【ICS】异常控制流 - Exceptional Control Flow

1. What ECF?

异常控制流（Exceptional Control Flow）。

1.1 控制流

程序的处理器从上电到断电，运行了一连串指令。下图中的每个 `ai` 表示某条指令的地址：

$$a_0, a_1, \ldots, a_{n-1}$$

这整一条指令序列就是控制流。

最简单的控制流：平滑的序列。 `I_k` 和 `I_k+1` 是相邻的。

所以什么是异常控制流？就是控制流发生突变了。**计算机系统**通过**异常控制流**来对**系统状态的变化做出反应**。

比如：

- 硬件层：硬件检测到的事件会触发控制突然转移到异常处理程序
- 操作系统层：内核通过上下文切换将控制从一个用户进程转移到另一个
- 应用层：一个进程可以发送信号到另一个进程，接收者会将控制突然转移到它的一个信号处理程序

> 《操作系统玩具——设计与实现》
>
> - 首先，什么是CPU？就是无情的执行指令的机器。
> - 指令分为两种：
>   - 计算用的指令，把寄存器的值拿出来，算一算丢到内存或者寄存器
>   - 就是所谓的系统调用——**把我的状态暴露给外面，然后把控制权交给外面，外面也可以把状态传进来**

操作系统玩具：API

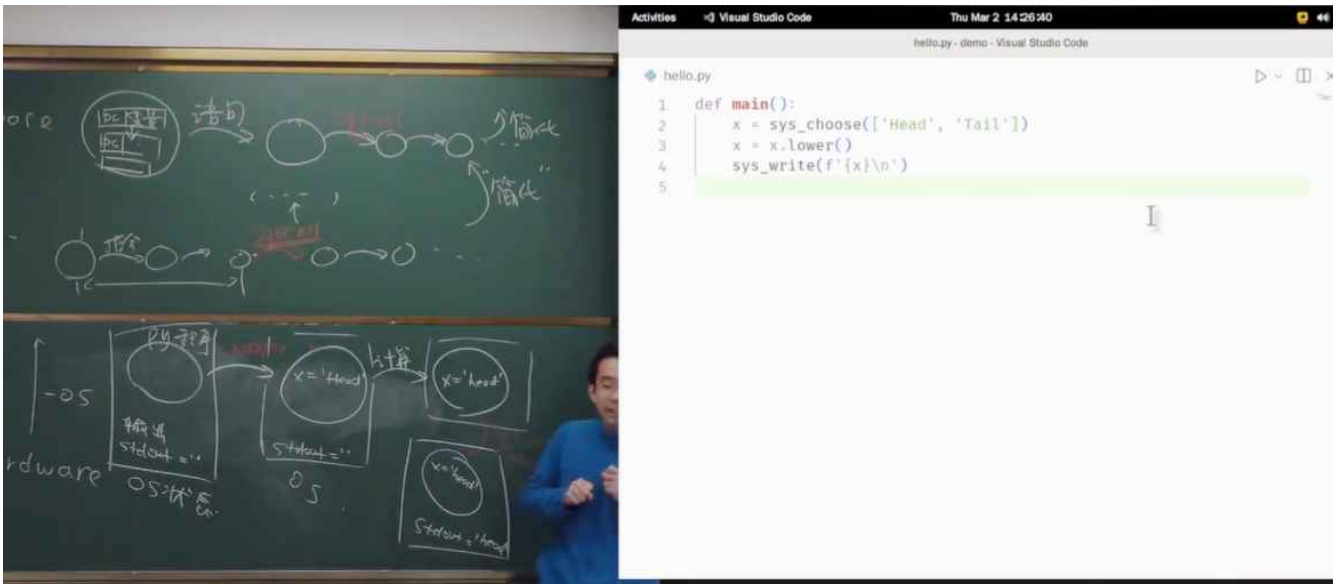
四个“系统调用” API

- choose(xs): 返回 xs 中的一个随机选项
- write(s): 输出字符串 s
- spawn(fn): 创建一个可运行的状态机 fn
- sched(): 随机切换到任意状态机执行

除此之外，所有的代码都是确定 (deterministic) 的纯粹计算

- 允许使用 list, dict 等数据结构

- 程序就是状态机。操作系统就是c程序，整个计算机系统也是一个状态机——
  - c程序是有自己的行为的定义的，就是每次执行一个语句，从c语言的一个状态到下一个状态（当然我们这里指的是普通的计算），如果c语言我们禁止你调用任何的库函数，禁止你写汇编，不向外部调用，你甚至都不能终止（除非你程序bug爆了），那么必然是有向外部的系统调用的。



我们执行的时候，圆圈里面就是我们的程序；当我们执行到sys_write的时候，我们发现：（从程序的视角）状态似乎没有改变啊！这个时候我们要关注的就是os本身——屏幕上也输出了！os单独我们也可以拉出来认为是一个状态机（操作系统的状态，一个更大的状态机）。

同时，操作系统里面有很多很多程序！所以操作系统可以认为是状态机的管理者！

- `spawn`：创建一个新的状态机出来。我们可以给它传一个函数，可以给这个函数传一个参数。`spawn` 的行为是？（感觉你在创建一个进程/线程对不对？）当前的状态机的pc动了，同时在操作系统的世界里，多出来了一个状态机！
  - **哦！这就是今天的操作系统！**

2. Why ECF?

为什么我们要学习ECF?

- 正如上面我们引用的绿导师的内容。我们不难意识到，理解ECF可以让我们明白：
  - 应用程序是如何与OS进行交互的
  - 什么是并发?
  - 还有高级软件的异常处理——比如CPP/Java（所谓try-catch其实就是高级语言提供的应用级别ECF）

3. How ECF?

3.1 异常

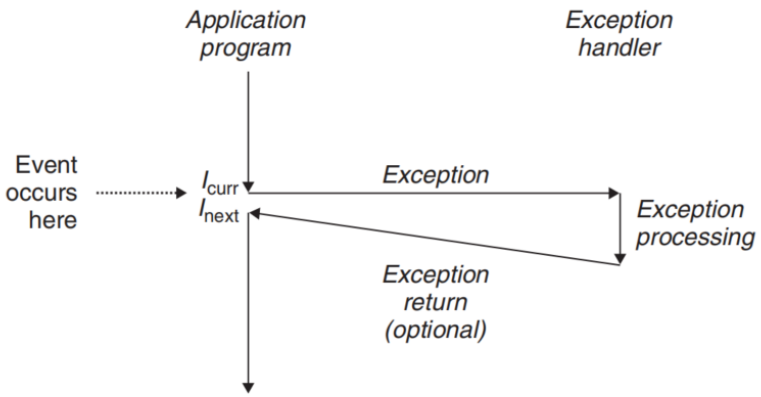


Figure 8.1
Anatomy of an exception. A change in the processor's state (an event) triggers an abrupt control transfer (an exception) from the application program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program or aborts.

In any case, when the processor detects that the event has occurred, it makes an indirect procedure call (the exception), through a jump table called an *exception table*, to an operating system subroutine (the *exception handler*) that is specifically designed to process this particular kind of event. When the exception handler finishes processing, one of three things happens, depending on the type of event that caused the exception:

1. The handler returns control to the current instruction I_{curr}, the instruction that was executing when the event occurred.
2. The handler returns control to I_{next}, the instruction that would have executed next had the exception not occurred.
3. The handler aborts the interrupted program.

Section 8.1.2 says more about these possibilities.

需要注意的点：

- 异常处理程序运行在kernal mode——意味着它们对所有系统资源有着完全的访问权限。
  - 如果一个用户程序触发异常，然后返回，那么整个流程就是：用户态-内核态-用户态

3.2 进程

正因为有了所谓的“异常控制流”也就是指令可以跳来跳去，我们才有了“进程”的概念！

什么是进程?

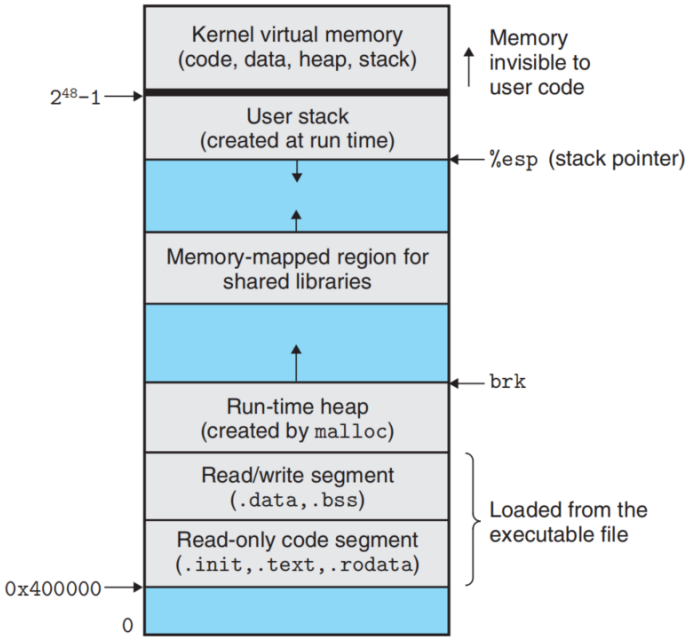
又回到了上面的黑板——说穿了，就是一个正在跑的状态机，不是么:)

进程给应用程序提供的关键抽象：

- 我们的程序在独占cpu
- 私有的地址空间——我们的程序在独占内存

经典老图了：

Figure 8.13
Process address space.



3.2.1 User mode, kernal mode

我们说了程序都是状态机，操作系统也是程序，那凭什么我是最XX的呢？那处理器就必须限制应用程序可以执行的指令和它可以访问的地址空间的范围了！

- 只需要寄存器中的一个mode bit就可以了！
- 总之，用户程序必须通过**系统调用接口间接**访问内核代码和数据。

3.2.2 Context Switch

什么是上下文？

> ## 实现系统调用
>
> 有些“系统调用”的实现是显而易见的
>
> ```
> def sys_write(s): print(s)
> def sys_choose(xs): return random.choice(xs)
> def sys_spawn(t): runnables.append(t)
> ```
>
> 有些就困难了
>
> ```
> def sys_sched():
>     raise NotImplementedError('No idea how')
> ```
>
> 我们需要
>
> - 封存当前状态机的状态
> - 恢复另一个“被封存”状态机的执行
>   - 没错，我们离真正的“分时操作系统”就只差这一步

实现系统调用

有些“系统调用”的实现是显而易见的

```
def sys_write(s): print(s)
def sys_choose(xs): return random.choice(xs)
def sys_spawn(t): runnables.append(t)
```

有些就困难了

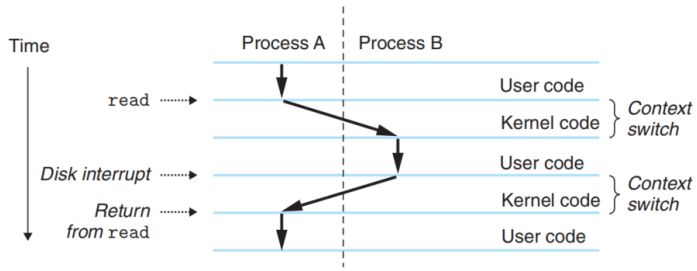
```
def sys_sched():
    raise NotImplementedError('No idea how')
```

我们需要

- 封存当前状态机的状态
- 恢复另一个“被封存”状态机的执行
  - 没错，我们离真正的“分时操作系统”就只差这一步

之前封存的状态机的状态就是“上下文”。

Figure 8.14
Anatomy of a process context switch.



3.3 系统调用错误处理

> 凡是调用sys call，检查返回值！！！！！
>
> ——大爷

3.4 进程控制

从程序员的角度，进程处于下面三种状态之一：

Running. The process is either executing on the CPU or waiting to be executed and will eventually be scheduled by the kernel.

Stopped. The execution of the process is *suspended* and will not be scheduled. A process stops as a result of receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal, and it remains stopped until it receives a SIGCONT signal, at which point it becomes running again. (A *signal* is a form of software interrupt that we will describe in detail in Section 8.5.)

Terminated. The process is stopped permanently. A process becomes terminated for one of three reasons: (1) receiving a signal whose default action is to terminate the process, (2) returning from the main routine, or (3) calling the `exit` function.

3.4.1 Fork

这里只需要一些简单的例子就能说明白很多问题了：

code/ecf/fork.c

```
1   int main()
2   {
3       pid_t pid;
4       int x = 1;
5
6       pid = Fork();
7       if (pid == 0) {  /* Child */
8           printf("child : x=%d\n", ++x);
9           exit(0);
10      }
11
12      /* Parent */
13      printf("parent: x=%d\n", --x);
14      exit(0);
15  }
```

code/ecf/fork.c

Figure 8.15 **Using `fork` to create a new process.**

line7开始的代码是父子共用的。

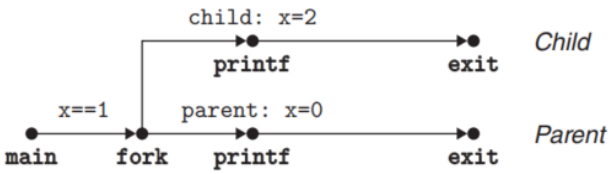
When we run the program on our Unix system, we get the following result:

```
linux> ./fork
parent: x=0
child : x=2
```

有几个需要注意的有趣的细节：

- 相同但是各自独立的地址空间：父子进程各自都是独立的进程，因此它们都有自己私有的地址空间，因此对变量 `x` 做出的任何改变都是独立的
- 共享文件：注意到父子进程都往屏幕上输出了东西，因为子进程继承了父进程所有的打开文件！父进程调用fork时， `stdout` 文件是打开的，并且指向屏幕。

Figure 8.16
Process graph for the example program in Figure 8.15.



```
1   int main()
2   {
3       Fork();
4       Fork();
5       printf("hello\n");
6       exit(0);
7   }
```

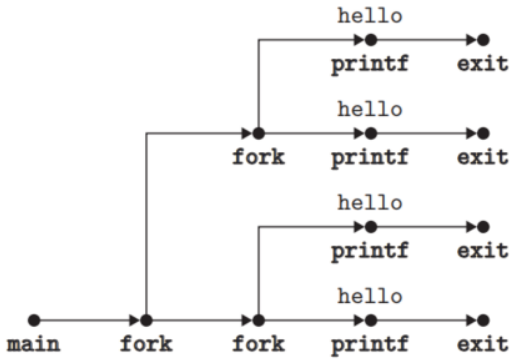


Figure 8.17 **Process graph for a nested `fork`.**

3.4.2 Waitpid: zombie!

进程终止时，内核并不马上将其清除；相反，进程被保存在一种已经终止的状态中，直到被它爹回收。当父进程回收已经终止的子进程，内核将该子进程退出状态传递给爸爸，然后彻底扔了它。

一个终止了但是还没被回收的进程就是zombie。

如果一个父进程终止了，它的儿子们全部变成了孤儿！内核就安排 `init` 进程成为这群可怜的小朋友的养父—— `init` 进程的pid是1，是所有进程的祖先，内核安排它回收那些可怜的孤儿们。

对于长时间运行的进程——比如shell、服务器，总算应该回收它们的僵尸进程，来释放资源。

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *statusp, int options);
                    Returns: PID of child if OK, 0 (if WNOHANG), or −1 on error
```

code/ecf/waitpid1.c

```
1   #include "csapp.h"
2   #define N 2
3
4   int main()
5   {
6       int status, i;
7       pid_t pid;
8
9       /* Parent creates N children */
10      for (i = 0; i < N; i++)
11          if ((pid = Fork()) == 0)  /* Child */
12              exit(100+i);
13
14      /* Parent reaps N children in no particular order */
15      while ((pid = waitpid(-1, &status, 0)) > 0) {
16          if (WIFEXITED(status))
17              printf("child %d terminated normally with exit status=%d\n",
18                     pid, WEXITSTATUS(status));
19          else
20              printf("child %d terminated abnormally\n", pid);
21      }
22
23      /* The only normal termination is if there are no more children */
24      if (errno != ECHILD)
25          unix_error("waitpid error");
26
27      exit(0);
28  }
```

code/ecf/waitpid1.c

Figure 8.18 **Using the `waitpid` function to reap zombie children in no particular order.**

```
linux> ./waitpid1
child 22966 terminated normally with exit status=100
child 22967 terminated normally with exit status=101
```

Notice that the program reaps its children in no particular order. The order that they were reaped is a property of this specific computer system. On another system, or even another execution on the same system, the two children might have been reaped in the opposite order. This is an example of the *nondeterministic* behavior that can make reasoning about concurrency so difficult. Either of the two possible outcomes is equally correct, and as a programmer you may *never* assume that one outcome will always occur, no matter how unlikely the other outcome appears to be. The only correct assumption is that each possible outcome is equally likely.

3.4.3 加载并运行程序

The execve function loads and runs a new program in the context of the current process.

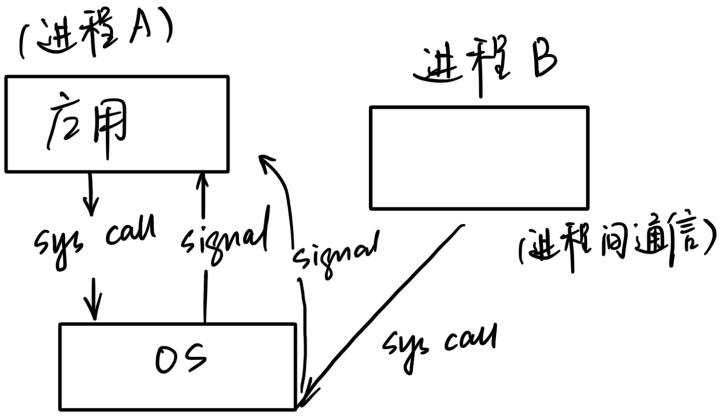
```
#include <unistd.h>

int execve(const char *filename, const char *argv[],
           const char *envp[]);
                                    Does not return if OK; returns −1 on error
```

The execve function loads and runs the executable object file `filename` with the argument list `argv` and the environment variable list `envp`. `Execve` returns to the calling program only if there is an error, such as not being able to find `filename`. So unlike `fork`, which is called once but returns twice, `execve` is called once and never returns.

Aside Programs versus processes

This is a good place to pause and make sure you understand the distinction between a program and a process. A program is a collection of code and data; programs can exist as object files on disk or as segments in an address space. A process is a specific instance of a program in execution; a program always runs in the context of some process. Understanding this distinction is important if you want to understand the `fork` and `execve` functions. The `fork` function runs the same program in a new child process that is a duplicate of the parent. The `execve` function loads and runs a new program in the context of the current process. While it overwrites the address space of the current process, it does *not* create a new process. The new program still has the same PID, and it inherits all of the file descriptors that were open at the time of the call to the `execve` function.

3.5 信号



信号：提供了一种机制，通知用户进程发生了什么异常。

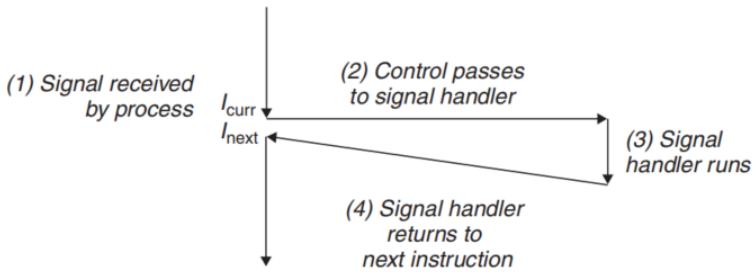
> signal的本质更像是邮件，邮递员扔到你家邮箱，你未必及时收信（异步）。

The transfer of a signal to a destination process occurs in two distinct steps:

Sending a signal. The kernel *sends* (*delivers*) a signal to a destination process by updating some state in the context of the destination process. The signal is delivered for one of two reasons: (1) The kernel has detected a system event such as a divide-by-zero error or the termination of a child process. (2) A process has invoked the `kill` function (discussed in the next section) to explicitly request the kernel to send a signal to the destination process. A process can send a signal to itself.

Receiving a signal. A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal. The process can either ignore the signal, terminate, or *catch* the signal by executing a user-level function called a *signal handler*. Figure 8.27 shows the basic idea of a handler catching a signal.

Figure 8.27
Signal handling. Receipt of a signal triggers a control transfer to a signal handler. After it finishes processing, the handler returns control to the interrupted program.



一个发出而没有被接收的信号叫做待处理信号(pending signal)。在任何时刻，一种类型至多只会有一个待处理信号。如果一个进程有一个类型为 k 的待处理信号，那么任何接下来发送到这个进程的类型为 k 的信号都不会排队等待；它们只是被简单地丢弃。一个进程可以有选择性地阻塞接收某种信号。当一种信号被阻塞时，它仍可以被发送，但是产生的待处理信号不会被接收，直到进程取消对这种信号的阻塞。

一个待处理信号最多只能被接收一次。内核为每个进程在 pending 位向量中维护着待处理信号的集合，而在 blocked 位向量中维护着被阻塞的信号集合。只要传送了一个类型为 k 的信号，内核就会设置 pending 中的第 k 位，而只要接收了一个类型为 k 的信号，内核就会清除 pending 中的第 k 位。

pending bit vector
blocked bit vector.

8.5.2 发送信号

3.5.1 发信号

1. `linux> /bin/kill -9 15213`
2. `ctrl+c`

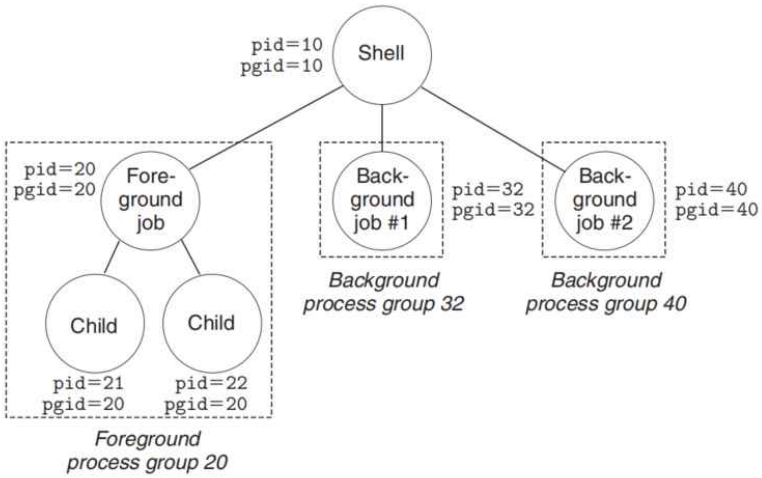
还是举个例子就明白了：

Unix shells use the abstraction of a *job* to represent the processes that are created as a result of evaluating a single command line. At any point in time, there is at most one foreground job and zero or more background jobs. For example, typing

```
linux> ls | sort
```

creates a foreground job consisting of two processes connected by a Unix pipe: one running the `ls` program, the other running the `sort` program. The shell creates a separate process group for each job. Typically, the process group ID is taken from one of the parent processes in the job. For example, Figure 8.28 shows a shell with one foreground job and two background jobs. The parent process in the foreground job has a PID of 20 and a process group ID of 20. The parent process has created two children, each of which are also members of process group 20.

Figure 8.28
Foreground and background process groups.



输入 `ctrl+c` 会导致内核发送一个 `SIGINT` 信号给前台进程组中的每个进程，默认情况下是终止前台作业。（为什么发送给进程组？比如一个进程有除零错误，那么实际上大家整个组都寄了！所以一起死吧！）

3.5.2 收信号

当kernal将进程p从内核模式切换到用户模式时（比如从系统调用返回或者完成了一次context switch），它会检查p的为被阻塞的待处理信号集合。

> **Signal handler是用户态的，是用户程序的一部分**。

> Tip:
>
> 还记得我们前面讲的异常处理程序吗，它是内核态的。这里我们讲的signal handler是用户态的！

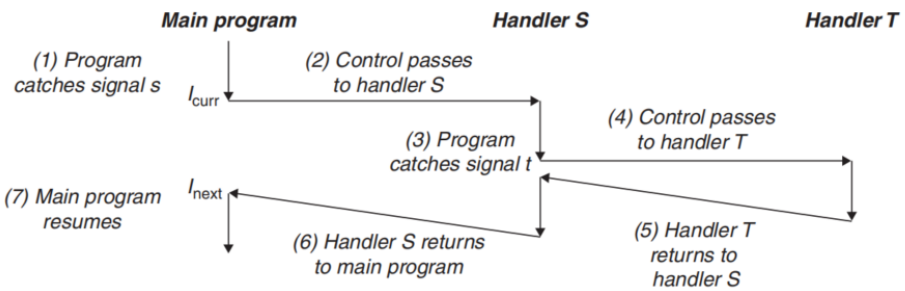


Figure 8.31 **Handlers can be interrupted by other handlers.**

3.5.3 🌶️编写安全的信号处理程序

Signal handlers are tricky because they can run concurrently with the main program and with each other, as we saw in Figure 8.31. If a handler and the main program access the same global data structure concurrently, then the results can be unpredictable and often fatal.

We will explore concurrent programming in detail in Chapter 12. Our aim here is to give you some conservative guidelines for writing handlers that are safe to run concurrently. If you ignore these guidelines, you run the risk of introducing subtle concurrency errors. With such errors, your program works correctly most of the time. However, when it fails, it fails in unpredictable and unrepeatable ways that are horrendously difficult to debug. Forewarned is forearmed!

G0. Keep handlers as simple as possible. The best way to avoid trouble is to keep your handlers as small and simple as possible. For example, the handler might simply set a global flag and return immediately; all processing associated with the receipt of the signal is performed by the main program, which periodically checks (and resets) the flag.

G1. Call only async-signal-safe functions in your handlers. A function that is *async-signal-safe*, or simply *safe*, has the property that it can be safely called from a signal handler, either because it is *reentrant* (e.g., accesses only local variables; see Section 12.7.2), or because it cannot be interrupted by a signal handler. Figure 8.33 lists the system-level functions that Linux guarantees to be safe. Notice that many popular functions, such as `printf`, `sprintf`, `malloc`, and `exit`, are *not* on this list.

The only safe way to generate output from a signal handler is to use the `write` function (see Section 10.1). In particular, calling `printf` or `sprintf` is unsafe. To work around this unfortunate restriction, we have

G3. Protect accesses to shared global data structures by blocking all signals. If a handler shares a global data structure with the main program or with other handlers, then your handlers and main program should temporarily block all signals while accessing (reading or writing) that data structure. The reason for this rule is that accessing a data structure d from the main program typically requires a sequence of instructions. If this instruction sequence is interrupted by a handler that accesses d, then the handler might find d in an inconsistent state, with unpredictable results. Temporarily blocking signals while you access d guarantees that a handler will not interrupt the instruction sequence.

G4. Declare global variables with `volatile`. Consider a handler and `main` routine that share a global variable g. The handler updates g, and `main` periodically reads g. To an optimizing compiler, it would appear that the value of g never changes in `main`, and thus it would be safe to use a copy of g that is cached in a register to satisfy every reference to g. In this case, the `main` function would never see the updated values from the handler.

You can tell the compiler not to cache a variable by declaring it with the `volatile` type qualifier. For example:

```
volatile int g;
```

The `volatile` qualifier forces the compiler to read the value of g from memory each time it is referenced in the code. In general, as with any shared data structure, each access to a global variable should be protected by temporarily blocking signals.

注：

- G4：对于全局变量，告诉编译器，不要放在寄存器里，放在内存里！因为寄存器里的值会被修改！