言，流可能交错的数量与指令的数量呈指数关系。这些交错中的一些会产生正确的结果，而有些则不会。基本的问题是以某种方式同步并发流，从而得到最大的可行的交错的集合，每个可行的交错都能得到正确的结果。

code/src/csapp.c

```
1  handler_t *Signal(int signum, handler_t *handler)
2  {
3      struct sigaction action, old_action;
4
5      action.sa_handler = handler;
6      sigemptyset(&action.sa_mask); /* Block sigs of type being handled */
7      action.sa_flags = SA_RESTART; /* Restart syscalls if possible */
8
9      if (sigaction(signum, &action, &old_action) < 0)
10         unix_error("Signal error");
11     return (old_action.sa_handler);
12 }
```

code/src/csapp.c

图 8-38　Signal：sigaction 的一个包装函数，它提供在 Posix 兼容系统上的可移植的信号处理

并发编程是一个很深且很重要的问题，我们将在第 12 章中更详细地讨论。不过，在本章中学习的有关异常控制流的知识，可以让你感觉一下与并发相关的有趣的智力挑战。例如，考虑图 8-39 中的程序，它总结了一个典型的 Unix shell 的结构。父进程在一个全局作业列表中记录着它的当前子进程，每个作业一个条目。addjob 和 deletejob 函数分别向这个作业列表添加和从中删除作业。

当父进程创建一个新的子进程后，它就把这个子进程添加到作业列表中。当父进程在 SIGCHLD 处理程序中回收一个终止的(僵死)子进程时，它就从作业列表中删除这个子进程。

乍一看，这段代码是对的。不幸的是，可能发生下面这样的事件序列：

1) 父进程执行 fork 函数，内核调度新创建的子进程运行，而不是父进程。

2) 在父进程能够再次运行之前，子进程就终止，并且变成一个僵死进程，使得内核传递一个 SIGCHLD 信号给父进程。

3) 后来，当父进程再次变成可运行但又在它执行之前，内核注意到有未处理的 SIGCHLD 信号，并通过在父进程中运行处理程序接收这个信号。

4) 信号处理程序回收终止的子进程，并调用 deletejob，这个函数什么也不做，因为父进程还没有把该子进程添加到列表中。

5) 在处理程序运行完毕后，内核运行父进程，父进程从 fork 返回，通过调用 addjob 错误地把(不存在的)子进程添加到作业列表中。

因此，对于父进程的 main 程序和信号处理流的某些交错，可能会在 addjob 之前调用 deletejob。这导致作业列表中出现一个不正确的条目，对应于一个不再存在而且永远也不会被删除的作业。另一方面，也有一些交错，事件按照正确的顺序发生。例如，如果在 fork 调用返回时，内核刚好调度父进程而不是子进程运行，那么父进程就会正确地把子进程添加到作业列表中，然后子进程终止，信号处理函数把该作业从列表中删除。

这是一个称为竞争(race)的经典同步错误的示例。在这个情况中，main 函数中调用 addjob 和处理程序中调用 deletejob 之间存在竞争。如果 addjob 赢得进展，那么结果