本篇将深入讨论驱动开发内核驱动读写，主要涵盖内核开发中的一些关键技术。这些技术包括内核远程堆分配和销毁、CR3切换的实现、MDL的读写内存、R3和R0之间的通信、进程汇编和反汇编实现、四级页表的解析、浮点数的读写、多级偏移的读写、以及物理页的读写技术。同时，本文还会介绍这些技术在驱动保护方面的应用。

在本文中，我们将详细探讨每个主题，包括其实现方法、技术细节和关键步骤。我们将通过实例演示和代码示例，帮助读者更好地理解这些技术的实现方法。本文旨在为读者提供一个全面而深入的内核开发指南，帮助读者掌握内核驱动读写方面的核心技术，以及如何在驱动保护方面应用这些技术。

总之，本文将提供内核驱动读写方面的深入介绍和实例演示，帮助读者更好地掌握这些关键技术，并将重点介绍这些技术在驱动保护方面的应用。希望本文可以为读者提供有价值的内核开发知识，帮助读者成为一名出色的内核安全类程序员。

在开始学习内核内存读写篇之前，我们先来实现一个简单的内存分配销毁堆的功能，在内核空间内用户依然可以动态的申请与销毁一段可控的堆空间，一般而言内核中提供了`ZwAllocateVirtualMemory`这个函数用于专门分配虚拟空间，而与之相对应的则是`ZwFreeVirtualMemory`此函数则用于销毁堆内存，当我们需要分配内核空间时往往需要切换到对端进程栈上再进行操作，接下来`LyShark`将从API开始介绍如何运用这两个函数实现内存分配与使用，并以此来作为驱动读写篇的入门知识。

内存分配堆

首先以内存分配为例`ZwAllocateVirtualMemory()`函数，该系列函数在`ntifs.h`头文件内，且如果需要使用则最好提前在程序头部进行声明，该函数的微软官方定义如下所示；

```
NTSYSAPI NTSTATUS ZwAllocateVirtualMemory(
  [in]      HANDLE    ProcessHandle,          // 进程句柄
  [in, out] PVOID     *BaseAddress,           // 指向将接收已分配页面区域基址的变量的指针
  [in]      ULONG_PTR ZeroBits,               // 节视图基址中必须为零的高顺序地址位数
  [in, out] PSIZE_T   RegionSize,             // 指向将接收已分配页面区域的实际大小
  [in]      ULONG     AllocationType,         // 包含指定要执行的分配类型的标志的位掩码
  [in]      ULONG     Protect                 // 包含页面保护标志的位掩码
);
```

参数`ProcessHandle`用于传入一个进程句柄此处我们可以通过`NtCurrentProcess()`获取到当前自身进程的句柄。

参数`BaseAddress`则用于接收分配堆地址的首地址，此处指向将接收已分配页面区域基址的变量的指针。

参数`RegionSize`则用于指定需要分配的内存空间大小，此参数的初始值指定区域的大小（以字节为单位）并向上舍入到下一个主机页大小边界。

参数`AllocationType`用于指定分配内存的属性，通常我们会用到的只有两种`MEM_COMMIT`指定为提交类型，`MEM_PHYSICAL`则用于分配物理内存，此标志仅用于地址窗口扩展AWE内存。 如果设置了`MEM_PHYSICAL`则还必须设置`MEM_RESERVE`不能设置其他标志，并且必须将保护设置为`PAGE_READWRITE`。

参数`Protect`用于设置当前分批堆的保护属性，通常当我们需要分配一段可执行指令的内存空间时会使用`PAGE_EXECUTE_READWRITE`，如果无执行需求则推荐使用`PAGE_READWRITE`属性。

在对特定进程分配堆时第一步就是要切入到该进程的进程栈中，此时可通过`KeStackAttachProcess()`切换到进程栈，于此对应的是`KeUnstackDetachProcess()`脱离进程栈，这两个函数的具体定义如下；

```
// 附加到进程栈
void KeStackAttachProcess(
        PRKPROCESS    PROCESS,      // 传入EProcess结构
  [out] PRKAPC_STATE ApcState      // 指向KAPC_STATE结构的不透明指针
);
// 接触附加
void KeUnstackDetachProcess(
  [in] PRKAPC_STATE ApcState       // 指向KAPC_STATE结构的不透明指针
);
```

此处如果需要附加进程栈则必须提供该进程的`PRKPROCESS`也就是`EProcess`结构，此结构可通过`PsLookupProcessByProcessId()`获取到，该函数接收一个进程PID并将此PID转为`EProcess`结构，函数定义如下;

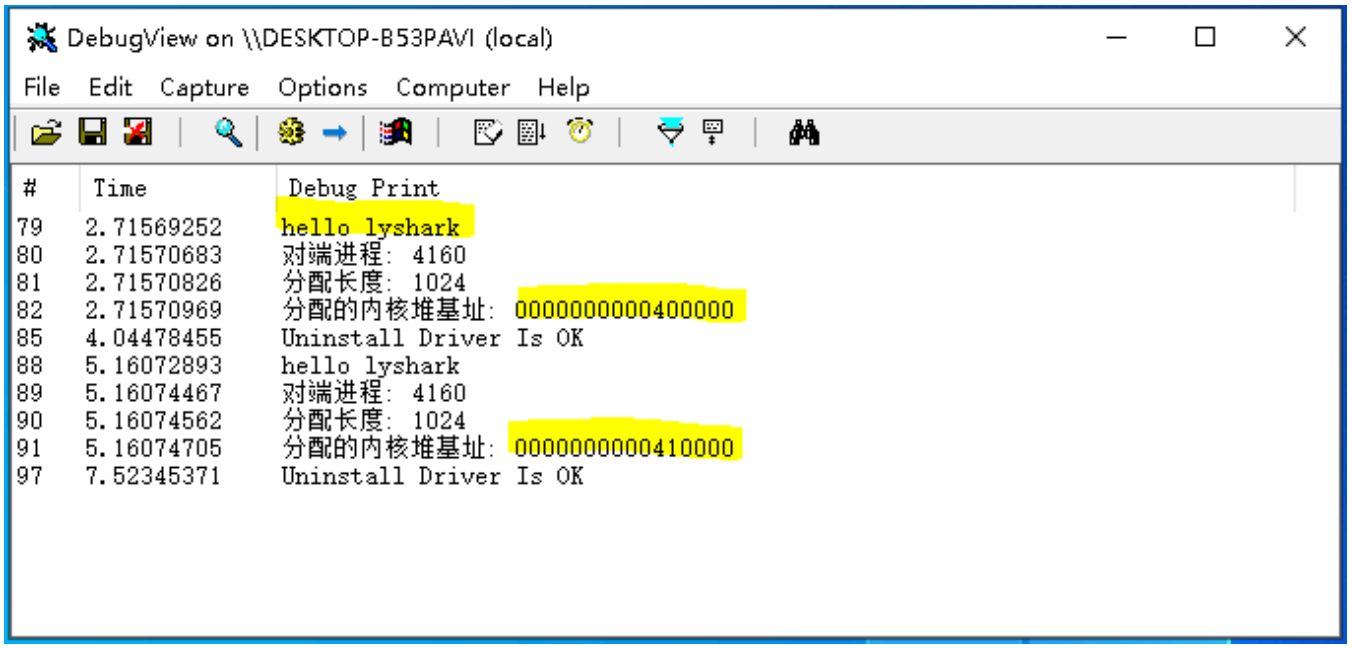
```
NTSTATUS PsLookupProcessByProcessId(
  [in]  HANDLE    ProcessId,          // 进程PID
  [out] PEPROCESS *Process            // 输出EP结构
);
```

基本的函数介绍完了，那么这段代码应该不难理解了，如下代码中需要注意一点，参数`OUT PVOID Buffer`用于输出堆地址而不是输入地址。

```
#include <ntifs.h>
#include <windef.h>

// 定义声明
NTSTATUS ZwAllocateVirtualMemory(
    __in HANDLE  ProcessHandle,
    __inout PVOID  *BaseAddress,
    __in ULONG_PTR  ZeroBits,
    __inout PSIZE_T  RegionSize,
    __in ULONG  AllocationType,
    __in ULONG  Protect
);

// 分配内存空间
NTSTATUS AllocMemory(IN ULONG ProcessPid, IN SIZE_T Length, OUT PVOID Buffer)
{
    NTSTATUS Status = STATUS_SUCCESS;
    PEPROCESS pEProcess = NULL;
    KAPC_STATE ApcState = { 0 };
    PVOID BaseAddress = NULL;

    // 通过进程PID得到进程EProcess
    Status = PsLookupProcessByProcessId((HANDLE)ProcessPid, &pEProcess);
    if (!NT_SUCCESS(Status) && !MmIsAddressValid(pEProcess))
    {
        return STATUS_UNSUCCESSFUL;
    }

    // 验证内存可读
```

```
    if (!MmIsAddressValid(pEProcess))
    {
        return STATUS_UNSUCCESSFUL;
    }
    __try
    {
        // 附加到进程栈
        KeStackAttachProcess(pEProcess, &ApcState);

        // 分配内存
        Status = ZwAllocateVirtualMemory(NtCurrentProcess(), &BaseAddress, 0, &Length,
MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        RtlZeroMemory(BaseAddress, Length);

        // 返回内存地址
        *(PVOID*)Buffer = BaseAddress;

        // 脱离进程栈
        KeUnstackDetachProcess(&ApcState);
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        KeUnstackDetachProcess(&ApcState);
        Status = STATUS_UNSUCCESSFUL;
    }

    ObDereferenceObject(pEProcess);
    return Status;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint(("Uninstall Driver Is OK \n"));
}

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

    DWORD process_id = 4160;
    DWORD create_size = 1024;
    DWORD64 ref_address = 0;

    NTSTATUS Status = AllocMemory(process_id, create_size, &ref_address);

    DbgPrint("对端进程: %d \n", process_id);
    DbgPrint("分配长度: %d \n", create_size);
    DbgPrint("分配的内核堆基址: %p \n", ref_address);

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

运行如上代码片段，则会在进程 PID=4160 中开辟一段堆空间，输出效果如下；



内存销毁堆

与创建堆相对 ZwFreeVirtualMemory() 则用于销毁一个堆，其微软定义如下所示；

```
NTSYSAPI NTSTATUS ZwFreeVirtualMemory(
  [in]      HANDLE  ProcessHandle,    // 进程句柄
  [in, out] PVOID   *BaseAddress,     // 堆基址
  [in, out] PSIZE_T RegionSize,       // 销毁长度
  [in]      ULONG   FreeType          // 释放类型
);
```

相对于创建来说，销毁堆则必须传入堆空间 BaseAddress 基址，以及堆空间的 RegionSize 长度，需要格外注意 FreeType 参数，这是一个位掩码，其中包含描述 ZwFreeVirtualMemory 将为页面指定区域执行的任意操作类型。如果传入 MEM_DECOMMIT 则将取消提交页面的指定区域，页面进入保留状态。而如果设置为 MEM_RELEASE 则堆地址将被立即释放。

销毁堆空间 FreeMemory() 的完整代码如下所示，销毁是我们使用 MEM_RELEASE 参数即立即销毁。

```
#include <ntifs.h>
#include <windef.h>

// 申请堆
NTSTATUS ZwAllocateVirtualMemory(
    __in HANDLE  ProcessHandle,
    __inout PVOID  *BaseAddress,
    __in ULONG_PTR  ZeroBits,
    __inout PSIZE_T  RegionSize,
    __in ULONG  AllocationType,
    __in ULONG  Protect
);

// 销毁堆
```

```
NTSYSAPI NTSTATUS ZwFreeVirtualMemory(
    __in      HANDLE  ProcessHandle,
    __inout PVOID   *BaseAddress,
    __inout PSIZE_T RegionSize,
    __in      ULONG   FreeType
);

// 分配内存空间
NTSTATUS AllocMemory(IN ULONG ProcessPid, IN SIZE_T Length, OUT PVOID Buffer)
{
    NTSTATUS Status = STATUS_SUCCESS;
    PEPROCESS pEProcess = NULL;
    KAPC_STATE ApcState = { 0 };
    PVOID BaseAddress = NULL;

    // 通过进程PID得到进程EProcess
    Status = PsLookupProcessByProcessId((HANDLE)ProcessPid, &pEProcess);
    if (!NT_SUCCESS(Status) && !MmIsAddressValid(pEProcess))
    {
        return STATUS_UNSUCCESSFUL;
    }

    // 验证内存可读
    if (!MmIsAddressValid(pEProcess))
    {
        return STATUS_UNSUCCESSFUL;
    }
    __try
    {
        // 附加到进程栈
        KeStackAttachProcess(pEProcess, &ApcState);

        // 分配内存
        Status = ZwAllocateVirtualMemory(NtCurrentProcess(), &BaseAddress, 0, &Length,
MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        RtlZeroMemory(BaseAddress, Length);

        // 返回内存地址
        *(PVOID*)Buffer = BaseAddress;

        // 脱离进程栈
        KeUnstackDetachProcess(&ApcState);
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        KeUnstackDetachProcess(&ApcState);
        Status = STATUS_UNSUCCESSFUL;
    }

    ObDereferenceObject(pEProcess);
    return Status;
}
```

```
// 注销内存空间
NTSTATUS FreeMemory(IN ULONG ProcessPid, IN SIZE_T Length, IN PVOID BaseAddress)
{
    NTSTATUS Status = STATUS_SUCCESS;
    PEPROCESS pEProcess = NULL;
    KAPC_STATE ApcState = { 0 };

    Status = PsLookupProcessByProcessId((HANDLE)ProcessPid, &pEProcess);
    if (!NT_SUCCESS(Status) && !MmIsAddressValid(pEProcess))
    {
    return STATUS_UNSUCCESSFUL;
    }

    if (!MmIsAddressValid(pEProcess))
    {
        return STATUS_UNSUCCESSFUL;
    }

    __try
    {
        // 附加到进程栈
        KeStackAttachProcess(pEProcess, &ApcState);

        // 释放内存
        Status = ZwFreeVirtualMemory(NtCurrentProcess(), &BaseAddress, &Length,
MEM_RELEASE);

        // 脱离进程栈
        KeUnstackDetachProcess(&ApcState);
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        KeUnstackDetachProcess(&ApcState);
        Status = STATUS_UNSUCCESSFUL;
    }

    ObDereferenceObject(pEProcess);
    return Status;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint(("Uninstall Driver Is OK \n"));
}

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

    DWORD process_id = 4160;
    DWORD create_size = 1024;
    DWORD64 ref_address = 0;
```

```
    NTSTATUS Status = AllocMemory(process_id, create_size, &ref_address);

    DbgPrint("对端进程: %d \n", process_id);
    DbgPrint("分配长度: %d \n", create_size);
    DbgPrint("分配的内核堆基址: %p \n", ref_address);

    Status = FreeMemory(process_id, create_size, ref_address);
    DbgPrint("销毁堆地址: %p \n", ref_address);

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

编译并运行如上这段代码，代码会首先调用`AllocMemory()`函数实现分配堆，然后调用`FreeMemory()`函数销毁堆，并输出销毁地址，如下图所示；