当今操作系统普遍采用64位架构，这意味着CPU寄存器的位数、内存地址总线宽度、指令集等都可以支持64位，从而提高了系统的计算能力和内存管理能力。具体来说，在64位操作系统中，CPU的最大寻址能力为64位，可以寻址的内存空间大小为2^64，这是一个非常庞大的地址空间。

然而，由于实际系统所使用的物理内存大小是有限的，为了高效利用内存，操作系统采用了分页机制进行内存管理。分页机制将整个物理内存划分成固定大小的页框，一般为4KB或者2MB，每个页框都有一个对应的页表项来描述它的物理地址。而虚拟内存则被分成对应的虚拟页，每个虚拟页也有对应的页表项，用来映射到物理页框上。

在64位操作系统中，内存管理采用了`9-9-9-9-12`的分页模式，表示物理地址拥有`四级页表`。具体来说，微软将这四级依次命名为`PXE、PPE、PDE、PTE`这四项。其中，PXE表示最高级别的页表，每个PXE项可以映射512GB的物理地址范围；PPE表示第二级页表，每个PPE项可以映射1GB的物理地址范围；PDE表示第三级页表，每个PDE项可以映射2MB的物理地址范围；PTE表示最低级别的页表，每个PTE项可以映射4KB的物理地址范围。

通过这种分页模式，操作系统可以高效地管理大量的内存，同时还可以提高内存的访问效率。每个进程都有自己独立的虚拟地址空间，通过虚拟地址可以方便地访问相应的物理内存。这种机制还可以实现虚拟内存，使得进程可以访问比实际物理内存更大的虚拟内存，从而提高了系统的运行效率和稳定性。

总结为一段话就是CPU最大寻址能力虽然达到了64位，但其实仅仅只是用到了48位进行寻址，其内存管理采用了`9-9-9-9-12`的分页模式，`9-9-9-9-12`分页表示物理地址拥有四级页表，微软将这四级依次命名为PXE、PPE、PDE、PTE这四项。

首先一个PTE管理1个分页大小的内存也就是`0x1000`字节，PTE结构的解析非常容易，打开WinDBG输入`!PTE 0`即可解析，如下所示，当前地址0位置处的PTE基址是`FFFF898000000000`，由于PTE的一个页大小是`0x1000`所以当内存地址高于`0x1000`时将会切换到另一个页中，如下`FFFF898000000008`则是另一个页中的地址。

```
0: kd> !PTE 0
                                           VA 0000000000000000
PXE at FFFF89C4E2713000    PPE at FFFF89C4E2600000    PDE at FFFF89C4C0000000    PTE at
FFFF898000000000
contains 8A0000000405F867  contains 0000000000000000
pfn 405f      ---DA--UW-V  not valid

0: kd> !PTE 0x1000
                                           VA 0000000000001000
PXE at FFFF89C4E2713000    PPE at FFFF89C4E2600000    PDE at FFFF89C4C0000000    PTE at
FFFF898000000008
contains 8A0000000405F867  contains 0000000000000000
pfn 405f      ---DA--UW-V  not valid
```

由于PTE是动态变化的，找到该地址的关键就在于通过`MmGetSystemRoutineAddress`函数动态得到`MmGetVirtualForPhysical`的内存地址，然后向下扫描特征寻找`mov rdx,0FFFF8B0000000000h`并将内部的地址提取出来。

Command

```
1: kd> uf MmGetVirtualForPhysical
nt!MmGetVirtualForPhysical:
fffff802`674c1d00 488bc1          mov     rax,rcx
fffff802`674c1d03 48c1e80c        shr     rax,0Ch
fffff802`674c1d07 488d1440        lea     rdx,[rax+rax*2]
fffff802`674c1d0b 4803d2          add     rdx,rdx
fffff802`674c1d0e 48b808000000008cffff mov rax,0FFFF8C0000000008h
fffff802`674c1d18 488b04d0        mov     rax,qword ptr [rax+rdx*8]
fffff802`674c1d1c 48c1e019        shl     rax,19h
fffff802`674c1d20 48ba00000000008bffff mov rdx,0FFFF8B0000000000h
fffff802`674c1d2a 48c1e219        shl     rdx,19h
fffff802`674c1d2e 81e1ff0f0000    and     ecx,0FFFh
fffff802`674c1d34 482bc2          sub     rax,rdx
fffff802`674c1d37 48c1f810        sar     rax,10h
fffff802`674c1d3b 4803c1          add     rax,rcx
fffff802`674c1d3e c3              ret
```

这段代码完整版如下所示，代码可动态定位到PTE的内存地址，然后将其取出；

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

// 指定内存区域的特征码扫描
PVOID SearchMemory(PVOID pStartAddress, PVOID pEndAddress, PUCHAR pMemoryData, ULONG
ulMemoryDataSize)
{
    PVOID pAddress = NULL;
    PUCHAR i = NULL;
    ULONG m = 0;

    // 扫描内存
    for (i = (PUCHAR)pStartAddress; i < (PUCHAR)pEndAddress; i++)
    {
        // 判断特征码
        for (m = 0; m < ulMemoryDataSize; m++)
        {
            if (*(PUCHAR)(i + m) != pMemoryData[m])
            {
                break;
            }
        }
        // 判断是否找到符合特征码的地址
        if (m >= ulMemoryDataSize)
        {
            // 找到特征码位置，获取紧接着特征码的下一地址
            pAddress = (PVOID)(i + ulMemoryDataSize);
            break;
        }
    }

    return pAddress;
}

// 获取到函数地址
PVOID GetMmGetVirtualForPhysical()
{
    PVOID VariableAddress = 0;
```

```
    UNICODE_STRING uioiTime = { 0 };

    RtlInitUnicodeString(&uioiTime, L"MmGetVirtualForPhysical");
    VariableAddress = (PVOID)MmGetSystemRoutineAddress(&uioiTime);
    if (VariableAddress != 0)
    {
        return VariableAddress;
    }
    return 0;
}

// 驱动卸载例程
VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("Uninstall Driver \n");
}

// 驱动入口地址
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("Hello LyShark \n");

    // 获取函数地址
    PVOID address = GetMmGetVirtualForPhysical();

    DbgPrint("GetMmGetVirtualForPhysical = %p \n", address);

    UCHAR pSecondSpecialData[50] = { 0 };
    ULONG ulFirstSpecialDataSize = 0;

    pSecondSpecialData[0] = 0x48;
    pSecondSpecialData[1] = 0xc1;
    pSecondSpecialData[2] = 0xe0;
    ulFirstSpecialDataSize = 3;

    // 定位特征码
    PVOID PTE = SearchMemory(address, (PVOID)((PUCHAR)address + 0xFF), pSecondSpecialData,
ulFirstSpecialDataSize);
    __try
    {
        PVOID lOffset = (ULONG)PTE + 1;
        DbgPrint("PTE Address = %p \n", lOffset);
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        DbgPrint("error");
    }

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

运行如上代码可动态获取到当前系统的PTE地址，然后将PTE填入到`g_PTEBASE`中，即可实现解析系统内的四个标志位，完整解析代码如下所示；

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

INT64 g_PTEBASE = 0;
INT64 g_PDEBASE = 0;
INT64 g_PPEBASE = 0;
INT64 g_PXEBASE = 0;

PULONG64 GetPteBase(PVOID va)
{
    return (PULONG64)((((ULONG64)va & 0xFFFFFFFFFFFF) >> 12) * 8) + g_PTEBASE;
}

PULONG64 GetPdeBase(PVOID va)
{
    return (PULONG64)((((ULONG64)va & 0xFFFFFFFFFFFF) >> 12) * 8) + g_PDEBASE;
}

PULONG64 GetPpeBase(PVOID va)
{
    return (PULONG64)((((ULONG64)va & 0xFFFFFFFFFFFF) >> 12) * 8) + g_PPEBASE;
}

PULONG64 GetPxeBase(PVOID va)
{
    return (PULONG64)((((ULONG64)va & 0xFFFFFFFFFFFF) >> 12) * 8) + g_PXEBASE;
}

// 驱动卸载例程
VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("Uninstall Driver \n");
}

// 驱动入口地址
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("Hello LyShark \n");

    g_PTEBASE = 0XFFFFF20000000000;

    g_PDEBASE = (ULONG64)GetPteBase((PVOID)g_PTEBASE);
    g_PPEBASE = (ULONG64)GetPteBase((PVOID)g_PDEBASE);
    g_PXEBASE = (ULONG64)GetPteBase((PVOID)g_PPEBASE);

    DbgPrint("PXE = %p \n", g_PXEBASE);
    DbgPrint("PPE  = %p \n", g_PPEBASE);
    DbgPrint("PDE  = %p \n", g_PDEBASE);
    DbgPrint("PTE  = %p \n", g_PTEBASE);
```

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

我的系统内PTE地址为`0XFFFFF20000000000`，填入变量内解析效果如下图所示；