**2021/05/04_Windows32位内核_第9课_中断异常、使用未公开函数遍历进程**

**笔记本:** Windows32位内核
**创建时间:** 2021/5/4 星期二 15:01
**作者:** ileemi

---



# 通过页目录表读取进程内存

代码示例:

```
/*
强制读写进程内存
pDirBase: 目标进程的页目录表的地址
pAddress: 访问的地址
pBuf: 目标地址
nLen: 读取的字节数
*/
void MyReadProcessMemory(void* pDirBase,
  void* pAddress,
  void* pBuf,
  unsigned int nLen)
{

  void* pOldCR3; // 保存旧的CR3

  //PHYSICAL_ADDRESS PA = MmGetPhysicalAddress(NULL);
  //void* va = MmMapIoSpace(PA);
```

```
    // 保存旧的cr3后进行修改
    __asm
    {
      mov eax, cr3
      mov pOldCR3, eax
      mov eax, pDirBase
      mov cr3, eax
    }

    // pBuf -- 零环地址，防止出现进程问题
    RtlCopyMemory(pBuf, pAddress, nLen);

    // 恢复原CR3
    __asm
    {
      mov eax, pOldCR3
      mov cr3, eax
    }
}
```

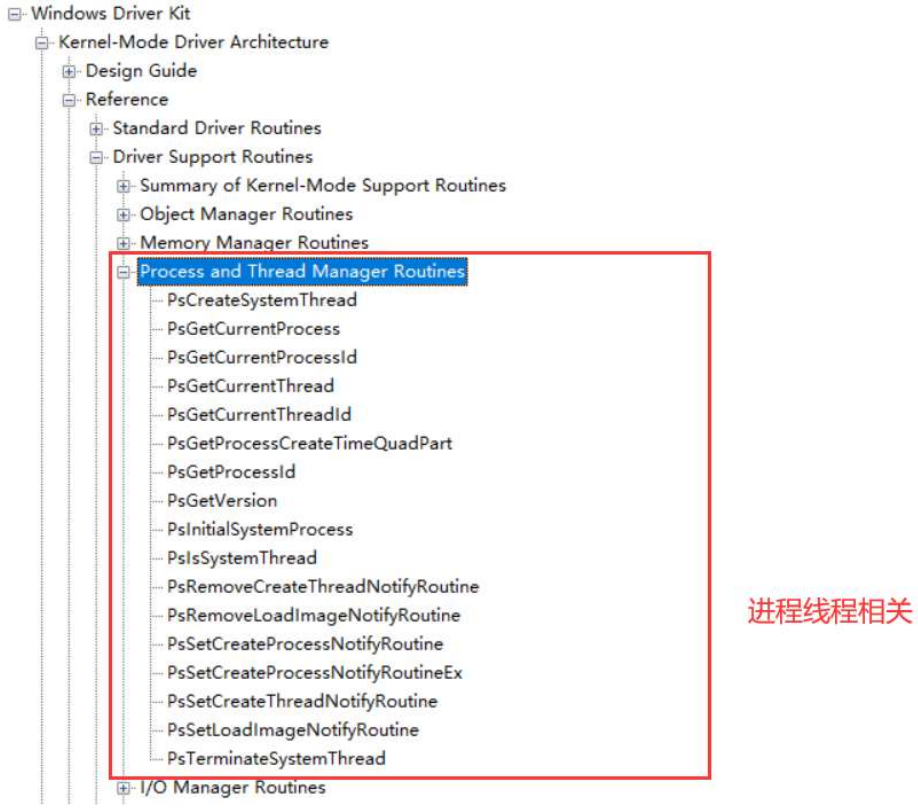
> 存在CR3遍历的问题，页目录表需要手工填写。正确的写法应该通过进程PID定位对应的页目录表，然后在进行内存操作。在内核中封装函数最后给 "NTSTATUS" 状态。

# 进程的遍历

读取进程内存，需要指定进程PID以及页目录项。

页目录表由操作系统管理，线程切换不需要切换页目录表（多个进程同属于一个进程），所以页目录表属于对应的进程中。通过上面的关系，可以通过进程对象定位其对应的页目录表的地址以及PID。

操作系统系统内核中获取进程相关信息的API：




在切换进程的时候会用到页目录表的地址，可通过搜索切换进程的内核API（没有公开的API，系统内部可以存在一些未公开的API函数）。

通过Windbg在指定模块中进行单词匹配搜索：

- Ring0：`x nt!*swap*`、`x nt!*process*`
- Ring3：`x ntdll!*swap*`、`x ntdll!*process*`

```
76f41a62 cc              int     3
0:000> x ntdll!*swap*
76ef5174          ntdll!RtlReleaseSwapReference (void)
76ef5334          ntdll!RtlAcquireSwapReference (void)
76ef7836          ntdll!SwapSplayLinks (void)
76f4e970          ntdll!RtlUshortByteSwap (@RtlUshortByteSwap@4)
76f4e950          ntdll!RtlUlonglongByteSwap (@RtlUlonglongByteSwap@8)
76f4e940          ntdll!RtlUlongByteSwap (@RtlUlongByteSwap@4)
0:000> x ntdll!*process*
76eba107          ntdll!RtlpProcessIFEOKeyFilter (void)
76f97fc0          ntdll!PsspDumpObject_Process (void)
76ef645a          ntdll!LdrpProcessInitializationComplete (void)
76eef1c0          ntdll!RtlSetThreadSubProcessTag (void)
76eea4b3          ntdll!LdrpProcessDetachNode (void)
76eb8307          ntdll!RtlCreateProcessParametersInternal (void)
76eed5d0          ntdll!RtlExitUserProcess (void)
76ee05ec          ntdll!LdrpProcessMappedModule (void)
76eebd70          ntdll!RtlWow64GetProcessMachines (void)
```

## API 分析

**PsGetCurrentProcessId** -- 获取进程PID，从对应的进程对象中获取。通过分析该函数，必然可以找到这个进程对象，通过进程对象可以定位对应的页目录表地址。

> **PsGetCurrentThread** 函数在内核中的实现代码：线程对象 = FS 首地址 + 偏移（124H），FS 是一个指向 _KPCR 的结构体（处理器控制区）。KPCR_BASE 每个CPU有一个，固定地址为：0xFFDFF000

```
kd> u PsGetCurrentThread
nt!PsGetCurrentThread:
8052890c 64a124010000    mov     eax,dword ptr fs:[00000124h]
80528912 c3              ret
80528913 cc              int     3
```

> **PsGetCurrentProcess** 函数在内核中的实现代码：进程对象在 _EPROCESS 中。

```
kd> u PsGetCurrentProcess
nt!IoGetCurrentProcess:
804ef608 64a124010000    mov     eax,dword ptr fs:[00000124h]
804ef60e 8b4044          mov     eax,dword ptr [eax+44h]
804ef611 c3              ret
804ef612 cc              int     3
804ef613 cc              int     3
804ef614 cc              int     3
```

进程对象 = 线程对象 + 偏移44H

> **PsGetCurrentProcessId** 函数在内核中的实现代码：

```
kd> u PsGetCurrentProcessId
nt!PsGetCurrentProcessId:
80528650 64a124010000    mov     eax,dword ptr fs:[00000124h]
80528656 8b80ec010000    mov     eax,dword ptr [eax+1ECh]
8052865c c3              ret
8052865d cc              int     3
8052865e cc              int     3
8052865f cc              int     3
```

进程PID = 线程对象 + 偏移1ECH

通过 dt _ETHREAD 查看线程结构体，验证上面的偏移地址：

```
kd> dt _CLIENT_ID
ntdll!_CLIENT_ID
   +0x000 UniqueProcess    : Ptr32 Void
   +0x004 UniqueThread     : Ptr32 Void
```

+0x1ec Cid : _CLIENT_ID

通过线程获取进程对象

可以看出其是通过线程获取进程对象（线程属于进程）。使用 dt _EPROCESS -b 命令可以展开结构体中的数据成员以及子结构。

每个进程都有自己的 _EPROCESS，同时都有自己的页目录表：

```
kd> dt _EPROCESS -b
ntdll!_EPROCESS
   +0x000 Pcb              :  KPROCESS
      +0x000 Header           : _DISPATCHER_HEADER
         +0x000 Type             : UChar
         +0x001 Absolute         : UChar
         +0x002 Size             : UChar
         +0x003 Inserted         : UChar
         +0x004 SignalState      : Int4B
         +0x008 WaitListHead     : _LIST_ENTRY
            +0x000 Flink            : Ptr32
            +0x004 Blink            : Ptr32
      +0x010 ProfileListHead  : _LIST_ENTRY
         +0x000 Flink            : Ptr32
         +0x004 Blink            : Ptr32
      +0x018 DirectoryTableBase : Uint4B
      +0x020 LdtDescriptor    : _KGDTENTRY
         +0x000 LimitLow         : Uint2B
         +0x002 BaseLow          : Uint2B
         +0x004 HighWord         : __unnamed
            +0x000 Bytes            : __unnamed
```

> 进程的遍历，通过 _EPROCESS 中的 ActiveProcessLinks 进行遍历，其是一个双向链表，指向下一个进程。

```
+0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x000 Flink              : Ptr32
   +0x004 Blink              : Ptr32
```

> _KPCR + 0x120 = _KPRCB + 0x4 --> 就可以得到当前正在运行的线程

```
   +0x0d4 InterruptMode    : Uint4B
   +0x0d8 Spare1           : UChar
   +0x0dc KernelReserved2  : [17] Uint4B
   +0x120 PrcbData         : _KPRCB
kd> dt _KPRCB
ntdll!_KPRCB
   +0x000 MinorVersion     : Uint2B
   +0x002 MajorVersion     : Uint2B
   +0x004 CurrentThread    : Ptr32 _KTHREAD
   +0x008 NextThread       : Ptr32 _KTHREAD
   +0x00c IdleThread       : Ptr32 _KTHREAD
```

_KPCR

_ETHREAD 第一个成员就是 _KTHREAD

> 经过分析，通过当前线程可以获取当前进程，通过当前进程就可以获取进程链表，即可进行遍历。也就是说可以通过 "FS" 寄存器遍历进程。

```
   +0x034 ApcState         : _KAPC_STATE
      +0x000 ApcListHead      : _LIST_ENTRY
         +0x000 Flink            : Ptr32
         +0x004 Blink            : Ptr32
      +0x010 Process          : Ptr32
      +0x014 KernelApcInProgress : UChar
      +0x015 KernelApcPending : UChar
      +0x016 UserApcPending   : UChar
   +0x04c ContextSwitches  : Uint4B
kd> dt _EPROCESS
ntdll!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
```

_KTHREAD

## 代码实现

> 使用偏移（硬编码）来获取进程相关的信息（存在操作系统版本兼容性问题）。早期的做法是在遍历进程前先获取系统的版本号（PsGetVirsion），根据系统版本做相应的处理（返回所需结构体成员的偏移值）。

```
  PEPROCESS Process = NULL;          // 保存进程对象
  __asm
    {
        mov eax, fs: [124h]      // ETHREAD
        mov eax, [eax + 44h]     // EPROCESS
        mov Process, eax
    }
```

> 采用硬编码编写内核程序存在操作系统版本问题，可使用操作系统公开的内核API：

```
PEPROCESS Process = NULL;          // 保存进程对象
Process = PsGetCurrentProcess(); // 等价上面三行汇编代码的结果
```

> 防止使用公开的内核API被Hook，还可通过内核API的代码特征获取偏移：

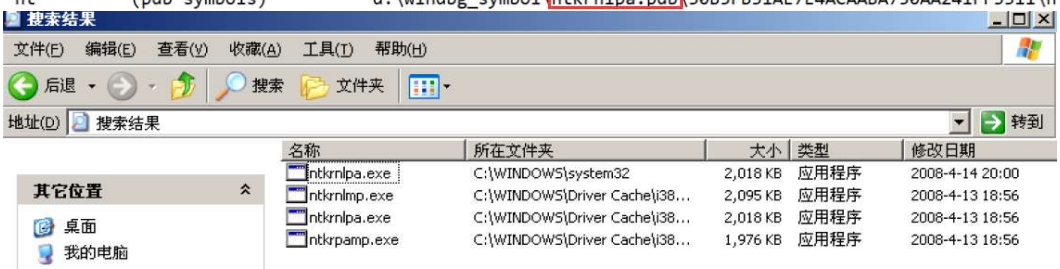
```
CHAR* pCode = PsGetCurrentProcess; // 获取内核API的地址
//CHAR* pCode = MmGetSystemAddress("PsGetCurrentProcess"); // 动态获取导出函数
// 804ef608 64a124010000        mov eax,dword ptr fs:[00000124h]
ULONG offse1 = *(ULONG*)(pCode + 2);
// 804ef60e 8b4044                      mov eax,dword ptr [eax+44h]
ULONG offse2 = *(pCode + 8); // 2 + 6 -- 上一条指令的长度
```

## 使用未公开的内核函数

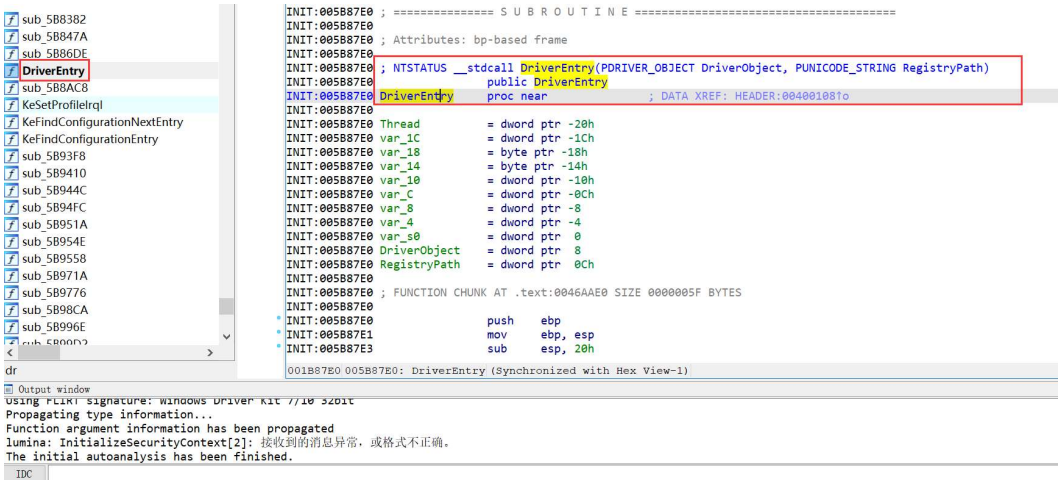
> 对于公开的内核函数没有使用偏移，也就是通过未公开函数进行操作。如果公开的内核函数内部调用了未公开内核函数，可通过公开函数找到未公开函数进行特征搜索。如果运气好，未公开的内核函数中使用了偏移，整个操作系统的内核代码在 "nt.dll" 模块中（ntkrpamp.exe文件）。

ntkr*.exe 有多个，内核代码不固定，受物理地址扩展、单核、多核的影响（其对应的操作系统代码页就不相同）。__asm cpuid // 操作系统判断CPU的型号，加载对应的内核代码。
当前被调试的操作系统为单核开启PAE，对应的系统代码为：ntkrnlpa.exe

```
module name
ntdll      (pdb symbols)          d:\windbg_symbol\ntdll.pdb\1751003260CA42598C0FB326585000ED2\ntdl
nt         (pdb symbols)          d:\windbg_symbol\ntkrnlpa.pdb\30B5FB31AE7E4ACAABA750AA241FF3311\n
```

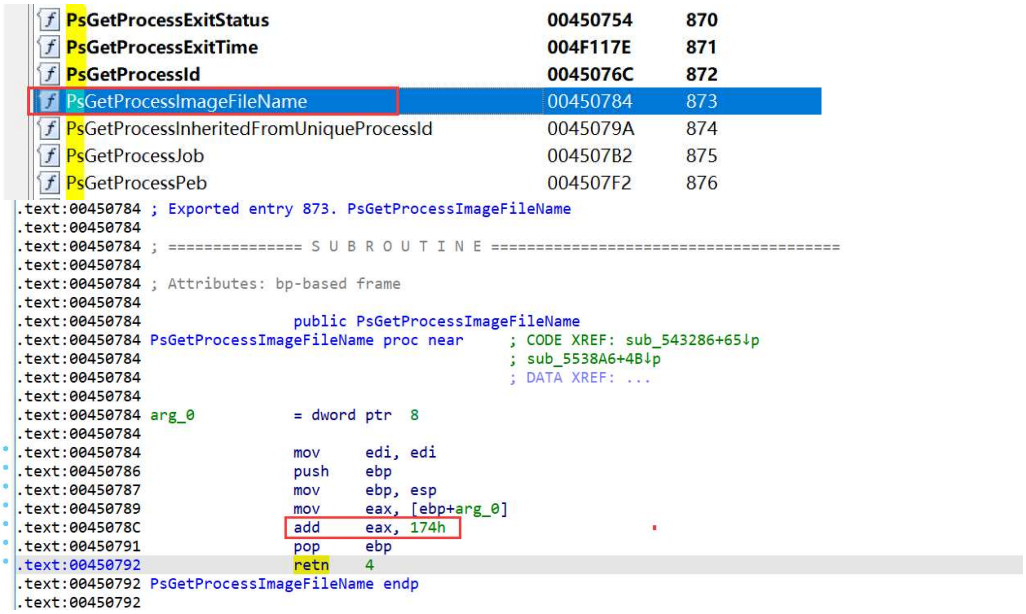



ntkrnlpa.exe 可执行文件就是一个驱动程序，可以看成是一个dll，其中的API也就相当于导出函数（官方没有对应文档说明的就是未公开的函数）。




可以在 ntkr*.exe（操作系统内核代码）中，搜索内部的导出函数（查看其参数，判断返回值以及调用约定）。

例如：获取进程名




通过 "PsGetProcessImageFileName" 即可替换硬编码来获取进程名，在驱动中使用需要向前声明，代码示例：

```
// 声明
UCHAR* PsGetProcessImageFileName(PEPROCESS Process);

//pImageFileName = (char*)curProcess + 0x174;
pImageFileName = PsGetProcessImageFileName(Process);
```

> 提高内核代码的通用性，尽量将 "硬编码" 转换为公开的内核API或者未公开的内核API。

# 完整代码示例

```
#include <ntddk.h>
```

```
UCHAR* PsGetProcessImageFileName(PEPROCESS Process);

void DriverUnload(struct _DRIVER_OBJECT* DriverObject) {
  UNREFERENCED_PARAMETER(DriverObject);
  DbgPrint("[51asm]  DriverUnload\n");
}

// 枚举进程页目录
/*
页目录地址由操作系统维护，进程页由操作系统创建
*/
NTSTATUS EnumProcessDirBase() {
  PEPROCESS Process = NULL;      // 保存进程对象
  PEPROCESS curProcess = NULL;  // 保存当前进程（记录链表起点）
  HANDLE Pid; // 保存进程PID
  UCHAR* pImageFileName = NULL; // 保存进程名

  DbgPrint("[51asm]  EnumProcessDirBase Begin\n");

  /*
  * 数据关系：进程对象  PID  NAME DIRBASE
     线程对象    TID  PRO  FK(PROCESS)
  */

  //__asm
  //{
  //  mov eax, fs: [124h]   // ETHREAD
  //  mov eax, [eax + 44h]  // EPROCESS
  //  mov Process, eax
  //}
  //

  /*
  _KPCR + 0x120(_KPRCB) +
          0x4(_RTHREAD) +
          0x34(ApcState) +
          0x10(Process)
  */
  Process = PsGetCurrentProcess();
  //// 通过代码特征获取偏移
  //CHAR* pCode = PsGetCurrentProcess; // 获取内核API的地址
  //CHAR* pCode = MmGetSystemAddress("PsGetCurrentProcess"); // 动态获取
导出函数
  //// 804ef608 64a124010000    mov eax,dword ptr fs:[00000124h]
  //ULONG offse1 = *(ULONG*)(pCode + 2);
  //// 804ef60e 8b4044          mov eax,dword ptr [eax+44h]
  //ULONG offse2 = *(pCode + 8); // 2 + 6 -- 上一条指令的长度
```

```
  //未公开内核API使用了偏移
  curProcess = Process;
  do
  {
    // 获取进程PID
    Pid = PsGetProcessId(curProcess);
    // 通过 _EPROCESS 的偏移获取进程PID  -- 使用偏移来获取进程相关的信息存在操作系统兼容性问题
    //dwPid = *(ULONG*)((char*)curProcess + 0x84);
    //pImageFileName = (char*)curProcess + 0x174;
    pImageFileName = PsGetProcessImageFileName(curProcess);

    DbgPrint("[51asm] _EPROCESS:%p PID:%u ImageFileName:%s\n",
      curProcess, Pid, pImageFileName);

    // 获取下一个进程的 _EPROCESS -- ActiveProcessLinks 链表位置
    curProcess = (PEPROCESS)*(void**)((char*)curProcess + 0x88);
    // 减去 0x88，获取  _EPROCESS 首地址
    curProcess = (PEPROCESS)((char*)curProcess - 0x88);

    //__asm cpuid //判断CPU的型号
    //__asm lidt
    //__asm sidt
   }
  while (Process != curProcess);

  DbgPrint("[51asm]  EnumProcessDirBase End\n");
  return STATUS_SUCCESS;
}


NTSTATUS
DriverEntry(
  __in struct _DRIVER_OBJECT* DriverObject,
  __in PUNICODE_STRING  RegistryPath
)
{
  char ch = 0;
  UNREFERENCED_PARAMETER(DriverObject);
  UNREFERENCED_PARAMETER(RegistryPath);

  EnumProcessDirBase();

  DbgPrint("[51asm] DriverEntry ch:%02x\n", ch);
  DriverObject->DriverUnload = DriverUnload;
```

```
    return 0;
}
```

# 任务管理

任务是处理器可以分派、执行和挂起的一个工作单元。它可以用于执行程序、任务或进程、操作系统服务实用程序、中断或异常处理程序，或内核或执行实用程序。

任务管理：线程切换最大的问题就是保存环境，恢复环境。由硬件自动保存环境切换环境效率会很快。

任务由两个部分组成：任务执行空间和任务状态段(TSS)。任务执行空间由一个代码段、一个堆栈段和一个或多个数据段（见图7-1）组成。如果操作系统或执行人员使用处理器的特权级别保护机制，则任务执行空间还将为每个权限级别提供一个单独的堆栈。

TSS指定构成任务执行空间的段，并提供任务状态信息的存储位置。在多任务处理系统中，TSS还提供了一种链接任务的机制。任务由其TSS的段选择器标识。当任务加载到处理器中进行执行时，TSS的段选择器、基本地址、限制和段描述符属性将加载到任务寄存器中（参见第2.4.4节 "任务寄存器（TR) "）。

如果为该任务实现了分页，则该任务所使用的页面目录的基本地址将被加载到控制寄存器CR3中。

任务寄存器（TR) 指定段选择子。指向一个内存地址，保存寄存器的值。当线程切换的时候，CPU会将此时所有的寄存器环境保存到TR寄存器指向的内存地址中。线程切换不旦需要切换寄存器也可能需要独立的一个栈（任务在哪环上运行，就切换哪环的栈）。

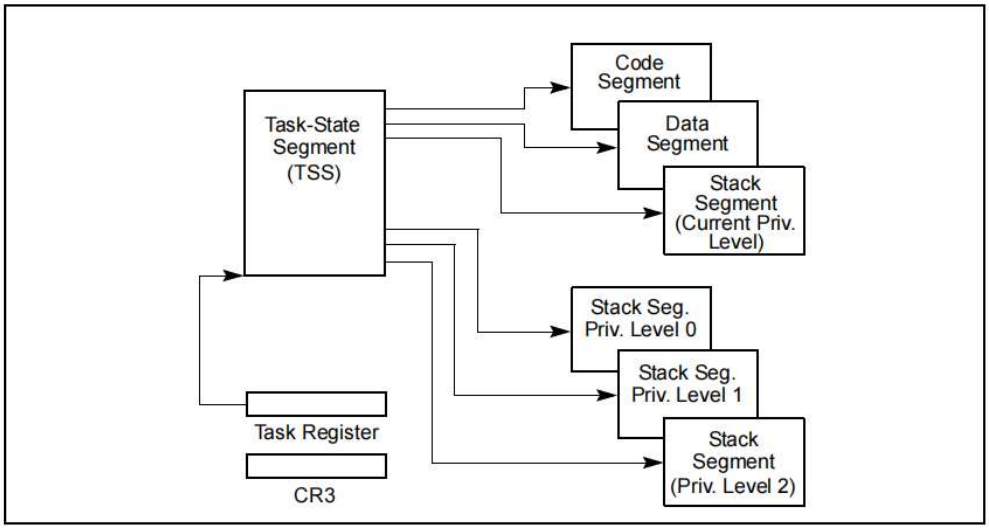



Figure 7-1. Structure of a Task

寄存器环境保存在 "_ETHREAD" --> "_KTHREAD" --> "_KTRAP_FRAME" 中，如下图所示：

```
   +0x130 Win32Thread      : Ptr32 Void
   +0x134 TrapFrame        : Ptr32 _KTRAP_FRAME
   +0x138 ApcStatePointer  : [2] Ptr32 _KAPC_STATE
   +0x140 PreviousMode     : Char
   +0x141 EnableStackSwap  : UChar
kd> dt _KTRAP_FRAME
ntdll!_KTRAP_FRAME
   +0x000 DbgEbp           : Uint4B
   +0x004 DbgEip           : Uint4B
   +0x008 DbgArgMark       : Uint4B
   +0x00c DbgArgPointer    : Uint4B
   +0x010 TempSegCs        : Uint4B
   +0x014 TempEsp          : Uint4B
   +0x018 Dr0              : Uint4B
   +0x01c Dr1              : Uint4B
   +0x020 Dr2              : Uint4B
   +0x024 Dr3              : Uint4B
   +0x028 Dr6              : Uint4B
   +0x02c Dr7              : Uint4B
   +0x030 SegGs            : Uint4B
   +0x034 SegEs            : Uint4B
   +0x038 SegDs            : Uint4B
```

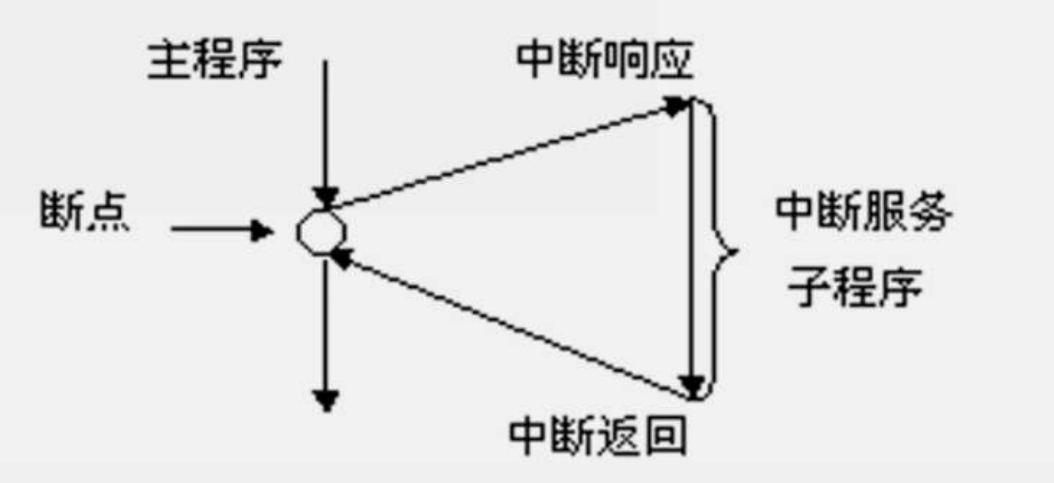
修改 "_KTRAP_FRAME" 结构体中成员的值也就修改了对应进程的寄存器环境。

TSS：任务状态段，也称为系统描述符。

# 中断异常

3章 - 6节

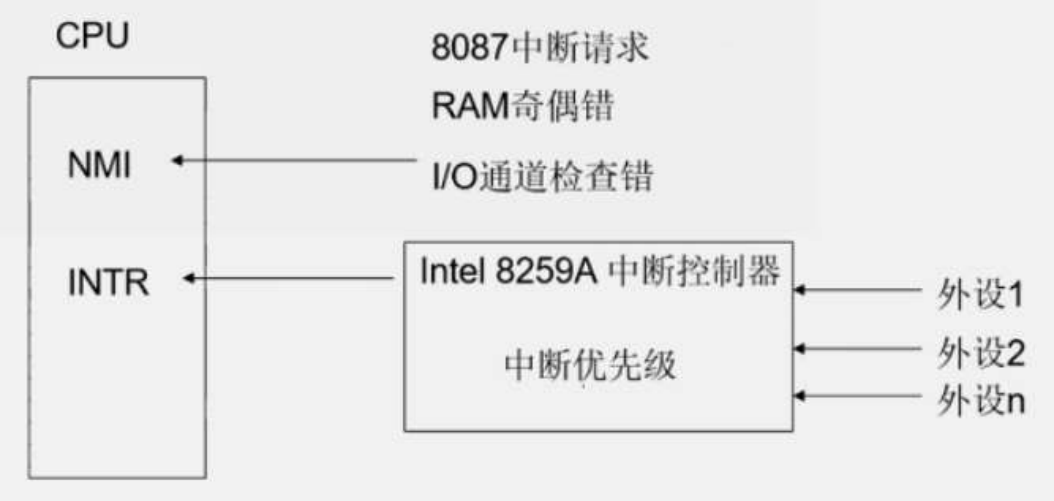
中断请求：IRQ
中断处理例程（ISR）：为中断提供对应的回调函数。

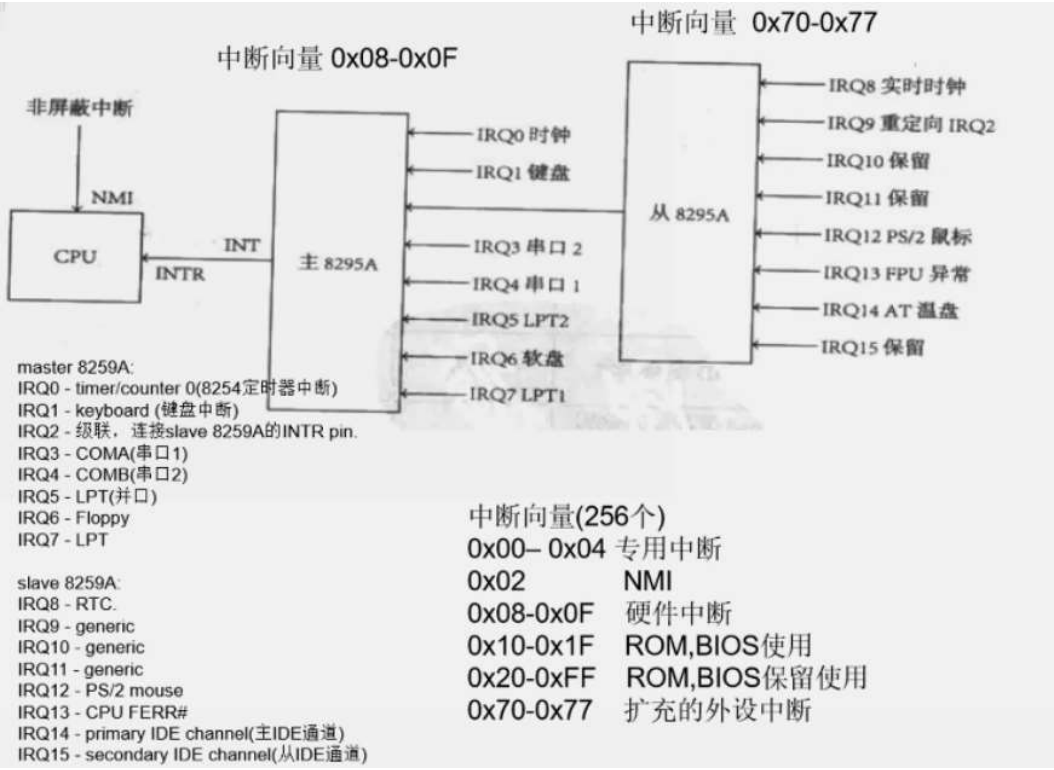



## 中断机制

CPU中的两个针脚：
CLI（clear interrupt）：指令清除标志寄存器。
NMI（I/O通道检查错）：不可以屏蔽中断，需要立即处理。通过NMI引脚发给CPU。
INTR（Inter8295A中断控制器）：通过 "cli" 可以屏蔽中断信号，控制中断优先级。

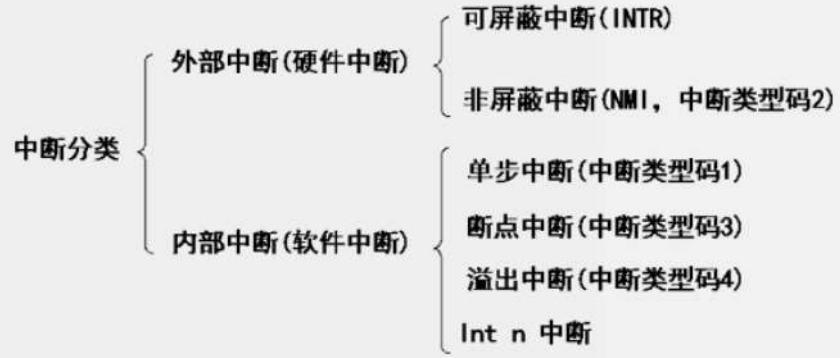
请求通过INTR引脚发送给CPU。




## 中断分类

外部中断（硬件中断）：硬件产生叫做中断，软件产生叫做异常。
内部中断（软件中断）




int n n占一个字节，总共有256个中断向量号，表示中断类型码
8086CPU在收到中断信息后，中断过程：
1.(从中断信息)取得中断类型码
2.标志寄存器入栈
3.设置标志寄存器TF=0 IF=0
4.CS入栈，IP入栈
5.jmp far ptr [n*4+2]:[n*4]
ISR 写法：
1.保存用到的寄存器
2.处理中断
3.恢复用到的寄存器
4.iret指令返回

## 中断向量表

Table 6-1. Protected-Mode Exceptions and Interrupts

| Vector | Mnemonic | Description | Type | Error Code | Source |
|---|---|---|---|---|---|
| 0 | #DE | Divide Error | Fault | No | DIV and IDIV instructions. |
| 1 | #DB | Debug Exception | Fault/ Trap | No | Instruction, data, and I/O breakpoints; single-step; and others. |
| 2 | — | NMI Interrupt | Interrupt | No | Nonmaskable external interrupt. |
| 3 | #BP | Breakpoint | Trap | No | INT3 instruction. |
| 4 | #OF | Overflow | Trap | No | INTO instruction. |
| 5 | #BR | BOUND Range Exceeded | Fault | No | BOUND instruction. |
| 6 | #UD | Invalid Opcode (Undefined Opcode) | Fault | No | UD instruction or reserved opcode. |
| 7 | #NM | Device Not Available (No Math Coprocessor) | Fault | No | Floating-point or WAIT/FWAIT instruction. |
| 8 | #DF | Double Fault | Abort | Yes (zero) | Any instruction that can generate an exception, an NMI, or an INTR. |
| 9 | | Coprocessor Segment Overrun (reserved) | Fault | No | Floating-point instruction.[1] |
| 10 | #TS | Invalid TSS | Fault | Yes | Task switch or TSS access. |
| 11 | #NP | Segment Not Present | Fault | Yes | Loading segment registers or accessing system segments. |
| 12 | #SS | Stack-Segment Fault | Fault | Yes | Stack operations and SS register loads. |
| 13 | #GP | General Protection | Fault | Yes | Any memory reference and other protection checks. |
| 14 | #PF | Page Fault | Fault | Yes | Any memory reference. |

Table 6-1. Protected-Mode Exceptions and Interrupts (Contd.)

| Vector | Mnemonic | Description | Type | Error Code | Source |
|---|---|---|---|---|---|
| 15 | — | (Intel reserved. Do not use.) | | No | |
| 16 | #MF | x87 FPU Floating-Point Error (Math Fault) | Fault | No | x87 FPU floating-point or WAIT/FWAIT instruction. |
| 17 | #AC | Alignment Check | Fault | Yes (Zero) | Any data reference in memory.[2] |
| 18 | #MC | Machine Check | Abort | No | Error codes (if any) and source are model dependent.[3] |
| 19 | #XM | SIMD Floating-Point Exception | Fault | No | SSE/SSE2/SSE3 floating-point instructions[4] |
| 20 | #VE | Virtualization Exception | Fault | No | EPT violations[5] |
| 21 | #CP | Control Protection Exception | Fault | Yes | RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at target of an indirect call or jump. |
| 22-31 | — | Intel reserved. Do not use. | | | |
| 32-255 | — | User Defined (Non-reserved) Interrupts | Interrupt | | External interrupt or INT *n* instruction. |

## IDTR与IDT的关系

因为中断只传递到处理器核心一次，所以配置错误的IDT可能会导致不完整的中断处理和/或中断交付的阻塞。需要遵循IA-32设置IDTR基础/限制/访问字段和门描述符中的每个字段的IA-32架构规则。英特尔64的架构也是如此。这包括通过GDT或LDT隐式引用目标代码段并访问堆栈。

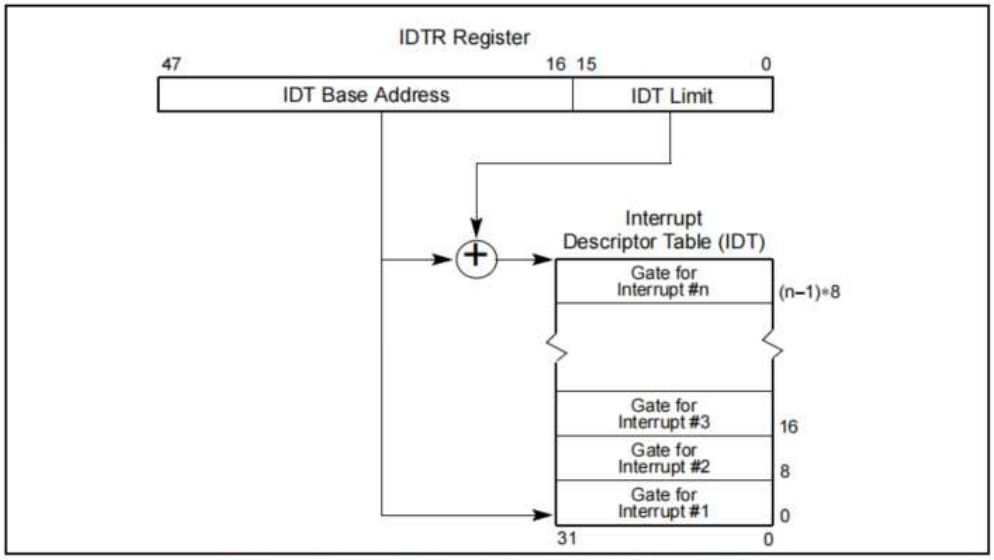
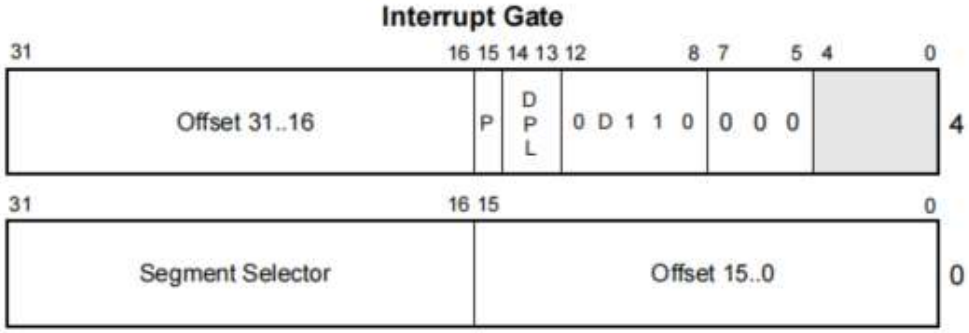



Figure 6-1. Relationship of the IDTR and IDT

## 中断描述符表的格式

idtr（48位）：寄存器 idtr 的改、获取可通过 lidt 和 sidt 指令进行操作。

idt表（中断描述符表）：CPU手册中称为门描述符，属于系统描述符。




> dq idtr 可获取中断表项，idtr表的解析如下：

```
kd> dq idtr
8003f400 80538e00`0008f19c 80538e00`0008f314
8003f410 00008500`0058113e 8053ee00`0008f6e4 // 第3项负责断点异常
8003f420 8053ee00`0008f864 80538e00`0008f9c0
...
// 异常处理完毕必须使用 iret 返回，表示异常处理完毕。
// 8053efd1 cf              iretd -- d 需要平衡堆栈
idtr表 第0项的解析：
80538e00 - 0008f19c

offset：8053 f19c
Segment Selector：0008
8e00 -- 1000 1110 0000 0000
P：1
DPL：00 -- Ring0
```

```
函数地址：cs : ip -- 0008 : 8053f19c
```

> 在Windows中代码段的基址为0，所以偏移 8053f19c 也就是函数首地址：

```
kd> dg 8
                                    P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0008 00000000 ffffffff Code RE Ac 0 Bg Pg P  Nl 00000c9b
kd> u 8053f19c
nt!KiTrap00:
8053f19c 6a00            push    0
8053f19e 66c74424020000  mov     word ptr [esp+2],0
8053f1a5 55              push    ebp
8053f1a6 53              push    ebx
8053f1a7 56              push    esi
8053f1a8 57              push    edi
8053f1a9 0fa0            push    fs
8053f1ab bb30000000      mov     ebx,30h
```

> !idt -a 命令在Windbg中用来显示所有的中断描述符表项（256项）：

```
f3:   8053e2cc nt!KiUnexpectedInterrupt195
f4:   8053e2d3 nt!KiUnexpectedInterrupt196
f5:   8053e2da nt!KiUnexpectedInterrupt197
f6:   8053e2e1 nt!KiUnexpectedInterrupt198
f7:   8053e2e8 nt!KiUnexpectedInterrupt199
f8:   8053e2ef nt!KiUnexpectedInterrupt200
f9:   8053e2f6 nt!KiUnexpectedInterrupt201
fa:   8053e2fd nt!KiUnexpectedInterrupt202
fb:   8053e304 nt!KiUnexpectedInterrupt203
fc:   8053e30b nt!KiUnexpectedInterrupt204
fd:   8053e312 nt!KiUnexpectedInterrupt205
fe:   8053e319 nt!KiUnexpectedInterrupt206
ff:   8053e320 nt!KiUnexpectedInterrupt207
```

> uf addr 命令在Windbg中用来显示当前函数地址开始的所有汇编代码。

> eb addr cf -- 修改idtr表第三项（中断处理函数）一个字节为 iretd，中断处理函数直接返回，使系统断点功能失效。

```
kd> eb 8053f6cf cf
kd> u 8053f6cf
nt!V86_kit3_a+0x13:
8053f6cf cf              iretd
8053f6d0 006689          add     byte ptr [esi-77h],ah
8053f6d3 45              inc     ebp
8053f6d4 50              push    eax
```

bl 查看所有断点
bc idx 删除指定下标的断点

键盘中断：idt 表的第93项

> dq idtr + 93 * 8
> u 函数首地址偏移
> bp addr -- 下断，在调试的操作系统中按键盘，查看断点是否断下。