在内核中，可以使用`ObRegisterCallbacks`这个内核回调函数来实现监控进程和线程对象操作。通过注册一个`OB_CALLBACK_REGISTRATION`回调结构体，可以指定所需的回调函数和回调的监控类型。这个回调结构体包含了回调函数和监控的对象类型，还有一个`Altitude`字段，用于指定回调函数的优先级。优先级越高的回调函数会先被调用，如果某个回调函数返回了一个非NULL值，后续的回调函数就不会被调用。

当有进程或线程对象创建、删除、复制或重命名时，内核会调用注册的回调函数。回调函数可以访问被监控对象的信息，如句柄、进程ID等，并可以采取相应的操作，如打印日志、记录信息等。

首先我们先来解释一下`OB_CALLBACK_REGISTRATION`结构体，`OB_CALLBACK_REGISTRATION`结构体是用于向内核注册回调函数的结构体，其中包含了回调函数和监控的对象类型等信息。

OB_CALLBACK_REGISTRATION结构体的定义：

```
typedef struct _OB_CALLBACK_REGISTRATION {
    PVOID RegistrationContext;
    POB_OPERATION_REGISTRATION OperationRegistration;
} OB_CALLBACK_REGISTRATION, *POB_CALLBACK_REGISTRATION;
```

其中，`RegistrationContext`是一个指针，指向一个可以在回调函数中访问的上下文数据结构，可以用来传递一些参数或状态信息。`OperationRegistration`是一个指向`OB_OPERATION_REGISTRATION`结构体的指针，该结构体指定了回调函数的相关信息，包括回调函数的地址、监控的对象类型、回调函数的优先级等。

OB_OPERATION_REGISTRATION结构体的定义如下：

```
typedef struct _OB_OPERATION_REGISTRATION {
    POBJECT_TYPE ObjectType;
    OB_OPERATION Operations;
    POB_PRE_OPERATION_CALLBACK PreOperation;
    POB_POST_OPERATION_CALLBACK PostOperation;
} OB_OPERATION_REGISTRATION, *POB_OPERATION_REGISTRATION;
```

其中，`ObjectType`是一个指针，指向要监控的对象类型的`OBJECT_TYPE`结构体。

Operations是一个枚举类型，表示要监控的操作类型，可以是如下之一：

- OB_OPERATION_HANDLE_CREATE：创建对象句柄
- OB_OPERATION_HANDLE_DUPLICATE：复制对象句柄
- OB_OPERATION_HANDLE_CLOSE：关闭对象句柄
- OB_OPERATION_HANDLE_WAIT：等待对象句柄
- OB_OPERATION_HANDLE_SET_INFORMATION：设置对象句柄信息
- OB_OPERATION_HANDLE_QUERY_INFORMATION：查询对象句柄信息
- OB_OPERATION_HANDLE_OPERATION：其他操作

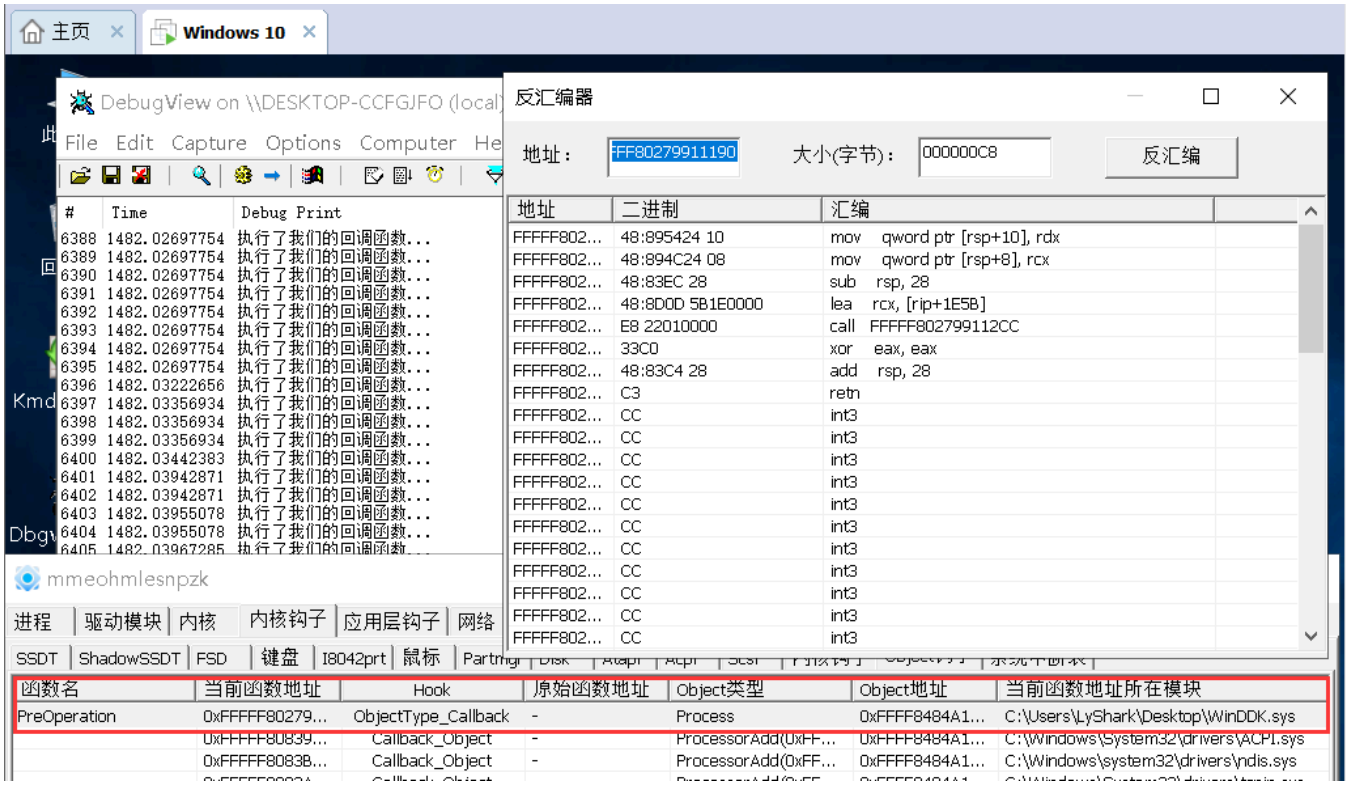
PreOperation和PostOperation分别是指向回调函数的指针，用于指定在进行指定操作之前和之后要执行的回调函数。这两个回调函数的参数和返回值等信息可以参考Microsoft官方文档。

我们以创建一个简单的监控进程对象为例，实现一个自己的进程回调函数`MyObjectCallBack()`当有新进程被加载时，自动路由到我们自己的回调中来；

首先在驱动程序入口处，定义`Base`结构，并初始化`Base.ObjectType`为`PsProcessType`标志着用于监控进程，接着填充`Base.Operations`并初始化为`OB_OPERATION_HANDLE_CREATE`表示当有进程句柄操作时触发回调，最后填充`Base.PreOperation`在回调前触发执行`MyObjectCallBack`自己的回调，当结构体被填充好以后，直接调用`ObRegisterCallbacks()`向内核申请回调事件即可，这段代码实现如下所示；

```
#include <ntddk.h>
#include <ntstrsafe.h>

PVOID Globle_Object_Handle;

OB_PREOP_CALLBACK_STATUS MyObjectCallBack(PVOID RegistrationContext,
POB_PRE_OPERATION_INFORMATION OperationInformation)
{
    DbgPrint("执行了我们的回调函数...");
    return STATUS_SUCCESS;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    ObUnRegisterCallbacks(Globle_Object_Handle);
    DbgPrint("回调卸载完成...");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    OB_OPERATION_REGISTRATION Base;                          // 回调函数结构体(你所填的结构都在这
里)
    OB_CALLBACK_REGISTRATION CallbackReg;

    CallbackReg.RegistrationContext = NULL;                  // 注册上下文(你回调函数返回参数)
    CallbackReg.Version = OB_FLT_REGISTRATION_VERSION;       // 注册回调版本
    CallbackReg.OperationRegistration = &Base;
    CallbackReg.OperationRegistrationCount = 1;               // 操作计数(下钩数量)
    RtlUnicodeStringInit(&CallbackReg.Altitude, L"600000");   // 长度

  Base.ObjectType = PsProcessType;                          // 进程操作类型.此处为进程操作
    Base.Operations = OB_OPERATION_HANDLE_CREATE;             // 操作句柄创建
    Base.PreOperation = MyObjectCallBack;                     // 你自己的回调函数
    Base.PostOperation = NULL;

  // 注册回调
    if (ObRegisterCallbacks(&CallbackReg, &Globle_Object_Handle))
  {
        DbgPrint("回调注册成功...");
  }

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

上方代码运行后，我们可以打开`Xuetr`扫描一下内核`Object`钩子，可以看到已经成功挂钩了，输出效果如下图所示；



监控进程打开与关闭

接下来我们实现一个简单的需求，通过编写一个自定义 `MyObjectCallBack` 回调函数实现保护 `win32calc.exe` 进程不被关闭，本功能实现的关键在于如何获取到监控进程的进程名 `GetProcessImageNameByProcessID` 函数就是用来实现转换的，通过向此函数内传入一个进程PID则会通过 `PsGetProcessImageFileName` 输出该进程的进程名。

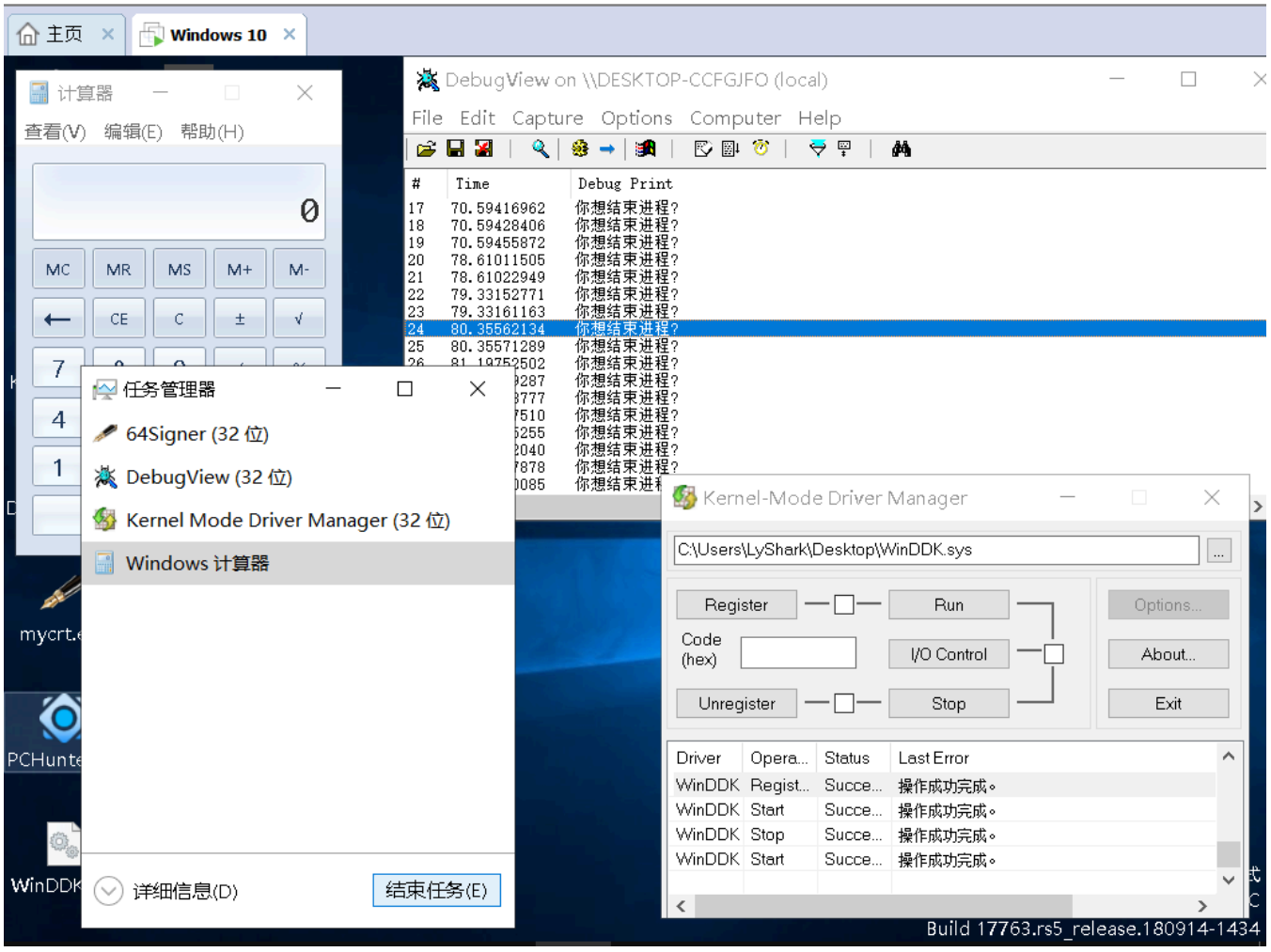
而当回调函数内接收到此进程名时，则可以通过 `strstr(ProcName, "win32calc.exe")` 对进程名进行判断，如果匹配到结果，则直接通过 `Operation->Parameters->CreateHandleInformation.DesiredAccess = ~THREAD_TERMINATE` 去掉 `TERMINATE_PROCESS` 或 `TERMINATE_THREAD` 权限即可，此时的进程将会被保护而无法被关闭，这段代码实现如下所示；

```
#include <ntddk.h>
#include <wdm.h>
#include <ntstrsafe.h>
#define PROCESS_TERMINATE 1

PVOID Globle_Object_Handle;
NTKERNELAPI UCHAR * PsGetProcessImageFileName(__in PEPROCESS Process);

char* GetProcessImageNameByProcessID(ULONG ulProcessID)
{
    NTSTATUS  Status;
    PEPROCESS  EProcess = NULL;
    Status = PsLookupProcessByProcessId((HANDLE)ulProcessID, &EProcess);
    if (!NT_SUCCESS(Status))
        return FALSE;
    ObDereferenceObject(EProcess);
    return (char*)PsGetProcessImageFileName(EProcess);
}
```

```
OB_PREOP_CALLBACK_STATUS MyObjectCallBack(PVOID RegistrationContext,
POB_PRE_OPERATION_INFORMATION Operation)
{
    char ProcName[256] = { 0 };
    HANDLE pid = PsGetProcessId((PEPROCESS)Operation->Object);           // 取出当前调用函数的
PID
    strcpy(ProcName, GetProcessImageNameByProcessID((ULONG)pid));        // 通过PID取出进程名,
然后直接拷贝内存
    //DbgPrint("当前进程的名字是：%s", ProcName);

    if (strstr(ProcName, "win32calc.exe"))
    {
        if (Operation->Operation == OB_OPERATION_HANDLE_CREATE)
        {
            if ((Operation->Parameters->CreateHandleInformation.OriginalDesiredAccess &
PROCESS_TERMINATE) == PROCESS_TERMINATE)
            {
                DbgPrint("你想结束进程?");

                // 如果是计算器，则去掉它的结束权限，在Win10上无效
                Operation->Parameters->CreateHandleInformation.DesiredAccess =
~THREAD_TERMINATE;
                return STATUS_UNSUCCESSFUL;
            }
        }
    }
    return STATUS_SUCCESS;
}
VOID UnDriver(PDRIVER_OBJECT driver)
{
    ObUnRegisterCallbacks(Globle_Object_Handle);
    DbgPrint("回调卸载完成...");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    NTSTATUS obst = 0;
    OB_CALLBACK_REGISTRATION obReg;
    OB_OPERATION_REGISTRATION opReg;

    memset(&obReg, 0, sizeof(obReg));
    obReg.Version = ObGetFilterVersion();
    obReg.OperationRegistrationCount = 1;
    obReg.RegistrationContext = NULL;
    RtlInitUnicodeString(&obReg.Altitude, L"321125");
    obReg.OperationRegistration = &opReg;

    memset(&opReg, 0, sizeof(opReg));
    opReg.ObjectType = PsProcessType;
    opReg.Operations = OB_OPERATION_HANDLE_CREATE | OB_OPERATION_HANDLE_DUPLICATE;
    opReg.PreOperation = (POB_PRE_OPERATION_CALLBACK)&MyObjectCallBack;
    obst = ObRegisterCallbacks(&obReg, &Globle_Object_Handle);
```

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

首先运行计算器，然后启动驱动保护，此时我们在任务管理器中就无法结束计算器进程了。



监控进程中的模块加载

系统中的模块加载包括用户层模块DLL和内核模块SYS的加载，在内核环境下我们可以调用 `PsSetLoadImageNotifyRoutine` 内核函数来设置一个映像加载通告例程，当有驱动或者DLL被加载时，回调函数就会被调用从而执行我们自己的回调例程。

PsSetLoadImageNotifyRoutine 函数用来设置一个映像加载通告例程。该函数需要传入一个回调函数的指针，该回调函数会在系统中有驱动程序或 DLL 被加载时被调用。

该函数函数的原型为：

```
NTKERNELAPI
NTSTATUS
PsSetLoadImageNotifyRoutine(
  PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine
);
```

其中，NotifyRoutine 参数是一个指向映像加载通告例程的函数指针。该函数将在系统中有驱动程序或 DLL 被加载时被调用。

当一个映像被加载时，Windows 内核会检查是否已注册了映像加载通告例程。如果已注册，则内核会调用该例程，将被加载的模块的信息作为参数传递给该例程。通常，该例程会记录或处理这些信息。

需要注意的是，映像加载通告例程应该尽可能地简短，不要执行复杂的操作，以避免影响系统性能。同时，该例程应该是线程安全的，以免发生竞态条件或死锁。

如上函数中`PLOAD_IMAGE_NOTIFY_ROUTINE`用于接收一个自定义函数，该自定义函数需要声明成如下原型；

```
VOID MyLoadImageNotifyRoutine(
  PUNICODE_STRING FullImageName,
  HANDLE ProcessId,
  PIMAGE_INFO ImageInfo
);
```

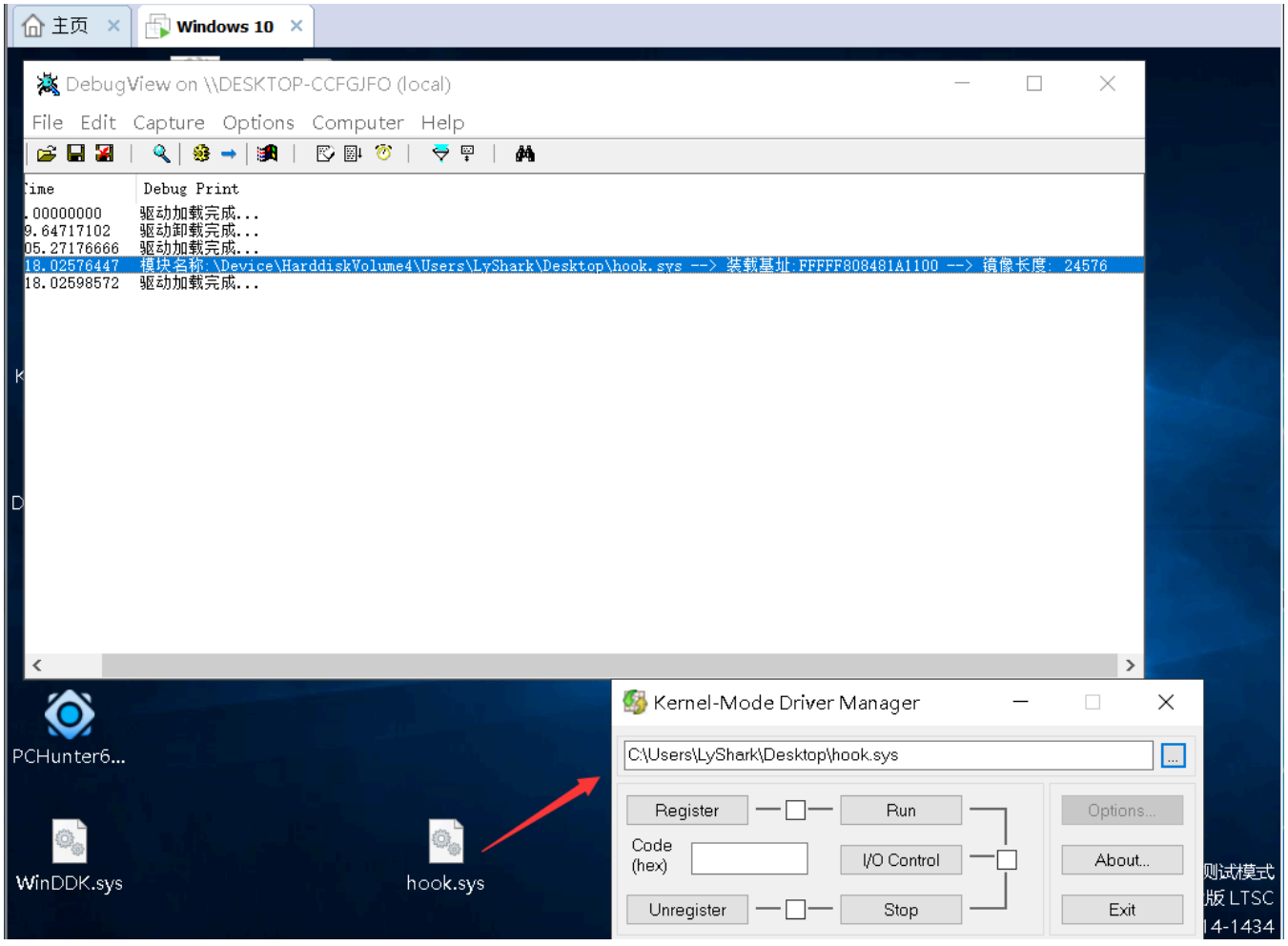
- 参数`FullImageName`参数是指被加载的模块的完整路径名；
- 参数`ProcessId`参数是指加载该模块的进程ID；
- 参数`ImageInfo`参数是指与该模块相关的信息，包括其基地址、大小等。在回调函数中，可以对这些信息进行处理，以实现对模块加载的监控。

有了如上知识体系，实现监控的目的就会变得简单，其监控的实现重点是实现自己的`MyLoadImageNotifyRoutine`如下代码中简单实现了当有新的DLL被装载到内存是，则通过`DbgView`输出该模块的具体信息；

```
#include <ntddk.h>
#include <ntimage.h>

// 用与获取特定基地址的模块入口
PVOID GetDriverEntryByImageBase(PVOID ImageBase)
{
    PIMAGE_DOS_HEADER pDOSHeader;
    PIMAGE_NT_HEADERS64 pNTHeader;
    PVOID pEntryPoint;
    pDOSHeader = (PIMAGE_DOS_HEADER)ImageBase;
    pNTHeader = (PIMAGE_NT_HEADERS64)((ULONG64)ImageBase + pDOSHeader->e_lfanew);
    pEntryPoint = (PVOID)((ULONG64)ImageBase + pNTHeader-
>OptionalHeader.AddressOfEntryPoint);
    return pEntryPoint;
}

VOID MyLoadImageNotifyRoutine(PUNICODE_STRING FullImageName,HANDLE ProcessId,PIMAGE_INFO
ImageInfo)
{
    PVOID pDrvEntry;

  // MmIsAddress 验证地址可用性
    if (FullImageName != NULL && MmIsAddressValid(FullImageName))
    {
        if (ProcessId == 0)
        {
            pDrvEntry = GetDriverEntryByImageBase(ImageInfo->ImageBase);
            DbgPrint("模块名称:%wZ --> 装载基址:%p --> 镜像长度: %d", FullImageName,
pDrvEntry,ImageInfo->ImageSize);
        }
```

```
    }
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    PsRemoveLoadImageNotifyRoutine((PLOAD_IMAGE_NOTIFY_ROUTINE)MyLoadImageNotifyRoutine);
    DbgPrint("驱动卸载完成...");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    PsSetLoadImageNotifyRoutine((PLOAD_IMAGE_NOTIFY_ROUTINE)MyLoadImageNotifyRoutine);
    DbgPrint("驱动加载完成...");
    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

输出效果图如下所示：

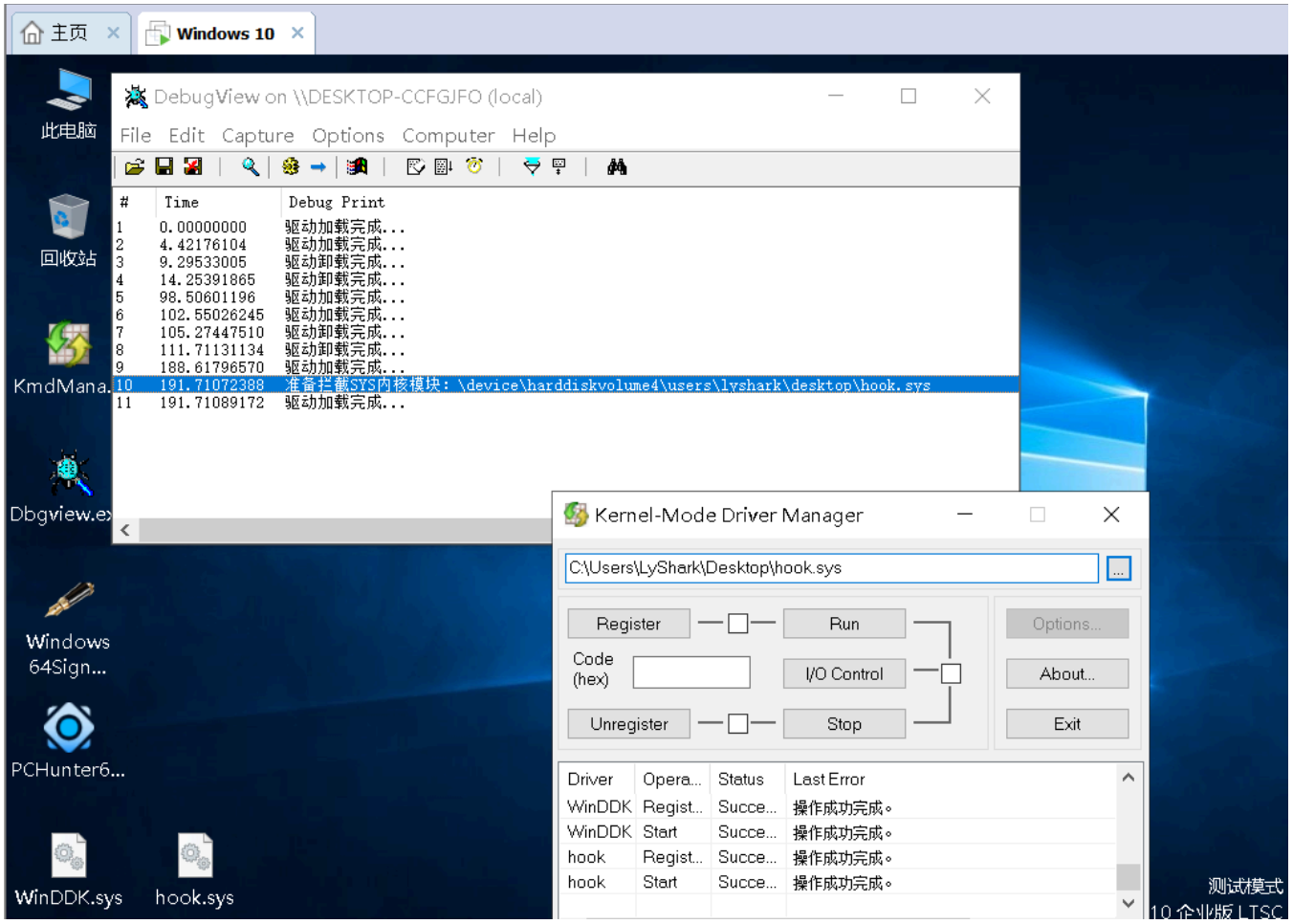


通过上方代码的学习，相信读者已经学会了如何监视系统内加载驱动与DLL功能，接着我们给上方的代码加上判断，只需在上方代码的基础上进行一定的改进，但需要注意 `MyLoadImageNotifyRoutine` 回调函数中的 `ModuleStyle` 参数，该参数用于判断加载模块的类型，如果返回值为零则表示加载SYS，如果返回非零则表示加载的是DLL模块。

```
VOID UnicodeToChar(PUNICODE_STRING dst, char *src)
{
    ANSI_STRING string;
```

```
    RtlUnicodeStringToAnsiString(&string, dst, TRUE);
    strcpy(src, string.Buffer);
    RtlFreeAnsiString(&string);
}

VOID MyLoadImageNotifyRoutine(PUNICODE_STRING FullImageName,HANDLE ModuleStyle,PIMAGE_INFO
ImageInfo)
{
    PVOID pDrvEntry;
    char szFullImageName[256] = { 0 };

  // MmIsAddress 验证地址可用性
    if (FullImageName != NULL && MmIsAddressValid(FullImageName))
    {
    // ModuleStyle为零表示加载sys非零表示加载DLL
        if (ModuleStyle == 0)
        {
            pDrvEntry = GetDriverEntryByImageBase(ImageInfo->ImageBase);
            UnicodeToChar(FullImageName, szFullImageName);
            if (strstr(_strlwr(szFullImageName), "hook.sys"))
            {
                DbgPrint("准备拦截SYS内核模块：%s", _strlwr(szFullImageName));
            }
        }
    }
}
```

输出效果图如下所示：



如上方代码中所示，`MyLoadImageNotifyRoutine`函数只能接收全局模块加载动作，但是此模式无法判断到底是哪个进程加载的`hook.sys`驱动，因为回调函数本就很底层了，到了一定的深度之后就无法判断到底是谁主动引发的行为，一切回调行为都会变成系统的行为，而某些驱动过滤软件，通常会使用特征扫描等方式来判断驱动是否是我们所需；

当判断特定模块是我们所要拦截的驱动时，则下一步就要进行驱动的屏蔽工作，对于驱动屏蔽来说最直接的办法就是在程序的入口位置写入`Mov eax,c0000022h;ret`这两条汇编指令从而让模块无法被执行，此时模块虽然被加载了但却无法执行功能，本质上来说已经起到了拒绝加载的效果；

通过`ImageInfo->ImageBase`来获取被加载驱动程序`hook.sys`的映像基址，然后找到NT头的`OptionalHeader`节点，该节点里面就是被加载驱动入口的地址，通过汇编在驱动头部写入`ret`返回指令，即可实现屏蔽加载特定驱动文件，这段代码完整实现如下所示；

```
#include <ntddk.h>
#include <intrin.h>
#include <ntimage.h>

PVOID GetDriverEntryByImageBase(PVOID ImageBase)
{
    PIMAGE_DOS_HEADER pDOSHeader;
    PIMAGE_NT_HEADERS64 pNTHeader;
    PVOID pEntryPoint;
    pDOSHeader = (PIMAGE_DOS_HEADER)ImageBase;
    pNTHeader = (PIMAGE_NT_HEADERS64)((ULONG64)ImageBase + pDOSHeader->e_lfanew);
    pEntryPoint = (PVOID)((ULONG64)ImageBase + pNTHeader-
>OptionalHeader.AddressOfEntryPoint);
```

```
    return pEntryPoint;
}

VOID UnicodeToChar(PUNICODE_STRING dst, char *src)
{
    ANSI_STRING string;
    RtlUnicodeStringToAnsiString(&string, dst, TRUE);
    strcpy(src, string.Buffer);
    RtlFreeAnsiString(&string);
}

// 使用开关写保护需要在C/C++优化中启用内部函数

// 关闭写保护
KIRQL WPOFFx64()
{
    KIRQL irql = KeRaiseIrqlToDpcLevel();
    UINT64 cr0 = __readcr0();
    cr0 &= 0xfffffffffffeffff;
    _disable();
    __writecr0(cr0);
    return irql;
}

// 开启写保护
void WPONx64(KIRQL irql)
{
    UINT64 cr0 = __readcr0();
    cr0 |= 0x10000;
    _enable();
    __writecr0(cr0);
    KeLowerIrql(irql);
}

BOOLEAN DenyLoadDriver(PVOID DriverEntry)
{
    UCHAR fuck[] = "\xB8\x22\x00\x00\xC0\xC3";
    KIRQL kirql;
    /* 在模块开头写入以下汇编指令
    Mov eax,c0000022h
    ret
    */

    if (DriverEntry == NULL)
  {
    return FALSE;
  }

    kirql = WPOFFx64();
    memcpy(DriverEntry, fuck,sizeof(fuck) / sizeof(fuck[0]));
    WPONx64(kirql);
    return TRUE;
}
```

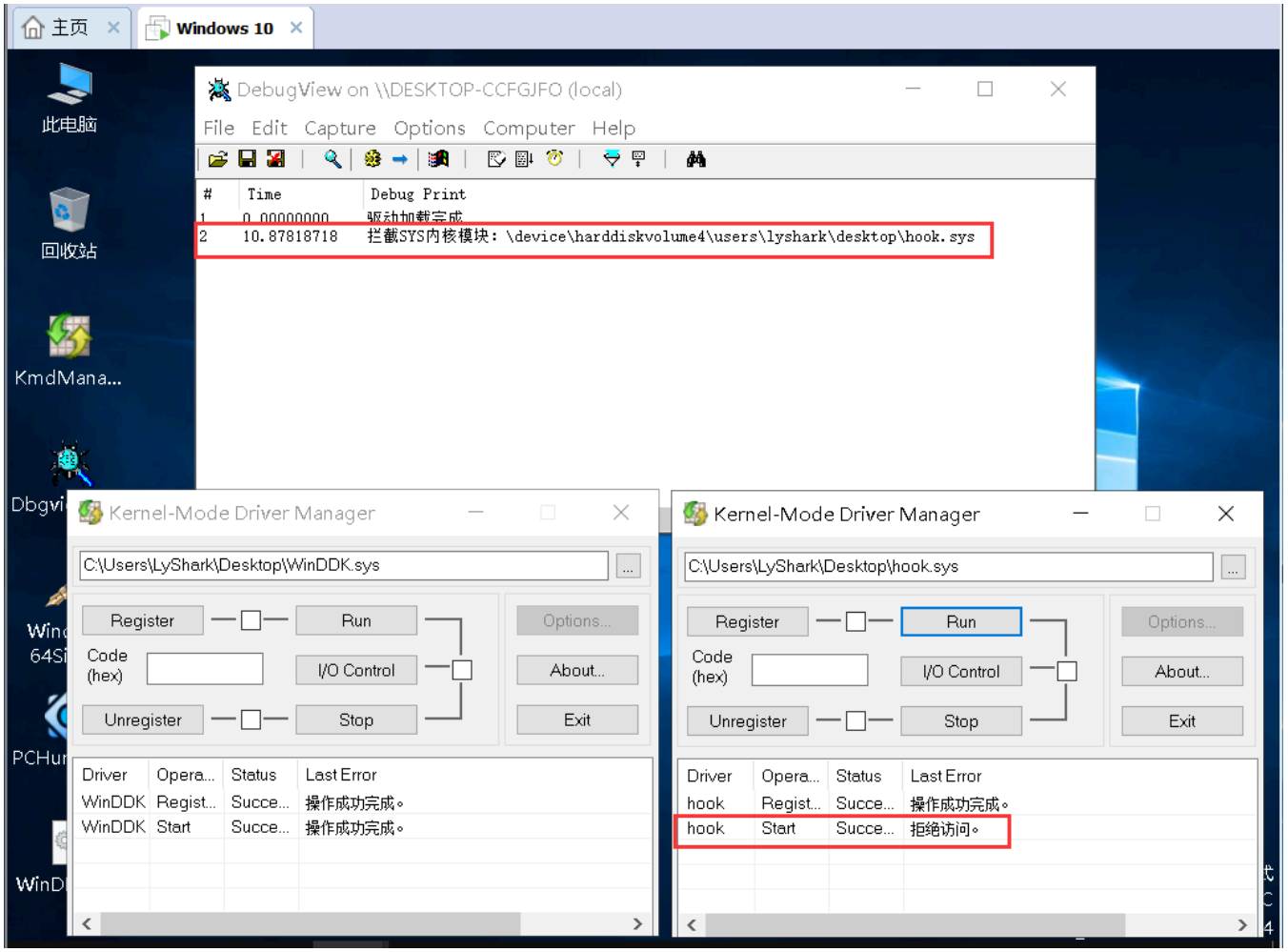
```
VOID MyLoadImageNotifyRoutine(PUNICODE_STRING FullImageName, HANDLE ModuleStyle, PIMAGE_INFO
ImageInfo)
{
    PVOID pDrvEntry;
    char szFullImageName[256] = { 0 };

  // MmIsAddress 验证地址可用性
    if (FullImageName != NULL && MmIsAddressValid(FullImageName))
    {
    // ModuleStyle为零表示加载sys非零表示加载DLL
        if (ModuleStyle == 0)
        {
            pDrvEntry = GetDriverEntryByImageBase(ImageInfo->ImageBase);
            UnicodeToChar(FullImageName, szFullImageName);
            if (strstr(_strlwr(szFullImageName), "hook.sys"))
            {
                DbgPrint("拦截SYS内核模块: %s", szFullImageName);
                DenyLoadDriver(pDrvEntry);
            }
        }
    }
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    PsRemoveLoadImageNotifyRoutine((PLOAD_IMAGE_NOTIFY_ROUTINE)MyLoadImageNotifyRoutine);
    DbgPrint("驱动卸载完成...");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    PsSetLoadImageNotifyRoutine((PLOAD_IMAGE_NOTIFY_ROUTINE)MyLoadImageNotifyRoutine);
    DbgPrint("驱动加载完成...");
    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

而对于屏蔽DLL模块加载同样如此，仅仅只是在判断`ModuleStyle`参数时将非零作为过滤条件即可，要实现该功能只需要在上面的代码上稍微修改一下即可；

```
char *UnicodeToLongString(PUNICODE_STRING uString)
{
    ANSI_STRING asStr;
    char *Buffer = NULL;;
    RtlUnicodeStringToAnsiString(&asStr, uString, TRUE);
    Buffer = ExAllocatePoolWithTag(NonPagedPool, uString->MaximumLength * sizeof(wchar_t),
0);
    if (Buffer == NULL)
  {
        return NULL;
  }
```

```
    RtlCopyMemory(Buffer, asStr.Buffer, asStr.Length);
    return Buffer;
}

VOID MyLoadImageNotifyRoutine(PUNICODE_STRING FullImageName, HANDLE ModuleStyle, PIMAGE_INFO
ImageInfo)
{
    PVOID pDrvEntry;
    char *PareString = NULL;

    if (MmIsAddressValid(FullImageName))
    {
    // 非零则监控DLL加载
        if (ModuleStyle != 0)
        {
            PareString = UnicodeToLongString(FullImageName);
            if (PareString != NULL)
            {
                if (strstr(PareString, "hook.dll"))
                {
                    pDrvEntry = GetDriverEntryByImageBase(ImageInfo->ImageBase);
                    if (pDrvEntry != NULL)
                        DenyLoadDriver(pDrvEntry);
                }
            }
        }
    }
}
```

我们以屏蔽SYS内核模块为例，当驱动文件`WinDDK.sys`被加载后，尝试加载`hook.sys`会提示拒绝访问，说明我们的驱动保护生效了；



关键的内核进程操作已经分享，该功能作用使用非常广泛，例如杀软的主动防御系统，游戏的保护系统等都会用到这些东西，以QQ电脑管家为例，默认进程在运行后都会挂钩进程保护钩子用于监控系统内进程与线程操作；