当这段代码正确执行的时候，循环在浪费处理器资源。我们可能会想要修补这个问题，在循环体内插入 pause：

```
while (!pid)  /* Race! */
    pause();
```

注意，我们仍然需要一个循环，因为收到一个或多个 SIGINT 信号，pause 会被中断。不过，这段代码有很严重的竞争条件：如果在 while 测试后和 pause 之前收到 SIGCHLD 信号，pause 会永远睡眠。

另一个选择是用 sleep 替换 pause：

```
while (!pid) /* Too slow! */
    sleep(1);
```

当这段代码正确执行时，它太慢了。如果在 while 之后 pause 之前收到信号，程序必须等相当长的一段时间才会再次检查循环的终止条件。使用像 nanosleep 这样更高精度的休眠函数也是不可接受的，因为没有很好的方法来确定休眠的间隔。间隔太小，循环会太浪费。间隔太大，程序又会太慢。

合适的解决方法是使用 sigsuspend。

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

返回：−1。

sigsuspend 函数暂时用 mask 替换当前的阻塞集合，然后挂起该进程，直到收到一个信号，其行为要么是运行一个处理程序，要么是终止该进程。如果它的行为是终止，那么该进程不从 sigsuspend 返回就直接终止。如果它的行为是运行一个处理程序，那么 sigsuspend 从处理程序返回，恢复调用 sigsuspend 时原有的阻塞集合。

sigsuspend 函数等价于下述代码的原子的(不可中断的)版本：

```
1        sigprocmask(SIG_SETMASK, &mask, &prev);
2        pause();
3        sigprocmask(SIG_SETMASK, &prev, NULL);
```

原子属性保证对 sigprocmask(第 1 行)和 pause(第 2 行)的调用总是一起发生的，不会被中断。这样就消除了潜在的竞争，即在调用 sigprocmask 之后但在调用 pause 之前收到了一个信号。

图 8-42 展示了如何使用 sigsuspend 来替代图 8-41 中的循环。在每次调用 sigsuspend 之前，都要阻塞 SIGCHLD。sigsuspend 会暂时取消阻塞 SIGCHLD，然后休眠，直到父进程捕获信号。在返回之前，它会恢复原始的阻塞集合，又再次阻塞 SIGCHLD。如果父进程捕获一个 SIGINT 信号，那么循环测试成功，下一次迭代又再次调用 sigsuspend。如果父进程捕获一个 SIGCHLD，那么循环测试失败，会退出循环。此时，SIGCHLD 是被阻塞的，所以我们可以可选地取消阻塞 SIGCHLD。在真实的有后台作业需要回收的 shell 中这样做可能会有用处。

sigsuspend 版本比起原来的循环版本不那么浪费，避免了引入 pause 带来的竞争，又比 sleep 更有效率。