CR3是一种控制寄存器，它是CPU中的一个专用寄存器，用于存储当前进程的页目录表的物理地址。在x86体系结构中，虚拟地址的翻译过程需要借助页表来完成。页表是由页目录表和页表组成的，页目录表存储了页表的物理地址，而页表存储了实际的物理页框地址。因此，页目录表的物理地址是虚拟地址翻译的关键之一。

在操作系统中，每个进程都有自己的地址空间，地址空间中包含了进程的代码、数据和堆栈等信息。为了实现进程间的隔离和保护，操作系统会为每个进程分配独立的地址空间。在这个过程中，操作系统会将每个进程的页目录表的物理地址存储在它自己的CR3寄存器中。当进程切换时，操作系统会修改CR3寄存器的值，从而让CPU使用新的页目录表来完成虚拟地址的翻译。

利用CR3寄存器可以实现强制读写特定进程的内存地址，这种操作需要一定的权限和技术知识。在实际应用中，这种操作主要用于调试和漏洞挖掘等方面。同时，由于CR3寄存器的读写属于有痕读写，因此许多驱动保护都会禁止或者修改CR3寄存器的值，以提高系统的安全性，此时CR3读写就失效了，当然如果能找到CR3的正确地址，此方式也是靠谱的一种读写机制。

在读写进程的时候，我们需要先找到目标进程的`PEPROCESS`结构。PEPROCESS结构是`Windows`操作系统中描述进程的一个重要数据结构，它包含了进程的各种属性和状态信息，如进程ID、进程优先级、内存布局等等。对于想要读写目标进程的内存，我们需要获得目标进程的PEPROCESS结构，才能进一步访问和操作进程的内存。

每个进程都有一个唯一的进程`ID（PID）`，我们可以通过遍历系统中所有进程的方式来查找目标进程。对于每个进程，我们可以通过读取进程对象的名称来判断它是否是目标进程。如果找到了目标进程，我们就可以从其进程对象中获得指向`PEPROCESS`结构的指针。

具体的查找过程可以分为以下几个步骤：

- 遍历系统中所有进程，获取每个进程的句柄（handle）和进程对象（process object）。
- 从进程对象中获取进程的名称，并与目标进程的名称进行比较。
- 如果找到了目标进程，从进程对象中获取指向PEPROCESS结构的指针。

需要注意的是，查找进程的过程可能会受到安全策略和权限的限制，因此需要在合适的上下文中进行。在实际应用中，需要根据具体的场景和要求进行合理的安全措施和权限管理。

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

NTKERNELAPI NTSTATUS PsLookupProcessByProcessId(HANDLE ProcessId, PEPROCESS *Process);
NTKERNELAPI CHAR* PsGetProcessImageFileName(PEPROCESS Process);

// 定义全局EProcess结构
PEPROCESS Global_Peprocess = NULL;

// 根据进程名获得EPROCESS结构
NTSTATUS GetProcessObjectByName(char *name)
{
    NTSTATUS Status = STATUS_UNSUCCESSFUL;
    SIZE_T i;

    __try
    {
        for (i = 100; i<20000; i += 4)
        {
            NTSTATUS st;
```

```
            PEPROCESS ep;
            st = PsLookupProcessByProcessId((HANDLE)i, &ep);
            if (NT_SUCCESS(st))
            {
                char *pn = PsGetProcessImageFileName(ep);
                if (_stricmp(pn, name) == 0)
                {
                    Global_Peprocess = ep;
                }
            }
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        return Status;
    }
    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");

    NTSTATUS nt = GetProcessObjectByName("Tutorial-i386.exe");

    if (NT_SUCCESS(nt))
    {
        DbgPrint("[+] eprocess = %x \n", Global_Peprocess);
    }

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

以打开`Tutorial-i386.exe`为例，打开后即可返回他的`Proces`，当然也可以直接传入进程PID同样可以得到进程`Process`结构地址。

```
// 根据PID打开进程
PEPROCESS Peprocess = NULL;
DWORD PID = 6672;
NTSTATUS nt = PsLookupProcessByProcessId((HANDLE)PID, &Peprocess);
```

CR3读取内存数据

通过CR3读取内存`Tutorial-i386.exe`里面的`0x0009EDC8`这段内存，读出长度是4字节，核心读取函数`CR3_ReadProcessMemory`其实现原理可概括为以下4步；

- 首先，函数的输入参数包括目标进程的`PEPROCESS`结构指针`Process`、要读取的内存地址`Address`、读取的字节数`Length`以及输出缓冲区的指针`Buffer`。函数的返回值为布尔类型，表示读取操作是否成功。
- 接着，函数使用了`CheckAddressVal`函数获取了目标进程页目录表的物理地址，并将其保存在变量`pDTB`中。如果获取页目录表的物理地址失败，函数直接返回FALSE，表示读取操作失败。
- 接下来，函数使用了汇编指令`_disable()`来禁用中断，然后调用`__readcr3()`函数获取当前系统的CR3寄存器的值，保存在变量`OldCr3`中。接着，函数使用`__writecr3()`函数将CR3寄存器的值设置为目标进程的页目录表的物理地址pDTB。这样就切换了当前系统的地址空间到目标进程的地址空间。
- 然后，函数使用了`MmIsAddressValid()`函数来判断要读取的内存地址是否可访问。如果可访问，函数就调用`RtlCopyMemory()`函数将目标进程内存中的数据复制到输出缓冲区中。函数最后打印了读入数据的信息，并返回TRUE表示读取操作成功。如果要读取的内存地址不可访问，函数就直接返回FALSE表示读取操作失败。

最后，函数使用了汇编指令`_enable()`来恢复中断，并使用`__writecr3()`函数将CR3寄存器的值设置为原来的值`OldCr3`，从而恢复了当前系统的地址空间。

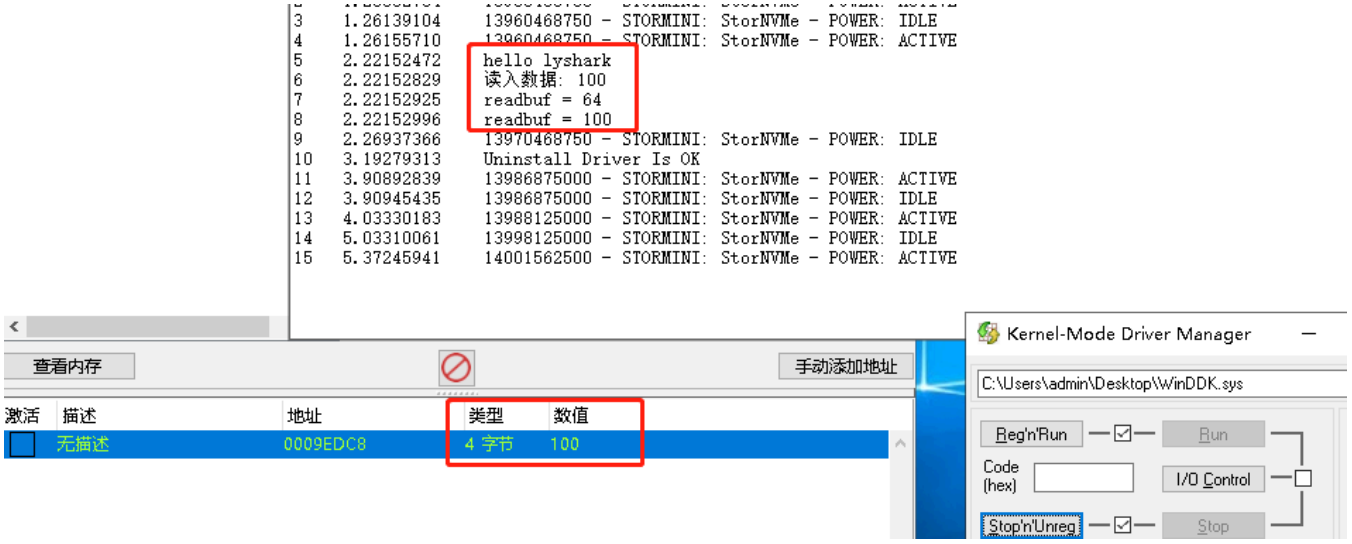
需要注意的是，这段代码仅仅实现了对目标进程内存的读取操作，如果需要进行写操作，还需要在适当的情况下使用类似的方式切换地址空间，并使用相关的内存操作函数进行写操作。另外，这种方式的内存读取操作可能会受到驱动保护的限制，需要谨慎使用。

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

#define DIRECTORY_TABLE_BASE 0x028

#pragma  intrinsic(_disable)
#pragma  intrinsic(_enable)

NTKERNELAPI NTSTATUS PsLookupProcessByProcessId(HANDLE ProcessId, PEPROCESS *Process);
NTKERNELAPI CHAR* PsGetProcessImageFileName(PEPROCESS Process);

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

// 开启写保护
void Close(KIRQL irql)
{
    UINT64 cr0 = __readcr0();
    cr0 |= 0x10000;
```

```
    _enable();
    __writecr0(cr0);
    KeLowerIrql(irql);
}

// 检查内存
ULONG64 CheckAddressVal(PVOID p)
{
    if (MmIsAddressValid(p) == FALSE)
        return 0;
    return *(PULONG64)p;
}

// CR3 寄存器读内存
BOOLEAN CR3_ReadProcessMemory(IN PEPROCESS Process, IN PVOID Address, IN UINT32 Length, OUT
PVOID Buffer)
{
    ULONG64 pDTB = 0, OldCr3 = 0, vAddr = 0;
    pDTB = CheckAddressVal((UCHAR*)Process + DIRECTORY_TABLE_BASE);
    if (pDTB == 0)
    {
        return FALSE;
    }

    _disable();
    OldCr3 = __readcr3();
    __writecr3(pDTB);
    _enable();

    if (MmIsAddressValid(Address))
    {
        RtlCopyMemory(Buffer, Address, Length);
        DbgPrint("读入数据: %ld", *(PDWORD)Buffer);
        return TRUE;
    }

    _disable();
    __writecr3(OldCr3);
    _enable();
    return FALSE;
}

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

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

    // 根据PID打开进程
    PEPROCESS Peprocess = NULL;
```

```
    DWORD PID = 6672;
    NTSTATUS nt = PsLookupProcessByProcessId((HANDLE)PID, &Peprocess);

    DWORD buffer = 0;

    BOOLEAN bl = CR3_ReadProcessMemory(Peprocess, (PVOID)0x0009EDC8, 4, &buffer);

    DbgPrint("readbuf = %x \n", buffer);
    DbgPrint("readbuf = %d \n", buffer);

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

读出后输出效果如下：



CR3写入内存数据

如下所示是一个在Windows操作系统下写入目标进程内存的函数，函数 `CR3_WriteProcessMemory()` 使用了 `CR3` 寄存器的切换来实现对特定进程内存地址的强制写操作。

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

#define DIRECTORY_TABLE_BASE 0x028

#pragma  intrinsic(_disable)
#pragma  intrinsic(_enable)

NTKERNELAPI NTSTATUS PsLookupProcessByProcessId(HANDLE ProcessId, PEPROCESS *Process);
NTKERNELAPI CHAR* PsGetProcessImageFileName(PEPROCESS Process);

// 关闭写保护
KIRQL Open()
{
    KIRQL irql = KeRaiseIrqlToDpcLevel();
    UINT64 cr0 = __readcr0();
```

```
    cr0 &= 0xfffffffffffeffff;
    __writecr0(cr0);
    _disable();
    return irql;
}

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

// 检查内存
ULONG64 CheckAddressVal(PVOID p)
{
    if (MmIsAddressValid(p) == FALSE)
        return 0;
    return *(PULONG64)p;
}

// CR3 寄存器写内存
BOOLEAN CR3_WriteProcessMemory(IN PEPROCESS Process, IN PVOID Address, IN UINT32 Length, IN
PVOID Buffer)
{
    ULONG64 pDTB = 0, OldCr3 = 0, vAddr = 0;

    // 检查内存
    pDTB = CheckAddressVal((UCHAR*)Process + DIRECTORY_TABLE_BASE);
    if (pDTB == 0)
    {
        return FALSE;
    }

    _disable();

    // 读取CR3
    OldCr3 = __readcr3();

    // 写CR3
    __writecr3(pDTB);
    _enable();

    // 验证并拷贝内存
    if (MmIsAddressValid(Address))
    {
        RtlCopyMemory(Address, Buffer, Length);
        return TRUE;
    }
    _disable();
```

```
    // 恢复CR3
    __writecr3(OldCr3);
    _enable();
    return FALSE;
}

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

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

    // 根据PID打开进程
    PEPROCESS Peprocess = NULL;
    DWORD PID = 6672;
    NTSTATUS nt = PsLookupProcessByProcessId((HANDLE)PID, &Peprocess);

    DWORD buffer = 999;

    BOOLEAN bl = CR3_WriteProcessMemory(Peprocess, (PVOID)0x0009EDC8, 4, &buffer);
    DbgPrint("写出状态: %d \n", bl);

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

写出后效果如下：