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

图 8-18　使用 waitpid 函数不按照特定的顺序回收僵死子进程

当回收了所有的子进程之后，再调用 waitpid 就返回 −1，并且设置 errno 为 ECHILD。第 24 行检查 waitpid 函数是正常终止的，否则就输出一个错误消息。在我们的 Linux 系统上运行这个程序时，它产生如下输出：

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

注意，程序不会按照特定的顺序回收子进程。子进程回收的顺序是这台特定的计算机系统的属性。在另一个系统上，甚至在同一个系统上再执行一次，两个子进程都可能以相反的顺序被回收。这是*非确定性*行为的一个示例，这种非确定性行为使得对并发进行推理非常困难。两种可能的结果都同样是正确的，作为一个程序员，你绝不可以假设总是会出现某一个结果，无论多么不可能出现另一个结果。唯一正确的假设是每一个可能的结果都同样可能出现。

图 8-19 展示了一个简单的改变，它消除了这种不确定性，按照父进程创建子进程的相同顺序来回收这些子进程。在第 11 行中，父进程按照顺序存储了它的子进程的 PID，然后通过用适当的 PID 作为第一个参数来调用 waitpid，按照同样的顺序来等待每个子进程。