第六章 异常控制流

异常控制流

- 第一讲：异常
  - 异常基本概念和分类、异常的响应和处理
  - x86-64+Linux下的异常机制
- 第二讲：进程与进程的上下文切换
  - 程序和进程的概念
  - 进程的逻辑控制流
  - 进程与进程的上下文切换

控制流

- CPU的控制流： CPU所执行的指令的地址序列
  - 正常控制流：以下两种方式得到的控制流
    1. 按顺序取下一条指令执行
    2. 通过CALL/RET/Jcc/JMP等指令跳转到转移目标地址处执行

```
int sum(int a[ ], unsigned len)
{
  int i, sum = 0;
  for (i = 0; i <= len-1; i++)
    sum += a[i];
  return sum;
}
```

```
sum:
  …
.L3:
  …
  movl  -4(%ebp),  %eax
  movl  12(%ebp),  %edx
  subl  $1,  %edx
  cmpl  %edx,  %eax
  jbe   .L3
  …
```

  - 异常控制流（Exceptional Control of Flow，ECF）：由于某些特殊情况引起用户程序的正常执行被打断所形成的意外控制流

异常控制流

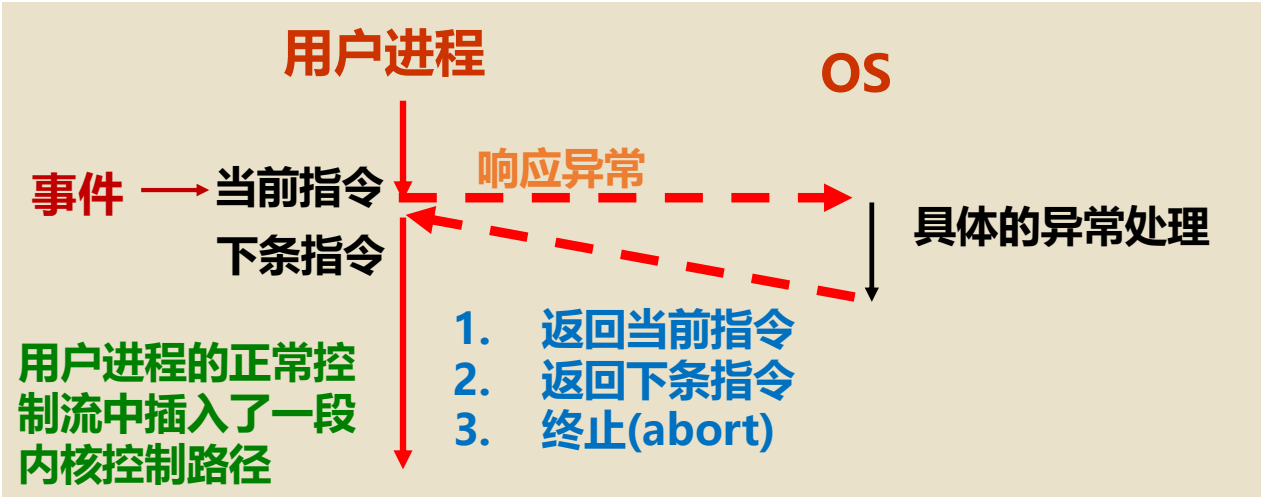
- CPU会因为遇到异常事件而打断程序的正常控制流，转去执行操作系统提供的针对这些特殊事件的处理程序。
- 异常控制流的形成原因：
  - 内部（缺页、越权、越级、整除0、溢出等）
  - 外部（Ctrl-C、打印缺纸、DMA结束等）

  } 发生在硬件层

  - 进程的上下文切换 — 发生在操作系统层
  - 一个进程直接发送信号给另一个进程 — 发生在应用软件层

异常的处理

- 发生异常事件后，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）
- 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。



异常的分类

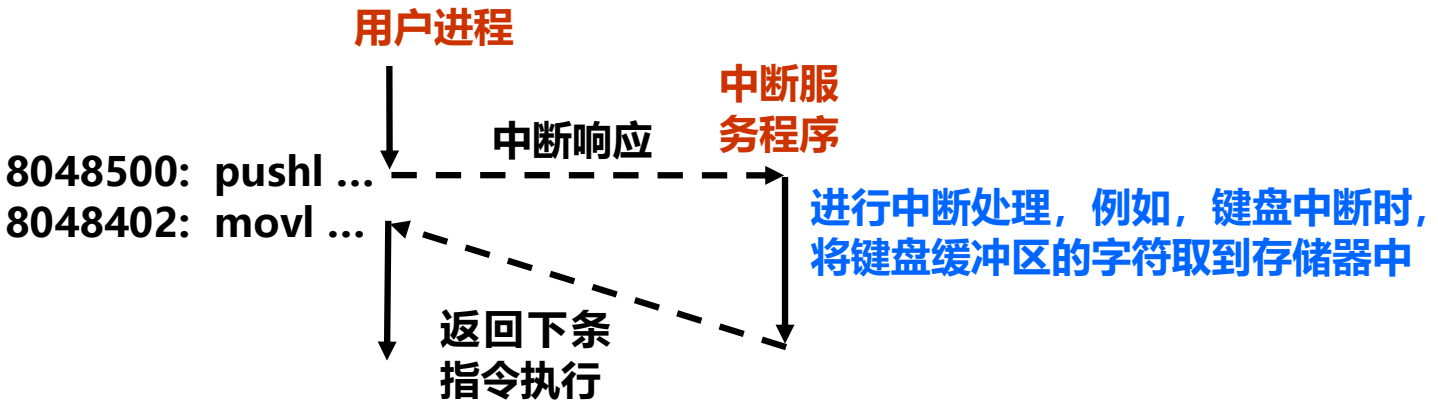
- 异常分为中断、陷阱、故障和终止四类

类别	原因	异步 / 同步	返回行为
中断	来自 I/O 设备的信号	异步	总是返回到下一条指令
陷阱	有意的异常	同步	总是返回到下一条指令
故障	潜在可恢复的错误	同步	可能返回到当前指令
终止	不可恢复的错误	同步	不会返回

- 同步异常：执行指令的直接产物
- 异步异常：处理器外部的I/O设备中的事件产生

中断

- 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”处继续执行下一条指令。

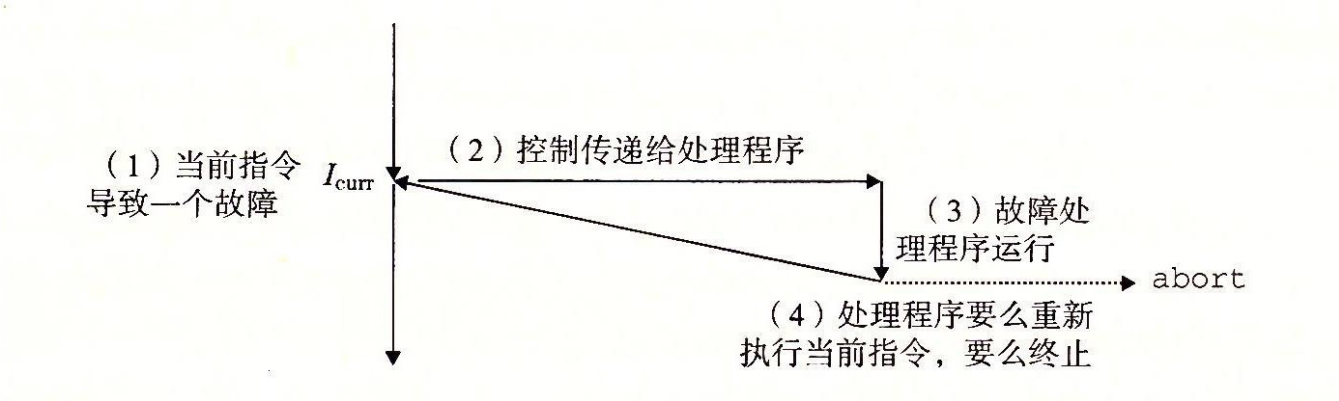


中断

- Intel将中断分成可屏蔽中断（maskable interrupt）和不可屏蔽中断（non-maskable interrupt，NMI）。
  - 可屏蔽中断：通过中断请求（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。

故障

- 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。



- 补救后可继续执行的故障
  - 缺页
- 只好终止当前进程的故障
  - 溢出、除0、非法指令、内存保护错误等

故障

- 页故障
  - 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”
  - 软件发现还是硬件发现？
    - 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。
  - 几种情形
    - 缺页：页表项有效(存在)位为0 —— 可通过读磁盘恢复
    - 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
      - 越权：读写权限不相符（如对只读段进行了写操作）

页故障

- x86-64/Linux中一个C语言源程序P如下左；编译、汇编和链接后，第5、6、7行源码对应的指令序列如下右。

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

```
5  400300: movl  $0x1, 0x401028
6  40030a: movl  $0x3, 0x401fa0
7  400314: movl  $0x4, 0x409c40
```

正常控制流：
…, 400300, 40030a, 400314, …

可能的异常控制流是什么？

- 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？

页故障

- x86-64/Linux中一个C语言源程序P如下左；编译、汇编和链接后，第5、6、7行源码对应的指令序列如下右。

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

```
5  400300: movl  $0x1, 0x401028
6  40030a: movl  $0x3, 0x401fa0
7  400314: movl  $0x4, 0x409c40
```

三条指令在读指令时都不会发生缺页。

它们都位于起始地址为0x400000（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。

- 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?

页故障

- x86-64/Linux中一个C语言源程序P如下左；编译、汇编和链接后，第5、6、7行源码对应的指令序列如下右。

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

```
5  400300: movl  $0x1, 0x401028
6  40030a: movl  $0x3, 0x401fa0
7  400314: movl  $0x4, 0x409c40
```

第5行指令数据访问时发生页故障，缺页，可恢复。

对a[10]（地址0x401028）的访问是对所在页面（首址为0x401000）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。

- 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?

页故障

- x86-64/Linux中一个C语言源程序P如下左；编译、汇编和链接后，第5、6、7行源码对应的指令序列如下右。

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

```
5  400300: movl  $0x1, 0x401028
6  40030a: movl  $0x3, 0x401fa0
7  400314: movl  $0x4, 0x409c40
```

第6行指令数据访问没有页故障。

对a[1000]（地址0x401fa0）的访问是对所在页面（首址为0x040100）的第2次访问，已在主存，不会发生缺页。但a[1000]实际超出数组边界，0x401fa0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。

- 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？

页故障

- x86-64/Linux中一个C语言源程序P如下左；编译、汇编和链接后，第5、6、7行源码对应的指令序列如下右。

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

```
5  400300: movl  $0x1, 0x401028
6  40030a: movl  $0x3, 0x401fa0
7  400314: movl  $0x4, 0x409c40
```

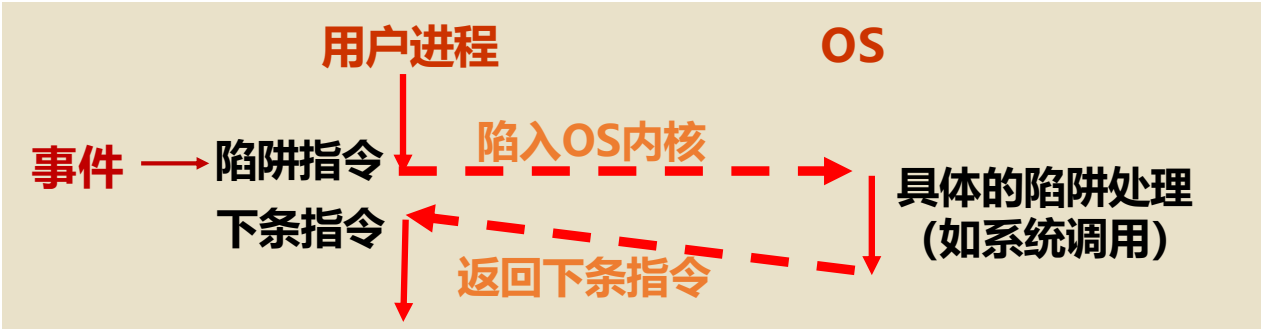
- 已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则：
  1. 对于上述三条指令的执行，在取指令时是否可能发生页故障？
  2. 在数据访问时分别会发生什么问题？
  3. 哪些问题是可恢复的？哪些问题是不可恢复的？

第7行指令数据访问时发生页故障，地址越界/越权，不可恢复。

地址0x409c40偏离数组首址0x401000已达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。

陷阱

- 陷阱（trap）也称自陷或陷入，执行陷阱指令（自陷指令/访管指令）时，CPU调出特定的操作系统控制程序进行相应处理，处理结束后返回到陷阱指令下一条指令执行。
  - 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。
  - x86-64处理器中的 int N（软中断，N为中断类型）指令和 syscall 指令（N为系统调用号）、MIPS处理器中的 syscall 指令等都属于陷阱指令。
  - 陷阱指令异常称为编程异常（programmed exception），由程序而非硬件发起



陷阱

- 系统调用（陷阱）是特殊异常事件，是OS为用户程序提供服务的手段。
- Linux提供了几百种系统调用，主要分为以下几类：
  - 进程控制、文件操作、文件系统操作、系统控制、内存管理、网络管理、用户管理、进程通信等
  - 系统调用号是系统调用跳转表索引值，跳转表给出系统调用服务例程首址

编号	名字	描述	编号	名字	描述
0	read	读文件	33	pause	挂起进程直到信号到达
1	write	写文件	37	alarm	调度告警信号的传送
2	open	打开文件	39	getpid	获得进程 ID
3	close	关闭文件	57	fork	创建进程
4	stat	获得文件信息	59	execve	执行一个程序
9	mmap	将内存页映射到文件	60	_exit	终止进程
12	brk	重置堆顶	61	wait4	等待一个进程终止
32	dup2	复制文件描述符	62	kill	发送信号到一个进程

x86-64/Linux的系统调用

- 系统调用 syscall N
  - Linux中系统调用所用参数通过寄存器传递，传递参数的寄存器顺序依次为：rax（调用号）、rdi、rsi、rdx、r10、r8、r9，除调用号以外，最多6个参数。系统调用的返回值在rax中。
- 系统调用通常被封装成用户程序能直接调用的函数，如exit()、read()和open()，这些是标准C库中系统调用对应的封装函数，称为系统级函数。
  - 封装函数机器级代码的统一结构：总是若干条传送指令后跟一条陷阱指令。
    - 传送指令用来传递系统调用的参数
    - 陷阱指令用来陷入内核进行处理
- 系统级函数可继续封装为高级语言提供的标准I/O库函数
  - 如在C语言程序中可以直接使用像fopen、fread、fwrite和fclose等文件操作函数，或printf、putc、scanf和getc等控制台I/O函数
  - 程序移植性好

陷阱

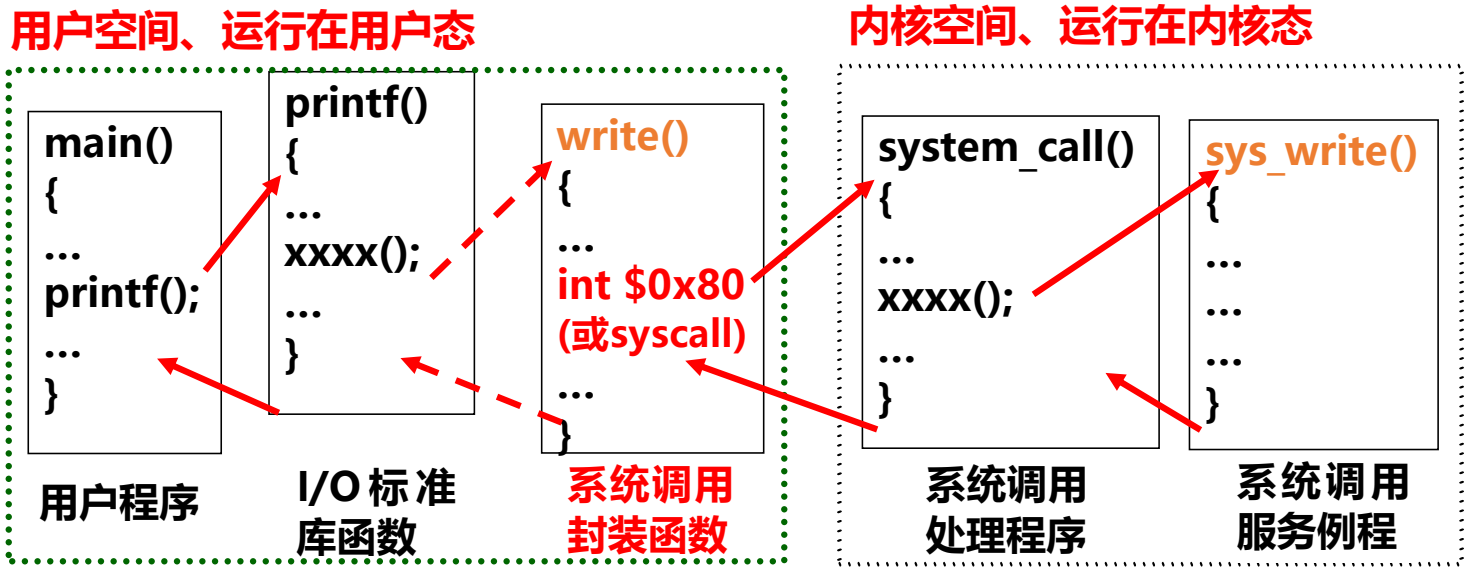
```
1   int main()
2   {
3       write(1, "hello, world\n", 13);
4       _exit(0);
5   }
```

```
1   .section .data
2   string:
3     .ascii "hello, world\n"
4   string_end:
5     .equ len, string_end - string
6   .section .text
7   .globl main
8   main:
      First, call write(1, "hello, world\n", 13)
9     movq $1, %rax          write is system call 1
10    movq $1, %rdi          Arg1: stdout has descrip
11    movq $string, %rsi     Arg2: hello world string
12    movq $len, %rdx        Arg3: string length
13    syscall                Make the system call
```

系统调用

- 以I/O请求为例，用户软件由两种请求方式
  - 使用语言提供的标准I/O库函数
    - 例如，在C语言程序中可以直接使用像fopen、fread、fwrite和fclose等文件操作函数，或printf、putc、scanf和getc等控制台I/O函数。
    - 程序移植性好
  - 使用OS提供的API函数或系统调用
    - 例如，在Windows中直接使用像CreateFile、ReadFile、WriteFile、CloseHandle等文件操作API函数，或ReadConsole、WriteConsole等控制台I/O的API函数。
    - 对于Unix或Linux用户程序，则直接使用像open、read、write、close等系统调用封装函数。

Linux系统中printf()函数的执行过程



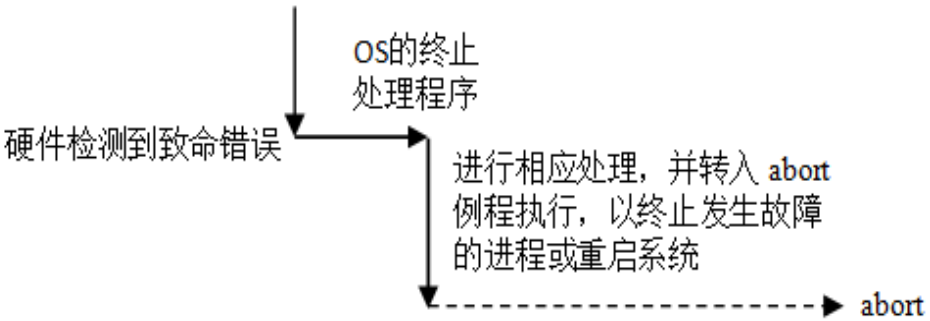
- 某函数调用执行到printf()语句时，会转到C语言I/O标准库函数printf()去执行
- printf()通过一系列函数调用，最终会转到系统调用write()
- 调用write()时会通过一系列步骤在内核空间中找到write对应的系统调用服务例程sys_write来执行
  - 系统调用号是系统调用跳转表索引值，跳转表给出系统调用服务例程首址

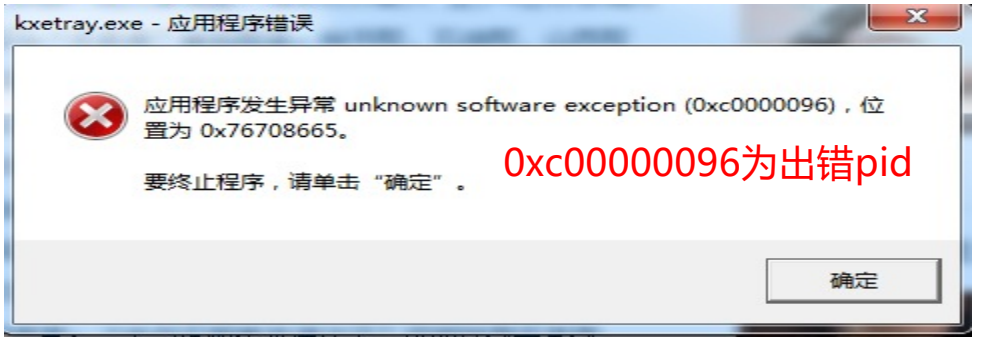
陷阱

- 利用陷阱机制实现程序调试功能，包括设置断点和单步跟踪
  - 调试器在断点处加一条“断点设置”陷阱指令 int 3（机器码 CCH）。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。
  - CPU单步跟踪状态（陷阱标记TF=1且中断标记IF=1）时，每条指令都被设置成陷阱指令，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”。

终止

- 执行指令过程中发生严重错误，程序无法继续执行，只好终止发生问题的进程，甚至在有些严重的情况下重启系统。
  - 例如，控制器出现问题，访问DRAM或SRAM时发生校验错等
- 这种异常是随机发生的，无法确定发生异常的是哪条指令





0xc00000096为出错pid

x86-64/Linux异常类型

- 256种不同类型的异常
- 每个异常都有唯一编号，称之为异常号（也称中断类型号，向量号）
  - 类型0为“除法错”
  - 类型2为“NMI中断”
  - 类型14为“缺页”
  - …
- 每个异常有对应的异常处理程序，入口地址放在一个专门的异常表中
- 前32个类型（0~31）保留给CPU使用，剩余的由OS自行定义
- 通过执行INT n（指令第二字节给出异常号n，n=32~255）使CPU自动转到OS给出的中断服务程序执行

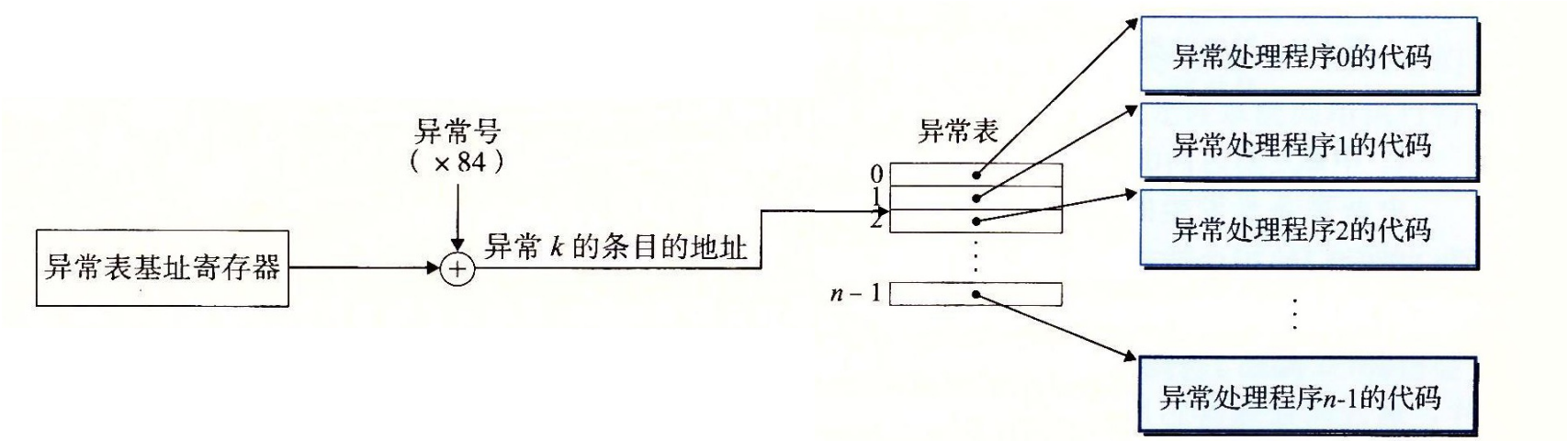
x86-64/Linux异常类型

- 操作系统定义的异常号为32~255
  - 可屏蔽中断通过CPU的INTR引脚向CPU发出中断请求
  - 陷阱
    - 软中断指令int n或系统调用syscall
    - Linux通过int $0x80指令将128号设定为系统调用
    - Windows通过int $0x21指令系统调用

类型号	助记符	含义描述	起因或发生源
0	#DE	除法出错	div 和 idiv 指令
1	#DB	单步跟踪	任何指令和数据引用
2		NMI 中断	不可屏蔽外部中断
3	#BP	断点	int 3 指令
4	#OF	溢出	into 指令
5	#BR	边界检测（BOUND）	bound 指令
6	#UD	无效操作码	不存在的指令操作码
7	#NM	协处理器不存在	浮点或 wait/fwait 指令
8	#DF	双重故障	处理一个异常时发生另一个
9	#MF	协处理器段越界	浮点指令
19	#XM	SIMD 浮点异常	SIMD 浮点指令
20-31		保留	
32-255		可屏蔽中断和软中断	INTR 中断或 INT *n* 指令

异常表

- 保护模式下，通过异常表（或称中断描述符表）获得异常处理程序或中断服务程序入口地址
- 中断描述符表（Interrupt Descriptor Table，IDT）是OS内核中的一个表，共有256个表项
  - IDTR中存放IDT在内存的首地址
  - 每一个表项是一个中断门描述符、陷阱门描述符

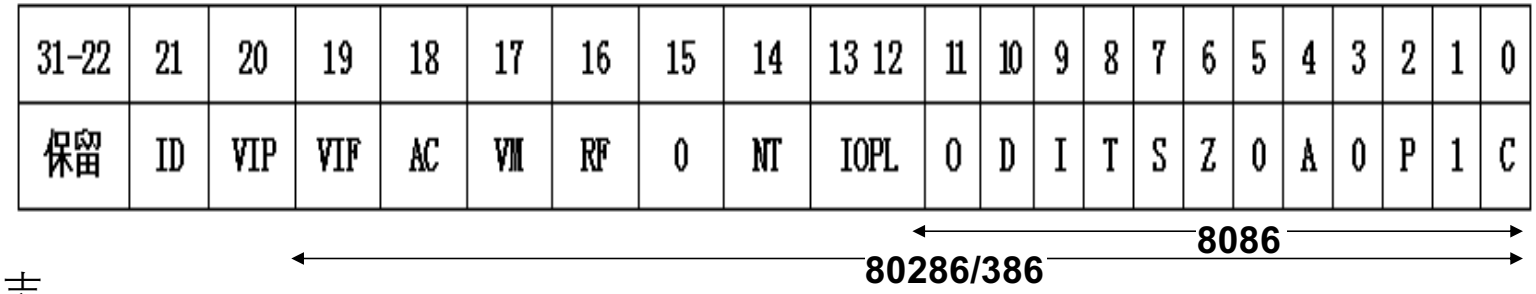


异常响应过程

- 检测到异常时，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EFLAGS寄存器
  3. 识别异常事件
     - 所有异常事件都被分配一个异常号，每个异常都有相应的“异常处理程序”
     - 可根据异常号找到异常处理程序的入口地址。

控制标志

- 标志寄存器



  - 6个条件标志
    - OF、SF、ZF、CF
    - AF：辅助进位标志（BCD码运算）
    - 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

异常控制流

- 第一讲：异常
  - 异常基本概念和分类、异常的响应和处理
  - x86-64+Linux下的异常机制
- 第二讲：进程与进程的上下文切换
  - 程序和进程的概念
  - 进程的逻辑控制流
  - 进程与进程的上下文切换

“程序”和“进程”

- 程序（program）
  - 按某种方式组合形成的代码和数据集合，代码即机器指令序列
  - 程序是一种静态概念
- 进程（process）
  - 程序的一次运行过程
  - 更确切说，进程是具有独立功能的一个程序关于某个数据集合的一次运行活动
  - 进程具有动态含义
  - 同一个程序处理不同的数据就是不同的进程
  - 进程是OS对CPU执行的程序的运行过程的一种抽象
    - 进程有自己的生命周期，由于任务的启动而创建，随着任务的完成（或终止）而消亡，它所占用的资源也随着进程的终止而释放。
  - 一个可执行目标文件（程序）可被加载执行多次，即一个程序可能对应多个不同的进程。

进程

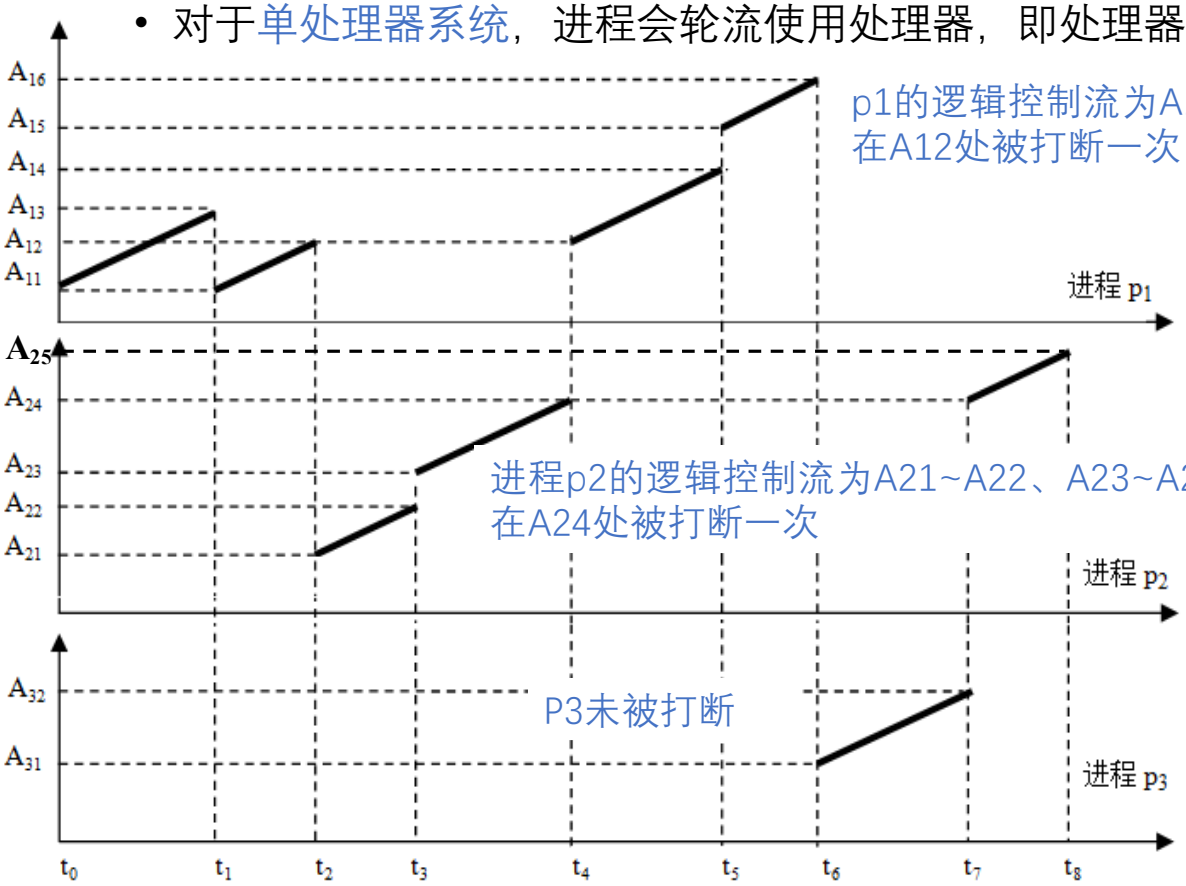
- 操作系统（管理任务）以外的都属于“用户”的任务。
- 计算机处理的所有“用户”的任务由进程完成。
- 为强调进程完成的是用户的任务，通常将进程称为用户进程。
- 计算机系统中的任务通常就是指进程。例如，
  - Linux内核中通常把进程称为任务，每个进程主要通过一个称为进程描述符（process descriptor）的结构来描述，其结构类型定义为task_struct，包含了一个进程的所有信息。
  - 所有进程通过一个双向循环链表实现的任务列表（task list）来描述，任务列表中每个元素是一个进程描述符。

引入进程的好处

- 进程的引入为应用程序提供了以下两方面的抽象：
  - 一个独立的逻辑控制流
    - 每个进程拥有一个独立的逻辑控制流，使得程序员以为自己的程序在执行过程中独占使用处理器
  - 一个私有的虚拟地址空间
    - 每个进程拥有一个私有的虚拟地址空间，使得程序员以为自己的程序在执行过程中独占使用存储器
- 进程的引入简化了程序员的编程以及语言处理系统的处理，即简化了编程、编译、链接、共享和加载等整个过程。

逻辑控制流

- 对于确定的数据集，某进程指令执行地址序列是确定的。称为进程的逻辑控制流。
- 对于单处理器系统，进程会轮流使用处理器，即处理器的物理控制流由多个逻辑控制流组成。



p1的逻辑控制流为A11~A13、A11~A14、 A15~A16
在A12处被打断一次

进程p2的逻辑控制流为A21~A22、A23~A25
在A24处被打断一次

P3未被打断

逻辑控制流不会因被其他进程打断而改变，还能回到“断点”处继续执行。

不同进程的逻辑控制流在时间上交错或重叠的情况称为并发（concurrency）

P1和P2是并发执行
P2和P3是并发执行
P1和P3不是并发执行

进程的上下文

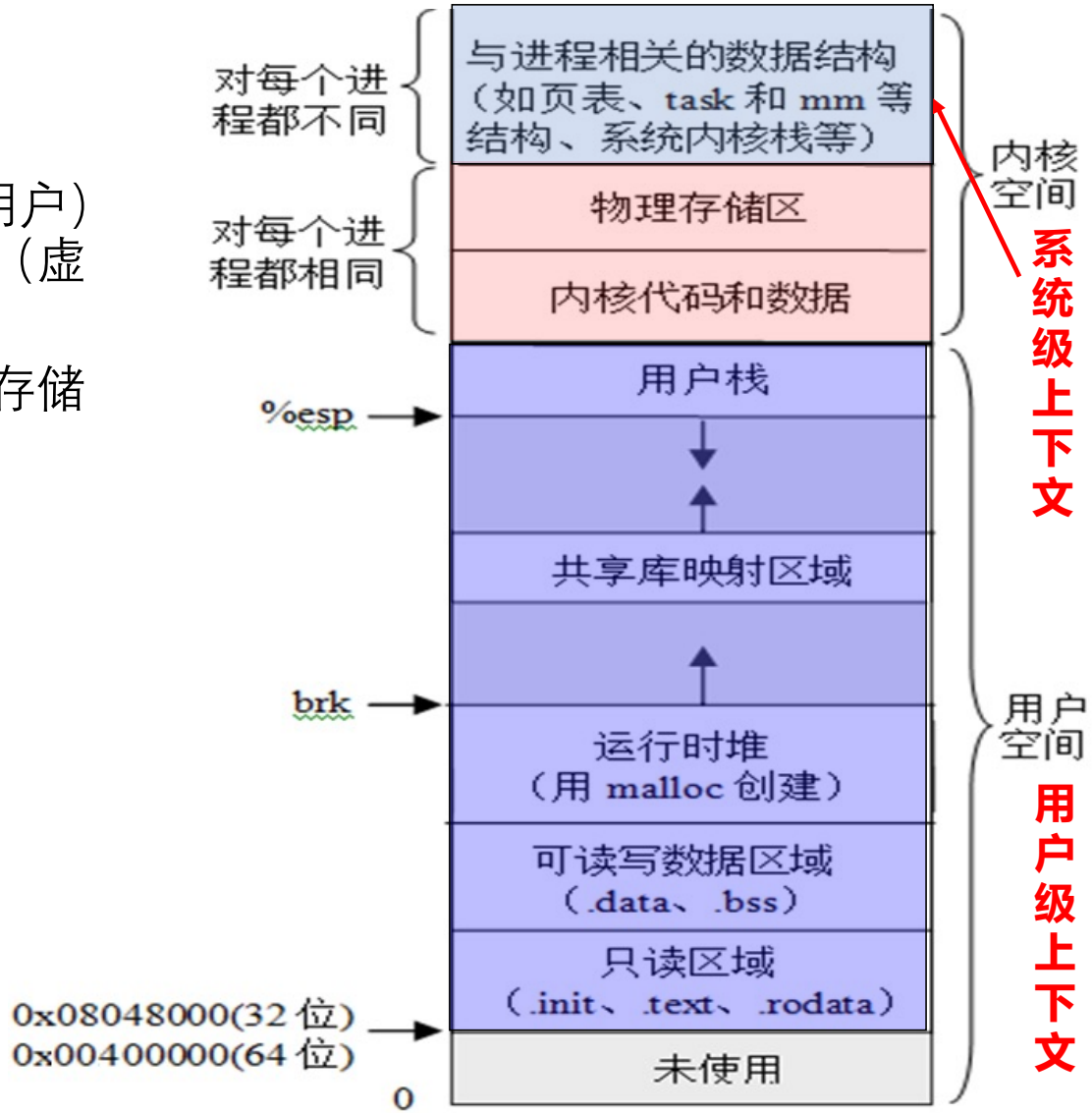
- 进程的物理实体（代码和数据等）和支持进程运行的环境合称为进程的上下文。
  - 由进程的程序块、数据块、运行时的堆和用户栈（两者统称为用户堆栈）等组成的用户空间信息被称为用户级上下文;
  - 由进程标识信息、进程现场信息、进程控制信息和系统内核栈等组成的内核空间信息被称为系统级上下文;
  - 处理器中各寄存器的内容被称为寄存器上下文（也称硬件上下文），即进程现场信息。
  - 在进行进程上下文切换时，操作系统把换下进程的寄存器上下文保存到系统级上下文中的现场信息位置。
  - 用户级上下文地址空间和系统级上下文地址空间一起构成了一个进程的整个存储器映像



进程的存储器映像

回顾：进程的地址空间

- x86-64/Linux平台下，每个（用户）进程具有独立的私有地址空间（虚拟地址空间）
- 每个进程的地址空间划分（即存储映像）布局相同



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 switch）
  1. 保存当前进程的上下文
  2. 恢复某个先前被抢占的进程被保存的上下文
  3. 将控制传递给这个新恢复的进程
- 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。
  - 中断或系统调用
  - 时间片轮转
  - 进程阻塞
  - 优先级调度

用户模式和内核模式

- 处理器通常是用某个控制寄存器中的一个模式位来提供这种功能的，该寄存器描述了进程当前享有的特权。
- 用户模式
  - 用户模式中的进程不允许执行特权指令，比如停止处理器、改变模式位，或者发起一个I/O操作。也不允许用户模式中的进程直接引用地址空间中内核区内的代码和数据。任何这样的尝试都会导致保护故障。
  - 用户程序必须通过系统调用接口间接地访问内核代码和数据。
- 内核模式
  - 运行在内核模式（或称超级用户模式）的进程可以执行指令集中的任何指令，并且可以访问系统中的任意内存位置。

用户模式和内核模式

- 运行应用程序代码的进程初始时是在用户模式中的。
- 进程从用户模式变为内核模式的唯一方法是通过诸如中断、故障或者陷入系统调用这样的异常。
- 当异常发生时，控制传递到异常处理程序，处理器将模式从用户模式变为内核模式。
- 处理程序运行在内核模式中，当它返回到应用程序代码时，处理器就把模式从内核模式改回到用户模式。

