异常控制流

- 分以下两个部分介绍
  - 第一讲：进程与进程的上下文切换
    - CPU的控制流、异常控制流
    - 程序和进程、引入进程的好处
    - 逻辑控制流和物理控制流
    - 进程与进程的上下文切换
    - 程序的加载和运行
  - 第二讲：异常和中断
    - 异常和中断的基本概念
    - 异常和中断的响应、处理
    - IA-32/Linux下的异常/中断机制

异常和中断

- 程序执行过程中CPU会遇到一些特殊情况，使正在执行的程序被“中断”
  - CPU中止原来正在执行的程序，转到处理异常情况或特殊事件的程序去执行，结束后再返回到原被中止的程序处（断点）继续执行。
- 程序执行被 “中断” 的事件（在硬件层面）有两类
  - 内部“异常”：在CPU内部发生的意外事件或特殊事件

    按发生原因分为硬故障中断和程序性中断两类

    硬故障中断：如电源掉电、硬件线路故障等

    程序性中断：执行某条指令时发生的“例外(Exception)”事件，如溢出、缺页、越界、越权、越级、非法指令、除数为0、堆/栈溢出、访问超时、断点设置、单步、系统调用等
  - 外部“中断”：在CPU外部发生的特殊事件，通过“中断请求”信号向CPU请求处理。如实时钟、控制台、打印机缺纸、外设准备好、采样计时到、DMA传输结束等。

回顾:逻辑地址向线性地址转换举例

- 已知变量y和数组a都是int型，a的首地址为0x8048a00。假设编译器将a的首地址分配在ECX中，数组的下标变量i分配在EDX中，y分配在EAX中，C语言赋值语句“y=a[i];”被编译为指令“movl (%ecx, %edx, 4), %eax”。若在IA-32/Linux环境下执行指令地址为0x80483c8的该指令时，CS段寄存器对应的描述符cache中存放的是表6.2中所示的用户代码段信息且CPL=3，DS段寄存器对应的描述符cache中存放的是表6.2中所示的用户数据段信息，则当i=100时，取指令操作过程中MMU得到的指令的线性地址是多少？取数操作过程中MMU得到的操作数的线性地址是多少?

```
int func(int a[ ], int c)
{
  int i,  y = 0;
  for(i = 0; i<c; i++) {
        y= a[i];
        …
  }
  ……
}
```

段	基地址	G	限界	S	TYPE	DPL	D	P
用户代码段	0x0000 0000	1	0xFFFFF	1	10	3	1	1
用户数据段	0x0000 0000	1	0xFFFFF	1	2	3	1	1

y= a[i]; ⟶ 80483c8: movl (%ecx, %edx, 4), %eax

…代码和数据段DPL都为3，即CPL最低应为3，而CPL=3，故访问未越级

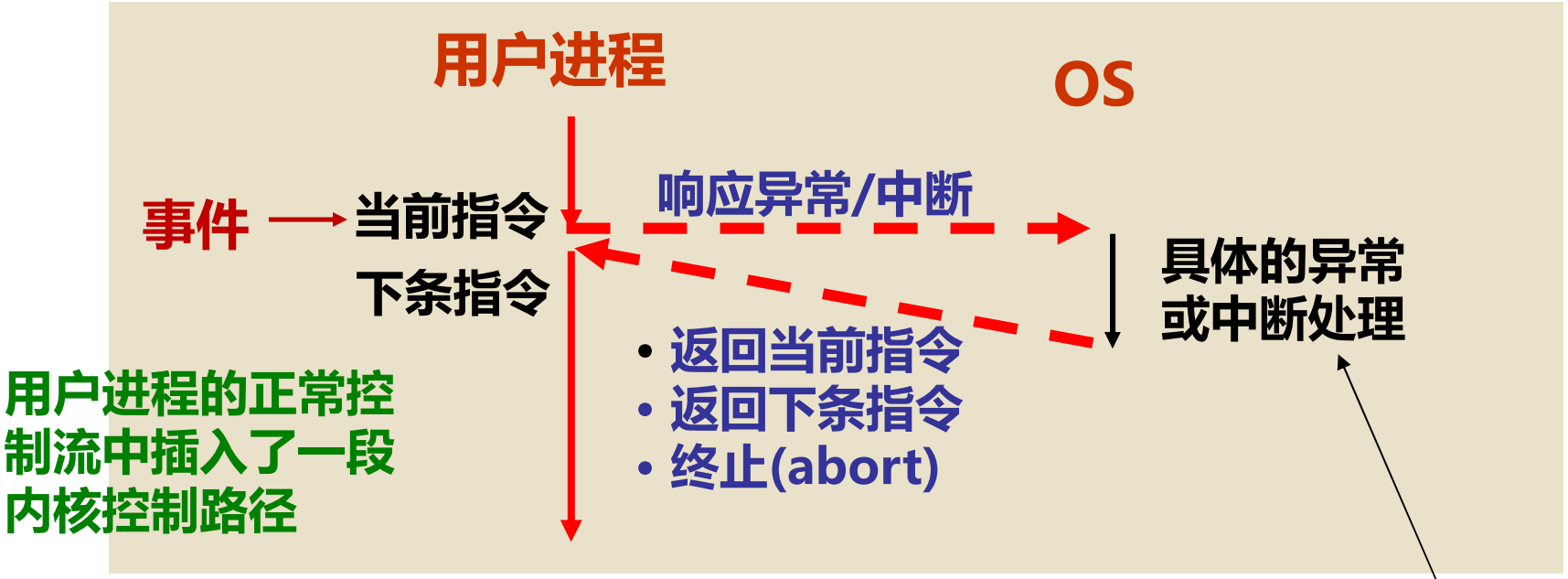
指令的线性地址：代码段基地址+EA=0+0x80483c8=0x80483c8

操作数的线性地址：数据段基地址+EA=0+R[ecx]+R[edx]×4

=0x8048a00+100×4=0x8048b90

异常和中断的处理

- 发生异常(exception)和中断(interrupt)事件后，系统将进入OS内核态对相应事件进行处理，即改变处理器状态（用户态→内核态）



中断或异常处理执行的代码不是一个进程，而是“内核控制路径”，它代表异常或中断发生时正在运行的当前进程在内核态执行一个独立的指令序列。内核控制路径比进程更“轻”，其上下文信息比进程上下文信息少得多。而上下文切换后CPU执行的是另一个用户进程。

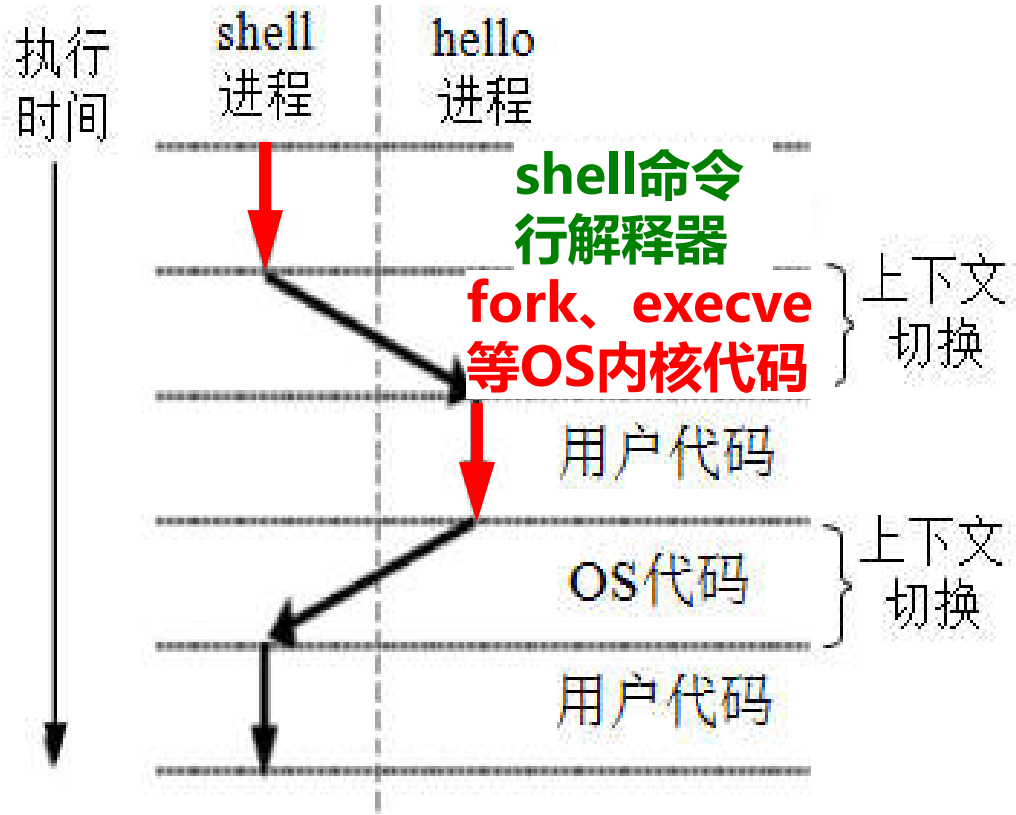
回顾：“进程” 与“上下文切换”

OS通过处理器调度让处理器轮流执行多个进程。实现不同进程中指令交替执行的机制称为进程的上下文切换（context switching）

```
$./hello
hello, world
$
```

“$”是shell命令行提示符，说明正在运行shell进程。

上下文切换后，是另一个用户进程！



处理器调度等事件会引起用户进程正常执行被打断，因而形成异常控制流。
进程的上下文切换机制很好地解决了这类异常控制流，实现了从一个进程安全切换到另一个进程执行的过程。

异常的分类

“异常” 按处理方式分为故障、自陷和终止三类

故障(fault)：执行指令引起的异常事件，如溢出、非法指令、缺页、访问越权等。 “断点”为发生故障指令的地址

自陷(Trap)：预先安排的事件（“埋地雷”），如单步跟踪、断点、系统调用 (执行访管指令) 等。是一种自愿中断。

“断点”为自陷指令下条指令地址

终止(Abort)：硬故障事件，此时机器将“终止”，调出中断服务程序来重启操作系统。 “断点”是什么？ 随便！

思考1：自陷处理完成后回到哪条指令执行？ 回到下条指令

思考2：哪些故障补救后可继续执行，哪些只好终止当前进程？

缺页、TLB缺失等：补救后可继续，回到发生故障的指令重新执行。

溢出、除数为0、非法指令、内存保护错等：终止当前进程。

“断点”：异常处理结束后回到原来被“中断”的程序执行时的起始指令。

异常举例—页故障

“页故障”事件何时发现？如何发现？

执行每条指令都要访存（取指令、取操作数、存结果）

在保护模式下，每次访存都要进行逻辑地址向物理地址转换

在地址转换过程中会发现是否发生了“页故障”！

“页故障”事件是软件发现的还是硬件发现的？

逻辑地址向物理地址的转换由硬件（MMU）实现，故“页故障”事件由硬件发现。所有异常和中断事件都由硬件检测发现！

- 以下几种情况都会发生“页故障”
  - 缺页：页表项有效位为0 ← 可通过读磁盘恢复故障
  - 地址越界：地址大于最大界限
  - 访问越级或越权（保护违例）：

  } 不可恢复，称为“段故障（segmentation fault）”

    - 越级：用户进程访问内核数据（CPL=3 / DPL=0）
    - 越权：读写权限不相符（如对只读段进行了写操作）

异常举例—页故障

假设在IA-32/linux系统中一个C语言源程序 P 如下：

```
1   int a[1000];
2   int x;
3   main( )
4   {
5      a[10]=1;
6      a[1000]=3;
7      a[10000]=4;
8   }
```

正常的控制流为

…、0x8048300、 0x8048309、 0x8048313、…

可能的异常控制流是什么?

假设编译、汇编和链接后，第5、6和7行源代码对应的指令序列如下：

```
5   8048300: c7 05 28 90 04 08 01 00 00 00   movl   $0x1, 0x8049028
6   8048309: c7 05 a0 9f 04 08 03 00 00 00   movl   $0x3, 0x8049fa0
7   8048313: c7 05 40 2c 05 08 04 00 00 00   movl   $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则：

(1) 对于上述三条指令的执行，在取指令时是否可能发生页故障?

(2) 在数据访问时分别会发生什么问题?

(3) 哪些问题是可恢复的？哪些问题是不可恢复的?

异常举例—页故障

假设在IA-32/linux系统中一个C语言源程序 P 如下：

```
1   int a[1000];
2   int x;
3   main( )
4   {
5       a[10]=1;
6       a[1000]=3;
7       a[10000]=4;
8   }
```

三条指令在读指令时都不会发生缺页，Why?

它们都位于起始地址为0x08048000（是一个4KB页面的起始位置）的同一个页面，执行这三条指令之前，该页已经调入内存。因为没有其他进程在系统中运行，所以不会因为执行其他进程而使得调入主存的页面被调出到磁盘。因而都不会在取指令时发生页故障。

假设编译、汇编和链接后，第5、6和7行源代码对应的指令序列如下：

```
5   8048300: c7 05 28 90 04 08 01 00 00 00   movl   $0x1, 0x8049028
6   8048309: c7 05 a0 9f 04 08 03 00 00 00   movl   $0x3, 0x8049fa0
7   8048313: c7 05 40 2c 05 08 04 00 00 00   movl   $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则：

(1) 对于上述三条指令的执行，在取指令时是否可能发生页故障?

(2) 在数据访问时分别会发生什么问题?

(3) 哪些问题是可恢复的？哪些问题是不可恢复的?

异常举例—页故障

假设在IA-32/linux系统中一个C语言源程序 P 如下：

```
1   int a[1000];
2   int x;
3   main( )
4   {
5      a[10]=1;
6      a[1000]=3;
7      a[10000]=4;
8   }
```

第5行指令数据访问时是否发生页故障，Why?

对a[10]（地址0x8049028）的访问是对所在页面（首址为0x08049000）的第一次访问，故不在主存，缺页处理结束后，再回到这条movl指令重新执行，再访问数据就没有问题了。

假设编译、汇编和链接后，第5、6和7行源代码对应的指令序列如下：

```
5   8048300: c7 05 28 90 04 08 01 00 00 00   movl   $0x1, 0x8049028
6   8048309: c7 05 a0 9f 04 08 03 00 00 00   movl   $0x3, 0x8049fa0
7   8048313: c7 05 40 2c 05 08 04 00 00 00   movl   $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则：

(1) 对于上述三条指令的执行，在取指令时是否可能发生页故障?

(2) 在数据访问时分别会发生什么问题?

(3) 哪些问题是可恢复的？哪些问题是不可恢复的?

异常举例—页故障

假设在IA-32/linux系统中一个C语言源程序 P 如下：

```
1   int a[1000];
2   int x;
3   main( )
4   {
5      a[10]=1;
6      a[1000]=3;
7      a[10000]=4;
8   }
```

第6行指令数据访问时是否发生页故障，Why?

对a[1000]（地址0x8049fa0）的访问是对所在页面（首址为0x08049000）的第2次访问，故在主存，不会发生缺页。但a[1000]实际不存在，只不过编译器未检查数组边界，0x8049fa0处可能是x的地址，故该指令执行结果可能是x被赋值为3

假设编译、汇编和链接后，第5、6和7行源代码对应的指令序列如下：

```
5   8048300: c7 05 28 90 04 08 01 00 00 00   movl   $0x1, 0x8049028
6   8048309: c7 05 a0 9f 04 08 03 00 00 00   movl   $0x3, 0x8049fa0
7   8048313: c7 05 40 2c 05 08 04 00 00 00   movl   $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则：

(1) 对于上述三条指令的执行，在取指令时是否可能发生页故障?

(2) 在数据访问时分别会发生什么问题?

(3) 哪些问题是可恢复的？哪些问题是不可恢复的?

异常举例—页故障

假设在IA-32/linux系统中一个C语言源程序 P 如下：

```
1   int a[1000];
2   int x;
3   main( )
4   {
5      a[10]=1;
6      a[1000]=3;
7      a[10000]=4;
8   }
```

第7行指令数据访问时是否发生页故障，Why?

地址0x8052c40偏离数组首址0x8049000已达4×10000+4=40004个单元，即偏离了9个页面，很可能超出可读写区范围，故执行该指令时可能会发生保护违例。页故障处理程序发送一个“段错误”信号（SIGSEGV）给用户进程，用户进程接受到该信号后就调出一个信号处理程序执行，该信号处理程序根据信号类型，在屏幕上显示“段故障(segmentation fault)”信息，并终止用户进程。

假设编译、汇编和链

```
5   8048300: c7 05 28 90 04 08 01 00 00 00   movl   $0x1, 0x8049028
6   8048309: c7 05 a0 9f 04 08 03 00 00 00   movl   $0x3, 0x8049fa0
7   8048313: c7 05 40 2c 05 08 04 00 00 00   movl   $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则：

(1) 对于上述三条指令的执行，在取指令时是否可能发生页故障?

(2) 在数据访问时分别会发生什么问题?

(3) 哪些问题是可恢复的？哪些问题是不可恢复的?

异常的分类

“异常” 按处理方式分为故障、自陷和终止三类

故障(fault)：执行指令引起的异常事件，如溢出、缺页、堆栈溢出、访问超时等。 “断点”为发生故障指令的地址

自陷(Trap)：预先安排的事件（“埋地雷”），如单步跟踪、断点、系统调用 (执行访管指令) 等。是一种自愿中断。

“断点”为自陷指令下条指令地址

终止(Abort)：硬故障事件，此时机器将“终止”，调出中断服务程序来重启操作系统。 “断点”是什么？ 随便！

思考1：自陷处理完成后回到哪条指令执行？ 回到下条指令

思考2：哪些故障补救后可继续执行，哪些只好终止当前进程？

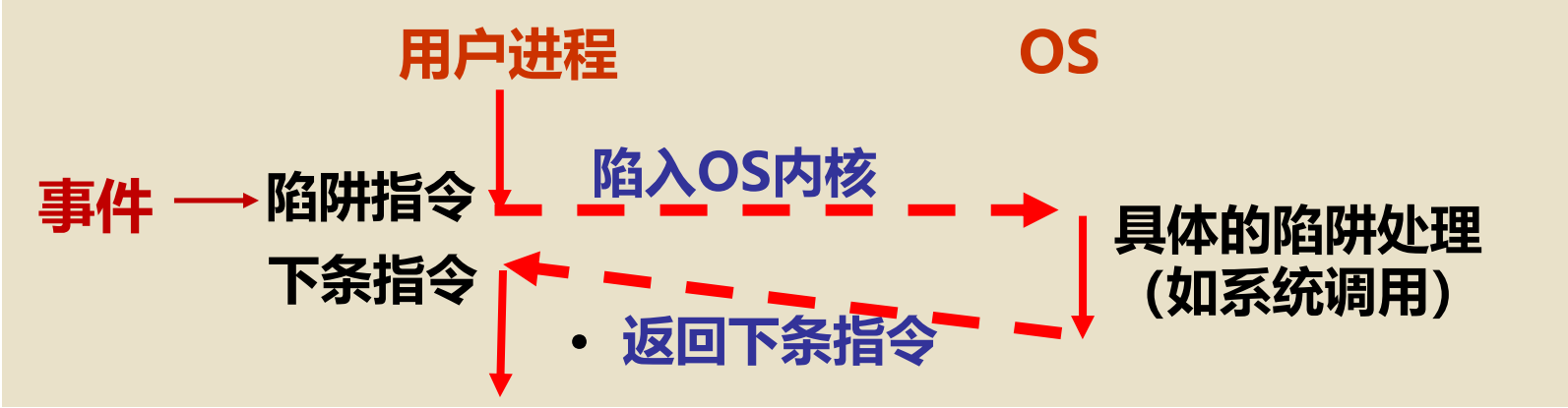
缺页、TLB缺失等：补救后可继续，回到发生故障的指令重新执行。

溢出、除数为0、非法操作、内存保护错等：终止当前进程。

- 不同体系结构和教科书对“异常”和“中断”定义的内涵不同，在看书时要注意！

陷阱（Trap）异常

- 陷阱也称自陷或陷入，执行陷阱指令（自陷指令）时，CPU调出特定程序进行相应处理，处理结束后返回到陷阱指令下一条指令执行。



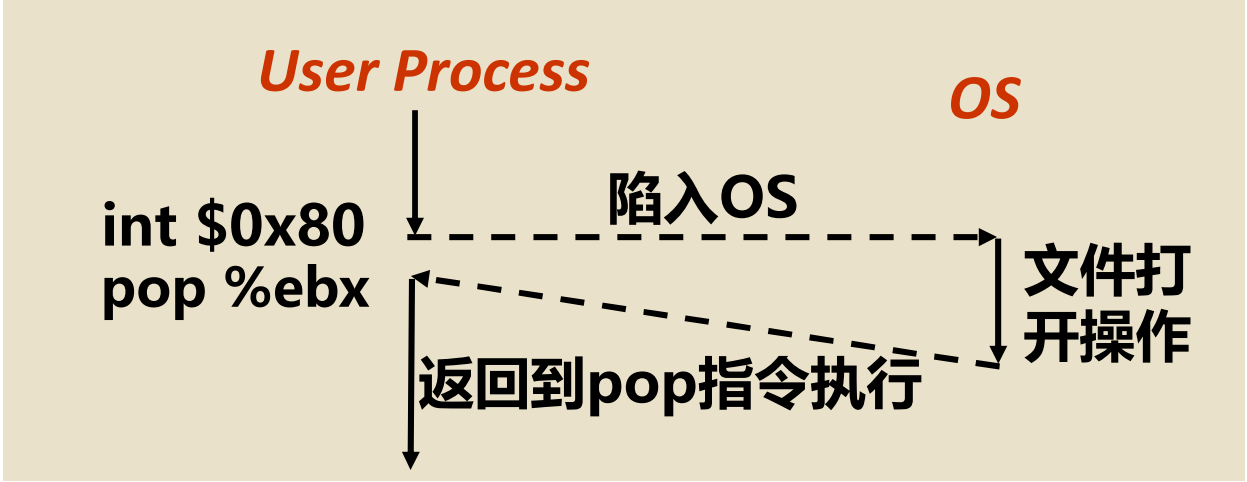
- 陷阱的作用之一是在用户和内核之间提供一个像过程一样的接口，这个接口称为系统调用，用户程序利用这个接口可方便地使用操作系统内核提供的一些服务。操作系统给每个服务编一个号，称为系统调用号。例如，Linux系统调用fork、read和execve的调用号分别是1、3和11。
- IA-32处理器中的 int 指令和 sysenter 指令、MIPS处理器中的 syscall 指令等都属于陷阱指令（相当于“地雷”）。 有条件“爆炸”
- 陷阱指令异常称为编程异常（programmed exception），这些指令包括 INT n、int 3、into（溢出检查）、bound（地址越界检查）等

Trap举例: Opening File

- 用户程序中调用函数 open(filename, options)
- open函数执行陷阱指令（即系统调用指令“int”）

这种“地雷”一定“爆炸”

```
0804d070 <__libc_open>:
 . . .
 804d082:      cd 80          int    $0x80
 804d084:      5b             pop    %ebx
 . . .
```



通过执行“int $0x80”指令，调出OS完成一个具体的“服务”（称为系统调用）

Open系统调用（system call）：OS must find or create file, get it ready for reading or writing，Returns integer file descriptor

陷阱（Trap）异常

问题：你用过单步跟踪、断点设置等调试功能吗？你知道这些功能是如何实现的吗？ 通过“埋地雷”的方式实现

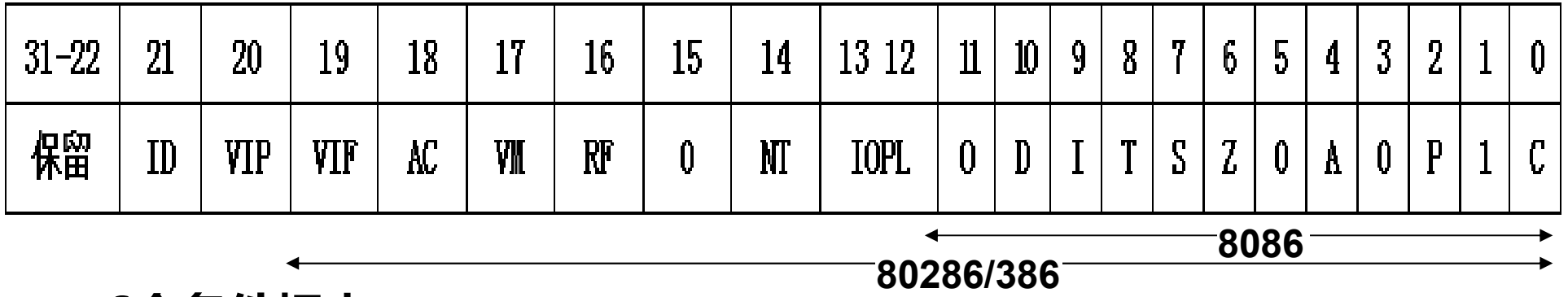
- 利用陷阱机制可实现程序调试功能，包括设置断点和单步跟踪
  - IA-32中，当CPU处于单步跟踪状态（TF=1且IF=1）时，每条指令都被设置成了陷阱指令，执行每条指令后，都会发生中断类型号为1的“调试”异常，从而转去执行“单步跟踪处理程序”。

    注意： 当陷阱指令是转移指令时，不能返回到转移指令的下条指令执行，而是返回到转移目标指令执行。

    （在一定的条件下，每条指令都变成“地雷”）
  - IA-32中，用于程序调试的“断点设置”陷阱指令为int 3，对应机器码为CCH。若调试程序在被调试程序某处设置了断点，则调试程序就在该处“加”一条int 3指令（该首字节）。执行到该指令时，会暂停被调试程序的运行，并发出“EXCEPTION_BREAKPOINT”异常，以调出调试程序执行，执行结束后回到被调试程序执行。

    （int 3是一定爆炸的“地雷”）

IA-32的标志寄存器



- 6个条件标志
  - OF、SF、ZF、CF各是什么标志（条件码）？
  - AF：辅助进位标志（BCD码运算时才有意义）
  - PF：奇偶标志
- 3个控制标志
  - DF（Direction Flag）：方向标志（自动变址方向是增还是减）
  - IF（Interrupt Flag）：中断允许标志 （仅对外部可屏蔽中断有用）
  - TF（Trap Flag）：陷阱标志（是否是单步跟踪状态）
- ……

异常的分类

“异常” 按处理方式分为故障、自陷和终止三类

故障(fault)：执行指令引起的异常事件，如溢出、缺页、堆栈溢出、访问超时等。 “断点”为发生故障指令的地址

自陷(Trap)：预先安排的事件（“埋地雷”），如单步跟踪、断点、系统调用 (执行访管指令) 等。是一种自愿中断。

“断点”为自陷指令下条指令地址

终止(Abort)：硬故障事件，此时机器将“终止”，调出中断服务程序来重启操作系统。 “断点”是什么？ 随便！

思考1：自陷处理完成后回到哪条指令执行？ 回到下条指令

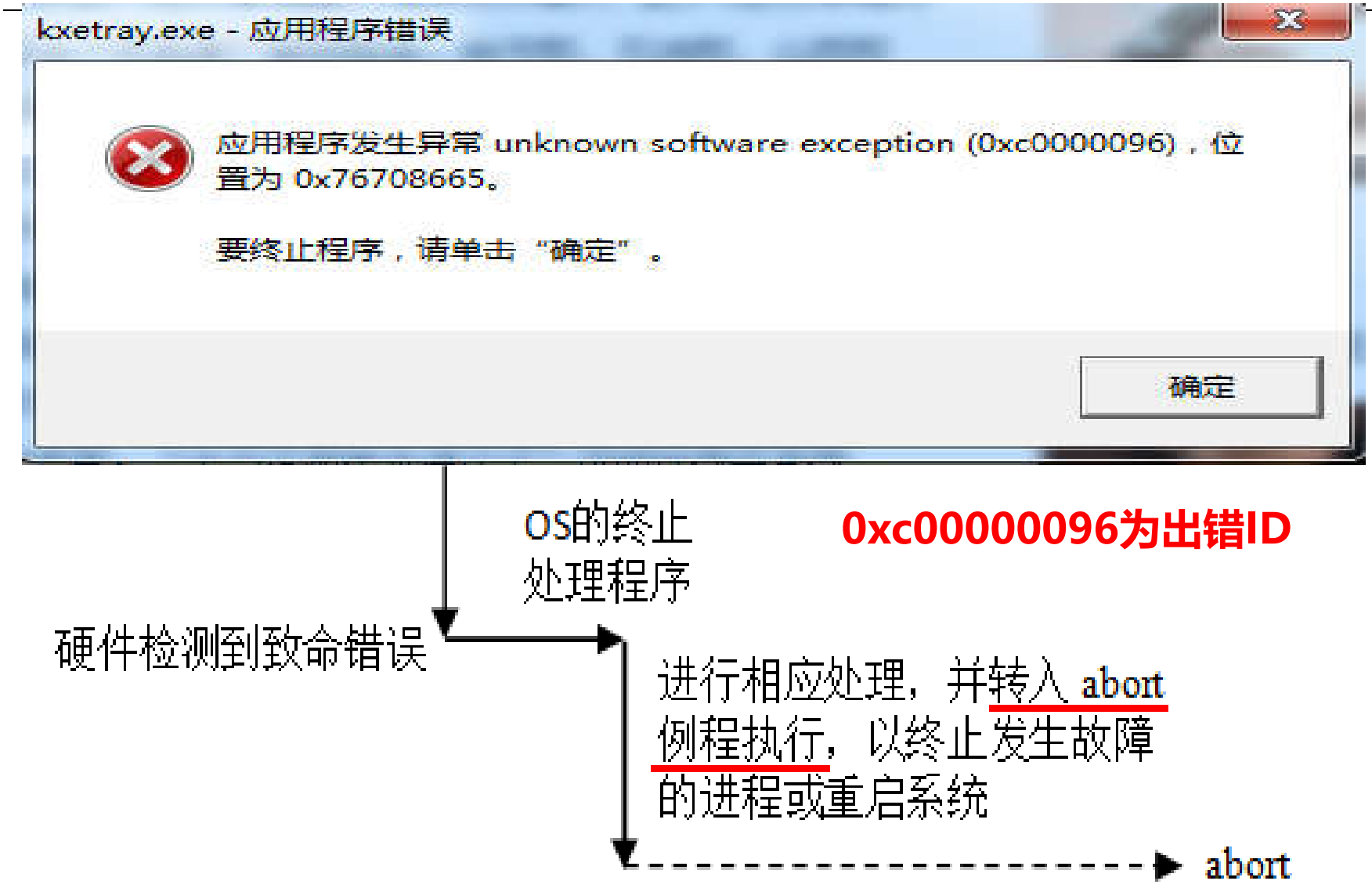
思考2：哪些故障补救后可继续执行，哪些只好终止当前进程？

缺页、TLB缺失等：补救后可继续，回到发生故障的指令重新执行。

溢出、除数为0、非法操作、内存保护错等：终止当前进程。

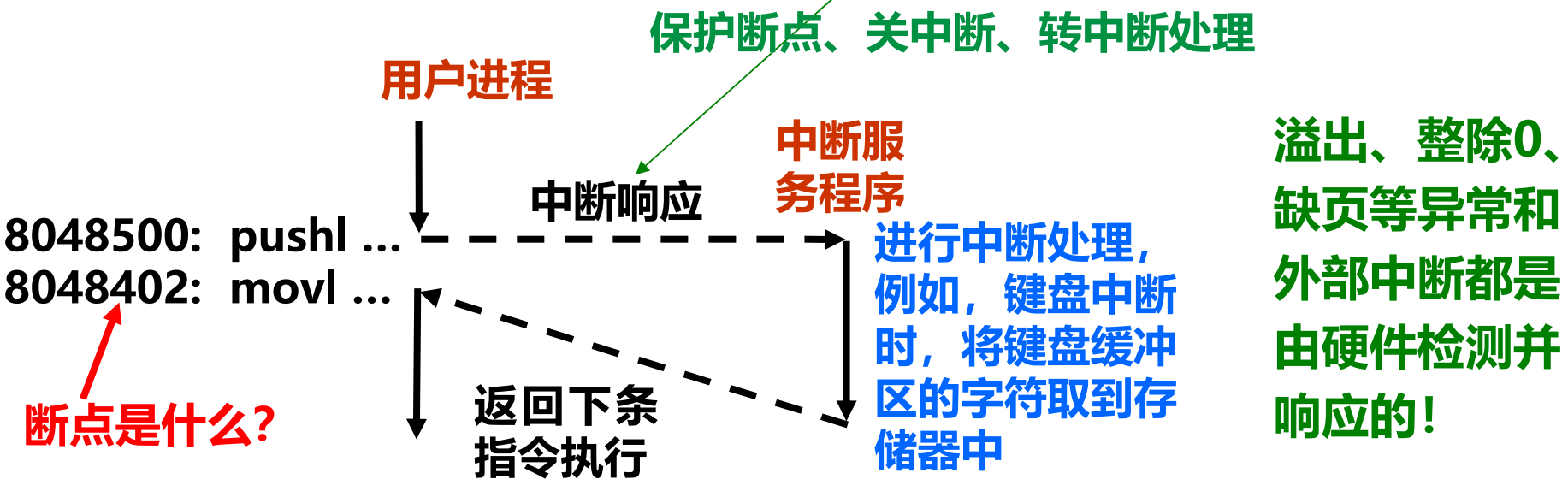
- 不同体系结构和教科书对“异常”和“中断”定义的内涵不同，在看书时要注意！

终止（Abort）异常


kxetray.exe - 应用程序错误
应用程序发生异常 unknown software exception (0xc0000096)，位置为 0x76708665。
要终止程序，请单击“确定”。
确定
OS的终止处理程序
0xc00000096为出错ID
硬件检测到致命错误
进行相应处理，并转入 abort 例程执行，以终止发生故障的进程或重启系统
abort

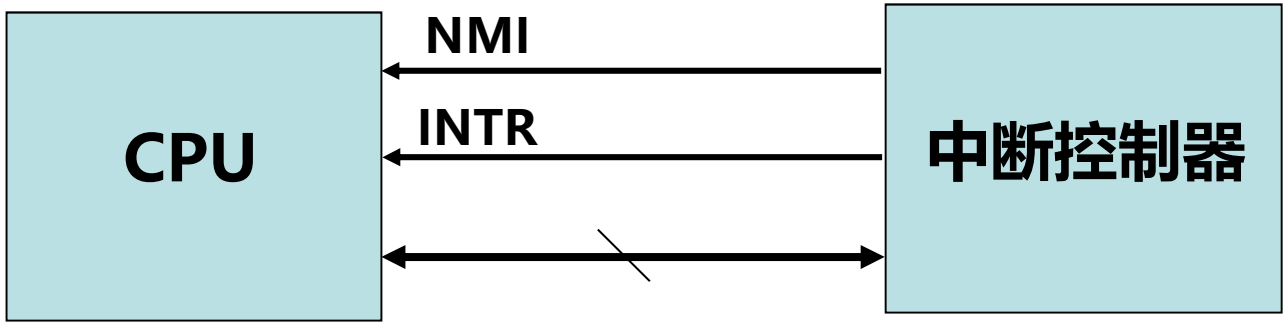
中断的概念

- 外设通过**中断请求信号线**向CPU提出“中断”请求，不由指令引起，故中断也称为**异步异常**。
- 事件：Ctrl-C、DMA传送结束、网络数据到达、打印缺纸、……
- 每执行完一条指令，CPU就查看中断请求引脚，若**引脚的信号有效**，则进行**中断响应**：将当前PC（断点）和当前机器状态保存到栈中，并“关中断”，然后，从数据总线读取中断类型号，根据中断类型号跳转到对应的中断服务程序执行。**中断检测及响应过程由硬件完成**。
- 中断服务程序执行具体的中断处理工作，中断处理完成后，再回到被打断程序的“断点”处继续执行。



中断的分类

- Intel将中断分成**可屏蔽中断**（maskable interrupt）和**不可屏蔽中断**（nonmaskable interrupt，NMI）。
  - **可屏蔽中断**：通过 INTR 向CPU请求，可通过设置**屏蔽字**来屏蔽请求，若中断请求被屏蔽，则不会被送到CPU。
  - **不可屏蔽中断**：非常紧急的硬件故障，如：电源掉电，硬件线路故障等。通过 NMI 向CPU请求。一旦产生，就被立即送CPU，以便快速处理。这种情况下，中断服务程序会尽快保存系统重要信息，然后在屏幕上显示相应的消息或直接重启系统。



异常/中断响应过程

检测到异常或中断时，CPU须进行以下基本处理：

① 关中断（“中断允许位” 清0）：使CPU处于“禁止中断”状态，以防止新中断破坏断点（PC）、程序状态（PSW）和现场（通用寄存器）。

② 保护断点和程序状态：将断点和程序状态保存到栈或特殊寄存器中

PC→栈 或 EPC（专门存放断点的寄存器）

PSWR →栈 或 EPSWR （专门保存程序状态的寄存器）

PSW（Program Status Word）：程序状态字

PSWR（PSW寄存器）：如IA-32中的的EFLAGS寄存器

③ 识别中断事件

有软件识别和硬件识别（向量中断）两种不同的方式。

IA-32中，响应异常时不关中断，只在响应中断时关中断

异常/中断响应过程

有两种不同的识别方式：软件识别和硬件识别（向量中断）。

（1）软件识别（MIPS采用）

设置一个异常状态寄存器（MIPS中为Cause寄存器），用于记录异常原因。操作系统使用一个统一的异常处理程序，该程序按优先级顺序查询异常状态寄存器，识别出异常事件。

（例如：MIPS中位于内核地址0x8000 0180处有一个专门的异常处理程序，用于检测异常的具体原因，然后转到内核中相应的异常处理程序段中进行具体的处理）

（2）硬件识别（向量中断）（IA-32采用）

用专门的硬件查询电路按优先级顺序识别异常，得到“中断类型号”，根据此号，到中断向量表中读取对应的中断服务程序的入口地址。

所有事件都被分配一个“中断类型号”，每个中断都有相应的“中断服务程序”，可根据中断类型号找到中断服务程序的入口地址。

中断类型号相当于中断向量表的索引，表中存放中断服务程序首地址