目录

第 12 章　中断和异常处理

中断是计算机系统中非常重要的一个概念，在操作系统、计算机组成原理等课程中都会介绍中断的概念、用途和实现机制。本章介绍中断和异常的基本概念，给出了异常处理程序实例，并从机器语言的角度来分析异常处理运行的基本过程。

12．1 中断与异常的基础知识

12.1.1 中断和异常的概念

“中断”在人们的日常生活当中屡见不鲜。假如你正在做某一件事情的过程中（如写作业），发生了另外一件事情（如电话铃声响起），此时，你停下原来执行的任务（写作业），转去处理新事件（接电话），处理完新事件后（通话结束），再继续执行原来的任务（继续写作业）。这一套过程就是中断。

在计算机世界里，中断（Interruption）是指 CPU 获知发生了某件事件，CPU 暂停当前正在执行的程序，转而为临时出现的事件服务，即执行中断处理程序，执行完中断处理程序后又能自动恢复执行原来程序的一种功能。要比较清晰的理解中断，就要回答以下问题：

(1) 有哪些事情会引起中断？

(2)　CPU 为什么能感知中断？

(3) 在何处去找中断处理程序？

(4) 如何从中断处理程序返回？

这些问题的答案分别对应中断源、中断系统、中断描述符表、中断调用和中断返回指令等概念。当然，还有其他问题，如一个中断发生后能否有被其他事件打断？如果同时产生了多个中断，如何确定优先响应的中断等等。下面一一介绍。

引起中断的事件称为中断源。常见的中断源包括键盘中断、鼠标中断、时钟中断；其他的中断还有电源掉电、存储器出错、总线奇偶效验错误等由硬件故障引起的中断。这些来源于 CPU 之外的中断称为外部中断，也称为异步中断，它是有其他硬件设备产生的。外部中断可分为不可屏蔽中断（Non Maskable Interrupt，NMI）和可屏蔽中断。对于可屏蔽中断，是否响应取决于标志寄存器 EFLAGS 中的 IF 标志位。若 IF=1，则响应中断，否则不响应。使用指令 sti 和 cli 可分别将 IF 置为 1 和 0。在生活中发生的一些的事件后，我们也可以置之不理。例如，在上课的时候，我们不响应手机电话，即不中断上课。对于一个中断处理程序而言，能否再次被其他可屏蔽中断所打断，同样取决于 IF 是否为 1。

异常（Exception）指 CPU 内部出现的中断，也称为同步中断。它是一条指令的执行过程中，CPU 检测到了某种预先定义条件，要终止该指令的执行而产生的一个异常信号，进而调用异常处理程序对该异常进行处理。与中断处理程序执行后会返回到被中断处继续执行不同，在异常处理程序执行后有三种后续不同的操作：重新执行引起异常的指令、执行引起异常指令之下的指令、终止程序运行。这取决于异常的类型。在 Intel CPU 中，异常分为三类：故障（faults）、陷阱（traps）、中止（aborts）。

故障异常是在引起异常的指令之前，或者在指令执行期间，检测到故障或者预先定义的条件不能满足的情况下产生的。常见的故障异常有：除法出错（除数为 0，或者除数很小，被除数很大，导致商要溢出）、数据访问越界（访问一个不准本程序访问的内存单元）、缺页等。故障异常通常可以纠正，处理完异常时，引起故障的指令被重新执行。

陷阱异常与故障异常不同，它不是在异常的指令执行之前发出信号，而是在执行引起异常的指令之后，把异常情况通知给系统。根据将要执行的指令的地址是由 CS:EIP 来决定的这一原则，在产生故障异常信号时，CS：EIP 是要引起异常的指令的地址；而产生陷阱异常信号时，CS：EIP 是异常处理之后将要执行的指令的地址。对于软中断之类的陷阱异常，实际上就是产生异常信号指令之下的一条语句的地址。所有软中断，就是在程序中写了中断指令，执行该语句就会去调用中断处理程序，中断处理完后又继续运行下面的程序。这就像我们要做的作业中有一项任务是打电话一样，看到这一作业任务后，就拨打电话，通话结束继续做其他作业。软中断调用与调用一般的子程序非常类似。借助于中断处理这一模式，可以调用操作系统提供的很多服务程序。另外一种常见的陷阱异常是是单步异常，单步异常用于防止一步步跟踪程序。

中止是在系统出现严重情况时，通知系统的一种异常。引起中止的指令是无法确定的。产生中止时，正执行的程序不能被恢复执行。系统接收中止信号后，处理程序要重新建立各种系统表格，并可能重新启动操作系统。中止的例子包括硬件故障和系统表中出现非法值或不一致的值。

为了实现中断和异常处理，在计算机中要有相应的软、硬件装置，称为中断系统，该系统能够“感知”发生了引起的事件。事实上，CPU 在执行完每一条指令后（或者下一条语句之前），都会主动的“查看”有无中断和异常信号，从而实现中断的感知。中断和异常产生的原因虽然不同，但是它们的相应处理机制是相同的。CPU 都要暂停正在执行的程序，而去调用中断处理程序或者异常处理程序，这就需要 CPU 能够找到中断和异常处理程序的入口地址。下面将介绍计算机中是如何管理这些处理程序的入口地址的。

12.1.2 中断描述符表

在 IA-32 体系结构中，对每一种中断和异常都进行了编号，称为中断向量号或者简称为中断号。表 12.1 列出了中断和异常的编号、名称、类型和产生的原因，中断向量号在 0-255 之间。

表 12.1　中断和异常一览表

向量号	名　称	类型	中断和异常源
0	除法出错	故障异常	DIV、IDIV 指令
1	调试异常（单步中断）	陷阱异常/中断	任何指令或数据访问
2	非屏蔽中断	中断	不可屏蔽的外部中断源
3	断点（单字节）	陷阱异常	INT 3 指令
4	溢出	陷阱异常	INTO 指令
5	边界检查	故障异常	BOUND 指令
6	非法操作码	故障异常	非法指令编码或操作数
7	协处理器无效	故障异常	浮点指令或 WAIT/FWAIT 指令
8	双重故障	中止异常	任何可以产生异常或中断的指令
9	协处理器段超越	中止异常	访问存储器的浮点指令
10	无效 TSS	故障异常	JMP、CALL、中断、IRET
11	段不存在	故障异常	装载段寄存器的指令或访问系统段

12	堆栈段异常	故障异常	装载 SS 寄存器的指令，堆栈操作指令
13	通用保护异常	故障异常	任何访问内存的指令，其他保护检查
14	页失效	异故障常	任何访问内存的指令
15	保留		
16	浮点出错（算术错误）	故障异常	浮点指令或 WAIT
17	对齐检查	故障异常	内存中的数据访问
18	机器检查	中止异常	错误代码和来源取决于 CPU 型号
19	SIMD 浮点异常	故障异常	SIMD 浮点指令
20	虚拟化异常		仅出现在支持 EPT-violations 的 CPU 中
21-31	保留		
32-255	可屏蔽中断、软中断	可屏蔽中断/陷阱异常	可屏蔽中断或者 INT 指令

每一个中断或异常处理程序都有一个入口地址。在计算机中，往往将中断和异常处理程序的入口地址等信息称为“门”（gate），就像一栋楼房的门代表了该楼房的入口一样。根据中断和异常处理程序的类别，将与之连接的中断描述符划分为三种门：任务门（执行中断处理程序时将发生任务转移）、中断门（主要用于处理外部中断，响应中断时自动将 0→TF 和 0→IF）和陷阱门（主要用于处理异常，响应中断时只将 0→TF）。CPU 根据门提供的信息（由 IDT 中的门属性字节提供）进行切换，对不同的门，处理过程是有些差异的。

在保护方式下，寻找一个中断或异常处理程序的入口地址是一个稍微有些复杂的过程，其基本原理与 3.6 节中介绍的保护方式下的地址形成是相同的。对于一个中断或异常处理程序，它是在一个代码段中。对于该代码段要采用 4 个字节的段描述符进行描述，在描述符中包含段基址、段长度、段类型、特权级等信息。将段描述符放在全局部描述符表 GDT 或者局部描述符表 LDT 中。因此，确定一个中断或异常处理程序入口地址时，应能够从 GDT 或者 LDT 中找出相应的段描述符，从而确定它所在代码段的基址；同时还需要知道入口地址在代码段中的偏移地址。这些信息放在中断描述符中。一个中断描述符占 8 个字节，也称为“门描述符”。中断门、陷阱门、任务门的描述符略有差别，它们的结构分别如图 12.1、12.2、12.3 所示，图中 P 为存在位，1 表示描述符对应的段在内存中；DPL 为描述符所描述的段的特权级； D 表示门的大小，1 表示是 32 位，0 表示 16 位。

31 16 15 14 13 12 8 7 5 4 0

偏移量 （高 16 位， 即 31-16 位）	P	DPL	0 D 1 1 0	0 0 0	未 用
段选择符（16 位）	偏移量 （低 16 位， 即 15-0 位）				

图 12.1 中断门描述符

31 16 15 14 13 12 8 7 5 4 0

偏移量 （高 16 位， 即 31-16 位）	P	DPL	0 D 1 1 1	0 0 0	未 用
段选择符（16 位）	偏移量 （低 16 位， 即 15-0 位）				

图 12.2 陷阱门描述符

31 16 15 14 13 12 8 7 0

未 用	P	DPL	0 0 1 0 1	未 用
TSS 段选择符（16 位）	未 用			

图 12.3 任务门描述符

将各个中断门或陷阱门的描述符按照中断向量编号的顺序存放在一起，形成的表就是中断描述符表（Interrupt Descriptor Table，IDT）。该表的大小为 2kb(256 项*8 字节/项=2048 字节)。中断描述符表是中断向量号与对应的中断和异常处理程序之间的连接表。

在保护方式下，中断描述符表可以放在内存的任何位置。CPU 中使用中断描述符寄存器 IDTR（Interrupt Descriptor Table Register）保存该表的起始位置。显然，由 IDTR 和中断向量号很容易就可以找到相应的中断和异常处理程序的入口地址。只要修改 IDTR 的内容，就可以让 CPU 将当前使用的中断描述符表切换成另外一个新的中断描述符表。

在保护方式下，当 CPU 响应一个编号为 n 的中断或异常时，CPU 通过 IDTR 得到 IDT 的基址，以 n*8 作为偏移值，就可以指向对应“门”（即中断描述符）的起始地址。从门描述符到最终的中断处理程序入口地址还需要进行下一步的查表转换，图 11.4 表示了从中断门或陷阱门转到中断处理程序的查表过程。如果是任务门，则通过任务门中的段选择符（偏移值无效）从全局描述符表 GDT 中找到 TSS 描述符，再通过 TSS 描述符获得中断处理程序的入口地址（CS 和 EIP）。

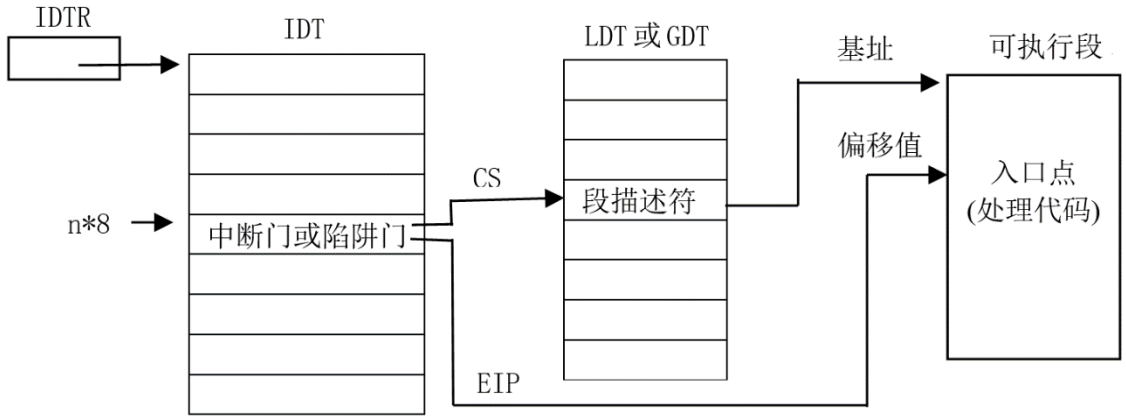


图 12.4　确定中断处理程序入口地址的过程示意图

在实方式下，每一个中断处理程序的入口地址是 4 个字节，包含 2 个字节的段内偏移和 2 个字节的段地址。这些信息也是按中断号的顺序放在一起的，形成中断矢量表。中断矢量表放在内存的最低端。CPU 获得 n 号中断处理程序入口地址的方法是：（0：[n*4]）→IP，（0：[n*4+2]）→CS。

12.1.3 中断和异常的响应过程

在一个中断或者异常发生后，如果当前正在执行的程序与中断/异常处理程序具有相同的特权级，那么中断/异常处理程序将使用被中断程序所使用的栈，而不用切换栈。如果两者的特权级不同，则要切换栈。

在不切换栈时，中断和异常的响应过程如下：

① （EFLAGS）入栈
② （CS）入栈
③ （EIP）入栈
④ 如果有错误编码，错误编码入栈
⑤ 从中断门或陷阱门获取段选择符送入 CS，偏移地址送 EIP

⑥ 如果是中断门，标志寄存器中的 IF 置为 0（关中断）。

在此之后，就会根据新的 CS 和 EIP，取到中断/异常处理程序的入口处的指令，开始中断/异常服务程序的运行。在该服务的最后，应有一条指令 IRET，它实现从服务程序返回到被中断的程序。执行过程是：

① 弹出 EIP

② 弹出 CS

③ 弹出 EFLAGS

④ ESP 增加适当的值，以消除参数占用的空间

在此之后，根据新的 CS、EIP，就可以取到被中断程序中的指令，恢复被中断程序的执行。

在切换栈时，中断和异常的响应过程略微复杂一些。因为是要用服务程序的栈，所有的信息都要保存在服务程序所使用的栈中。处理器具体处理过程如下：

① 临时存储 SS、ESP、EFLAGS、CS 和 EIP；

② 从任务状态段(Task State Ssegment，TSS)中找到该服务程序所使用的堆栈段选择子和栈顶指针送给 SS 和 ESP；

③ 将在① 中暂存的 SS、ESP 、EFLAGS、CS 和 EIP 入栈（即入服务程序的栈）；

④ 将被中断程序栈中的参数拷贝到新栈中（在门描述符中有参数个数的信息）

⑤ 错误代码入栈

⑥ 从中断门或陷阱门获取段选择符送入 CS，偏移地址送 EIP

⑦ 如果是中断门，标志寄存器中的 IF 置为 0（关中断）。

从服务程序返回的过程与不切换栈时的服务程序返回是类似的。

比较一般的子程序调用和中断/异常处理程序的调用，它们有一些共同点：在堆栈中要保存以后将要返回的主程序或者被中断程序的地址；要将子程序或者服务程序的入口地址送给 CS/EIP；在子程序或者服务程序运行结束时，会执行一条指令，从堆栈中弹出前面保存的地址送给 CS/EIP。但是，也有一些差别。主程序通过 CALL 指令调用子程序，这是事先就在程序中写好的，但是外部中断发生具有随机性，主程序中并没有类似于 CALL 的指令，除非是软中断或者陷阱异常。CPU 进行中断/异常响应时，会保存标志寄存器。保存标志寄存器的原因很简单，就是中断具有随机性，中断返回时要像什么事也没有发生过。例如，程序中有比较指令 CMP，之后有条件转移指令，若执行 CMP 指令后被中断，在运行中断处理程序的过程中，会改变标志寄存器的值。当返回到被中断的程序时，不恢复原来的标志寄存器，执行有条件转移指令所依据的标志位与 CMP 指令所设置的标志位不同，逻辑上就会有错误。而子程序调用是程序开发者自己安排的程序逻辑，不应该在 CMP 和有条件转移指令之间插入一条 CALL 指令。如果非要这样写，也只能编程者对程序的逻辑正确性负责。当然，中断和一般子程序调用的其他差别还有错误代码压栈、关中断等。

12.1.4 软中断指令

软中断通过程序中的软中断指令实现，它可以用显式的方法调用一个中断或者异常处理程序，所以又称它为程序自中断。软中断指令的执行过程与 12.1.3 节介绍的中断响应过程完全一样。

软中断指令有 INT n、INTO、INT 3、BOUND，其中，n 为中断向量号，取值范围为 0~255。

INTO 是在标志寄存器中 OF 标志位为 1 时，调用溢出异常处理程序。注意，在 OF=1 时，如果不显式的写出 INTO 指令，是不会调用溢出异常处理程序的；例如：执行“MOV AX，7FFFH”

和“ADD AX， 3”之后，尽管 OF=1，但并不表示前面的运算出现了问题，将两个加数当成无符号数看待，32767+3=32770 = 8002H，是很正常的。此外，如果 OF=0，即使写了 INTO 指令，也不产生异常，继续执行 INTO 后面的指令。INTO 指令的核心是调用“INT 4”，只是在调用前增加了一个 OF 标志位的判断而已。

INT 3 是“INT n”的一个特例（n=3），其机器码是 0CCH。它显式地调用断点异常处理程序。在调试程序的时候，我们可以在程序中设置断点，当程序运行到断点处时，会暂停运行，此时可以使用调试工具观察各种各样的信息。使用“INT 3”指令后，调试程序时，不用设置断点，程序执行“INT 3”时，就会调用异常处理程序，使得程序在此处暂停运行，达到设置断点的效果。

BOUND 是边界检查异常指令，其格式为 “BOUND r, mem”。其中 r 是一个寄存器；mem 是与 r 同类型的存储单元地址，该地址单元中对应的数据是下边界，该单元的下一个单元中的内容是上边界。当寄存器 r 中的值在上、下边界值的范围内，不会引发异常，即不会执行“INT 5”，否则执行“INT 5”，执行边界异常处理程序。

在中断/异常处理程序中，应有 IRET 指令。IRET 执行从堆栈中弹出 EIP、CS、EFLAGS，使得程序能够恢复到被中断的位置继续执行。

12．2 Windows 中的结构化异常处理

对于每一个中断或者异常，操作系统都为它们设定了中断/异常处理程序。在 Windows 系统中，预定义的异常处理程序的功能是弹出一个错误对话框，然后终止程序的运行。在很多情况下，人们希望处理异常后让程序继续运行下去而不是终止。这就需要程序员自己为异常编写相应的处理程序，而不使用系统预先安排的处理程序。

在 C 语言程序设计中，可以采用“__try……__except”、“try …… catch”等语句来实现这一处理方法。虽然我们可以依葫芦画瓢，使用这些语句来处理异常，但是它们内部实现的机制是被隐藏了。本节将介绍 Windows 系统中采用结构化异常处理（Structured Exception Handling，SEH）机制，从更底层来揭示异常处理的奥秘。值得注意的是，在 Windows 平台上除了结构化异常处理机制外，还有 Windows XP 中引入的向量化异常处理（Vectored Exception Handling，VEH）、C++异常处理（C++ Exception Handler，C++EH）。C++EH 由 C++编译器实现的异常处理机制，通过阅读 C 语言程序的反汇编代码，可以更好地领悟其异常的处理过程。

12.2.1 编写异常处理函数

当执行一条指令出现异常时，操作系统可以自动调用程序开发者编写的一个异常处理函数，该函数也称为回调函数。当然，前提是程序员不但要写出接口符合规范的异常处理函数，而且还要向操作系统注册该函数。回调函数可以显示错误信息、修复错误或者完成其他任务。无论回调函数做什么，它最后都要返回一个值来告诉操作系统下一步做什么。

异常处理回调函数的格式如下：

```
EXCEPTION_DISPOSITION  __cdecl  _except_handler(
        struct  _EXCEPTION_RECORD  *ExceptionRecord,
        void    * EstablisherFrame,
```

```
    struct    _CONTEXT    *ContextRecord,
    void      * DispatcherContext);
```

函数_except_handler 中有 4 个参数，这 4 个参数是操作系统在调用函数_except_handler 时自动传入的，本节只介绍其中的第一和第三个参数。

函数_except_handler 的第一个参数是指向结构_EXCEPTION_RECORD 的指针。结构 EXCEPTION_RECORD 定义在 winnt.h 中，它含有异常的编码、异常标志、异常发生的地址、参数个数、异常信息等等，结构定义如下。

```
typedef struct _EXCEPTION_RECORD {
    DWORD     ExceptionCode;
    DWORD     ExceptionFlags;
    struct       _EXCEPTION_RECORD    *ExceptionRecord;
    PVOID       ExceptionAddress;
    DWORD    NumberParameters;
    ULONG_PTR    ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

在操作系统中，已定义了各种异常的编码，例如：访问异常 STATUS_ACCESS_VIOLATION 的编码是 0xC0000005；整数除 0 异常 STATUS_INTEGER_DIVIDE_BY_ZERO 的编码是 0xC0000094 等，这些信息都定义在 winnt.h 中。

函数_except_handler 中的第三个参数是指向结构_CONTEXT 的指针。结构 CONTEXT 也定义在 winnt.h 中，存放的信息是发生异常时各个寄存器的值，包括通用寄存器、段寄存器、指令指示器、调试寄存器的内容，即是上下文信息或者环境信息。

函数_except_handler 的返回值为 EXCEPTION_DISPOSITION，这是一个枚举类型，定义如下：

```
typedef enum _EXCEPTION_DISPOSITION {
    ExceptionContinueExecution,
    ExceptionContinueSearch,
    ExceptionNestedException,
    ExceptionCollidedUnwind
} EXCEPTION_DISPOSITION;
```

ExceptionContinueExecution 表示已经处理了异常，回到异常触发点继续执行；

ExceptionContinueSearch 表示继续遍历异常链表，寻找其他的异常处理方法；

ExceptionNestedException 表示在处理过程中再次触发异常。

在一个异常处理程序中，可以根据操作系统传递进来的参数，决定是采用的处理方法，并在最后返回下一步准备采取的动作。

在例 12.1 中，给出了异常处理函数的示例。

12.2.2 异常处理程序的注册

结构化异常处理是基于线程的，每个线程都有它自己的异常处理程序（回调函数）。

在 windows 系统中，对每一个进程都创建一个线程信息块 TIB（Thread Information Block，）来保存与线程有关的信息。线程信息块的数据结构 NT_TIB 的定义在 winnt.h 中。在该结构中的

第一个字段为：

```
struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
```

其中 EXCEPTION_REGISTRATION_RECORD 的定义如下：

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD  *Next;
    PEXCEPTION_ROUTINE  Handler;
} EXCEPTION_REGISTRATION_RECORD;
```

这是一个链表结构，每一个节点中含有一个异常处理程序的入口地址 Handler。即各个异常处理程序的入口地址形成一个链表，其头节点的指针为 ExceptionList。

在基于 Intel 处理器的 Win32 平台上，段寄存器 FS 总是指向当前的 TIB。因此在 FS:[0]处可以找到一个指向 EXCEPTION_REGISTRATION_RECORD 结构的指针，从而找到各异常处理程序的入口地址。链表的结构如图 12.5 所示。

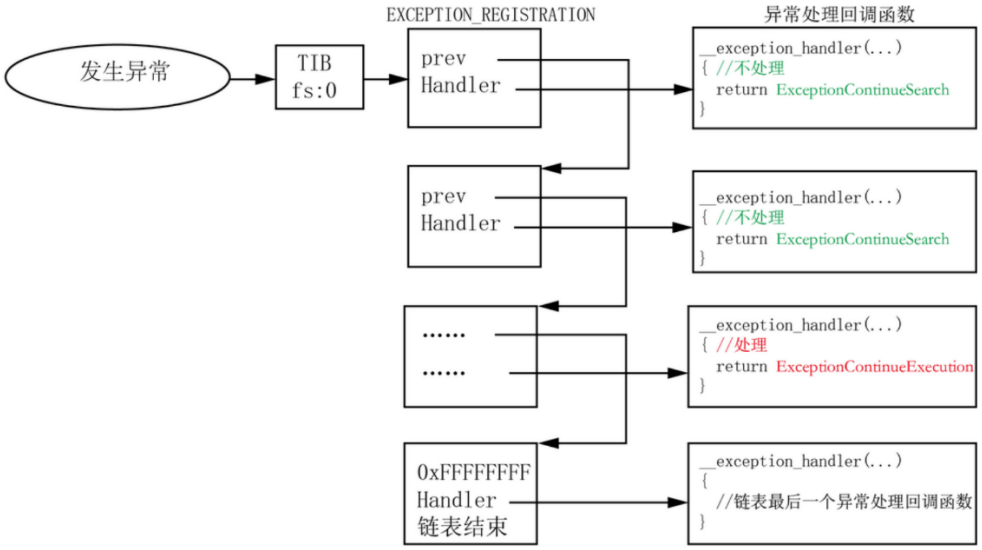


图 12.5 异常处理注册记录链表示意图

在程序执行过程中遇到异常事件时，操作系统中的函数 RtlDispatchException 会从 FS:[0]指向的链表表头开始，依次调用每个节点指明的异常处理回调函数，直到某个异常处理回调函数的返回值为 0 为止（即函数的返回值为 ExceptionContinueExecution）。异常处理回调函数的返回值为 0，表示已经处理了该异常，该线程可以恢复执行。链表最末一项是操作系统在装入线程时设置的指向 UnhandledExceptionFilter 函数，该函数总是向用户显示“Application error”对话框。

下面给出了一个 C 语言中嵌入汇编的程序示例，展示了异常处理程序的安装，即将异常处理程序的入口地址加入到异常处理链表中的方法，以及恢复异常处理链表的方法。运行程序，可以看到异常发生、调用了异常处理程序后的结果。有些内容在后面的小节中介绍。

【例 12.1】地址非法访问异常的修复

```
#include   <windows.h>
#include   <stdio.h>
int   modify_var=0;
EXCEPTION_DISPOSITION   __cdecl _except_handler(
          struct   _EXCEPTION_RECORD   *ExceptionRecord,
          void     * EstablisherFrame,
```

```
        struct    _CONTEXT    *ContextRecord,
        void      * DispatcherContext)
{
    ContextRecord->Eax = (DWORD)&modify_var;
    return ExceptionContinueExecution;
    //该值会导致引起异常的语句 "mov[eax],1"重新执行，即修改 modify_var
}
int main()
{
    DWORD handler = (DWORD)_except_handler;
    __asm       // 将自己写的异常处理程序入口地址加入异常处理注册记录链表
    {     push handler     // handler 函数的地址
          push FS : [0]    // 原 EXECEPTION_REGISTRATION_RECORD 的头指针
          mov FS : [0], esp // FS:[0] 保存新的异常注册记录头指针
    }
    printf("before    modification:    %d\n", modify_var);
    __asm
    {     mov eax, 0
          mov[eax], 1 // 此处会引发一个异常，访问地址非法
    }
    printf("After     modification:    %d\n", modify_var);
    __asm
    {     // 移除 新加的 EXECEPTION_REGISTRATION_RECORD 节点，恢复原异常处理
          mov eax, [esp]
          mov FS : [0], eax
          add esp, 8
    }
    return 0;
}
```

在本例中，我们自己采用汇编语言完成了异常处理程序的注册工作。程序运行后将显示：

```
Before    modification: 0
After     modification: 1
```

从程序中，可以看淡，在 main 函数的开头用嵌入汇编语句的方法，将自己写的异常处理程序入口地址加入异常处理注册记录链表，在程序运行结束前，又将注册记录从链表上摘除。

12.2.3 全局异常处理程序的注册

在 12.2.2 节中介绍的是每个线程都有自己的异常处理函数入口地址链表，链表最末一项是操作系统在装入线程时设置的指向 UnhandledExceptionFilter 函数。UnhandledExceptionFilter 是整个进程作用范围内的异常处理函数，也是最后处理异常的机会。如果自己的程序未注册全局异常处理程序，则会使用操作系统默认的 UnhandledExceptionFilter 函数，该函数总是向用户显示“Application error”对话框。

如果在程序中使用 API 函数 SetUnhandledExceptionFilter 注册了全局异常处理函数，则用该函数代替操作系统提供的默认函数。操作系统内部使用一个全局变量来记录这个顶层的处理函数，因此只有一个全局性的异常处理函数，而线程内的异常处理函数可以有多个。

【例 12.2】用汇编语言编写异常处理程序

```
.686P
.model flat,c
 option casemap:none    ; 必须先申明大小写敏感，否则对 windows.inc 处理会有错
 include windows.inc    ; 见第 13 章中的说明
 SetUnhandledExceptionFilter proto stdcall :DWORD
 MessageBoxA proto stdcall :DWORD,:DWORD, :DWORD,:DWORD
 ExitProcess proto stdcall :DWORD
 sprintf  proto  :vararg
 includelib      msvcrt.lib ; 见第 13 章中的说明
.data
  szMsg      db  "exception occur at %08x, code: %08x, flag :%08x",0
  szTitle    db  "异常已处理",0
  szContent  db  "运行正常了",0
  szNotOccur db  "该信息不会显示",0
  lpOldHandle  dd  ?
.code
; 自定义的异常处理程序 MyExceptionProcess
MyExceptionProcess  proc  lpExceptionPoint
    local szBuf[256]:byte
    pushad
    mov  esi, lpExceptionPoint
    mov  edi, (EXCEPTION_POINTERS ptr [esi]).ContextRecord
    mov  esi, (EXCEPTION_POINTERS ptr [esi]).pExceptionRecord
    invoke sprintf, addr szBuf, addr szMsg, [edi].CONTEXT.regEip,
                   [esi].EXCEPTION_RECORD.ExceptionCode,
                   [esi].EXCEPTION_RECORD.ExceptionFlags
    invoke MessageBoxA, NULL,addr szBuf,NULL, MB_OK
    mov   [edi].CONTEXT.regEip, offset SafePlace
    popad
    mov  eax,EXCEPTION_CONTINUE_EXECUTION
    ret
MyExceptionProcess  endp
;  自定义异常处理程序结束。下面是主程序
start:
   invoke SetUnhandledExceptionFilter, addr  MyExceptionProcess
    mov  lpOldHandle,  eax
    xor  eax,eax
     mov  dword ptr [eax], 0    ; 产生异常的语句，非法访问
          ; 下面的 MessageBoxA 函数不会执行
          ; 在异常处理程序中修改了 EIP，使之指向了 SafePlace
     invoke MessageBoxA,NULL,addr szNotOccur, addr szTitle,MB_OK
          ; 异常处理后，从 SafePlace 处开始执行
SafePlace:
     invoke MessageBoxA,NULL,addr szContent, addr szTitle,MB_OK
     invoke SetUnhandledExceptionFilter, lpOldHandle
```

```
    invoke ExitProcess, 0
END start
```

在 C 语言程序中，可以用如下形式的函数来定义异常处理程序：

```
long __stdcall  MyExceptionProcess(EXCEPTION_POINTERS *excp)
{
    ……
   return EXCEPTION_CONTINUE_EXECUTION;
}
```

在主程序中，使用如下语句实现异常处理程序的注册。

```
SetUnhandledExceptionFilter(MyExceptionProcess);
```

12．3 C 异常处理程序反汇编分析

【例 12.3】 用 C 语言编写的一个简单异常处理程序，在出现除 0 异常后，显示异常已处理，然后继续运行。

完整的程序 test.cpp 如下。

```
#define EXCEPTION_EXECUTE_HANDLER    1
#include <stdio.h>
int main() {
    int    x = 10;
    int    y = 0;
    int    result;
    __try {
        printf("enter try block .....\n");
        result = x / y;
        printf("this message will not occur .....\n");
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        printf("exception is processing    .... over \n");
    }
    printf("program    continue ..... hello \n");
    return 0;
}
```

运行该程序，将显示：

```
enter try block .....
this message will not occur .....
program    continue ..... hello
```

从该结果来看，“result = x / y; ”之下的“printf("This message will not occur .....\n");”未执行。简单地说，因为 y=0，执行 x/y 时发生了异常，CPU 自动地调用异常处理程序，进入__except 块。如果将 y 的初值修改为 2，则程序的运行结果为：

```
enter try block .....
exception is processing    .... over
program    continue ..... hello
```

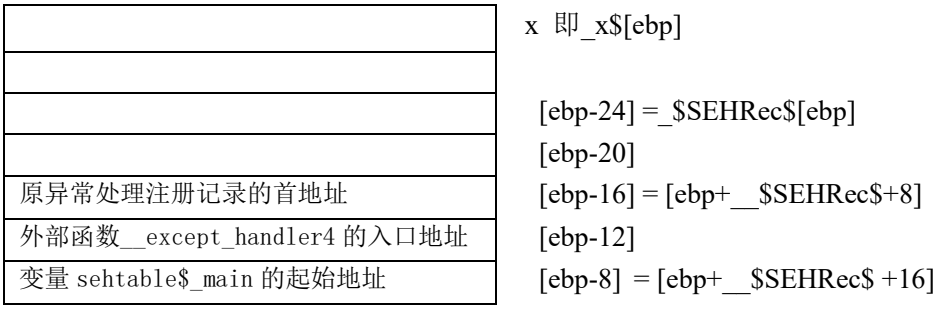
下面，从机器语言的角度来分析上述过程是如何实现的。将 VS2019 下工程的“项目属性

→C/C++→输出文件→汇编源程序输出”设置为“带源代码的程序集 (/FAs)”，在生成执行程序时，将产生汇编源程序。下面仅给出与异常处理有关的部分代码。

```
EXTRN   __except_handler4:PROC
EXTRN   ___security_cookie:DWORD
xdata$x    SEGMENT
__sehtable$_main DD 0ffffffeH
DD  00H
DD  0fffffef4H
DD  00H
DD  0ffffffeH
DD  FLAT:$LN13@main
DD  FLAT:$LN6@main
xdata$x    ENDS
_TEXT    SEGMENT
_result$ = -56                          ; size = 4
_y$ = -44                               ; size = 4
_x$ = -32                               ; size = 4
__$SEHRec$ = -24                            ; size = 24
_main      PROC                                 ; COMDAT
; 4       : int main() {
push ebp
mov ebp, esp
push -2                             ; ffffffeH
push OFFSET __sehtable$_main        ; sehtable$_main  数据区中存放异常处理需要的信息
push OFFSET __except_handler4      ; 这是一个外部函数的入口地址
mov eax, DWORD PTR fs:0             ; 将原异常处理注册记录的首地址压栈
push eax
mov eax, DWORD PTR ___security_cookie
xor   DWORD PTR __$SEHRec$[ebp+16], eax
xor   eax, ebp
push eax
lea    eax, DWORD PTR __$SEHRec$[ebp+8]
mov DWORD PTR fs:0, eax          ; 新异常处理注册记录的入口地址
mov DWORD PTR __$SEHRec$[ebp], esp
        ……
; 6       :   int    x = 10;
mov DWORD PTR _x$[ebp], 10                  ; 0000000aH
; 7       :   int    y = 2;
mov DWORD PTR _y$[ebp], 2
; 8       :   int    result;
; 10      :   __try {
mov DWORD PTR __$SEHRec$[ebp+20], 0
; 11      :           printf("enter try block .....\n");
push OFFSET ??_C@_0BH@MMMFMBDH@enter?5try?5block?5?4?4?4?4?4?6@
```

```
call  _printf
add  esp, 4
; 12     :          result = x / y;
mov eax, DWORD PTR _x$[ebp]
cdq
idiv  DWORD PTR _y$[ebp]
mov DWORD PTR _result$[ebp], eax
; 13     :          printf("This message will not occur .....\n");
push OFFSET ??_C@_0CD@JPAGMIHA@This?5message?5will?5not?5occur?5?4?4?4@
call  _printf
add  esp, 4
; 14     :  }
mov DWORD PTR __$SEHRec$[ebp+20], -2      ; fffffffeH
jmp  SHORT $LN8@main
$LN13@main:
; 15     :  __except (EXCEPTION_EXECUTE_HANDLER) {
mov eax, 1
ret   0
$LN6@main:
mov esp, DWORD PTR __$SEHRec$[ebp]
; 16     :          printf("exception is processing    .... over \n");
push OFFSET ??_C@_0CF@NDLHPMFO@exception?5is?5processing?5?5?4?4?4?4?5o@
call  _printf
add  esp, 4
; 14     :  }
mov DWORD PTR __$SEHRec$[ebp+20], -2      ; fffffffeH
$LN8@main:
; 17     :  }
; 18     :  printf("program    continue ..... hello \n");
      ……
mov ecx, DWORD PTR __$SEHRec$[ebp+8]   ; 恢复异常处理注册记录链表
mov DWORD PTR fs:0, ecx
      ……
_main     ENDP
_TEXT    ENDS
```

注意，上述汇编源程序是从编译结果中摘选的部分，完整的代码需要读者自己去实践生成。程序运行时的栈如图 12.6 所示。



-2	[ebp-4]
原(ebp)	←ebp

图 12.6 程序运行的堆栈示意图

在本例程序中，使用了一个宏定义“#define EXCEPTION_EXECUTE_HANDLER 1”。该宏定义出现在 excpt.h 中。这是异常处理的一个返回值，表明异常处理完毕。

C 程序中的语句： __except (EXCEPTION_EXECUTE_HANDLER) 编译译的结果是：

```
mov eax, 1
ret  0      ; 等价于 ret
```

在 excpt.h 中还定义了异常处理的另外两个返回值：

EXCEPTION_CONTINUE_SEARCH ；表明异常没被识别，交由上一级处理函数处理；

EXCEPTION_CONTINUE_EXECUTION：表明忽略此异常，从异常点继续运行。如果此时再发生异常，还会调用异常处理函数。

对比用汇编语言编写的程序，会直观感受到：对 C 语言程序编译时，会自动建立和恢复异常处理注册记录链表，使用FS:[0]指向链表表头，在异常处理所需信息的数据区__sehtable$_main中，会设置异常处理程序的入口地址。

习题 12

12.1 什么是中断和异常？

12.2 引起中断的原因有哪些？

12.3 什么是中断描述符表？

12.4 简述确定中断和异常处理程序的入口地址的过程。

12.5 简述中断和异常响应的过程。

12.6 简述中断返回的处理过程。

12.7 中断和异常处理程序的调用与一般的子程序调用有何共同之处和差异之处？

上机实践 12

12.1 从 CPU 运行的角度，分析如下 C 程序中的异常处理的运行机理。

说明：本程序的后缀用 .c 和.cpp 均可。

```
#include <stdio.h>
#include <windows.h>
#include <winternl.h>
int exception_filter(LPEXCEPTION_POINTERS p_exinfo)
{
    printf("Error address %x \n", p_exinfo->ExceptionRecord->ExceptionAddress);
    printf("Error Code %08x \n", p_exinfo->ExceptionRecord->ExceptionCode);
    printf("CPU register: \n");
    printf("eax= %08x ebx=%08x  ecx=%08x  edx=%08x \n", p_exinfo->ContextRecord->Eax,
```

```
            p_exinfo->ContextRecord->Ebx, p_exinfo->ContextRecord->Ecx,
            p_exinfo->ContextRecord->Edx);
        if (p_exinfo->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
        {   puts("存储保护异常"); }
        if (p_exinfo->ExceptionRecord->ExceptionCode == EXCEPTION_INT_DIVIDE_BY_ZERO)
        {           puts("被 0 除异常");   }
        return 1;
}
void main()
{
        int   x = 10;
        int   y = 0;
        int   result = 3;
        int   select;
        TEB* p;
        puts("hello");
        scanf_s("%d", &select);
        __try {
            if (select == 1) {
                int* p;
                p = 0;
                *p = 45;  // 该语句会导致一个异常   EXCEPTION_ACCESS_VIOLATION
                puts("This Message will not occur ...ACCESS_VIOLATION ");
            }
            if (select == 2) {
                result = x / y;   // 该语句会导致一个异常   EXCEPTION_INT_DIVIDE_BY_ZERO
                puts("This Message will not occur ...INT_DIVIDE_BY_ZERO ");
            }
            if (select != 1 && select != 2) {
                puts("no exception occur");
            }
        }
        __except (exception_filter(GetExceptionInformation()))
        {    puts("异常已处理");   }
        puts("world");
        getchar();
}
```

12.2 从 CPU 运行的角度，分析如下 cpp 程序中的异常处理的运行机理。

说明，本程序是一个 cpp 程序，该文件名的后缀不能用.c。

```
#include <stdio.h>
double divide(const double& dividend, const double& divisor) {
        if (divisor == 0) {
            throw "除数为 0！";
```

```
        }
        return dividend / divisor;
}
int main() {
        double num1 = 10, num2 = 0;
        double result = -1;
        try {
                result = divide(num1, num2);
        }
        catch (const char* msg) {    // 捕获相应类型的异常
                printf("%s\n", msg);
        }
        catch (...) {    // 能处理任何类型的异常
                printf("error\n");
        }
        printf("%.2f\n", result);
        return 0;
}
```