```
25          while(1) {};
26      }
27      Kill(pid, SIGUSR1);
28      Waitpid(-1, NULL, 0);
29
30      Sigfillset(&mask);
31      Sigprocmask(SIG_BLOCK, &mask, &prev_mask);  /* Block sigs */
32      printf("%ld", ++counter);
33      Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Restore sigs */
34
35      exit(0);
36  }
```

code/ecf/signalprob0.c

3. 可移植的信号处理

Unix 信号处理的另一个缺陷在于不同的系统有不同的信号处理语义。例如：

- signal 函数的语义各有不同。有些老的 Unix 系统在信号 k 被处理程序捕获之后就把对信号 k 的反应恢复到默认值。在这些系统上，每次运行之后，处理程序必须调用 signal 函数，显式地重新设置它自己。
- 系统调用可以被中断。像 read、write 和 accept 这样的系统调用潜在地会阻塞进程一段较长的时间，称为慢速系统调用。在某些较早版本的 Unix 系统中，当处理程序捕获到一个信号时，被中断的慢速系统调用在信号处理程序返回时不再继续，而是立即返回给用户一个错误条件，并将 errno 设置为 EINTR。在这些系统上，程序员必须包括手动重启被中断的系统调用的代码。

要解决这些问题，Posix 标准定义了 sigaction 函数，它允许用户在设置信号处理时，明确指定他们想要的信号处理语义。

```
#include <signal.h>

int sigaction(int signum,  struct sigaction *act,
              struct sigaction *oldact);
```

返回：若成功则为 0，若出错则为 −1。

sigaction 函数运用并不广泛，因为它要求用户设置一个复杂结构的条目。一个更简洁的方式，最初是由 W. Richard Stevens 提出的[110]，就是定义一个包装函数，称为 Signal，它调用 sigaction。图 8-38 给出了 Signal 的定义，它的调用方式与 signal 函数的调用方式一样。

Signal 包装函数设置了一个信号处理程序，其信号处理语义如下：

- 只有这个处理程序当前正在处理的那种类型的信号被阻塞。
- 和所有信号实现一样，信号不会排队等待。
- 只要可能，被中断的系统调用会自动重启。
- 一旦设置了信号处理程序，它就会一直保持，直到 Signal 带着 handler 参数为 SIG_IGN 或者 SIG_DFL 被调用。

我们在所有的代码中实现 Signal 包装函数。

8.5.6　同步流以避免讨厌的并发错误

如何编写读写相同存储位置的并发流程序的问题，困扰着数代计算机科学家。一般而