第 4 章 信号和会话管理

4.1 简 介

信号提供了一种通知进程系统事件发生的机制。它也是作为用户进程之间进行通信和同步的一种原始机制。对于不同版本的 UNIX 系统来说，信号的编程接口，行为特性和内部实现都不尽相同。不同版本中的同一个信号也可能是互不相同的。此外，操作系统提供了其他的系统调用和库例程来支持早期的编程接口和维持向上的兼容性，但是这增加了编程人员的困难①。

最早的 System 5 的信号机制，开始实现就是不可靠的，而且存在很多不足之处。许多问题都在 BSD4.2 UNIX 中得到了解决，而在 BSD4.3 中得到加强和改善。BSD4.2 UNIX 推出一种更加健壮的信号机制。但是 BSD4.2 的信号接口和 System 5 的信号接口在有些方面不兼容。这将带来下述两方问题。一方是那些想写出可移植代码的应用程序开发者。另一方是那些 UNIX 的供应商，他们希望自己的 UNIX 版本和 System 5 及 BSD 都能兼容。

POSIX 1003.1 标准[IEEE 90](也被叫做 POSIX.1)强行制定一些规定来缓解由于各式各样的信号机制实现手段带来的混乱。它定义一个标准的信号接口，所有兼容 UNIX 版本的信号机制实现都必须支持。然而，POSIX 标准并不规定这些接口如何实现。操作系统可以自由的选择是在内核实现，还是通过用户级的例程库实现。或是两者都用。

SVR4 的开发者提出一种新的和 POSIX 标准兼容的信号机制实现方法,而且结合了许多 BSD 的信号机制的特点。几乎所有的 UNIX 变体(如 Solaris,AIX,HP-UX,4.4BSD 以及 Digital UNIX)都提供了和 POSIX 标准兼容的信号实现机制。其中 SVR4 中的信号机制实现保持和所有早期的 System 5 版本的向上兼容。

本章首先解释信号的基本含义，分析在早期的 System 5 的信号机制实现中出现的问题。接下来，介绍在能提供可靠的信号机制的 UNIX 系统中是如何解决这些问题的。最后，还要阐述一下和信号有密切关系的作业控制和会话管理。

4.2 信号生成和处理

当定义的一组事件之一发生时，信号机制提供了调用一个过程的一种方式。这些事件用整数来标识，通常它们用符号常数来表示。有些事件是异步发生的(如用户通过在终端上按 CTRL-C 键来向一个进程发送中断信号)，其他的一些事件是同步错误或异常(如，进程访问一个非法的地址空间)。

在一个信号的过程中有两个阶段：生成和传送。当一个事件发生时，需要通知一个进程，这时生成一个信号。当该进程识别出信号的到来，就采取适当的动作来传送或处理信号。在信号到来和进程对信号进行处理之间，信号在这个进程上挂起(pending)。

最初的 System 5 信号实现定义了 15 种不同的信号。4BSD 和 SVR4 都支持 31 种信号。每个信号都被赋予 1～31 之间的一个整数值(对于不同的函数来说，把信号数值设置为零有特殊的含义，如“没有信号”)。对 System 5 和 BSD UNIX 来说，信号和信号数值之间的映射是不同的(比如，在 4.3BSD 中信号 SIGSTOP 的数值为 17，而在 SVR4 中它的数值为 23)。进一步，许多商用的 UNIX 系统的变体(如 AIX)支持多于 31 个信号。因此，程序员使用符号常数来识别信号。POSIX 003.1 为它定义的所有的信号指定符号名。最低程度这些符号名对于所有的 POSIX 兼容的实现是可以移植的。

4.2.1 信号处理

每一个信号有一个缺省动作，它是当进程没有给这个信号指定处理程序时内核对信号的

处理。有 5 种缺省的动作：

异常中止(abort)在生成一个 core 映像后，终止进程。也就是说，在当前进程的目录下把进程的地址空间的内容和寄存器的内容保存在一个叫做 core 的文件之中②。这个文件以后可以为 debugger 等调试工具来分析使用。

退出(exit)不产生 core 映像文件就终止进程。

忽略(ignore)忽略这个信号。

停止(stop)挂起这个进程。

继续(continue)如果进程被挂起，则恢复进程的运行。否则，就忽略该信号。

一个进程可以对任何信号指定另一个动作或重载缺省动作。这个替换的动作可以是忽略这一信号，或是调用用户定义的信号处理程序。在任何时候，进程可以指定一个新的动作或把它重新设置为缺省动作。一个进程可以暂时地阻塞一个信号(不是在 SVR2 或更早的版本)。在这种情况下，直到被解除阻塞，信号才能被传送。信号 SIGSTOP 和信号 SIGKILL 是非常特殊的——用户不能忽略、阻塞或是为它们指定一个处理程序。表 4-1 列出了所有的信号和它们的缺省动作及一些限制。

表 4-1 UNIX 中的信号

信号	说明	缺省动作	可用性	注释
SIGABRT	进程异常终立	异常终止	APSB	
SIGALRM	实时报警	退出	OPSB	
SIGBUS	总线错误	异常终止	OSB	
SIGCHLD	子进程死亡或挂起	忽略	OJSB	6
SIGCONT	恢复被挂起的进程	继续/忽略	JSB	4
SIGEMT	仿真器陷阱	异常终止	OSB	
SIGFPE	算术失效	异常终止	OAPSB	
SIGHUP	挂断	退出	OPSB	
SIGILL	非法指令	异常终止	OAPSB	2
STGINFO	状态请求(control-T)	忽略	B	
SIGINT	tty 中断(control-C)	退出	OAPSB	
SIGIO	异步 I/O 事件	退出/忽略	SB	3
SIGIOT	I/O 陷阱	异常终止	OSB	
SIGKILL	终止进程	退出	OPSB	1
SIGPIPE	向没有读者的管道中写数据	退出	OPSB	
SIGPOLL	可轮询事件	退出	S	
SIGPROF	统计信息计时器	退出	SB	
SIGPWR	电源失效	忽略	OS	
SIGQUIT	tty 退出信号 (control-\)异常终止			OPSB
SIGSEGV	段失效	异常终止	OAPSB	
SIGStOP	停止进程	停止	JSB	1
SIGSYS	无效的系统调用	退出	OAPSB	
SIGTERM	终止进程	退出	OAPSB	
SIGTRAP	硬件失效	异常终止	OSB	2
SIGTSTP	tty 停止信号(control-Z)停止		JSB	
SIGTTIN	后台进程读 tty	停止	JSB	
SIGTTOU	后台进程写 tty	停止	JSB	5
SIGURG	I/O 通道上的紧急事件	忽略	SB	

SIGUSR1	用户自定义	退出 OPSB
SIGUSR2	用户自定义	退出 OPSB
SIGVTALRM	虚时钟报警	退出 SB
SIGWINCH	窗口大小变化	忽略 SB
SIGXCPU	超出 CPU 限量	异常终止 SB
SIGXFSZ	超出文件大小限量	异常终止 SB

可用性： O 最初的 SVR2 信号　　A ANSI C
B 4.3BSD　　S SVR4
P POSIX.1　　J POSIX.1，仅用于支持作业控制

备注： 1 不能捕捉，阻塞或者忽略。
2 即便在 System V 的实现中也没有重新置为缺省值。
3 SVR4 中的缺省动作是退出，4.3BSD 中是忽略。
4 如果挂起其缺省动作就是继续，否则忽略。不能被阻塞。
5 进程可以选择允许向后台写，而不产生这个信号。
6 在 SVR3 及更早版本中称为 SIGCLD。

需要指出的是，任何动作，包括终止进程，都只能由接收到信号的进程来执行。而这至少要求该进程被调度去执行。在一个非常忙的系统中，如果进程具有一个较低的优先级，那么该进程对信号的处理就需要很长时间。如果进程被换出(swapped out)、挂起或是处在一个不可中断的阻塞状态下将会有更多的延迟。

当内核调用 issig()函数来为进程检查挂起的信号时，接收进程就会知道发送给它的信号。内核仅仅在下列时刻调用 issig()：

·从系统调用或是中断返回到用户态之前。
·进程在阻塞到一个可中断的事件之前。
·刚刚从一个可中断的事件醒来后。

如果 issig()返回的是 TRUE，那么内核调用 psig()函数去分派信号。psig()则终止进程，或在必要的时候生成 core 映象文件，或是调用用户定义的信号处理程序。sendsig()让进程返回到用户态，把控制转移到用户处理程序，当信号处理程序执行完时安排进程恢复执行被中断的代码、因为 sendsig()必须管理用户堆栈并且对进程上下文进行保存、恢复和修改等操作，它的实现是和指定的机器密切相关的。

由异步事件引起的信号可以发生在进程代码路径(code path)的任何一条指令完成之后。当信号处理程序执行完后，进程从被信号中断的地方恢复执行(见图 4-1)。如果信号到来时，进程正处于系统调用运行之中，那么内核通常取消这个系统调用并返问一个 EINTR 错。4.2BSD 引进一种机制，在信号到来和处理完成之后，自动重新执行某个系统调用(见 4.4.3 节)。4.3BSD 提供一个系统调用 siginterrupt 在每一个信号的基础上禁止这个特性。

图 4-1 信号处理

4.2.2 信号生成

内核为进程生成信号，来响应不同的事件。这些事件可以是由接收信号的进程本身引起，也可以是由另一个进程，或是中断或是外部动作引起的。信号的主要的源(source)如下：

异常　　当一个进程出现异常(比如，试图执行一个非法指令)，内核通过向进程发送一个消息来通知进程异常的发生。

其他进程　一个进程可以通过信号 kill 或是 sigsend 系统调用向另一个进程或另一个进程组发送消息。一个进程甚至可以向自身发送消息。

终端中断　某些键盘字符如 Ctrl-C 或是 Ctrl-＼等向该终端的前台进程发送消息。通

过命令 stty 用户可以把每一个终端产生的信号同一个特定的键或是组合键结合起来。

作业控制　　发送作业控制信号给那些想要读或写终端的后台进程。作业控制外壳程序(shells)如 csh 和 ksh 使用信号来管理前台和后台的进程。当一个进程终止或挂起时，内核通过信号来通知它的父进程。

分配额　　当一个进程使用超过分配给它的 CPU 时间或是文件大小的限制，内核向进程发一个消息。

通知　一个进程也许要求能被通知某些事件的发生。如，设备已经就绪等待 I/O 操作。内核通过发送信号来通知进程。

报警　　一个进程设定一段时间，到时后，内核通过信号来通知进程，这称为报警。有三种不同类型的报警，它们使用不同的定时器。ITIMER_REAL 测量实时(时钟)时间，它产生信号 SIGALRM。ITIMER_VIRTUAL 测量虚拟时间。也就是说，只有进程在用户态下运行它才开始计时，产生信号 SIGVTALRM。ITIMER_PROF 测量进程使用的全部时间，不论是在内核态还是在用户态下运行。它产生的是 SIGPROF 信号。对不同的厂家的 UNIX 支持的报警和定时器有很大的不同。

4.2.3 典型情景

我们考虑信号生成(generation)和传递(delivery)的一些例子。假设用户在终端按下 Ctrl-C，这将产生终端中断(对其他字符也是这样)。终端驱动程序能够识别出这是一个产生信号的字符，同时向这个终端的前台进程发送一个 SIGINT 信号。(如果前台进程不只是一个进程，那么终端驱动程序向每一个进程发送一个信号。)当这个进程被调度去执行时，它将在上下文切换后返回到用户态时看见这个信号。有时，前台进程就是中断发生时的当前运行的进程。在这种情况下.信号处理程序中断前台进程并把信号发送给它③。当进程从中断返回时，它会检查并发现这个信号。

异常是一种同步信号。它们通常是由程序错误引起的(如被零除，非法指令等)。如果程序能以同样的方式运行(也就是说，重复相同的执行路径时)，则这些异常将在相同的地方重新发生。当程序中发生一个异常时，将引起一个到内核态的陷入(Trap)。内核中陷入处理程序识别出这个异常并发送合适的信号到当前进程。当陷入处理程序将要返回用户态时，它调用 issig()，这样就接收到信号。

不能同时向一个进程挂起多个信号。这种情况下，这些信号一次处理一个。一个信号也可能在执行信号处理程序时到来，这将引起信号处理程序的嵌套。在大多数的信号机制的实现中，用户可以在唤起一个指定的信号处理程序之前，让内核有选择地封锁某些信号(见 4.4.3 节)。这样就可以允许用户禁止或控制信号处理程序的嵌套。

4.2.4 睡眠和信号

当一个睡眠进程接收到一个信号时会发生什么呢？这个进程会被提前唤醒来处理这个信号还是让这个信号保持在这个进程上挂起等待进程醒来？

这个答案和进程睡眠的原因有关。如果这个进程进入睡眠是等待一个事件如磁盘 I/O 操作的完成，而且这个事件很快就会发生，那么它就会让信号保持挂起。另一方面，如果这个进程是等待用户去敲键盘上的一个字符，那么它可能无限制的等下去。我们就通过信号的方法来中断这个进程。

因此，在 UNIX 中有两种类型的睡眠——可中断的和不可中断的，一个进程睡眠是等待短时间(short term)事件，如磁盘 I/O 操作就是不可中断的睡眠，它不能被信号干扰。而如果进程等待的事件是终端 I/O 操作，这类事件也许很长时间不会发生，那么它睡眠在可中断状态。如果有信号向该进程发送，它将被唤醒。

如果对于在不可中断睡眠状态下的进程产生一个信号，这个信号将被标记为挂起(pending)。在此时进程不会采取任何动作。进程甚至在被唤醒后也不会看到这个信号，直到

该进程要返回到用户态或是它要被阻塞到一个可中断的事件上,它才会看见并处理这个信号。

如果一个进程将要因一个可中断的事件而被封锁时，它将在封锁前检查是会有信号。如果有信号存在，它将处理信号而忽略系统调用。如果在进程被封锁之后产生了信号，内核将唤起这个进程。当进程醒来后执行时——无论是因为进程等待的事件发生了还是因为它的睡眠被一个信号中断——它都要执行 issig()来检查信号。如果有一个信号挂起，那么紧跟着要调用 psig()。如下所示；

```
if(issig())
    psig();
```

4.3 不可靠信号

早期(SVR2 和更早版本的 UNIX)的信号机制的实现[Bach 86]是不可靠的而且有缺陷的。尽管它和在 4.2 节中描述的模型一致，但是它有一些不足之处。

最严重的问题是关于可靠的信号传递。信号处理程序不是永久存在的，而且不能屏蔽相同的信号再次发生。假设用户为一个特定的信号安装了处理程序，当这个信号发生时，内核将在唤起这个处理程序前重新设置信号处理动作为缺省。如果用户希望捕获信号的再次发生，那么他必须每次都要重新安装这个信号处理程序。参见例子 4-1。

实例 4-1 重新安装信号处理程序

```
void  sigint_handler(sig)
int     sig;
{
    signal(SIGINT sigint_hanler); /* reinstall the handleer */
    ...                                 /* handle the signal */
}
main ( )
{
  signal(SIGINT, sigint_handler); /* install the handler */
    ...
}
```

然而这将导致竞争条件的出现、假设用户连续两次很快地键入 Ctrl-C。第一个 Ctrl-C 引起一个 SIGINT 信号，它将重新设置信号处理程序为缺省动作，接着调用信号处理程序。如果第二个 Ctrl-C 在处理程序安装之前键人，内核将采用缺省动作终止进程。这将在信号处理程序被唤起时和重新安装处理程序之间流下一个窗口 (window)。在这个窗口期间信号不能被捕获。因此，以前的信号实现通常被人们认为是不可靠的信号。

关于睡眠进程也存在一个性能的问题。在早期的信号实现机制中，所有关于信号本质的信息都被保存在 u 区的一个数组 u_signal[]中，对于每一种类型的信号都包含一个项。这个项包括用户定义的信号处理程序的地址，SIG_DFL 指定了缺省采取的动作，SIG_IGN 说明信号将被忽略。

因为内核只能读当前进程的 u 区，它没有方法知道另一个进程是如何处理信号的。特别地，如果内核必须在一个可中断睡眠中寄一个信号给进程，它不可能知道是否进程在忽略信号。这样它就会假定进程正在处理信号因而邮寄信号并唤醒进程，如果进程发现被它忽略的信号唤醒，它就会简单地返回睡眠。这种错误唤醒会导致不必要的上下文切换，浪费处理时间。最好内核能识别并丢弃被忽略的信号而不唤醒进程。

最后，则于 SVR2 实现缺乏临时锁定信号的机制，它会延迟传递直到解锁。它也缺乏作业控制支持，此处进程组将被挂起并重起以便控制对终端的访问。

4.4 可靠的信号

以上的问题最先是在 4.2BSD 中得到解决的，4.2BSD 提出了一个可靠的灵活的信号管理框架。4.3BSD 中加人一些改进，但是基本的方法没有改变。同时，AT&T 在 SVR3 中提供了自己的可靠的信号处理机制 [AT&T 86]。这一版本的 SVR3 与 BSD 不兼容，功能也没有 BSD 的那么强大。SVR3 仅保持和 SVR2 实现的兼容性。SVR3 和 4.2BSD 使用不同的方法去解决同样的问题，它们分别有自己的系统调用集合来行使自己的信号管理功能，这些调用在名字和语义上都有所不同。

POSIX.1 标准试图通过定义一套符合 POSIX 标准的系统都必须实现的函数标准集来为这种混乱的局面理出头绪。这些函数既可以用系统调用实现又可以用库例程实现。根据这些要求，SVR4 引入了一套符合 POSIX 标准，同时又与 BSD 和 AT&T 以在版本兼容的新接口。

本节首先介绍可靠信号的基本功能。然后简要地介绍 SVR3 和 4.3BSD 的接口，最后，详细地看看 SVR4 的信号接口。

4.4.1 主要特性

所有可靠的信号处理实现机制提供某些公共的方法。这包括：

·永久的信号处理程序 信号处理程序要在信号发生之后仍然保持安装，而不需要显示地再进行重新的安装。这样就消除了唤起信号处理程序和重新安装之间的时间窗口，这样在窗口之间的信号就不能中止这个这个进程。

·屏蔽(masking)一个信号可以被暂时的屏蔽(术语屏蔽(masked)和封锁(blocked)是同义的，在指信号的时候可以互换使用。)如果产生一个被正在封锁的信号时，内核将记住它，但是并不立即向进程发送。当进程解除对这个信号的封锁时，信号将被发送和处理。这使得程序员能够保护临界区代码不能被某些信号中断。

·睡眠进程 不管这个进程是否运行，进程的有关信号的一些处理信息对内核是可见的(保存在 prco 结构中，而不是在 u 区中的)。因此，如果内核对一个处于可中断睡眠状态的进程产生一个信号的话，且进程不能忽略(Ignoring)或阻塞(Blocking)这个信号，那么内核就不需要唤醒这个睡眠进程。

解锁和等待(unblock and wait)系统调用 pause 在信号到来之前一直阻塞进程。可靠的信号机制提供另一个系统调用 sigpause 自动地解除对一个信号的屏蔽,并且在进程接收到一个信号之前阻塞它。如果在系统调用发出之前没有被屏蔽的信号已经挂起(pending)，这个调用将立即返回。

4.4.2 SVR3 的实现

SVR3 包括以前章节提到的所有特性。然而，它的信号机制的实现有一些严重的缺点。为了清楚地明白这一点，先让我们看着使用系统调用 sigpause 的一个例于。

假设一个进程已经声明了一个信号处理程序去捕获信号 SIGQUIT。当信号被捕获时，信号处理程序置全局标志。过一段时间，它将检查这个全局标志是否被设置，如果没有，它将等待它被置。这个检查和随后的等待一起构成了一段临界区代码。如果信号到达是在检测(check)之后但是在等待这个信号之前这段时间内，那么信号将丢失，进程将一直等下去。因此，当测试标志时进程必须屏蔽信号 SIGQUIT。如果当进入等待状态时信号被屏蔽，那么这个信号将不能被传递。因此，我们需要一个原子调用能够解除对信号的屏蔽和阻塞进程。系统调用 sigpause 就提供这种功能。例子 4.2 的代码说明了 SVR3 中信号的工作机制。

例子 4-2 使用 Siggpause 等待信号

```
int  sig_received = 0;
void handler (int sig)
{
```

```
    sig_received ++;
}
main( )
{
    sigset(SIGQUIT, handler);
  /* Now wait for the signal, if it is not already pending  */
    sighold (SIGQUIT);
    while(sig_received == 0) /* signal arrived */
            sigpause(SIGINT);
  /* signal has been received, carry on  */
          ...
}
```

这个例子解释了 SVR3 中信号的一些特点，系统调用 sighold 和 sigrelse 允许对信号阻塞和解除阻塞。系统调用 sigpause 原子地对一个信号解除阻塞并让一个进程进人睡眠状态去等待一个不能被忽略或封锁的信号的到来。sigset 系统调用指定一个水久的信号处理程序，当信号发生时，这个信号处理程序不会被重新设置为缺省，老的 signal 调用被保留以便向前兼容。由 signal 指定的信号处理程序不是永久的。

SVR3 中的这个接口仍然有些不足[Stev 90]。重要的是，系统调用 sighold sigrelse sigpause 每次只能处理一个信号。SVR3 中不能原子的对多个信号进行阻塞和解除阻塞。在例子 4-2 中如果那个信号处理程序被多个信号同时使用，将没有令人满意的途径来对临界区进行编码。我们可以每次封锁一个信号，但是系统调用 sigpause 不能原子的将所有的信号解除阻塞然后去等待。

SVR3 也缺少对作业控制的支持，也没有像自动重新执行原子系统调用等手段。这些特性及许多其他的特性在 4BSD 框架中提出。

4.4.3 BSD 信号管理

4.2BSD 是最早提供可靠的信号机制的。BSD 所提供的信号机制 [Leff 89] 比 SVR3 的信号机制在功能上有了很大的改进。大多数的系统调用使用一个信号屏蔽参数。系统调用对这个 32 位(每个信号一位)的屏蔽参数进行操作。通过这种方式一个系统调用能够对多个信号进行操作。sigsetmask 系统调用指定将被封锁的信号的集合。sigblock 系统调用住这个集合中添加一个或多个信号。同样，BSD 实现的 sigpause 原子地设置一个新的封锁信号的屏蔽，并让进程睡眠去等待信号的到来。

4.2BSD 中用系统调用 sigvec 取代了 signal。像 signal 一样，sigvec 为每一个信号设置了一个处理程序。除此之外，它还能指定一个和这个信号相关的屏蔽参数。当信号产生时，内核在调用处理程序之前会设置一个新的阻塞信号的屏蔽参数，它是当前屏蔽参数的联合(U-nion)。这个屏蔽参数由 sigvec 和当前信号来指定。

因此，一个处理程序运行时，当前的信号要被阻塞。这样在处理程序完成之前，被处理信号的第二个实例不会被传递。这些话义与那些包括信号处理程序的典型情况非常接近。因为信号处理程序木身也通常是临界区代码。处理程序运行时常要求阻塞额外信号。当处理程序返回时，被封锁信号的屏蔽参数恢复到以前的值。

BSD 的另一个重要的特性就是能够在一个分离的堆栈上处理信号，现在我们考虑一个进程能够管理它自己的堆栈，它可以在自己堆栈溢出时为产生的 SIGSEGV 信号设置一个处理程序。通常这个处理程序将在同一个堆栈(已经溢出)上运行，从而进一步产生 SIGSEGV 信号。如果信息处理程序能在分离的堆栈上运行，这个问题就可以解决。其他一些应用程序，比如

用户级上的线程库，也将从分离的信号堆栈的使用中受益。系统调用 sigstack 指定一个信号处理程序可以在其上运行的分离的堆栈。因为内核并不知道这个分离堆栈的边界，因此必须由用户负责保证推栈的足够大。

BSD 也引进了其他一些信号，包括一些专门用于作业管理的信号。一个作业是一组相关的进程，通常形成了一个流水线。一个用户可以从一个终端会话(session)同时运行多个作业。但是只能有一个为前台作业，前台作业允许读或写终端，后台作业想要访问终端时它将得到信号，典型的情况下将挂起进程。Korn shell(ksh)和 C shell(csh)[Joy 80]使用作业控制信号来管埋作业，通过向前台或后台作业发送信号，挂起或是恢复这些作业。4.9.1 小节中将更详细的介绍作业控制。

最后，4.3BSD 可以自动重新启动由信号中断的慢系统调用。慢系统调用包括对字符设备，网络连接和管道的 read 和 write 操作，此外还有 wait，waitpid 和 ioctl。当这些调用被信号中断后,它们会自动在处理函数返回后重新启动而不是返回 EINTR 而异常终止。4.3BSD 增加了 siginterrupt 系统调用，它可以有选择性地对每个信号独立地使用或不使用这个功能。

BSD 的信号接口非常强大和灵活。最大的缺点就是和最初的 AT&T 接口(甚至和以后发行的 SVR3接口)的不兼容性。这种不兼容迫使第三方厂商开发了各种各样的库来满足这两大阵营。最后，SVR4 引入了符合 POSIX 标准的接口，同时又与以往的 System V 系统和 BSD 系统的语义兼容。

4.5 SVR4 信号机制

SVR4 提供了一组系统调用 [UNIX 92]，它是 SVR3 和 BSD 信号的功能的超集。同时，它也支持老的不可靠的信号。这个集合包括：

· sigprocmask(how,setp，osetp);

使用参数 setp 去修改阻塞信号的屏蔽参数。如果参数 how 是 SIG_BLOCK，那么 setp 参数与现有的屏蔽码相或(or 操作)。如果 how 是 SIG_UNBLOCK，那么参数中的信号从现有的屏蔽中解除阻塞。如果 how 是 SIG_SETMASK，那么用 setp 代替当前的屏蔽码。调用返回时，osetp 含有修改之前的屏蔽码的值。

· sigaltstack(stack,old_stack)；

指定一个新的堆栈来处理信号。当安装时信号处理程序需特别要求使用替换栈。其他的信号处理程序使用缺省的堆栈。返回时，参数 old_stack 指向以前的替换栈。

· sigsuspend(sigmask);

设置封锁信号屏蔽 sigmask，并让进程进入睡眠状态，直到一个传来不能被忽略或封锁的信号。如果对这样一个信号解除封锁，调用立即返回。

· sigpending(setp);

返回时，setp 包含挂在进程的信号集合。这个调用不修改任何信号状态，只是简单地用来获得信息。

· sigsendset(procset,sig);

是 kill 的增强版本，发送信号 sig 到被 procset 指定的一组进程。

· sigaction(signo,act,oact);

指定对信号 signo 的处理程序，与 BSD 的系统调用 sigvec 相似。参数 act 指向一个 sigaction 结构，这个结构包含信号的设置(SIG_IGN，SIG_DFL 或处理程序的地址)、信号相关的屏蔽码(和 BSD 的 sigvec 调用的屏蔽码相似)、及一个或多个下列标志。

SA_NOCLDSTOP 当子进程挂起时不产生 SIGCHLD。

SA_RESTART 如果被这个信号中断则自动重新启动系统调用。

SA_ONSTACK 如果已经有一个被 sigaltstack 指定的堆栈，则在替换栈上处理信号。

SA_NOCLDWAIT 同 SIGCHLD 一起使用——当调用进程的子进程终止时，不让系统产生僵尸进程(见 2.8.7 小节)。如果这个进程接下来调用 wait，它将睡眠直到所有它的子进程终结。

SA_SIGINFO　为信号处理程序提供附加的信息，用来处理硬件异常等等。

SA_NODEFER　当一个信号的处理程序在运行时，不会自动封锁这个信号。

SA_RESETHAND　在调用处理程序之前将重新把动作设置为缺省。

SA_NODEFER 和 SA_RESETHAND 提供了和最初的不可靠信号实现的向上兼容性。

在所有情况下，参数 oact 返回以前安装的 sigaction 数据。

兼容接口

为了提供和老版本 UNIX 的兼容性，SVR4 也支持 signal，sigset，sighold，sigrelse，sigignore 和 sigpause 系统调用。系统不需要代码级兼容性(binary cornpatibilty)可以把这些调用作为例程库来实现。

除了最后一组函数外，这些系统调用直接和 POSIX.1 的函数在名字上，调用语法和语义上相对应。

4.6 信号机制的实现

为了高效地实现信号机制，内核必须维持 u 区和 proc 结构中的一些状态。在这里讨论的 SVR4 的信号机制实现和 BSD 的信号很相似。主要区别在一些变量和函数的名字。u 区里用来正确地唤起信号处理程序的信息包括以下字段(fields)：

u_signal[]　每个信号的信号处理程序向量。

u_sigmask[]　信号屏蔽同每个处理程序有关。

u_sigaltstack　指向替换的信号堆栈。

u_sigonstack　在替换堆栈上处理的信号屏蔽。

u_oldsig　一组处理程序列出以前的，不可靠的行为。

proc 结构中的有关信号生成和传递(posting)字段，如下所示：

P_cursig　正在被处理的信号。

P_sig　挂起信号屏蔽。

P_hold　封锁信号屏蔽。

P_ignore　忽略信号屏蔽。

现在让我们看看内核如何实现和信号传递相关的各种函数。

4.6.1 信号生成

当一个信号产生时，内核检查接收进程的 proc 结构。如果信号被忽略，则内核不采取任何动作就返回。如果信号没有被忽略，那么内核把它放入挂起信号的集合 p_cursig 中。因为 p_cursig 仅仅是一个位掩码(bitmask)，一个信号占一位。因此内核不能记录一个信号的多个实例，也就是进程仅仅能够知道至少该信号的一个实例在挂起等待处理。

如果进程处于可中断睡眠状态，并且信号没有被封锁，则内核唤醒这个进程来接收信号。进一步地讲，如果是作业控制信号比如 SIGSTOP 和 SIGCONT 则直接挂起或恢复执行进程，而不是传递给进程：

4.6.2 信号传递和处理

当进程在系统调用或中断处理后要从内核状态返回时，它通过调用 issig()来检查信号。当它要进入可中断的睡眠状态之前或从可中断的睡眠中被唤醒后，进程也调用 issig()。函数 issig()检查 p_cursig 去找被置的位，如果任何一位被置，issig()检查 p_hold 去发现这个信号当前是否被封锁，如果没有，则 issig()在 p_sig 中保存这个信号并返回 TRUE。

如果有信号要发送，内核就调用 psig()来处理它。psig()在 u 区中检查与这个信号有关的信息。如果没有声明处理函数，psig()就采取缺省动作，通常就是终止进程运行，如果需

要调用处理函数，在存放阻塞信号的屏蔽字 p_hold 中增加这个信号。如果对这个处理函数设置了 SA_NODEFER 标志，则不在这个屏蔽字中加入该信号。与此相似，如果设置了 SARESETHAND，u_signal[]数组中的动作就被重新置为 SIG_DFL。

如果一个信号正在挂起，内核调用 psig()来处理它。psig()检查 u 区内维护这个信号的信息。如果没有被声明的处理程序，则 psig()采用缺省动作，通常是终止进程。如果一个处理程序被唤醒，通过添加当前信号及在 u_sigmask 项中指定的与该信号相关的任意信号可以改变 p_hold 的屏蔽信号掩码。同样地，如果指定了标志 SA_RESETHAND，则在数组 u_signal[]中的动作被重新设置为 SIG_DFL。

最后，psig()调用 sendsig()来安排使进程返回到用户状态并把控制传递给处理程序。sendsig()也保证当处理程序执行完毕时，进程能够恢复执行它在接收到这个信号前执行的代码。如果替换栈必须使用的话，sendsig()在替换栈上唤醒这个处理程序。因为必须知道堆栈和上下文管理的细节，所以 sendsig()的实现是和机器相关的。

4.7 异常

当程序遇到一个不正常的条件，通常是一个错误时，异常⑤发生了。这样的例子包括访问一个非法的地址和试图被零除，这将导致一个内核陷入。通常要产生一个信号来通知进程异常的发生。

在 UNIX 中，内核使用信号来通知用户这些异常。信号的类型和异常的本质有关。比如，一个非法的寻址异常可能产生一个 SIGSEGV 信号。如果用户对这个信号声明了一个处理程序，内核将调用这个处理程序。如果没有的话，缺省动作是终止进程，这允许每个程序安装自己的异常处理程序，有一些编程语言，如 Ada 有内置的异常处理机制，这些机制通过作为信号处理程序的语言库来实现。

异常也广泛地为调试器使用。被调试(跟踪)的程序在断点和执行完 exec 系统调用时产生异常。调试器必须拦截这些异常来控制程序，调试器也希望拦截其他选择的异常和由调试的程序产生信号，UNIX 中的系统调用 ptrace()使这种拦截成为可能。在 6.2.4 中将做进一步的讨论。

UNIX 处理异常的方法中有一些缺点。首先，信号处理程序和异常在相同的上下文中执行，这意味着信号处理程序不能向发生了异常时那样访问所有的寄存器的上下文。异常发生时，内核传递 ·‘ 些异常上下文给处理程序。传递的上下文的数量和指定的不同的 UNIX 有关，也和它运行的硬件有关。通常，一个线程(thread)必须处理两个上下文— 信号处理程序的上下文和异常发生的上下文。

第二，信号是为单线程的进程设计的。支持多线程的进程的不同 UNIX 版本发现很难和这种环境中的信号相适应。最后，由于系统调用 ptrace()的局限性，传统的基于 ptrace()的调试器仅能控制它的直接子进程。

4.8 Mach 中的异常处理

UNIX 中异常的局限性促进了在 Mach[Blac 88]中开发一种统一的异常处理方法。Mach 需要一种机制来和 UNIX 代码级兼容，同时也能适于多线程应用。这种方法也是基于 Mach 的 OSF/1 的一部分。

Mach 抛弃了信号处理程序和异常处理程序共用一个上下文的想法，UNIX 那样做只是因为异常处理程序需要在和异常发生的相同地址空间访问和执行。因为 Mach 是多线程的，它能通过在同一个任务的不同线程牛执行处理程序来达到这个目标(Mach 的线程和任务在 6.4 节中讨论，简单的一个任务拥有一组资源，包括一个地址空间，一个线程是一个在 task 中执行的上下文或控制点。传统的 UNIX 中进程由单线程的一个任务组成)。

Mach 识别两类不同的实体——victim 线程(引起异常的线程)和处理程序。图 4-2 描述两者的交互作用。首先，victim 线程引发一个异常，并通知内核它的发生，接下来它等待异常处理的完成。处理程序捕获异常，也就是从内核得到异常发出的通知。这个通知标识 victim 线程并指定异常的本质。接下来它处理异常并清除它，允许 victim 线程恢复执行。相反地，如果处理程序不能成功地处理异常，它将终止 victim 线程，

这些交互作用和对 UNIX 异常的流控制有些相似之处,除了处理程序是在一个分离的线程中执行。作为一种结果，操作 raise、wait、catch 和 clear 一起组成了 PRC。Mach 用 IPC 方法来实现它，这在 6.4 节中将详细描述。

处理一个异常包括两个消息，当 victim 线程引发一个异常时，它向异常处理程序发送一个消息并等待它应答。当异常处理程序接到这个消息时，它捕获到这个异常，并通过发送给 victim 线程一个应答信号来清除这个异常。当 victim 线程收到应答后，它能恢复执行。

4.8.1 异常端口

在 Math 中，消息能发送到端口。端口是一个受保护的消息队列，几个任务可能拥有对一个给定端口的发送权力(发送消息的权力),但是只有一个任务可以从这个端口接收消息。Mach 把一个任务同一个异常端口联系起来，对那个任务中的每一线程也有一个端口与其相联系。这就提供了两种方式来处理异常，对应两种异常应用——错误处理和调试，因为错误处理程序通常只影响 victim 线程，所以它和线程联系在一起，每一个线程有一个不同错误处理程序。处理程序的端口登记为线程的异常端口。当一个新的线程被创建时，它的异常端口被初始化为 NULL 端口，表示开始时线程没有错误处理程序。

一个调试程序通过登记其一个端口为调试任务的异常端口和一个任务联系起来。这个

图 4-2 Mach 的异常处理

调试程序作为一个单独的任务运行,它对那个端口具有接收权力(接收发送到那个端口的消息的权力)。每个任务从它的父任务那里继承它的异常端口，这样就使调试程序能够控制它调试的任务的所有后代任务。

既然一个异常既可以使用任务，也可以使用线程的异常端口。我们需要一种方法来解决这个冲突。为了做到这一点，我们发现线程异常端口是用于错误处理程序，错误处理程序对调试器(debugger)来说应该是透明的。举个例子，一个处理程序可以用零替代运算结果末简单地处理浮点下溢错误。调试器通常不介意这类异常，它通常只希望截获不可恢复的错误，因此，如果安装了一个错误处理函数，Mach 将先于调试器调用这个函数。

当一个异常发生时，它将被发送到线程的异常端口。因此唤起错误处理程序的异常对调试程序并不可见。如果安装的错误处理程序不能成功地清除异常，则这个异常继续发送到任务的异常端口(因为错误处理程序是任务中的一个线程，因此它能够访问 victim 线程的任务异常端口)。如果两个处理程序都不处理这个异常，内核将终止这个 victim 线程。

4.8.2 错误处理

当 victim 线程引发(raise)一个异常时，发送到任务或线程异常端口的初始化消息包含应答端口(用来识别引起异常的线程和任务)和异常的类型。当对这个异常进行处理完后，处理程序向应答端口发送一个应答消息。发生了异常的任务对这个应答端口具有接收的权力，victim 线程一直在等待读这个应答，当这个应答消息到来时，victim 线程接收它并恢复正常的执行。

因为异常处理程序和 victim 线程是同一任务中的线程，它们共享地址空间。所以异常处理程序可以通过使用 thread_get_state 和 thread_set_state 来访问 victim 线程的寄存器上下文。

Mach 提供对 UNIX 的兼容性，UNIX 的信号处理程序希望在引起异常的那个线程相同的上下文内被唤起执行。这与 Mach 的使用一个分离的线程去处理错误的思想相对立。Mach 通过

使用一个系统唤起的错误处理程序来调和这种差距。当一个异常发生时，这个异常的 UNIX 信号处理程序也已经安装了，一个特殊的消息发送到特殊的系统调用的处理程序。这个处理程序修改 victim 线程，这样当 victim 线程恢复执行时，信息处理程序会被执行，接下来它清除这个异常，导致 victim 运行和处理那个信号，这个应用程序负责在信号处理程序完成后恢复堆栈。

4.8.3 调试器的交互

一个调试器(Debugger)通过注册一个端口来控制一个任务。它把这个端口当作任务的异常端口并对这个端口拥有接收权。当任务中的一个线程有不能被它的错误处理程序清除的一个异常，内核将往这个端口发送一个消息，这样调试器(Debugger)将收到这个消息。这个异常只停止 victim 线程，任务中的所有其他线程继续执行。调试器可以在需要的时候使用系统调用 task_suspend 来挂起整个任务。

Mach 提供了几种方法，调试器可用来控制任务。调试器可以使用 vm_read 和 vm_write 访问 victim 线程的地址空间，或者通过 thread_get_state 或 thread_set_state 来访问 victim 线程的寄存器上下文，调试器也可以挂起或恢复执行一个应用程序，或用 task_terminate 来终止一个应用程序。

Mach 的 IPC 机制是位置无关的。也就是说，消息可以发送到相同的机器或远程的机器的端口。一个特殊的用户级任务 netmsgserver 扩展 Math IPC 机制，使它对网络是透明的。它为所有的远程端口分配代理(proxy)端口，接收向它们发送的消息，并且在网络上转发这些消息，这些对发送者都是透明的。这种机制就允许调试器能够像控制本地任务那样控制网络中的任何一个节点。

4.8.4 分析

Math 的异常处理机制解决了摆在 UNIX 面前的许多问题。它也比 UNIX 系统更加健壮，并提供了 UNIX 中不可得的功能。Mach 中的一些主要的优点有：

·调试器不仅只能控制它的直接子进程，只要具有必要的许可它可以调试任何任务。

·调试器可以附在运行的任务上⑥。它通过注册它的一个端口为任务的异常端口来实现这一点。它也可以通过重新设置任务的异常端口为以前的值来使调试器与任务分离。这个端口作为调试器和目标任务的唯一的连接手段，内核对调试不提供特殊的支持。

·网络上扩展的 Mach IPC 机制允许分布式调试程序的发展。

·Math 有一个独立的错误处理线程。能够完全分开处理程序和 victim 线程上下文，并允许处理程序访问 victim 线程的所有上下文，

·多线程的进程被完全处理。只有引起异常的线程才被挂起，其他的线程不受影响。如果几个线程都引起异常，则每一个线程都产生一个独立的消息，并被独立的处理。

4.9 进程组和终端管理

UNIX 提供了进程组的概念来控制终端访问和支持登录会话(login session)。在不同版本的 UNIX 中，这些机制的设计和实现有很大的不同。本节先回顾一下基本的概念，接下来介绍一些重要的实现。

4.9.1 基本概念

进程组 每个进程属于一个进程组。进程组用进程组 ID 来标志。内核使用这种机制来向一个进程组中的所有进程采取某些动作。每个进程组有一个组长，它的进程 ID(PID)和进程组 ID(GID)是相同的。通常一个进程从它的父进程那继承 GID，一个进程组中所有其他的进程都是组长(leader)的后代。

控制终端 每个进程都可以有一个控制终端。这个控制终端通常是创建进程的登录终端。同一个进程组中的所有进程共享这个相同的控制终端。

文件/dev/tty 特殊文件/dev/tty 是和进程的每一个控制终端联系起来的。/dev/tty 的设备驱动程序所做的只是把所有的请求送到合适的终端。比如，在 4.3BSD 中，控制终端的设备号保存在 u 区的字段 u_ttyd 中。因此，该终端的读操作可以这样来实现：

```
(* cdevsw[major(u. u_ttyd)].d_read)(u.u_ttyd, flags);
```

这样，如果属于不同登录会话的两个进程，它们都打开了/dev/tty，它们将访问不同的终端。

控制组　每一个终端同一个进程组联系在一起，这个进程组叫做终端控制组，它由终端保留的 tty 结构中的 t_pgrp 字段来标识⑦。键盘产生的信号，如 SIGINT 和 SIGQUIT 发送到终端控制组的所有进程。也就是说，这些信号被发送到那些 p_pgrp 等于终端的 t_pgrp 的所有进程。

作业控制　这是 4BSD 和 SVR4 中提供的一种机制，通过作业控制可以挂起或恢复执行一个进程组，并控制它对终端的访问。作业控制外壳程序如 csh 和 ksh 提供控制字符(典型的如 ctrl-z)以及命令 fg 和 bg 来访问这些特性。终端驱动程序通过禁止不在终端控制组的进程读或写终端来提供其他的控制。

在最初的系统实现中，进程组模型仅仅代表登录会话并不提供作业控制机制。4BSD 把每一个 shell 命令行同一个新进程组联系起来(因此，通过 shell 管道连在一起的进程属于同一个进程组)，这样它代表了作业的含义。SVR4 通过引进会话这个抽象概念来统一这些分歧和不兼容的问题。下面我们将看看这三种实现方法并分析它们的优缺点。

4.9.2　SVR3 模型

在 SVR3(和早期 AT&T 版本)中，进程组表现出终端登录会话的特点。图 4-3 描述了 SVR3 中如何解决了终端访问。以下是它的重要特性：

图 4-3　SVR3 UNIX 中的进程组

进程组　在系统调用 fork 中，每一个进程继承它的父进程的进程组 ID(GID)。改变这个进程组的唯一方式就是通过调用 setpgrp，它将改变调用者的进程组为调用者的 PID，因此调用者变为新组的组长。这样它接下来用 fork 创建的子进程将加入到这个组中来。

控制终端　终端属于终端控制组。当一个进程形成一个新的组时，它将失去它的控制终端。因此，这个进程打开的第一个终端(不是控制终端)，将变成它的控制终端。该终端的 t_pgrp 将被置为这个进程的 p_pgrp。所有的子进程，将从组长那里继承这个控制终端。没有两个进程组拥有相同的控制终端。

典型情景　进程 init 为每一个在/etc/inittab 文件中列出的终端创建(fork)一个子进程。这个子进程调用 setpgrp 变成了进程组的组长，接下来它运行(execs)程序 getty，将产生一个登录提示符，并等待输入。当一个用户输入其登录名后，getty 执行(execs)程序 login，要求用户输入密码并验证密码。最后，执行(exec)登录外壳程序。因此登录外壳程序是 init 的直接子进程，它也是一个组长。通常，其他进程不创建自己的组(除了登录会话的守护进程外)。因此，所有属于登录会话的进程将属于同一个组。

终端访问　SVR3 中没有对作业控制的支持。无论在前台还是在后台，所有有一个打开终端的进程能平等的访问它。这些进程的输出将随机混合出现在屏幕上，如果 n 个进程同时读该终端，那么哪一个进程将能读到输入的特定行则完全是一种偶然。

终端信号　在键盘上产生的信号如 SIGQUIT 和 SIGINT 被送到终端控制组的所有进程，通常也就是这些信号发送到登录会话中的所有进程。这些信号实际上是仅仅针对前台进程的，因此，如果外壳程序创建了一个进程并让它在后台运行，那么外壳程序将让这类后台进程忽略这些信号。同时它也重定向这些进程的标准输入为/dev/null。因此这些进程也不能用描述符来读终端(它们打开其他描述符去读终端)。

和终端分离　当终端的 t_pgrp 字段被置 0 时，终端就要和它的控制组分开。这将发生在当没

有进程打开这个终端的情况下或进程组的组长(通常是登录进程)退出时。
进程组组长的死亡　组长(leader)是进程组终端的控制进程，它负责为整个进程组来管理终端。当组长死亡时，它的控制终端将从进程组中失去联系(终端的 t_pgrp 被置为 0)。进一步，这个组中的所有其他进程被发送一个 SIGHUP 信号，它们的 p_pgrp 字段被置 0。因此它们不再属于一个进程组(它们成为孤儿进程)。
实现 进程的 proc 结构的 p_pgrp 字段含有进程组标志(GID)，u 区中有两个字段和终端有关，u_ttyp(指向控制终端的 try 结构)，以及 u_ttyd(控制终端的设备号)，try 结构的 t_pgrp 字段包含终端的控制进程组。

4.9.3　局限性

SVR3 进程组框架有以下一些局限性[lenn 86]：

·对一个进程组来说没有办法关闭它的控制终端和分配另一个终端。
·尽管进程组是按登录会话来设计模型的，但是当一个登录会话从它的控制终端失去连接后，没有一种方法能够保存这个会话。在理想的情况下，我们希望这样一个会话能永久存在一个系统中，这样在晚一些时候它能连接到另一个终端上，同时保存它的状态。
·通过控制终端没有一个一致的方式来处理载波丢失。也就是说，控制终端是否能保持分配给它的这个组，并能重新和这个组连接的语义可以根据它们实现的不同而不同。
·对于进程组中的不同进程访问终端，内核不进行同步。前台进程和后台进程可以以一种不规则的方式访问终端。
·当进程组组长终止时，内核向这个组内的所有进程发一个 SIGHUP 信号。忽略这个信号的进程甚至它被分到另一个组后，仍然可以访问控制终端，这将引起一个新的用户可从这样的进程接收非请求输出，或更糟糕，这个进程能读新用户输入数据，从而可以引起安全泄漏。
·如果一个不是登录进程的进程调用 setpgre，它将与控制终端失去连接。虽然可以继续通过现存的文件描述符访问这个终端，但是这个进程不受这个终端的控制，将不能收到 SIGHUP 信号。
·SVR3 中没有作业控制方法，比如没有在前台和后台之间移动进程的能力。
·一些程序像终端仿真器，它打开的设备不是它的控制终端，就没有方法从那些设备收到载波丢失通知。

4BSD 解决了这些问题中的一些，下一节我们介绍在 BSD 用到的方法。

4.9.4　4.3BSD 中的进程组和终端

在 4.3BSD 中，一个进程组代表登录会话中的一个作业(也叫做任务)，一个作业就是相关进程的一个集合，关于终端访问，它们被当作一个单位实体来被控制。基本的概念在图 4-4 中解释。

图 4-4　4.3BSD UNIX 的进程组

进程组　一个进程从它的父进程继承 GID，一个进程可以通过调用 setpgrp 来改变它自己的 GID 或其他进程的 GID(这必须得到允许，调用者必须拥有其他进程所有权或者是超级用户)，4.3BSD 的 setpgrp 调用接受 2 个参数，即目标进程的 PID 和分配给它的新的 GID。这样新的 GID 就分配给了它。因此，4.3BSD 中一个进程可以除去组中的领导权或是加入到另一个无关的组中。实际上，进程组可以没有组长。
作业　作业控制外壳程序如 csh，典型地为每一个命令行创建一个新的进程组，而不管这个命令是在前台执行还是在后台执行。因此，作业通常由一个进程或多个通过管道连接的进程组成⑧。这些进程的后代仍然在相同的组内。
登录会话　在 4.3BSD 中，单个登录会话可以生成几个进程组(作业)，这些进程组同时活动。它们共享相同的控制终端，终端的 tty 结构的 t_pgrp 宇段总是含有前台作业的进程组。
控制终端　如果一个 GID 为 0 的进程打开一个终端，则这个终端成为该进程的控制终端，并

且该进程加入终端的当前控制组(进程的 p_pgrp 被置为终端的 t_pgrp)。如果这个终端目前不是另一个组的控制终端，那么该进程首次成为进程组组长(group leader)(进程的 p_pgrp 和终端的 t_pgrp 都设为进程的 PID)。进程 init 的直接子进程(所有的登录外壳进程)初始时 GID 为 0。不仅如此，只有超级用户能重新设置一个进程的 GID 终端访问　前台进程(终端的当前控制组，可以从 t_pgrp 获得)总是拥有对终端的不可阻挡的访问权。如果一个后台进程想读终端，终端驱动程序发一个信号 SIGTTIN 到终端进程组的所有进程。缺省时，SIGTTIN 信号将挂起接收进程。缺省时也允许后台进程写终端。4.3BSD 提供一个终端选项(LTOSTOP 位被 TIOCLSET iotcl 管理)，引发信号 SIGTTOU 到想要写终端的后台进程，被 SIGTTIN 和 SIGTJOU 信号停止的作业可以通过向它们发送信号 SIGCONT 来恢复执行。

控制组　一个具有对终端读权限的进程可以使用 TIOCSPGPR ioctl 调用来改变终端的控制组(t_pgrp)为任意其他值。外壳程序利用这种方法在前台或后台之间移动作业。比如，一个用户可以通过让一个进程组作为控制进程组并且发送给它一个 SIGCONT 信号来恢复执行一个挂起的进程组，并把它移到前台。为此，csh 和 ksh 提供了 fg 和 bg 命令。

关闭终端　当没有进程打开终端时，终端同进程组失去联系，并且它的 t_pgrp 被置为 0，这通过终端驱动程序的 close 例程来完成。这个例程在终端的最后一个描述符关闭时被调用。

重新初始化终端行 4.3BSD 提供了一个系统调用 vhangup，典型地由 init 使用终止一个登录会话并启动另一个。vhangup 遍历打开的文件表，找到映射到这个终端的每项并令它不可用。它可以通过删除文件表项的打开模式来实现或是在支持 v 节点接口的实现中(见 8.6 节)，把 vnodeops 指针指向简单返回一个错误的一组函数。vhangup 调用终端的 close()例程，最后发一个 SIGHUP 信号到终端的控制组，这就是 4.3BSD 的解决方案，在登录会话终止后继续处理进程。

4.9.5　缺点

尽管 4.3BSD 作业控制功能强大而且灵活，但它有以下几个缺点

·没有登录会话的清楚明确地表示。最初的登录进程并不特殊，甚至它都可能不当组长。当登录进程终止时，一般并不发送信号 SIGHUP。

·没有一个负责控制终端的进程。因此，载波条件的丢失会发送一个信号 SIGHUP 到它的当前控制组，这个控制组甚至会忽略这个信号。比如，如果一个远程用户简单地切断了线路，他经由 modem 的连接会依然保持登录。

·一个进程可以改变终端的控制组为任意值，甚至是一个不存在的值。如果以后一个过程组用那个 ID 创建，它将继承那个终端而且可从那个终端接收信号。

·编程接口与 System 5 不兼容。

显然，我们需要一种办法能够保留登录会话和会话中任务的概念。下一节我们将看到 SVR4 中的会话体系结构以及它是如何解决这些问题的。

4.10　SVR4 会话的体系结构

SVR3 和 4.3BSD 模型的局限性可以归结为一个基本问题，单一的进程组抽象概念不能充分地表示一个登录会话和其中的作业。SVR3 在控制登录会话行为方面做了许多工作，但是它不支持作业控制。4.3BSD 提供强有力的作业控制方法，但是在隔离登录会话方面做得很差。

现代 UNIX 系统，如 SVR4 和 4.4BSD 已经解决了这些问题。通过分离但是有联系地表示会话和作业，进程组识别单个作业，一个新的会话对象可用宋表示登录会话。下面将描述 SVR4 会话的体系结构。4.10.5 小节描述 4.4BSD 的会话实现，它在功能上和 SVR4 相似，并且是 POSIX 兼容的。

4.10.1　目的(动机)

会话体系结构解决了在早期模型中的几个不是。它的主要目标包括：

·适当支持登录会话和作业抽象概念。

·提供 BSD 方式的作业控制。

·保持同以前 System 5 版本的向后兼容性。

·允许一个登录会话在它的生存期内同几个控制终端连接和断开连接(当然，在任意时刻，它只能有一个控制终端)，任何这样地改变都会透明地传给会话中的所有进程。

·让会话组长(创建登录会话的那个进程)，负责维护会话的完整性和完全性。

·允许终端访问只建立在文件访问许可的基础上。特别地如果一个进程成功地打开一个终端，只要终端仍打开，那么这个进程就能连续地访问它。

·消除以前实现的不一致性。比如，SVR3 中当一个进程组长在分配控制终端之前，它用系统调用 fork()创建子进程的情况下就会显出不规则来(anomaly)。这些子进程将从终端接收到信号 SIGINT，但它们不能通过使用/dev/tty 来访问终端。

4.10.2 会话和进程组

图 4-5 描述了 SVR4 中的会话体系结构[Will 89]。每一个进程同时属于一个会话和一个进程组。同样地，一个控制终端和一个会话及一个前台进程组(终端的控制组)联系起来。会话起的作用同 SVR3 中的进程组的概念相同。会话组长负责管理登录会话，并且把这个登录会话同其他会话分开，只有会话 leader 能够分配和释放一个控制终端。

一个进程通过调用 setsid 创建一个新的会话，setsid 按照进程组的 PID 值设置会话的 ID 和 GID。因此，调用 setsid 使调用者成为会话的组长和进程组组长。如果一个进程已经是一个组长，那么它不能成为一个会话组长，setsid 调用失败。

SVR4 的进程组具有 4.3BSD 进程组的基本特点，一般表示一个登录会话中的一个作业。因此，一个登录会话可能有几个同时活动的进程组存在。其中之一是前台组，它具有无限制地对终端的访问权(它是终端的控制组)。像在 4.3BSD 中一样，想要访问控制终端的后台进程被发送 SIGTTIN 和 SIGTTOU 信号(SIGTTOU 信号必须显式地被使能。如 4.9.4 节中描述)。

一个进程从它的父进程继承它的进程组，它可以通过调用 setpgid 或 setpgrp 来改变它的进程组。系统调用 setpgrp 和 SVR3 版本中的相似,它设置调用者的组和调用者的 PID 相同。这样使调用者成为进程组的组长。系统调用 setpgid 和 4.3BSD 中的 setpgrp 相似，但是

图 4-5 SVR4 会话结构

对这个操作加上一些重要的限制。语法如下：

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

它的功能是改变目标进程的进程组。目标进程用 vid 来标识，把它的进程组改变为指定值 pgid。如果 pgid 值为 0，那么进程织值被设为进程的 pid。因此使这个进程变为进程组的组长。如果 pid 值为 o，那么调用作用于调用进程本身。然而，有些重要的限制。目标进程必须是调用者本身或是调用者的子进程，并且子进程还没有调用 exec。调用者和目标进程必须同时属于相同的会话，如果 pgid 和目标进程的 PID 不相同(或为 o，具有相同效果)，它必须只能和相同会话中的另一个存在的组 ID 相同。

这样，进程能在会话内从一个进程组移到另一进程组。进程能够离开这个会话唯一的方法是通过调用 setsid 来启动一个新的会话，它是这个会话的唯一成员。一个进程组长可以通过移到另一个进程组来取消它的领导权。但是，这样一个进程，只要它的 PID 是任何其他进程的 GID(也就是说，这个取消进程领导权的组还没有空)，它就不能启动一个新的对话，这样就能防止混乱的情况，一个进程组和一个会话具有相同的 ID，但它不是这个会话的一部分。

同样的，一个终端的前台(控制)组可以只通过会话中控制终端的那个进程来改变，并且它只能被改为同一会话中另一个有效的组。这个特点被作业控制外壳程序用来在前台和后台之间移动作业。

4.10.3 数据结构

图 4-6 描述了管理会话和进程组的数据结构。调用 setsid 分配一个新的会话结构并且重新设置 proc 结构中的 p_sessp 和 p_pgidp 宇段。初始时，会话没有控制终端。

当会话组长第一次打开一个终端时(在进程变为会话的 leader 后),终端变为这个会话的控制终端，除非是调用者在 open 调用中使用了 ONOCTTY 标志。会话结构初始化指向这个终端的 vnode。这个 vnode 指向设备流的头部。

会话 leader 的子进程继承 p_sessp 指针，这样对于会话中对象的变化它立即可见。因此这些进程继承控制终端，而不管子进程创建后终端是否仍然打开。

4.10.4 控制终端

文件/dev/tty 文件充当控制终端的别名。控制终端的驱动程序通过在 proc 结构中查找会话指针并通过间接引用控制终端的 vnode 来解析任何对/dev/tty 的调用。如果释放控制终端，内核设置会话中对象的 vhode 指针为 null。任伺对/dev/tty 的访问将失效。如果一个进程直接打开了一个指定的终端(而不是打开/dev/tty)，那么甚至在终端同它当前的登录会话失去联系之后，它能够继续访问这个终端。

图 4-6 SVR4 UNIX 的会话管理数据结构

当一个用户登录到系统时，登录进程完成以下操作：

1.调用 setsid 变成组长。

2.关闭 stdin、stdout 和 stderr。

3.调用 open 打开指定的终端。因为这个终端是会话组长打开的第一个终端，因此它成为这个会话的控制终端。调用返回的描述符指向终端的实际设备文件。

4.在其他地方复制这个描述符。这样就不用 stdin、stdout 和 stderr 来指向实际设备文件。在复制后关闭原始的描述符，通过复制的描述符控制终端仍然打开。

5.打开/dev/tty 作为 stdin。复制它到 stdout 和 stderr，这将有效地通过假名设备，重新打开控制终端。这样会话组长和会话中的所有其他进程(继承这个描述符)只通过/dev/tty 访问控制终端(除非另一个进程显式地打开终端的设备文件)。

6.最后，关闭保存的描述符，移去任何直接同控制终端的连接。

如果终端驱动程序检测到一个断开的连接(比如，因为在一个 modem 线上的载波丢失)，它只发送信号 SIGHUP 到会话组长。这与在 4.3BSD 中发送信号到前台组相反，在 SVR3 中则是发送信号到控制组(会话)中的所有进程。在这种意义上，会话组长是一个可信任的进程。当它失去控制终端时能够采取正确的动作。

除此之外，驱动程序发一个 SIGTSTP 信号到前台进程组。如果这和会话组长不同，就防止前台进程访问终端时收到意外的错误。控制终端仍然分配给会话，这就可以给会话组长选择的机会。当连接重新建立时，会话组长可以重新连接终端。

一个会话组长可以断开当前的控制终端，并打开一个新的终端。内核会将这个会话的 vnode 指针设置为指向新的终端 vhode。这样一来，这个登录会话中的所有进程将透明地切换到这个新的控制终端，/dev/tty 提供的这种间接引用使得控制终端的转换很容易传播。

当会话组长终止时,它结束登录会话,通过设置会话的 vnode 指针为 null 释放控制终端。这样会话中的所有进程都不能通过/dev/tty 访问终端(如果它们已经显式地打开终端的设备文件，它们能继续访问终端)。终端前台组中的每一个进程都被发送一个 SIGHUP 信号，当前进程的所有直接子进程为 init 进程所继承接管。

4.10.5 4.4BSD 中会话的实现

SVR4 会话的体系结构适当地表示一个登录会话和会话中的一个作业，同时能维护同 POSIX 1003.1 以及 System 5 早期版本的兼容性。4.4BSD 和 DSF/1 中会话的实现本质上相似，并提供可比较的方法，在实现细节中它们同 SVR4 不同。

为了做对比,图 4-7 描述了 4.4BSD[Stev 92]中的数据结构 :一个重要的不同就是 proc 结

构不直接指向会话对象。实际上，它们指向进程组对象(struct pgrp)，反过来进程组对象又指向会话结构。

4.11 小　结

POSIX 1003.1 标准帮助我们把各种信号、控制终端的背道而驰的、互不兼容的处理方式结合到一起。最终的接口是健壮的并且能紧密的适应典型应用和达到用户的期望。

4.12 练　习

注意：某些问题对每一种 UNIX 变体都有不同的答案。学生可以回答他(她)熟悉的 UNIX 系统的相应答案。

1.为什么信号处理函数不能跨越 exec 系统调用?

2.为什么信号 SIGCHLD 的缺省动作是忽略?

3.当进程正在执行 fork，exec 或 exit 系统调用时，进程产生了信号，此时 WFC 怎样?

4.在什么条件下发送 kill 信号不会使进程立即终止?

5.传统 UNIX 系统的睡眠优先级有两个目的——决定一个信号是否能够唤醒睡眠的进程和判断唤醒后进程的调度优先级。这种方法的缺点是什么，现代系统是如何解决这个问题的?

图 4-7　4.4BSDUNIX 中的会话管理数据结构

6.持久性信号处理函数(在其被调用后仍然保持)有哪些缺点?有哪些信号不应该有持久性的处理函数?

7.4.3BSD 的 sigpause 调用与 SVR3 的 sigpause 有什么不同?试找出这样一种情况，了解上述区别会在这种情况非常有用。

8.为什么更希望由内核来重新启动被中断的系统调用，而不是由用户来重新启动?

9.如果在进程处理第一个收到的信号前它又收到了若干个同样信号将会怎样?对这种情况其他语义会更好吗?

10.假设一个进程有两个待处理的信号，并且每个都有了声明过的处理函数。内核如何保证进程在处理第一个信号后立即处理第二个信号?

11.在进程处理信号时又收到了另一个信号时会怎样?此时进程如何控制自己的行为?

12.进程何时应该使用 SVR4 的 SA_NOCLDWAIT 功能?何时不应该使用?

13.为什么异常处理函数需要引起异常的进程的完整上下文?

14.在(a)4.3BSD 和(b)SVR4 中哪个进程可以创建新进程组?

15.与 4.3BSD 终端和作业控制工具相比，SVR4 会话体系结构提供了哪些益处?

16.[Ball 8 8]描述了一个支持登录会话的用户级会话管理程序，它与 SVR4 会话体系结构相比有什么特性?

17. 当会话组长释放其控制终端时 SVR4 内核应该做什么?

18.SVR4 如何允许一个会话与它的控制终端重新连接?在哪种情况下这很有用?

①这就产生了另一个问题。如果使用一套信号接口的库被链接到使用另一套接口的应用上，这个程序很可能工作不正常。

②4.4BSD 中这个文件为 core.prog，这里 prog 是执行中接收到信号的应用程序名的前 16 个字符。

③在多处理器中，目标进程可能运行在另一个处理器上、而不是处理终端中断的处理器。此时，中断处理函数安排一次特殊的处理器问中断，这样目标进程就可以接收到信号了。

⑤本节介绍硬件异常，请不要将它与某些语言，如 c++中支持的软件异常相混淆。

⑥当前，大多数 UNIX 调试器都利用了/proc 文件系统，通过/proc 文件系统可以访问无关进

程的地址空间。这样调试器就可以方便地捕捉和释放运行中的进程。在 Math 系统的异常处理正在设计时，这个功能尚未被广泛了解。

⑦终端驱动程序为每一个终端维护一个 tty 结构。

⑧也可以很方便地将两个或多个无关进程组成一个进程组，只需通过分号，括号在同一行中执行多个 shell 命令即可。例如：

```
%(cc tenman.c; cp file1 file 2; echo done > newfile)
```