# 中断、异常和系统调用

中断处理和系统调用模块是操作系统与硬件、用户程序交互的接口。当程序执行时发生异常或机器硬件发出中断请求后，操作系统暂时挂起正在运行的程序，转而去对中断或异常进行处理，在中断上下文中调用相应的异常处理函数或硬件驱动函数，等到中断或异常处理结束后再恢复原来程序的执行。

系统调用是一种进程主动陷入内核，使用内核提供的一些服务的方式。用户程序通过系统调用接口请求操作系统为其服务，在系统调用上下文中，进程运行于核心态下执行操作系统内核代码完成用户请求。

在Unix V6++的设计中，由于体系结构不同，中断模块的设计会与原操作系统有较大的差异，保护模式下的IDT表代替了原有的中断向量表，由于CPU寄存器的构造不同，中断隐指令所保存的数据也有所不同，但中断流程是基本保持一致的，即a.硬件保护现场 b.软件保护现场 c.执行相应中断处理函数 d.软件恢复现场 e.硬件恢复现场。在Unix V6++的设计中仍将被保留，但保存与中断处理函数本身将会有较大变化。

## I386保护模式下的中断

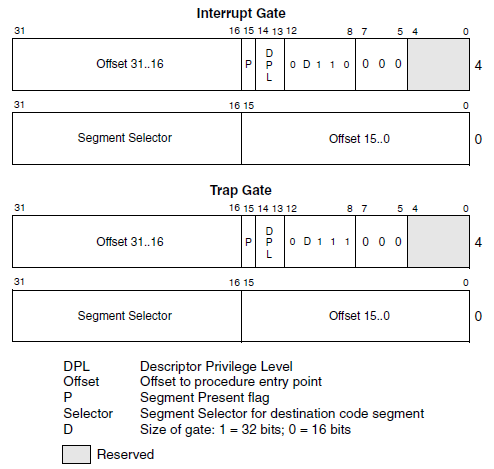
### 中断描述符表(IDT)

中断描述符表(Interrupt Descriptor Table, IDT)的作用是将中断向量号与对应的中断处理程序联系起来。中断描述符表本质上也是由一系列描述符构成的一个数组，每个描述符都是8字节长度。

中断描述符表由中断描述符表地址寄存器(IDTR)定位于内存任意位置。IDTR长48位，其中32位的基地址规定IDT的基地址，16位的界限规定IDT的段界限。由于I386体系结构下最多只支持256个中断/异常，所以IDT表最大长度是2K，IDTR段界限为7FFH。

中断描述符表的表项是一种称为门的数据结构，分为任务门，中断门，陷入门，调用门四种。在Unix V6++中，并没有用到任务门和调用门，只用到中断门和陷入门。下图是中断门和陷入门描述符的定义(图6.1)。

中断门和陷入门含有中断处理程序的入口地址，在中断发生时依据该入口地址转移到相应的中断处理函数代码执行。由中断门和陷入门的定义可以看出，中断门和陷入门使用不同的类型码， 110表示中断门，111为陷阱门。



#### 图6.1 中断门和陷入门定义

中断门和陷入门的唯一区别在于，通过中断门进入中断处理程序时CPU会自动屏蔽中断，也就是将CPU中EFLAGS寄存器的IF标志位清0，防止嵌套中断的发生，而根据陷入门进入中断处理函数则不清除IF位，允许中断的进入。

### I386保护模式下中断响应机制

当中断或异常发生时，CPU接收到中断请求，并获得对应的中断向量号，CPU使用该中断向量号作为查找中断描述符表的索引，找到对应的中断门或陷入门，获取门描述符中记录的中断处理函数入口地址，即该函数的段选择子和段偏移，然后跳转至处理函数的第一条指令开始执行，直至全部执行完成后返回。

I386保护模式下中断响应过程如下图所示(图6.2)，从中断请求产生到查询IDT获得中断处理程序入口地址并执行，这整个过程都是CPU完成的。唯一需要做的就是在IDT中为每个可能产生的中断或异常正确设置对应的中断门或陷入门，它们记录着各种中断或异常处理程序的函数入口地址信息。

GDT

段基址

GDTR

IDT

中断、异常处理函数

入口地址

中断向量号

IDTR

中断门、陷入门

选择子、段内偏移

#### 图6.2 保护模式下中断响应过程

### 中断描述符表的初始化

如前所述，I386保护模式下的中断处理需要通过查询IDT中的门描述符来完成，因此正确设置IDT中的中断门和陷入门尤其重要。

在操作系统初始化过程中，IDT类会注册相应的中断处理函数。在IDT类中，主要有一下两个方法设置中断处理函数：SetInterruptGate()和SetTrapGate()这两个函数注册相应的中断或异常处理函数到IDT表中的相应位置，两个函数分别用于注册中断门与陷入门。

/\* 初始化INTEL 已定义的异常 \*/

this->GetIDT().SetTrapGate(0, (unsigned int)Exception::DivideErrorEntrance);

this->GetIDT().SetTrapGate(1, (unsigned int)Exception::DebugEntrance);

this->GetIDT().SetTrapGate(2, (unsigned int)Exception::NMIEntrance);

/\* 设置磁盘中断对应中断门 \*/

this->GetIDT().SetInterruptGate(0x2E, (unsigned int)DiskInterrupt::DiskInterruptEntrance);

/\* 设置时钟中断对应中断门 \*/

this->GetIDT().SetInterruptGate(0x20, (unsigned int)Time::TimeInterruptEntrance);

#### 代码6.1 中断描述符表的初始化

在初始化阶段，总共需要在IDT表中注册256项中断处理程序。其中，前32项为INTEL使用的异常 。前20项为已被定义的错误，包括0号除零错，2号NMI不可屏蔽中断等，后12项为INTEL保留异常。在对异常注册完成后，注册相应设备的中断，包括时钟（0x20）中断，键盘（0x21）中断，硬盘（0x2E）中断等。最后在0x80位置注册系统调用的入口函数。代码中给出了初始化IDT部分中断门、陷入门的示例(代码6.1)。

## 预备知识：函数调用栈帧(Stack Frame)结构

### 栈帧(Stack Frame)

对于函数调用而言，栈是用于传递参数，存放返回地址、函数返回结果以及存储函数局部变量的。每次调用一个函数，系统都会在栈上为该函数调用实例分配相应的空间，这部分分配的空间就叫做栈帧(Stack Frame)，它存储了当前函数调用实例的运行状态。

函数调用的栈帧在函数调用发生时刻被建立，直至被调函数返回时刻销毁。如果当前函数调用实例A在其返回之前又调用了新的函数B，那么A调用实例的栈帧，即计算中间结果会保存在栈上，不被销毁；并且为函数调用实例B建立新的栈帧，直至计算获得函数B的返回值，销毁函数调用B的栈帧；然后接着执行函数A中剩下的部分，直至函数调用A返回时刻才销毁A的栈帧。

### 函数调用的栈帧结构

这里通过一个简单的函数调用的例子来解释栈帧建立、销毁的过程，以及栈帧的结构。假设被调用的函数叫做foo()，下面给出的是其源代码和对应的反汇编代码(AT&T汇编，其左操作数为源操作数，右操作数为目的操作数，与INTEL X86汇编恰好相反)。

c0100294 <\_foo>:

c0100294: 55 push %ebp

c0100295: 89 e5 mov %esp,%ebp

c0100297: 83 ec 08 sub $0x8,%esp

c010029a: c9 leave

c010029b: c3 ret

void foo()

{

int a, b;

return;

}

**反汇编结果**

**源代码**

下面的图示(图6.3)模拟了调用一次foo函数时，依次执行其中的每一条汇编指令之后栈状态变化的全过程。反汇编结果总共包含5条指令，分别用编号～表示。

**栈增长方向**

栈初始状态

返回地址

esp

**栈增长方向**

执行完指令①

返回地址

ebp

esp

esp

**栈增长方向**

执行完指令②

返回地址

ebp

ebp

**栈增长方向**

执行完指令③

返回地址

ebp

变量a空间

变量b空间

esp

ebp

**栈增长方向**

执行完指令④

返回地址

esp

**栈增长方向**

执行完指令⑤

上一层函数

esp

#### 图6.3 函数调用foo执行过程栈的变化

* 栈的初始状态

当函数foo被调用时，在正式开始执行foo函数的第一条指令之前，调用者会将函数返回地址压栈，以便foo函数执行完之后根据先前压入栈中的返回地址，从该返回地址继续执行主调函数未完的部分。因此栈中初始状态时的内容即是foo函数返回地址位于栈顶，由esp指向当前栈顶元素。

* 栈帧的建立

建立栈帧的工作由①②两条指令共同完成。指令①将ebp压栈，指令②将esp的值赋给ebp，这使得esp、ebp现在同时指向了栈顶存放原先ebp单元的位置。

这样做的目的是，让ebp记录当前栈帧的起始位置，而esp可以随着后续指令的执行，在栈上为局部变量分配空间的需要而移动，时刻记录“最新”的栈顶位置，同时也不至于丢失栈帧的起始位置。

指令③作用就是移动esp在栈上为局部变量开辟空间，而ebp始终记录栈帧的起始位置。

* 函数处理逻辑

在指令③和④之间的部分才是用户使用高级语言编写函数包含的处理逻辑。由于函数的源代码中没有相应的代码，所以本例中没有产生对应的汇编指令。我们只需要知道函数处理逻辑对应的汇编代码位置介于栈帧的创建与销毁两部分指令之间。

* 栈帧的销毁

指令④的作用是销毁栈帧，通常是在函数末尾使用leave指令来实现。leave指令等价于2条指令的效果：mov %ebp，%esp；和pop %ebp；这2条指令恰好完成了①②两条指令的逆过程，达到销毁栈帧的目的。

当调用函数的过程基本执行结束之后，也就不需要esp继续记录着为局部变量分配空间后的栈顶了，此时mov %ebp，%esp；指令将ebp中记录的栈帧起始位置恢复到esp中，ebp的作为一个“固定esp”的任务也就圆满完成了，继而从栈中弹出旧的ebp，esp指向栈顶存放的返回地址，一切就如最初尚未建立栈帧时的情况一样，最后ret指令根据栈中的返回地址从调用foo函数返回。

* 关于ebp的说明：

在调用foo函数建立的栈帧中，我们保存ebp原来的旧值，然后用ebp记录了当前foo调用实例的栈帧起始位置。那么问题是，保存在foo栈帧中ebp的旧值是什么内容呢？ebp旧值记录的应该是主调foo函数的上一层函数栈帧的起始位置，只不过是进入foo函数之后ebp旧值被保存入栈，继而将ebp用于记录foo函数栈帧的起始位置。换言之，ebp记录的始终是当前调用函数栈帧的起始位置，保存入栈中的旧值用于在当前层次函数调用返回时，用来恢复上一层调用函数的栈帧。

所有的函数编译后都会具有上面的foo函数产生的5条汇编指令相类似的反汇编特征，即在函数开始处建立栈帧，在函数末尾处销毁栈帧，不同函数的区别只是在建立和销毁栈帧之间有着各自的逻辑。

## Unix V6++中断处理流程

### 中断、异常处理概述

不论是异常中断还是设备中断（由8259A中断控制芯片发送的中断）都使用相同的中断处理流程。中断处理流程分为两部分，一部分由硬件实施，也称为中断隐指令。另一部分由软件处理，它又分为以下部分：a.保护现场和传递参数，b.调用中断处理函数，c.恢复现场和中断返回(图6.4)。

异常处理过程与中断处理流程类似， 但中断处理函数改为异常处理函数。下面对这些部分顺次进行说明。

中断请求

中断隐指令

保护现场

调用中断处理函数

恢复现场

中断返回

**硬件实施部分**

**软件执行部分**

#### 图6.4 中断处理流程

* 中断隐指令

当系统中发生中断或异常时，CPU会执行中断隐指令，将当前EFLAGS、CS、EIP寄存器的值依次入栈。CS、EIP寄存器中的值构成返回地址，记录了中断发生时刻CPU正在执行指令的下一条指令地址，以便处理完中断后可以根据栈中保存的返回地址，继续执行被打断的程序。

对于异常而言，int 0-31号异常中部分异常会产生出错码。对于会产生出错码的异常，除了将基本的EFLAGS、CS、EIP压入堆栈之外，还会在后面压入一个ErrorCode。下图给出了异常发生时中断隐指令在堆栈上压入值的情况(图6.5)。

中断前

中断后

**用户态堆栈**

**栈增长方向**

×××

中断前

中断后

**核心态堆栈**

**栈增长方向**

ESP

EFLAGS

CS

EIP

ErrorCode

esp

**占4字节**

SS

×××

#### 图6.5 中断隐指令对堆栈的影响(含特权级变化)

* 保护现场和传递参数

中断的产生是随机的，被打断的进程不可能预知中断。所以中断处理一定要保证系统环境在中断前后要一致。所以在中断处理前必须要将系统所有的寄存器保存起来。

至于参数传递的问题，对于后面即将调用的中断、异常处理函数来说，函数调用本身就是通过栈来传递参数的。这里手工保存系统中寄存器当前值到核心栈上，即等同于向中断、异常处理函数完成参数传递。

保存现场需要完成的工作就是将CPU的所有寄存器一并保存到核心栈上。等所有保存现场的工作完成之后，核心栈的情况应当如下图所示(图6.6)。

**中断前为核心态(有出错码)**

**栈增长方向**

×××

EFLAGS

CS

EIP

ErrorCode

ES

esp

DS

EAX

EBP

EDI

ESI

EDX

ECX

EBX

隐指令保存现场部分(硬件实施)

系统手工保存现场部分(软件实施)

中断前

中断后

#### 图6.6保存现场后核心栈

手工保存中断现场部分对于入栈的CPU寄存器先后顺序没有特别要求，可以随意安排，图中寄存器入栈的顺序只是所有可能顺序之一。

* 切换至核心态

因为中断处理函数需要在核心态下运行，在中断隐指令阶段，CPU中代码段CS寄存器和堆栈段SS寄存器已经切换至GDT中核心态代码段、数据段的选择子，而DS、ES寄存器仍然存放着指向用户态数据段的选择子。切换至核心态就是要将核心态数据段的选择子加载进DS、ES寄存器中，使得程序完全运行在核心态下。

* 调用中断处理函数

对于不同的中断或者异常都有各自的处理逻辑，这一步的任务就是对于不同的中断或异常，调用它们各自不同的处理函数进行处理。

函数调用通过栈传递参数，因而之前保存现场压入到核心栈上的寄存器值，就可以作为传递给中断处理函数的参数。假设中断处理函数Handler(arg1,…,argN)包含n个参数，由于函数调用时参数压栈的顺序是逆序入栈的，即argN最先压栈，arg1最后压栈；以图6.6中保存现场为例，参数arg1对应核心栈中现场保护区中的EBX寄存器，arg2对应ECX寄存器，依次类推，argN对应EFLAGS寄存器。

中断处理函数Handler需要能够知道中断发生时刻程序运行现场的状态，譬如中断前系统的状态，用户态或是核心态，进而区别进行处理；如果Handler没有任何传入参数的话，它将无从知晓需要的信息。因此，上述将核心栈现场保护区中寄存器现场直接作为中断处理函数Handler传入参数为解决这个问题提供了方法。

* 恢复现场和中断返回

恢复现场的工作和先前保存现场相对应，执行中断处理函数过程中寄存器的旧值会被覆盖，这也就是需要保存现场的原因；恢复现场步骤将保存在核心栈中各个寄存器的旧值逆序逐个恢复，以保证被中断的用户程序可以继续执行，就好像中断从未发生过一样。

对于有出错码的异常来说，由中断隐指令(即硬件实施)保存入栈的出错码，在恢复现场之后，中断返回之前需要我们从核心栈中手工弹出(即由软件执行)。最后，使用中断返回指令iret返回被中断的程序继续执行。

### 中断、异常处理框架

从上面的介绍中我们对中断、异常处理的整个流程大致掌握，整个过程可以分为中断/异常请求🡪中断隐指令🡪保存现场🡪进入核心态🡪调用中断处理函数🡪恢复现场🡪退出中断几个独立的步骤。

从设计的角度考虑，中断处理流程中的每一个步骤分别由一个函数来完成。中断隐指令由硬件完成，不属于考虑的范围。从保存现场直至退出中断的每个步骤可以分别为它们设计函数诸如：

1. SaveContext(); /\*保存现场\*/
2. SwitchToKernel(); /\*切换至核心态\*/
3. CallExceptionHandler(Handler); /\*调用中断、异常处理函数\*/
4. RestoreContext(); /\*恢复现场\*/
5. InterruptReturn(); /\*中断返回\*/

有了这样一些封装了中断处理流程每个子环节的函数之后，对于系统中所有中断、异常的处理过程就简化为依次调用上述函数即可。所有中断、异常对于保存现场、恢复现场、退出中断部分是完全一样的，存在区别的仅仅是CallExceptionHandler()中调用不同的处理函数执行相应处理逻辑。下图给出了所有中断、异常采用相似处理流程的示意图(图6.7)。

IDT

中断门、陷入门

**SaveContext();**

**ExceptionHandler();**

**RestoreContext();**

**InterruptReturn();**

**入口地址**

核心态代码段

中断门、陷入门

N#中断、异常处理函数

中断向量

**N**

**N+1**

N+1#中断、异常处理函数

**SaveContext();**

**ExceptionHandler();**

**RestoreContext();**

**InterruptReturn();**

相似处理过程

**N+2**

中断门、陷入门

N+2#中断、异常处理函数

#### 图6.7 所有中断、异常相似的处理流程

前面已经解释了如何将核心栈中的中断现场保护区作为中断处理函数的参数，使得我们可以在中断处理函数体内部对现场保护区进行访问。现场保护区进一步可以分为两个部分：(1) 由硬件实施，中断隐指令保护的现场；(2)由软件执行，系统手工保存的现场。根据这样的划分，可以向中断处理函数Handler提供两个参数，第一参数为系统手工保存的现场，即通用寄存器中的值；第二参数为中断隐指令保存的现场，即EFLAGS、CS、EIP、ErrorCode这部分现场。

第一参数(手工保存现场)和第二参数(隐指令现场)之间顺序是取决于这两部分现场在核心栈中的位置，因为隐指令现场先于手工保存现场压栈，隐指令保存的现场位于靠近栈底方向，手工保存现场部分靠近栈顶部分，根据函数调用规则规定的参数逆序入栈方式，靠近栈顶部分为第一参数，由此决定了中断处理函数第一参数为手工保存现场部分，第二参数为隐指令保存现场部分。

struct pt\_context{

unsigned int eip;

unsigned int xcs;

unsigned int eflags;

unsigned int esp;

unsigned int xss;

};

struct pte\_context{

unsigned int error\_code;

unsigned int eip;

unsigned int xcs;

unsigned int eflags;

unsigned int esp;

unsigned int xss;

};

**pt\_context结构体**

**pte\_context结构体**

Unix V6++中使用struct类型的数据结构来描述这两部分的现场。中断隐指令保存现场部分使用结构体pt\_context和pte\_context，其区别在于是否包含出错码字段。结构体中每个字段占用4个字节，字段顺序需要和核心栈中隐指令保存现场的压栈顺序相吻合，首字段对应最后压栈靠近栈顶方向上的寄存器，末字段对应最先压栈的寄存器。

系统手工保存现场部分使用结构体pt\_regs来描述，其各个字段如下所示：(与图6.6现场保护区中压栈顺序相吻合)

struct pt\_regs{

unsigned int ebx;

unsigned int ecx;

unsigned int edx;

unsigned int esi;

unsigned int edi;

unsigned int ebp;

unsigned int eax;

unsigned int xds;

unsigned int xes;

};

**pt\_regs结构体**

至此，我们设计了中断处理函数Handler的两个参数及其类型，中断处理函数声明也就可以相应确定下来，如下：

void Handler(struct pt\_regs regs, struct pt(e)\_context context);

#### 代码6.2 Handler函数声明

这样的中断处理函数声明似乎符合逻辑；传入的两个结构体参数也似乎与核心栈现场保护区中内容完全吻合，与图6.6中描绘的情况相一致。问题恰恰在于出现图6.6中描绘的核心栈有前提条件的，必须保证在核心栈中隐指令保护的现场与系统手工保护的现场两部分现场之间不存在“间隙”，上述中断处理函数Handler的参数声明才能与核心栈中保存的现场吻合，否则Handler声明参数与核心栈中实际数据就可能出现错位的情况。

现实情况是两部分现场之间的“间隙”确实存在并且难以避免，原因在于C++函数编译后产生的栈帧插入了“间隙”。

究竟是什么函数的栈帧插入了“间隙”，这需要进一步深入分析中断响应的过程。中断请求被响应之后首先是中断隐指令将EFLAGS、返回地址和出错码压入核心栈，然后根据中断向量查IDT中对应表项，根据中断门或陷入门中存放的函数入口地址转而执行相应函数(暂且将此函数称为入口函数)，该入口函数依次执行保存现场、调用中断处理程序、恢复现场和退出中断一系列步骤，如图6.7中所示。

该入口函数采用C++编写，而采用高级语言编写的函数在经过编译后会产生栈帧，即便它是个空白函数或者内部完全采用内联汇编(Inline Assembly)，编译后该函数生成汇编指令的收尾也会包括push %ebp; mov %esp, %ebp; … leave; ret;用于创建、销毁栈帧的指令。

正是入口函数在中断隐指令之后，在执行入口函数本身处理逻辑(即手工保存现场步骤)之前，进行了栈帧的创建，导致核心栈中产生了“间隙”(图6.8)

**栈增长方向**

×××

EFLAGS

CS

EIP

ErrorCode

ES

DS

EAX

EBP

EDI

ESI

EDX

ECX

EBX

隐指令保存现场部分(硬件实施)

系统手工保存现场部分(软件实施)

中断前

中断后

EBP

局部变量

esp

栈帧产生的“间隙”

**局部变量**占用核心栈大小由编译器在编译时期确定，无法预估

#### 图6.8 栈帧导致核心栈中产生的“间隙”

“间隙”占据的大小=EBP(4字节)+局部变量(编译时确定)，所以“间隙”的大小是不能预估来确定的，会受到局部变量个数以及其它因素影响而变化，以至于我们无法将中断处理函数Handler参数声明为如下形式，使得结构体pad大小恰好等于“间隙”字节数(代码6.3)。

void Handler(struct pt\_regs regs, struct pad padding, struct pt(e)\_context context);

#### 代码6.3 无法声明的padding

对于“间隙”问题的解决方法需要将中断处理函数Handler的参数声明为结构体指针类型，从而消除相邻两个结构体类型中字段需要和核心栈中现场保护区相吻合的限制；以结构体指针作为参数只需要指明两个结构体的地址，至于两者之间是否存在空隙或者有多大空隙都没有影响。最终改进后的中断处理函数声明如下(代码6.4)。

void Handler(struct pt\_regs\* regs, struct pt(e)\_context\* context);

#### 代码6.4 改进后使用结构体指针作为参数

相应地，我们需要位于核心栈栈顶的两个元素分别是指向系统手工保存现场区域和隐指令保存现场区域位于核心栈中起始位置的指针(图6.9)。

**栈增长方向**

×××

EFLAGS

CS

EIP

ErrorCode

ES

DS

EAX

EBP

EDI

ESI

EDX

ECX

EBX

隐指令保存现场部分(硬件实施)

系统手工保存现场部分(软件实施)

中断前

中断后

EBP

局部变量

esp

栈帧产生的“间隙”

向中断处理函数传入的结构体指针参数

pt\_context \*

pt\_regs \*

结构体指针指向各自现场保护区起始位置

#### 图6.9 结构体指针类型参数

### 中断、异常处理流程实现

###### SaveContext()

SaveContext()完成存现场兼向中断处理函数Handler传递参数的工作，目前为止我们将它设计成一个函数，其中完成将所有寄存器压入核心栈的工作。该函数用于在所有中断、异常处理流程的第一步进行调用；但用函数来实现方式会存在栈帧的问题(图6.10)。

**栈增长方向**

×××

ES

DS

EAX

EBP

EDI

ESI

EDX

ECX

EBX

SaveContext()函数实现保存现场

EBP

局部变量

**esp**

调用SaveContext()函数后

pt\_context \*

pt\_regs \*

调用SaveContext()函数前

**栈增长方向**

×××

ES

DS

EAX

EBP

EDI

ESI

EDX

ECX

EBX

EBP

局部变量

**esp**

pt\_context \*

pt\_regs \*

注意SaveContext()函数返回前、后栈顶指针ESP变化情况。

**SaveContext()函数返回前**

**SaveContext()函数返回后**

#### 图6.10 SaveContext()的函数方式实现

出现的问题在于当SaveContext()函数执行完手工保存现场的逻辑之后，函数返回时刻会销毁进入函数时创建的栈帧，栈顶指针ESP会恢复到调用SaveContext()函数之前核心栈状态。也就是如图中描绘的那样，虽然在SaveContext()函数中手工保存现场部分仍在核心栈中，但在后续函数调用时栈中这部分区域很快就会被新内容覆盖，保存的现场随机也就被破坏，在后面需要恢复现场时无法得到正确的数据，程序的执行会发生不可预知的错误。

究其根本原因，当使用函数方式来实现SaveContext()，函数体内保存现场部分数据被当作该函数中的局部变量存储在核心栈上，函数返回时一旦栈帧被销毁，核心栈中保存的现场数据就名存实亡了。

解决此问题的方法是使用宏展开方式来实现保存现场工作，这样就不存在栈帧销毁导致保存现场丢失的问题。宏展开方式实现的SaveContext()相关代码中使用Inline Assembly将所有寄存器保存至核心栈上(代码6.5)。

#define SaveContext() \

\_\_asm\_\_ \_\_volatile\_\_(" cld; \

pushl %%es; \

pushl %%ds; \

pushl %%eax; \

pushl %%ebp; \

pushl %%edi; \

pushl %%esi; \

pushl %%edx; \

pushl %%ecx; \

pushl %%ebx; \

lea 0x4(%%ebp), %%edx; \

pushl %%edx; \

lea 0x4(%%esp), %%edx; \

pushl %%edx"::)

#### 代码6.5 SaveContext()宏实现代码

其中，最后4条内联汇编指令完成的工作是将pt\_context \*和pt\_regs \*两个结构体指针依次压入核心栈，作为传入给中断处理函数Handler的参数。

###### 调用中断、异常处理函数

由于上一步中向中断处理函数Handler传递参数使用汇编语言的形式，因而这里调用中断处理函数时需要使用call指令，而不是通常C++之类高级语言的函数调用语句。

对不同中断、异常来说，其处理逻辑各不相同，这里需要调用其各自中断、异常处理函数，因而将具体的处理函数作为宏参数传入，进行相应的调用(代码6.6)。

#define CallHandler(Class, Handler) \

\_\_asm\_\_ \_\_volatile\_\_(" call \*%%eax" :: "a" (Class::Handler) );

#### 代码6.6 CallHandler()宏实现代码

###### RestoreContext()

与保存现场的步骤相对应，实现保存现场的逆向逻辑，使得被中断程序可以在中断返回后继续正常执行。将保存在核心栈中的现场依次从栈顶到栈底方向恢复到各个相应寄存器中，寄存器恢复现场时出栈的顺序应当恰好和之前压栈的顺序相反。

同样需要使用宏展开的方式实现，而不应当采用函数方式来实现，以免栈帧大小的问题导致恢复发生错位。下面给出的是RestoreContext()宏的实现代码，其中addl $8, %%esp;这条指令是用于跳过位于核心栈顶pt\_context \*和pt\_regs \*两个结构体指针占据的8个字节空间，使得位于栈顶的是真正的中断保存现场。

#define RestoreContext() \

\_\_asm\_\_ \_\_volatile\_\_(" addl $8, %%esp; \

popl %%ebx; \

popl %%ecx; \

popl %%edx; \

popl %%esi; \

popl %%edi; \

popl %%ebp; \

popl %%eax; \

popl %%ds; \

popl %%es"::)

#### 代码6.7 RestoreContext()宏实现代码

###### 中断退出前的进程调度检查

在执行完中断处理子程序后，会有一个例行的检查过程：如果中断先前为用户态，则先要检查重新调度标志 runrun，如果runrun标志被设置，则要进行进程调度(代码6.8)。

if( context->xcs & 0x3 ) /\*先前为用户态\*/

{

while(true)

{

X86Assembly::CLI(); /\* 屏蔽中断 \*/

if( Kernel::Instance().GetProcessManager().RunRun > 0 )

{

X86Assembly::STI(); /\* 允许中断 \*/

Kernel::Instance().GetProcessManager().Swtch();

}

else

{

break; /\* runrun == 0，则退栈回到用户态继续用户程序的执行 \*/

}

}

}

#### 代码6.8中断退出前的进程调度检查

### 异常处理函数的实现

异常处理函数的实现主要针对IDT中INT 0 – 31号异常向量。简要来说，就是内核需要为可能发生的异常提供处理函数，函数入口地址存放在IDT陷入门中。这类函数的实现抽象了异常处理的全过程，负责调用相应的函数来完成整个异常处理流程的每一步。

考虑到所有异常处理过程的相似性，仅有的差异是在入口函数中调用不同的异常处理函数，以及各个异常有无出错码的区别；所以使用了两个宏来实现异常处理入口函数，分别是IMPLEMENT\_EXCEPTION\_ENTRANCE\_ERRCODE(Exception\_Entrance, Exception\_Handler) 以及 IMPLEMENT\_EXCEPTION\_ENTRANCE(Exception\_Entrance, Exception\_Handler)，对应有无出错码异常的情况(代码6.9)。

#define IMPLEMENT\_EXCEPTION\_ENTRANCE(Exception\_Entrance, Exception\_Handler) \

void Exception::Exception\_Entrance() \

{ \

SaveContext(); \

SwitchToKernel(); \

CallExceptionHandler(Exception\_Handler); \

RestoreContext(); \

\_\_asm\_\_ \_\_volatile\_\_("leave"::); \

\_\_asm\_\_ \_\_volatile\_\_("iret"::); \

}

#### 代码6.9 无出错码异常处理入口函数

#define IMPLEMENT\_EXCEPTION\_ENTRANCE\_ERRCODE(Exception\_Entrance, \

Exception\_Handler) \

void Exception::Exception\_Entrance() \

{ \

SaveContext(); \

SwitchToKernel(); \

CallExceptionHandler(Exception\_Handler); \

RestoreContext(); \

\_\_asm\_\_ \_\_volatile\_\_("leave"::); \

\_\_asm\_\_ \_\_volatile\_\_("addl $4, %%esp" ::); \

\_\_asm\_\_ \_\_volatile\_\_("iret"::); \

}

#### 代码6.10 有出错码异常处理入口函数

其中，Exception\_Entrance为入口函数的函数名，Exception\_Handler为异常处理函数的函数名。以INT 0号异常除法错(无出错码)为例，函数声明方式如下(代码6.11)：

IMPLEMENT\_EXCEPTION\_ENTRANCE(DivideErrorEntrance, DivideError)

#### 代码6.11 DivideError异常处理函数

上述宏定义语句展开之后即为int 0号异常处理入口函数DivideErrorEntrance的函数定义，在其中调用DivideError异常处理函数对int 0号异常执行特定的处理逻辑，如输出异常相关信息之类。

对于有出错码的异常，其处理流程相对于无出错码处理流程差别仅仅在于中断返回之前需要手工从核心栈中弹出出错码，该出错码是由中断隐指令入栈的，因此两个宏之间的差别也就是语句\_\_asm\_\_ \_\_volatile\_\_("addl $4, %%esp" ::); 用于跳过核心栈中出错码占据的4字节空间。前提是在使用leave指令销毁入口函数的栈帧之后，使得出错码位于栈顶位置时刻进行。

这里需要我们自己插入leave指令的原因是，对于C++函数编译后产生的汇编指令来说，其最后的leave和ret两条指令是紧密相邻的，也即是使用高级语言的情况下，无法做到在leave和ret指令之间插入自己的处理逻辑；而这里我们又不得不这样做----在销毁入口函数栈帧之后弹出出错码；并且需要使用iret而不是ret指令，从中断返回而不是一般函数调用的返回。正因为如此，才需要在恢复完中断现场之后插入自己的leave指令，继而弹出出错码，最后使用专用于中断返回的iret指令返回到被中断的进程。

## 系统调用接口

系统调用是用户主动使用int中断指令进入内核，使用内核提供的一些服务的方式。I386体系结构上系统调用的汇编形式总是以int指令开头。Unix V6++中使用0x80作为系统调用号。

系统调用的处理流程与中断、异常的处理流程基本一致，同样都需要经过保护现场、进入核心态、执行中断或系统调用处理函数、恢复现场以及返回用户程序几个阶段。两者的区别在于，用户程序有意识地将系统调用的传入参数放在寄存器中，当执行到int指令陷入内核时寄存器现场被保存下来，并作为传递给系统调用处理函数的参数，正是通过这样的方式系统将调用参数从用户程序传入内核，内核也是经由寄存器将系统调用结果返回给用户程序；而中断则不然，中断请求是异步产生的，而且也没有参数需要经过寄存器传递给内核，中断处理过程中保存现场的目的仅仅是为了以后恢复被中断程序的执行。

### 系统调用入口函数

系统调用的入口函数与中断或异常的入口函数在实现上大致相似，该函数封装了整个系统调用处理流程：保护现场、进入核心态、执行系统调用处理函数、恢复现场以及返回用户程序，函数的入口地址存放在(代码6.12)。

void SystemCall::SystemCallEntrance()

{

SaveContext();

SwitchToKernel();

CallHandler(SystemCall, Trap); /\* 系统调用处理子程序Trap() \*/

RestoreContext();

\_\_asm\_\_ \_\_volatile\_\_("leave; iret"::);

}

#### 代码6.12 系统调用入口函数SystemCallEntrance()

系统调用入口函数会调用SystemCall::Trap(struct pt\_regs\* regs, struct pt\_context\* context)函数，该函数会根据regs.EAX中的系统调用号在系统调用表中执行相应的系统调用处理逻辑。

### 系统调用参数传递

系统调用都是有用户态程序发起的，而系统调用过程都运行于核心态，因此需要将用户态数据传递到核心态。

Unix V6++使用寄存器传递用户态数据到核心态，传递参数的数量取决于寄存器个数，每次最多传递8个双字长度(DWORD)，Unix V6++中还没有系统调用的参数个数超出这一限制。而对于寄存器无法容纳的大块数据的传输，则可以使用寄存器传递存放大块数据的目标内存地址。

在执行INT指令前先将所需要传递的参数存放在寄存器中。在执行INT指令后，先进行保护现场工作，也就是将所有的寄存器压入栈中，再调用系统调用子程序。由于系统调用子程序使用C语言来编写的，C语言使用栈来传递函数参数，所以保护现场就相当于为系统调用子程序提供了运行的参数(图6.11)。

用户程序

寄存器

核心栈

#### 图6.11 系统调用参数传递过程

参数从用户程序到核心栈的步骤一般都是由用户态下函数库实现，从寄存器到核心栈的环节则是由系统调用入口函数SystemCall::SystemCallEntrance()中保护现场SaveContext()宏实现的。

我们以read系统调用为例，来解释在用户态下如何将系统调用参数放入寄存器中(代码6.13)。

int read(int fd, char\* buf, int nbytes)

{

int res;

\_\_asm\_\_ \_\_volatile\_\_ ("int $0x80":"=a"(res):"a"(3),"b"(fd),"c"(buf),"d"(nbytes));

if ( res >= 0 )

return res;

return -1;

}

#### 代码6.13 read()系统调用的实现

read()系统调用有三个传入参数：fd、buf和nbytes，read()的系统调用号为3。其中使用了内联汇编，作用是将系统调用号3和三个参数fd、buf、nbytes分别与EAX、EBX、ECX、EDX寄存器对应，EAX存放系统调用号3，EBX存放fd，ECX存放buf，EDX存放nbytes，然后执行INT 0x80陷入指令进入内核；res对应系统调用的返回值，经由EAX寄存器返回给用户程序。

EAX寄存器在进入系统调用前存放系统调用号传递给内核，而在系统调用完成后用于将返回值从内核返回给用户程序。

### 系统调用处理子程序SystemCall::Trap()

系统调用处理子程序SystemCall::Trap(struct pt\_regs\*, struct pt\_context\*)函数完成具体系统调用逻辑的执行，它负责的主要任务有：

(1) 根据传入参数pt\_regs类型的结构体指针获取保存在核心栈上的参数，其中pt\_regs->eax中保存的是系统调用号，而调用之后，被当作系统调用的返回值使用。

(2) 设置现运行进程User结构中的u\_ar0字段，使其指向核心栈现场保护区中pt\_regs->eax单元，这样系统调用函数就可以通过类似u.u\_ar0[IDX]的方式修改核心栈现场保护区中的内容。Unix V6++中，返回给用户程序的系统调用返回值就是通过修改核心栈中u.u\_ar0[User::EAX]单元来实现的。

(3) 以系统调用号作为查找系统调用入口表的索引，然后转入执行内核中相应的系统调用程序。

### 系统调用入口表

系统调用使用中断描述符表中的0x80项，每个系统调用的中断入口都是一样的，系统调用的外层程序根据系统调用号来区分不同的系统调用，具体行为就是利用系统调用号作为索引来查找系统调用处理程序入口表。

在系统调用类SystemCall中，有一个静态的存储所有系统调用处理函数的静态数组m\_SystemEntranceTable[SYSTEM\_CALL\_NUM]，该数组中每项使用如下数据结构对系统调用进行描述(代码6.14)。

/\*系统调用入口表表项的定义\*/

struct SystemCallTableEntry

{

unsigned int count; /\* 系统调用的参数个数 \*/

int (\*call)(); /\* 相应系统调用处理函数的指针 \*/

};

#### 代码6.14系统调用入口表表项的定义

在系统调用入口表被初始化时，按照表现定义的结构体字段填写系统调用处理函数参数个数与函数入口地址(代码6.15)。

/\*对系统调用入口表初始化\*/

SystemCallTableEntry SystemCall::m\_SystemEntranceTable[SYSTEM\_CALL\_NUM] =

{

{ 0, &NullSystemCall }, /\* 0 = indir \*/

{ 1, &Rexit }, /\* 1 = rexit \*/

{ 0, &Fork }, /\* 2 = fork \*/

{ 3, &Read }, /\* 3 = read \*/

{ 3, &Write }, /\* 4 = write \*/

……

};

#### 代码6.15系统调用入口表初始化

各系统调用在入口表中占用的位置就是它们的系统调用类型号。例如，创建新进程系统调用fork的类型数是2，退出进程系统调用的类型数是1等。反之，根据系统调用类型数可以在表中找到该系统调用参数个数和处理函数入口。