漫谈兼容内核之一：
ReactOS 怎样实现系统调用

毛德操

有网友在论坛上发贴，要求我谈谈 ReactOS 是怎样实现系统调用的。另一方面，我上次已经谈到兼容内核应该如何实现 Windows 系统调用的问题，接着谈谈 ReactOS 怎样实现系统调用倒也顺理成章，所以这一次就来谈谈这个话题。不过这显然不属于“漫谈 Wine”的范畴，也确实没有必要再来个“漫谈 ReactOS”，因此决定把除 Wine 以外的话题都纳入“漫谈兼容内核”。

ReactOS 这个项目的目标是要开发出一个开源的 Windows。不言而喻，它要实现的系统调用就是 Windows 的那一套系统调用，也就是要忠实地实现 Windows 系统调用界面。本文要说的不是 Windows 系统调用界面本身，而是 ReactOS 怎样实现这个界面，主要是说说用户空间的应用程序怎样进入/退出内核、即系统空间，怎样调用定义于这个界面的函数。实际上，ReactOS 正是通过“int 0x2e”指令进入内核、实现系统调用的。虽然 ReactOS 并不是 Windows，它的作者们也未必看到过 Windows 的源代码；但是我相信，ReactOS 的代码、至少是这方面的代码，与“正本”Windows 的代码应该非常接近，要有也只是细节上的差别。

下面以系统调用 NtReadFile()为例，按“自顶向下”的方式，一方面说明怎样阅读 ReactOS 的代码，一方面说明 ReacOS 是怎样实现系统调用的。

首先，Windows 应用程序应该通过 Win32 API 调用这个接口所定义的库函数，这些库函数基本上都是在“动态连接库”、即 DLL 中实现的。例如，ReadFile()就是在 Win32 API 中定义的一个库函数。实现这个库函数的可执行程序在 Windows 的“系统 DLL”之一 kernel32.dll 中，有兴趣的读者可以在 Windows 上用一个工具 depends.exe 打开 kernel32.dll，就可以看到这个 DLL 的导出函数表中有 ReadFile()。另一方面，在微软的 VC 开发环境(Visual Studio)中、以及 Win2k DDK 中，都有个“头文件”winbase.h，里面有 ReadFile()的接口定义：

```
WINBASEAPI
BOOL
WINAPI
ReadFile(
    IN HANDLE hFile,
    OUT LPVOID lpBuffer,
    IN DWORD nNumberOfBytesToRead,
    OUT LPDWORD lpNumberOfBytesRead,
    IN LPOVERLAPPED lpOverlapped
    );
```

函数名前面的关键词 WINAPI 表示这是个定义于 Win32 API 的函数。

在 ReactOS 的代码中同样也有 winbase.h，这是在目录 reactos/w32api/include 中：

```
BOOL WINAPI ReadFile(HANDLE, PVOID, DWORD, PDWORD, LPOVERLAPPED);
```

显然，这二者实际上是相同的(要不然就不兼容了)。当然，微软没有公开这个函数的代码，但是 ReactOS 为之提供了一个开源的实现，其代码在 reactos/lib/kernel32/file/rw.c 中。

```
BOOL STDCALL
ReadFile( HANDLE hFile,    LPVOID lpBuffer,    DWORD nNumberOfBytesToRead,
          LPDWORD lpNumberOfBytesRead,    LPOVERLAPPED lpOverLapped )
{
    ……

    errCode = NtReadFile(hFile,
                         hEvent,
                         NULL,
                         NULL,
                         IoStatusBlock,
                         lpBuffer,
                         nNumberOfBytesToRead,
                         ptrOffset,
                         NULL);

  ……
  return(TRUE);
}
```

我们在这里只关心 NtReadFile()，所以略去了别的代码。

如前所述，NtReadFile()是 Windows 的一个系统调用，内核中有个函数就叫 NtReadFile()，它的实现在 ntoskrnl.exe 中(这是 Windows 内核的核心部分)，这也可以用 depends.exe 打开 ntoskrnl.exe 察看。ReactOS 代码中对内核函数 NtReadFile()的定义在 reactos/include/ntos/zw.h 中，同样的定义也出现在 reactos/w32api/include/ddk/winddk.h 中：

```
NTSTATUS
STDCALL
NtReadFile(
    IN HANDLE FileHandle,
    IN HANDLE Event OPTIONAL,
    IN PIO_APC_ROUTINE UserApcRoutine OPTIONAL,
    IN PVOID UserApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    OUT PVOID Buffer,
    IN ULONG BufferLength,
    IN PLARGE_INTEGER ByteOffset OPTIONAL,
    IN PULONG Key OPTIONAL
    );
```

而相应的实现则在 reactos/ntoskrnl/io/rw.c 中。

表面上看这似乎挺正常，ReadFile()调用 NtReadFile()，reactos/ntoskrnl/io/rw.c 则为其提供了被调用的 NtReadFile()。可是仔细一想就不对了。这 ReadFile()是在用户空间运行的，而 reactos/ntoskrnl/io/rw.c 中的代码却是在内核中，是在系统空间。难道用户空间的程序竟能如此这般地直接调用内核中的函数吗？如果那样的话，那还要什么陷阱门、调用门这些机制呢？再说，编译的时候又怎样把它们连接起来呢？

这么一想，就可以断定这里面另有奥妙。仔细一查，原来还另有一个 NtReadFile()，在 msvc6/iface/native/syscall/Debug/zw.c 中：

```
__declspec(naked) __stdcall
NtReadFile(int dummy0, int dummy1, int dummy2)
{
    __asm {
        push ebp
        mov ebp, esp
        mov eax,152
        lea   edx, 8[ebp]
        int   0x2E
        pop  ebp
        ret   9
    }
}
```

原来，用户空间也有一个 NtReadFile()，正是这个函数在执行自陷指令“int 0x2e”。我们看一下这段汇编代码。这里面的 152 就是 NtReadFile()这个系统调用的调用号，所以当 CPU 自陷进入系统空间后寄存器 eax 持有具体的系统调用号。而寄存器 edx，在执行了 lea 这条指令以后，则持有 CPU 在调用这个函数前夕的堆栈指针，实际上就是指向堆栈中调用参数的起点。在进行系统调用时如何传递参数这个问题上，Windows 和 Linux 有着明显的差别。我们知道，Linux 是通过寄存器传递参数的，好处是效率比较高，但是参数的个数受到了限制，所以 Linux 系统调用的参数都很少，真有大量参数需要传递时就把它们组装在数据结构中，而只传递数据结构指针。而 Windows 则通过堆栈传递参数。读者在上面看到，ReadFile()在调用 NtReadFile()时有 9 个参数，这 9 个参数都被压入堆栈，而 edx 就指向堆栈中的这些参数的起点(地址最低处)。我们在这个函数中没有看到对通过堆栈传下来的参数有什么操作，也没有看到往堆栈里增加别的参数，所以传下来的 9 个参数被原封不动地传了下去(作为 int 0x2e 自陷的参数)。这样，当 CPU 自陷进入内核以后，edx 仍指向用户空间堆栈中的这些参数。当然，CPU 进入内核以后的堆栈是系统空间堆栈，而不是用户空间堆栈，所以需要用 copy_from_user()一类的函数把这些参数从用户空间拷贝过来，此时 edx 的值就可用作源指针。至于寄存器 ebp，则用作调用这个函数时的“堆栈框架”指针。

当内核完成了具体系统调用的操作，CPU 返回到用户空间时，下一条指令是“pop ebp”，即恢复上一层函数的堆栈框架指针。然后，指令“ret 9”使 CPU 返回到上一层函数，同时调整堆栈指针，使其跳过堆栈上的 9 个调用参数。在“正宗”的 x86 汇编语言中，用在 ret 指令中的数值以字节为单位，所以应该是“ret 24h”，而这里却是以 4 字节长字为单位，这显然是因为用了不同的汇编工具。

子程序的调用者可以把参数压入堆栈，通过堆栈把参数传递给被调用者。可是，当 CPU

从子程序返回时，由谁负责从堆栈中清除这些参数呢？显然，要么就是由调用者负责，要么就是由被调用者负责，这里需要有个约定，使得调用者和被调用者取得一致。在上面NtReadFile()这个函数中，我们看到是由被调用者负起了这个责任、在调整堆栈指针。函数代码前面的__stdcall 就说明了这一点。同样，在.h 文件中对 NtReadFile()的定义(申明)之前也加上了 STDCALL，也是为了说明这个约定。“Undocumented Windows 2000 Secrets”这本书中(p51-53)对类似的各种约定有一大段说明，读者可以参考。另一方面，在上面这个函数的代码中，函数的调用参数是 3 个而不是 9 个。但是看一下代码就可以知道这些参数根本就没有被用到，而调用者、即前面的 ReadFile()、也是按 9 个参数来调用 NtReadFile()的。所以，这里的三个参数完全是虚设的，有没有、或者有几个、都无关紧要，难怪代码中称之为“dummy”。

用户空间的这个 NtReadFile()向上代表着内核函数 NtReadFile()，向下则代表着想要调用内核函数 NtReadFile()的那个函数，在这里是 ReadFile()；但是它本身并不提供什么附加的功能，这样的中间函数称为“stub”。

当然，ReactOS 的这种做法很容易把读者引入迷茫。相比之下，Linux 的做法就比较清晰，例如应用程序调用的是库函数 write()，而内核中与之对应的函数则是 sys_write()。

那么为什么 ReactOS 要这么干呢？我只能猜测：

1） Windows 的源代码中就是这样，例如用 depends.exe 在 ntdll.dll 和 ntoskrnl.exe 中都可看到有名为 NtReadFile()的函数，而 ReactOS 的人就依葫芦画瓢。

2） 作为一条开发路线，ReactOS 可能在初期不划分用户空间和系统空间，所有的代码全在同一个空间运行，所以应用程序可以直接调用内核中的函数。这样，例如对文件系统的开发就可以简单易行一些。然后，到一些主要的功能都开发出来以后，再来划分用户空间和系统空间，并且补上如何跨越空间这一层。从 zw.c 这个文件在 native/syscall/Debug 目录下这个迹象看，ReactOS 似乎正处于走出这一步的过程中。

3） ReactOS 的作者们可能有意让它也可以用于嵌入式系统。嵌入式系统往往不划分用户空间和系统空间，而把应用程序和内核连接在同一个可执行映像中。这样，如果需要把代码编译成一个嵌入式系统，就不使用 stub；而若要把代码编译成一个桌面系统，则可以在用户空间加上 stub 并在内核中加上处理自陷指令“int 0x2e”的程序。

在 Windows 中，stub 函数 NtReadFile()在 ntdll.dll 中。实际上，所有 0x2e 系统调用的 stub 函数都在这个 DLL 中。显然，所有系统调用的 stub 函数具有相同的样式，不同的只是系统调用号和参数的个数，所以 ReactOS 用一个工具来自动生成这些 stub 函数。这个工具的代码在 msvc6/iface/native/genntdll.c 中，下面是一个片断：

```
void write_syscall_stub(FILE* out, FILE* out3, char* name, char* name2,
                char* nr_args, unsigned int sys_call_idx)
{
    int i;
    int nArgBytes = atoi(nr_args);

#ifdef PARAMETERIZED_LIBS
  ……
#else
```

```
    fprintf(out,"__asm__(\"\\n\\t.global _%s\\n\\t\"\n",name);
    fprintf(out,"\".global _%s\\n\\t\"\n",name2);
    fprintf(out,"\"_%s:\\n\\t\"\n",name);
    fprintf(out,"\"_%s:\\n\\t\"\n",name2);
#endif
    fprintf(out,"\t\"pushl\t%%ebp\\n\\t\"\n");
    fprintf(out,"\t\"movl\t%%esp, %%ebp\\n\\t\"\n");
    fprintf(out,"\t\"mov\t$%d,%%eax\\n\\t\"\n",sys_call_idx);
    fprintf(out,"\t\"lea\t8(%%ebp),%%edx\\n\\t\"\n");
    fprintf(out,"\t\"int\t$0x2E\\n\\t\"\n");
    fprintf(out,"\t\"popl\t%%ebp\\n\\t\"\n");
    fprintf(out,"\t\"ret\t$%s\\n\\t\");\n\n",nr_args);
    ……
}
```

代码中的'\t'表示 TAB 字符，读者阅读这段代码应该没有什么问题。这段代码根据 name、nr_args、sys_call_idx 等参数为给定系统调用生成 stub 函数的汇编代码。那么这些参数从何而来呢？在 ReactOS 代码的 reactos/tools/nci 目录下有个文件 sysfuncs.lst，下面是从这个文件中摘出来的几行：

```
NtAcceptConnectPort 6
NtAccessCheck 8
NtAccessCheckAndAuditAlarm 11
NtAddAtom 3
……
NtClose 1
……
NtReadFile 9
……
```

这里的 NtAcceptConnectPort 就是调用号为 0 的系统调用 NtAcceptConnectPort()，它有 6 个参数。另一个系统调用 NtClose()只有 1 个参数。而 NtReadFile()有 9 个参数，并且正好是这个表中的第 153 行，所以调用号是 152。

用户空间的程序一执行 int 0x2e，CPU 就自陷进入了系统空间。其间的物理过程这里就不多说了，有需要的读者可参考“情景分析”或其它有关资料。我这里就从 CPU 怎样进入 int 0x2e 的自陷处理程序说起。

像别的中断向量一样，ReactOS 在其初始化程序 KeInitExceptions()中设置了 int 0x2e 的向量，这个函数的代码在 reactos/ntoskrnl/ke/i386/exp.c 中：

```
VOID INIT_FUNCTION
KeInitExceptions(VOID)
/*
 * FUNCTION: Initalize CPU exception handling
```

```
 */
{
    ……
    set_trap_gate(0, (ULONG)KiTrap0, 0);
    set_trap_gate(1, (ULONG)KiTrap1, 0);
    set_trap_gate(2, (ULONG)KiTrap2, 0);
    set_trap_gate(3, (ULONG)KiTrap3, 3);
    ……
    set_system_call_gate(0x2d,(int)interrupt_handler2d);
    set_system_call_gate(0x2e,(int)KiSystemService);
}
```

显然，int 0x2e 的向量指向 KiSystemService()。

ReactOS 在其内核函数的命名和定义上也力求与 Windows 一致，所以 ReactOS 内核中也有前缀为 ke 和 ki 的函数。前缀 ke 表示属于“内核”模块。注意 Windows 所谓的“内核(kernel)”模块只是内核的一部分，而不是整个内核，这一点我以后在“漫谈 Wine”中还要讲到。而前缀 ki，则是指内核中与中断响应和处理有关的函数。KiSystemService()是一段汇编程序，其作用相当于 Linux 内核中的 system_call()，这段代码在 reactos/ntoskrnl/ke/i386/syscall.S 中。限于篇幅，我在这篇短文中就不详细讲解这个函数的全部代码了，而只是分段对一些要紧的关节作些说明。一般而言，能读懂 Linux 内核中 system_call()那段代码的读者应该能至少大体上读懂这个函数。

_KiSystemService:

```
    /*
     * Construct a trap frame on the stack.
     * The following are already on the stack.
     */
    // SS                                          + 0x0
    // ESP                                         + 0x4
    // EFLAGS                                      + 0x8
    // CS                                          + 0xC
    // EIP                                         + 0x10
    pushl $0                                       // + 0x14
    pushl %ebp                                     // + 0x18
    pushl %ebx                                     // + 0x1C
    pushl %esi                                     // + 0x20
    pushl %edi                                     // + 0x24
    pushl %fs                                      // + 0x28

    /* Load PCR Selector into fs */
    movw $PCR_SELECTOR, %bx
    movw %bx, %fs
```

```
/* Save the previous exception list */
pushl %fs:KPCR_EXCEPTION_LIST                    // + 0x2C

/* Set the exception handler chain terminator */
movl $0xffffffff, %fs:KPCR_EXCEPTION_LIST

/* Get a pointer to the current thread */
movl %fs:KPCR_CURRENT_THREAD, %esi
```

前面的一些指令主要是在保存现场，类似于 Linux 内核中的宏操作 SAVE_ALL。这里关键的一步是从%fs:KPCR_CURRENT_THREAD 这个地址取得当前线程的指针并将其存放在寄存器%esi 中。每个线程在内核中都有个 KTHREAD 数据结构，某种意义上相当于 Linux 内核中的“进程控制块”、即 task_struct。Windows 内核中也有“进程控制块”，但只是相当于把进程内各线程所共享的信息剥离了出来，而“线程控制块”则起着更重要的作用。所谓当前线程的指针，就是指向当前线程的 KTHREAD 数据结构的指针。当内核调度一个线程运行时，就将其 KTHREAD 数据结构的地址存放在%fs:KPCR_CURRENT_THREAD 这个地址中，而(CPU 在系统空间的)%fs 的值则又固定存放在 PCR_SELECTOR 这个地址中(定义为 0x30)。附带提一下，Win2k 内核把%fs:0 映射到线性地址 0xffdff000(见“Secrets”一书 p428)。

总之，从现在起，寄存器%esi 就指向了当前线程的 KTHREAD 数据结构。那么这一步对于系统调用为什么重要呢？我们看一下这个数据结构中的几个成分就可以明白：

```
typedef struct _KTHREAD
{
    /* For waiting on thread exit */
    DISPATCHER_HEADER       DispatcherHeader;       /* 00 */
    ……
    SSDT_ENTRY                  *ServiceTable;          /* DC */
    ……
    UCHAR                       PreviousMode;           /* 137 */
    ……
} KTHREAD;
```

每个成分后面的注释说明这个成分在数据结构中以字节为单位的相对位移，例如指针 ServiceTable 的相对位移就是 0xdc。事实上，这个指针正是我们此刻最为关注的，因为它直接与系统调用的函数跳转表有关。每个线程的这个指针都指向一个 SSDT_ENTRY 结构数组。既然每个线程都有这么个指针，就说明每个线程都可以有自己的 ServiceTable。不过，实际上每个线程的 ServiceTable 通常都指向同一个结构数组，我们等一下再来看这个结构数组，现在先往下看代码。

```
/* Save the old previous mode */
pushl %ss:KTHREAD_PREVIOUS_MODE(%esi)            // + 0x30

/* Set the new previous mode based on the saved CS selector */
movl 0x24(%esp), %ebx
```

```
        andl $1, %ebx
        movb %bl, %ss:KTHREAD_PREVIOUS_MODE(%esi)

        /* Save other registers */
        pushl %eax                                                  // + 0x34
        pushl %ecx                                                  // + 0x38
        pushl %edx                                                  // + 0x3C
        pushl %ds                                                   // + 0x40
        pushl %es                                                   // + 0x44
        pushl %gs                                                   // + 0x48
        sub $0x28, %esp                                             // + 0x70

#ifdef DBG
        ……
#else
        pushl 0x60(%esp) /* DebugEIP */                             // + 0x74
#endif
        pushl %ebp              /* DebugEBP */                      // + 0x78

        /* Load the segment registers */
        sti
        movw $KERNEL_DS, %bx
        movw %bx, %ds
        movw %bx, %es

        /* Save the old trap frame pointer where EDX would be saved */
        movl KTHREAD_TRAP_FRAME(%esi), %ebx
        movl %ebx, KTRAP_FRAME_EDX(%esp)

        /* Allocate new Kernel stack frame */
        movl %esp,%ebp

        /* Save a pointer to the trap frame in the TCB */
        movl %ebp, KTHREAD_TRAP_FRAME(%esi)

CheckValidCall:

#ifdef DBG
        ……
#endif
        /*
          * Find out which table offset to use. Converts 0x1124 into 0x10.
          * The offset is related to the Table Index as such: Offset = TableIndex x 10
          */
```

```
movl %eax, %edi
shrl $8, %edi
andl $0x10, %edi
movl %edi, %ecx

/* Now add the thread's base system table to the offset */
addl KTHREAD_SERVICE_TABLE(%esi), %edi
```

这里我们关注的是最后这一小段。首先，KTHREAD_SERVICE_TABLE(%esi)就是当前线程的 ServiceTable 指针。常数 KTHREAD_SERVICE_TABLE 定义为 0xdc：

```
#define KTHREAD_SERVICE_TABLE        0xDC
```

这跟前面 KTHREAD 数据结构的定义显然是一致的。

上面讲过，实际上一般情况下所有线程的 ServiceTable 指针都指向同一个结构数组，那就是 KeServiceDescriptorTable[ ]：

```
SSDT_ENTRY
__declspec(dllexport)
KeServiceDescriptorTable[SSDT_MAX_ENTRIES] = {
    { MainSSDT,   NULL,   NUMBER_OF_SYSCALLS,   MainSSPT },
    { NULL,       NULL,    0,    NULL    },
    { NULL,       NULL,    0,    NULL    },
    { NULL,       NULL,    0,    NULL    }
};
```

这个数组的大小一般是 4，但是只用了前两个元素。这里只用了第一个元素，这就是常规 Windows 系统调用的跳转表。

我以前曾经谈到，Windows 在发展的过程中把许多原来实现于用户空间的功能(主要是图形界面操作)移到了内核中，成为一个内核模块 win32k.sys，并相应地增加了一组“扩充系统调用”。这个数组的第二个元素就是为扩充系统调用准备的，但是在源代码中这个元素是空的，这是因为 win32k.sys 可以动态安装，安装了以后才把具体的数据结构指针填写进去。扩充系统调用与常规系统调用的区别是：前者的系统调用号均大于等于 0x1000，而后者则小于 0x1000。显然，内核需要根据具体的系统调用号来确定应该使用哪一个跳转表，或者说上述数组内的哪一个元素。每个元素的大小是 16 个字节，所以只要根据具体的系统调用号算出一个相对位移量，就起到了选择使用跳转表的作用。具体地，如果算得的位移量是 0，那就是使用常规跳转表，而若是 0x10 就是使用扩充跳转表。

上面的代码中正是这样做的。把系统调用号的副本(在%edi 中)右移 8 位，再跟 0x10 相与，就起到了这个效果。于是，指令“addl KTHREAD_SERVICE_TABLE(%esi), %edi”就使寄存器%edi 指向了应该使用的跳转表结构，即 SSDT_ENTRY 数据结构。代码的作者加了个注释，说是“把 0x1124 转换成 0x10”，其意思实际上是：“如果系统调用号是 0x1124，那么计算出来的相对位移是 0x10”；后面一句说的是“相对位移 = 数组下标乘上 0x10”。

SSDT_ENTRY 数据结构中的第三个成分，即相对位移为 8 之处是个整数，说明在函数

跳转表中有几个指针，也即所允许的最大系统调用号。对于常规系统调用，这个整数是NUMBER_OF_SYSCALLS，在 ReactOS 的代码中定义为 232，比 Win2K 略少一些。

我们继续往下看代码：

```
/* Get the true syscall ID and check it */
movl %eax, %ebx
andl $0x0FFF, %eax
cmpl 8(%edi), %eax

/* Invalid ID, try to load Win32K Table */
jnb KiBBTUnexpectedRange

/* Users's current stack frame pointer is source */
movl %edx, %esi

/* Allocate room for argument list from kernel stack */
movl 12(%edi), %ecx
movb (%ecx, %eax), %cl
movzx %cl, %ecx

/* Allocate space on our stack */
subl %ecx, %esp
```

正如代码中的注释所说，开始是检查系统调用号是否在合法范围之内，这里比较的对象显然就是 NUMBER_OF_SYSCALLS。

前面讲过，寄存器%edx 指向用户空间堆栈上的函数调用框架，实际上就是指向所传递的参数，现在把这个指针复制到%esi 中，这是在为从用户空间堆栈复制参数做准备。但是，光有复制的起点还不够，还需要有复制的长度(字节数)、即参数的个数乘 4，所以需要知道具体的系统调用有几个参数。这个信息保存在一个以系统调用号为下标的无符号字节数组中(所以每个系统调用的参数总长度不能超过 255 字节)，SSDT_ENTRY 数据结构中的第三个成分(相对位移为 12、或 0xc)就是指向这个数组的指针。对于常规系统调用，这个数组是MainSSPT。可想而知，这个数组的内容也应来自 sysfuncs.lst。代码中先让%ecx 指向MainSSPT，再以%eax 中的系统调用号与其相加，就使其指向了数组中的相应元素，而 movb 指令就把这个字节取了出来。所以，最后%ecx 持有给定系统调用的参数复制长度。从%esp 的内容中减去%ecx 的内容，就在系统空间堆栈上保留了若干字节，其长度等于参数复制长度，这样就为把参数从用户空间堆栈复制到系统空间堆栈做好了准备。再往下看：

```
/* Get pointer to function */
movl (%edi), %edi
movl (%edi, %eax, 4), %eax

/* Copy the arguments from the user stack to our stack */
shr $2, %ecx
movl %esp, %edi
```

```
    cld
    rep movsd

    /* Do the System Call */
    call *%eax
    movl %eax, KTRAP_FRAME_EAX(%ebp)

    /* Deallocate the kernel stack frame    */
    movl %ebp, %esp
```

前面，寄存器%edi 已经指向常规系统调用的 SSDT_ENTRY 数据结构，也就是指向了该数据结构中的第一个成分。SSDT_ENTRY 数据结构的第一个成分是个指针，指向一个函数指针数组。对于常规系统调用，这就是 MainSSDT。指令“movl (%edi), %edi”把%edi 所指处的内容赋给了%edi，使原来指向这个指针的%edi 现在指向了 MainSSDT。这也是个以系统调用号为下标的数组，其定义为：

```
SSDT MainSSDT[] = {
        { (ULONG)NtAcceptConnectPort },
        { (ULONG)NtAccessCheck },
        { (ULONG)NtAccessCheckAndAuditAlarm },
        ……
        { (ULONG)NtReadFile },
        ……
}
```

在我们这个例子中，指令“movl (%edi, %eax, 4), %eax”，即“把%edi 加相对位移为‘系统调用号乘 4’之处的内容装入%eax”，使%eax 指向了 NtReadFile()。然后就是把参数从用户空间堆栈拷贝到系统空间堆栈，注意%ecx 中的长度是以字节为单位的，所以要右移两位变成以长字为单位。

最后，指令“call *%eax”就使 CPU 进入了内核里面的 NtReadFile()，其代码在 reactos/ntoskrnl/io/rw.c 中。如果按 Linux 的规矩，这应该是 sys_NtReadFile()：

```
NTSTATUS STDCALL
NtReadFile (IN HANDLE FileHandle,
        IN HANDLE Event OPTIONAL,
        IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
        IN PVOID ApcContext OPTIONAL,
        OUT PIO_STATUS_BLOCK IoStatusBlock,
        OUT PVOID Buffer,
        IN ULONG Length,
        IN PLARGE_INTEGER ByteOffset OPTIONAL,
        IN PULONG Key OPTIONAL)
{
    ……
```

}

这个函数的调用界面与应用程序在用户空间进行这个系统调用时所遵循的界面完全相同，而应用程序压入用户空间堆栈的 9 个参数已经被拷贝到了系统空间堆栈中合适的位置上。于是，对于这个函数而言，就好像其调用者、在我们这个情景中是 ReadFile()、就在系统空间中一样。

回到上面的汇编代码中。当 CPU 从目标函数返回时，寄存器%eax 持有该函数的返回值，这是要返回给用户空间的，所以把它保存在堆栈框架中。

下面就是从内核返回到用户空间的过程，我把代码留给读者自己研究。不过需要给一点提示：

- 代码中的 APC 指“异步过程调用(Asynchronous Procedure Call)”，相当于 Linux 中的 Signal。
- Windows 把内核的运行状态分成若干级别。最高的一些级别是不允许硬件中断(不允许级别更低的硬件中断)；其次(2 级和 1 级)是不允许进程调度(但是允许硬件中断)，DPC(2 级，相当于 bh 函数)和 APC(1 级，相当于 signal)都应该在禁止调度的条件下执行；最低(0 级)就是允许进程调度。
- 从内核中也可以通过_KiSystemService()进行系统调用(不过要经过一个内核版本的 stub 函数)，所以代码中需要检测和区分 CPU 进入_KiSystemService()之前的运行模式，并且线程的 KTHREAD 数据结构中也有个成分 PreviousMode，用来保存这个信息。而 KTHREAD_PREVIOUS_MODE(%esi)就指向当前进程的 PreviousMode。

```
KeReturnFromSystemCall:

    /* Get the Current Thread */
    movl %fs:KPCR_CURRENT_THREAD, %esi

    /* Restore the old trap frame pointer */
    movl KTRAP_FRAME_EDX(%esp), %ebx
    movl %ebx, KTHREAD_TRAP_FRAME(%esi)

_KiServiceExit:

    /* Get the Current Thread */
    cli
    movl %fs:KPCR_CURRENT_THREAD, %esi

    /* Deliver APCs only if we were called from user mode */
    testb $1, KTRAP_FRAME_CS(%esp)
    je KiRosTrapReturn

    /* And only if any are actually pending */
    cmpb $0, KTHREAD_PENDING_USER_APC(%esi)
    je KiRosTrapReturn
```

```
    /* Save pointer to Trap Frame */
    movl %esp, %ebx

    /* Raise IRQL to APC_LEVEL */
    movl $1, %ecx
    call @KfRaiseIrql@4

    /* Save old IRQL */
    pushl %eax

    /* Deliver APCs */
    sti
    pushl %ebx
    pushl $0
    pushl $UserMode
    call _KiDeliverApc@12
    cli

    /* Return to old IRQL */
    popl %ecx
    call @KfLowerIrql@4

KiRosTrapReturn:

    /* Skip debug information and unsaved registers */
    addl $0x30, %esp                                    // + 0x48
    popl %gs                                            // + 0x44
    popl %es                                            // + 0x40
    popl %ds                                            // + 0x3C
    popl %edx                                           // + 0x38
    popl %ecx                                           // + 0x34
    popl %eax                                           // + 0x30

    /* Restore the old previous mode */
    popl %ebx                                           // + 0x2C
    movb %bl, %ss:KTHREAD_PREVIOUS_MODE(%esi)

    /* Restore the old exception handler list */
    popl %fs:KPCR_EXCEPTION_LIST                        // + 0x28

    /* Restore final registers from trap frame */
    popl %fs                                            // + 0x24
    popl %edi                                           // + 0x20
```

```
    popl %esi                                   // + 0x1C
    popl %ebx                                   // + 0x18
    popl %ebp                                   // + 0x14
    add $4, %esp                                // + 0x10

    /* Check if previous CS is from user-mode */
    testl $1, 4(%esp)

    /* It is, so use Fast Exit */
    jnz FastRet

    /*
     * Restore what the stub pushed, and return back to it.
     * Note that we were CALLed, so the first thing on our stack is the ret EIP!
     */
    pop %edx                                    // + 0x0C
    pop %ecx                                    // + 0x08
    popf                                        // + 0x04
    jmp *%edx

IntRet:
    iret

FastRet:

    /* Is SYSEXIT Supported/Wanted? */
    cmpl $0, %ss:_KiFastSystemCallDisable
    jnz IntRet
    ……
```

熟悉 Linux 的读者知道 CPU 在返回用户空间之前应该调用有关进程(线程)调度的函数，因而会期待在这段代码中也看到这样的操作，然而却没有看到。但是实际上确实有这样的操作，只不过是深藏在函数 KfLowerIrql()里面而已。

搞懂了这个函数的读者现在应该知道我们将要怎样做了。不过，我们的目标不是把 KiSystemService()与 Linux 的 system_call()堆积、并列在一起，而是要把前者溶入到后者中去。再说，即使照搬了 KiSystemService()，总不能因为这个程序调用了 KfLowerIrql()，就又照搬 KfLowerIrql()吧。如果按这样类推，那就势必要把整个 ReactOS 内核堆积到 Linux 内核中去了。由此可见，我们既要参考、借鉴 ReactOS 内核的实现，又要研究怎样把它融合、嫁接到 Linux 内核中去，这当然是一项富有挑战性的工作。