第11章　进程间通信

本章介绍有关进程间通信方面的内容。

系统中的进程和系统内核之间，以及各个进程之间需要相互通信，以便协调它们的运行。Linux系统支持多种内部进程通信机制（IPC），其中最为常用的是信号、管道以及Unix系统支持的System V IPC机制。

11.1　信号机制

信号(signal)机制是Unix系统中最为古老的进程之间的通信机制。它用于在一个或多个进程之间传递异步信号。信号可以由各种异步事件产生，例如键盘中断等。 Shell 也可以使用信号将作业控制命令传递给它的子进程。

Linux系统中定义了一系列的信号，这些信号可以由内核产生，也可以由系统中的其他进程产生，只要这些进程有足够的权限。你可以使用 kill命令（kill -l）在你的机器上列出所有的信号，一般如下所示：

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGIOT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	17) SIGCHLD
18) SIGCONT	19) SIGSTOP	20) SIGTSTP	21) SIGTTIN
22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO
30) SIGPWR			

进程可以屏蔽掉大多数的信号，除了SIGSTOP和SIGKILL。SIGSTOP用于暂停一个正在运行的进程，而SIGKILL使得正在运行的进程退出运行。进程可以选择系统的缺省方式处理信号，也可以选择自己的方式处理产生的信号。信号之间不存在相对的优先权，系统也无法处理同时产生的多个同种的信号，也就是说，进程不能分辨它收到的是1个或者是42个SIGCONT信号。

Linux使用进程中task_struct结构中的信息来实现信号。系统支持的信号数量受到处理器中字的大小的限制，例如， 32位的处理器可以支持 32个信号。当前等待处理的信号保存在task_struct中的signal字段中。每个进程的task_struct 中包括一个指向数组sigaction的指针，此数组是关于每一个进程如何处理可产生的信号的信息，其中有处理信号的子过程的地址，或者包括某种标志位，可以通知Linux系统进程希望忽略此信号或者进程希望 Linux系统本身处理此信号。进程可以通过系统调用修改系统缺省的信号处理程序。

除了内核和超级用户，并不是每个进程都可以向其他的进程发送信号。一般的进程只能向具有相同uid 和gid的进程发送信号，或向相同进程组中的其他进程发送信号。可以通过设置进程中task_struct结构中的signal字段的适当位来产生信号。如果进程没有屏蔽信号并且处于等待运行可中断状态，那么进程将被唤醒并进入到运行状态。这样调度程序将会在下一次调度时考虑运行此进程。

信号在产生时并不马上送给进程，信号必须等待直到进程再一次运行。每当一个进程从系统调用中退出时，系统都将检查进程的信号和屏蔽字段。如果有任何不屏蔽的信号，则可以立

写入进程使用标准的写入函数将数据写入到管道中。这些函数把文件描述符传递到进程的文件数据结构中，每一个文件数据结构都代表一个打开的文件，或者在此是一个打开的管道。Linux的系统调用使用描述此管道文件数据结构中的指针指向写入子进程，在 VFS索引节点存储有关此写入子进程的信息。

如果有足够的空间可以一次将所有的字节写入到管道中，并且读取子进程没有锁定管道，那么Linux系统将会使写入子进程锁定管道，并将从进程的地址空间中将写入的字节复制到共享数据页面中。如果管道没有足够的字节或者读取进程锁定了管道，那么当前进程在管道索引节点的等待队列中睡眠，这时调度进程可以调度运行其他等待运行的进程。因为睡眠的进程是可以中断的，所以它可以接收信号，当有足够的写入空间或者管道解锁时，读取进程可以唤醒睡眠的写入进程。

从管道中读取数据的过程和写入数据的过程十分类似。

系统允许进程使用无屏蔽的读取，也就是说，当没有数据可供读取时，或者管道锁定时，读取进程将会返回一个错误信息。这意味着，进程可以不必等待而继续运行。当写入和读取进程都完成对管道的操作时，系统将会放弃管道的索引节点和共享的内存页面。

Linux系统也支持命名管道，也叫做 FIFO。FIFO遵循先入先出的原则和一般的管道不同，FIFO不是临时的，而是文件系统的一部分。你可以用 mkfifo命令创建一个FIFO。只要有适当的权限，进程可以自由地使用 FIFO。FIFO的打开方式和一般管道的打开方式不一样。 Linux必须处理以下的情况，例如读取进程试图在写入进程打开 FIFO之前打开FIFO，或者在写入进程写入任何数据之前读取进程试图读取。除此之外， FIFO和一般管道一样使用相同的数据结构和操作。

11.2 System V IPC 机制

Linux系统支持三种类型的进程间通信机制，它们是信息队列、信号量和共享内存。这三种通信机制首先出现在Unix TM System V (1983)中,它们使用相同的授权方法。进程只有通过使用系统调用传递给系统内核一个唯一的参考标识符来存取这些资源。进程可以使用系统调用来设置System V IPC目标的存取权限。每个通信机制都使用目标的参考标识符作为资源列表的索引。这些标识符并不是真正的索引，而必须通过一定的转化才能生成实际的索引。

系统中所有代表System V IPC的目标数据结构中都包括一个ipc_perm结构，此结构中包含进程的用户和工作组标识符的拥有者和创建者，以及此目标的存取方式和 IPC目标关键字。其中的关键字作为一种定位System V IPC目标的参考标识符的方法。系统支持两类关键字：公共的和私人的。如果是公共关键字，那么系统中的任何进程，只要遵循权限检查，都可以利用它找到System V IPC目标的参考标识符。

11.2.1 信息队列

信息队列即一个或多个进程写入信息，同时一个或多个进程也可以读取此信息。 Linux系统中有一个信息队列的链表 msgque，此链表中每一个元素都指向一个叫做 msqid_ds的数据结构。msqid_ds结构保存着信息队列的有关信息。每当一个信息队列创建时，系统将从内存中分配一个新的数据结构msqid_ds，并将其插入到链表中。

每一个msqid_ds结构中都包含一个ipc_perm结构以及指向此队列中的信息的指针。另外，msqid_ds中还包括有关队列修改时间的一些信息，例如最后一次修改的时间等。它还包含两个等待队列：一个是写入进程的等待队列，另一个是读取进程的等待队列。

系统将把第二个进程挂起，直到第一个进程完成对数据文件的操作为止。第一个进程完成对数据文件的操作时，将会对信号量的值加 1，使得信号量的值从0又恢复到了1。现在，被挂起的进程将会被唤醒，它将再一次试图把信号量的值加1，而这一次它将会允许运行。

Linux系统使用数据结构semid_ds表示 System V IPC中的信号量目标，如图11-3所示。系统中的每一个信号量目标都代表一个信号量数组。系统中一个叫做 semary的指针数组保存着系统中所有的semid_ds的地址。每一个信号量数组都包括 sem_nsems，而每个sem_nsems又都描述了由semid_ds中的指针sem_base指向的sem结构。系统中所有允许使用信号量数组的进程可以通过系统调用来执行信号量的操作。系统调用可以指定很多的操作，每一个操作都由三个输入来描述：信号量索引、操作值和一系列的标志。信号量索引是信号量数组中的索引值，操作值是希望和当前信号量的值相加的数值。首先 Liunx将会检测所有的操作是否会成功。如果操作值和当前信号量值相加后大于 0，或者如果操作值和信号量当前值都为 0的话，操作就会成功。如果任何一个操作失败的话，并且当操作标志没有要求系统调用是无屏蔽时， Linux将会挂起此操作。如果进程被挂起， Linux就将把要执行的信号量操作的状态保存起来，并将当前的进程插入到等待队列中。系统需要在堆栈中建立一个数据结构 sem_queue，并把进程的状态保存到sem_queue中。新建立的数据结构 sem_queue将会被插入到等待队列的末尾（使用 sem_pending 和 sem_pending_last指针）。

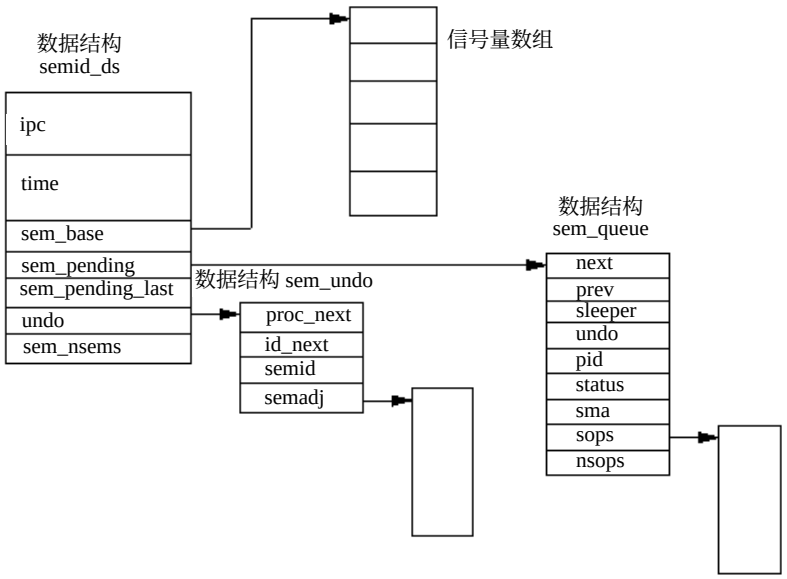

图11-3 信号量结构示意图

如果所有的信号量操作都取得成功，那么 Linux将会继续进行信号量数组中适当元素的操作。此时， Linux系统也必须检查任何等待的挂起进程是否可以进行它们的操作。如果有任何的进程可以进行信号量操作，那么它将从等待队列中移走数据结构 sem_queue，并对信号量数组中适当的元素进行信号量操作。

信号量操作可能会出现僵局。当一个进程修改了信号量以后，它开始运行信号量以后的操作，但在此期间，此进程由于崩溃或者由于被撤消而不能正常地离开操作区域，这时将会发生僵局。Linux系统通过维护信号量数组的调整链表来避免这个问题。其方法是，当信号量改变后，信号量将被调整到操作以前的值。系统把此调整保存在数据结构 sem_undo中，而数据结

到由shmid_ds中的指针指向的vm_area_struct数组中。vm_area_struct结构使用vm_next_shared和 vm_prev_shared指针将vm_area_struct链接起来。

当进程第一次存取共享虚拟内存的其中一个页面时，就会产生页面错误。当 Linux系统恢复页面错误时，系统将会查找描述页面错误的 vm_area_struct结构。vm_area_struct结构中包含指向这种共享虚拟内存错误的处理程序的指针。共享内存错误处理程序将会进一步在内存页面表链表中查找此shmid_ds结构的入口，以便了解共享虚拟内存的页面是否存在。如果此共享内存的页面不存在，系统将为其分配一个物理页，并同时为其创建一个新的页面表入口。此入口既会插入到进程的页面表中，也会保存到 shmid_ds中。这就意味着当下一个试图存取此页面的进程发生页面错误时，共享内存处理程序将会使用这个新创建的物理页。所以，第一个存取共享内存页面的进程使得系统创建共享内存页面，而以后其他进程存取此共享内存页面将导致此页面添加到进程的虚拟地址空间中。

当进程不希望再使用共享内存时，它将从共享内存上脱离。只要还有其他的进程在使用此共享内存页面，则进程的脱离只会影响到该进程本身。进程的 vm_area_struct结构将从 shmid_ds 结构中移走，进程的页面表也将被更新，其中共享的虚拟内存将变成无效的区域。当共享内存上的最后一个进程从共享内存中脱离时，共享内存页面所占用的物理内存将会被释放，共享内存的shmid_ds 结构也将会被释放掉。

如果共享内存没有被锁定到物理内存中，那么处理的情况将会比较复杂。在这种情况下，共享内存页面有可能被交换到系统的交换文件中。有关内存交换的过程请参考第 10章“ 内存管理 ”。