表示一个事件对象。在使用事件对象前，需要进行初始化。内核函数 KeInitializeEvent 负责对事件对象初始化。

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

Event:这个参数是初始化事件对象的指针。
Type:这个参数是事件的类型。一个是通知事件，NotificationEvent,另一个是同步事件，SynchronizationEvent.
State:这个参数如果为真，事件对象的初始化状态为激发状态。如果此参数为假，则事件对象的初始化状态为未激发状态。

如果创建的事件对象是通知事件，当事件对象变为激发状态时，程序员需要手动将其改回未激发状态。如果创建的事件对象是同步事件，当事件对象为激发态时，如遇到 KeWaitForXX 等内核函数，事件对象则自动变回未激发状态。

8.5.4 驱动程序与应用程序交互事件对象

前面已经介绍过,应用程序中创建的事件和在内核中创建的事件对象,本质上是同一个东西。

本节介绍的就是如何在应用程序和驱动程序中共同使用一个事件对象。

需要解决的第一个问题就是如何将用户模式下创建的事件传递给驱动程序。解决的办法是采用 DeviceIoControlAPI 函数。在用户模式下创建一个同步事件。然后用 DeviceIoControl 把事件句柄传递给驱动程序。需要指出的是，句柄是与进程相关的，也就意味着一个进程中的句柄只能在这个进程中有效。句柄相当于事件对象在进程中的索引。通过这个索引操作系统就会得到事件对象的指针。DDK 提供了内核函数将句柄转化为指针。该函数是 ObReferenceObjectByHandle.

ObReferenceObjectByHandle 函数在得到指针的同时，会为对象的指针维护一个计数。

ObReferenceObjectByHandle 函数会返回一个状态值。表明是否成功得到指针。

8.5.5 驱动程序与驱动程序交互事件对象

有时候，还需要在驱动程序与驱动程序中交互事件对象。例如，驱动程序 A 的某个派遣函数要与驱动程序 B 的派遣函数进行同步。就需要两个驱动程序之间交互事件对象。

关键的问题就是如何让驱动程序 A 获取驱动程序 B 中创建的事件对象。最简单的方法是驱动程序 B 创建一个有“名字”的事件对象。这样驱动程序 A 就可以根据名字寻找到事件对象的指针。

IoCreateNotificationEvent. IoCreateSynchronizationEvent.都是内核函数。

这两个函数都用 UICODE_STRING 字符串描述内核事件对象的名字。
在驱动程序之间的交互中得到内核事件对象的句柄，再用 ObReferenceObjectByHandle 内核函数获取内核事件对象的指针。

8.5.6 内核模式下的信号灯

在内核中还有另一种同步对象，这就是信号灯。和事件对象一样，信号灯在用户模式下合内核模式下是完全统一的。只不过操作方式不同。在用户模式下，信号灯对象用句柄代表，而在内核模式下，信号灯对象用 KSEMAPHORE 数据结构表示。

```
VOID    KeInitializeSemaphore(
IN PRKSEMAPHORE    Semaphore,
IN LONG Count,
IN LONG Limit);
```

Semaphore:这个参数获得内核信号灯对象指针。

KeReadStateSemaphore 函数可以读取信号灯当前的计数。

释放信号灯会增加信号灯计数。它对应的内核函数是 KeReleaseSemaphore.程序员可以用这个函数指定增量值。

获得信号灯可以使用 KeWaitXX 系列函数。如果获得就将计数减一，否则陷入等待。

和事件对象一样，信号灯对象也可以在应用程序和驱动程序中交互。

8.5.7 内核模式下的互斥体

互斥体在内核中的数据结构是 KMUTEX.

```
VOID    KeInitializeMutex(
IN PRKMUTEX    Mutex,
IN ULONG Level);
```

Mutex:这个参数可以获得内核互斥体对象的指针。
Level:保留至，一般设为 0.

初始化后的互斥体对象，就可以使线程之间互斥了。获得互斥体对象用 KeWaitForSingleObject。
释放互斥体用 KeReleaseMutex 内核函数。

8.5.8 快速互斥体

快速互斥体是 DDK 提供 的另外一种内核同步对象。
因为执行的速度比普通互斥体速度快（这里指获取和释放的速度）。然而，快速互斥体比普通互斥体多了一个缺点。就是不能递归获取互斥体对象。

快速互斥体在内核中是用 FAST_MUTEX 数据结构描述的。
ExInitializeFastMutex.ExAcquireFastMutex,ExReleaseFastMutex..

下面的例子演示如何在驱动程序中使用快速互斥体。
跟普通互斥体差不多一样。

8.6 其他同步方法

上一节主要介绍了内核中主要的同步对象。除了使用同步对象外，还有几种方法可以实现同

步。

8.6.1 使用自旋锁进行同步

在驱动程序中，经常使用自旋锁作为一种有效的同步机制。
对于要同步的代码，需要用一把自旋锁进行同步。

```
KeAcquireSpinLock();///A 点
...
KeReleaseSpinLock();///B 点。
```

A-B 同步区域。。。。。

8.6.2 使用互锁操作进行同步

```
int number=0;
void Foo()
{
InterlockedIncrement(&number);
InterlockedDecrement(&number);
}
```

8.7 小结

本章介绍了驱动程序中常用的同步处办法，并且将内核模式下的同步处理方法和用户模式下的同步处理方法做了比较。