在某些时候我们需要读写的进程可能存在虚拟内存保护机制，在该机制下用户的`CR3`以及`MDL`读写将直接失效，从而导致无法读取到正确的数据，本章我们将继续研究如何实现物理级别的寻址读写。

首先，驱动中的物理页读写是指在驱动中直接读写物理内存页（而不是虚拟内存页）。这种方式的优点是它能够更快地访问内存，因为它避免了虚拟内存管理的开销，通过直接读写物理内存，驱动程序可以绕过虚拟内存的保护机制，获得对系统中内存的更高级别的访问权限。

想要实现物理页读写，第一步则是需要找到`UserDirectoryTableBase`的实际偏移地址，你一定会问这是个什么？别着急，听我来慢慢解释；

在操作系统中，每个进程都有一个`KPROCESS`结构体，它是进程的内部表示。该结构体中包含了一些重要的信息，包括`UserDirectoryTableBase`字段，它指向进程的页表目录表`（Page Directory Table）`，也称为`DirectoryTable`页目录表。

`Page Directory Table`是一种数据结构，它在虚拟内存管理中起着重要的作用。它被用来存储将虚拟地址映射到物理地址的映射关系，其内部包含了一些指向页表的指针，每个页表中又包含了一些指向物理页面的指针。这些指针一起构成了一个树形结构，它被称为页表树`（Page Table Tree）`。

```
kd> dt _KPROCESS
ntdll!_KPROCESS
+0x278 UserTime         : Uint4B
+0x27c ReadyTime        : Uint4B
+0x280 UserDirectoryTableBase : Uint8B
+0x288 AddressPolicy    : UChar
+0x289 Spare2           : [71] UChar

#define GetDirectoryTableOffset 0x280
```

`UserDirectoryTableBase`字段包含了进程的页表树的根节点的物理地址，通过它可以找到进程的页表树，从而实现虚拟内存的管理。在`WinDbg`中，通过输入`dt _KPROCESS`可以查看进程的`KPROCESS`结构体的定义，从而找到`UserDirectoryTableBase`字段的偏移量，这样可以获取该字段在内存中的地址，进而获取`DirectoryTable`的地址。不同操作系统的`KPROCESS`结构体定义可能会有所不同，因此它们的`UserDirectoryTableBase`字段的偏移量也会不同。

通过上述原理解释，我们可知要实现物理页读写需要实现一个转换函数，因为在应用层传入的还是一个虚拟地址，通过`TransformationCR3`函数即可实现将虚拟地址转换到物理地址，函数内部实现了从虚拟地址到物理地址的转换过程，并返回物理地址。

```
// 从用户层虚拟地址切换到物理页地址的函数
// 将 CR3 寄存器的末尾4个比特清零，这些比特是用于对齐的，不需要考虑
/*
    参数 cr3：物理地址。
    参数 VirtualAddress：虚拟地址。
*/
ULONG64 TransformationCR3(ULONG64 cr3, ULONG64 VirtualAddress)
{
  cr3 &= ~0xf;
  // 获取页面偏移量
  ULONG64 PAGE_OFFSET = VirtualAddress & ~(~0ul << 12);

  // 读取虚拟地址所在的三级页表项
  SIZE_T BytesTransferred = 0;
  ULONG64 a = 0, b = 0, c = 0;
```

```
  ReadPhysicalAddress((PVOID)(cr3 + 8 * ((VirtualAddress >> 39) & (0x1ffll))),
&a, sizeof(a), &BytesTransferred);

  // 如果 P（存在位）为0，表示该页表项没有映射物理内存，返回0
  if (~a & 1)
  {
    return 0;
  }

  // 读取虚拟地址所在的二级页表项
  ReadPhysicalAddress((PVOID)((a & ((~0xfull << 8) & 0xfffffffffull)) + 8 *
((VirtualAddress >> 30) & (0x1ffll))), &b, sizeof(b), &BytesTransferred);

  // 如果 P 为0，表示该页表项没有映射物理内存，返回0
  if (~b & 1)
  {
    return 0;
  }

  // 如果 PS（页面大小）为1，表示该页表项映射的是1GB的物理内存，直接计算出物理地址并返回
  if (b & 0x80)
  {
    return (b & (~0ull << 42 >> 12)) + (VirtualAddress & ~(~0ull << 30));
  }

  // 读取虚拟地址所在的一级页表项
  ReadPhysicalAddress((PVOID)((b & ((~0xfull << 8) & 0xfffffffffull)) + 8 *
((VirtualAddress >> 21) & (0x1ffll))), &c, sizeof(c), &BytesTransferred);

  // 如果 P 为0，表示该页表项没有映射物理内存，返回0
  if (~c & 1)
  {
    return 0;
  }
  // 如果 PS 为1，表示该页表项映射的是2MB的物理内存，直接计算出物理地址并返回
  if (c & 0x80)
  {
    return (c & ((~0xfull << 8) & 0xfffffffffull)) + (VirtualAddress & ~(~0ull <<
21));
  }
  // 读取虚拟地址所在的零级页表项，计算出物理地址并返回
  ULONG64 address = 0;
  ReadPhysicalAddress((PVOID)((c & ((~0xfull << 8) & 0xfffffffffull)) + 8 *
((VirtualAddress >> 12) & (0x1ffll))), &address, sizeof(address),
&BytesTransferred);
  address &= ((~0xfull << 8) & 0xfffffffffull);
  if (!address)
  {
    return 0;
  }

  return address + PAGE_OFFSET;
}
```

这段代码将输入的`CR3`值和虚拟地址作为参数，并将`CR3`值和虚拟地址的偏移量进行一系列计算，最终得出物理地址。

其中，`CR3`是存储页表的物理地址，它保存了虚拟地址到物理地址的映射关系。该函数通过读取`CR3`中存储的页表信息，逐级访问页表，直到找到对应的物理地址。

该函数使用虚拟地址的`高9位`确定页表的索引，然后通过读取对应的页表项，得到下一级页表的物理地址。该过程重复执行，直到读取到页表的最后一级，得到物理地址。

最后，该函数将物理地址的`低12位`与虚拟地址的偏移量进行`OR`运算，得到最终的物理地址，并将其返回。

需要注意的是，该函数还会进行一些错误处理，例如在读取页表项时，如果该项没有被设置为有效，函数将返回0，表示无法访问对应的物理地址。

此时用户已经获取到了物理地址，那么读写就变得很容易了，当需要读取数据时调用`ReadPhysicalAddress`函数，其内部直接使用`MmCopyMemory`对内存进行拷贝即可，而对于写入数据而言，需要通过调用`MmMapIoSpace`先将物理地址转换为一个用户空间的虚拟地址，然后再通过`RtlCopyMemory`向内部拷贝数据即可实现写入，这三段代码的封装如下所示；

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

#define GetDirectoryTableOffset 0x280
#define bit64 0x28
#define bit32 0x18

// 读取物理内存封装
// 这段代码实现了将物理地址映射到内核空间，然后将物理地址对应的数据读取到指定的缓冲区中。
/*
    address：需要读取的物理地址；
    buffer：读取到的数据需要保存到的缓冲区；
    size：需要读取的数据大小；
    BytesTransferred：实际读取到的数据大小。
*/
NTSTATUS ReadPhysicalAddress(PVOID address, PVOID buffer, SIZE_T size, SIZE_T*
BytesTransferred)
{
  MM_COPY_ADDRESS Read = { 0 };
  Read.PhysicalAddress.QuadPart = (LONG64)address;
  return MmCopyMemory(buffer, Read, size, MM_COPY_MEMORY_PHYSICAL,
BytesTransferred);
}

// 写入物理内存
// 这段代码实现了将数据写入物理地址的功能
/*
    参数 address：要写入的物理地址。
    参数 buffer：要写入的数据缓冲区。
    参数 size：要写入的数据长度。
    参数 BytesTransferred：实际写入的数据长度。
*/
NTSTATUS WritePhysicalAddress(PVOID address, PVOID buffer, SIZE_T size, SIZE_T*
BytesTransferred)
{
  if (!address)
```

```
  {
    return STATUS_UNSUCCESSFUL;
  }

  PHYSICAL_ADDRESS Write = { 0 };
  Write.QuadPart = (LONG64)address;

    // 将物理空间映射为虚拟空间
  PVOID map = MmMapIoSpace(Write, size, (MEMORY_CACHING_TYPE)PAGE_READWRITE);

  if (!map)
  {
    return STATUS_UNSUCCESSFUL;
  }

    // 开始拷贝数据
  RtlCopyMemory(map, buffer, size);
  *BytesTransferred = size;
  MmUnmapIoSpace(map, size);
  return STATUS_SUCCESS;
}

// 从用户层虚拟地址切换到物理页地址的函数
// 将 CR3 寄存器的末尾4个比特清零，这些比特是用于对齐的，不需要考虑
/*
    参数 cr3：物理地址。
    参数 VirtualAddress：虚拟地址。
*/
ULONG64 TransformationCR3(ULONG64 cr3, ULONG64 VirtualAddress)
{
  cr3 &= ~0xf;
  // 获取页面偏移量
  ULONG64 PAGE_OFFSET = VirtualAddress & ~(~0ul << 12);

  // 读取虚拟地址所在的三级页表项
  SIZE_T BytesTransferred = 0;
  ULONG64 a = 0, b = 0, c = 0;

  ReadPhysicalAddress((PVOID)(cr3 + 8 * ((VirtualAddress >> 39) & (0x1ffll))),
&a, sizeof(a), &BytesTransferred);

  // 如果 P（存在位）为0，表示该页表项没有映射物理内存，返回0
  if (~a & 1)
  {
    return 0;
  }

  // 读取虚拟地址所在的二级页表项
  ReadPhysicalAddress((PVOID)((a & ((~0xfull << 8) & 0xfffffffffull)) + 8 *
((VirtualAddress >> 30) & (0x1ffll))), &b, sizeof(b), &BytesTransferred);

  // 如果 P 为0，表示该页表项没有映射物理内存，返回0
  if (~b & 1)
  {
    return 0;
  }
```

```
  // 如果 PS（页面大小）为1，表示该页表项映射的是1GB的物理内存，直接计算出物理地址并返回
  if (b & 0x80)
  {
    return (b & (~0ull << 42 >> 12)) + (VirtualAddress & ~(~0ull << 30));
  }

  // 读取虚拟地址所在的一级页表项
  ReadPhysicalAddress((PVOID)((b & ((~0xfull << 8) & 0xfffffffffull)) + 8 *
((VirtualAddress >> 21) & (0x1ffll))), &c, sizeof(c), &BytesTransferred);

  // 如果 P 为0，表示该页表项没有映射物理内存，返回0
  if (~c & 1)
  {
    return 0;
  }
  // 如果 PS 为1，表示该页表项映射的是2MB的物理内存，直接计算出物理地址并返回
  if (c & 0x80)
  {
    return (c & ((~0xfull << 8) & 0xfffffffffull)) + (VirtualAddress & ~(~0ull <<
21));
  }
  // 读取虚拟地址所在的零级页表项，计算出物理地址并返回
  ULONG64 address = 0;
  ReadPhysicalAddress((PVOID)((c & ((~0xfull << 8) & 0xfffffffffull)) + 8 *
((VirtualAddress >> 12) & (0x1ffll))), &address, sizeof(address),
&BytesTransferred);
  address &= ((~0xfull << 8) & 0xfffffffffull);
  if (!address)
  {
    return 0;
  }

  return address + PAGE_OFFSET;
}
```

有了如上封装，那么我们就可以实现驱动读写了，首先我们实现驱动读取功能，如下这段代码是`Windows`驱动程序的入口函数`DriverEntry`，主要功能是读取指定进程的虚拟地址空间中指定地址处的`4`个字节数据。

代码首先通过`PsLookupProcessByProcessId`函数获取指定进程的`EPROCESS`结构体指针。然后获取该进程的`CR3`值，用于将虚拟地址转换为物理地址。接下来，循环读取指定地址处的`4`个字节数据，每次读取`PAGE_SIZE`大小的物理内存数据。最后输出读取到的数据，并关闭对`EPROCESS`结构体指针的引用。

需要注意的是，该代码并没有进行有效性检查，如没有检查读取的地址是否合法、读取的数据是否在用户空间，因此存在潜在的风险。另外，该代码也没有考虑内核模式下访问用户空间数据的问题，因此也需要进行进一步的检查和处理。

```
// 驱动卸载例程
extern "C" VOID DriverUnload(PDRIVER_OBJECT pDriver)
{
  UNREFERENCED_PARAMETER(pDriver);
  DbgPrint("Uninstall Driver \n");
}
```

```
// 驱动入口地址
extern "C" NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING path)
{
  DbgPrint("Hello LyShark \n");

  // 通过进程ID获取eprocess
  PEPROCESS pEProcess = NULL;
  NTSTATUS Status = PsLookupProcessByProcessId((HANDLE)4116, &pEProcess);

  if (NT_SUCCESS(Status) && pEProcess != NULL)
  {

    ULONG64 TargetAddress = 0x401000;
    SIZE_T TargetSize = 4;
    SIZE_T read = 0;

    // 分配读取空间
    BYTE* ReadBuffer = (BYTE *)ExAllocatePool(NonPagedPool, 1024);

    // 获取CR3用于转换
    PUCHAR Var = reinterpret_cast<PUCHAR>(pEProcess);
    ULONG64 CR3 = *(ULONG64*)(Var + bit64);
    if (!CR3)
    {
      CR3 = *(ULONG64*)(Var + GetDirectoryTableOffset);
    }

    DbgPrint("[CR3] 寄存器地址 = 0x%p \n", CR3);

    while (TargetSize)
    {
      // 开始循环切换到CR3
      ULONG64 PhysicalAddress = TransformationCR3(CR3, TargetAddress + read);
      if (!PhysicalAddress)
      {
        break;
      }

      // 读取物理内存
      ULONG64 ReadSize = min(PAGE_SIZE - (PhysicalAddress & 0xfff), TargetSize);
      SIZE_T BytesTransferred = 0;

      // reinterpret_cast 强制转为PVOID类型
      Status = ReadPhysicalAddress(reinterpret_cast<PVOID>(PhysicalAddress),
reinterpret_cast<PVOID>((PVOID *)ReadBuffer + read), ReadSize,
&BytesTransferred);
      TargetSize -= BytesTransferred;
      read += BytesTransferred;

      if (!NT_SUCCESS(Status))
      {
        break;
      }

      if (!BytesTransferred)
```

```
        {
          break;
        }
      }

      // 关闭引用
      ObDereferenceObject(pEProcess);

      // 输出读取字节
      for (size_t i = 0; i < 4; i++)
      {
        DbgPrint("[读入字节 [%d] ] => 0x%02X \n", i, ReadBuffer[i]);
      }
    }

    // 关闭引用
    UNREFERENCED_PARAMETER(path);

    // 卸载驱动
    pDriver->DriverUnload = DriverUnload;
    return STATUS_SUCCESS;
}
```

编译并运行上述代码片段，则会读取进程ID为 `4116` 的 `0x401000` 处的地址数据，并以字节的方式输出前四位，输出效果图如下所示；

写出数据与读取数据基本一致，只是调用方法从 `ReadPhysicalAddress` 变为了 `WritePhysicalAddress` 其他的照旧，但需要注意的是读者再使用写出时需要自行填充一段堆用于存储需要写出的字节集。

```
// 驱动卸载例程
extern "C" VOID DriverUnload(PDRIVER_OBJECT pDriver)
{
    UNREFERENCED_PARAMETER(pDriver);
    DbgPrint("Uninstall Driver \n");
}

// 驱动入口地址
extern "C" NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING path)
{
    DbgPrint("Hello LyShark \n");

    // 物理页写
    PEPROCESS pEProcess = NULL;
    NTSTATUS Status = PsLookupProcessByProcessId((HANDLE)4116, &pEProcess);

    // 判断pEProcess是否有效
    if (NT_SUCCESS(Status) && pEProcess != NULL)
    {
        ULONG64 TargetAddress = 0x401000;
        SIZE_T TargetSize = 4;
        SIZE_T read = 0;
```

```
    // 申请空间并填充写出字节0x90
    BYTE* ReadBuffer = (BYTE *)ExAllocatePool(NonPagedPool, 1024);

    for (size_t i = 0; i < 4; i++)
    {
      ReadBuffer[i] = 0x90;
    }

    // 获取CR3用于转换
    PUCHAR Var = reinterpret_cast<PUCHAR>(pEProcess);
    ULONG64 CR3 = *(ULONG64*)(Var + bit64);
    if (!CR3)
    {
      CR3 = *(ULONG64*)(Var + GetDirectoryTableOffset);
      // DbgPrint("[CR3] 寄存器地址 = 0x%p \n", CR3);
    }

    while (TargetSize)
    {
      // 开始循环切换到CR3
      ULONG64 PhysicalAddress = TransformationCR3(CR3, TargetAddress + read);
      if (!PhysicalAddress)
      {
        break;
      }

      // 写入物理内存
      ULONG64 WriteSize = min(PAGE_SIZE - (PhysicalAddress & 0xfff), TargetSize);
      SIZE_T BytesTransferred = 0;
      Status = WritePhysicalAddress(reinterpret_cast<PVOID>(PhysicalAddress),
reinterpret_cast<PVOID>(ReadBuffer + read), WriteSize, &BytesTransferred);
      TargetSize -= BytesTransferred;
      read += BytesTransferred;

      // DbgPrint("[写出数据] => %d | %0x02X \n", WriteSize, ReadBuffer + read);
      if (!NT_SUCCESS(Status))
      {
        break;
      }

      if (!BytesTransferred)
      {
        break;
      }
    }

    // 关闭引用
    ObDereferenceObject(pEProcess);
  }

  // 关闭引用
  UNREFERENCED_PARAMETER(path);

  // 卸载驱动
  pDriver->DriverUnload = DriverUnload;
  return STATUS_SUCCESS;
```

```
}
```

如上代码运行后，会向进程ID为`4116`的`0x401000`处写出4字节的`0x90`机器码，读者可通过第三方工具验证内存，输出效果如下所示；