第 9 章 System Ⅴ IPC

Unix 从开发的早期就提供了管道的机制，管道在同一机器的两个进程间的双向通信方面工作的相当出色。后来，BSD（Berkeley Software Development）的 Unix 版本又提供了通用的套接字 socket，它用来在不同机器的两个进程之间进行通信（或者是同一机器的）。

Unix System V 版本增加了被视为一体的三个机制，现在它们被统称为 System V IPC。像管道一样，这些机制都可以用于同一机器上的进程间通信，不过与管道和套接字不同的是，System V 的 IPC 特性使得同一机器上的许多进程之间都可以互相通信，而不是仅限于两个进程。而且，管道——不是套接字——还有一个更大的限制就是两个通信中的进程必须相关。它们必须有一个建立管道的共同祖先进程——通常情况下，一个进程是另一个的父进程，或者这两个都是为它们建立管道的父进程的子进程。System V IPC 像套接字一样使得进程间通信（IPC）不需要有共同的继承关系，只需要一个经过协商的的协议。

组成 System V IPC 的三个进程间通信机制是：消息队列、信号量和共享内存。

消息队列

System V 的消息队列（message queues）是进程之间互相发送消息的一种异步（asynchronously）方式，在这种情形之下，发送方不必等待接收方检查它的消息——即在发送完消息后，发送方就可以从事其它工作了——而接收方也不必一直等待消息[①]。对消息进行编码和解码是发送者和接受者进程的工作；消息队列的执行并不会给它们特别的帮助。这就形成了一个实现起来相对比较简单的通用机制，尽管是以增加应用程序的复杂度为代价来获得这种简明性的。

这里是一个可能发生在 SMP 机器上的简单的应用情景。运行在一个 CPU 上的调度程序把工作请求发送到一个特定的消息队列上。工作请求可能以各种形式出现：用来破译代码的一组密码、需要进行计算的在不规则图形里的象素范围、在一个原子系统里要更新的一部分空间，或者诸如此类的任务。与此同时，工作者进程在其它 CPU 上运行，只要它们空闲就从消息队列中检索消息，然后再把结果消息发送到另一个消息队列上去。

这种体系结构很容易实现，而且假定选择好了每个消息中被请求工作的粒度，就能极大的提高机器中 CPU 的利用效率。（还要注意的是，因为调度进程可能不用做许多工作，所以调度进程的 CPU 上大部分时间也可以运行一个工作者进程。）以这种方式，消息队列可以被用作是远程过程调用（RPC）的一种低级形式。

新消息总是加在队列的末尾，不过它们并不总是从排头移出；你将能够在本章中看到，消息可以从队列的任何地方被移出。在某个方面，消息队列与语音邮件类似：新消息总是在末尾，不过消息接收方可以从列表的中间接收（以及删除）消息。

消息队列概述

首先对消息队列进行介绍是因为它的实现最简单，不过它仍然体现出了几个所有三种

[①] 原文为：”the receiver doesn’t have to go to sleep if no messages are waiting.”，直译是：如果没有消息正被等待，接收方也不必进入休眠。

System V IPC 机制都具有的共同结构特征。

给进程提供了四种与队列相关的系统调用：

- **msgget**——一个不合时宜的名字：读者可能认为这会得到一个等待的消息。但实际它不会。调用者提供一个消息队列键标（key），如果存在一个队列，**msgget** 就用该键标为它返回一个标识号，如果没有队列，就用它为一个新的消息队列返回一个标识号。因此，**msgget** 所得到的不是一个消息，而是唯一标识一个消息队列的标识号。
- **msgsnd**——向一个消息队列发送一条消息。
- **msgrcv**——从一个消息队列中接受一条消息。
- **msgctl**——在消息队列上执行一组管理操作——检索关于它的限制的信息（比如队列所允许的最大消息数据量）、删除一个队列，等等。

Struct msg

15919：**struct msg** 代表在队列中等待的一个消息。它有如下成员：

- **msg_next**——指向队列中的下一个消息——当然假如这是最后一个消息就为 **NULL**。
- **msg_type**——用户指定类型编码；它的使用在本章讨论消息如何被接收时再进行分析。
- **msg_spot**——指向消息内容的开头。读者后面将看到，为消息分配的空间总是紧靠在 **struct msg** 的上边，因此 **msg_spot** 恰恰指向 **struct msg** 末尾之后的位置。
- **msg_stime**——记录消息被发送的时间。因为消息以先进先出（FIFO）顺序保存，所以队列中的消息拥有的 **msg_stime** 值就是单调非递减的。
- **msg_ts**——记录消息的大小容量（" ts " 是 " text size " 的缩写，尽管消息不一定非要是人们可以读懂的文本）。一条消息的最大容量是 **MSGMAX**，它在 15902 行定义为 4056 字节。推测一下，这应该是 4K（4096 字节）减去一个 **struct msg** 的结果。不过 **b** 只有 20 字节，因此还有另外的 20 字节有待说明。

Struct msqid_ds

15865：msqid_ds 代表一个消息队列。它有如下成员：

- **msg_perm**——说明哪一个进程可以读写该消息队列。
- **msg_first** 和 **msg_last**——指向队列中的第一个和最后一个消息。
- **msg_stime** 和 **msg_rtime**——分别记录消息被发送入队列的最后时间和消息从队列中读出的最后时间。（一项挑战：什么时候队列中最后一条消息的 **msg_stime** 成员不等于队列本身的 **msg_stime** 成员？至少有两个答案，但是你所掌握的信息现在只能得出一个——你将不得不仔细阅读代码以寻求另一个解答。）
- **msg_ctime**——上一次队列改变的时间——它被创建的时间，或是上一次利用 **msgctl** 系统调用设置参数被确信的时间。
- **wwait**——等待写消息队列的进程队列。因为消息发送是异步的，通常进程把一个消息写入消息队列后就可离开。但是，为了避免拒绝服务（denial-of-service）的攻击，队列有一个最大容量——若没有这个限制，一个进程就可以不断的向一个没有读者的队列发送消息，强迫内核为每个消息分配内存直至空间耗尽。因此，当一个队列达到其最大容量时，想要发送消息给该队列的进程必须等待，直到队列中有了空间容纳新的消息，或者发送消息的尝试被立刻拒绝为止（读者将看到，进程能够选择它所希望的执行方式）。**wwait** 队列保留那些决定等待的进程。

- **rwait**——与之类似，消息通常可以从消息队列中立刻读出。但是如果没有正等待被读的消息将怎么办呢？进程再次进行选择：它们要么立刻重获控制（用一个错误代码表示读消息失败）要么进入休眠等待消息到来。
- **msg_cbytes**——当前在队列中的所有消息的总字节总数。
- **msg_qnum**——队列中消息的总数。对于能够进入队列的消息数目没有明确的限制——这也是一个问题，本章随后还要进行解释。
- **msg_qbytes**——队列中允许存储的所有消息的最大字节数；把 **msg_cbytes** 和 **msg_qbytes** 进行比较来确定是否还有空间容纳新消息。**msg_qbytes** 缺省为 **MAGMNB**，尽管这个限制可以被有适当权限的用户动态地增加。

  MAGMNB 在 15904 行定义为 16384。有四个理由说明为什么这个界限被定的这样低。第一，实际上，你通常不需要把太多的信息包括在一个给定的消息中，所以这个界限并不是十分苛刻的。第二，如果消息发送方的速度远远领先于接收方，那么让消息能多包含些信息可能也没有意义——它们还将是接收方要费些时间才能得到的一大块数据。第三，这个每队列 16K 的界限可以与潜在的 128 个队列相乘，总计达 2MB。

  但是采用这个界限的主要原因还是为了避免先前提及的拒绝服务攻击。然而，没有什么能防止应用程序发送长度为零的（也就是空的）消息。**msg_qbytes** 不会被这样的消息影响，而且仍然要给消息头分配内存，因此拒绝服务攻击仍然是可行的。解决这个问题的一个方案是引入一个独立的、对允许进入队列的消息总数进行限制的界限；另一个方案是从 **msg_qbytes** 中减去整个消息长度——包括消息头。再一种解决方法当然是不允许有空消息，但这又将同兼容性相抵触。
- **msg_lspid** 和 **msg_lrpid**——最后消息发送方和最后消息接收方的 PID。

Msgque

20129：消息队列实现中的主要数据结构是 **msgque**，一个指向 **struct msqid_ds** 的指针数组。这些指针有一个是 **MSGMNI**（15900 行定义为 128），它等于 128 个消息队列的最大值。为什么不只是用一个的数组而要用一个指针数组呢？一个原因是为了节省空间：替代一个 128 个 56 字节结构体的数组（7168 字节，7K），**msgque** 是一个 128 个 4 字节指针的数组（512 字节）。在正常情况下，当很少的消息队列投入使用，这能够节约好几千字节的空间。在最坏的情况时，所有的消息队列都被分配了，最大的消耗也只是 512 字节。唯一会引发的真正缺点是附加了一层间接转换，这意味着速度要有少许损失。

主要消息队列数结构之间的关系如图 9.1 所示。

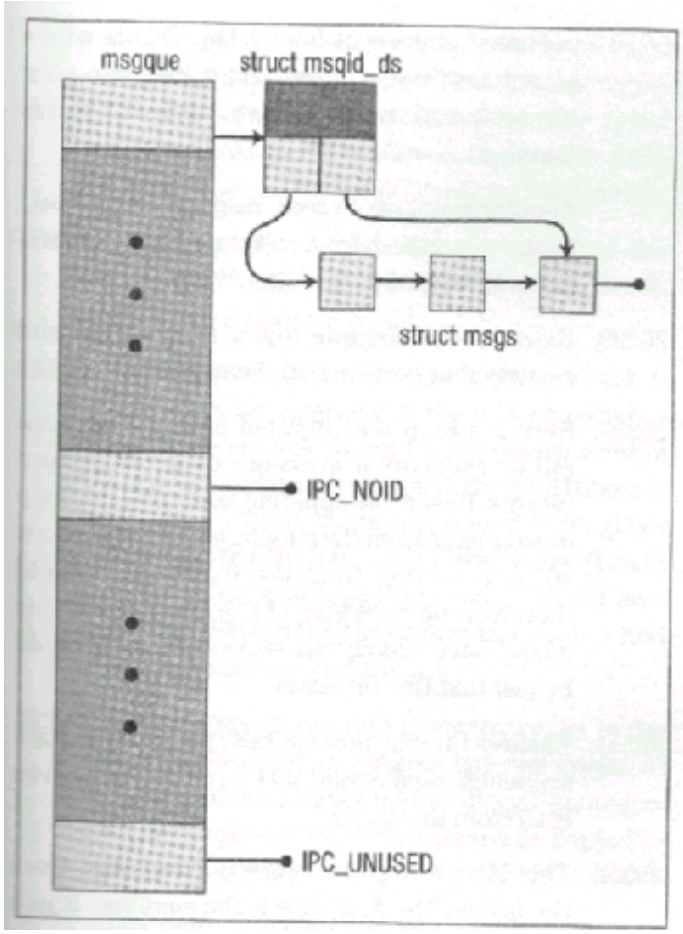

图 9.1 消息队列数据结构

Msg_init

20137：**msg_init** 用于消息队列实现时变量的初始化。它的大部分都是不必要，因为同样的变量已经在函数前面紧挨本段代码的声明中被初始化为同样的值了。

20141：然而这个把 **msgque** 的条目设置为 **IPC_UNUSED** 的循环是必要的。**IPC_UNUSED** 不在本书讨论之列，值为-1（能够更好的被映射为 **void***）；它代表一个没有使用的消息队列。**msgque** 条目可能接纳的其它特殊值是 **IPC_NOID**（也不在本书讨论之列）——这只是暂时的，也就是在消息队列被创建的时候。

Real_msgsnd

20149：**real_msgsnd** 实现 **sys_msgsnd** 的实质内容，即 **msgsnd** 系统调用。这里和内核的约定有一些偏差，该约定要在命名系统调用的“ 内脏函数[①] ”时使用一个“ do_ ”前缀。在 20338 行调用了 **real_msgsnd** 函数，在那里它处于 **lock_kernel/unlock_kernel** 函数对之中。（那两个函数在第 10 章中讨论——基本上，每次只能有一个 CPU 对内核加锁，这与 SMP 机器有关。）这是一种确保 **unlock_kernel** 得到执行的最佳方式——否则，**real_msgsnd** 复杂的流程控制将因需要在它退出时确保调用 **unlock_kernel** 而变得更加复杂。

[①] 原文为：”guts function”。”gut”通常的意思是“ 内脏 ”，这里是指本质的东西。

正如读者已经熟悉的，内核大多使用返回代码变量和 **goto** 语句来解决这样的问题。虽然它不能很好的适应每种情况，但是 **sys_msgsnd** 函数的方法更加清晰。例如，当一个函数必须获得多项资源，其中一些只有在以前所有资源请求都成功地被满足时才能提出请求，考虑这样可能引发什么样的后果。简单扩展的解决方法将需要大量函数——就像下边代码段所描述的：

P523 —1

很快，这样的代码将变得臃肿不堪，内核不这样做的原因就在于此。

20158：开始一系列条件判断。假如有了第一个测试，本行三项测试中的第二项就是不必要的——任何不能通过第二项测试的消息同样也不能通过第一项测试。虽然以后这种说法可能会不成立，假如 **MSGMAX** 的界限增加到足够高的话。（事实上，在写作本书时，完全消除 **MSGMAX** 界限的工作已在开展之中了。）

20164：消息队列标识号对两段信息进行编码：与之对应的 **msgque** 元素的索引在低端 7 位，一个序列编号（其作用随后就将讨论到）就位于紧挨这 7 位之前的 16 位里。现在所需要全部的就是数组下标部分。

20166：如果指定的数组下标处没有消息队列，或者正在创建一个，那么就没有消息可以进入队列。

20171：保存在消息队列中的序列编号必须和那个 **msgque** 参数里的编码相匹配。其思想是：在正确的数组下标处有一个消息队列并不代表它就是调用者所需要的消息队列。自从调用者引用一个队列之后，原先处于那个下标的消息队列可能已经被移去了而且在同一下标处创建了一个新的消息队列。16 位序列编号被周期性的增加，所以在同一下标处的新队列将有一个和旧队列不同的序列编号。（除非正好先创建了 65535 个其它的新队列，这是相当不可能的——或者是 131071 个其它的新队列，这就更不可能了。本章随后将对其进行解释，实际情况并非这样简单。）不管怎样，只要序列编号不匹配，**real_msgsnd** 就返回一个 **EIDRM** 错误来指示调用者所需要的消息队列已经被移出了。

20174：确保调用者有写消息队列的权限。类似的一个方法将在第 11 章详细介绍；在这里，可以简单的把它看作是类似于 Unix 文件权限应用的一个方法。

20177：检查如果提供的消息被写入队列，是否会超过队列所允许的最大容量。接下来一行代码再次检查同一件事，这显然是当代码从 2.0 系列的内核版本被转换过来时留下的一个编辑疏漏。在两次检查之间，曾经有过一些能够释放队列中的部分空间的代码。

20180：队列中没有空间。如果在 **msgflg** 里的 **IPC_NOWAIT** 位被设置了，这种情况发生时调用者就不会等待，这样的结果是返回 **EAGAIN** 错误。

20182：进程将要进入休眠状态。**real_msgsnd** 首先检查是否一条消息正在等待该进程。如果存在等待消息的话，就会用进程的休眠被该消息所中断的方式来处理它（进程可能已经休眠，就如随后所示的那样）。

20184：假如没有正在等待进程的信号，进程就进入休眠状态，直到有信号到达或移出队列中的一条消息时它才被唤醒。当进程被唤醒之后，它将再次向读列写入。

20190：为消息队列头（**struct msg**）和消息体分配足够的空间——正如前面所说，消息体将紧接在消息头后面存放。消息头的 **msg_spot** 直接指向该头部之后消息体开始的地方。

20196：从用户空间复制消息体。

20202：再次检查消息队列的合法性。**Msgque** 入口可能已经在 20184 行这个进程休眠时被其它进程修改过了，因此直到通过检查之前不能认为 **msg** 是有效的指针。

即便如此，这里看起来也有一个潜在的缺陷。如果在当前进程执行到这一步之前，该消息队列已被删除而另一个消息队列被设置在同一个数组下标的地方那又将怎样呢？在 UP 机器上是不会发生这种情况的，因为销毁消息队列的函数，**freeque**（20440 行），在销毁它之前将唤醒任何休眠于该队列的进程，而且在 **real_msgsnd** 完成之前 **freeque** 不会继续进行（本章后面将对 **freeque** 进行分析）。然而，在 SMP 机器上，这仍然是一个小小的隐患。

假如发生这种情形，**msgque[id]**将不是 **IPC_UNUSED** 或 **IPC_NOID**，但是 **msq** 指向的内存已经被 **freeque** 释放了，因此在 20203 行将废除无效的指针引用。

20209：填写消息头，将其入队，并更新队列自己相应的统计值（比如消息的总共大小）。注意只要有可能就推迟填写消息头的工作，所以假如在分配和当前阶段之间检测到错误时，这样就不会浪费时间。

20226：唤醒所有等待消息到达这个队列的进程，然后返回 0 以示成功。

Real_msgrcv

20230：同 **real_msgsnd** 函数一样，**real_msgrcv** 函数实现 **msgrcv** 系统调用。**Msgtyp** 参数含义灵活，这可以从在 20248 行开始的标题注释之中看出。**Struct msg** 的 **msg_type** 域在这里发挥作用：在该函数中它要与 **msgtyp** 参数相比较。

另一个与 **real_msgsnd** 相同的地方是 **real_msgrcv** 函数也是从 20349 行的 **lock_kernel /unlock_kernel** 函数对内调用的。

20239：从 **msgid** 提取 **msgque** 下标并确保在那个下标所指的空间中有合法的一项。

20262：这个 **if/else** 语句对从队列中选择一个消息。第一种情况最简单：它只需要得到队列中的第一条消息，使得 **nmsg** 或者为 **NULL** 或者指向队列的第一个元素。

20266：**msgtyp** 为正值，并且 **msgflg** 里的 **MSG_EXCEPT** 位（15862 行）被设置。**real_msgrcv** 函数沿着队列搜索第一个类型和 **msgtyp** 不匹配的项。

20272：**msgtyp** 为正值，但是 **MSG_EXCEPT** 位未被设置。**real_msgrcv** 函数沿着队列搜索第一个类型和 **msgtyp** 匹配的项。

20279：**msgtyp** 是负值。**real_msgrcv** 函数用最小的 **msgtyp** 编号来搜索消息，如果最小值比 **msgtyp** 的绝对值还要小的话。注意 20281 行在比较时使用<而不是<=，这样队列中消息的选择就不再有利于第一个消息了。这样的结果不仅令人满意——尽量遵循 FIFO 方式是一个好的策略——而且效率也稍有提高，因为这种方式减轻了赋值工作。如果比较采用<=，每个连接（tie）都将意味一次赋值操作。

20287：此时，如果有消息满足给定的标准，**nmsg** 就指向它。否则，**nmsg** 就是 **NULL**。

20288：即使找到一个合适的消息，它也有可能不被返回。如果调用程序的缓冲没有足够大的空间来容纳整个消息体，调用者通常会得到 **E2BIG** 错误。然而，假如 **msgflg** 的 **MSG_NOERROR** 位（15860 行）被设置，那么这个错误就不会被公布。（我找不出什么理由可以让一个应用程序去设置 **MSG_NOERROR** 标志位，我也找不出任何一个使用它的应用程序。）

20292：如果 **msgsz** 指定了多于消息体的字节数，**real_msgrcv** 函数就把 **msgsz** 减少到消息的实际大小。当程序执行过这里之后，**msgsz** 就是应该被复制到调用者缓冲区的字节数。

虽然此处代码的更加传统的写法有时比较慢，不过平均起来还是要更快一些：

```
if ( msgsz > nmsg -> msg_ts )
  msgsz = nmsg -> msg_ts;
```

20294：把选中的消息从队列中移出。队列是一个单向链表，不是双向链表，所以当不是队列中第一个的消息被移出时，**real_msgrcv** 函数必须先要在队列中进行循环以寻找它的前趋队列节点。

通过将队列转换为双向链接，前趋节点就能在恒定时间里被找到。这个改变将引入空间损耗（需要额外的指针），时间损耗（用来更新附加的指针），以及复杂度的提高（需要增加完成这些工作的代码）。尽管如此，那些代价都是很小的，而在被移出的消息处于队列中部的情况下，它们可以显著地提高速度。

不过实际情况中，大部分应用程序从队列中移出的都是第一个消息。其结果是，额外花费在管理 **msg _prev** 指针（假定我们这样称呼它们）上的时间通常被完全的浪费了。只在从队列中间移出消息时它才会有所补偿，但应用程序又很少这样做。结论是为了提高特殊情形时的速度而降低了普遍情况下的效率——这几乎总是一个坏主意。甚至于确实要移出队列中间节点的应用程序也不会等很长时间，因为通常消息队列很短，典型情况下最多也就是几十个消息而已，而且平均在循环进行到一半时就能找到选择的消息了。

因此，只有当消息队列有成百上千条消息而且应用程序又要移出队列中间的节点时，应用程序才会经历一次明显的速度减慢过程。考虑到这种情况的罕见程度，开发者的决定就是正确的。除此而外，如果一个应用程序真的陷入这种困境，而且它的开发者又不顾一切的需要这额外一点点速度——那么好吧，这就是 Linux，他们可以自己修改内核源代码以满足需要。

20305：处理移去队列中唯一节点的情况。

20308：更新消息队列统计值。

20313：唤醒所有等待写入这个消息队列的进程——也就是所有被 **real_msgsnd** 函数设置为休眠状态的进程。

20314：把消息复制到用户空间并释放队列节点（头部和体部）。

20318：返回正被返回的消息的容量大小——这对可变长度消息来说至关重要，因为应用程序的消息格式可能无法说明消息在哪里结束。

20320：没有符合调用程序标准的消息。接下来发生的操作将取决于调用者：如果调用者设置 **msgflg** 的 **IPC_NOWAIT** 位，那么 **real_msgrcv** 函数可以立刻返回一个失败错误。

20323：否则，调用者宁愿在没有可用的消息时进入休眠状态。如果一个信号正等待调用进程则返回 **EINTR** 错误；否则，调用者进入休眠状态直到一个信号到达或者别的进程写队列为止。

20329：永远不会执行到这里，但是编译器并不知道这一点。所以，这儿有一个假 **return** 语句，只是为了满足 gcc 的要求而已。

Sys_msgget

20412：因为 **sys_msgget** 的流程控制比 **sys_msgsnd** 和 **sys_msgrcv** 的要简单，所以就没有必要把 **sys_msgget** 的所有实质操作转移到一个独立的辅助函数上。尽管它确实有自己的辅助函数，本章随后还将对它进行分析。

20414：跟踪函数所需的返回值的 **ret** 不必初始化成**-EPERM**。**Ret** 会在函数的每一分支路径上被赋值，所以这一行的赋值就是多余的。然而，gcc 的优化器的聪明程度足以发现并消除这种无效赋值，因此这一点是没有意义的。

20418：特殊键值 **IPC_PRIVATE**（未包括在内——它的值是 0）表明调用者需要一个新队列，无论是否有其它具有相同键值的消息队列存在。在这种情况下，通过使用 **newque**

（20370 行）能够立刻创建该队列，随后我们还将对 **newque** 进行详细讨论。

20420：否则，**key** 唯一地标识出了调用者需要使用的消息队列。一般地，开发员选择键值时或多或少带有随机性（或者给用户一种办法来选择一个）而且希望它不会与任何运行中的应用程序的键值发生冲突。

这可能听起来耸人听闻，但是临时文件名也存在同样的问题——你只能期望没有其它应用程序选择了同样的命名方式。实际上，很少出现问题——**key_t** 是 **int** 类型的 **typedef**，所以在 32 位机上有超过 40 亿个可能值，而在 64 位机上超过了 9×10^{18}[①]个！这个键值空间容量的巨大程度有助于降低偶然冲突的机率。而且对于消息队列键值，或者对于文件，即使偶然发生冲突，一个授权方案也能进一步减小问题发生的可能性。

即便如此，难道我们不能做得更好吗？像标准 C 库函数 **tmpnam** 一样的函数可以极大地帮助产生能够保证在系统范围内唯一的临时文件名，但是却没有类似的办法能够产生一个消息队列键值而且保证它的唯一性。

如果对这个问题进行进一步研究的话，这些因素看起来应该是两个不同的问题。应用程序大体上并不关心临时文件的名字是什么，只要它不是正在使用的文件就可以。但是应用程序一般需要提前知道应把消息发送到哪一个队列中。如果一个应用程序动态的选择它的消息队列键值，那么它有时又莫名其妙地需要把被选择的键值告诉其它应用程序。（等价的，它可以发送 **msgid** 来代替键值。）而且，假如被涉及的应用程序已经有办法来像那样进行彼此之间的发送消息，那么它们还要消息队列做什么？因此，这可能不是一个值得解决的问题。如果一个应用程序需要一个非专有（nonprivate）队列的唯一键值，但是它对实际键值是什么并不太关心，那么它就可以通过尝试键值 1 来得到一个（记住 0 就是 **IPC_PRIVATE**）并且可以从那儿逐步尝试直到成功为止——那只需少量的工作，尽管不太可能需要。

无论如何，这一行使用 **findkey**（20354 行，后边讨论）来查找拥有给定键值的存在着的一个消息队列。

20421：如果键值没有被使用，那么 **sys_msgget** 就可以创建它。如果 **IPC_CREAT** 位没有被设置，则将返回 **ENOENT** 错误；否则，**newque** 函数（20370 行）创建队列。

20425：键值已被使用。如果调用者把 **IPC_CREAT** 和 **IPC_EXCL** 位都设置了，那么在这种情况下调用者就希望能够产生一个错误，因此它就得到了一个。（这是为了故意与 **open** 的 **O_CREAT** 和 **O_EXCL** 位恰好能够并列。）

若不加考虑，很难分辨出如原文所写的 **if** 判断和下面的等价形式相比那个更快：

P526—1

两种判断方式都检查是否那两个标志位都被设置了，但是，出于种种原因，你可能会期望任意一个快于对方。然而，结果是 gcc 为这两者产生同样的代码，至少是在优化编译的时候。（如果你对此有兴趣的话，它所选择的方案是把我建议的替代品直接进行翻译的结果——它的转换方案在内核中看起来就好像是同时测试两个标志位的代码。）这是一个相当棒的优化过程，而且也是我过去所没有期望得到的。

20428：否则，使用该健值，调用者将接受具有那个健值的存在着的队列。（这是最普遍的情况。）如果在期望的地方没有消息队列（考虑到 **findkey** 的执行，那应该是决不会出现的情况）或者调用者缺少访问它的权限许可，那么将返回一个错误。

20434：序列编号和 **msgque** 下标被编码在返回值里。这将成为调用者要传递给 **sys_msgsnd**、**sys_msgrcv**，以及 **sys_msgctl** 的 **msgid** 参数。

---

① 原文是：”9 quintillion”，“ quintillion ” (美、法) 百万的三次方, (英、德)百万的五次方，用科学技术法表示应该为：9.e+18。

这种编码方案有两个重要特征。更明显的一个特征是如何把序列编号部分和数组下标部分分离开来：因为 **id** 是一个索引 **msgque** 的数组下标，它只能具有最大到（但不包括）**msgque** 中含有元素的数目，即 **MSGMNI** 的值。通过把这个值与序列编号相乘，就可以使低位空出来以保存 **id** 了——它很像是一种标准的 **MSGMNI** 算法。这里还需要注意的是返回值永远不会是负值——这一点是非常重要的，因为 C 库执行时可能会把负的返回值当作是一个错误。因为当前值是 128，所以数组下标占据返回值的低端 7 位。序列编号是 16 位，因此只有 **ret** 的低 23 位可以被这次赋值设置成 1，而且所有高位应是 0。特别地，符号位是 0，所以 **ret** 是 0 或正值。

20437：不管 **ret** 被如何计算，它现在都将返回。

Sys_msgctl

20468：**sys_msgctl** 函数无疑是消息队列实现中最大的一个函数。这部分上是因为它要完成许多不同的功能——类似于 **ioctl** 函数，它是一个功能联系松散的函数聚合体。（顺便要说明的是，不要因为此处的混乱而责备 Linux 的开发者们；他们只是想要提供与 System V 那蹩脚的设计相一致的兼容性。）

msqid 参数指定了一个消息队列，**cmd** 指出 **sys_msgctl** 函数应该对它如何操作。很快读者就会看到，需不需要 **buf** 取决于 **cmd**，而且即使当它被使用时它的含义也将随情况的不同而不同。

20477：拒绝明显非法的参数。在不经常出现的，参数无效情况已经是不容置疑时，在调用 **lock_kernel** 函数之前执行本操作能够挽救不必要的内核锁定。（当然，流程控制将不得不相应作出调整——必须跳过 **lock_kernel** 函数）

20481：在 **IPC_INFO** 和 **MSG_INFO** 情况中，调用者需要有关消息队列实现的属性信息。它可能要用这些信息来选择消息容量，比如说，在最大消息容量较大的机器上，调用进程可以提高它自己在每个消息中发送的信息量界限。

所有清晰地在消息队列实现中定义那些缺省界限的常数都是通过 **struct msginfo**（15888 行）对象复制回来的。假如 **cmd** 是 **MSG_INFO** 而不是 **IPC_INFO** 时，还要包括一些额外信息，读者可以在 20495 行看到这一点，不过这两种情况在其它方面是相同的。

注意一下调用程序的缓存 **buf**，它被定义成了指向一种不同类型 **struct msqid_ds** 的指针。不过没有关系。复制是由 **copy_to_user** 函数（13735 行）完成的，它并不关心它的参数的类型，尽管当被要求向一块不可访问的内存写入时该函数也会产生错误。如果调用者提供了一个指向一块足够大空间的指针，**sys_msgctl** 函数将把请求的数据复制到那里；使得类型（或至少是容量）正确是取决于调用程序的。

20505：如果复制成功，**sys_msgctl** 函数返回一个附加的信息段，即 **max_msqid**。注意这种情况完全忽略了 **msqid** 参数。这样做有重要的意义，因为它返回了有关消息队列执行情况的总体信息，而不是某个特别的消息队列的具体信息。不过，就这种情况下是否应该拒绝负的 **msqid** 值仍是一个各人看法不同的问题。不可否认的是，即使没有使用 **msqid** 时也拒绝一个无效的 **msqid** 值一定能够简化代码。

20508：**MSG_STAT** 请求内核对给定消息队列持续作出的统计性信息——它的当前和最大容量、它的最近的读者和写者的 PID，等等。

20512：如果 **msqid** 参数不合法，在给定位指处没有队列存在，或者调用者缺少访问该队列的许可，则返回一个错误。因此，队列上的读许可不仅意味着是对入队消息的读许可，而且也是对关于队列本身“元数据（metadata）”的读许可。

顺便提及一下，要注意命令 **MSG_STAT** 假定 **msqid** 只是 **msgque** 下标，并不包括

序列编号。

20521：调用者通过了测试。**sys_msgctl** 函数把请求的信息复制到一个临时变量中，然后再把临时变量复制回调用者的缓存。

20533：返回“完全的”标识符——序列编号现在已经被编码在其中了（在 20520 行完成）。

20535：还剩下三种情况：**IPC_SET**、**IPC_STAT**，和 **IPC_RMID**。与读者迄今为止所见的那些情况有所不同的是，那些情况都在 **switch** 语句里被完全的处理了，而剩余的这三种在此仅进行部分处理。第一种情况，**IPC_SET** 只要确保用户提供的缓冲区非空，就将它复制到 **tbuf** 里以便后面函数的进一步处理。（注意拷贝操作之后在 20540 行对 **err** 的赋值是不必要的——因为在它使用之前的 20550 行，**err** 将被再次赋值。）

20542：剩余三种情况中的第二种，**IPC_STAT** 仅仅执行一次健全性检查——它的真正工作还在后边的函数体中。最后一种情形，**IPC_RMID** 在这个语句中不工作；它所有的工作都推迟到后边的函数中完成。

20548：这段代码对所有剩余的情况都是共同的，而且大家现在都应该对它比较熟悉了：它从 **msqid** 里提取出数组下标，确保在指定的下标处存在着一个有效的消息队列，并验证序列编号的合法性。

20559：处理 **IPC_STAT** 命令的剩余部分。假如用户有从队列中读出的许可，**sys_msgctl** 函数就把统计信息复制进调用者的缓冲区里。如果你认为这与先前 **MSG_STAT** 的情形非常类似，那你就是对的。这两者之间的唯一不同之处在于：正如读者所见，**MSG_STAT** 期望一个“不完全”的 **msqid**，而 **IPC_STAT** 却期望一个“完全”的 **msqid**（就是说包括序列编号）。

20572：复制统计数据到用户空间。如果按照如下方式重写这三行代码，那么运行速度或许稍快一些：

P527—1

毕竟，对于写入用户空间来说成功要肯定比失败更为普遍。基于同样的原因，**MSG_STAT** 情况下（始于 20530 行）的相应的代码如果被重写成以下形式也可能更快：

P527—2

或者，下边的一个甚至可能更快，因为没有一次多余的赋值操作：

P528—1

然而和直觉相反的是，我对所有这三种修改都作了测试，结果却发现是内核的版本执行起来更快。这必然与 gcc 生成目标代码的方式有关：显然，我的版本中的一条额外跳转要比内核版本的额外赋值所花费的代价高得多。（从 C 源代码来考虑额外的跳转并不直观——你不得不考察 gcc 的汇编输出代码。）回想前边章节所讨论过的，跳转会带来明显的性能损失，这是因为它们会使得 CPU 丧失其内在的并行性所带来的好处。CPU 的设计者们竭尽全力要避免分支造成的性能损失影响，不过很明显，他们并不总是成功的。

最终，对 gcc 优化器的进一步改善可能消除内核版本和我的代码之间的差别。每当两种形式逻辑相同而一个较快时，假如 gcc 能够发现这种等价并为两者生成同样的代码，那将非常令人愉快。不过这个问题是要比看上去难得多的。为了生成最快的代码，gcc 将需要能够猜测哪一次赋值最易发生——另一种情况则涉及了分支。（对 gcc 的最近版本所作的工作已为这样的改进打下了基础。）

20576：在 **IPC_SET** 情形里，调用者需要设置消息队列的某些参数：它的最大容量、属主，和模式（mode）。

20578：为了操纵消息队列的参数，调用者必须拥有该队列或者拥有 **CAP_SYS_ADMIN** 权

能（14092 行）。权能已在第 7 章中讨论过。

20584：把消息队列中最大字节数的界限提高到正常限制以上，这就类似于提高任何其它资源的硬界限一样，因此它也需要与之相同的权能，即 **CAP_SYS_RESOURCE**（14117 行）。资源限制在第 7 章已经讨论过。

20587：调用者应该被允许执行该操作，所以被选择的参数根据调用者提供的 **tbuf** 被设置。

20595：**IPC_RMID** 意味着删除特定的队列——不是队列中的消息，而是队列本身。假如调用者拥有该队列或者有 **CAP_SYS_ADMIN** 权能，这个队列就可以用 **freeque** 函数调用（20440 行）来释放。

20605：**cmd** 最终不是经过验证的命令中的一条，所以调用程序得到 **EINVAL** 错误。在这种情况下，在 20548 行所作的工作原本是可以避免的。假设我们要试图尽早检测无效的 **cmd**，通过删除 **switch** 语句里的 **default** 情况并把下列代码附加到函数第 20546 行的第一个 **switch** 后：

P528—2

这样就会改变函数的行为状态。当调用者提供了一个无效 **cmd** 和一个无效 **msqid** 时，它将得到一个与现在所得的不同的错误——有了这种改变之后，无效的 **cmd** 将先于无效的 **msqid** 而被检查出来。虽然有关 **msgctl** 的文档并没有诺许任何一种行为，但是这样我们就可以自由的来改变它。其结果能够少许提高这种无效 **cmd** 情形下的速度。

然而，要注意这种解决方案很不幸地需要在第一个 **switch** 开关处引入一个空的 **IPC_RMID case**。没有它，函数将错误的把 **IPC_RMID** 也当作一种无效 **cmd** 情况而抛弃掉。这个额外的 **case** 减缓了 **cmd** 合法这种正常条件下的速度——虽然不很严重，但情况的确如此。而且，正如你所知道的，用普遍情形的代价来换取特殊情形时速度的提高从来就不是一个良好的解决办法。因此还是原来的方式更好。

Findkey

20354：**findkey** 函数为 **sys_msgget** 系统调用（调用在第 20420 行）定位具有给定键值的消息队列。

20359：开始对 **msgque** 里所有可能被占据的单元槽进行循环。**max_msqid** 跟踪 **msgque** 里被占据的最大数组元素；在这里使用到了它，并且在很快就要提到的 **newque** 和 **freeque** 里将对它进行维护。若没有 **max_msqid**，这个循环将需要在 **msgque** 的所有 **MSGMNI** 个元素里反复进行，就算是只有前 5 个在使用也要如此。

20360：如果当前数组元素值是 **IPC_NOID**，那么就会在那里创建一个消息队列。这个消息队列可能具有正被搜寻的键值，所以 **findkey** 函数将等待该队列的创建工作完成。（当 20385 行的 **kmalloc** 调用使进程休眠时就会进入这种状态。）

20362：如果该 **msgque** 的项目是未被使用的，那么它明显不具有匹配的键值。

20364：若匹配的键值被找到，相应的数组下标就被返回。

20367：如果循环结束仍未找到匹配的键值，就返回-1 以示失败。

Newque

20370：**newque** 函数定位一个没有使用的 **msgque** 项目，并尝试在那里创建一个新的消息队列。

20376：循环 **msgque** 以查找未用的一项。如果找到了一项，就用 **IPC_NOID** 来标记它，控制随之跳转到 20383 行的 **found** 标记处。

20381：如果循环结束却没有发现未用的项目，**msgque** 就是满的。**Newque** 返回 **ENOSPC**

错误表示表里没有剩余的空间。

20384：分配一个 **struct msqid_ds** 来代表新的队列。

20387：如果分配失败，该 **msgque** 项目被设置回 **IPC_UNUSED** 标志。

20388：一旦发现有 **IPC_NOID** 就激活任何已经休眠的 **findkey**。

20391：初始化新队列。

20404：如果这个队列被建立在 **msgque** 中原来最高的已用单元槽之后，**newque** 就相应的增加 **max_msqid**。

20406：在 **msgque** 里建立新队列。

20408：唤醒每个可能一直在等待该队列初始化完成的 **findkey**。

20409：返回序列编号和 **msgque** 的数组下标。（创建一组宏来处理此处的编码和随后的解码不会有什么损害。）奇怪的是，没有在这里增加序列编号——它要由接下来讨论的 **freeque** 来完成。如果读者考虑一下，这里的决定是有一定道理的。你并不需要每个队列都有一个唯一的序列编号——你只是想让每次 **msgque** 元素被重用时有一个不同的序列编号，以便数组下标和序列编号二者的组合（combination）不可能重复而已。数组下标直到建立在该位置的队列被释放后才能重新使用，所以增加序列编号的操作也可以推迟到那个时候。

为了把这个含义说的更明确一些，一个序列编号是可以被两个 **msgque** 元素同时使用的。

Freeque

20440：我们将以 **freeque** 函数来结束这次内核消息队列实现的讨论，它的作用是删除一个队列并释放相应的 **msgque** 元素项。

20449：如果正在被释放的是最高的被使用项，**freeque** 函数将尽可能地减低 **max_msqid**。循环之后，**max_msqid** 将再次成为被使用的 **msgque** 项的最高下标值，或者在所有元素项都没有使用时变成 0。要注意的是如果 **max_msqid** 是 0，则 **msgque** 要么是空，要么就只有一个元素项。

20452：**msgque** 数组的元素被标识成为未使用，尽管此时 **struct msqid_ds** 还没有被释放（在 **msq** 里，**freeque** 函数仍然有一个指向该 **struct msqid_ds** 的指针）。

20454：假如有某个进程正等待读出或写入这个队列，必须警告它们该队列即将消失。这里的循环唤醒所有那些进程。每个正等着向该队列发送消息的进程将在第 20171 行知道被改变了的序列编号；每个等待从该队列里读取消息的进程也将在第 20254 行进行同样的工作。

20458：调用 **schedule** 函数（26686 行，在第 7 章讨论过）来赋予被唤醒了的进程运行的机会。有趣的是，被唤醒了的进程可能还没有得到 CPU 使用权——当前进程仍然有最大的优先权。假如这种情况发生，新近被唤醒的进程将不会从各自的等待队列中被移出；而 **freeque** 又会注意到这一点并继续重复以图再次唤醒进程。最终，执行 **freeque** 的进程会因耗尽它的时间片而将（CPU）让出给其它进程。在考虑了这一切之后，在调用之前明确设置当前进程的 **SCHED_YIELD** 标志（16202 行）可能是更好的方法，这样可以给其它进程更好的使用 CPU 的机会。

20460：没有被挂起的读者和写者，所以该队列和它的消息可以被安全的释放掉。

信号量

信号量（Semaphores）是一种对资源访问进行保护的方式。信号量在通常概念上的模型是指一种发送信号的标志（名称由此而来），但是我认为更好的象征是一把钥匙（key）。不要把它与我们已经讲过的整数类型的键值（key）搞混了——在这个类比中，我所指的意思是你的前门钥匙。

在最简单的情况下，信号量只是悬挂在一扇锁着的门旁吊钩上单独的一把钥匙。为了穿过这道门，你必须把钥匙从吊钩上拿下来；当你出来时再把钥匙重新放回吊钩之上。如果你到达时钥匙不再那里，你就不得不等待它的拥有者把它放回原处——假如你已决定要通过这道门，就必须这样。而作为另一种选择，如果无法立刻得到钥匙，你也可以因没有耐心等待而就此放弃。

上边描述了某资源每次只能由一个实体（entity）来使用的情形；在这种只有一把钥匙的情况下，信号量可以被看作是一个二元信号量（binary semaphore）。对于每次可以被多个实体占用的资源而言，信号量可被看作是计数信号量（counted semaphores）。这与前边一样，只不过是吊钩上悬挂了更多的钥匙而已。如果资源同时可供四个用户使用（或者假如有四个等价的可用资源，它们基本上是相同的），那么就有四把钥匙。依次可自然的进行类推。

进程使用信号量来协调它们的动作。比如，假设你正在写一个程序而且想保证每次在给定的机器上最多只有该程序的一个实例可运行。这方面的一个好例子是声音文件播放器——可能你不会想让它同时播放多个声音文件，因为其结果将是令人烦恼的一团糟。另一个例子是 X 服务器。当然偶尔也会有充分的理由使得在同一个机器上同时运行多个 X 服务器，但是对于一个 X 服务器来说禁止这样做也是很合理的，至少缺省的做法就是如此。

信号量提供了一种解决这个问题的方案。你的音响播放器、或 X 服务器，或是其它任何程序都可以定义一个信号量，检查该信号量是否在使用，若没有使用则继续运行。如果该信号量已被使用，则表明程序的另一个实例在运行之中——你的程序可以等待信号量被释放（音响播放器可能的行为），只是放弃并退出（X 服务器可能的行为），或者暂时继续其它工作稍候再试信号量。顺便说一句，这样一种信号量的用法由于显而易见的原因而通常被叫做相互排斥（mutual exclusion）；它的通用简称，互斥（mutex）将在内核源代码中反复出现。

锁文件是获得与二元信号量同样效果的更为普遍的一种方式，至少在某种情况下如此。锁文件更易使用，而且锁文件的一些实现可工作在网络上；但是信号量则不行。另一方面，锁文件在超出二元的情况时就不容易使用并推广了。但无论如何，锁文件都超出了本书的范围。

信号量和消息队列二者的代码是如此相似以至于没有必要再讨论 **sem_init**（20695 行）、**findkey**（20706 行）、**sys_semget**（20770 行）、**newary**（20722 行），以及 **freeary**（20978 行）了，因为它们几乎同它们所对应的消息队列部分是一样的。

Struct sem

16983：**struct sem** 结构体代表一个单独的信号量。它有两个成员：

- **semval**——如果是 0 或为正值，**semval** +1 就是仍然挂在这个信号量吊钩上的钥匙数目。若为负值，它的绝对值就比正等待访问它的进程数目大一。缺省的信号量是二元的，但是它们也可以通过使用 **sys_semctl** 变为计数型的；信号量的最大值是 **SEMVMX**（在 16971 行定义为 32767）。
- **Sempid**——存储最后一个操作该信号量的进程的 PID。

Struct semid_ds

16927：**struct semid_ds** 与 **struct msqid_ds** 相对应：它跟踪所有关于单独一个信号量以及在它上面所执行的一系列操作的信息。我们所感兴趣的、有别于 **struct msqid_ds** 中的成员如下所述：

- **sem_base**——指向一个 **struct sem** 数组——换句话说，指向一个信号量数组。如同单独一个 **struct msqid_ds** 可以包含多个消息一样，一个 **struct semid_ds** 也可以包含多个信号量——该数组中信号量总和被具有代表性地称为一个信号量集合（semaphore set）。然而与消息队列不同的是，被一个 **struct semid_ds** 所跟踪的信号量的数目并不在它的生存期里变化。数组的容量大小是固定的。这些数组中一个的最大长度是 **SEMMSL**，它在第 16968 行被定义为 32。数组的实际长度记录在 **struct semid_ds** 的 **sem_nsems** 成员中。
- **sem_pending**——跟踪挂起的信号量操作组的一个队列。信号量操作一有可能就立刻完成，正如读者所预期的那样，所以只有当操作必须等待时，这个队列才会增加节点。此成员与 **struct msqid_ds** 的 **rwit** 和 **wwait** 成员大致等价。
- **sem_pending_last**——跟踪上述同一队列的队尾。它并不直接指向最后一个节点——它指向一个指向最后节点的指针，这将有利于稍微提高后面代码的速度（尽管这给理解增加了难度）。（需要顺便提一下的是，我不知道为什么同样的思想没有被应用于消息队列。）
- **sem_undo**——当各个进程退出时所应该执行操作的一个队列。这将在后续章节中讨论。

Struct sem_queue

16989：**struct sem_queue** 结构体是单个 **struct semid_ds** 之上休眠着的操作队列中的一个节点。它有如下成员：

- **next** 和 **prev**——队列中的下一个和前一个节点。正如 **sem_pending_last** 一样，**prev** 是指向一个指向前面节点的指针的指针。读者将能够在本章后面章节中看到为什么系统要这样做的原因。**prev** 永远不会变成 **NULL**；在退化的情况里，即当队列为空时，**prev** 指向 **next**。
- **sleeper**——当某进程必须等待完成一个信号量操作时使用的等待队列。等待队列在第 2 章中介绍过。
- **undo**——一个将要撤销由 **sops** 所暗示的操作的操作数组——用另一种方式表示它就是 **sops** 的反转。
- **pid**——尝试完成这个队列节点操作的进程的 PID。
- **status**——记录一个休眠进程被唤醒的过程。
- **sma**——向后指向这个结构体 **struct** 所存在的 **sem_pending** 队列的 **struct semid_ds**。
- **sops**——指向这个队列节点所代表的一个或多个操作的一个数组；它永远不为 **NULL**。这个队列节点所描述的工作目的是执行 **sops** 里所有的操作。
- **nsops**——**sops** 数组的长度。
- **alter**——说明是否操作会影响信号量集合里的任何一个信号量。这个问题的回答看起来总是肯定的，但是要记住等待信号量变成 0（即成为可用）并不会影响信号量本身。

Struct sembuf

16939：**struct sembuf** 结构体表示在信号量上执行的单个操作。它有如下成员：

- **sem_num**——是 **struct semid_ds** 的 **sem_base** 数组的数组下标，该数组由这种操作适用的信号量构成。因为 **struct sembuf** 是 **struct sem_queue** 的一部分，而且 **struct sem_queue** 知道它与哪一个 **struct semid_ds** 相关联，这样就从不会出现该操作应使用哪一个 **struct semid_ds** 的信号量数组的疑惑了。在其它情况下，一个 **struct sembuf** 数组与一个索引 **semary** 的下标组成一对，这也蕴含了一个信号量数组。
- **sem_op**——要执行的信号量操作。通常，它的值是-1、0，或 1：-1 表示获得（procure）信号量（从吊钩上取走钥匙），1 表示交出（vacate）信号量（把钥匙重新放回到吊钩上），而 0 表示等待该信号量变成 0。除了这些值以外的值也是有用的，不过它们只是被翻译为获得或交出更多的信号量值而已——也就是说，取走或放回吊钩上更多的钥匙。（这段文字里的“获得”和“交出”可能看起来有点儿怪——无须担心；这只是普通的信号量术语。）
- **sem_flg**——可以修改操作执行方法的一个或多个标志位（在这个 **short** 里的每一个位）。

这些数据结构之间的关系如图 9.2 所示。

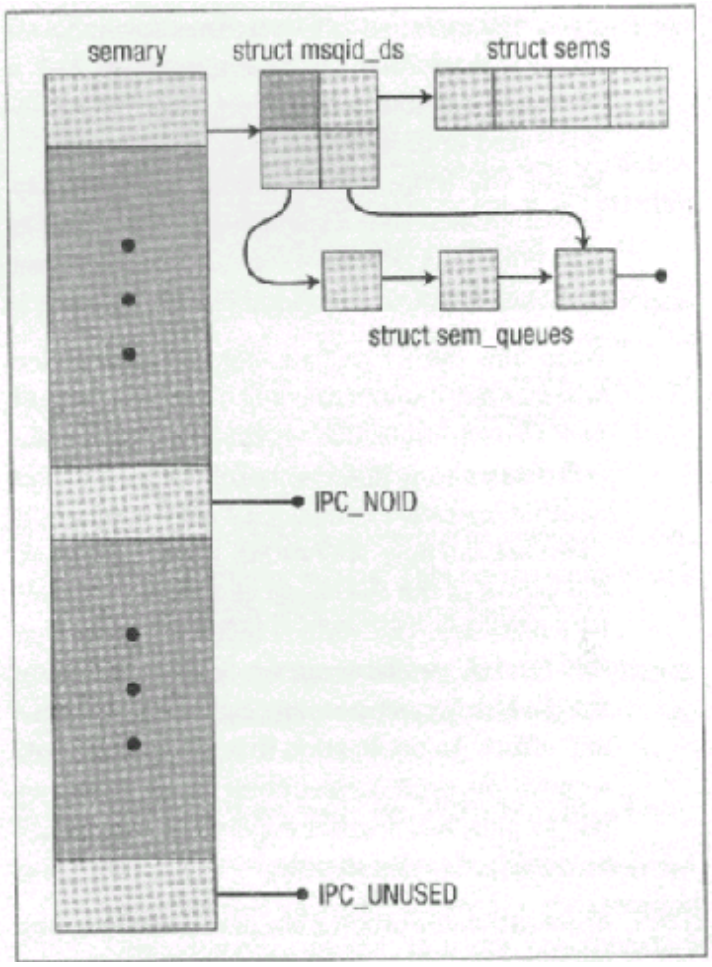

图 9.2 信号量数据结构

Struct sem_undo

17014：**struct sem_undo** 含有足够的撤销单个信号量的操作的信息。当一个进程执行信号量操作同时设置了 **SEM_UNDO** 标志位时就创建一个用来撤销该操作的 **struct sem_undo**。进程的 **struct sem_undo** 列表所包含的所有撤销操作在该进程退出时都会执行。熟悉设计模式（design patterns）的读者可能发现这是命令模式（Command pattern）的一个实例。

这个特性保证不管进程如何退出，都将自动为它执行相应的清理工作——这样一来，就不会意外的让其它进程空等一个永远也不会被释放的信号量了。（除非进程获得信号量后陷入死循环之中，但是避免这个问题不是内核的工作——在这种情况下，目的是要提供给进程正确的工作方法，而不是对其进行人工干预。）

struct sem_undo 有如下成员：

- **proc_next**——指向固有进程 **struct sem_undo** 列表里的下一个 **struct sem_undo**。
- **id_next**——指向与信号量集合相关联的 **struct sem_undo** 列表里的下一个 **struct sem_undo**。你所看到的是正确的，同一个 **struct sem_undo** 确实是同时存在两个不同的列表之中。读者将在本章的后面看到这两者的作用。
- **semid**——标识出这个 **struct sem_undo** 所归属的 **semary** 元素项。
- **semadj**——一个调整器的数组，这些调节将使用在和这个 **struct sem_undo** 相关联的信号量集合中的每个信号量上。这种结构所不知道的信号量在数组中有一个 0——并没有进行调整。

Sys_semop

21244：**sys_semop** 函数实现了 **semop** 系统调用。消息队列代码中没有直接同 **sys_semop** 函数对等的函数——它是 **sys_msgsnd**、**sys_msgrcv**，或者同时是两者的对应函数，这取决于你如何看待它。在任意一种情况下，它的工作都是在一个或多个信号量上完成一种或多种操作。它将自动尝试完成所有操作（即无需中断）。假如无法全部完成，它将不会执行其中的任何一项操作。

21254：同消息队列函数非常类似，这像是在比必要的时机稍微提前一些的时候就锁住内核。加锁也应该可以被推迟到第 21265 行左右再执行。

21255：参数的健全性检查。特别注意 **nsops** 受到 **SEMOPM** 的限制，它是可以被立刻尝试的信号量操作的最大数目。在第 16970 行它被定义为 32。

21261：把请求的操作描述从用户空间复制到一个临时缓冲区，即 **sops** 中。

21265：确保在指定的数组位置存在一项。正如读者所见，同消息队列代码的 **msgque** 对等的是 **semary**（20688 行）。还要注意的是数组下标和序列编号以与消息队列代码相同的方式被打包进了 **semid** 参数。当然这里应用的常量稍有不同——**SEMMNI** 在第 16967 行定义为 128（而巧合的是，**MSGMNI** 也是一样的值）。

21272：开始一个遍历所有特定操作的循环。首先检查在操作中给出的信号量数目是否超出范围，如果是的话就放弃它。但是令人奇怪的是，这里返回的失败信息是 **EFBIG** 错误（意思是“文件太大”）而不是 **EINVAL** 错误（“非法参数”）。尽管这也是符合文档规范的。

21275：记录设置了 **SEM_UNDO** 标志位的操作的数目。**undos** 只是一个标志——重要的是它是否为 0——因此，当条件满足时给它赋值 1（或任何非零值）将产生同样的效果。不过，内核的版本更快一点。而且因为循环重复的循环次数最多是 **SEMOPM** 次，

undos 就不可能被增加多次以至于回到原点再次变为 0。

21277：接下来的几个测试更新两个局部标志：**decrease** 和 **alter**。它们分别用来跟踪集合里的任何操作是否在减少某个信号量的值以及是否在修改一个信号量的值。直到循环结束之后，**alter** 才会在第 21282 行被计算出来——在循环里，它只是跟踪是否有操作增加信号量的值；这个结果与后边 **decrease** 里的信息结合起来最终决定是否发生了改变。

要注意这里的代码没有检查是否组合在一起的操作将彼此抵消——可能一个操作把某个信号量减 1，而另一个操作又会把它加 1。如果这仅有一个操作，那么从某种意义上讲，**decrease** 和 **alter** 标志的值将是很容易引起误解的。内核可以尝试着优化这种情况（并得到实现同样内容的更加精致的版本），不过相比较于所花费的时间和精力，这可能并不值得：一个愚蠢到执行这样一种奇怪得空操作的应用程序就应该这么慢，而一个聪明的应用程序则不应该这么愚蠢。

21285：确保进程具有在信号量上执行特定操作的许可。如果 **alter** 为真，那么修改信号量的进程就需要写许可；否则，它只是在等待一个或多个信号量的值变为 0，这样该进程就只需要读许可。

21291：包括某些撤销操作的一组操作。如果当前进程已经有了在退出时要在这个信号量上执行的一组撤销操作，那么新的撤销操作的数据就应该合并到那一组中去。这个循环查找存在的撤销操作集合，假如有，就使 **un** 指向它，若没有，则 **un** 为 **NULL**。

21295：进程还没有这个信号量集合的一个取消集（undo set），所以需要为它分配一个新的。在读者已经在消息队列代码中熟悉了的一段编码的实践经验之后，为撤销调节（**semadj** 数组）分配的空间将被安排在紧靠 **struct sem_undo** 本身之后，并作为同一分配的一部分。接着就填入 **struct sem_undo**。

21311：在提供的操作集合里没有撤销操作，所以 **un** 被设置为 **NULL**。

21313：调用 **try_atomic_semop**（20838 行，后边讨论）来尝试在单个的槽内执行所有操作。如果有任何引起变化的操作，**un** 即为非空；若失败，就需要利用它来在函数返回之前取消任何已经完成的部分操作。

21315：**try_atomic_semop** 返回 0 表示成功，负值表示错误。无论何种情况，控制流程都向前跳转到第 21359 行。

21321：否则，**try_atomic_semop** 返回一个正值。这表示此刻无法执行所有操作，但是该进程希望等待且稍后再试。一个局部 **struct sem_queue** 将首先被填写。

21328：代表修改信号量操作的节点被放在队列的末尾；代表等待信号量值归 0 的操作的节点位于队列的前边。在本章后边探究 **update_queue** 函数时（20900 行）读者将对这种做法的原因有所了解。

注意在挂起操作的队列中放置了一个局部变量——这很不寻常；这样的数据结构通常具有以堆形式分配（heap-allocated）的节点。在这种情况下这样做是安全的，因为节点在函数返回之前将被从队列中移出；而上下文转换部分将负责剩下的工作。或者进程也可先退出，此时由 **sem_exit**（21379 行）来负责进行收尾工作。

21333：开始一个反复尝试执行这些操作的循环，仅当所有要求的操作都成功完成或者发生一个错误时该循环才会退出。

21336：一直休眠到被一个信号中断或有某断点（point）被再次尝试为止。

21342：如果进程由于此刻具有成功的机会而被 **update_queue** 唤醒，则它重新尝试进行该操作。

21358：把这个进程从等待修改信号量集合的进程队列中移出。

21360：假如这个操作的集合改变了队列，那么某个其它进程所等待的条件可能就已经具备。

Sys_semop 调用 **update_queue** 来寻找并唤醒这样的进程。

Sys_semctl

21013：实现 **semctl** 系统调用的 **sys_semctl** 函数具有与 **sys_msgctl** 相似的许多共同之处。相应的，这里的讨论只涉及那些感兴趣的不同点，比如在 **sys_msgctl** 里没有对应部分的 **sys_semctl** 命令（command）。

21093：**GETVAL**、**GETPID**、**GETNCNT**、**GETZCNT**，以及 **SETVAL** 命令对单个信号量、而不是信号量集合进行操作，所以在这些情形里提供的 **semnum** 参数必须首先进行范围检查。若 **semnum** 在范围之内，**curr** 就指向相应的信号量。

21115：几乎是同样的命令集——**GETVAL**、**GETPID**、**GETNCNT**，以及 **GETZCNT**——涉及了对关于信号量的一段信息进行的阅读和计算。这里就完成那些工作。注意 **sempid** 成员的高位在第 21116 行被屏蔽掉了——通过后面的讨论你将知道这样做的原因。

21121：**GETALL** 命令请求这个信号量集合里所有信号量的值。和许多其它命令一样，此命令的工作并非在一处全部完成；稍后读者将见到其余的命令。

21126：**SETVAL** 命令把信号量的值设置成给定的值——当然，是在规定的限制内。同样地，此时只完成部分工作——主要是范围检查。

21142：**SETALL** 是 **SETVAL** 的一个普遍化结果，它设置集合中所有的信号量值。同 **SETVAL** 类似，在此只完成诸如范围检查一类的准备工作。

21173：**GETALL** 的剩余部分由此开始。

21175：确保进程有读取信号量值的许可。这里的许可检查与第 21112 行的相重复。

21177：把信号量值复制到局部数组 **sem_io** 里，然后再从那里将它们复制到用户空间。

21183：**SETVAL** 的剩余工作由此开始。

21187：因为信号量取得新值，所以任何有记录的为 **semnum** 信号量所进行的取消调节操作都将变为无效。这个循环通过把它们设置成 0 以使它们失去作用。

21189：把信号量的值设置成调用者提供的值，并调用 **update_queue**（20900 行）来唤醒那些等待该条件成立的进程。

21220：**SETALL** 的主要部分由此开始。

21224：所有信号量的值都被设置成了调用者提供的值。

21226：与集合内各个信号量相关的所有取消调节操作都被设置成 0。当信号量被设置为它已经拥有的值时，这并没有什么特别的地方——它也不应该有什么特别之处。如果调用程序需要为除了一个信号量之外的所有信号量都赋予新值，那么就不能通过设置那个信号量为原值的方法来欺骗它。取而代之的做法是，必须要对不应改变其值以外的集合中所有信号量施用 **SETVAL** 命令。

Sem_exit

21379：**sem_exit** 函数在消息队列代码里没有对应函数。它实现进程在退出时所要求自动执行的撤销操作。所以，它在进程退出时调用（23285 行）。

21389：如果进程的 **semsleeping** 成员非空，那么以下二者必有其一成立：要么进程正在某个 **sem_queue** 队列上处于休眠状态，要么它已经从该队列被移出但 **semsleeping** 还未更新。假如是前者，进程将被从休眠队列里移出。

21395：开始遍历当前进程的 **struct sem_undo** 列表。轮流对每个条目进行分析然后在循环的更新部分释放它们。

21397 如果对应于这个撤销结构体的信号量已被释放，就继续循环。**struct semundo** 的 **semid**

域可以被 **freeary** 设置成-1，本章随后将对其进行介绍。

21399：类似地，如果相应的 **semque** 项不再有效，则继续循环。

21406：与从消息队列中间移出一条消息的情形相当类似，这个循环遍历 **sma** 的 **struct semundos** 列表以找寻将被移出的前一个节点。当找到时，**sem_exit** 向前跳转到第 21413 行的 **found** 标记处。

21411：如果在 **sma** 的列表里没有找到撤销结构体，那么就发生了错误。**sem_exit** 显示一条警告消息并停止外层循环。这种反应看来有点过激，因为可能会有更多的撤销结构体能够依照这种处理方式进行释放。不应该因一个烂苹果就糟踏整整一桶苹果。尽管这几乎是“不可能发生”的情形，仅当内核逻辑错误才会导致其发生。我的推测是这样的，若检测到这样一个错误的话，剩下的数据就不再可信了。

21414：在 **sma** 的列表里找到了撤销结构体，**unp** 就指向一个指向其前驱的指针。接着把 **un** 从队列里移出。

21417：执行这个撤销结构体里对所有信号量的调节。

21427：像往常一样，调用 **update_queue** 以免被这个函数所执行的操作满足了唤醒某个休眠进程的条件。

21429：所有的 **struct sem_undo** 都已经处理过了——或者在 21412 行就检测到了错误并结束了循环。不管哪一种结果，当前进程的队列被设置成为 **NULL** 然后函数返回。

Append_to_queue

20805：把 **q** 附加在 **sma** 的 **sem_pending** 队列之后。这里的实现很紧凑；通常类似如下的形式：

P534_1

真正的优点在于内核的实现方式避免了潜在的代价昂贵的分支。通过使得 **sem_pending_last** 成为指向一个指向队列节点的指针的指针，而不仅仅是一个指向队列节点的指针，可能会部分的提高执行的效率。

Prepend_to_queue

20812 :把 **q** 附加在 **sma** 的 **sem_pending** 队列之前。由于 **sem_pending** 不是一个指针的指针，这种实现就同前面考虑过的简单的实现一样具有相同的形式。

Remove_from_queue

20823：这是 **struct sem_queue** 队列上的最后一个原语操作，它把一个节点从队列中移出。

20826：通过修改前一队列节点的 **next** 指针，部分解除 **q** 与队列的链接。

20828：如果有下一个节点，还要更新下一个节点的 **prev** 指针；或者假如这已经是队列的最后一个节点，就使用 **sma->sem_pending_last**。要注意的是没有非常清楚的代码被用于移出队列中唯一的节点——假设你还没有发现原因的话，这就值得你花些时间研究一下为什么这种情形下代码也可工作。

20831：把已移出节点的 **prev** 指针设置成 **NULL**，以便第 21350 和 21390 行代码能有效地发现该节点是否仍在队列之内。

Try_atomic_semop

20838：该函数上方的标题注释说明它被用于测试是否给定的操作集能被全部执行。该注释没有说明这些操作是否能够被全部被执行，通常情况下它们是能够被执行的。

20846：开始循环所有通过检查的操作并依次对其进行尝试执行。

20850：**sem_op** 为 0 表示调用程序希望等待 **curr->semval** 变为 0。因此，如果 **curr->semval** 不是 0，调用程序不得不阻塞（block），这意味着操作无法自动被执行（由于这个进程被阻塞时要完成其它工作）。

20853：调用程序的 PID 被暂时保存在 **curr->sempid** 的低 16 位里；从前的 PID 现在被移进高 16 位。

20854：**curr->semval** 依照 **sem_op** 所要求的进行调整——还是临时性的。虽然该操作的结果的范围在随后的代码段中进行了核查，但是不管是在这里还是在其调用者中 **sem_op** 都未进行范围检查。由于 **sem_op** 的值过大或者过小都将造成 **semval** 回绕（wrap around），这样将导致意想不到的结果。

20855：如果这条操作的 **SEM_UNDO** 标志位被设置了，就表示在进程退出时该操作应当被自动取消，相应的撤销结构体会被更新。要注意这里假定 **un** 是非空的——确保这一点是调用程序的责任。

20858：对新的 **semval** 进行范围检查。

20864：循环即将完成，所有操作都将成功完成。如果调用程序只想知道操作能否成功，但此刻并不想执行它们，这些操作就可以马上被取消。否则，操作已经被执行了，所以 **try_atomic_semop** 就继续执行下去并返回成功。

20874：当一个操作把 **semval** 增加得过大时，跳到 **out_of_range** 标记处。函数安排返回 **ERANGE** 错误，并向前跳转到撤销代码。

20878：当进程因为它必须等待信号量归 0 或者操作不能立刻获得信号量而不得不等待信号量时，程序将跳至 **would_block** 标记处。如果这种情形之下调用程序不愿等待，就返回 **EAGAIN** 错误。否则，函数返回 1 表示调用者将需要休眠。

20884：在 **undo** 标记之后的代码取消所有从第 20846 行开始 **for** 循环里所作的工作。

20888：这一行代码显而易见的部分是用来把在第 20853 行暂存的值保存在 **curr->sempid** 的低 16 位中。其隐含的部分是高 16 位（在此假定是 32 位平台）没有必要被设置成 0：C 标准有意给予编译器用 0 或用符号位拷贝来填充空余位的自由。在实际实现中，低级机器指令怎样能最快的工作，编译器就如何工作，结果有时是这些操作的一种，而有时则是另一种。（C 语言标准为什么不拘限于任何一种实现的原因正在于此。）这样的结果时，高位可以是全 0 也可以是全 1，这也正是在第 21116 行里只有低 16 位被屏蔽的原因。

Update_queue

20900：**update_queue** 函数在信号量的值发生改变时被调用。它完成那些此刻可以成功（或者将要失败）的挂起操作，并把它们从挂起队列中移出。

20907：如果这个节点的 **status** 标志已经被前一次 **update_queue** 调用增加过了，那么与该节点相关的进程就还没有机会把它自己从队列中移出。为了提供其它进程机会来执行它的挂起操作并从队列脱离，函数返回。

20910：检查是否此刻能够完成当前一组挂起操作。**q->alter** 是最后一个被通过的参数，所以即将成功的变异（mutating）操作就会自动被取消。这是因为进程将要继续亲自尝试这些操作，而它们是不应该被执行两次的。

20914：假设错误或者成功状态已经能够被判定（对立于需要继续等待），这个节点就被从队列中移出，并且与它关联的进程也会被唤醒。否则，节点留在队列中以便在将来某处被再次尝试。

20917：如果该操作集包括一些变异的操作，标志就被提高以便进程知道唤醒它是由于现在

能够成功了；进程将尝试那些操作并把自己从队列中移出。前边讨论过，第 21342 行要对这个标志进行检查。

20920：函数现在返回，以便多个变异进程不会同时尝试进行它们的那些可能并不互相兼容的改变。回忆一下，非变异的操作位于队列头部，而变异的操作是位于末尾的。其结果是，所有的非变异进程（它们不会彼此干扰）被首先唤醒，然后最多唤醒一个变异进程。

20922：否则，将产生一个错误。该错误代码被保存在 **q->status** 里，接着队列节点被移出。

Count_semncnt

20938：**count_semncnt** 函数从 21117 行被调用来实现 **sys_semctl** 内的 **GETNCNT** 命令。它的工作是记录因等待获得信号量而阻塞的任务数目。

20949：这个循环用于执行在 **sma** 的挂起队列中等待着的每个任务中的每个挂起操作。每当找到一个满足的操作时它就递增 **semncnt**——该操作试图获得特定的以及没有设置 **IPC_NOWAIT** 标志的信号量。

Count_semzcnt

20957：**count_semzcnt** 函数在 21119 行被调用以实现 **sys_semctl** 内的 **GETZCNT** 命令。它除了要对等待信号量归 0 的任务（也就是等待信号量变得可用的任务）进行计数之外，它和 **count_semncnt** 函数几乎一样。因此唯一的区别就在第 20970 行，在那里它使用等于 0 而不是小于 0 来进行测试。

共享内存

共享内存（shared memory）顾名思义就是：一块预留出的内存区域，而且一组进程均可对其进行访问。因为它涉及 IPC 和内存管理两方面的内容，这部分讨论将融合本章及第 8 章以前的材料进行分析。

截至目前为止，共享内存是本章要介绍的三种 IPC 机制里最快的一种，而且也是最简单的一种——对于进程来说，获得共享内存后它和任何其它内存看起来都是一样的。由一个进程对共享内存所作出的改变对所有其它进程都是立即可见的——它们只需通过一个指向共享内存空间的指针来读取，然后就轻松的获得了结果。然而，System V 共享内存没有确保互斥的内置方案：一个进程可以向共享内存中的给定地址写入而同时另一个进程从相同的地址读出，这会导致读者所看到的将是不一致的数据。这个问题在 SMP 机器上非常明显，但是它也会发生在 UP 机器之上——举个例子，假设正当把某个较大的结构写入共享内存空间时写者被转换出了上下文环境，而读者又在写者完成操作之前读取了共享内存的时候。

这样的结果是，使用共享内存的进程必须努力确保读操作与写操作的严格分离（考虑一下，写操作和写操作之间也是如此）。锁和原子操作的相关概念将在下一章详细论述。但是读者已经了解了保证互斥访问共享内存区域的一种方法：使用信号量。这种思想是一旦获得信号量就全速访问内存区域，工作一完成后就立即释放该信号量。

共享内存在一些用到消息队列的情况下也具有同样的帮助作用——一个调度进程可以把工作请求写入共享内存区域的一部分，同时工作者进程可以把结果写入另一部分。这就意味着应用程序要预先为请求和结果空间限制界限，但这样的内存分配和结果写入还是要比使用消息队列快。

对于每个进程来说共享内存区域不必看起来具有相同的地址。如果进程 A 和进程 B 都

在使用同一块共享内存区域，那么 A 可能看到它在一个地址，而 B 则可能会看它在另一个地址。当然，共享内存区域中给定的一个页面将最多被映射为一个物理页面。前一章介绍过的虚拟内存机制只需要为每个进程进行不同的逻辑地址转换即可。

在内核代码中，共享内存区域被称为段（segments），这正是有时会被误用于 VMA 的一个术语。为了预先防止任何混淆，这是一个该术语的非正式用法；它与第 8 章里讨论过的硬件增强的（MMU）段是不同的。为了避免这种说法所可能引起的迷惑，我将继续使用区域（regions）这个术语。

共享内存代码从设计到实现都与消息队列及信号量的代码有一些相似之处。因此，没有必要再介绍 **shm_init**（21482 行）和 **findkey**（21493 行）函数。出于同样的原因，剩下的一些函数和数据结构的讨论也会相应缩短。

Struct shmid_ds

17042：多少有点打破了已经建立的模式，**struct shmid_ds** 不是内核用来跟踪共享内存区域的数据结构。取而代之的是，**struct shmid_ds** 包含这种信息的绝大部分，而剩下的信息则位于下边要介绍的 **struct shmid_kernel** 中。以下是 **struct shmid_ds** 的那些与其对应对象所不同的成员：

- **shm_segsz**——这块共享内存区域的大小尺寸，用字节（不是页面）度量。
- **shm_nattch**——用典型的术语，是指“附属（attached）”到这块区域的任务数目——换句话说，就是使用该共享内存区域的任务数。这个成员是一个参考计数（reference count）。
- **shm_unused**、**shm_unused2** 和 **shm_unused3**——从它们的名字就可推断，这些成员不再用于实现之中；它们的唯一角色看来是为了保持该结构体大小的向后兼容性。

Struct shmid_kernel

17056：**struct shmid_kernel** 用于分离“私有（private）”的共享内存相关信息和“公有（public）”的信息。**struct shmid_ds** 里那些对用户应用程序可见的部分还保留在 **struct shmid_ds** 之内，而关系到内核的私有信息则位于 **struct shmid_kernel** 之内。用户应用程序需要能够通过 **struct shmid_ds** 来进行 **shmctl** 系统调用，所以它的定义必须对它们是可见的，但是内核私有实现的细节就不应该出现在 **struct** 的定义之中。否则，改变内核的执行可能会中断应用程序。**struct shmid_kernel** 具有如下成员；

- **u**——即 **struct shmid_ds**，也就是数据的公共部分。
- **shm_npages**——用页面数表示的共享内存区域的容量。它恰为 **shm_segsz** 成员除以 **PAGE_SIZE**（10791 行）的结果。
- **shm_pages**——用于跟踪这块共享内存区域页面分配的一个“页表”——“页表”在这里加了引号，是因为它不是一个同前一章里一样真正的、硬件支持的页表。不过它完成同样的工作。
- **attaches**——代表各自进程对这块共享内存区域进行映射的 VMA 的一个链表。VMA 在第 8 章里已经介绍过。

Newseg

21511：是与 **newque** 和 **newary** 相对应的函数。它分配并初始化一个 **struct shmid_kernel**，然后把它安置在 **shm_segs** 数组之中。

21537：分配“页表”。它和紧随 **struct shmid_kernel** 之后的对这块内存空间进行分配的另一个 IPC 代码一样，它们都是一个大的分配过程中的一部分。不过，**struct shmid_kernel** 是由 **kmalloc** 分配的（在不可交换的内核内存里），然而“页表”是由 **vmalloc** 分配的（在可交换内存里）。

21546：以把页表填零为起点来初始化所有分配了的元素项。

Sys_shmget

21573：这个函数自然是对应于 **sys_msgget** 和 **sys_semget** 的。唯一新颖的特征是它对进程 **struct mm_struct** 的信号量获取和释放过程。这是一个内核信号量，它与 System V 信号量并不相同——内核信号量将在第 10 章介绍。

Killseg

21610：这个函数对应于 **freeque** 和 **freeary**。它的代码也同那些函数的非常相似，但是有几个特征值得注意。

21616：如果用一个未被占用的 **shm_segs** 元素的索引调用 **killseg** 函数，它就显示一条警告并立刻返回。它的两个对应函数中都不存在相似的代码。

21629：如果元素项的 **shm_pages** 成员是 **NULL**，那么就在某处有一个逻辑错误。**struct shmid_kernel** 要么是没有完全构建好，要么就是已经销毁但还没有被从数组中删除，再或者就是某个类似的看起来“不可能发生”的情况发生了。

21635：释放为页表分配的页面。

21638：如果页表没有映射这个页面，在释放这一项时就无需执行什么操作。

21640：如果页面在物理内存里，则它将被释放回可用页面的缓冲池里，同时递减驻留页面的数目。

21643：否则，页面位于交换空间，它将从那里被释放。

21648：释放页表本身。

Sys_shmctl

21654：这个函数明显是对应于 **sys_msgctl** 和 **sys_semctl** 的，而且和它们有许多共同点。在此只介绍两个共享内存所特有的命令。

21733：**SHM_UNLOCK** 命令是 **SHM_LOCK** 的反作用命令，**case** 在第 21742 行。**SHM_LOCK** 允许拥有足够权能的进程锁住物理内存里的一整块区域，以防止它被交换出去。而 **SHM_UNLOCK** 则对一块加锁区域进行解锁，使得其中的页面再次可以被用于交换。

在这两个 **case** 里的工作看来不甚相似：它只是确定调用者有合适的权能、要被解锁的区域当前是加锁的（或反之亦然），然后设置或者清除适当的模式位。但是这就是所要完成的一切了——其效果会在 **shm_swap**（22172 行）中显现出来。

注意有一个分离的权能用于加锁和解锁共享内存，即 **CAP_IPC_LOCK**（14021 行）。

Insert_attach

21823：这个短小的函数只是把一个 VMA 添加到附属于给定 **struct shmid_kernel** 的 VMA 列表中。注意该 VMA 是添加到列表头部的——顺序并不重要，而且这样处理最为简单。否则的话，**attaches** 的头和尾都将不得不分别被进行跟踪。

Remove_attach

21833：这个函数自然是从附属于给定 **struct shmid_kernel** 的列表中移出一个 VMA。关于此函数的奇怪之处是它并不依赖于它的 **shp** 参数——该参数是一个指针，指向存储在 VMA 列表第一个 VMA 里的 **shp** 的 **attaches** 列表，它位于第 21829 行，而且用于更新列表的过程同样不考虑是否该 VMA 为列表里的第一项（如果它是，相应的 **attaches** 也被更新）。

Sys_shmat

21898：这个函数实现了 **shmat** 系统调用，调用进程借助它可以同一个共享内存区域建立联系。

21923：在一些熟悉的准备工作之后，**sys_shmat** 开始对共享内存区域应出现在调用进程内存空间中的地址进行计算。首先，它要检查调用者传过来的地址。如果它是 **NULL**，而且 **SHM_REMAP** 标志位也未被设置（参见 21959 行），那么请求必须被抛弃——**NULL** 永远不可被读和写。

21929：调用者传入 **NULL** 作为目标地址，这意味着 **sys_shmat** 应该在该进程的内存空间里选择一个地址。**get_unmapped_area** 将提供一个候选的地址（33432 行），顺便需要提及的是该函数在前一章已经讨论过。如果它返回 0（在所有内核支持的平台上都等价于 **NULL**），那么就是无法找到足够大的区域。

21932：如果候选的地址不是恰好在一个页面的边界上，它就会被向上舍入到更高的下一个页面边界，然后用调整过的地址将原先的地址取而代之。**get_unmapped_area** 返回在给定地址上或超过它的第一个可用地址，因此假如上舍入的地址是可用的，它将被采用。

现在解释一下为什么地址要向上舍入而不是向下舍入（那样能够更快和更简单一些）：假如 **sys_shmat** 进行向下舍入而所得地址不可用，那么代码将陷入死循环。下一次调用 **get_unmapped_area** 将从下舍入地址处向上搜索并返回到原先未经舍入的地址处，而它将再次被向下舍入，发现不合适，又传送给 **get_unmapped_area**……要注意在这里使用的是 **SHMLBA**（11777 行）而不是 **PAGE_SIZE**（10791 行）来决定地址的适宜性。不过，正如你所见到的，**SHMLBA** 恰好被定义为 **PAGE_SIZE**，所以效果是相同的。

如果 **SHMLBA** 和 **PAGE_SIZE** 是一样的，那么二者又为什么要兼有呢？答案在于 **SHMLBA** 在绝大多数平台上就是 **PAGE_SIZE**，但并不是在所有平台上都是如此。在 MIPS 上——CPU 具有 4K 的 **PAGE_SIZE**——Linux 把 **SHMLBA** 定义为非常大的 0x40000（256K），其注释说明选择这样大的值是为了遵守基于 MIPS 机器的 SGI 应用程序二进制接口（ABI——Application Binary Interface）。然而，MIPS ABI 的版本 2 和 3 却明确声明了 **SHMLBA** 的值“ 在符合标准的实现上是允许有所不同的 ”，所以不清楚为什么内核开发人员认为 256K 的值是必要的。或许该值是非常早期的 ABI 版本所要求的，但是我向回检查 ABI 一直到 1.2 版仍没有发现任何这样的要求。还有，在 SPARC-64 上，**SHMLBA** 是 **PAGE_SIZE** 的两倍；不幸的是，这个区别没有在代码中进行解释。

21936：否则，调用者传送一个建议的地址。如果有必要而且是被允许的，该地址就被向下取整。

21945：确保从被选地址开始的大小为 **len** 的内存块在进程的允许内存空间之内。（**len** 已在几行之前计算出来，21913 行。）当调用者提供候选地址时进行检查明显是必要的，

而且粗看起来当 **sys_shmat** 用 **get_unmapped_area** 来选择一个地址时进行检查也是必要的。 尽管区域的大小已经被传递给它，**get_unmapped_area** 还是要执行一个相似的检查，**struct shmid_ds** 的 **shm_segsz** 成员不必和 **len** 相同——**len** 是 **PAGE_SIZE** 的一个倍数，而 **shm_segsz** 则可以不是。

不过，因为所有被 **get_unmapped_area** 使用的地址都是页对准的，所以传递给它的区域大小是否是页面尺寸的倍数都不会影响它的计算。

21951：正如注释中所说明的，被选区域必须为进程的栈留出一些空间。这个缓冲区间有四个页面——这个数字并没有什么特别之处，只要达到进程有足够的栈空间的目的即可。在上一章中曾提到过如果某任务耗尽了它的栈，它将被杀死。综合考虑起来，让单个系统调用失败可能要比让整个进程被无理的杀死更好一些——进程可以从前者中逐渐恢复，但是后者却不行。

21959：**SHM_REMAP**（17075 行）的主要作用在此体现：如果 **SHM_REMAP** 被设置了而且调用者提供的区域已在使用，那么就没有错误，这是因为 **SHM_REMAP** 用于允许调用者把一块共享内存区域映射到它自己的内存里——比如是一个全局缓冲区。如果这个标志没有被设置，被选的地址就一定不能和进程已经拥有的任何内存相互重叠。

21971：如果调用者缺少使用这块内存区域的许可，系统调用失败。如果 **SHM_RDONLY**（只读）标志被提供，调用者只需要读许可；否则，调用者需要读许可和写许可。

21991：填充新的 VMA。特别注意它的 **vm_ops** 成员被初始化为指向 **shm_vm_ops**（21809 行），就像在第 8 章里讨论过的一样。

22004：增加这块区域的引用计数，以便它不会被过早的销毁。

22005：调用 **shm_map**（21844 行）把共享内存页面映射到进程的内存空间里。如果失败，它就会返回，同时递减引用计数，如果这是第一个和唯一的引用，那么还需要销毁该区域，然后释放 VMA，这样整个工作就结束了。

注意即使这是最后一个引用，VMA 也不必被释放；该区域也必须要用 **SHM_DEST** 标志（17106 行）来进行标记。**SHM_DEST** 可以在由调用者来设置的标志位之中；它也可以在后面 **sys_shmctl** 的 **IPC_RMID** 情况里被设置——参见 21780 行。以这样的方式，一块共享内存区域可以比它所有的附属进程生存更长时间。出于同保留一个检查点（checkpoint）文件会在某些情况下非常有用相类似的原因，这样的处理方式也是有用的：你可能会有一个每晚都要运行几个小时的耗时进程，要把它的处理结果保存在一个即使该进程当前工作完成之后仍然继续存在的共享内存区域。只要通过附属到剩下的共享内存区域，它就能够恰好在下一个晚上从停下的地方重新开始。（当然，由于共享内存区域——不同于文件——在计算机关闭后就会消失，所以这种方案不适用于不能有丢失危险的工作。）

22014：添加到附属于这块区域的 VMA 列表中，然后更新一些关于每区域统计的数据。

22019：返回在调用者空间里真正被选择的共享内存区域地址，然后成功地返回。

Shm_open

22028：**shm_open** 函数像是 **sys_shmat** 的一个简化版本（21898 行）。它把一个给定的 VMA 附加到一个共享内存区域里。提供的 VMA 是从一个已经附属于目标区域的 VMA 复制而来，所以这个 VMA 本身已经被正确填写了；**shm_open** 函数的工作基本上只是要完成附属连结。

正如 **shm_open** 上方的注释所陈述的，这个函数是从 **do_fork**（23953 行）里被调用的，该函数在第 7 章里已经介绍过。更准确的说，这个函数是在 **dup_mmap** 里（23654

行）第 23692 行被调用的。然后，**dup_mmap** 在 **copy_mm** 里（23774 行）的第 23801 行被调用；而 **copy_mm** 又是在 **do_fork** 里的第 24051 行被调用的。

22033：从 VMA 的 **vm_pte** 成员里抽取 **shm_segs** 下标，然后确保该处有一合法项。注意下标无需进行范围检查，这是因为和 **SHM_ID_MADK** 所进行的按位与操作（11757 行）已强迫它合乎范围了。

22040：添加 VMA 到区域里并更新区域的统计数字。

Shm_close

22050：**shm_close** 明显是 **shm_open** 的反作用函数，它把一个 VMA 从它附属的共享内存区域里分离出来。尽管在其它地方内核也可以调用 VMA 的 **close** 操作，但 33821 行看来是能结束调用 **shm_close** 的唯一之处。这是 **exit_mmap** 的一部分（33802 行），而它又是被 **mmput**（23764 行）调用、**mmput** 被 **__exit_mm**（23174 行）调用，而 **__exit_mm** 又被 **do_exit**（23267 行）所调用，**do_exit** 函数在第 7 章就已经讨论过。要注意还有其它到达 **shm_close** 的路径，我们很快就会对其中之一进行介绍。

22056：从 VMA 的 **vm_pte** 成员里抽取 **shm_segs** 下标然后把该 VMA 分离出区域。出于和 **shm_open** 同样的原因，下标不用进行范围检查。还要注意的是 **shm_close** 不检查是否在指示的下标处存在一个合法的 **shm_segs** 项。读者已经看到，**remove_attach** 不依赖于它的 **shp** 参数，所以它对此并不关心。然而 **shm_close** 剩下部分将假定其它共享内存代码被正确的使用和执行，所以这种“不可能发生”的情形真的是不可能发生的。

22058：从共享内存区域分离 VMA 然后更新区域的统计数字。

22061：减少该区域的引用计数，如果可能的话还需要将其释放。

Sys_shmdt

22068：与 **sys_shmat** 相反，**sys_shmdt** 函数把一个进程从一块共享内存区域里分离出去。

22074：开始对所有代表进程内存的 VMA 进行循环处理。

22076：如果 VMA 代表一块共享内存区域（这可以通过检查它的 **vm_ops** 成员进行精巧的测试），而且该 VMA 始于目标地址，就应解除该 VMA 的映射。

22079：**do_munmap**（33689 行）调用 **unmap_fixup**（33578 行），它又间接的在 33592 行调用 **shm_close**。**do_munmap** 和 **unmap_fixup** 都在第 8 章里介绍过。