MDL内存读写是一种通过创建MDL结构体来实现跨进程内存读写的方式。在Windows操作系统中，每个进程都有自己独立的虚拟地址空间，不同进程之间的内存空间是隔离的。因此，要在一个进程中读取或写入另一个进程的内存数据，需要先将目标进程的物理内存映射到当前进程的虚拟地址空间中，然后才能进行内存读写操作。

MDL结构体是Windows内核中专门用于描述物理内存的数据结构，它包含了一系列的数据元素，包括物理地址、长度、内存映射的虚拟地址等信息。通过创建MDL结构体并调用系统函数将其映射到当前进程的虚拟地址空间中，即可实现跨进程内存读写的操作。

相比于CR3切换方式，MDL内存读写更加稳定、安全，且不会受到寄存器的影响。同时，使用MDL内存读写方式还可以充分利用Windows操作系统的内存管理机制，从而实现更为高效的内存读写操作。因此，MDL内存读写是Windows操作系统中最为常用和推荐的一种跨进程内存读写方式。

MDL读取内存步骤

- 1.调用 `PsLookupProcessByProcessId` 得到进程 `Process` 结构，这个函数是用于根据进程ID查找对应的进程对象的函数，通过传入的参数 `data->pid` 获取到对应的进程ID，然后通过调用 `PsLookupProcessByProcessId` 函数获取对应的 `PEPROCESS` 结构。如果获取失败则返回 FALSE。
- 2.调用 `KeStackAttachProcess` 附加到对端进程内，在内核模式下，读取其他进程的内存时需要先附加到对应进程的上下文中，才能读取该进程的内存。因此，这里调用 `KeStackAttachProcess` 函数将当前线程切换到目标进程的上下文中。同时，为了在后面可以正确地从目标进程的上下文中返回，还需要保存当前进程的上下文状态。
- 3.调用 `ProbeForRead` 检查内存是否可读写，在内核模式下，需要保证访问其他进程的内存是合法的，因此需要先调用 `ProbeForRead` 函数检查读取的内存空间是否可读写。如果该空间不可读写，则会触发异常，这里通过异常处理机制来处理这种情况。
- 4.拷贝内存空间中的数据到自己的缓冲区内，在完成对内存空间的检查后，使用 `RtlCopyMemory` 函数将目标进程的内存数据拷贝到自己的缓冲区中。这里需要注意的是，由于内存空间可能很大，因此可能需要多次进行拷贝操作。
- 5.调用 `KeUnstackDetachProcess` 接触绑定，在读取完内存数据后，需要将当前线程从目标进程的上下文中解除绑定，以便返回到原来的上下文中。这里调用 `KeUnstackDetachProcess` 函数完成解绑操作，同时恢复之前保存的当前进程的上下文状态。
- 6.调用 `ObDereferenceObject` 使对象引用数减1，由于在第一步中调用了 `PsLookupProcessByProcessId` 函数获取了对应进程的 `PEPROCESS` 结构，因此需要调用 `ObDereferenceObject` 函数将其引用计数减1，以便释放对该对象的引用。

有了上述具体实现方法，那么我们就可以封装 `MDLReadMemory()` 内存读函数了，代码如下，该函数用于在 Windows 内核模式下读取指定进程的内存数据。下面是对这个函数的详细步骤分析：

- 1.通过进程 ID 找到对应的进程对象： `PsLookupProcessByProcessId` 用于通过进程 ID 查找对应的进程对象。如果找不到该进程对象，则直接返回 FALSE。

```
PsLookupProcessByProcessId(data->pid, &process);
```

- 2.在内核模式下，必须使用内核提供的函数来分配内存。这里使用的是 `ExAllocatePool` 函数，用于在内核堆中分配指定大小的内存缓冲区。如果分配失败，则返回 FALSE。

```
BYTE* GetData;
__try
{
    GetData = ExAllocatePool(PagedPool, data->size);
}
__except (1)
{
    return FALSE;
}
```

- 3.在内核模式下，访问其他进程的内存必须先将当前进程的上下文切换到目标进程的上下文。这里使用的是 `KeStackAttachProcess` 函数，将当前进程的上下文切换到目标进程的上下文。同时，为了在后面可以正确地从目标进程的上下文中返回，还需要保存当前进程的上下文状态。

```
KAPC_STATE stack = { 0 };
KeStackAttachProcess(process, &stack);
```

- 4.读取目标进程的内存数据，这段代码使用 `ProbeForRead` 函数检查要读取的内存区域是否合法，并且将目标进程的内存数据读取到之前分配的内存缓冲区中。如果读取过程中出现异常，则返回 FALSE。

```
__try
{
    ProbeForRead(data->address, data->size, 1);
    RtlCopyMemory(GetData, data->address, data->size);
}
__except (1)
{
    bRet = FALSE;
}
```

- 5.恢复当前进程的上下文，这里使用的是 `ObDereferenceObject` 函数和 `KeUnstackDetachProcess` 函数，用于恢复之前保存的当前进程的上下文状态，同时解除对目标进程的引用计数。

```
ObDereferenceObject(process);
KeUnstackDetachProcess(&stack);
```

- 6.将读取的数据拷贝到输出参数中，将读取到的数据拷贝到输出参数中，并释放之前分配的内存缓冲区。

```
RtlCopyMemory(data->data, GetData, data->size);
```

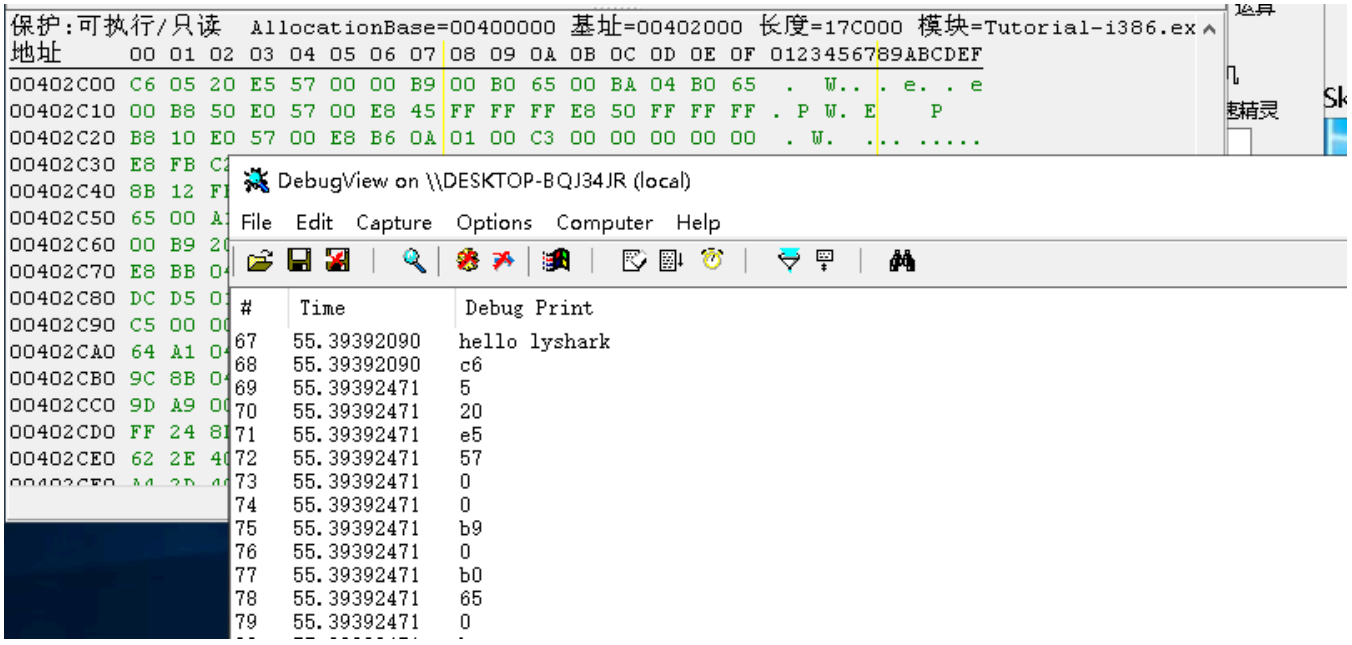
将如上代码片段整合起来即可得到一个完整的内存读数据案例，读者可传入一个结构体实现对特定进程特定内存的动态读取功能，完整代码如下所示；

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

typedef struct
{
    DWORD pid;                  // 要读写的进程ID
```

```
    DWORD64 address;            // 要读写的地址
    DWORD size;                 // 读写长度
    BYTE* data;                 // 要读写的数据
}ReadMemoryStruct;

// MDL读内存
BOOL MDLReadMemory(ReadMemoryStruct* data)
{
    BOOL bRet = TRUE;
    PEPROCESS process = NULL;

    PsLookupProcessByProcessId(data->pid, &process);

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

    BYTE* GetData;
    __try
    {
        GetData = ExAllocatePool(PagedPool, data->size);
    }
    __except (1)
    {
        return FALSE;
    }

    KAPC_STATE stack = { 0 };
    KeStackAttachProcess(process, &stack);

    __try
    {
        ProbeForRead(data->address, data->size, 1);
        RtlCopyMemory(GetData, data->address, data->size);
    }
    __except (1)
    {
        bRet = FALSE;
    }

    ObDereferenceObject(process);
    KeUnstackDetachProcess(&stack);
    RtlCopyMemory(data->data, GetData, data->size);
    ExFreePool(GetData);
    return bRet;
}

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

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

    ReadMemoryStruct ptr;

    ptr.pid = 6672;
    ptr.address = 0x402c00;
    ptr.size = 100;

    // 分配空间接收数据
    ptr.data = ExAllocatePool(PagedPool, ptr.size);

    // 读内存
    MDLReadMemory(&ptr);

    // 输出数据
    for (size_t i = 0; i < 100; i++)
    {
        DbgPrint("%x \n", ptr.data[i]);
    }

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

读取内存地址`0x402c00`效果如下所示：



MDL写入内存步骤

- 1.首先需要通过调用`PsLookupProcessByProcessId`函数获取目标进程的进程结构，该函数将根据传递的进程ID返回对应进程的`PEPROCESS`结构体，该结构体中包含了进程的各种信息。
- 2.接下来使用`KeStackAttachProcess`函数附加到目标进程的上下文环境中，以便可以读取和写入该进程的内

存空间。该函数将当前线程的上下文环境切换到目标进程的上下文环境中，使得该线程可以访问和修改目标进程的内存。

- 3.在进行内存写入操作之前，需要调用 `ProbeForRead` 函数来检查要写入的内存空间是否可读写。这个步骤是为了确保要写入的内存空间没有被保护或被其他进程占用，以避免对系统造成不良影响。
- 4.如果检查通过，接下来需要将目标进程的内存空间中的数据拷贝到当前进程的缓冲区中，以便进行修改操作。
- 5.接下来需要调用 `MmMapLockedPages` 函数来锁定当前内存页面，以便可以对其进行修改。该函数将返回一个指向系统虚拟地址的指针，该地址是由系统自动分配的。在写入完成后，需要使用 `MmUnmapLockedPages` 函数来释放锁定的内存页面。
- 6.然后，使用 `RtlCopyMemory` 函数完成内存拷贝操作，将缓冲区中的数据写入到锁定的内存页面中。
- 7.写入操作完成后，需要调用 `IoFreeMdl` 函数来释放MDL锁。MDL锁用于锁定MDL描述的内存页面，以便可以对其进行操作。
- 8.最后使用 `KeUnstackDetachProcess` 函数解除当前进程与目标进程之间的绑定，使得当前线程的上下文环境恢复到原始的状态。

此外在完成MDL写入内存操作后，还需要调用 `ObDereferenceObject` 函数将MDL对象的引用计数减1，以便在不再需要该对象时释放它所占用的系统资源。

从如上分析来看写入时与读取基本类似，只是多了锁定页面和解锁操作，这段MDL写内存完整实现代码如下所示；

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

typedef struct
{
    DWORD pid;                  // 要读写的进程ID
    DWORD64 address;            // 要读写的地址
    DWORD size;                 // 读写长度
    BYTE* data;                 // 要读写的数据
}WriteMemoryStruct;

// MDL写内存
BOOL MDLWriteMemory(WriteMemoryStruct* data)
{
    BOOL bRet = TRUE;
    PEPROCESS process = NULL;

    PsLookupProcessByProcessId(data->pid, &process);
    if (process == NULL)
    {
        return FALSE;
    }

    BYTE* GetData;
    __try
    {
        GetData = ExAllocatePool(PagedPool, data->size);
    }
    __except (1)
    {
```

```
        return FALSE;
    }

    for (int i = 0; i < data->size; i++)
    {
        GetData[i] = data->data[i];
    }

    KAPC_STATE stack = { 0 };
    KeStackAttachProcess(process, &stack);

    PMDL mdl = IoAllocateMdl(data->address, data->size, 0, 0, NULL);
    if (mdl == NULL)
    {
        return FALSE;
    }

    MmBuildMdlForNonPagedPool(mdl);

    BYTE* ChangeData = NULL;

    __try
    {
        ChangeData = MmMapLockedPages(mdl, KernelMode);
        RtlCopyMemory(ChangeData, GetData, data->size);
    }
    __except (1)
    {
        bRet = FALSE;
        goto END;
    }

END:
    IoFreeMdl(mdl);
    ExFreePool(GetData);
    KeUnstackDetachProcess(&stack);
    ObDereferenceObject(process);

    return bRet;
}

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

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

    WriteMemoryStruct ptr;

    ptr.pid = 6672;
```

```
    ptr.address = 0x402c00;
    ptr.size = 5;

    // 需要写入的数据
    ptr.data = ExAllocatePool(PagedPool, ptr.size);

    // 循环设置
    for (size_t i = 0; i < 5; i++)
    {
        ptr.data[i] = 0x90;
    }

    // 写内存
    MDLWriteMemory(&ptr);

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

写出效果如下：