15 | 标准库：信号与操作系统软中断有什么关系？

2022-01-17 于航

《深入C语言和程序运行原理》 课程介绍

讲述：于航

时长 16:15 大小 14.89M

你好，我是于航。

相信你在第一次学习编程时，也写出过与下面这段类似的代码：

复制代码

```
#include <stdio.h>
int main(void) {
  int x = 10;
  int y = 0;
  printf("%d", x / y);
  return 0;
}
```


可以很明显地看到，这里在代码中，我们通过 printf 函数打印出了除法表达式 x / y 的计算结果。但定睛一看，你就会发现：在这个表达式中，除数变量 y 对应的值为 0。因此，程序在

运行时便会发生“除零异常”。

当然，你可能会说，这只是我们故意构造的一段带有问题的程序。但在真实场景中，四则运算作为最基本的数学运算，被广泛应用在各类大型 C 项目中。而当参与运算的具体操作数可以被来自于用户的输入影响，且程序实现并没有进行完备的边界条件检查时，类似的运行时异常难免会发生。

除此之外，程序在运行过程中，都会直接或间接地与操作系统，甚至底层硬件进行交互。因此，你可能会遇到下面这几种情况：

- 程序运行时，由于访问了非法的内存资源，导致出现 “Segmentation Fault” 等异常；
- 用户想要结束程序，急切地按下了 Ctrl+C / Command+C 键；
- 计算机底层硬件系统出现故障，导致无法实现某项特定功能；
- ……

在这些情况下，应用程序应该如何响应？其实，我上面提到的所有问题，都可以通过“信号（Signal）”来解决。今天我们就来看看什么是信号，以及如何在 C 代码中通过标准库提供的相关接口与信号进行交互。

什么是信号？

从广义上来讲，信号实际上是操作系统提供的一种可以用来传递特定消息的机制。通过这种方式，操作系统可以将程序运行过程中发生的各类特殊情况转发给程序，并按照其指定的逻辑进行处理。每一种具体的信号都有它对应的名称，这些名称以 “SIG” 作为前缀。比如，当程序访问非法内存时，便会产生名为 SIGSEGV 的信号。而除零异常则对应着名为 SIGFPE 的信号。

信号的产生是一个随机的过程。毕竟，异常的出现并不是人们预期中会发生的事情。而这也就意味着，程序无法通过简单轮询某个全局变量值的方式，来检测信号是否出现，然后再进行相应处理。相对地，程序需要提前“告诉”操作系统：当某个信号到来时，应该按怎样的既定方式进行处理。而这，就是典型的**异步事件处理方式**。

如果我们再进一步，深入到操作系统内核，你会发现信号其实是一种“软中断”。

信号与软件中断

在计算机中，中断是指 CPU 需要临时暂停当前正在进行的活动，保存相应状态，然后转而去执行某个具体的中断服务程序（后面我将其简称为 ISR），以响应外部环境变化的过程。而在 ISR 执行完毕后，在大多数情况下，CPU 将继续执行之前暂停的任务。

通常来说，中断的触发分为两种形式，即硬件中断与软件中断（也被简称为硬中断与软中断）。其中，硬件中断是指**与计算机硬件特定状态相关的中断过程**，该过程直接由硬件触发。比如，当磁盘完成了某次由用户程序指定的 IO 操作后，便会通过硬件中断的方式来通知 CPU 这一消息，并让 CPU 进行后续的处理。在这个过程中，便存在着 CPU 执行流程从应用程序的某行机器指令，到磁盘中断处理程序代码的转移。

相对地，软件中断则是指由计算机软件，通过 `int` 等机器指令引起的 **CPU 执行流程的临时转移过程**。而我在这门课中多次提到过的“系统调用”便是其中的一种。在早期的 i386 架构中，用户程序需要通过指令 `int 0x80` 才能够借由软件中断从用户态进入到内核态，并使用内核提供的系统调用函数。同样地，在这个过程中也存在着 CPU 从用户代码到内核代码的执行流程转移。

回过头我们再来看看，为什么说信号也是一种软中断呢？其实很好理解。当特定事件（比如“除零”这个操作）发生时，操作系统会将对应的信号值（即对应的 SIGFPE）发送给相关程序。而通常情况下，如果应用程序并未设置自定义的信号处理程序，则操作系统将会执行默认信号处理程序中的逻辑。对于大多数信号来说，这个逻辑便是直接终止当前程序的运行。同样地，整个信号处理的过程中，也存在着 CPU 从用户程序到信号处理程序的执行流程转移。

讲到这里，我相信你已经基本理解了信号的概念。那么，在 C 程序中，应该如何与信号进行交互呢？

在 C 代码中与信号交互

C 语言自 C90 标准开始，便为我们提供了名为 signal.h 的头文件，以用于支持信号的相关功能。同样地，还是先来看一段代码：

复制代码

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
void sigHandler(int sig) {
```

```
  printf("Signal %d catched!\n", sig);
  exit(sig);
}
int main(void) {
  signal(SIGFPE, sigHandler);
  int x = 10;
  int y = 0;
  printf("%d", x / y);
}
```

这里，我在本讲开头处代码的基础上，进行了适当改进。在这段代码中，我们首先在 main 函数的开头处，使用标准库提供的 signal 函数，为当前程序设置了一段针对信号 SIGFPE 的自定义处理逻辑。

其中，该函数的第一个参数为具体的某个信号。C 标准库一共提供了 6 种不同类型的信号，它们均以宏常量的形式被定义。我将它们的具体信息整理在了下面这个表格中，供你参考。

信号类型（宏）	说明
SIGABRT	程序异常终止，如调用 abort 函数
SIGFPE	算数运算异常，如除零计算
SIGILL	程序执行了非法硬件指令
SIGINT	程序遇到终端中断信号，如用户敲击键盘上的 Ctrl+C
SIGSEGV	程序进行了无效内存引用，如访问未初始化的指针
SIGTERM	程序收到的终止信号

需要注意的是，该参数也支持使用某些与具体操作系统实现相关的其他信号值，但需要确保程序运行所在的系统支持这些非 C 标准信号。

第二个参数为具体的信号处理函数，它的原型需要满足 `void (*handler) (int)` 这样一种形式，即接收一个整型的信号值，但不返回任何内容（void）。

随着程序的运行，当位于代码第 12 行的除法运算发生除零异常时，操作系统便会将信号 SIGFPE 发送给当前进程，并根据之前通过 signal 函数注册的信号处理信息，调用用户自定义或默认的信号处理函数。

需要注意的是，信号处理函数的调用过程是由来自操作系统内核的软中断触发的，因此，这个过程与我们平时见到的，通过 `call` 指令进行的函数调用过程并不相同。总的来看，上述应用程序和操作系统，围绕信号的基本交互逻辑可以被粗略地描述为以下几个步骤：

1. CPU 执行除法指令 `idiv`；
2. 发现除零异常，CPU 暂停当前程序运行，并将控制权转交给操作系统；
3. 操作系统将信号 SIGFPE 发送给出错的程序；
4. 操作系统根据情况执行相应的信号处理程序（函数）；
5. 信号处理程序执行完毕后，若程序未退出，则将程序执行恢复到之前的中断点，即 CPU 会重新执行 `idiv` 指令。

当然，除了上面这种自定义信号处理函数的方式外，C 标准库还为我们提供了两种基本的信号处理方式。同样地，它们也以宏的形式被定义，可以直接作为 signal 函数的第二个参数来使用。我将它们整理在了下面的表格中，供你参考：

信号处理函数（宏）	说明
SIG_DFL	使用默认的信号处理函数
SIG_IGN	忽略该信号

其中，第一种 SIG_DFL 表示对信号进行默认处理，操作系统会按照信号既定的默认处理方式来处理程序；SIG_IGN 则会忽略程序收到的信号。

但需要注意的是，并非所有类型的信号都可以被忽略，比如某些难以恢复的软硬件异常对应的信号，或是 Linux 上专有的 SIGKILL 和 SIGSTOP 信号等。关于它们的一个简单使用示例如

下所示：

```
1 #include <signal.h>
2 #include <stdio.h>
3 int main(void) {
4   signal(SIGTERM, SIG_IGN);  // 忽略信号 SIGTERM;
5   raise(SIGTERM);  // 向当前程序发送 SIGTERM 信号;
6   printf("Reachable!\n");  // Reachable code!
7   return 0;
8 }
```

这里，在代码的第 5 行，我们还使用了由标准库提供的 raise 函数。该函数可以让我们在代码中直接向当前程序发送指定的信号。在上面的代码中，我们为信号 SIGTERM 设置了 SIG_IGN 作为它的处理方式，因此，当执行到 raise 函数后，虽然它向程序发送了 SIGTERM 信号，但程序却不会被立即终止。相反，它将继续执行第 6 行的 printf 函数，然后以正常方式退出。

可以看到，操作系统对信号处理函数的调用，可能会发生在整个应用程序运行过程中的任意时刻。而在某些情况下，这可能会给程序的实际执行带来影响。接下来，让我们进一步看看这个过程是如何影响程序运行的。

可重入函数

试想这样一种情况：某一时刻，CPU 还在正常执行 main 函数内函数 A 的代码，而由于某些外部原因，程序在此刻收到了某个信号，操作系统便暂停当前程序运行，并将执行流程调度至对应的信号处理函数。而在这个信号处理函数中，函数 A 又被再次调用。当信号处理完毕后，执行流返回了之前的“中断点”，并继续处理 main 函数内，函数 A 中还未被执行到的指令。

那么，在这种情况下，信号处理函数中对函数 A 的再次调用，会不会影响之前还未调用完成的函数 A 的执行状态呢？让我们来看下面这段“模拟”代码：


```
1 #include <stdio.h>
2 #include <signal.h>
3 #include <string.h>
4 #define BUF_SIZE 16  // 全局静态数组大小;
5 #define FORMAT_NUM_(N) " $"#N
6 #define FORMAT_NUM(N) FORMAT_NUM_(N)
```

```
#define RAISE_EXP_false_ASM()
// 调用 raise 函数向当前程序发送信号；
#define RAISE_EXP_true_ASM() \
  "movl    $4, %%edi\n\t" \
  "call    raise\n\t"
// 内联汇编实现；
#define INLINE_ASM(ID, HAS_EXP) \
  "mov     %0, %%r8\n\t" /* 复制传入的字符串数据到全局静态数组 */ \
  "testq   %%rsi, %%rsi\n\t" \
  "je      .L1" #ID "\n\t" \
  "xorl    %%eax, %%eax\n\t" \
  ".L3" #ID ":\n\t" \
  "movzbl  (%%rdi,%%rax), %%ecx\n\t" \
  "movb    %%cl, (%%r8,%%rax)\n\t" \
  "addq    $1, %%rax\n\t" \
  "cmpq    %%rsi, %%rax\n\t" \
  "jne     .L3" #ID "\n\t" \
  ".L1" #ID ":\n\t" \
  RAISE_EXP_##HAS_EXP##_ASM() /* 选择性调用 raise 函数 */ \
  "mov     $1, %%rax\n\t" \
  "mov     $1, %%rdi\n\t" \
  "mov     %0, %%rsi\n\t" \
  "mov" FORMAT_NUM(BUF_SIZE) ", %%rdx\n\t" \
  "syscall\n\t"  /* 触发系统调用，打印内容 */

static char buf[BUF_SIZE];  // 用于保存字符的全局静态数组；
void print_with_exp(const char* str, size_t len) {  // 会引起信号中断的版本；
  asm(INLINE_ASM(a, true) :: "g" (buf));
}
void print_normal(const char* str, size_t len) {  // 正常的版本；
  asm(INLINE_ASM(b, false) :: "g" (buf));
}
void sigHandler(int sig) {
  const char* str = "Hello";
  print_normal(str, strlen(str));
}
int main(void) {
  signal(SIGILL, sigHandler);
  const char* str = ", world!";
  print_with_exp(str, strlen(str));
  return 0;
}
```


第一眼看上去，这段代码可能稍显复杂，但它的执行逻辑实际上十分简单。在 main 函数中，我们首先通过 signal 函数，为信号 SIGILL 注册了一个自定义信号处理函数。接下来，我们调用了名为 print_with_exp 的函数，并传入了内容为 “Hello” 的字符串 str，以及相应的长度信息。函数在调用后会打印出这个字符串的完整内容。这里，借助函数 print_with_exp，我们将会模拟上面提到的，位于 main 函数中的函数 A 的执行流程。

但与真实案例稍有不同的是，为了模拟操作系统向程序随机发送信号的过程，这里我们选择直接在函数 print_with_exp 中，通过调用 raise 函数的方式来做到这一点。当然，你可能会说“这并不随机”，但在现实案例中，类似的情况却可能发生。

为了从更加细微的角度，直观地观察信号处理函数的调用对“函数重入”的影响，这里我们选择直接用内联汇编的方式来实现 print_with_exp 函数。相应地，名为 print_normal 的函数，其功能与 print_with_exp 相同，只是在这个函数中，我们不会调用 raise 函数。你可以简单地将这两个函数视为功能完全一致的 “print” 函数。

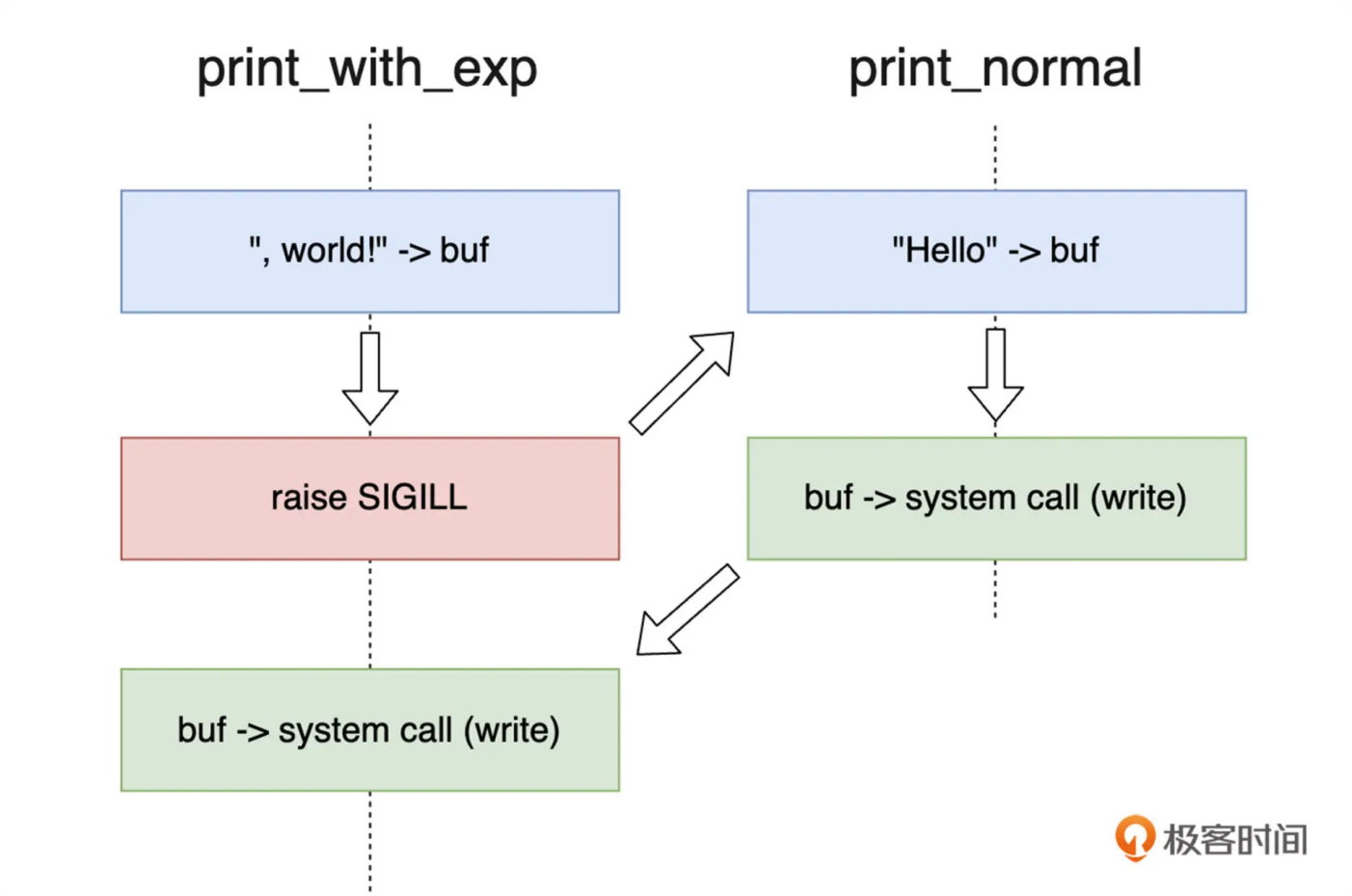
为了方便组织代码，以将相同功能的实现部分进行抽离，我们通过宏的方式“封装”了这两个函数的内联汇编内容。位于代码第 13 行的宏函数 INLINE_ASM，将会根据传入参数的不同，来选择性地生成不同的汇编代码。

在这两个函数的汇编实现中，我们首先会将由第一个参数传入的字符串内容，根据第二个参数传入的长度信息逐字符复制到程序中的全局静态数组 buf 中。然后，通过相应的系统调用，再将该数组中的内容输出到终端。只是对于 print_with_exp 函数来说，在此之前，我们会通过调用 raise 函数将信号 SIGILL 发送给当前程序。你可以参考代码中的注释来了解这部分实现。

到这里，按照我们的理解，程序正常运行后，应该会按顺序进行两次输出（对应两次系统调用），并完整打印字符串 “Hello, world!”。其中，前半段由首先调用完成的信号处理函数 sigHandler 中的 print_normal 函数输出；后半段由 main 函数内的 print_with_exp 函数输出。

但真实情况却并非如此。程序运行后，你会发现实际的输出内容是 “Hellold!Hellold!”。这是为什么呢？

仔细查看代码可以看到，函数 print_with_exp 在调用时，存入全局静态数组 buf 中的字符串内容 “, world!”，会被信号处理代码中 print_normal 函数调用时存入的字符串 “Hello”，覆盖了一部分。而当执行流再次退回到 print_with_exp 函数中继续执行时，由于此时数组 buf 中的内容已经发生改变，因此便打印出了错误的结果。你可以参考下图，来理解上述程序的执行流程。



对于这种一旦在执行中途被临时打断，在此刻再次调用就可能影响前一次调用正常执行的函数，我们一般称之为“不可重入函数”。相反，**不受中途打断和重新调用影响的函数，便被称为“可重入函数”**。而在信号处理程序中，“尽量使用可重入函数”便是我们需要遵守的首要规则。虽然大多数情况下，在信号处理代码中对它们的调用都可以正常工作，但从程序健壮性的角度来看，却并不推荐这样做。

实际上，包括 printf、exit，以及 malloc 在内的众多常用 C 标准库函数，它们都并非可重入函数。因此，我们能够在信号处理代码中进行的操作也并不多。而为了进一步简化信号处理模型，C 标准库为我们提供了一个名为 `sig_atomic_t` 的整数类型，该类型可以保证，**即使存在可能由于中断导致的执行流程转移，对该类型变量的读写过程也都是原子的**（注意，不要与并发编程中的原子类型混淆，标准中仅规定了 `sig_atomic_t` 类型在异步信号处理场景下的原子性）。

在这种情况下，对信号的处理过程可以变为如下代码所示的模式：

复制代码

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
```

```c
#include <stdlib.h>
volatile sig_atomic_t sig = 0;
void sigHandler(int signal) {
  sig = signal;
}
int main(void) {
  signal(SIGINT, sigHandler);
  int counter = 0;  // 计数器变量;
  while(1) {
    switch (sig) {  // 信号筛选与处理;
      case SIGINT: {
        printf("SignalValue: %d", sig);
        /* 异常处理的主要逻辑 */
        exit(SIGINT);
      }
    }
    if (counter == 5) raise(SIGINT);
    printf("Counter: %d\n", counter++);
    sleep(1);
  }
  return 0;
}
```

这里，在 main 函数中，我们使用死循环来表示一个常驻程序（比如 HTTP 服务器）的执行状态。在代码的第 10 行，通过 signal 函数，我们为信号 SIGINT 注册了相应的处理函数 sigHandler。在该函数内部，`sig_atomic_t` 类型变量 sig 的值被更新为收到的信号值。回到主函数，在代码的第 13~19 行，我们在主循环每次迭代的开头处，通过检测变量 sig 的值，来判断当前程序是否收到信号，并作出相应处理。而在第 18 行，当 counter 变量的值被递增到 5 时，raise 函数向当前程序发送了信号 SIGINT，从而模拟了程序收到信号，发生软中断时的场景。

这是一种常见的信号处理模式，并非适合所有场景。但无论如何，你都需要注意：对不可重入函数在信号处理程序中的不当使用，可能会给程序的运行埋下问题。

多线程应用的信号处理

学习到这里，对于单线程应用，我们已经可以很好地在程序中与信号进行交互。但遗憾的是，C 语言并没有对并发编程中的信号处理做任何规范上的约束和建议。这意味着，在多线程应用中使用 signal 和 raise 函数，可能会产生未定义行为。这些函数会如何影响程序执行，则取决于不同 C 标准库在具体操作系统上的实现。

所以，如果你对程序的可移植性及健壮性有要求，那么请不要在多线程应用中使用信号处理。作为替代，你可以考虑通过宏的方式来区分程序运行所在的不同操作系统，然后分别使用 POSIX、Win32 等 API ，来以不同方式实现多线程模式下的信号处理过程。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

这一讲我主要介绍了 C 语言中与信号处理相关的内容，包括信号的基本概念，如何在 C 程序中使用信号处理函数，以及信号中断可能对程序运行带来的影响等。

借助信号，操作系统可以将程序运行过程中发生的特殊情况及时通知给程序，并按照其要求的特定方式，或系统的默认方式进行处理。而信号本质上是一种软件中断，即操作系统通过特殊机器指令引起的 CPU 执行流程的临时转移。

在 C 语言中，可以通过引入标准库头文件 signal.h 的方式，来使用相关的信号处理接口。其中，通过 signal 函数，我们可以为当前程序设置针对某个信号的自定义处理程序；而 raise 函数则为我们提供了可以向当前程序发送指定信号的能力。C 标准中一共规定了 6 种不同类型的信号，以及两种默认的信号处理方式，它们均以宏的形式被定义。

信号的中断过程可以在任意时刻发生，因此，为了保证程序的健壮性，我们只能在信号处理程序中使用“可重入函数”。这类函数的重新执行，不会对前一个还未执行完成的函数的最终执行结果产生影响。为了简化信号的处理方式，C 标准还为我们提供了名为 `sig_atomic_t` 的整数类型。这也就意味着，标准可以保证即使在发生信号中断的情况下，对该类型变量的读写也都是原子性的。

最后，C 标准并未对如何在多线程应用中进行信号处理作过多说明。因此，在并发编程场景中，对 signal 和 raise 等函数的不当使用方式可能会产生未定义行为。

思考题

你知道 Linux 中的 sys_kill 系统调用有什么作用吗？常见的命令行操作 “kill -9” 又是什么意思呢？欢迎在评论区告诉我你的理解。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

上一篇　14 | 标准库：如何使用互斥量等技术协调线程运行？

下一篇　16 | 标准库：日期、时间与实用函数

精选留言 (5)

写留言

白凤凰

2022-01-17

“信号处理程序执行完毕后，若程序未退出，则将程序执行恢复到之前的中断点，即 CPU 会重新执行 idiv 指令。“　这句话我不太理解，不应该是执行idiv的下一条指令吗？

作者回复: 因为 CPU 执行到 idiv 时发生了异常，而该指令实际上是还没有执行完成的（执行过程中发生了异常）。因此当信号处理函数结束后，还会再次“尝试”去重新执行这个指令。你可以试着去掉代码第 6 行的 exit 函数调用，然后再看看整个程序的执行结果。这个时候是不是会连续多次打印出 “Signal 8 catched!” 呢？

共 5 条评论 >　　1

shk1230
2022-03-02

“你知道 Linux 中的 sys_kill 系统调用有什么作用吗？常见的命令行操作 “kill -9” 又是什么意思呢？欢迎在评论区告诉我你的理解。”
我的理解：sys_kill是发送SIGKILL信号，程序不可忽略，也不能注册处理函数，保证操系统的控制权。kill -9也是发送这样的信号。

作者回复: 回答正确！

englefly
2022-01-21

print_with_exp 的那段代码怎么编译呢？
编译命令 gcc sigill.c -o ill -fPIE
得到错误如下：
/usr/bin/ld: /tmp/ccdPmLJC.o: relocation R_X86_64_32S against `.bss' can not be used when making a PIE object; recompile with -fPIE
collect2: error: ld returned 1 exit status

作者回复: 直接用 “gcc sigill.c -o ill” 就可以哈。

ZR2021
2022-01-18

原来是发现异常指令后主动将控制权交给操作系统，操作系统发信号给进程做相应处理的，以前一直不明白我不调用任何系统调用的时候操作系统怎么就发了个信号过来，不过有个问题想请教下老师，哪些异常指令会将控制权交给操作系统的，这些异常应该是跟具体平台如x86有关的吧，而不是和操作系统相关

作者回复: “哪些异常指令会将控制权交给操作系统”，这个我倒是没有系统总结过。一般规律是：只要是影响了程序正常执行的指令错误，操作系统都会以“信号”的方式通知应用程序。异常的发生与具体平台的指令有关，但信号是由 POSIX 标准制定的，具体的映射关系，由操作系统控制（比如硬件发生的除零异常会对应到 SIGFPE，等等）。


疯码

2022-01-17

这么多限制下，信号处理函数似乎做不到异常恢复。是不是还是高级语言的针对每一段代码的try catch才有用

作者回复: 当信号处理函数执行完毕后，是可以再次回到程序逻辑继续执行的，比如通过 longjmp 等方式。但哪些异常可以恢复、可以被忽略，就要具体看了。本质上来看，信号捕获和 try...catch 这两个概念还是比较独立的。一个与操作系统实现紧密相关（比如 POSIX 标准），另一个则是独立于系统的语言特性。