了3个子进程。同时，父进程等待来自终端的一个输入行，随后处理它。这个处理被模型化为一个无限循环。当每个子进程终止时，内核通过发送一个SIGCHLD信号通知父进程。父进程捕获这个SIGCHLD信号，回收一个子进程，做一些其他的清理工作(模型化为sleep语句)，然后返回。

图8-36中的signal1程序看起来相当简单。然而，当在Linux系统上运行它时，我们得到如下输出：

```
linux> ./signal1
Hello from child 14073
Hello from child 14074
Hello from child 14075
Handler reaped child
Handler reaped child
CR
Parent processing input
```

从输出中我们注意到，尽管发送了3个SIGCHLD信号给父进程，但是其中只有两个信号被接收了，因此父进程只是回收了两个子进程。如果挂起父进程，我们看到，实际上子进程14075没有被回收，它成了一个僵死进程(在ps命令的输出中由字符串“defunct”表明)：

```
Ctrl+Z
Suspended
linux> ps t
  PID TTY      STAT   TIME COMMAND
    .
    .
    .
14072 pts/3    T      0:02 ./signal1
14075 pts/3    Z      0:00 [signal1] <defunct>
14076 pts/3    R+     0:00 ps t
```

哪里出错了呢？问题就在于我们的代码没有解决信号不会排队等待这样的情况。所发生的情况是：父进程接收并捕获了第一个信号。当处理程序还在处理第一个信号时，第二个信号就传送并添加到了待处理信号集合里。然而，因为SIGCHLD信号被SIGCHLD处理程序阻塞了，所以第二个信号就不会被接收。此后不久，就在处理程序还在处理第一个信号时，第三个信号到达了。因为已经有了一个待处理的SIGCHLD，第三个SIGCHLD信号会被丢弃。一段时间之后，处理程序返回，内核注意到有一个待处理的SIGCHLD信号，就迫使父进程接收这个信号。父进程捕获这个信号，并第二次执行处理程序。在处理程序完成对第二个信号的处理之后，已经没有待处理的SIGCHLD信号了，而且也绝不会再有，因为第三个SIGCHLD的所有信息都已经丢失了。*由此得到的重要教训是，不可以用信号来对其他进程中发生的事件计数。*

为了修正这个问题，我们必须回想一下，存在一个待处理的信号只是暗示自进程最后一次收到一个信号以来，至少已经有一个这种类型的信号被发送了。所以我们必须修改SIGCHLD的处理程序，使得每次SIGCHLD处理程序被调用时，回收尽可能多的僵死子进程。图8-37展示了修改后的SIGCHLD处理程序。

当我们在Linux系统上运行signal2时，它现在可以正确地回收所有的僵死子进程了：

```
linux> ./signal2
Hello from child 15237
```