第9章　进 程 关 系

9.1　引言

在上一章我们已了解到进程之间具有关系。首先，每个进程有一个父进程。当子进程终止时，父进程会得到通知并能取得子进程的退出状态。在 8.6节说明waitpid函数时，我们也提到了进程组，以及如何等待进程组中的任意一个进程终止。

本章将更详细地说明进程组以及 POSIX.1引进的对话期新概念。还将介绍登录 shell（登录时所调用的）和所有从登录shell起动的进程之间的关系。

在说明这些关系时不可能不谈及信号，而谈论信号又需要很多本章介绍的概念。如果你不熟悉UNIX信号，则可能先要浏览一下第10章。

9.2　终端登录

先看一看登录到UNIX系统时所执行的各个程序。在早期的 UNIX系统中，例如V7，用户用哑终端(通过RS-232连到主机)进行登录。终端或者是本地的（直接连接）或者是远程的（通过调制解调器连接）。在这两种情况下，登录都经由内核中的终端设备驱动程序。例如，在PDP-11上常用的设备是DH-11和DZ-11。因为连到主机上的终端设备数已经确定，所以同时的登录数也就有了已知的上限。下面说明的登录过程适用于使用一个 RS-232终端登录到UNIX系统中。

9.2.1　4.3+BSD终端登录

登录过程在过去15年中并没有多少改变。系统管理者创建一个通常名为 /etc/ttys的文件，其中，每个终端设备有一行，每一行说明设备名和传到 getty程序的参数，这些参数说明了终端的波特率等。当系统自举时，内核创建进程 ID 1，也就是init进程。init进程使系统进入多用户状态。init读文件/etc/ttys,对每一个允许登录的终端设备， init调用一次fork，它所生成的子进程则执行程序getty。这种情况示于图9-1中。

图9-1中各个进程的实际用户 ID和有效用户 ID都是0(也就是它们都具有超级用户特权)。init以空环境执行getty程序。

getty对终端设备调用open函数，以读、写方式将终端打开。如果设备是调制解调器，则open可能会在设备驱动程序中滞留，直到用户拨号调制解调器，并且线路被接通。一旦设备被打开，则文件描述符 0、1、2就被设置到该设备。然后getty输出“ login：”之类的信息，并等待用户键入用户名。如果终端支持多种速度，则getty可以测试特殊字符以便适当地更改终端速度 (波特率)。关于getty程序以及有关数据文件的细节，请参阅

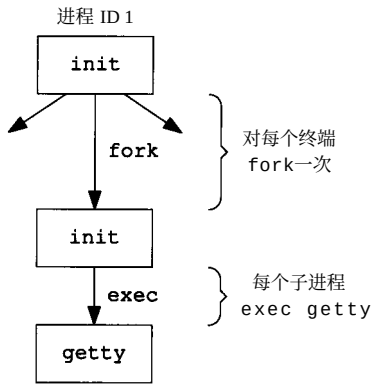

图9-1　init生成进程使终端可用于登录

UNIX手册。

当用户键入了用户名后，getty就完成了。然后它以类似于下列的方式调用login程序：

```
execle("/usr/bin/login", "login", "-p", username, (char *) 0, envp);
```

（在gettytab文件中可能会有一些选择项使其调用其他程序，但系统默认是 login程序）。init以一个空环境调用getty。getty以终端名（例如TERM=foo, 其中终端foo的类型取自gettytab文件）和在gettytab中的环境字符串为login创建一个环境（envp参数）。 -p标志通知login保留传给它的环境，也可将其他环境字符串加到该环境中，但是不要替换它。图 9-2显示了login刚被调用后这些进程的状态。

因为最初的init进程具有超级用户优先权，所以图 9-2中的所有进程都有超级用户优先权。图9-2中底部三个进程的进程ID相同，因为进程ID不会因执行exec而改变。并且，除了最初的init进程，所有的进程均有一个父进程ID。

login 能处理多项工作。因为它得到了用户名，所以能调用 getpwnam 取得相应用户的口令文件登录项。然后调用 getpass（3）以显示提示“ Password：”接着读用户键入的口令（自然，禁止回送用户键入的口令）。它调用crypt（3）将用户键入的口令加密，并与该用户口令文件中登录项的 pw_passwd字段相比较。如果用户几次键入的口令都无效，则 login 以参数1调用exit 表示登录过程失败。父进程（ init）了解到子进程的终止情况后，将再次调用fork，其后又跟随着执行 getty，对此终端重复上述过程。

如果用户正确登录， login就将当前工作目录更改为该用户的起始目录 (chdir)。它也调用chown改变该终端的所有权，使该用户成为所有者和组所有者。将对该终端设备的存取许可权改变成：用户读、写和组写。调用setgid及initgroups设置进程的组ID。然后用login所得到的所有信息初始化环境：起始目录(HOME)、shell(SHELL)、用户名(USER和LOGNAME)，以及一个系统默认路径(PATH)。最后，login进程改变为登录用户的用户 ID(setuid)并调用该用户的登录shell，其方式类似于：

```
execl("/bin/sh", "-sh", (char *) 0);
```

argv[0]的第一个字符－是一个标志，表示该shell被调用为登录shell。shell可以查看此字符，并相应地修改其起动过程。

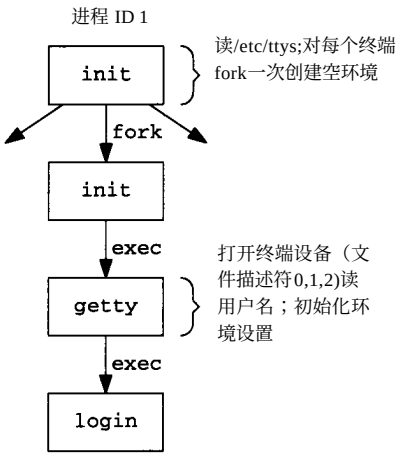

图9-2 login刚被调用后各进程的状态

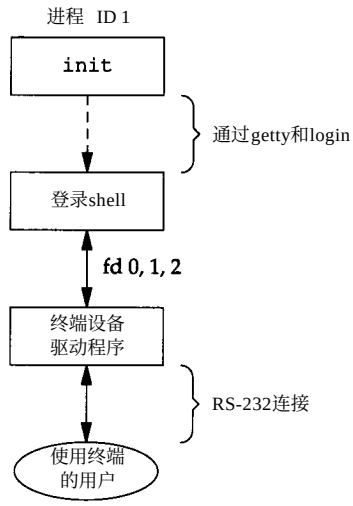

图9-3 终端登录结束后的有关进位

login所做的比上面说的要多。它可选地打印message-of-the-day文件，检查新邮件以及其他一些功能。但是考虑到本书的内容，我们主要关心上面所说的功能。

回忆在8.10节中对setuid函数的讨论，因为setuid是由超级用户调用的，它更改所有三个用户ID：实际、有效和保存的用户ID。login在较早时间调用的setgid对所有三个组ID也有同样效果。

到此为止，登录用户的登录shell开始运行。其父进程ID是init进程ID (进程ID 1)，所以当此登录shell终止时，init会得到通知(接到SIGCHLD信号)，它会对该终端重复全部上述过程。登录shell的文件描述符0，1和2设置为终端设备。图9-3显示了这种安排。

现在，登录shell读其起动文件(Bourne shell和KornShell是.profile, C shell是.cshrc和.login)。这些起动文件通常改变某些环境变量，加上一些环境变量。例如，很多用户设置他们自己的PATH，常常提示实际终端类型(TERM)。当执行完起动文件后，用户最后得到shell的提示符，并能键入命令。

9.2.2 SVR4终端登录

SVR4支持两种形成的终端登录：(a)getty方式，这与上面对4.3+BSD所说明的一样，(b)ttymon登录，这是SVR4的一种新功能。通常，getty用于控制台，ttymon则用于其他终端的登录。

ttymon是名为服务存取设施（Service Access Facility, SAF）的一部分。按照本书的目的，我们只简单说明从init到登录shell之间工作过程，最后结果与图9-3中所示相似。init是sac（服务存取控制器）的父进程，sac调用fork，然后其子进程执行ttymon程序，此时系统进入多用户状态。ttymon监视列于配置文件中的所有终端端口，当用户键入登录名时，它调用一次fork。在此之后该子进程又执行登录用户的登录shell，于是到达了图9-3中所示的位置。一个区别是登录shell的父进程现在是ttymon，而在getty登录中，登录shell的父进程是init。

9.3 网络登录

9.3.1 4.3＋BSD网络登录

在上节所述的终端登录中,init知道哪些终端设备可用来进行登录，并为每个设备生成一个getty进程。但是，对网络登录则情况有所不同，所有登录都经由内核的网络界面驱动程序（例如：以太网驱动程序），事先并不知道将会有多少这样的登录。不是使一个进程等待每一个可能的登录，而是必须等待一个网络连接请求的到达。在4.3+BSD中，有一个称为inetd的进程(有时称为Internet superserver)，它等待大多数网络连接。本书将说明4.3+BSD的网络登录中所涉及的进程序列。关于这些进程的网络程序设计方面的细节请参阅Stevens〔1990〕。

作为系统起动的一部分，init调用一个shell，使其执行shell脚本etc/rc。由此shell脚本起动一个精灵进程inetd。一旦此shell脚本终止，inetd的父进程就变成init。inetd等待TCP/IP连接请求到达主机，而当一个连接请求到达时，它执行一次fork，然后该子进程执行适当的程序。

我们假定到达了一个对于TELNET服务器的TCP连接请求。TELNET是使用TCP协议的远程登录应用程序。在另一个主机(它通过某种形式的网络，连接到服务器主机上)上的用户，或在同一个主机上的一个用户籍起动TELNET客户进程(client)起动登录过程：

```
telnet hostname
```

该客户进程打开一个到名为*hostname*的主机的TCP连接，在*hostname*主机上起动的程序被称为

TELNET服务器。然后，客户进程和服务器进程之间使用 TELNET应用协议通过TCP连接交换数据。所发生的是起动客户进程的用户现在登录到了服务器进程所在的主机。（自然，用户需要在服务器进程主机上有一个有效的账号）。图9-4显示了在执行 TELNET服务器进程 (称为telnetd)中所涉及的进程序列。

然后，telnetd进程打开一个伪终端设备，并用 fork生成一个子进程（第 19章将详细说明伪终端）。父进程处理通过网络连接的通信，子进程则执行 login程序。父、子进程通过伪终端相连接。在调用exec之前，子进程使其文件描述符0,1,2与伪终端相连。如果登录正确，login就执行9.2节中所述的同样步骤 —— 更改当前工作目录为起始目录，设置登录用户的组 ID和用户ID，以及登录用户的初始环境。然后 login用exec将其自身替换为登录用户的登录 shell。图9-5显示了到达这一点时的进程安排。

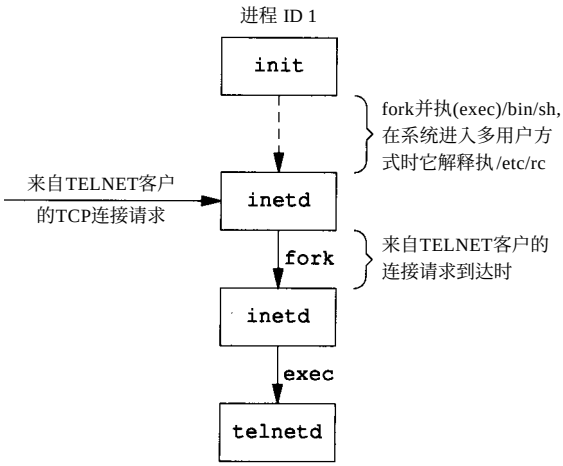

图9-4 执行TELNET服务进程中涉及的进程序列

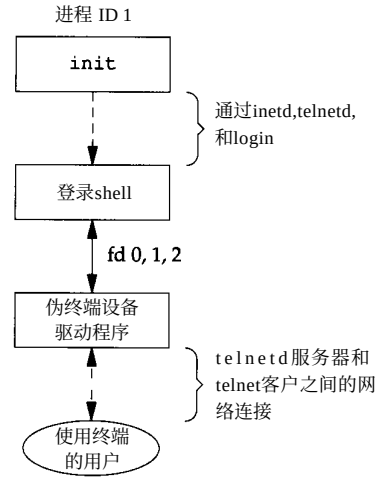

图9-5 为网络登录设置了fd 0, 1, 2后的进程安排

很明显，在伪终端设备驱动程序和终端实际用户之间有很多事情在进行着。第 19章详细说明伪终端时，我们会介绍与这种安排相关的所有进程。

需要理解的重点是：当通过终端（见图 9-3）或网络（见图9-5）登录时，我们得到一个登录shell，其标准输入、输出和标准出错连接到一个终端设备或者伪终端设备上。在下一节中我们会了解到这一登录 shell是一个POSIX.1对话期的开始，而此终端或伪终端则是会话期的控制终端。

9.3.2 SVR4网络登录

SVR4中网络登录的情况与 4.3+BSD中的几乎一样。同样使用了 inetd服务器进程，但是在SVR4中inetd是作为一种服务由服务存取控制器 sac调用的，其父进程不是 init。最后得到的结果与图9-5中一样。

9.4 进程组

每个进程除了有一进程ID之外，还属于一个进程组，第10章讨论信号时还会涉及进程组。

进程组是一个或多个进程的集合。每个进程组有一个唯一的进程组 ID。进程组ID类似于进程ID——它是一个正整数，并可存放在 pid_t数据类型中。函数 getpgrp返回调用进程的进程

组ID。

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpgrp(void);
```

返回：调用进程的进程组ID

> 在很多伯克利的系统中，包括4.3+BSD，这一函数的参数是*pid*，返回该进程的进程组。上面所示的原型是POSIX.1版本。

每个进程组有一个组长进程。组长进程的标识是，其进程组ID等于其进程ID。

进程组组长可以创建一个进程组，创建该组中的进程，然后终止。只要在某个进程组中有一个进程存在，则该进程组就存在，这与其组长进程是否终止无关。从进程组创建开始到其中最后一个进程离开为止的时间区间称为进程组的生命期。某个进程组中的最后一个进程可以终止，也可以参加另一个进程组。

进程调用setpgid可以参加一个现存的组或者创建一个新进程组 (下一节中将说明用setsid也可以创建一个新的进程组)。

```
#include <sys/types.h>
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

返回：若成功则为0，出错为-1

这将*pid* 进程的进程组ID设置为*pgid*。如果这两个参数相等，则由 *pid* 指定的进程变成进程组组长。

一个进程只能为它自己或它的子进程设置进程组ID。在它的子进程调用了exec后，它就不再能改变该子进程的进程组ID。

如果*pid*是0，则使用调用者的进程ID。另外，如果*pgid*是0，则由*pid*指定的进程ID被用作为进程组ID。

如果系统不支持作业控制(9.8节将说明作业控制)，那么就不定义_POSIX_JOB_CONTROL，在这种情况下，此函数返回出错，errno设置为ENOSYS。

在大多数作业控制shell中，在fork之后调用此函数，使父进程设置其子进程的进程组ID，然后使子进程设置其自己的进程组ID。这些调用中有一个是冗余的，但这样做可以保证父、子进程在进一步操作之前，子进程都进入了该进程组。如果不这样做的话，那么就产生一个竞态条件，因为它依赖于哪一个进程先执行。

在讨论信号时，将说明如何将一个信号送给一个进程 (由其进程ID标识)或送给一个进程组(由进程组ID标识)。同样，waitpid则可被用来等待一个进程或者指定进程组中的一个进程。

9.5 对话期

对话期（session）是一个或多个进程组的集合。例如，可以有图9-6中所示的安排。其中，

在一个对话期中有三个进程组。通常是由 shell的管道线将几个进程编成一组的。例如，图 9-6中的安排可能是由下列形式的shell命令形成的：

```
proc1 | proc2 &
proc3 | proc4 | proc5
```

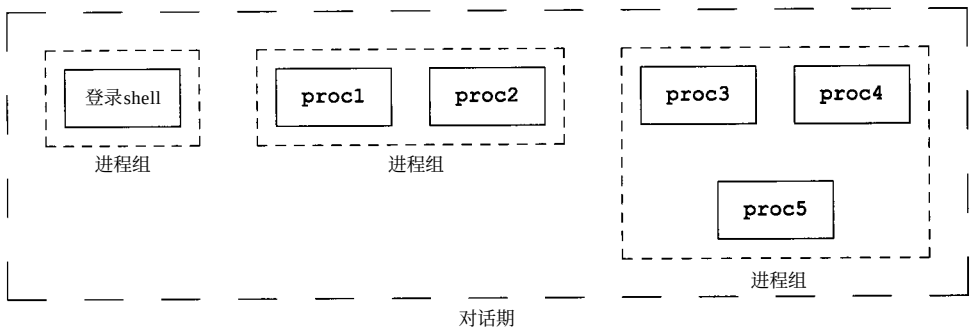

图9-6 进程组和对话期中的进程安排

进程调用setsid函数就可建立一个新对话期。

```
#include <sys/types.h>
#include <unistd.h>

pid_t setsid(void);
```

返回：若成功则为进程组 ID，若出错则为 - 1

如果调用此函数的进程不是一个进程组的组长，则此函数创建一个新对话期，结果为：

(1) 此进程变成该新对话期的对话期首进程（session leader，对话期首进程是创建该对话期的进程）。此进程是该新对话期中的唯一进程。

(2) 此进程成为一个新进程组的组长进程。新进程组ID是此调用进程的进程ID。

(3) 此进程没有控制终端（下一节讨论控制终端）。如果在调用setsid之前此进程有一个控制终端，那么这种联系也被解除。

如果此调用进程已经是一个进程组的组长，则此函数返回出错。为了保证不处于这种情况，通常先调用fork，然后使其父进程终止，而子进程则继续。因为子进程继承了父进程的进程组ID，而其进程ID则是新分配的，两者不可能相等，所以这就保证了子进程不是一个进程组的组长。

> POSIX.1只包括对话期首进程，而没有类似与进程ID和进程组ID的对话期ID。显然，对话期首进程是具有唯一进程ID的单个进程，所以可以将对话期首进程的进程ID视为对话期ID。SVR4就是这样处理的。SVID和SVR4的setsid(2)手册页谈到了以此种方式定义的对话期ID。这是一种实现细节，它不是POSIX.1中定义的，4.3+BSD也不支持它。
>
> SVR4有一个getsid函数，它返回一个进程的对话期ID。此函数不是POSIX.1的所属部分，4.3+BSD也不支持此函数。

9.6 控制终端

对话期和进程组有一些其他特性：

• 一个对话期可以有一个单独的控制终端（controlling terminal）。这通常是我们在其上登录的终端设备（终端登录情况）或伪终端设备（网络登录情况）。

• 建立与控制终端连接的对话期首进程，被称之为控制进程（controlling process）。

• 一个对话期中的几个进程组可被分成一个前台进程组（foreground process group）以及一个或几个后台进程组（background process group）。

• 如果一个对话期有一个控制终端，则它有一个前台进程组，其他进程组则为后台进程组。

• 无论何时键入中断键（常常是DELETE或Ctrl-C）或退出键（常常是Ctrl-\），就会造成将中断信号或退出信号送至前台进程组的所有进程。

• 如果终端界面检测到调制解调器已经脱开连接，则将挂断信号送至控制进程（对话期首进程。）这些特性示于图9-7中。

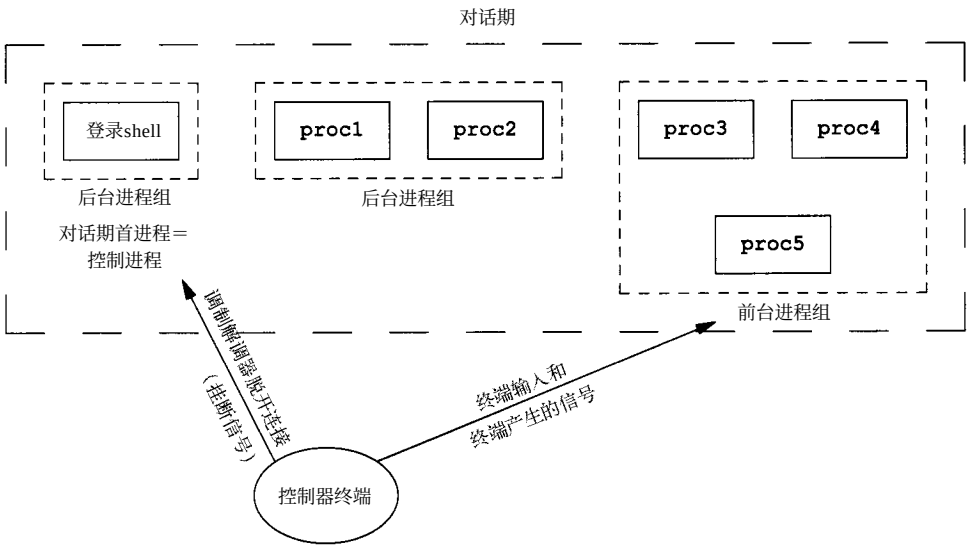

图9-7 进程组、对话期和控制终端

通常，我们不必担心控制终端——登录时，将自动建立控制终端。

> 系统如何分配一个控制终端依赖于实现。19.4节中将说明实际步骤。
>
> 当对话期首进程打开第一个尚未与一个对话期相关联的终端设备时，SVR4将此作为控制终端分配给此对话期。这假定对话期首进程在调用 open时没有指定O_NOCTTY标志(见3.3节)。
>
> 当对话期首进程用 TIOCSCTTY（第三个参数是空指针）的 request参数调用ioctl时，4.3+BSD为对话期分配控制终端。为使此调用成功执行，此对话期不能已经有一个控制终端（通常ioctl调用紧跟在setsid调用之后，setsid保证此进程是一个没有控制终端的对话期首进程）。4.3+BSD 不使用POSIX.1中对open函数所说明的O_NOCTTY标志。

有时不管标准输入、标准输出是否重新定向，程序都要与控制终端交互作用。保证程序读写控制终端的方法是打开文件/dev/tty，在内核中，此特殊文件是控制终端的同义语。自然，如果程序没有控制终端，则打开此设备将失败。

典型的例子是用于读口令的getpass(3)函数（终端回送被关闭）。这一函数由crypt(1)程序调用，而此程序则可用于管通线中。例如：

```
crypt < salaries | lpr
```

它将文件salaries解密，然后经由管道将输出送至打印假脱机程序。因为crypt从其标准输入读输入文件，所以标准输入不能用于输入口令。但是，crypt的一个设计特征是每次运行此程序时，都应输入加密口令，这样也就不需要将口令存放在文件中。

已经知道有一些方法可以破译crypt程序使用的密码。关于加密文件的详细情况请参见Garfinkel和Spafford [1991]。

9.7 tcgetpgrp和tcsetpgrp函数

需要有一种方法来通知内核哪一个进程组是前台进程组，这样，终端设备驱动程序就能了解将终端输入和终端产生的信号送到何处（见图9-7）。

```
#include <sys/types.h>
#include <unistd.h>

pid_t tcgetpgrp(int filedes);
```

返回：若成功则为前台进程组ID，若出错则为-1

```
int tcsetpgrp(int filedes, pid_t pgrpid);
```

返回：若成功则为0，若出错则为-1

函数tcgetpgrp返回前台进程组ID，它与在*filedes*上打开的终端相关。

如果进程有一个控制终端，则该进程可以调用tcsetpgrp将前台进程组ID设置为*pgrpid*。*pgrpid*值应当是在同一对话期中的一个进程组的ID。*filedes*必须引用该对话期的控制终端。

大多数应用程序并不直接调用这两个函数。它们通常由作业控制shell调用。只有定义了_POSIX_JOB_CONTROL，这两个函数才被定义了。否则它们返回出错。

9.8 作业控制

作业控制是伯克利在1980年左右加到UNIX的一个新特性。它允许在一个终端上起动多个作业(进程组)，控制哪一个作业可以存取该终端，以及哪些作业在后台运行。作业控制要求三种形式的支持：

(1) 支持作业控制的shell。

(2) 内核中的终端驱动程序必须支持作业控制。

(3) 必须提供对某些作业控制信号的支持。

> SVR3提供了一种不同形式的作业控制，称为shell层。但是POSIX.1选择了伯克利形式的作业控制，这也是我们在这里所说明的。回忆表2-7，如果系统支持作业控制，则定义常数_POSIX_JOB_CONTROL。
>
> FIPS 151-1要求POSIX.1作业控制。
>
> SVR4和4.3+BSD支持POSIX.1作业控制。

从shell使用作业控制功能角度观察，可以在前台或后台起动一个作业。一个作业只是几个进程的集合，通常是一个进程管道。例如：

```
vi main.c
```

在前台起动了只有一个进程的一个作业。下面的命令：

```
pr *.c | lpr &
make all &
```

在后台起动了两个作业。这两个后台作业所调用的进程都在后台运行。

正如前述，我们需要一个支持作业控制的shell以使用由作业控制提供的功能。对于早期的系统，shell是否支持作业控制比较易于说明。C shell支持作业控制，Bourne shell则不支持，而KornShell能否支持作业控制取决于主机是否支持作业控制。但是现在C shell已被移植到并不支持作业控制的系统上(例如系统V的早期版本)，而SVR4 Bourne shell当用名字jsh而不是sh调用时则支持作业控制。如果主机支持作业控制，则KornShell继续支持作业控制。各种shell之间的差别并不显著时，我们将只是一般地说明支持作业控制的shell和不支持作控制的shell。

当起动一个后台作业时，shell赋与它一个作业标识，并打印一个或几个进程ID。下面的操作过程显示了KornShell是如何处理这一点的。

```
$ make all > Make.out &
[1]     1475
$ pr *.c | lpr &
[2]     1490
$                       键入回车
[2] + Done          pr *.c | lpr &
[1] + Done          make all > Make.out &
```

make是作业号1，所起动的进程ID是1475。下一个管道线是作业号2，其第一个进程的进程ID是1490。当作业已完成而且键入回车时，shell通知我们作业已经完成。键入回车是为了让shell打印其提示符。shell并不在任何随意的时间打印后台作业的状态改变——它只在打印其提示符之前这样做。如果不这样处理，则当我们正输入一行时，它也可能输出。

我们可以键入一个影响前台作业的特殊字符——挂起键（一般采用Ctrl-Z）与终端进行交互作用。键入此字符使终端驱动程序将信号SIGTSTP送至前台进程组中的所有进程，后台进程组作业则不受影响。实际上有三个特殊字符可使终端驱动程序产生信号，并将它们送至前台进程组，它们是：

- 中断字符（一般采用DELETE或Ctrl-C）产生SIGINT。
- 退出字符（一般采用Ctrl-\）产生SIGQUIT。
- 挂起字符（一般采用Ctrl-Z）产生SIGTSTP。

第11章中将说明可将这三个字符更改为任一其他字符，以及如何使终端驱动程序不处理这些特殊字符。

终端驱动程序必须处理与作业控制有关的另一种情况。我们可以有一个前台作业，若干个后台作业，这些作业中哪一个接收我们在终端上键入的字符呢？只有前台作业接收终端输入。如果后台作业试图读终端，那么这并不是一个错误，但是终端驱动程序检测这种情况，并且发送一个特定信号SIGTTIN给后台作业。这通常会停止此后台作业，而有关用户则会得到这种情况的通知，然后就可将此作业转为前台作业运行，于是它就可读终端。下列操作过程显示了这一点：

```
$ cat > temp.foo &          在后台启动，但将从标准输入读
[1]     1681
$                           键入回车
[1] + Stopped (tty input)       cat > temp.foo &
$ fg %1                     使1号作业成为前台作业
cat > temp.foo              shell告诉我们现在哪一个作业在前台
hello, world                输入1行
^D                          键入文件结束符
$ cat temp.foo              检查该行已送入文件
hello, world
```

shell在后台起动cat进程，但是当cat试图读其标准输入（控制终端）时，终端驱动程序知道它是个后台作业，于是将SIGTTIN信号送至该后台作业。shell检测到其子进程的状态改变（回忆8.6节中对wait和waitpid的讨论），并通知我们该作业已被停止。然后，用 shell的fg命令将此停止的作业送入前台运行（关于作业控制命令，例如 fg和bg的详细情况，以及标识不同作业的各种方法请参阅有关shell的手册页）。这样做使shell将此作业转为前台进程组（tcsetpgrp），并将继续信号(SIGCONT)送给该进程组。因为该作业现在前台进程组中，所以它可以读控制终端。

如果后台作业输出到控制终端又将发生什么呢？这是一个我们可以允许或禁止的选择项。通常，可以用stty(1)命令改变这一选择项（第 11章将说明在程序中如何改变这一选择项）。下面显示了这种操作过程：

```
$ cat temp.foo &        在后台执行
[1]     1719
$ hello, world          在提示符后出现后台作业的输出
                        键入回车
[1] + Done          cat temp.foo &
$ stty tostop           禁止后台作业向控制终端输出
$ cat temp.foo &        在后台再次执行
[1]     1721
$                       键入回车，发现作业已停止
[1] + Stopped(tty output)       cat temp.foo &
$ fg %1                 将停止的作业恢复为前台作业
cat temp.foo            shell告诉我们现在哪一个作业在前台
hello, world            该作业的输出
```

图9-8摘录了我们已说明的作业控制的某些功能。穿过终端驱动程序框的实线表示：终端I/O和终端产生的信号总是从前台进程组连接到实际终端。对应于 SIGTTOU信号的虚线表示，后台进程组进程的输出是否出现在终端是可选择的。

是否需要作业控制是一个有很多争论的问题。作业控制是在窗口终端广泛得到应用之前设计和实现的。很多人认为设计得好的窗口系统已经免除了对作业控制的需要。某些人抱怨作业控制的实现要求得到内核、终端驱动程序、 shell以及某些应用程序的支持，是吃力不讨好的事情。某些人在窗口系统中使用作业控制，他们认为两者都需要。不管你的意见如何，作业控制是POSIX.1以及FIPS 151-1的组成部分，它还将继续存在。

9.9 shell执行程序

让我们检验一下shell是如何执行程序的，以及这与进程组、控制终端和对话期等概念的关系。为此，要再次使用ps命令。

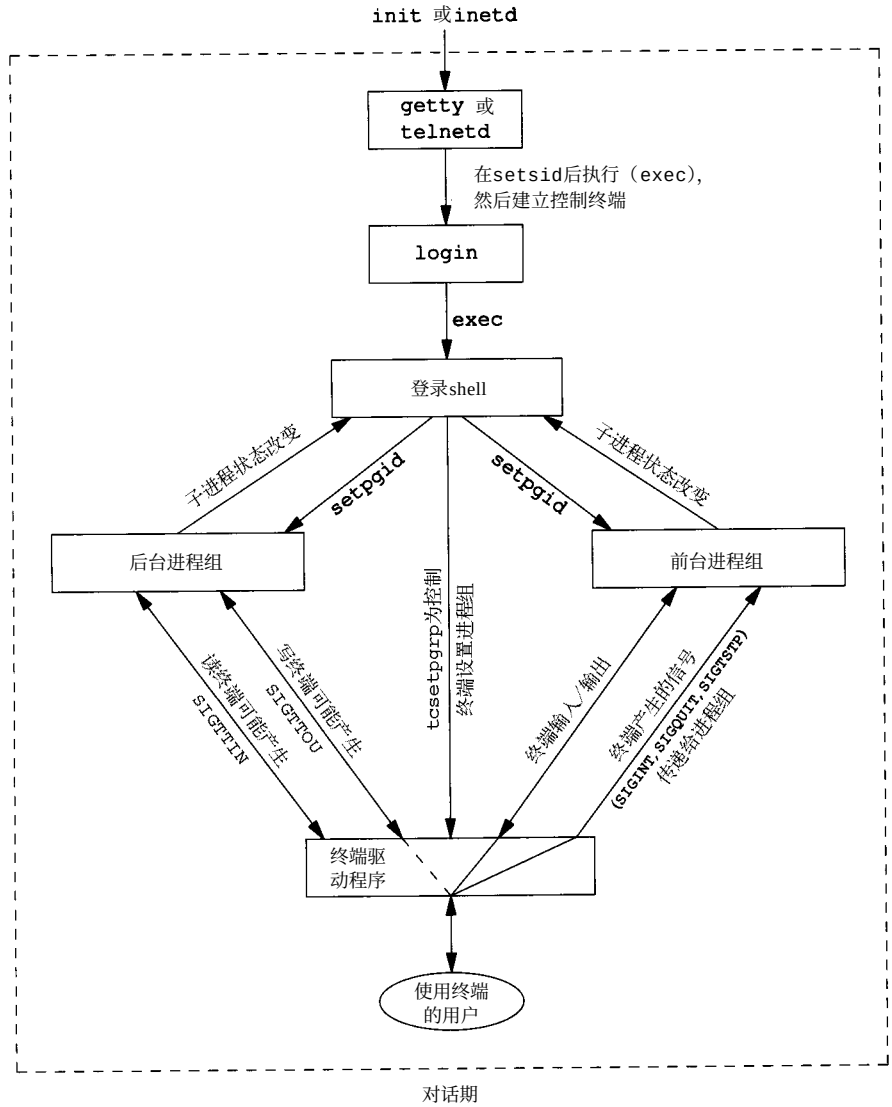

图9-8 对于前台、后台作业以及终端驱动程序的作业控制功能摘要

首先使用不支持作业控制的经典的Bourne shell。如果执行：

```
ps -xj
```

则其输出为：

```
PPID   PID   PGID   SID   TPGID   COMMAND
   1   163    163   163   163   -sh
 163   168    163   163   163   ps
```

其中，删除了一些我们现在不感兴趣的列——终端名、用户ID、CPU时间等。shell和ps命令两者位于同一对话期和前台进程组(163)中。因为163是在TPGID列中显示的进程组，所以称其为前台进程组。ps的父进程是shell，这正是我们所期望的。注意，登录shell是由login以—作为其第一个字符调用的。

不幸的是，ps(1)命令的输出在各个UNIX版本中都有所不同。在SVR4之下，使用命令ps -j1得到类似的输出，但SVR4不打印TPGID字段。在4.3+BSD之下，使用命令ps -xj -otpgid。

注意，说进程与终端进程组ID(TPGID列)相关联是用词不当。进程并没有终端进程控制组。进程属于一个进程组，而进程组属于一个对话期。对话期可能有，也可能没有控制终端。如果它确有一个控制终端，则此终端设备知道其前台进程的进程组 ID。这一值可以用tcsetpgrp函数在终端驱动程序中设置（见图9-8）。前台进程组ID是终端的一个属性，而不是进程的属性。取自终端设备驱动程序的该值是ps在TPGID列中打印的值。如果ps发现此对话期没有控制终端，则它在该列打印－1。

如果在后台执行该命令：

```
ps -xj &
```

则唯一改变的值是命令的进程ID。

```
PPID   PID   PGID   SID   TPGID   COMMAND
   1   163    163   163   163   -sh
 163   169    163   163   163   ps
```

因为这种shell不知道作业控制，所以后台作业没有构成另一个进程组，也没有从后台作业处取走控制终端。

现在看一看Bourne shell如何处理管道线。执行下列命令：

```
ps -xj | cat1
```

其输出是：

```
PPID   PID   PGID   SID   TPGID   COMMAND
   1   163    163   163     163   -sh
 163   200    163   163     163   cat1
 200   201    163   163     163   ps
```

（程序cat1只是标准cat程序的一个副本，但名字不同。本节还将使用 cat的另一个名为cat2的副本。在一个管道线中使用两个cat时，不同的名字可使我们将它们区分开来。）注意，管道中的最后一个进程是shell的子进程，该管道中的第一个进程则是最后一个进程的子进程。从中可以看出，shell fork一个它的副本，然后此副本再为管道线中的每条命令各fork一个进程。

如果在后台执行此管道线：

```
ps -xj | cat1 &
```

则只有进程ID改变了。因为shell并不处理作业控制，后台进程的进程组 ID仍是163，如同终端进程组ID一样。

如果一个后台进程试图读其控制终端，则会发生什么呢?例如，若执行：

```
cat > temp.foo &
```

在有作业控制时，后台作业被放在后台进程组，如果后台作业试图读控制终端，则会产生信号SIGTTIN。在没有作业控制时，其处理方法是：如果该进程自己不重新定向标准输入，则 shell自动将后台进程的标准输入重新定向到 /dev/null。读/dev/null则产生一个文件结束。这就意味着后台cat进程立即读到文件尾，并正常结束。

上面说明了对后台进程通过其标准输入存取控制终端的适当的处理方法，但是，如果一个后台进程打开/dev/tty并且读该控制终端，又将怎样呢?对此问题的回答是“看情况”。但是这很可能不是我们所要的。例如：

```
crypt < salaries | lpr &
```

就是这样的一条管道线。我们在后台运行它，但是 crypt程序打开/dev/tty，更改终端的特性(禁止回送)，然后从该设备读，最后复置该终端特性。当执行这条后台管道时， crypt在终端上打印提示符“ Password: ”，但是shell读取了我们所输入的加密码口令，并企图执行其中一条命令。我们输送给shell的下一行，则被crypt进程取为口令行，于是salaries也就不能正确地被译码，结果将一堆没有用的信息送到了打印机。在这里，我们有了两个进程，它们试图同时读同一设备，其结果则依赖于系统。前面说明的作业控制以较好的方式处理一个终端在多个进程间的转接。

返回到Bourne shell实例，在一条管道中执行三个进程：

```
ps -xj | cat1 | cat2
```

下面看一看shell所用的进程控制：

```
PPID   PID   PGID   SID   TPGID   COMMAND
   1   163    163   163     163   -sh
 163   202    163   163     163   cat2
 202   203    163   163     163   ps
 202   204    163   163     163   cat1
```

再重申一遍，该管道中的最后一个进程是shell的子进程，而执行管道中其他命令的进程则是该最后进程的子进程。图9-9显示了所发生的情况。

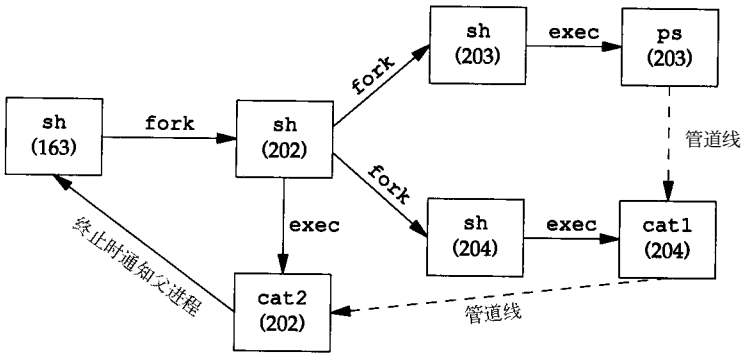

图9-9 Bourne shell执行管道线ps -xj | cat1 | cat2 时的进程

因为该道通线中的最后一个进程是登录 shell的子进程，当该进程(cat2)终止时， shell得到通知。

现在让我们用作业控制shell来检验一下同一个例子。这将显示这些shell处理后台作业的方法。在本例中将使用KornShell——用C shell得到的结果几乎是一样的。

```
ps -xj
```

其输出为：

```
PPID   PID   PGID   SID   TPGID   COMMAND
```

```
    1   700    700   700   708   -ksh
  700  708    708 700   708   ps
```

（从本例开始，以粗体显示前台进程组。）我们立即看到了与 Bourne shell例子的区别。KornShell将前台作业(ps)放入了它自己的进程组(708)。ps命令是进程组组长进程，并是该进程组的唯一进程。进一步而言，此进程组具有控制终端，所以它是前台进程。我们的登录 shell在执行ps命令时是后台进程组。但需要注意的是，这两个进程组700和708都是同一对话期的成员。事实上，在本书的实例中对话期决不会改变。

在后台执行此进程：

```
ps -xj &
```

其输出为：

```
PPID   PID   PGID   SID   TPGID   COMMAND
   1  700    700 700   700   -ksh
 700   709    709   700   700   ps
```

再一次，ps命令被放入它自己的进程组，但是此时进程组(709)不再是前台进程组。这是一个后台进程组。TPGID 700指示前台进程组是登录shell。

按下列方式在一个管道中执行两个进程：

```
ps -xj | cat1
```

其输出为：

```
PPID   PID   PGID   SID   TPGID   COMMAND
   1   700    700   700   710   -ksh
 700  710    710 700   710   ps
 700  711    710 700   710   cat1
```

两个进程ps和cat1都在一个新进程组(710)中，这是一个前台进程组。在本例和类似的 Bourne shell实例之间能看到另一个区别。Bourne shell首先创建将执行管道线中最后一条命令的进程，而此进程是第一个进程的父进程。在这里， KornShell是两个进程的父进程。但是，如果在后台执行此管道线：

```
ps -xj | cat1 &
```

其结果显示现在KornShell以与Bourne shell相同的方式产生进程。

```
PPID   PID   PGID   SID   TPGID   COMMAND
   1  700    700 700   700   -ksh
 700   712    712   700   700   cat1
 712   713    712   700   700   ps
```

两个进程712和713都处在后台进程组712中。

9.10 孤儿进程组

一个父进程已终止的进程称为孤儿进程 (orphan process)，这种进程由init进程收养。现在我们要说明整个进程组也可成为孤儿，以及POSIX.1如何处理它。

实例

考虑一个进程，它 fork了一个子进程然后终止。这在系统中是经常发生的，并无异常之

处，但是在父进程终止时，如果该子进程停止（用作业控制）又将如何呢？子进程如何继续，以及子进程是否知道它已经是孤儿进程？程序 9-1是这种情况的一个例子。下面要说明该程序的某些新特征。图9-10显示了程序9-1已经起动，父进程已经fork了子进程后的情况。

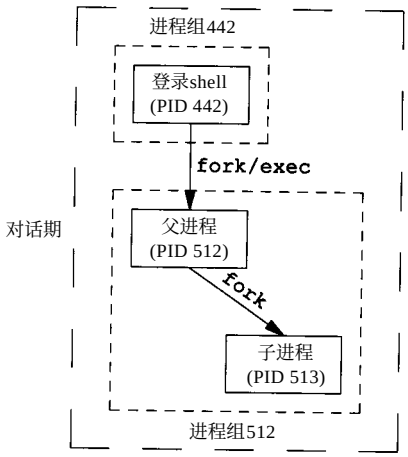

图9-10 将成为孤儿的进程组的实例

这里，假定使用了一个作业控制 shell。回忆前面所述，shell将前台进程放在一个进程组中 (本例中是512)，shell则留在自己的组内(442)。子进程继承其父进程(512)的进程组。在fork之后：

• 父进程睡眠5秒钟，这是一种让子进程在父进程终止之前运行的一种权宜之计。

• 子进程为挂断信号（SIGHUP）建立信号处理程序。这样就能观察到SIGHUP信号是否已送到子进程。（第10章将讨论信号处理程序。）

• 子进程用kill函数向其自身发送停止信号（SIGTSTP)。这停止了子进程，类似于用终端挂起字符（Ctrl-Z）停止一个前台作业。

• 当父进程终止时，该子进程成为孤儿进程，共父进程ID成为1，也就是init进程ID。

• 现在，子进程成为一个孤儿进程组的成员。 POSIX.1将孤儿进程组（orphaned process group）定义为：该组中每个成员的父进程或者是该组的一个成员，或者不是该组所属对话期的成员。对孤儿进程组的另一种描述可以是：一个进程组不是孤儿进程组的条件是：该组中有一个进程，其父进程在属于同一对话期的另一个组中。如果进程组不是孤儿进程组，那么在属于同一对话期的另一个组中的父进程就有机会重新起动该组中停止的进程。

在这里，进程组中所有进程的进程（如进程 513的父进程1)属于另一个对话期。所以此进程组是孤儿进程组。

• 因为在父进程终止后，进程组成为孤儿进程组， POSIX.1要求向新孤儿进程组中处于停止状态的每一个进程发送挂断信号（SIGHUP），接着又向其发送继续信号（SIGCONT)。

• 在处理了挂断信号后，子进程继续。对挂断信号的系统默认动作是终止该进程，为此必须提供一个信号处理程序以捕捉该信号。因此，我们期望 sig_hup函数中的printf会在pr_ids函数中的printf之前执行。

程序9-1 创建一个孤儿进程组

```
#include    <sys/types.h>
#include    <errno.h>
#include    <fcntl.h>
#include    <signal.h>
#include    "ourhdr.h"

static void sig_hup(int);
static void pr_ids(char *);

int
main(void)
{
    char    c;
    pid_t   pid;

    pr_ids("parent");
```

```
    if ( (pid = fork()) < 0)
        err_sys("fork error");

    else if (pid > 0) { /* parent */
        sleep(5);       /* sleep to let child stop itself */
        exit(0);        /* then parent exits */

    } else {            /* child */
        pr_ids("child");
        signal(SIGHUP, sig_hup);    /* establish signal handler */
        kill(getpid(), SIGTSTP);    /* stop ourself */
        pr_ids("child");    /* this prints only if we're continued */
        if (read(0, &c, 1) != 1)
            printf("read error from control terminal, errno = %d\n", errno);
        exit(0);
    }
}

static void
sig_hup(int signo)
{
    printf("SIGHUP received, pid = %d\n", getpid());
    return;
}

static void
pr_ids(char *name)
{
    printf("%s: pid = %d, ppid = %d, pgrp = %d\n",
            name, getpid(), getppid(), getpgrp());
    fflush(stdout);
}
```

下面是程序9-1的输出：

```
$ a.out
parent: pid = 512, ppid = 442, pgrp = 512
child: pid = 513, ppid = 512, pgrp = 512
$ SIGHUP received, pid = 513
child: pid = 513, ppid = 1, pgrp = 512
read error from control terminal, errno = 5
```

注意，因为两个进程，登录shell和子进程都写向终端，所以shell提示符和子进程的输出一起出现。正如我们所期望的那样，子进程的父进程ID变成1。

注意，在子进程中调用pr_ids后，程序企图读标准输入。正如前述，当后台进程组试图读控制终端时，则对该后台进程组产生SIGTTIN。但在这里，这是一个孤儿进程组，如果内核用此信号停止它，则此进程组中的进程就再也不会继续。 POSIX.1规定，read返回出错，其errno设置为EIO（在作者所用的系统中其值是5）。

最后，要注意的是父进程终止时，子进程变成后台进程组，因为父进程是由 shell作为前台作业执行的。

在19.5节的pty程序中将会看到孤儿进程组的另一个例子。

9.11 4.3+BSD实现

上面说明了进程、进程组、对话期和控制终端的各种属性，值得观察一下所有这些是如何实现的。下面简要说明4.3+BSD的实现。SVR4实现的某些详细情况则参见 Williams〔1989〕。

图9-11显示了4.3+BSD的各种数据结构。

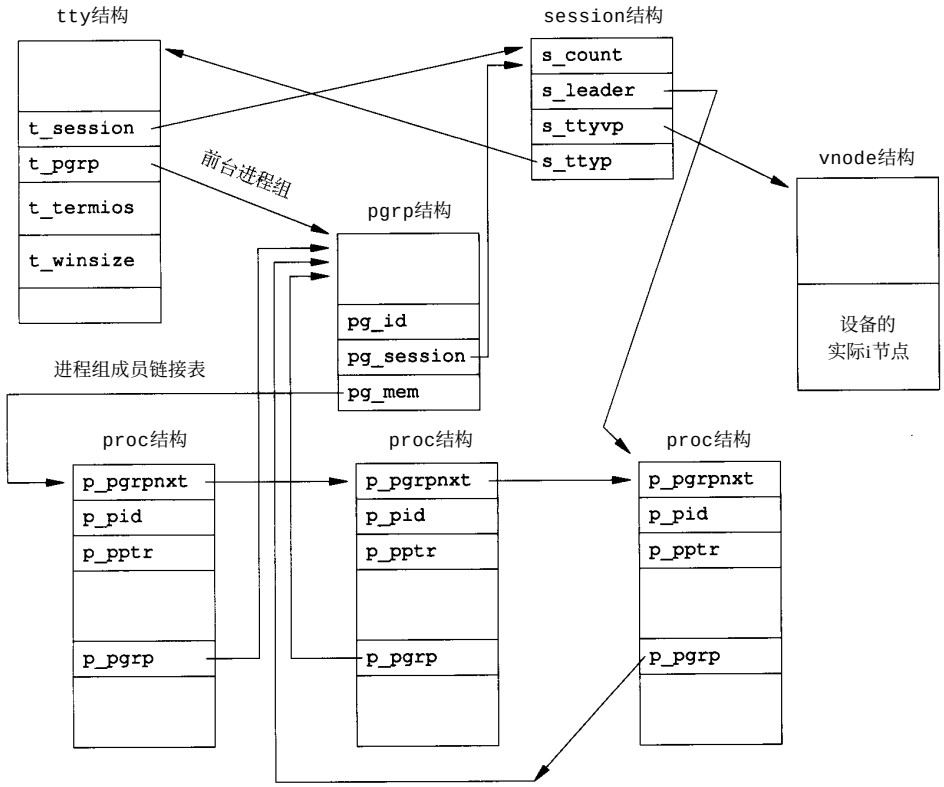

图9-11　对话期和进程组的4.3+BSD实现

下面说明图中的各个字段。从session结构开始。每个对话期都分配了这样一种结构（例如，每次调用setsid时）。

• s_count是该对话期中的进程组数。当此计数器减至0时，则可释放此结构。

• s_leader是指向对话期首进程proc结构的指针。如上所述，4.3+BSD不保持对话期ID字段，而SVR4则保持此字段。

• s_ttyvp是指向控制终端vnode结构的指针。

• s_ttyp是指向控制终端tty结构的指针。

在调用setsid时，在内核中分配一个新的对话期结构。s_count设置为1，s_leader设置为调用进程的proc结构的指针，因为新对话期没有控制终端，所以s_ttyvp和s_ttyp设置为空指针。

接看说明tty结构。每个终端设备和每个伪终端设备均在内核中分配这样一种结构（第 19章将对伪终端作更多说明。）

• t_session指向将此终端作为控制终端的session结构（注意，tty结构指向session结构，结构也指向tty结构）。终端在失去载波信号（见图9-7）时使用此指针将挂起信号送给对话期首进程。

• t_pgrp指向前台进程组的pgrp结构。终端驱动程序用此字段将信号送向前台进程组。由输入特殊字符(中断、退出和挂起)而产生的三个信号被送至前台进程组。

• t_termios是包含所有这些特殊字符和与该终端有关信息（例如，波特率、回送打开或关闭等）的结构。第11章将再说明此结构。

• t_winsize是包含终端窗口当前尺寸的 winsize结构。当终端窗口尺寸改变时，信号SIGWINCH被送至前台进程组。11.12节将说明如何设置和存取终端当前窗口尺寸。

注意，为了找到特定对话期的前台进程组，内核从 session结构开始，然后用s_ttyp得到控制终端的tty结构，然后用t_pgrp得到前台进程组的pgrp结构。

pgrp结构包含一个进程组的信息。

• pg_id是进程组ID。

• pg_session指向此进程组所属的session结构。

• pg_mem是指向此进程组第一个进程proc结构的指针。proc结构中的p_pgrpnxt指向此组中的下一个进程，进程组中最后一个进程proc中的p_pgrpnxt则为空指针。

proc结构包含一个进程的所有信息。

• p_pid包含进程ID。

• p_pptr是指向父进程proc结构的指针。

• p_pgrp指向本进程所属的进程组的pgrp结构。

• p_pgrpnxt是指向进程组中下一个进程的指针。

最后还有一个vnode结构。在打开控制终端设备时分配此结构。进程对 /dev/tty的所有访问都通过vnode结构。在图9-11中，实际i节点是v节点的一部分。3.10节曾提及这是4.3+BSD的实现方法，而SVR4则将v节点存在i节点中。

9.12 小结

本章说明了进程组之间的关系——对话期，它由若干个进程组组成。作业控制是当今很多UNIX系统所支持的功能，本章说明了它是如何由支持作业控制的 shell实现的。在这些进程关系中也涉及到了/dev/tty。

所有这些进程的关系都使用了很多信号方面的功能。下一章将详细讨论UNIX中的信号机制。

习题

9.1 考虑6.7节中说明的utmp和wtmp文件，为什么logout记录是由4.3+BSD的init 进程写的？对于网络登录的处理与此相同吗？

9.2 编写一段程序，要求调用fork并在子进程中建立一个新的对话期。验证子进程变成了进程组组长且不再有控制终端。