

进程控制

进程控制是操作系统对进程进行管理所提供的控制操作。进程控制至少应该包括进程创建、进程撤销、进程睡眠、进程唤醒、进程执行等操作，它们都使用原语实现。所谓原语是指在执行过程中不允许中断，它属于操作系统内核的一部分，以系统调用的形式提供给用户和操作系统使用。



Linux公社（LinuxIDC.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

LinuxIDC.com提供包括Ubuntu，Fedora，SUSE技术，以及最新IT资讯等Linux专业类网站。

并被收录到Google 网页目录-计算机 > 软件 > 操作系统 > Linux 目录下。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

包括：

[Ubuntu专题](#)

[Fedora专题](#)

[RedHat专题](#)

[SUSE专题](#)

[红旗Linux专题](#)

[Android专题](#)

[Linux公社简介](#) - [广告服务](#) - [网站地图](#) - [帮助信息](#) - [联系我们](#)

本站（LinuxIDC）所刊载文章不代表同意其说法或描述，仅为提供更多信息，也不构成任何建议。

本站带宽由[\[6688.CC\]](#)友情提供

Copyright © 2006-2011 [Linux公社](#) All rights reserved

Linux公社

www.Linuxidc.com

❖ www.Linuxidc.com

进程创建

1. 进程的创建与调度执行

不同的操作系统所提供的进程创建原语的名称和格式不尽相同，但执行创建进程原语后，操作系统所做的工作却大致相同，都包括以下几点：

- 给新创建的进程分配一个内部标识，并分配一个空白的**PCB**，同时在系统进程表中增加一个表目。
- 为该进程分配内存空间，包括进程映像所需要的所有元素（程序、数据、用户栈等），复制父进程内存空间的内容到该进程内存空间中。
- 初始化**PCB**。如果该进程是用户进程创建的子进程，则它将继承和共享父进程的资源。
- 置该进程的状态为就绪，插入就绪队列。
- 生成其它相关的数据结构。

Linux中终端用户进程的创建与调度执行

启动Linux的过程中，当操作系统被加载后首先创建0#进程init_task。除了0#进程是在系统初启时由系统创建以外，所有进程均由其它进程调用fork()创建。如0#进程调用fork()创建1#进程，再由1#进程使用fork()为每个可用于用户登录的通信端口创建一个进程为用户服务，如等待用户登录、执行shell命令解释程序等等。

当用户开始使用某端口时，系统进程创建一个login进程来接收用户标识和口令，并通过系统文件/etc/passwd中的信息核对该用户的身份。如果登录成功则login改变当前目录到用户主目录，并执行指定的shell程序，以使用户通过shell界面直接与login系统进程交互。

当用户在键盘上键入一个用户可执行文件名时，操作系统就为该文件创建一个相应的用户进程并投入运行。

Linux中用户子进程的创建与调度执行

用户可以在自己的进程中创建多个子进程以实现多个不同任务的并发执行。Linux提供的创建子进程的系统调用是 `fork()`。

格式: `int fork()`

返回值: `=0` 创建成功，从子进程返回；

`>0` 创建成功，从父进程返回，其值为子进程的PID号；

`=-1` 创建失败。

Linux中用户子进程的创建与调度执行

子进程创建时操作系统做以下工作：

- 检查同时运行的进程数目，若超过系统设定值则创建失败，返回-1；
- 为子进程分配进程控制块task_struct结构，并赋予唯一进程标识符pid；
- 子进程继承父进程打开的所有文件及资源，对父进程的当前目录和所有已打开系统文件表项中的引用记数加1；
- 为子进程创建进程映像：
 - ❧ 创建子进程映像静态部分：复制父进程映像静态部分
 - ❧ 创建子进程映像动态部分：初始化task_struct结构
 - ❧ 结束创建，置子进程为内存就绪状态，插入就绪队列，作为一个独立的进程被系统调度。
- 若调用进程（父进程）返回，则返回创建的子进程标识符pid值（此时返回值>0）；
- 若子进程被调度执行，则将其U区计时字段初始化然后返回（此时返回值=0）。

由于fork（）调用执行后，从父进程和子进程返回的值不同，因而用户能够以此为据在程序中使用分支结构将父子进程需要执行的不同程序分开。

应用程序的框架结构如下：

```
main( )
{
    int  p;                //存放子进程pid号
    while( ( p=fork( ) ) == -1); //创建子进程直到成功为止

    if (p == 0)            //返回值=0表示子进程返回
    {
        /*此处插入子进程程序段*/
    }
    else                   //返回值>0表示父进程返回
    {
        /*此处插入父进程程序段*/
    }
}
```


- 子进程创建后，子进程的映像复制了父进程的数据与堆栈空间，并继承父进程的用户代码、组代码、环境变量、已打开的文件数组、工作目录以及资源限制等等，这些继承是通过复制得来的，所以子进程映像与父进程映像是存储在两个不同的地址空间中内容相同的程序副本，这就意味着父进程和子进程在各自的存储空间上运行着内容相同的程序。因此，一个程序中如果使用了`fork()`，那么当程序运行后，该程序就会在两个进程实体中出现，就会因两个进程的调度而被执行两次。
- 也正因为父子映像有各自的存储空间，父子进程对于各自存储空间中的执行过程包括对变量的修改等等，就是各自的行为，不会传递到对方的存储空间中，因而双方都感知不到对方的行为。
- 操作系统对于父子进程的调度执行具有随机性，它们执行的先后次序不受程序源码中分支顺序的影响。只要父子进程之间没有使用同步工具来控制其执行序列，则父子进程并发执行的顺序取决于操作系统的调度，谁先谁后是随机的。

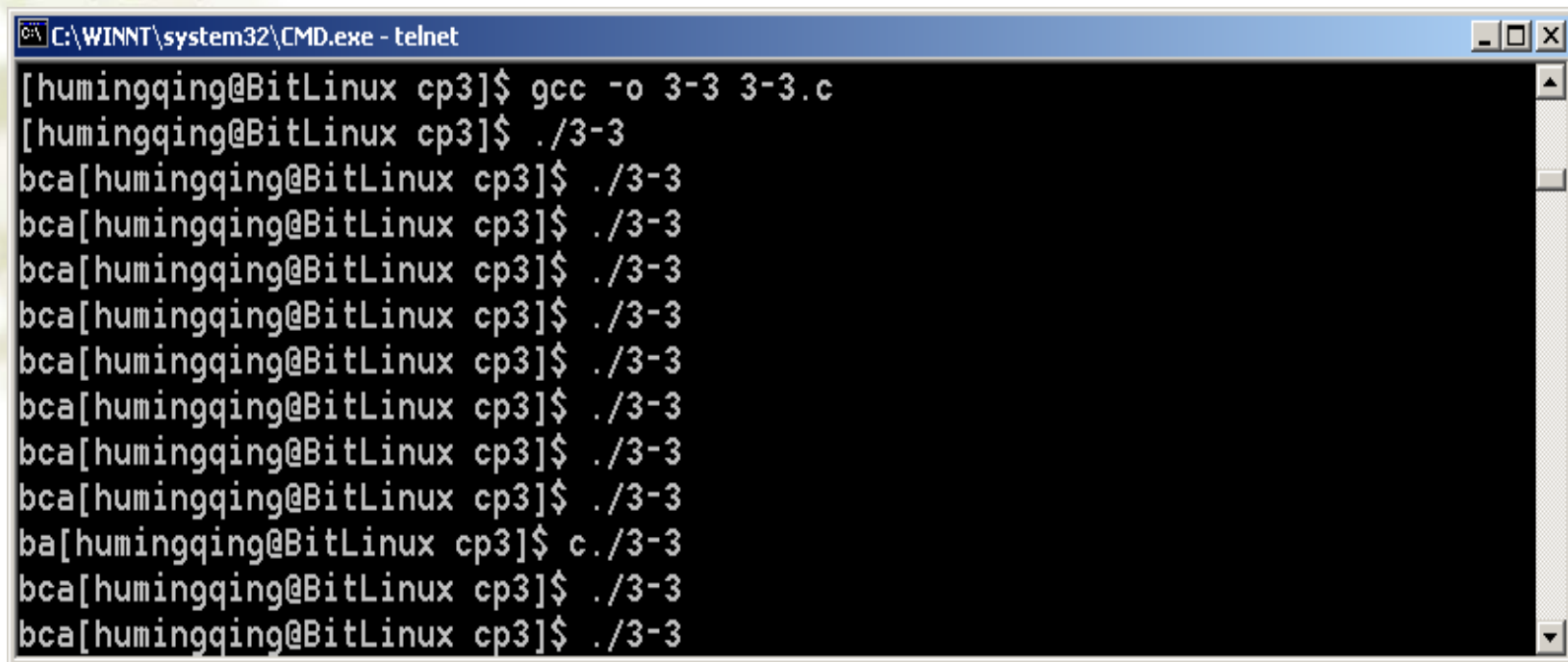
[例3-3] 父进程创建子进程P1、P2，父子进程分别输出字符a、b和c。假设文件名为3-3.c

```
#include<stdio.h>
main()
{
    int p1,p2;
    while ( (p1 = fork() ) == -1);    //创建子进程1，直至创建成功

    if(p1==0)                        //子进程P1返回输出'b'
        putchar('b') ;
    else                             //父进程返回
    {
        while( (p2 = fork()) == -1) ;    //创建子进程2

        if(p2==0)                      //子进程P2返回输出'c'
            putchar('c');
        else
            putchar('a');                //父进程返回输出'a'
    }
}
```

2011/07/08



```
C:\WINNT\system32\CMD.exe - telnet
[humingqing@BitLinux cp3]$ gcc -o 3-3 3-3.c
[humingqing@BitLinux cp3]$ ./3-3
bca[humingqing@BitLinux cp3]$ ./3-3
bca[humingqing@BitLinux cp3]$ ./3-3
bca[humingqing@BitLinux cp3]$ ./3-3
bca[humingqing@BitLinux cp3]$ ./3-3
bca[humingqing@BitLinux cp3]$ ./3-3
bca[humingqing@BitLinux cp3]$ ./3-3
bca[humingqing@BitLinux cp3]$ ./3-3
bca[humingqing@BitLinux cp3]$ ./3-3
ba[humingqing@BitLinux cp3]$ c./3-3
bca[humingqing@BitLinux cp3]$ ./3-3
bca[humingqing@BitLinux cp3]$ ./3-3
```

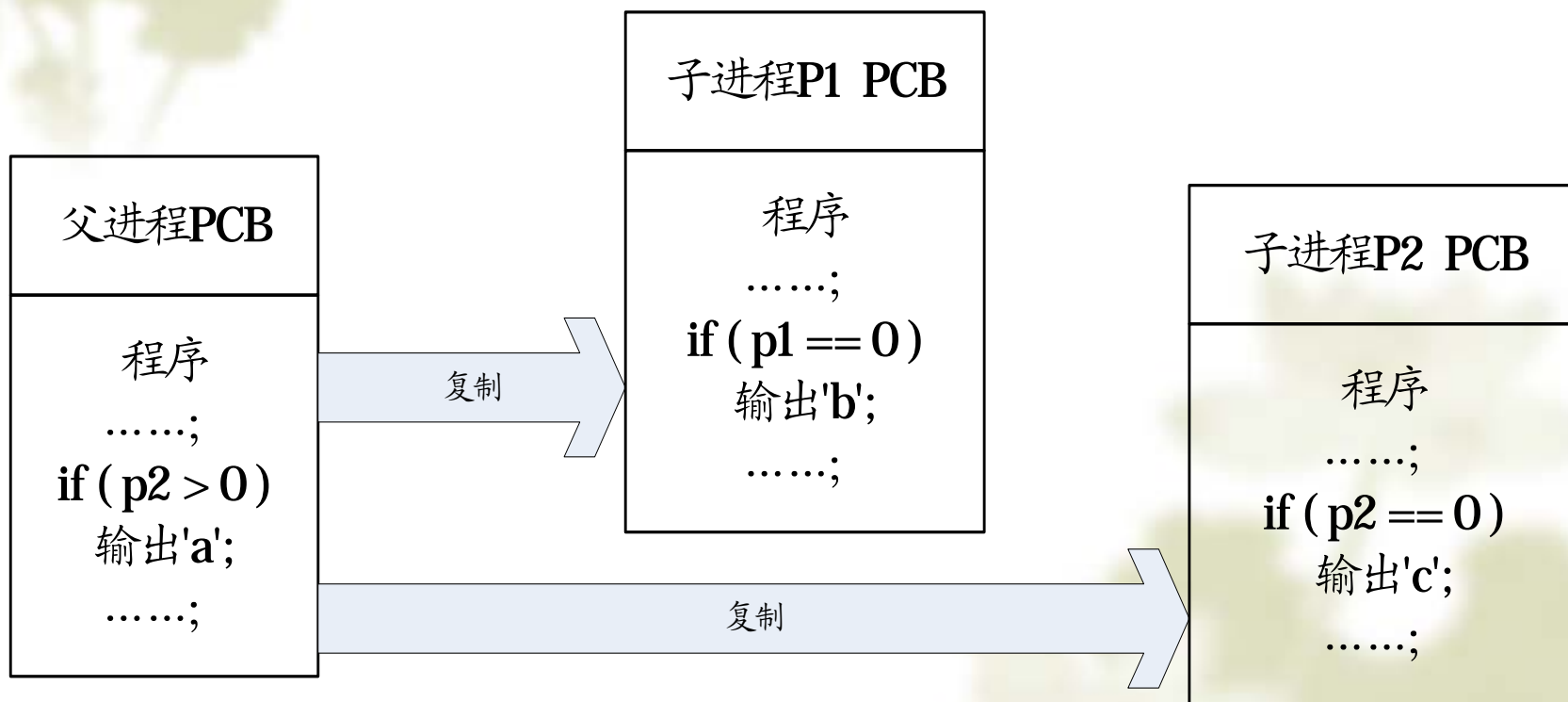
该程序多次运行后，输出的结果可能会是：

abc、acb、bca、cba、bac或cab等随机结果中的任意一种。

每次运行后都会产生父子3个进程，所以有3个字符输出。

其中系统屏幕输出进程的输出[humingqing@BitLinux chap3]\$显示会跟随在父进程输出'a'之后。

该例中，父子进程映像的组成参见下图：



Linux中父子进程对程序的共享部分和私有部分

由于父子进程通过复制共享同一个程序，该程序中哪些是父子共享的，哪些是私有的呢？

(1). 父进程创建子进程后，父子进程各自分支中的程序各自私有，其余部分，包括创建前和分支结束后的程序段，均为父子进程共享。

请参考下页的例[3-4]

[例3-4] 设文件名为3-4.c

```
#include<stdio.h>
```

```
main()
```

```
{
```

```
    int p1;
```

```
    putchar('x');           //父子共享部分，都要输出'x'
```

```
    while((p1=fork())!=-1);
```

```
    if(p1==0)
```

```
        putchar('b');       //子进程输出'b'
```

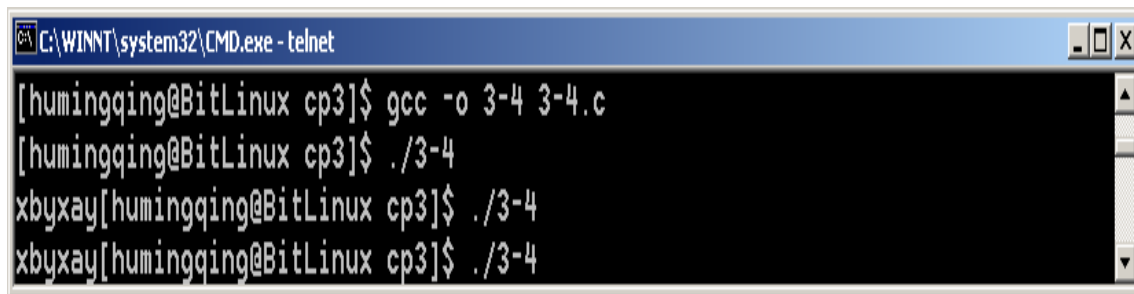
```
    else
```

```
        putchar('a');       //父进程输出'a'
```

```
    putchar('y');           //父子共享部分，都要输出'y'
```

```
}
```

运行后输出的结果可能是xayxby或xbyxay之中的任意一个。



```
C:\WINNT\system32\CMD.exe - telnet
[humingqing@BitLinux cp3]$ gcc -o 3-4 3-4.c
[humingqing@BitLinux cp3]$ ./3-4
xbyxay[humingqing@BitLinux cp3]$ ./3-4
xbyxay[humingqing@BitLinux cp3]$ ./3-4
```


(2). 如果子进程在其分支结束处使用了进程终止`exit()`系统调用而终止执行, 则不会再共享分支结束后的程序段。

```
#include<stdio.h>
```

```
main()
```

```
{
```

```
    int p1;
```

```
    putchar('x'); //父子共享部分, 都要输出'x'
```

```
    while((p1=fork())!=-1);
```

```
    if(p1==0)
```

```
    {
```

```
        putchar('b'); //子进程输出'b'后终止执行  
        exit(0);
```

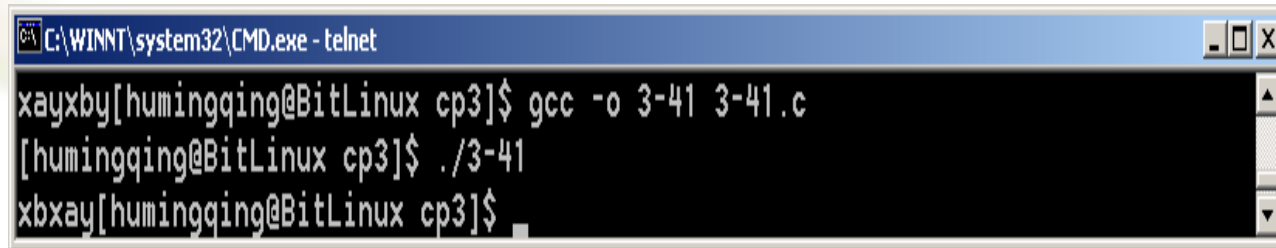
```
    }
```

```
    else
```

```
        putchar('a');
```

```
    putchar('y');          //只有父进程输出'y'
```

```
}
```



```
C:\WINNT\system32\CMD.exe - telnet  
xayxby[humingqing@BitLinux cp3]$ gcc -o 3-41 3-41.c  
[humingqing@BitLinux cp3]$ ./3-41  
xbxay[humingqing@BitLinux cp3]$
```

可以看出, 父子进程对于创建前的部分是共享的, 而分支结束后的部分是否共享取决于子进程是否已经终止。

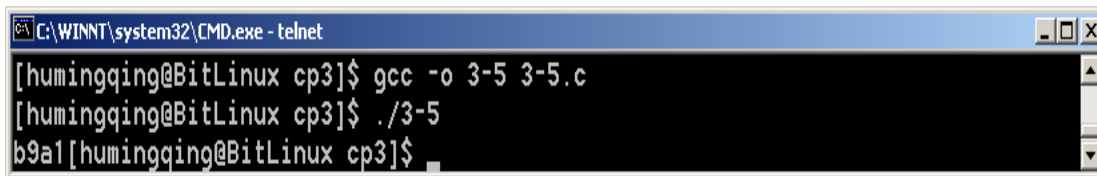
(3). 对于父子进程用if条件语句分开的各自程序段中的程序是不共享的, 无论父进程还是子进程, 它们具有各自不同的进程映像, 在自己的私有存储空间中对数据进行修改, 不会影响到对方。

[例3-5] 在子进程中对变量重新赋值, 观察在父进程中是否感知到值。设文件名为3-5.c

```
#include<stdio.h>
```

```
main()
```

```
{  
    int p1;  
    int x=1;    //x的初值父子进程共享  
    while((p1=fork())!=-1);  
  
    if(p1==0)  
    {  
        putchar('b');    //子进程提示符  
        x=9;    //子进程中对x重新赋值  
        printf("%d",x); //子进程中输出x  
    }  
    else  
    {  
        putchar('a');    //父进程提示符  
        printf("%d",x); //父进程中输出x  
    }  
}
```



```
C:\WINNT\system32\CMD.exe - telnet  
[humingqing@BitLinux cp3]$ gcc -o 3-5 3-5.c  
[humingqing@BitLinux cp3]$ ./3-5  
b9a1[humingqing@BitLinux cp3]$
```

运行后输出的结果是: a1b9或b9a1

2011/07/08

进程撤销

1. 进程的撤销

当一个进程结束时可以调用进程撤销原语自我撤销（终止）。
进程撤销原语执行时，操作系统完成以下任务：

- 找到该进程，将其所占有的资源归还给系统，将其生命周期中所占有的**CPU**时间累计到其父进程**PCB**中。
- 将该进程从系统进程表中删除，释放该进程的**PCB**。
- 转进程调度。因为该进程被终止执行，**CPU**被释放，所以要转进程调度来实施**CPU**的再分配。

进程一旦撤销就不可能再转换为其它任何进程状态了，该进程的生命周期就此消亡。所以，进程撤销的系统调用是不返回函数。

进程撤销

2. Linux中进程的终止

Linux中进程使用系统调用**exit(status)**来终止自己的执行，其中**status**是子进程向父进程发送的终止信息，父进程使用**wait()**系统调用来接收这个信息。

格式: `void exit (int status)`

功能:

- 将进程置僵死状态
- 释放其所占有的资源
- 向父进程发本进程死信号，并发送信息**status**给父进程，将自己及自己的子进程运行**CPU**的时间总和留待父进程使用**wait()**回收。

返回值: 进程终止**exit()**是个不返回函数，所以没有返回值。

所用头文件: `#include <stdlib. h>`

进程撤销

子进程终止时操作系统做以下工作：

- **关闭软中断**: 因为进程即将终止而不再处理任何软中断信号；
- **回收资源**: 关闭所有已打开文件，释放进程所有的区及相应内存，释放当前目录及修改根目录的索引节点；
- **写记帐信息**: 将进程在运行过程中所产生的记帐数据（其中包括进程运行时的各种统计信息）记录到一个全局记帐文件中；
- **置该进程为僵死状态**: 向父进程发送子进程死的软中断信号，将终止信息**status**送到指定的存储单元中；
- **转进程调度**: 因为此时**CPU**已经被释放，需要由进程调度进行**CPU**再分配。

进程睡眠

1. 进程的睡眠

当一个进程因为某种执行条件暂不满足或者等待某个事件的发生时，它必须放弃**CPU**，等待条件满足后才能继续执行。此时，进程将自动放弃**CPU**，进入睡眠状态。

以下皆是引起进程睡眠的事件：

- 进程请求I/O；
- 进程请求系统资源时需要排队或暂时得不到满足；
- 进程请求的同步或互斥信号没有满足或等待的同步消息没有到来；
- 进程请求延时。

进程睡眠

操作系统提供一系列系统调用函数，提供进程针对不同的请求加以调用。

当进程使用了其中的一个函数调用后，其申请的条件得不到满足或者请求的服务是I/O时，该进程就会自动进入睡眠状态。而进程一旦进入睡眠状态，则只能由其它进程将其唤醒。

假设引起睡眠的原因为chan，进程睡眠原语执行后操作系统要作如下工作：

- (1). 将当前需要睡眠的进程运行现场保护到其PCB中；
- (2). 置进程的状态为阻塞态；
- (3). 将该进程插入引起睡眠的chan的等待队列；
- (4). 转进程调度。因为此时CPU空闲，需激活调度进程以调度另一个进程投入运行。

进程睡眠

2. Linux中进程的睡眠

Linux提供多种可能导致进程睡眠或等待的系统调用，本章主要介绍父进程等待子进程终止的系统调用**wait()**和进程延时系统调用**sleep()**。**wait()**的使用格式在3.5.2节中介绍。此处介绍进程延时**sleep()**。

格式: **sleep** (n)

其中: n表示延时的秒数。

功能: 进程睡眠n秒。

进程唤醒

进程一旦由运行态转变为睡眠态后就被插入相应的等待队列。当其等待的事件发生时，需要由完成事件的进程执行进程唤醒原语将其唤醒。

执行唤醒原语，操作系统需要做：

- 检查该事件的等待队列上是否有等待进程；
- 如果有等待进程，则将它们从该等待队列中移出，修改其状态为就绪态，插入就绪队列；
- 转进程调度或继续执行。

唤醒进程的操作因涉及进程阻塞队列的操作，需要在核态下完成，它往往包含在操作系统所提供的同步机制中，而不会单独提供给用户使用，所以本章没有给出具体的例证说明。详情参阅4.4和4.7节。