華中科技大學

异常控制流

金良海

华中科技大学计算机科学与技术学院

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{    int i,j;
     int r = 0;
     char msg[20];
     for (i=0;i<20000;i++) {
        r = 0;
        for (j = 0;j < 100000;j++)
           r = r + 1;
     }
     printf("finish for ... \n");
     scanf("%s", msg);
     printf("your input : %s \n", msg);
     return 0;
}
```

循环语句的执行时间较长，循环执行后显示 finish for ...

Q：在出现finish…之前，输入一行 123456，运行结果结果是什么？

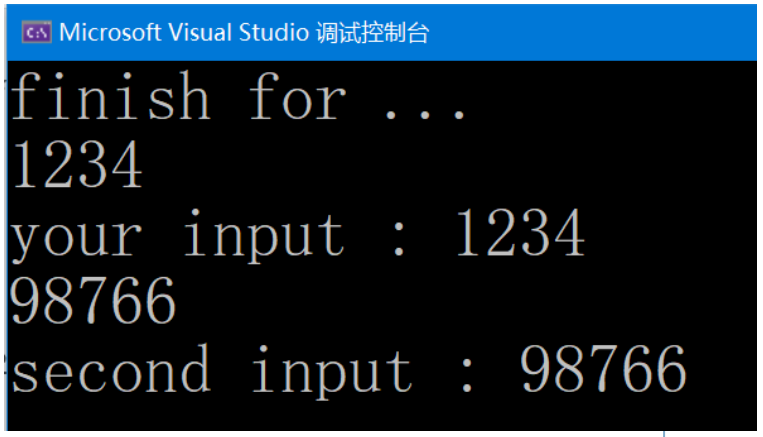
```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int main()
{    int i,j;
     int r = 0;
     char msg[20];
     for (i=0;i<20000;i++) {
          ……
     }
     printf("finish for ... \n");
     scanf("%s", msg);
     printf("your input : %s \n", msg);
     scanf("%s", msg);
      printf("second input : %s \n", msg);
     return 0;
}
```

Q：在出现finish…之前，

输入一行 1234，

再输入一行 ：98766

运行结果结果是什么？



Q：如何解释看到的现象？


```
#include <iostream>
#include <windows.h>
using namespace std;
#pragma comment(lib,"Winmm.lib")
#define delaytime 100
void CALLBACK TimeEvent(UINT uTimerID, UINT uMsg, DWORD_PTR dwUser,
DWORD_PTR dw1, DWORD_PTR dw2)
{    printf("\nHello\n");  }
int main()
{   int  i = 0, timeID;
    timeID = timeSetEvent(delaytime,10, //每100ms中断1次,10表示精度
    (LPTIMECALLBACK)TimeEvent,1,TIME_PERIODIC); //1为回调参数
    char displaychar = 'A';
    for (i = 0;i < 5000;i++) {
        printf("%c", displaychar);
        displaychar++;
        if (displaychar == 'z' + 1) displaychar = 'A';
    }
    timeKillEvent(timeID);
    return 0;
}
```

运行结果片段

```
wxyzABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_ abcdefghijklmnopqrstuvwx
yzABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrs
Hello
tuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstu
vwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvw
```

```
defghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcde
fghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR
Hello
STUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRST
UVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV
```

第12章 中断和异常处理

華中科技大學

12.1 中断与异常的基础知识

中断和异常的概念、中断描述符表

中断和异常的响应过程、软中断指令

12.2 Windows中的结构化异常处理

编写异常处理函数

异常处理程序的注册

全局异常处理程序的注册

12.3 C异常处理程序反汇编分析

12.1.1 中断和异常的概念

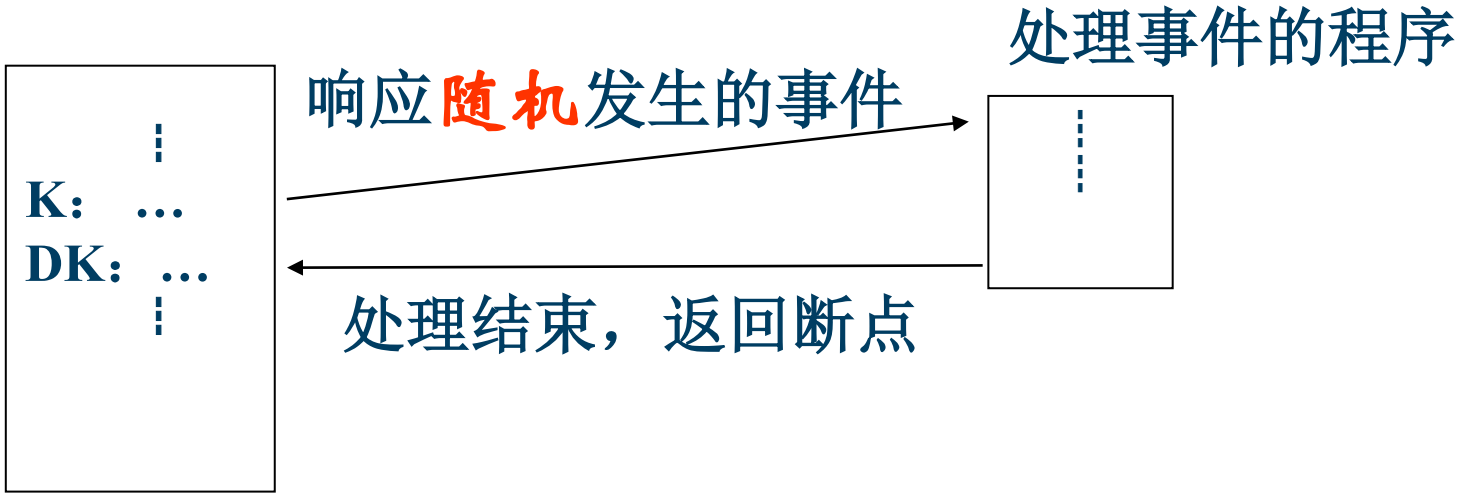
日常生活当中的“中断”

计算机世界中的“中断”

中断是CPU所具有的能打断当前执行的程序，转而为临时出现的事件服务，事后又能自动按要求恢复执行原来程序的一种功能。

12.1 中断与异常的基础知识

中断处理过程：



Q：与子程序调用有什么差别？

事先安排好的　VS　随机发生的

有转移指令　VS　无转移指令

12.1 中断与异常的基础知识

12.1.1 中断和异常的概念

(1) 为什么要引入中断机制？

(2) 有哪些事情会引起中断？ 中断源

(3) CPU为什么能感知中断？ 中断系统

(4) 在何处去找中断处理程序？ 中断描述符表
中断矢量表

(5) 如何从中断处理程序返回？

(6) 如何使用中断？

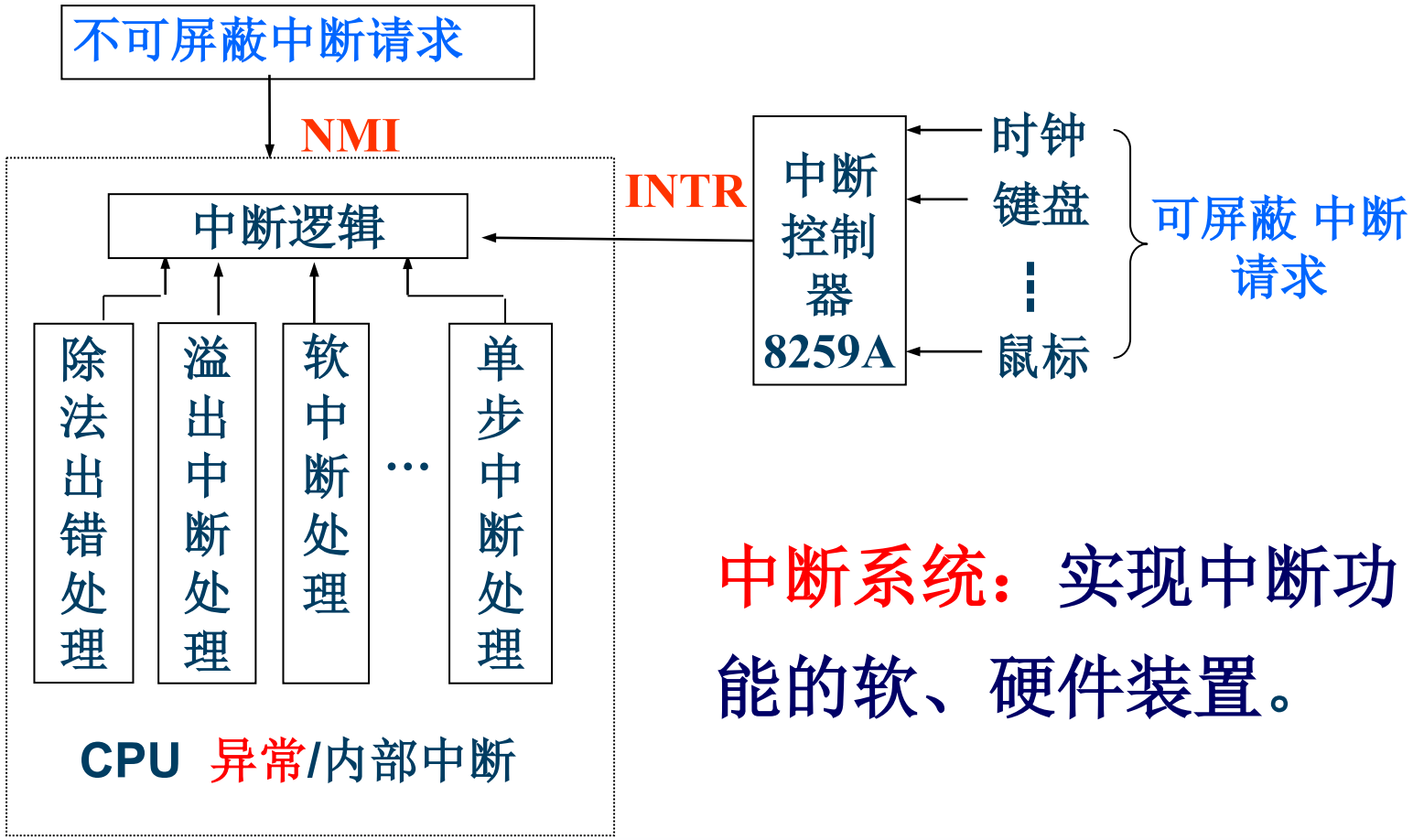
12.1 中断与异常的基础知识

中断源分类

中断源

- 外部中断（中断，随机性）
  - 不可屏蔽中断NMI:
    电源掉电、存储器出错
    或者总线奇偶校验错
  - 可屏蔽中断INTR:
    键盘、鼠标、时钟……
    开中断状态（STI，IF=1）
    关中断状态（CLI，IF=0）
- 内部中断（异常，与CPU的状态和当前执行的指令有关）
  - CPU检测:
    除法出错、单步中断、
    协处理器段超越等。
  - 程序检测:
    软中断，包括指令INTO、
    INT n 和 BOUND等。

12.1 中断与异常的基础知识



中断系统：实现中断功能的软、硬件装置。

12.1 中断与异常的基础知识

1、优先级

中断/异常类型	优先级
除调试故障以外的异常 异常指令INTO、INT n、INT 3 对当前指令的调试异常 对下条指令的调试异常 NMI INTR	最高 ↓ 最低

12.1 中断与异常的基础知识

Intel 8086/8088 称外中断、内中断

从 80286 开始，称为中断、异 常

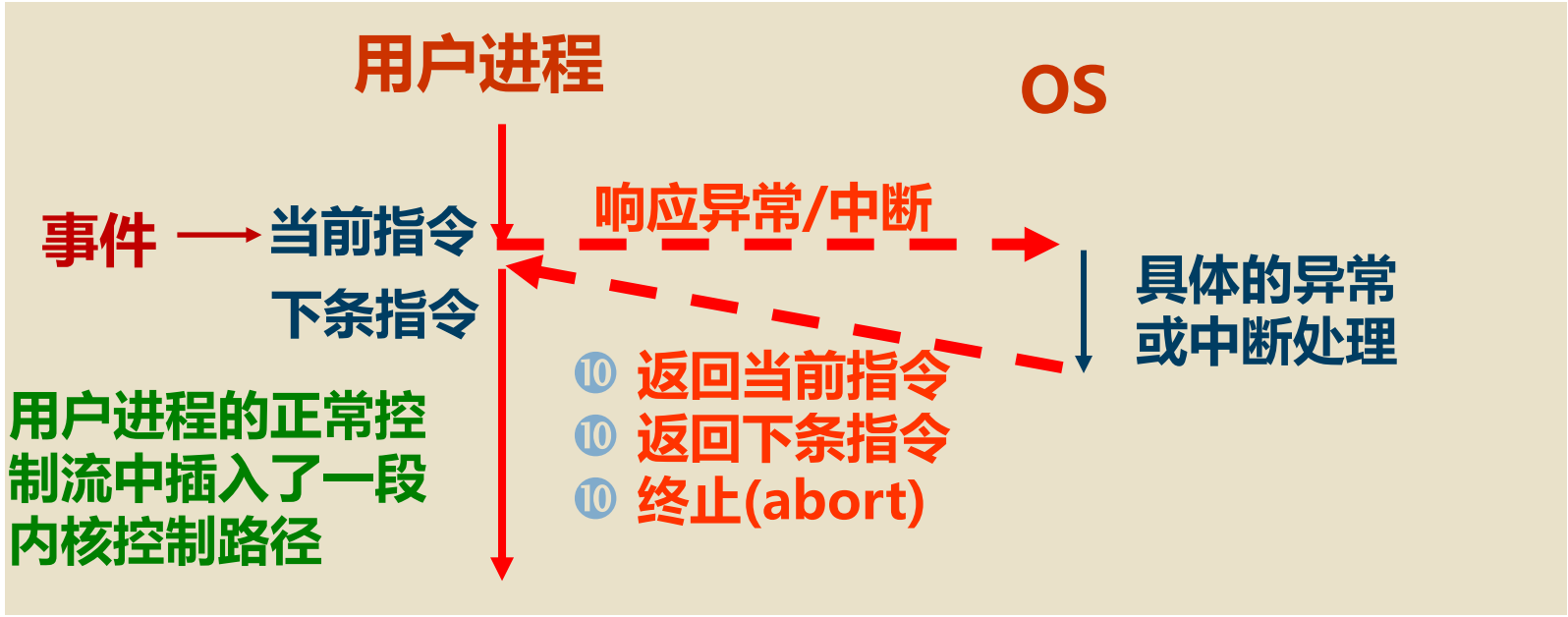
- 中断：由外部设备触发、与正在执行的指令无关、异步事件
- 异常：与正在执行的指令相关的 同步事件；CPU内部出现的中断，也称为同步中断。

一条指令的执行过程中，CPU检测到了某种预先定义条件，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。

12.1 中断与异常的基础知识

- 在Intel CPU中，异常分为三类：
  故障（faults）、陷阱（traps）、中止（aborts）。
- 在异常处理程序执行后，后续操作取决于异常的类型：
  - 重新执行引起异常的指令 —— 故障
  - 执行引起异常指令之下的指令 —— 陷阱
  - 终止程序运行 —— 中止
- 中断处理程序执行后会返回到被中断处继续执行

12.1 中断与异常的基础知识



12.1 中断与异常的基础知识

- 异常 —— 故障

  - 故障异常是在引起异常的指令之前或者指令执行期间，在检测到故障或者预先定义的条件不能满足时产生。
  - 常见的故障异常

    除法出错（除数为0；除数很小被除数很大，商溢出）

    数据访问越界（访问一个不准本程序访问的内存单元）

    缺页
  - 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

12.1 中断与异常的基础知识

- 异常 —— 陷阱
  - 在执行引起陷阱异常的指令后，把异常情况通知给系统。
  - 执行异常处理程序后，回到产生异常信号指令之下的一条语句。
  - 软中断是一种常见的陷阱。所谓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。
  - 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服务程序。
  - 另外一种常见的陷阱异常是单步异常，用于防止一步步跟踪程序。

12.1 中断与异常的基础知识

- 异常 —— 中止
  - 中止是在系统出现严重问题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。

12.1.2 中断描述符表

- 每一个中断或异常处理程序都有一个入口地址；
- 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）
  - 陷阱门（主要用于处理异常）
- 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。

12.1.2 中断描述符表

中断号	名　称	类型	相关指令	DOS下名称
0	除法出错	异常	DIV，IDIV	除法出错
1	调试异常	异常	任何指令	单步
2	非屏蔽中断	中断	-	非屏蔽中断
3	断点	异常	INT 3	断点
4	溢出	异常	INTO	溢出
5	边界检查	异常	BOUND	打印屏幕
6	非法操作码	异常	非法指令编码或操作数	保留
7	协处理器无效	异常	浮点指令或WAIT	保留

8	双重故障	异常	任何指令	时钟中断
9	协处理器段超越	异常	访问存储器的浮点指令	键盘中断
0DH	通用保护异常	异常	任何访问存储器的指令 任何特权指令	硬盘（并行口）中断
10H	协处理器出错	异常	浮点指令或WAIT	显示器驱动程序
13H	保留			软盘驱动程序
14H	保留			串口驱动程序
16H	保留			键盘驱动程序
17H	保留			打印驱动程序
19H	保留			系统自举程序
1AH	保留			时钟管理
1CH	保留			定时处理
20H~2FH	其它软/硬件中断			DOS使用
0~0FFH	软中断	异常	INT n	软中断

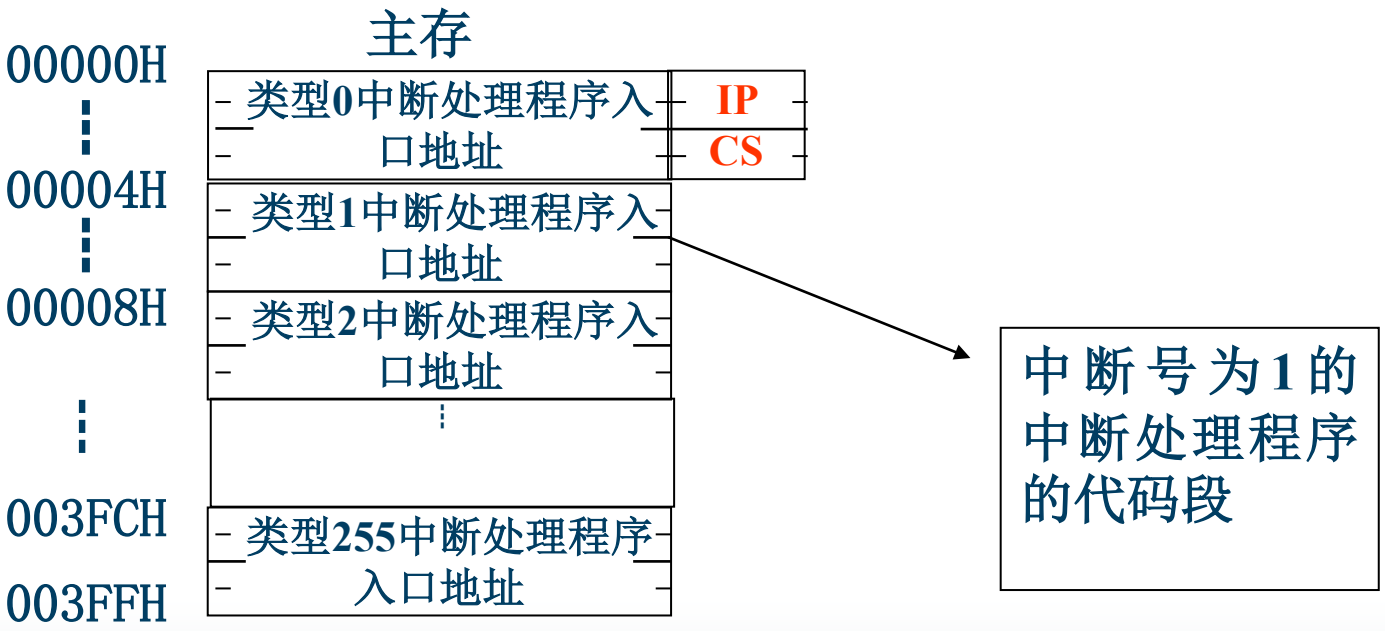
12.1.2 中断描述符表

中断类型码与对应的中断处理程序之间的连接表，存放的是中断处理程序的入口地址（也称为中断矢量或中断向量）。

12.1.2 中断描述符表

➢ 实方式下的中断矢量表

大小为1KB，起始位置固定地从物理地址0开始



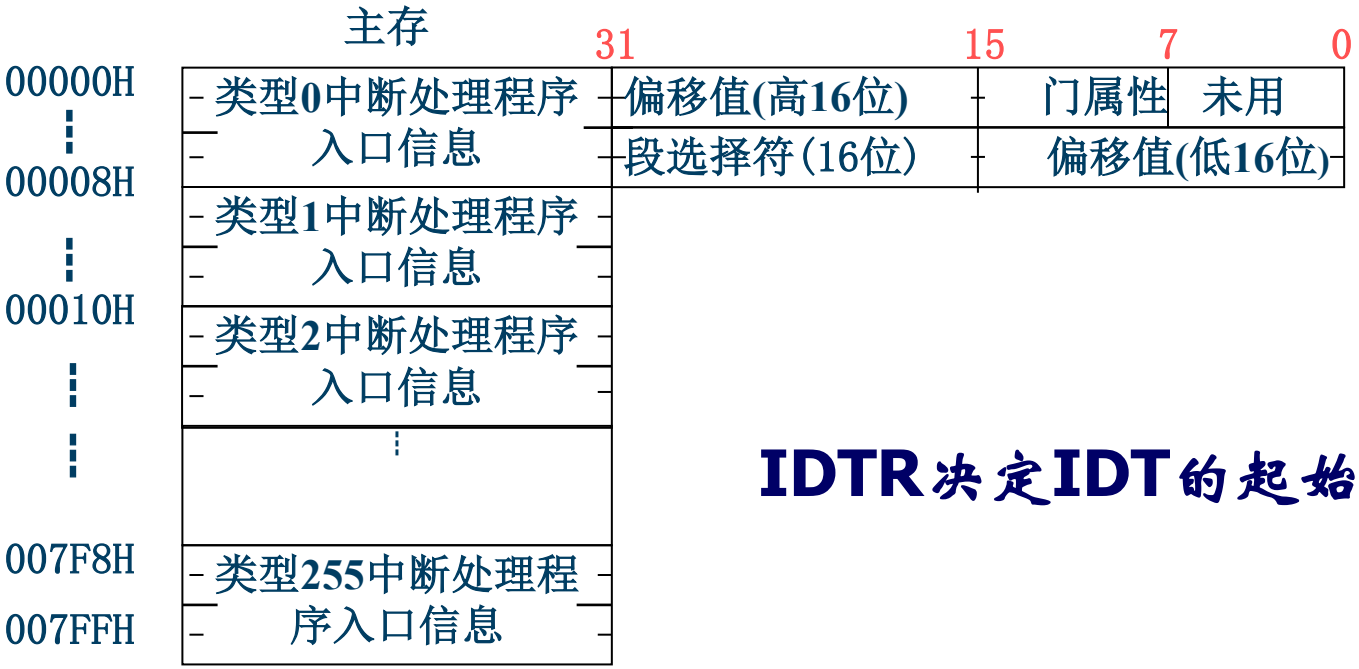
12.1.2 中断描述符表

➢保护方式下的中断矢量表

- 在保护方式下，中断矢量表称作中断描述符表(IDT，Interrupt Descriptor Table)
- 按照统一的描述符风格定义其中的表项；
- 每个表项(称作门描述符)存放中断处理程序的入口地址以及类别、权限等信息，占8个字节，共占用2KB的主存空间。

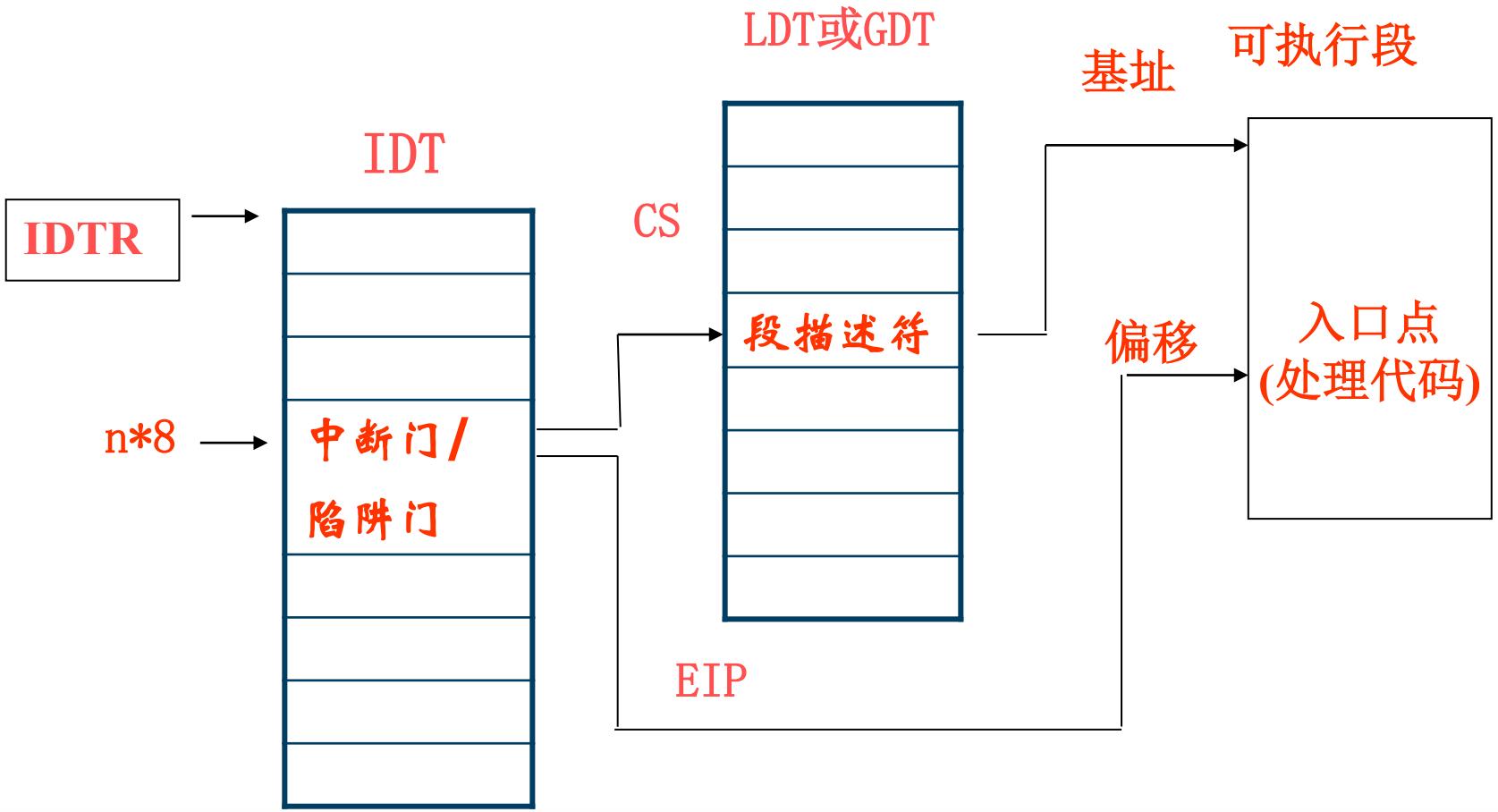
12.1.2 中断描述符表

保护方式下的中断描述符表



IDTR决定IDT的起始PA。

12.1.2 中断描述符表



从中断号到中断处理程序的转换过程

12.1.4 软中断指令

软中断通过程序中的**软中断指令**实现，所以又称它为**程序自中断**。

1. 软中断指令

格式：INT　n

功能：

（EFLAGS）→↓（ESP），

0→TF，　中断门还要将0→IF

12.1.4 软中断指令

②（CS）扩展成32位→↓（ESP）

从门或TSS描述符中分离出的段选择符→CS

③（EIP）→↓（ESP）

从门或TSS描述符中分离出的偏移值→EIP

12.1.4 软中断指令

2. 中断返回指令

格式：IRET

功能：① ↑（ESP）→EIP

② ↑（ESP）取低16位→CS

恢复断点地址

③ ↑（ESP）→EFLAGS

恢复标志寄存器的内容

实方式下中断处理程序示例

■ 上机演示

◆ 查看 INT 21H 中断处理程序的入口地址

内存的最低端，84H处

数据区　　显示 0 ：84 处的内容

◆ 跟踪进入 中断处理程序 （Alt+F7）

查看 CS、IP是否为 上一步看到的程序入口地址

查看 堆栈是否存放的INT 21H 之下的断点地址

实方式下中断处理程序示例

- 新增一个中断处理程序
- 修改已有的中断处理程序以扩充其功能

实方式下中断处理程序示例

1．新增一个中断处理程序的步骤

① 编制中断处理程序。
与子程序的编制方法类似，远过程, IRET。

② 为软中断找到一个空闲的中断号m；
或根据 硬件确定中断号。

③将新中断处理程序装入内存，将其入口地址
送入中断矢量表4*m - 4*m+3的四个字节中。

实方式下中断处理程序示例

1．新增一个中断处理程序的步骤

Q: 如何得到新中断处理程序的入口地址?

方法一：SEG 子程序的名称
OFFSET 子程序的名称

方法二：INTP_ADDRESS DD 子程序的名称

方法三：直接使用CS，当子程序与主程序在同一个段时

实方式下中断处理程序示例

1．新增一个中断处理程序的步骤

Q: 如何得到新中断处理程序的入口地址?

Q: 如何将该入口地址写入中断矢量表?

方法1： 直接写中断矢量表的相关位置;
方法2： DOS系统功能调用

实方式下中断处理程序示例

例：读出 08H 号中断处理程序的入口地址。

方法1：直接读中断矢量表的相关位置

```
MOV   AX，0
MOV   DS，AX
MOV   AX，ds:[0020H]      ；访问DS：[08H*4]单元
                          ；即0：0084H单元
MOV   BX，ds:[0034]
```

方法2： DOS系统功能调用

```
MOV    AX，3508H
INT    21H
```

(ES):(BX) 为 中断 08H 服务程序的入口地址

实方式下中断处理程序示例

1．新增一个中断处理程序的步骤

Q: 如何得到新中断处理程序的入口地址?

Q: 如何将该入口地址写入中断矢量表?

Q: 如何调用新的中断处理程序呢?

实方式下中断处理程序示例

2、修改已有中断处理程序以扩充其功能

- 编制程序段(根据扩充功能的要求，应注意调用原来的中断处理程序)
- 将新编制的程序段装入内存;
- 用新编制程序段的入口地址取代中断矢量表中已有中断处理程序的入口地址。

Q: 如何调用老的中断处理程序呢?
直接使用 INT * ，行不行呢?**

实方式下中断处理程序示例

2、修改已有中断处理程序以扩充其功能

要求：

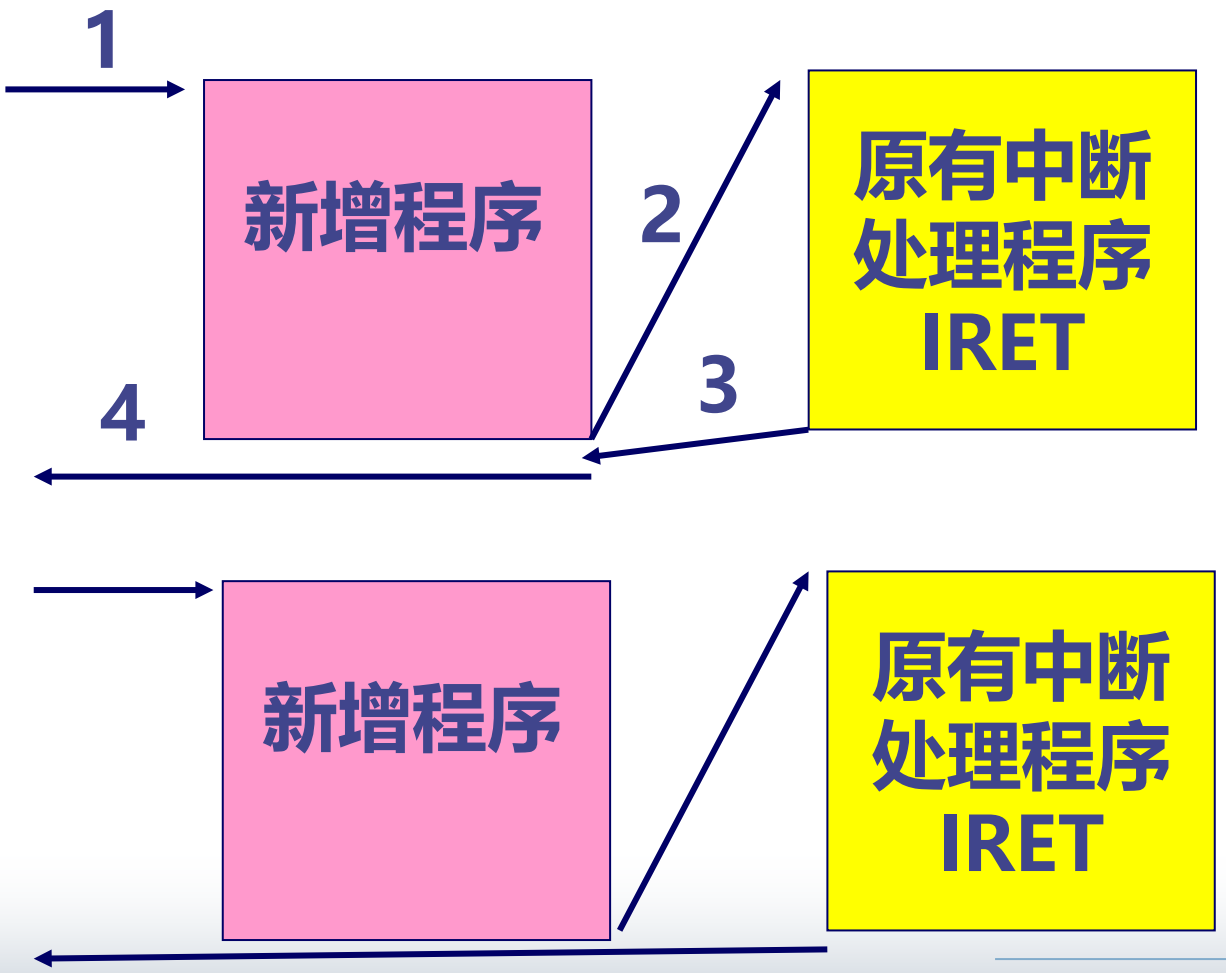
- **扩充后的程序 调用原来的中断处理程序**
  即保留原有程序的功能
- **不能改原有的中断处理程序**

新增程序

原有中断处理程序 IRET

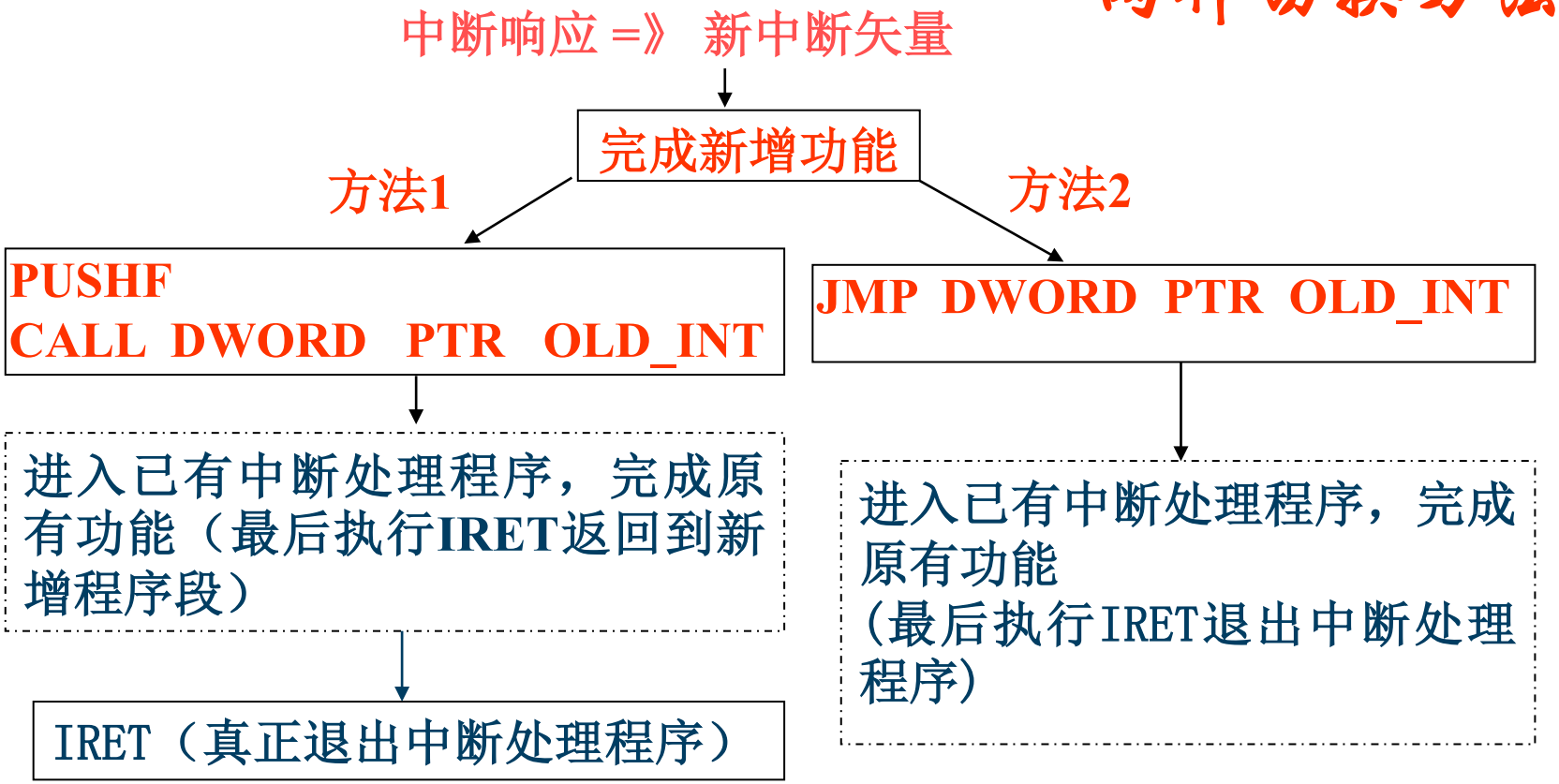
实方式下中断处理程序示例

2、修改已有中断处理程序以扩充其功能



实方式下中断处理程序示例

两种切换方法



实方式下中断处理程序示例

例: 编制时钟显示程序.

要求每隔1秒在屏幕的右上角显示时间。

(扩充原中断的功能)

在该程序运行结束后，时间显示仍然继续。

在运行其它程序时，还看得到显示的时间。

实方式下中断处理程序示例

程序涉及的知识要点分析:

(1)如何知道是否到达1秒？用什么中断合适?
(2)如何取当前时间?
(3)如何在指定位置显示时间?
(4)如何在显示时间后（改变了光标的位置），
　　不影响其他程序的运行?
(5)如何在退出程序后，仍能显示时间?

实方式下中断处理程序示例

程序涉及的知识要点分析:

(1)每隔1秒

定时中断，时钟中断

<PC中断大全 BIOS,DOS及第三方调用的程序员参考资料>

<PC中断调用大全>

由8254系统定时器的0通道每秒产生18.2次，该中断用于时钟更新。

实方式下中断处理程序示例

程序涉及的知识要点分析:

(1)每隔1秒

引入一个变量，记录进入中断处理程序的次数。
当达到18次时，取时间，然后显示时间。

(2) 在屏幕的右上角显示时间

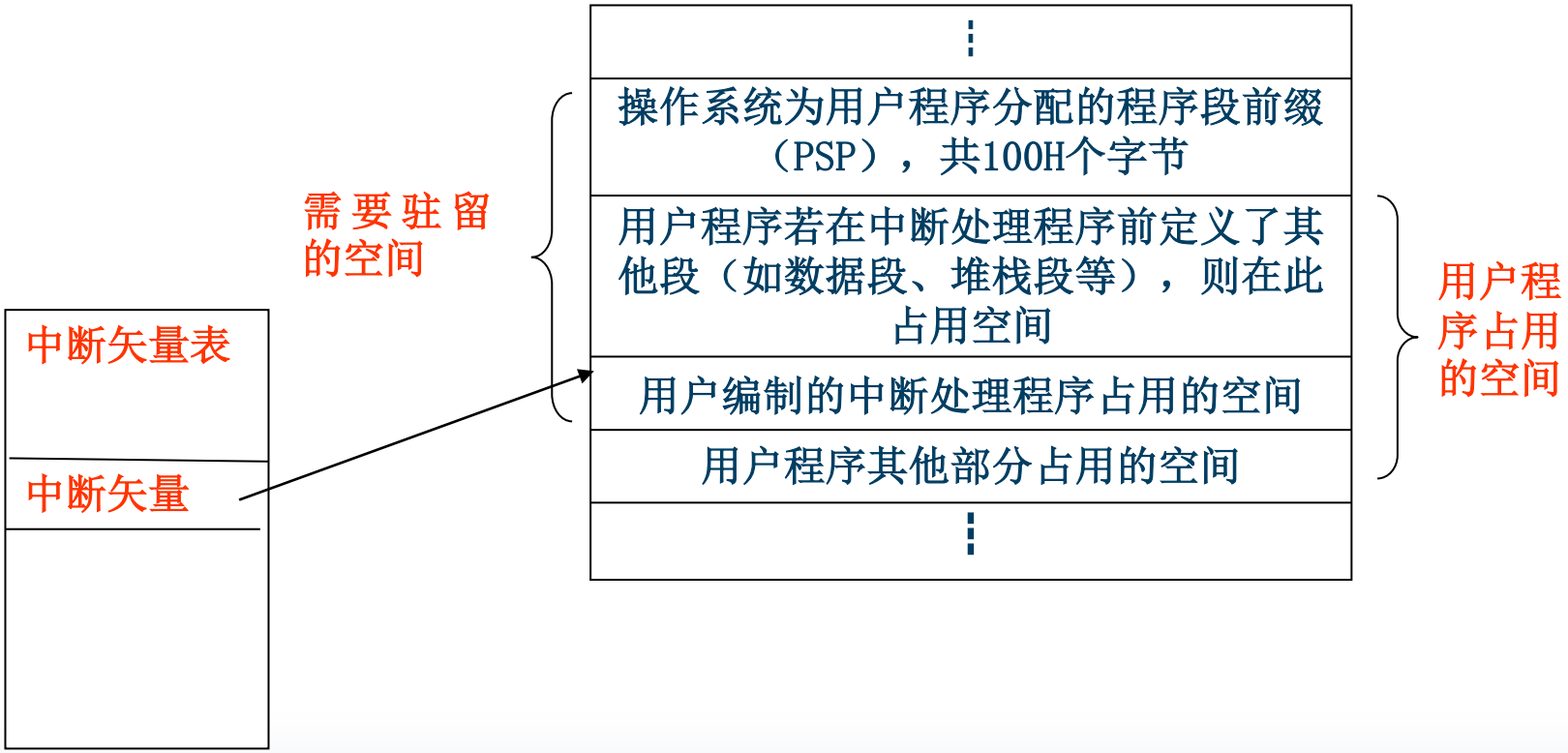
常用BIOS子程序,显示器驱动程序

实方式下中断处理程序示例

(3) 程序驻留

一个程序所占的主存储空间，在该程序运行结束后，不被回收。

实方式下中断处理程序示例



12.2 Windows中的结构化异常处理

華中科技大學

Windows系统：结构化异常处理

Structured Exception Handling，SEH

Windows XP： 向量化异常处理

Vectored Exception Handling，VEH

C++异常处理

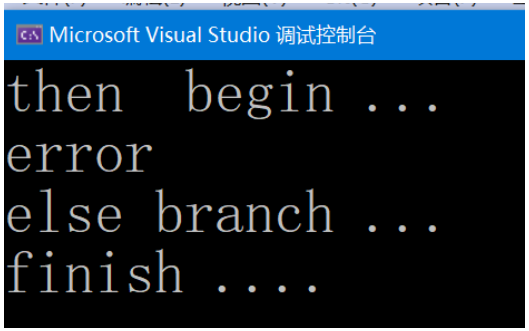
C++ Exception Handler，C++EH

12.2 Windows中的结构化异常处理

- 如何感知到出现了异常？
- 异常发生后如何处理？
- 如何去找到异常处理程序？
- 如何编写异常处理程序？
- 如何让系统找到自己的编写的异常处理程序？
- 异常处理程序执行后，再做什么？

12.2 Windows中的结构化异常处理


```
#include <signal.h>          #include <setjmp.h>
#include <stdio.h>           #include <Windows.h>
jmp_buf  buf;
LONG WINAPI Exceptionhandler(EXCEPTION_POINTERS *ExceptionInfo)
{   printf("error \n");
    longjmp(buf, 1);
}
int main()
{   int eax = setjmp(buf);
    if( eax != 0 ) {
        printf("then  begin ...\n");
        Exceptionhandler(NULL);
        printf("then  over ...\n");
    }
    else { printf("else branch ...\n"); }
    printf("finish ....\n");
    return 0;
}
```



如果删除语句
if(eax !=0)，???

12.2 Windows中的结构化异常处理

函数setjmp和longjmp配合使用，可以实现近程/远程GOTO功能（在函数内部/函数之间转跳）。setjmp函数调用的下一条语句就是需要转跳的目的地（Label），longjmp 函数可以实现 GOTO Label（必须在调用longjmp之前先调用setjmp）。setjmp将上下文信息（包括EIP）保存到一个结构变量中，longjmp从这个结构变量中恢复上下文信息（EIP被恢复，也就是实现了GOTO功能）。

int setjmp(jmp_buf env)

功能：将当前程序位置的上下文信息保存到env中。

返回：返回0。以后调用longjmp时就会GOTO到sejmp的下一条语句(相当于直接从setjmp返回), 并将longjmp函数的参数作为sejmp的返回值。

void longjmp(jmp_buf env, int val)

功能：从env中恢复上下文 (EIP被恢复)，val作为setjmp函数的返回值。

val必须是非0，如果val=0，longjmp()会将val修改为1。

12.2 Windows中的结构化异常处理

華中科技大學

```
typedef struct __JUMP_BUFFER
    {
        unsigned long Ebp;
        unsigned long Ebx;
        unsigned long Edi;
        unsigned long Esi;
        unsigned long Esp;
        unsigned long Eip;
        unsigned long Registration;
        unsigned long TryLevel;
        unsigned long Cookie;
        unsigned long UnwindFunc;
        unsigned long UnwindData[6];
    } _JUMP_BUFFER; //jmp_buf
```

12.2 Windows中的结构化异常处理

buf	0x00d6a138 {18
buf[5],x	0x00d618f2
eip,x	0x00d618f2
buf[0],x	0x0113f954
ebp,x	0x0113f954
esp,x	0x0113f880
buf[4],x	0x0113f87c
eax,x	0x00000000

buf[0]: ebp
buf[4]: esp
buf[5]: eip

setjmp 保存现场；
返回eax为 0；
之上有 push 0

```
    if (!setjmp(buf)) {
00D618E6  push        0  ;longjmp有2个参数
00D618E8  push        offset buf (0D6A138h)
00D618ED  call        __setjmp3 (0D61253h)
00D618F2  add         esp,8
00D618F5  test        eax,eax
00D618F7  jne         main+4Ch (0D6191Ch)
        printf("then  begin ...\n");
```

```
    else { printf("else branch ...\n"); }
00D6191C  push        offset string "else branch ...\
00D61921  call        _printf (0D610CDh)
00D61926  add         esp,4
```

12.2 Windows中的结构化异常处理

buf	0x00d6a138 {18086228,
[0]	18086228
[1]	15519744
[2]	14028835
[3]	14028835
[4]	18086012
[5]	14031090
[6]	18086336
[7]	0
[8]	1447244336
[9]	0

```
    longjmp(buf, 1);
00D617AE  push        1
00D617B0  push        offset buf (0D6A138h)
00D617B5  call        _longjmp (0D61343h)
}
```

```
    if (!setjmp(buf)) {
00D618E6  push        0
00D618E8  push        offset buf (0D6A138h)
00D618ED  call        __setjmp3 (0D61253h)
00D618F2  add         esp,8     已用时间 <= 2ms
00D618F5  test        eax,eax
00D618F7  jne         main+4Ch (0D6191Ch)
        printf("then  begin ...\n");
```

Longjmp 跳转成 buf 的状态；但 eax 置成了参数 1

12.2 Windows中的结构化异常处理

Windows异常处理机制 (1)

发生异常时，操作系统可以调用用户编写的异常处理函数。

- C++语言的关键字try ... catch只能捕捉/处理用语句throw显式抛出的异常（不能捕捉其他异常）。
- VC++自己定义了另外的语句_try ... __except（不是C++的关键字），用于捕捉/处理Windows所有的异常。
- try...catch和_try..._except的实现都是基于Windows SEH的（Structured Exception Handling，结构化异常处理）：按照SEH的规则，把异常处理程序加入到当前线程的SEH异常处理链中。

12.2 Windows中的结构化异常处理

Windows异常处理机制 (2)

- _try ..._except 能够处理的所有的Windows异常，C++的try ..catch 只能捕捉用throw显式抛出的异常。

```
try {
        int *p = NULL;
        *p = 0;
    }
catch (...) {
   printf("An exception.");
}
//捕捉不到上面的异常
```

```
__try {
        int *p = NULL;
        *p = 0;
    }
__except (exception filter) {
   printf("An exception.");
}
//可以捕捉到任何异常
```

Windows异常处理机制 (3)

- 对于VC++的_try ..._except，exception filter是一个表达式或者函数调用，表达式的值（函数调用的返回值）决定了处理异常后的下一步操作：

  EXCEPTION_EXECUTE_HANDLER 已经处理异常, 进程终止

  EXCEPTION_CONTINUE_EXECUTION 已经处理异常，从产生异常的指令继续执行

  EXCEPTION_CONTINUE_SEARCH 不处理, 交给Windows处理

12.2 Windows中的结构化异常处理

Windows处理异常的步骤 (1)：

(1)首先交给调试程序进行处理（调试器优先级最高），如果不是调试状态则转(2)；

(2)在当前函数中的try-catch (_try-_except)中查找是否处理该异常。如果没有try-catch (_try-_except) 或者 try-catch (_try-_except) 没有捕捉该异常 或者 _try-_except中表达式的值为 EXCEPTION_ CONTINUE_SEARCH，则转(3)；

(3)依次调用当前进程的SEH处理链节点中的异常处理函数（Windows为每个线程建立一个SEH异常处理链），直到异常处理函数的返回值不是ExceptionContinueSearch。如果所有节点的异常处理函数都返回ExceptionContinueSearch，则转(4)；

12.2 Windows中的结构化异常处理

Windows处理异常的步骤 (2)：

(4)系统再次检测进程是否正在被调试，如果被调试的话，则再一次通知调试器，如果调试器还是没有处理该异常，则转(5)；

(5)系统检查是否有安装筛选器回调函数，如果有则调用它。如果没有筛选器回调函数或者筛选器回调函数的返回值为EXCEPTION_CONTINUE_SEARCH，系统则直接调用默认的异常处理程序终止进程（也就是崩溃的界面）；
（筛选器回调函数使用SetUnhandledExceptionFilter()安装）

(6)如果SEH处理链没有处理当前异常（所有节点的异常处理函数都返回ExceptionContinueSearch），这时即使筛选器回调函数处理了当前异常，那么在终结之前系统会展开SEH链表，依次调用SEH链表中的回调函数（最后清理的机会）。

12.2 Windows中的结构化异常处理

异常处理的编程方法：

根据Windows的异常处理机制，用户程序有3种方法去捕捉和处理异常：

(1) C++规范的 try...catch 和 VC++的 _try ..._except。

(2)使用API函数SetUnhandledExceptionFilter去注册用于处理异常的筛选器回调函数。

(2)在SEH链表的表头插入一个新的结点（新的异常处理函数）。

12.2 Windows中的结构化异常处理

方法1：try...catch、_try ..._except (1)

```
try {
        int *p = NULL;
        *p = 0;
    }
catch (...) {
   printf("An exception.");
}
```

//捕捉不到上面的异常

```
try {
        int *p = NULL;
        if(p == NULL) throw -1;
        else *p = 0;
    }
catch (...) {
   printf("An exception.");
}
```

//使用throw显式抛出的异常才能够被catch捕捉到

12.2 Windows中的结构化异常处理

方法1：try...catch、_try ..._except (2)

```
__try {
        int *p = NULL;
        *p = 0;
    }
__except(EXCEPTION_EXECUTE_HANDLER ) {
        printf("An exception.");
}
//能够捕捉到任何异常
```

方法2：筛选器回调函数

安装用于处理异常的筛选器回调函数：

LPTOP_LEVEL_EXCEPTION_FILTER WINAPI SetUnhandledExceptionFilter(LPTOP_LEVEL_EXCEPTION_FILTER pTopLevelExceptionFilter)

说明：当一个异常没有被处理（ try和SEH）且进程没有处于调试状态（不在VS或者别的调试器里运行），操作系统会调用注册的筛选器回调函数进行处理。

参数：pTopLevelExceptionFilter 是回调函数指针。NULL表示取消已经安装的筛选器回调函数。

返回：返回以前安装的回调函数指针。

12.2 Windows中的结构化异常处理

筛选器回调函数：

回调函数原型：

```
LONG  WINAPI  TopLevelExceptionFilter (
_In_PEXCEPTION_POINTERS  ExceptionInfo)
```

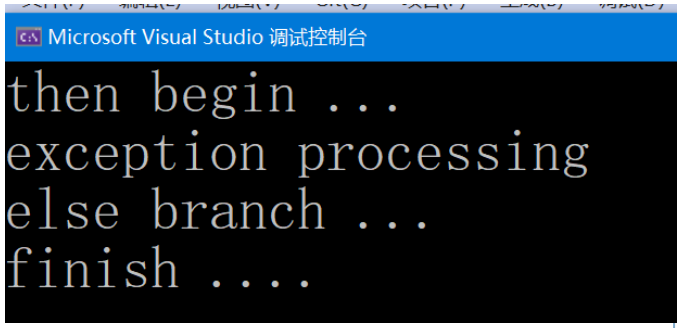
参数：ExceptionInfo 包含异常的详细信息（异常代码、异常地址、异常参数等）。

返回：EXCEPTION_EXECUTE_HANDLER （1）
EXCEPTION_CONTINUE_EXECUTION （-1）
EXCEPTION_CONTINUE_SEARCH （0）

12.2 Windows中的结构化异常处理


```
jmp_buf  buf;
LONG WINAPI Exceptionhandler(EXCEPTION_POINTERS *ExceptionInfo)
{   printf("exception processing \n");
    longjmp(buf, 1);
}
int main()
{   int a,t;
    SetUnhandledExceptionFilter(Exceptionhandler);
    int eax = setjmp(buf);
    if( eax != 0 ) {
        printf("then begin ...\n");
        a = 100;  t = 0;
        a = a / t; //触发一个异常
        printf("then over ...\n");
    }
    else printf("else branch ...\n");
    printf("finish ....\n");
    return 0;
}
```

如果删除语句
if(eax !=0)，???



12.2 Windows中的结构化异常处理


```
14  int main()
15  {   int a, t;
16    //  SetUnhandledExceptionFilter(Exceptionhandler);
17      if (!setjmp(buf)
18          printf("ther
19              a = 100;
20              t = 0;
21              a = a / t;
22              printf("then over ...\n");
```

未经处理的异常

0x00531918 处有未经处理的异常(在 c_divide_zero.exe 中):
0xC0000094: Integer division by zero。

复制详细信息 | 启动 Live Share 会话...

异常设置

使用SetUnhandledExceptionFilter函数设置的是一个筛选器异常处理回调函数。

方法3：插入SEH链表的表头结点 (1)

每个线程都有一个线程信息块TIB（Thread Information Block），用于保存线程的相关信息。TIB的数据结构 NT_TIB 定义在winnt.h中。TIB的地址保存在内存单元 FS:[0]。TIB 的第一个字段为：

struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList。

该字段指向线程的结构化异常处理（SEH）链。链表中结点的结构为：

```
struct EXCEPTION_REGISTRATON {
   prev      dd ?  //前一个节点的指针
   handler   dd ?  //指向_except_handler回调函数的指针
}
```

为了增加一个异常处理程序，需要创建一个新的结点，新结点中的handler指向新的异常处理程序，然后将该结点插入到SHE链表头部。

12.2.2 异常处理程序的注册

方法3：插入SEH链表的表头结点 (2)

```
typedef struct _NT_TIB {
   struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
   PVOID  StackBase;
   PVOID  StackLimit;
   PVOID  SubSystemTib;
#if defined( _MSC_EXTENSIONS )
   union {
      PVOID    FiberData;
      DWORD  Version;
   };
#else
   PVOID  FiberData;
#endif
   PVOID  ArbitraryUserPointer;
   struct _NT_TIB  *Self;
} NT_TIB;
```

方法3：插入SEH链表的表头结点 (3)

SEH处理链是建立在堆栈上的，所以必须在堆栈上创建新的EXCEPTION_REGISTARTION节点，并插入到SEH处理链的头部，如下面的代码：

```
push handler        //新的异常处理函数地址
push FS:[0]         //指向原来的SEH处理链
mov  FS:[0], ESP    //安装新的SHE链表
```

前2个语句实现在堆栈中建立一个新的节点并插入到SEH链的头部，第3个语句将新的SEH处理链的头部节点地址保存到内存单元FS:[0]。

方法3：插入SEH链表的表头结点 (4)

SEH处理链中的异常处理函数原型（excht.h）：

```
EXCEPTION_DISPOSITION _cdecl _except_handler (
  struct  _EXCEPTION_RECORD *ExceptionRecord,
  void    *EstablisherFrame, //指向establisher帧结构的指针（重要）
  struct  _CONTEXT *ContextRecord,  //包含异常发生时寄存器的值
  void    *Dispatchercontext )
```

方法3：插入SEH链表的表头结点 (5)

```
typedef struct  _EXCEPTION_RECORD  {  //winnt.h
    DWORD  Exceptioncode;   //异常的代码
    DWORD  ExceptionFlags;
    struct      _EXCEPTION_RECORD  *ExceptionRecord;
    PVOID     ExceptionAddress;   //异常发生的地址
    DWORD  NumberParameters;
    DWORD  ExceptionInformation
                  [EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

方法3：插入SEH链表的表头结点 (6)

```
typedef struct  _CONTEXT  {
    DWORD  ContextFlags;
    DWORD  Dro, Dr1, Dr2, Dr3, Dr4, Dr5, Dr6, Dr7;
    FLOATING_SAVE_AREA  FloatSave;
    DWORD  SegGs, SegFs, SegEs, SegDs;
    DWORD  Edi, Esi, Ebx, Edx, Ecx, Eax, Ebp, Eip;
    DWORD  SegCs, EFlags, Esp, SegSs;
} CONTEXT;
```

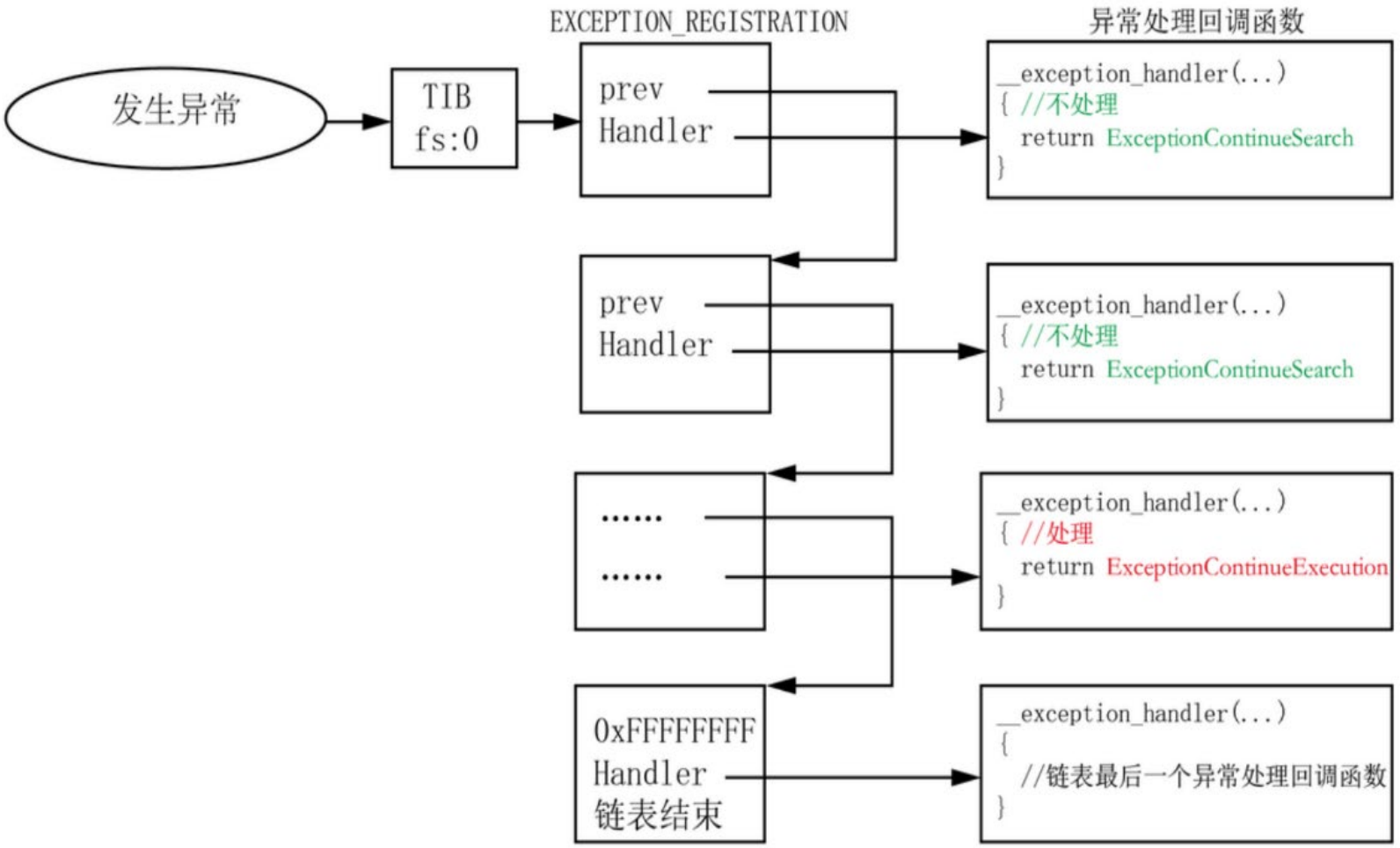
12.2.1 编写异常处理函数

方法3：插入SEH链表的表头结点 (7)

```
typedef enum  _EXCEPTION_DISPOSITION {
    ExceptionContinueExecution,
    // 已经处理了异常，回到异常触发点继续执行
    ExceptionContinueSearch,
    // 继续遍历异常链表，寻找其他的异常处理方法
    ExceptionNestedException,
    // 在异常处理过程中再次触发异常
    ExceptionCollidedUnwind
    // 冲突 松开；发生了嵌套的展开操作
} EXCEPTION_DISPOSITION;  //下一步准备采取的动作
```

12.2.2 异常处理程序的注册

方法3：插入SEH链表的表头结点 (8)



12.2.1 编写异常处理函数

方法3：插入SEH链表的表头结点 (9)

```
#include <stdio.h>   //Run in x86 debug
#include <windows.h>
EXCEPTION_DISPOSITION __cdecl
SEH_handler1(
  struct  _EXCEPTION_RECORD
*ExceptionRecord,
  void  *EstablisherFrame,
  struct _CONTEXT  *ContextRecord,
  void  *Dispatchercontext )
{
    printf("SEH1: exception code = %X\n",
    ExceptionRecord->ExceptionCode);
    return ExceptionContinueSearch;  //(1)
}
```

```
EXCEPTION_DISPOSITION __cdecl
SEH_handler2(
  struct _EXCEPTION_RECORD
*ExceptionRecord,
  void  *EstablisherFrame,
  struct  _CONTEXT  *ContextRecord,
  void  *Dispatchercontext )
{
    printf("SEH2: exception code = %X\n",
    ExceptionRecord->ExceptionCode);
    return ExceptionContinueSearch;  //(2)
}

LONG WINAPI  ExceptionFilterHandler
(EXCEPTION_POINTERS  *ExceptionInfo)
{
    printf("ExceptionFilterHandler\n");
    return EXCEPTION_EXECUTE_HANDLER;
}           //(3)
```

12.2.1 编写异常处理函数

方法3：插入SEH链表的表头结点 (10)

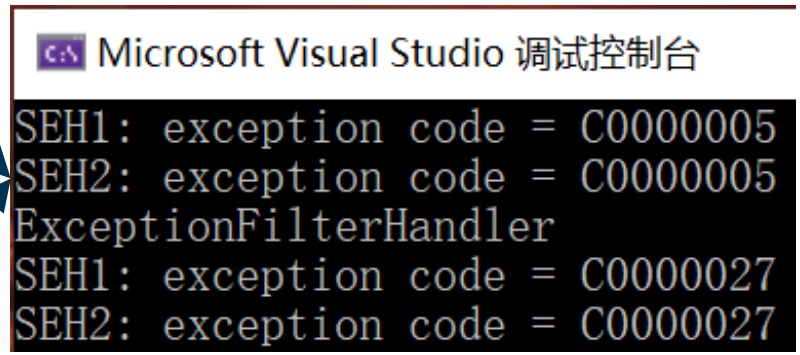
```
int main( )
{
  SetUnhandledExceptionFilter(ExceptionFilterHandler);  //安装筛选器回调函数
  DWORD handler1 = (DWORD)SEH_handler1;
  DWORD handler2 = (DWORD)SEH_handler2;
  _asm  {  //插入2个新的SEH结点到SEH链的表头
       push  handler2       //SEH_handler2函数的地址
       push  FS:[0]         //获取前一结点的地址
       mov   FS:[0], ESP    //安装新的SEH链表
       /**/
       push  handler1       //SHE_handler1函数的地址
       push  FS:[0]         //获取前一结点的地址
       mov   FS:[0], ESP    //安装新的SEH链表
   }
   *(int *)0 = 0;  //产生异常
   printf("Never get here!\n" ) ;
}
```

12.2.1 编写异常处理函数

方法3：插入SEH链表的表头结点 (11)

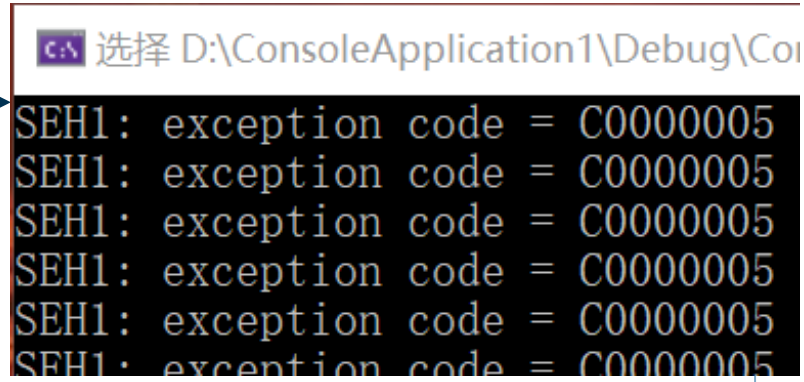
(1) ExceptionContinueSearch
(2) ExceptionContinueSearch
(3) EXCEPTION_EXECUTE_HANDLER

(1) ExceptionContinueSearch
(2) ExceptionContinueSearch
(3) EXCEPTION_CONTINUE_SEARCH



(1) ExceptionContinueExecution
(2) ExceptionContinue … (任意值)
(3) EXCEPTION_ … (任意值)

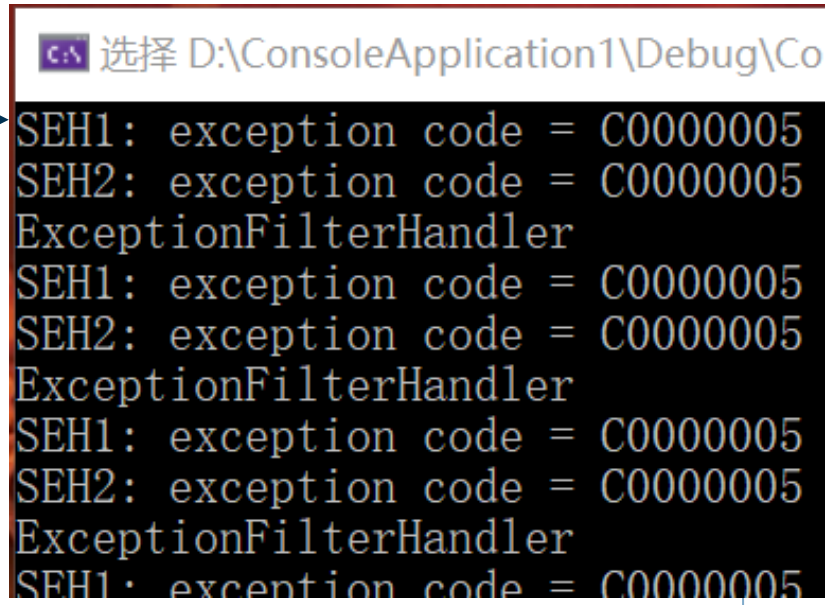
SEH1 无限循环 … …
不调用 SEH2
不调用 ExceptionFilterHandler !



12.2.1 编写异常处理函数

方法3：插入SEH链表的表头结点 (12)

(1) ExceptionContinueSearch
(2) ExceptionContinueSearch
(3) EXCEPTION_CONTINUE_EXECUTION
SEH1 => SEH2 => 筛选器回调函数
=> SEH1 … … 无限循环



总结

- 中断和异常的概念

  中断源、中断描述符表

- 中断和异常响应的过程
- 中断和异常处理程序的调用与返回

  INT　IRET

- 中断服务程序调用与一般的子程序调用的异同点

补充知识

异常示例

EXCEPTION_INT_DIVIDE_BY_ZERO
　　整数除法的除数是 0
EXCEPTION_ACCESS_VIOLATION
　　企图从一个不具有权限的虚拟地址读取或者写入
EXCEPTION_STACK_OVERFLOW
　　栈溢出
EXCEPTION_ILLEGAL_INSTRUCTION
　　企图执行一个无效的指令
EXCEPTION_FLT_OVERFLOW
　　浮点数的指数超过所能表示的最大值
EXCEPTION_ARRAY_BOUNDS_EXCEEDED
　　企图越界访问数组元素，并且底层硬件支持边界检查
EXCEPTION_BREAKPOINT
　　断点被触发

补充知识

异常编号示例 ： winnt.h

```
#define STATUS_INTEGER_DIVIDE_BY_ZERO ((DWORD)0xC0000094L)
#define STATUS_ACCESS_VIOLATION ((DWORD)0xC0000005L)
#define STATUS_STACK_OVERFLOW ((DWORD)0xC00000FDL)
#define STATUS_ILLEGAL_INSTRUCTION  ((DWORD)0xC000001DL)
```

补充知识

异常编号示例 ： minwinbase.h

```
#define EXCEPTION_INT_DIVIDE_BY_ZERO
        STATUS_INTEGER_DIVIDE_BY_ZERO

#define EXCEPTION_ACCESS_VIOLATION
        STATUS_ACCESS_VIOLATION

#define EXCEPTION_STACK_OVERFLOW
        STATUS_STACK_OVERFLOW

#define EXCEPTION_ILLEGAL_INSTRUCTION
        STATUS_ILLEGAL_INSTRUCTION
```