在应用层下的文件操作只需要调用微软应用层下的`API`函数及`C库`标准函数即可，而如果在内核中读写文件则应用层的API显然是无法被使用的，内核层需要使用内核专有API，某些应用层下的API只需要增加Zw开头即可在内核中使用，例如本章要讲解的文件与目录操作相关函数，多数ARK反内核工具都具有对文件的管理功能，实现对文件或目录的基本操作功能也是非常有必要的。

首先无论在内核态还是在用户态，我们调用的文件操作函数其最终都会转换为一个IRP请求，并发送到文件系统驱动上的`IRP_MJ_READ`派遣函数里面，这个读写流程大体上可分为如下四步；

- 对于FAT32分区会默认分发到`FASTFAT.SYS`，而相对于NTFS分区则会分发到`NTFS.SYS`驱动上。
- 文件系统驱动经过处理后，就把IRP传给磁盘类驱动的`IRP_MJ_READ`分发函数处理，当磁盘类驱动处理完毕后,又把IRP传给磁盘小端口驱动。
- 在磁盘小端口驱动里，无论是读还是写，用的都是`IRP_MJ_SCSI`这个分发函数。
- IRP被磁盘小端口驱动处理完之后，就要依靠`HAL.DLL`进行端口IO，此时数据就真的从硬盘里读取了出来。

**创建文件或目录:** 实现创建文件或目录，创建文件或目录都可调用`ZwCreateFile()`这个内核函数来实现，唯一不同的区别在于当用户传入参数中包含有`FILE_SYNCHRONOUS_IO_NONALERT`属性时则会默认创建文件，而如果包含有`FILE_DIRECTORY_FILE`属性则默认为创建目录，该函数的微软定义以及备注信息如下所示；

```
NTSYSAPI NTSTATUS ZwCreateFile(
  [out]          PHANDLE            FileHandle,         // 指向HANDLE变量的指针，该变量接收文件的句柄。
  [in]           ACCESS_MASK        DesiredAccess,      // 指定一个ACCESS_MASK值，该值确定对对象的请求访问权限。
  [in]           POBJECT_ATTRIBUTES ObjectAttributes,   // 指向OBJECT_ATTRIBUTES结构的指针，该结构指定对象名称和其他属性。
  [out]          PIO_STATUS_BLOCK   IoStatusBlock,      // 指向IO_STATUS_BLOCK结构的指针，该结构接收最终完成状态和有关所请求操作的其他信息。
  [in, optional] PLARGE_INTEGER     AllocationSize,     // 指向LARGE_INTEGER的指针，其中包含创建或覆盖的文件的初始分配大小（以字节为单位）。
  [in]           ULONG              FileAttributes,     // 指定一个或多个FILE_ATTRIBUTE_XXX标志，这些标志表示在创建或覆盖文件时要设置的文件属性。
  [in]           ULONG              ShareAccess,        // 共享访问的类型，指定为零或以下标志的任意组合。
  [in]           ULONG              CreateDisposition, // 指定在文件存在或不存在时要执行的操作。
  [in]           ULONG              CreateOptions,      // 指定要在驱动程序创建或打开文件时应用的选项。
  [in, optional] PVOID              EaBuffer,           // 对于设备和中间驱动程序，此参数必须是NULL指针。
  [in]           ULONG              EaLength            // 对于设备和中间驱动程序，此参数必须为零。
);
```

参数`DesiredAccess`用于指明对象访问权限的，常用的权限有`FILE_READ_DATA`读取文件，`FILE_WRITE_DATA`写入文件，`FILE_APPEND_DATA`追加文件，`FILE_READ_ATTRIBUTES`读取文件属性，以及`FILE_WRITE_ATTRIBUTES`写入文件属性。

参数`ObjectAttributes`指向了一个`OBJECT_ATTRIBUTES`指针，通常会通过`InitializeObjectAttributes()`宏对其进行初始化，当一个例程打开对象时由此结构体指定目标对象的属性。

参数 ShareAccess 用于指定访问属性，通常属性有 FILE_SHARE_READ 读取，FILE_SHARE_WRITE 写入，FILE_SHARE_DELETE 删除。

参数 CreateDisposition 用于指定在文件存在或不存在时要执行的操作，一般而言我们会指定为 FILE_OPEN_IF 打开文件，或 FILE_OVERWRITE_IF 打开文件并覆盖，FILE_SUPERSEDE 替换文件。

参数 CreateOptions 用于指定创建文件或目录，一般 FILE_SYNCHRONOUS_IO_NONALERT 代表创建文件，参数 FILE_DIRECTORY_FILE 代表创建目录。

相对于创建文件而言删除文件或目录只需要调用 ZwDeleteFile() 系列函数即可，此类函数只需要传递一个 OBJECT_ATTRIBUTES 参数即可，其微软定义如下所示；

```
NTSYSAPI NTSTATUS ZwDeleteFile(
  [in] POBJECT_ATTRIBUTES ObjectAttributes
);
```

接下来我们就封装三个函数 MyCreateFile() 用于创建文件，MyCreateFileFolder() 用于创建目录，MyDeleteFileOrFileFolder() 用于删除空目录。

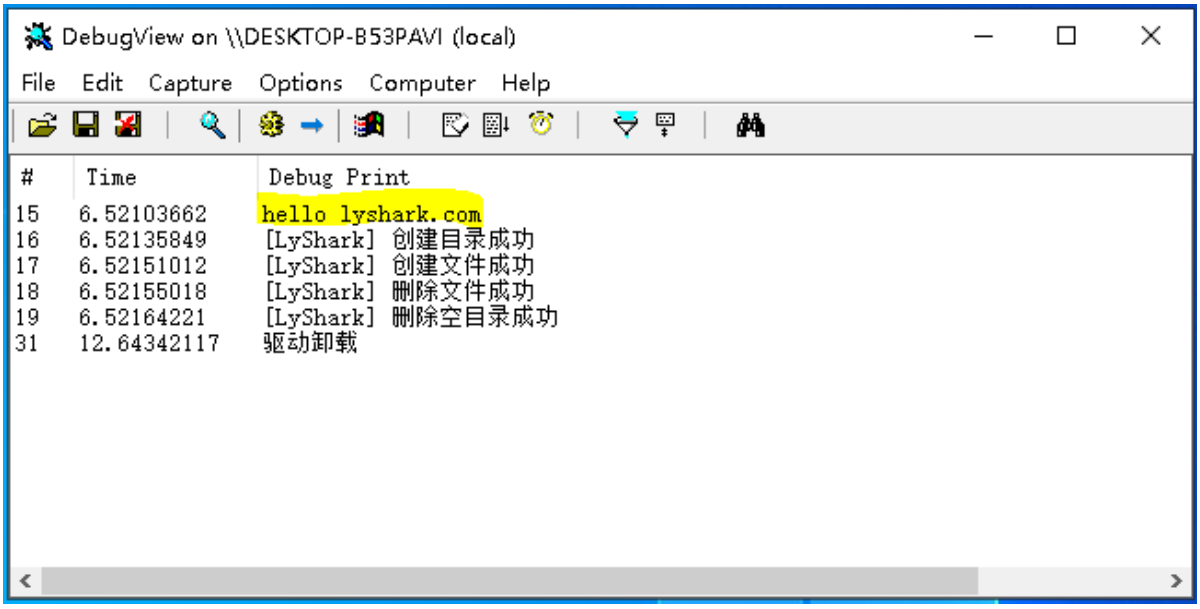
```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <ntifs.h>
#include <ntstrsafe.h>

// 创建文件
BOOLEAN MyCreateFile(UNICODE_STRING ustrFilePath)
{
    HANDLE hFile = NULL;
    OBJECT_ATTRIBUTES objectAttributes = { 0 };
    IO_STATUS_BLOCK iosb = { 0 };
    NTSTATUS status = STATUS_SUCCESS;

    // 初始化对象属性结构体 FILE_SYNCHRONOUS_IO_NONALERT
    InitializeObjectAttributes(&objectAttributes, &ustrFilePath,
OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

    // 创建文件
    status = ZwCreateFile(&hFile, GENERIC_READ, &objectAttributes, &iosb, NULL,
FILE_ATTRIBUTE_NORMAL, 0, FILE_OPEN_IF, FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
    if (!NT_SUCCESS(status))
    {
        return FALSE;
    }

    // 关闭句柄
    ZwClose(hFile);

    return TRUE;
}

// 创建目录
BOOLEAN MyCreateFileFolder(UNICODE_STRING ustrFileFolderPath)
```

```
{
    HANDLE hFile = NULL;
    OBJECT_ATTRIBUTES objectAttributes = { 0 };
    IO_STATUS_BLOCK iosb = { 0 };
    NTSTATUS status = STATUS_SUCCESS;

    // 初始化对象属性结构体
    InitializeObjectAttributes(&objectAttributes, &ustrFileFolderPath,
OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

    // 创建目录 FILE_DIRECTORY_FILE
    status = ZwCreateFile(&hFile, GENERIC_READ, &objectAttributes, &iosb, NULL,
FILE_ATTRIBUTE_NORMAL, 0, FILE_CREATE, FILE_DIRECTORY_FILE, NULL, 0);
    if (!NT_SUCCESS(status))
    {
        return FALSE;
    }
    // 关闭句柄
    ZwClose(hFile);

    return TRUE;
}

// 删除文件或是空目录
BOOLEAN MyDeleteFileOrFileFolder(UNICODE_STRING ustrFileName)
{
    NTSTATUS status = STATUS_SUCCESS;
    OBJECT_ATTRIBUTES objectAttributes = { 0 };

    // 初始化属性
    InitializeObjectAttributes(&objectAttributes, &ustrFileName,
OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

    // 执行删除操作
    status = ZwDeleteFile(&objectAttributes);
    if (!NT_SUCCESS(status))
    {
        return FALSE;
    }

    return TRUE;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    BOOLEAN ref = FALSE;

    // 删除文件
    UNICODE_STRING ustrDeleteFile;
    RtlInitUnicodeString(&ustrDeleteFile, L"\\??
\\C:\\LySharkFolder\\lyshark.txt");
    ref = MyDeleteFileOrFileFolder(ustrDeleteFile);
    if (ref != FALSE)
    {
        DbgPrint("[LyShark] 删除文件成功 \n");
    }
```

```
    else
    {
        DbgPrint("[LyShark] 删除文件失败 \n");
    }

    // 删除空目录
    UNICODE_STRING ustrDeleteFilder;
    RtlInitUnicodeString(&ustrDeleteFilder, L"\\??\\C:\\LySharkFolder");
    ref = MyDeleteFileOrFileFolder(ustrDeleteFilder);
    if (ref != FALSE)
    {
        DbgPrint("[LyShark] 删除空目录成功 \n");
    }
    else
    {
        DbgPrint("[LyShark] 删除空目录失败 \n");
    }

    DbgPrint("驱动卸载 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    BOOLEAN ref = FALSE;


    // 创建目录
    UNICODE_STRING ustrDirectory;
    RtlInitUnicodeString(&ustrDirectory, L"\\??\\C:\\LySharkFolder");
    ref = MyCreateFileFolder(ustrDirectory);
    if (ref != FALSE)
    {
        DbgPrint("[LyShark] 创建目录成功 \n");
    }
    else
    {
        DbgPrint("[LyShark] 创建文件失败 \n");
    }

    // 创建文件
    UNICODE_STRING ustrCreateFile;
    RtlInitUnicodeString(&ustrCreateFile, L"\\??
\\C:\\LySharkFolder\\lyshark.txt");
    ref = MyCreateFile(ustrCreateFile);
    if (ref != FALSE)
    {
        DbgPrint("[LyShark] 创建文件成功 \n");
    }
    else
    {
        DbgPrint("[LyShark] 创建文件失败 \n");
    }

    Driver->DriverUnload = UnDriver;
```

```
    return STATUS_SUCCESS;
}
```

运行如上代码，分别创建 `LySharkFolder` 目录，并在其中创建 `lyshark.txt` 最终再将其删除，输出效果如下；




**重命名文件或目录:** 在内核中重命名文件或目录核心功能的实现依赖于 `ZwSetInformationFile()` 这个内核函数，该函数可用于更改有关文件对象的各种信息，其微软官方定义如下；

```
NTSYSAPI NTSTATUS ZwSetInformationFile(
  [in]  HANDLE                 FileHandle,          // 文件句柄
  [out] PIO_STATUS_BLOCK       IoStatusBlock,       // 指向 IO_STATUS_BLOCK 结构的
指针
  [in]  PVOID                  FileInformation,     // 指向缓冲区的指针，该缓冲区包含
要为文件设置的信息。
  [in]  ULONG                  Length,              // 缓冲区的大小（以字节为单位）
  [in]  FILE_INFORMATION_CLASS FileInformationClass // 为文件设置的类型
);
```

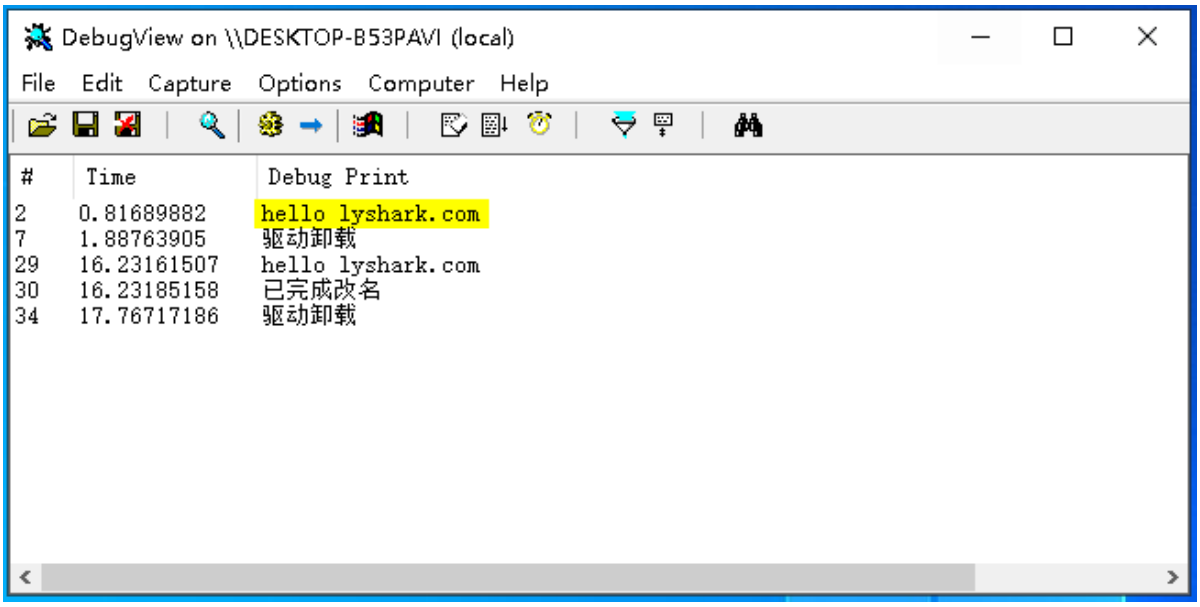
这其中最重要的参数就是 `FileInformationClass` 根据该参数的不同则对文件的操作方式也就不同，如果需要重命名文件则此处应使用 `FileRenameInformation` 而如果需要修改文件的当前信息则应使用 `FilePositionInformation` 创建链接文件则使用 `FileLinkInformation` 即可，以重命名为例，首先我们需要定义一个 `FILE_RENAME_INFORMATION` 结构并按照要求填充，最后直接使用 `ZwSetInformationFile()` 并传入相关信息后即可完成修改，其完整代码流程如下；

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <ntifs.h>
#include <ntstrsafe.h>

// 重命名文件或文件夹
BOOLEAN MyRename(UNICODE_STRING ustrSrcFileName, UNICODE_STRING ustrDestFileName)
{
    HANDLE hFile = NULL;
```

```
    OBJECT_ATTRIBUTES objectAttributes = { 0 };
    IO_STATUS_BLOCK iosb = { 0 };
    NTSTATUS status = STATUS_SUCCESS;

    PFILE_RENAME_INFORMATION pRenameInfo = NULL;

    ULONG ulLength = (1024 + sizeof(FILE_RENAME_INFORMATION));

    // 为PFILE_RENAME_INFORMATION结构申请内存
    pRenameInfo = (PFILE_RENAME_INFORMATION)ExAllocatePool(NonPagedPool,
ulLength);
    if (NULL == pRenameInfo)
    {
        return FALSE;
    }

    // 设置重命名信息
    RtlZeroMemory(pRenameInfo, ulLength);

    // 设置文件名长度以及文件名
    pRenameInfo->FileNameLength = ustrDestFileName.Length;
    wcscpy(pRenameInfo->FileName, ustrDestFileName.Buffer);
    pRenameInfo->ReplaceIfExists = 0;
    pRenameInfo->RootDirectory = NULL;

    // 初始化结构
    InitializeObjectAttributes(&objectAttributes, &ustrSrcFileName,
OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

    // 打开文件
    status = ZwCreateFile(&hFile, SYNCHRONIZE | DELETE, &objectAttributes, &iosb,
NULL, 0, FILE_SHARE_READ, FILE_OPEN, FILE_SYNCHRONOUS_IO_NONALERT |
FILE_NO_INTERMEDIATE_BUFFERING, NULL, 0);
    if (!NT_SUCCESS(status))
    {
        ExFreePool(pRenameInfo);
        return FALSE;
    }
    // 利用ZwSetInformationFile来设置文件信息
    status = ZwSetInformationFile(hFile, &iosb, pRenameInfo, ulLength,
FileRenameInformation);
    if (!NT_SUCCESS(status))
    {
        ZwClose(hFile);
        ExFreePool(pRenameInfo);
        return FALSE;
    }

    // 释放内存,关闭句柄
    ExFreePool(pRenameInfo);
    ZwClose(hFile);

    return TRUE;
}

VOID UnDriver(PDRIVER_OBJECT driver)
```

```
{
    DbgPrint("驱动卸载 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    // 重命名文件
    UNICODE_STRING ustrOldFile, ustrNewFile;
    RtlInitUnicodeString(&ustrOldFile, L"\\??\\C:\\MyCreateFolder\\lyshark.txt");
    RtlInitUnicodeString(&ustrNewFile, L"\\??
\\C:\\MyCreateFolder\\hello_lyshark.txt");
    BOOLEAN ref = MyRename(ustrOldFile, ustrNewFile);
    if (ref == TRUE)
    {
        DbgPrint("已完成改名 \n");
    }

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

运行后将会把 `C:\\MyCreateFolder\\lyshark.txt` 目录下的文件改名为 `hello_lyshark.txt`，前提是该目录与该文件必须存在；




那么如果你需要将文件设置为只读模式或修改文件的创建日期，那么你就需要看一下微软的定义 `FILE_BASIC_INFORMATION` 结构，依次填充此结构体并调用 `ZwSetInformationFile()` 即可实现修改，该结构的定义如下所示；

```
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;
```

当然如果你要修改日期你还需要自行填充 `LARGE_INTEGER` 结构，该结构的微软定义如下所示，分为高位和低位依次填充即可；

```
#if defined(MIDL_PASS)
typedef struct _LARGE_INTEGER {
#else // MIDL_PASS
typedef union _LARGE_INTEGER {
    struct {
        ULONG LowPart;
        LONG HighPart;
    } DUMMYSTRUCTNAME;
    struct {
        ULONG LowPart;
        LONG HighPart;
    } u;
#endif //MIDL_PASS
    LONGLONG QuadPart;
} LARGE_INTEGER;
```

我们就以修改文件属性为只读模式为例，其核心代码可以被描述为如下样子，相比于改名而言其唯一的变化就是更换了 `PFILE_BASIC_INFORMATION` 结构体，其他的基本一致；

```
HANDLE hFile = NULL;
OBJECT_ATTRIBUTES objectAttributes = { 0 };
IO_STATUS_BLOCK iosb = { 0 };
NTSTATUS status = STATUS_SUCCESS;

PFILE_BASIC_INFORMATION pReplaceInfo = NULL;

ULONG ulLength = (1024 + sizeof(FILE_BASIC_INFORMATION));

// 为FILE_POSITION_INFORMATION结构申请内存
pReplaceInfo = (PFILE_BASIC_INFORMATION)ExAllocatePool(NonPagedPool, ulLength);
if (NULL == pReplaceInfo)
{
  return FALSE;
}

RtlZeroMemory(pReplaceInfo, ulLength);

// 设置文件基础信息,将文件设置为只读模式
pReplaceInfo->FileAttributes |= FILE_ATTRIBUTE_READONLY;
```

**读取文件大小:** 读取特定文件的所占空间，核心原理是调用了 `ZwQueryInformationFile()` 这个内核函数，该函数可以返回有关文件对象的各种信息，参数传递上与 `ZwSetInformationFile()` 很相似，其 `FileInformationClass` 都需要传入一个文件类型结构，该函数的完整定义如下；

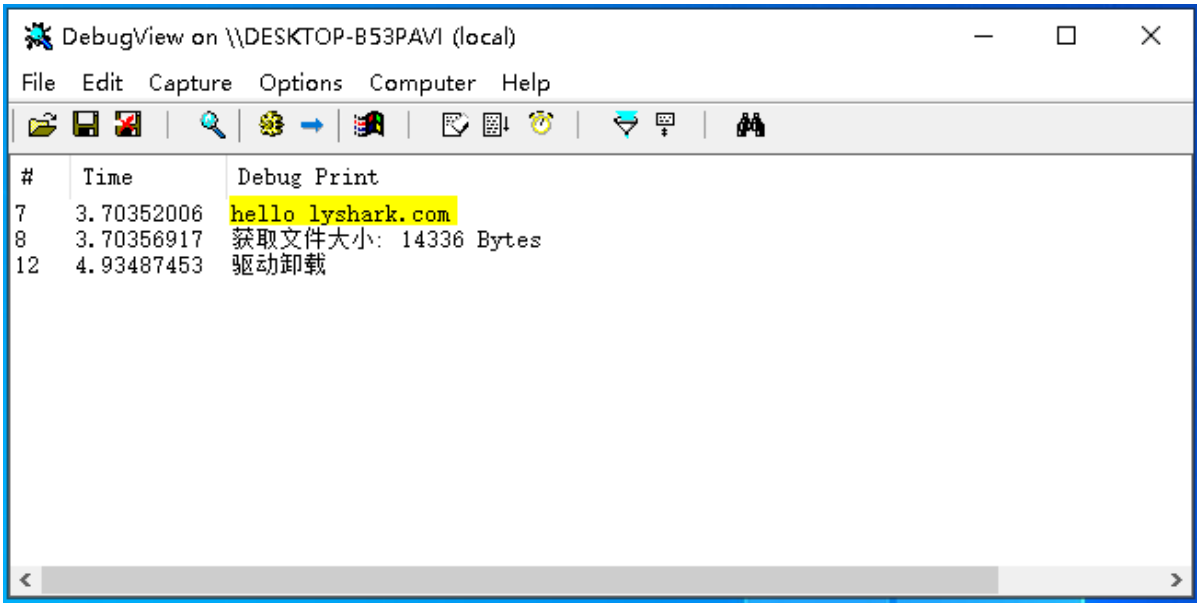
```
NTSYSAPI NTSTATUS ZwQueryInformationFile(
  [in]  HANDLE                 FileHandle,          // 文件句柄。
  [out] PIO_STATUS_BLOCK       IoStatusBlock,       // 指向接收最终完成状态和操作相关信息的 IO_STATUS_BLOCK 结构的指针。
  [out] PVOID                  FileInformation,     // 指向调用方分配的缓冲区的指针，例程将请求的有关文件对象的信息写入其中。
  [in]  ULONG                  Length,              // 长度。
  [in]  FILE_INFORMATION_CLASS FileInformationClass // 指定要在 FileInformation 指向的缓冲区中返回的有关文件的信息类型。
);
```

本例中我们需要读入文件的所占字节数，那么`FileInformation`字段就需要传入`FileStandardInformation`来获取文件的基本信息，获取到的信息会被存储到`FILE_STANDARD_INFORMATION`结构内，用户只需要解析该结构体`fsi.EndOfFile.QuadPart`即可得到文件长度，其完整代码如下所示；

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <ntifs.h>
#include <ntstrsafe.h>

// 获取文件大小
ULONG64 MyGetFileSize(UNICODE_STRING ustrFileName)
{
    HANDLE hFile = NULL;
    OBJECT_ATTRIBUTES objectAttributes = { 0 };
    IO_STATUS_BLOCK iosb = { 0 };
    NTSTATUS status = STATUS_SUCCESS;
    FILE_STANDARD_INFORMATION fsi = { 0 };

    // 初始化结构
    InitializeObjectAttributes(&objectAttributes, &ustrFileName, OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

    // 打开文件
    status = ZwCreateFile(&hFile, GENERIC_READ, &objectAttributes, &iosb, NULL, 0, FILE_SHARE_READ, FILE_OPEN, FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
    if (!NT_SUCCESS(status))
    {
        return 0;
    }

    // 获取文件大小信息
    status = ZwQueryInformationFile(hFile, &iosb, &fsi, sizeof(FILE_STANDARD_INFORMATION), FileStandardInformation);
    if (!NT_SUCCESS(status))
    {
        ZwClose(hFile);
        return 0;
    }
```

```
    return fsi.EndOfFile.QuadPart;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    // 获取文件大小
    UNICODE_STRING ustrFileSize;
    RtlInitUnicodeString(&ustrFileSize, L"\\??\\C:\\lyshark.exe");
    ULONG64 ullFileSize = MyGetFileSize(ustrFileSize);
    DbgPrint("获取文件大小: %I64d Bytes \n", ullFileSize);

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

编译并运行如上程序，即可读取到C盘下的`lyshark.exe`程序的大小字节数，如下图所示；




**内核文件读写:** 内核读取文件可以使用`ZwReadFile()`，内核写入文件则可使用`ZwWriteFile()`，这两个函数的参数传递基本上一致，如下是读写两个函数的对比参数。

```
NTSYSAPI NTSTATUS ZwReadFile(
  [in]           HANDLE           FileHandle,      // 文件对象的句柄。
  [in, optional] HANDLE           Event,           // （可选）事件对象的句柄，在读取操作完成后设置为信号状态。
  [in, optional] PIO_APC_ROUTINE  ApcRoutine,      // 此参数为保留参数。
  [in, optional] PVOID            ApcContext,      // 此参数为保留参数。
  [out]          PIO_STATUS_BLOCK IoStatusBlock,   // 接收实际从文件读取的字节数。
  [out]          PVOID            Buffer,          // 指向调用方分配的缓冲区的指针，该缓冲区接收从文件读取的数据。
  [in]           ULONG            Length,          // 缓冲区指向的缓冲区的大小（以字节为单位）。
```

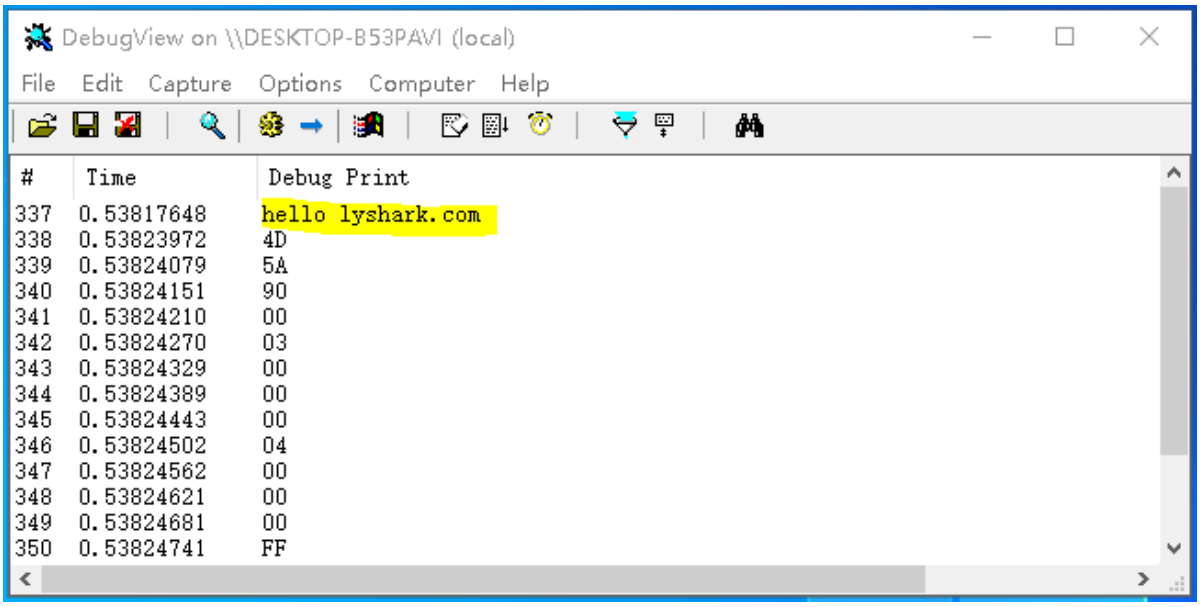
```
  [in, optional] PLARGE_INTEGER   ByteOffset,      // 指定将开始读取操作的文件中的起始字节偏移量。
  [in, optional] PULONG           Key
);

NTSYSAPI NTSTATUS ZwWriteFile(
  [in]           HANDLE           FileHandle,
  [in, optional] HANDLE           Event,
  [in, optional] PIO_APC_ROUTINE  ApcRoutine,
  [in, optional] PVOID            ApcContext,
  [out]          PIO_STATUS_BLOCK IoStatusBlock,
  [in]           PVOID            Buffer,
  [in]           ULONG            Length,
  [in, optional] PLARGE_INTEGER   ByteOffset,
  [in, optional] PULONG           Key
);
```

读取文件的代码如下所示，分配非分页`pBuffer`内存，然后调用`MyReadFile()`函数，将数据读入到`pBuffer`并输出，完整代码如下所示；

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <ntifs.h>
#include <ntstrsafe.h>

// 读取文件数据
BOOLEAN MyReadFile(UNICODE_STRING ustrFileName, LARGE_INTEGER liOffset, PUCHAR pReadData, PULONG pulReadDataSize)
{
    HANDLE hFile = NULL;
    IO_STATUS_BLOCK iosb = { 0 };
    OBJECT_ATTRIBUTES objectAttributes = { 0 };
    NTSTATUS status = STATUS_SUCCESS;

    // 初始化结构
    InitializeObjectAttributes(&objectAttributes, &ustrFileName, OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

    // 打开文件
    status = ZwCreateFile(&hFile, GENERIC_READ, &objectAttributes, &iosb, NULL,FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ | FILE_SHARE_WRITE, FILE_OPEN,FILE_NON_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
    if (!NT_SUCCESS(status))
    {
        return FALSE;
    }

    // 初始化
    RtlZeroMemory(&iosb, sizeof(iosb));

    // 读入文件
```

```
    status = ZwReadFile(hFile, NULL, NULL, NULL, &iosb, pReadData,
*pulReadDataSize, &liOffset, NULL);
    if (!NT_SUCCESS(status))
    {
        *pulReadDataSize = iosb.Information;
        ZwClose(hFile);
        return FALSE;
    }

    // 获取实际读取的数据
    *pulReadDataSize = iosb.Information;

    // 关闭句柄
    ZwClose(hFile);

    return TRUE;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    UNICODE_STRING ustrScrFile;
    ULONG ulBufferSize = 40960;
    LARGE_INTEGER liOffset = { 0 };


    // 初始化需要读取的文件名
    RtlInitUnicodeString(&ustrScrFile, L"\\??\\C:\\lyshark.exe");

    // 分配非分页内存
    PUCHAR pBuffer = ExAllocatePool(NonPagedPool, ulBufferSize);

    // 读取文件
    MyReadFile(ustrScrFile, liOffset, pBuffer, &ulBufferSize);

    // 输出文件前16个字节
    for (size_t i = 0; i < 16; i++)
    {
        DbgPrint("%02X \n", pBuffer[i]);
    }

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

编译并运行这段代码，并循环输出`lyshark.exe`文件的头16个字节的数据，效果图如下所示；



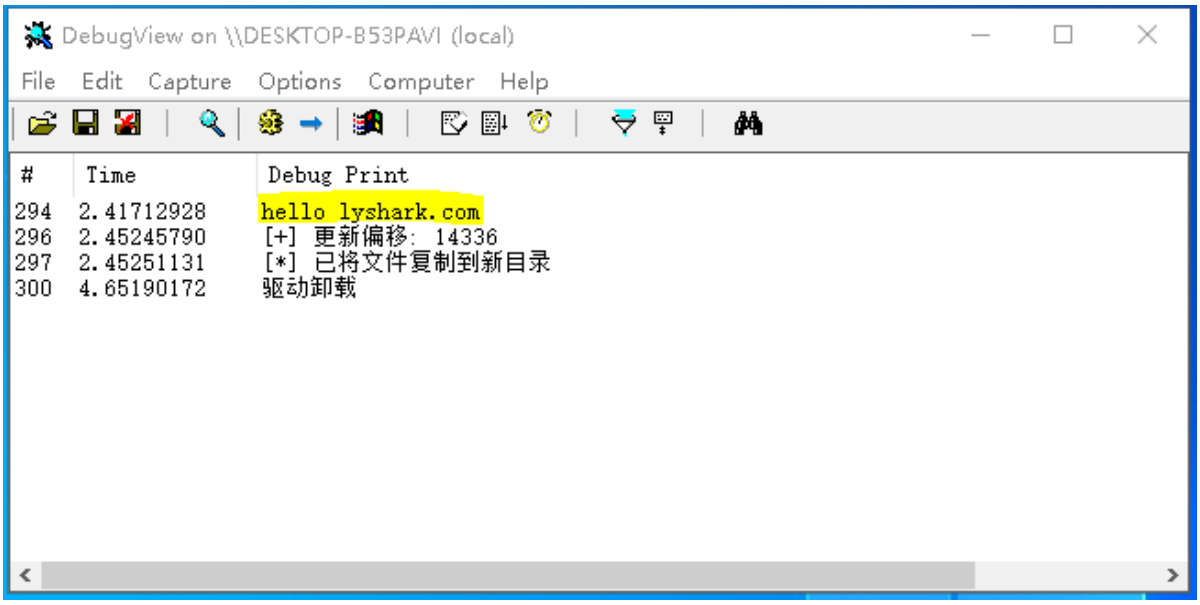

文件写入`MyWriteFile()`与读取类似，如下通过运用文件读写实现了`文件拷贝`功能，实现完整代码如下所示；

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#include <ntifs.h>
#include <ntstrsafe.h>

// 读取文件数据
BOOLEAN MyReadFile(UNICODE_STRING ustrFileName, LARGE_INTEGER liOffset, PUCHAR
pReadData, PULONG pulReadDataSize)
{
    HANDLE hFile = NULL;
    IO_STATUS_BLOCK iosb = { 0 };
    OBJECT_ATTRIBUTES objectAttributes = { 0 };
    NTSTATUS status = STATUS_SUCCESS;

    // 初始化结构
    InitializeObjectAttributes(&objectAttributes, &ustrFileName,
OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

    // 打开文件
    status = ZwCreateFile(&hFile, GENERIC_READ, &objectAttributes, &iosb,
NULL,FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ | FILE_SHARE_WRITE,
FILE_OPEN,FILE_NON_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
    if (!NT_SUCCESS(status))
    {
        return FALSE;
    }

    // 初始化
    RtlZeroMemory(&iosb, sizeof(iosb));

    // 读入文件
    status = ZwReadFile(hFile, NULL, NULL, NULL, &iosb, pReadData,
*pulReadDataSize, &liOffset, NULL);
```

```
    if (!NT_SUCCESS(status))
    {
        *pulReadDataSize = iosb.Information;
        ZwClose(hFile);
        return FALSE;
    }

    // 获取实际读取的数据
    *pulReadDataSize = iosb.Information;

    // 关闭句柄
    ZwClose(hFile);

    return TRUE;
}

// 向文件写入数据
BOOLEAN MyWriteFile(UNICODE_STRING ustrFileName, LARGE_INTEGER liOffset, PUCHAR
pWriteData, PULONG pulWriteDataSize)
{
    HANDLE hFile = NULL;
    IO_STATUS_BLOCK iosb = { 0 };
    OBJECT_ATTRIBUTES objectAttributes = { 0 };
    NTSTATUS status = STATUS_SUCCESS;

    // 初始化结构
    InitializeObjectAttributes(&objectAttributes, &ustrFileName,
OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

    // 打开文件
    status = ZwCreateFile(&hFile, GENERIC_WRITE, &objectAttributes, &iosb, NULL,
FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ | FILE_SHARE_WRITE, FILE_OPEN_IF,
FILE_NON_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
    if (!NT_SUCCESS(status))
    {
        return FALSE;
    }

    // 初始化
    RtlZeroMemory(&iosb, sizeof(iosb));

    // 写出文件
    status = ZwWriteFile(hFile, NULL, NULL, NULL, &iosb, pWriteData,
*pulWriteDataSize, &liOffset, NULL);
    if (!NT_SUCCESS(status))
    {
        *pulWriteDataSize = iosb.Information;
        ZwClose(hFile);
        return FALSE;
    }

    // 获取实际写入的数据
    *pulWriteDataSize = iosb.Information;

    // 关闭句柄
    ZwClose(hFile);
```

```
    return TRUE;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    // 文件读写
    UNICODE_STRING ustrScrFile, ustrDestFile;
    RtlInitUnicodeString(&ustrScrFile, L"\\??\\C:\\lyshark.exe");
    RtlInitUnicodeString(&ustrDestFile, L"\\??\\C:\\LyShark\\new_lyshark.exe");

    ULONG ulBufferSize = 40960;
    ULONG ulReadDataSize = ulBufferSize;
    LARGE_INTEGER liOffset = { 0 };

    // 分配非分页内存
    PUCHAR pBuffer = ExAllocatePool(NonPagedPool, ulBufferSize);

    do
    {
        // 读取文件
        ulReadDataSize = ulBufferSize;
        MyReadFile(ustrScrFile, liOffset, pBuffer, &ulReadDataSize);

        // 数据为空则读取结束
        if (0 >= ulReadDataSize)
        {
            break;
        }

        // 写入文件
        MyWriteFile(ustrDestFile, liOffset, pBuffer, &ulReadDataSize);

        // 更新偏移
        liOffset.QuadPart = liOffset.QuadPart + ulReadDataSize;
        DbgPrint("[+] 更新偏移: %d \n", liOffset.QuadPart);

    } while (TRUE);

    // 释放内存
    ExFreePool(pBuffer);
    DbgPrint("[*] 已将文件复制到新目录 \n");

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

编译并运行这段程序，则自动将 C:\\lyshark.exe 盘符下的文件拷贝到 C:\\LyShark\\new_lyshark.exe 目录下，实现效果图如下所示；




**实现文件读写传递:** 通过如上学习相信你已经掌握了如何使用文件读写系列函数了，接下来将封装一个文件读写驱动，应用层接收，驱动层读取；

此驱动部分完整代码如下所示；

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com
#include <ntifs.h>
#include <windef.h>

#define READ_FILE_SIZE_CODE
CTL_CODE(FILE_DEVICE_UNKNOWN,0x800,METHOD_BUFFERED,FILE_ALL_ACCESS)
#define READ_FILE_CODE
CTL_CODE(FILE_DEVICE_UNKNOWN,0x801,METHOD_BUFFERED,FILE_ALL_ACCESS)

#define DEVICENAME L"\\Device\\ReadWriteDevice"
#define SYMBOLNAME L"\\??\\ReadWriteSymbolName"

typedef struct
{
    ULONG64 size;       // 读写长度
    BYTE* data;         // 读写数据集
}FileData;

// 获取文件大小
ULONG64 MyGetFileSize(UNICODE_STRING ustrFileName)
{
    HANDLE hFile = NULL;
    OBJECT_ATTRIBUTES objectAttributes = { 0 };
    IO_STATUS_BLOCK iosb = { 0 };
    NTSTATUS status = STATUS_SUCCESS;
    FILE_STANDARD_INFORMATION fsi = { 0 };

    // 初始化结构
```

```
    InitializeObjectAttributes(&objectAttributes, &ustrFileName,
OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

    // 打开文件
    status = ZwCreateFile(&hFile, GENERIC_READ, &objectAttributes, &iosb, NULL,
0, FILE_SHARE_READ, FILE_OPEN, FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
    if (!NT_SUCCESS(status))
    {
        return 0;
    }

    // 获取文件大小信息
    status = ZwQueryInformationFile(hFile, &iosb, &fsi,
sizeof(FILE_STANDARD_INFORMATION), FileStandardInformation);
    if (!NT_SUCCESS(status))
    {
        ZwClose(hFile);
        return 0;
    }

    return fsi.EndOfFile.QuadPart;
}

// 读取文件数据
BOOLEAN MyReadFile(UNICODE_STRING ustrFileName, LARGE_INTEGER liOffset, PUCHAR
pReadData, PULONG pulReadDataSize)
{
    HANDLE hFile = NULL;
    IO_STATUS_BLOCK iosb = { 0 };
    OBJECT_ATTRIBUTES objectAttributes = { 0 };
    NTSTATUS status = STATUS_SUCCESS;

    // 初始化结构
    InitializeObjectAttributes(&objectAttributes, &ustrFileName,
OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);

    // 打开文件
    status = ZwCreateFile(&hFile, GENERIC_READ, &objectAttributes, &iosb, NULL,
FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ | FILE_SHARE_WRITE, FILE_OPEN,
FILE_NON_DIRECTORY_FILE | FILE_SYNCHRONOUS_IO_NONALERT, NULL, 0);
    if (!NT_SUCCESS(status))
    {
        return FALSE;
    }

    // 初始化
    RtlZeroMemory(&iosb, sizeof(iosb));

    // 读入文件
    status = ZwReadFile(hFile, NULL, NULL, NULL, &iosb, pReadData,
*pulReadDataSize, &liOffset, NULL);
    if (!NT_SUCCESS(status))
    {
        *pulReadDataSize = iosb.Information;
        ZwClose(hFile);
        return FALSE;
```

```
    }

    // 获取实际读取的数据
    *pulReadDataSize = iosb.Information;

    // 关闭句柄
    ZwClose(hFile);

    return TRUE;
}

NTSTATUS DriverIrpCtl(PDEVICE_OBJECT device, PIRP pirp)
{
    PIO_STACK_LOCATION stack;
    stack = IoGetCurrentIrpStackLocation(pirp);
    FileData* FileDataPtr;

    switch (stack->MajorFunction)
    {

    case IRP_MJ_CREATE:
    {
        break;
    }

    case IRP_MJ_CLOSE:
    {
        break;
    }

    case IRP_MJ_DEVICE_CONTROL:
    {
        // 获取应用层传值
        FileDataPtr = pirp->AssociatedIrp.SystemBuffer;
        switch (stack->Parameters.DeviceIoControl.IoControlCode)
        {
            // 读取内存函数
        case READ_FILE_SIZE_CODE:
        {
            LARGE_INTEGER liOffset = { 0 };
            UNICODE_STRING ustrFileSize;
            RtlInitUnicodeString(&ustrFileSize, L"\\??
\\C:\\Windows\\system32\\ntoskrnl.exe");

            // 获取文件长度
            ULONG64 ulBufferSize = MyGetFileSize(ustrFileSize);
            DbgPrint("获取文件大小: %I64d Bytes \n", ulBufferSize);

            // 将长度返回应用层
            FileDataPtr->size = ulBufferSize;
            break;
        }

        // 读取文件
        case READ_FILE_CODE:
        {
```

```
            FileData ptr;

            LARGE_INTEGER liOffset = { 0 };
            UNICODE_STRING ustrFileSize;
            RtlInitUnicodeString(&ustrFileSize, L"\\??
\\C:\\Windows\\system32\\ntoskrnl.exe");

            // 获取文件长度
            ULONG64 ulBufferSize = MyGetFileSize(ustrFileSize);
            DbgPrint("获取文件大小: %I64d Bytes \n", ulBufferSize);

            // 读取内存到缓冲区
            BYTE* pBuffer = ExAllocatePool(NonPagedPool, ulBufferSize);
            MyReadFile(ustrFileSize, liOffset, pBuffer, &ulBufferSize);

            // 返回数据
            FileDataPtr->size = ulBufferSize;
            RtlCopyMemory(FileDataPtr->data, pBuffer, FileDataPtr->size);

            break;
        }
        }

        pirp->IoStatus.Information = sizeof(FileDataPtr);
        break;
    }

    }

    pirp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(pirp, IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    if (driver->DeviceObject)
    {
        UNICODE_STRING SymbolName;
        RtlInitUnicodeString(&SymbolName, SYMBOLNAME);

        // 删除符号链接
        IoDeleteSymbolicLink(&SymbolName);
        IoDeleteDevice(driver->DeviceObject);
    }
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    NTSTATUS status = STATUS_SUCCESS;
    PDEVICE_OBJECT device = NULL;
    UNICODE_STRING DeviceName;

    DbgPrint("[LyShark] hello lyshark.com \n");

    // 初始化设备名
```

```
    RtlInitUnicodeString(&DeviceName, DEVICENAME);

    // 创建设备
    status = IoCreateDevice(Driver, sizeof(Driver->DriverExtension), &DeviceName,
FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN, FALSE, &device);
    if (status == STATUS_SUCCESS)
    {
        UNICODE_STRING SymbolName;
        RtlInitUnicodeString(&SymbolName, SYMBOLNAME);

        // 创建符号链接
        status = IoCreateSymbolicLink(&SymbolName, &DeviceName);

        // 失败则删除设备
        if (status != STATUS_SUCCESS)
        {
            IoDeleteDevice(device);
        }
    }

    // 派遣函数初始化
    Driver->MajorFunction[IRP_MJ_CREATE] = DriverIrpCtl;
    Driver->MajorFunction[IRP_MJ_CLOSE] = DriverIrpCtl;
    Driver->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DriverIrpCtl;

    // 卸载驱动
    Driver->DriverUnload = UnDriver;

    return STATUS_SUCCESS;
}
```

客户端完整代码如下所示;

```
// 署名权
// right to sign one's name on a piece of work
// PowerBy: LyShark
// Email: me@lyshark.com

#define _CRT_SECURE_NO_WARNINGS
#include <Windows.h>
#include <iostream>

#define READ_FILE_SIZE_CODE
CTL_CODE(FILE_DEVICE_UNKNOWN,0x800,METHOD_BUFFERED,FILE_ALL_ACCESS)
#define READ_FILE_CODE
CTL_CODE(FILE_DEVICE_UNKNOWN,0x801,METHOD_BUFFERED,FILE_ALL_ACCESS)

typedef struct
{
    DWORD size;      // 读写长度
    BYTE* data;      // 读写数据集
}FileData;

int main(int argc, char* argv[])
{
```

```
    // 连接到驱动
    HANDLE handle = CreateFileA("\\??\\ReadWriteSymbolName", GENERIC_READ |
GENERIC_WRITE, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    FileData data;
    DWORD dwSize = 0;

    // 首先得到文件长度
    DeviceIoControl(handle, READ_FILE_SIZE_CODE, 0, 0, &data, sizeof(data),
&dwSize, NULL);
    printf("%d \n", data.size);

    // 读取机器码到BYTE字节数组
    data.data = new BYTE[data.size];

    DeviceIoControl(handle, READ_FILE_CODE, &data, sizeof(data), &data,
sizeof(data), &dwSize, NULL);
    for (int i = 0; i < data.size; i++)
    {
        printf("0x%02X ", data.data[i]);
    }

    printf("\n");
    getchar();
    CloseHandle(handle);
    return 0;
}
```

通过驱动加载工具将`WinDDK.sys`拉起来，然后启动客户端进程，即可输出`ntoskrnl.exe`的文件数据，如下图所示;