《 UNIX 高级教程系统技术内幕 》
OCR by djq@SMTH ftp://166.111.136.159/incoming/UNIX_book/

第 7 章　同步和多处理器

7.1　简　　介

对更高处理能力的渴望导致了硬件体系结构的不断进步。在这个方向上的主要进步之一就是多处理器系统的发展，这些系统中包括了两个或多个共享主存和其他资源的处理器。这种配置具有几个好处。它们为项目提供了灵活的增长路线，开始可以用单处理器，随着计算需求的增大，通过向机器增加额外处理器进行无缝的扩充。专为计算用的系统常常受 CPU 限制，CPU 是主要瓶颈，以及其他系统资源诸如 I/O 总线和内存利用率不高。多处理器系统无需复制其他资源就可增加处理能力，因此对 CPU 密集型系统提供了有效的解决方案。

多处理器也提供了更高的可靠性；若其中一个处理器失败，系统可以无需中断继续运行。然而，任何事物都具有两面性，因此它发生错误的机会也增高了。为了确保高的平均无故障工作时间(MTBF)，多处理器系统应装备容错硬件和软件。特别是，系统应从一个发生错误的处理器上恢复而不会导致系统崩溃。

若干 UNIX 变体已经开始利用这一系统。最早的 UNIX 多处理器系统实现之一就运行于 AT&T3B20A 和 IBM 370 平台上[Bach 84]。当前，大多数 UNIX 主流实现自身就是支持多处理器系统(DECUNIX，Solaris 2.x)的，或者有多处理器系统的变体(SVR4/MP，SCO/MPX)。

理想情况我们希望系统性能随处理器数目增加而线性增长。实际系统由于若干原因而无法达到这个水平。由于系统中其他部分并不增加，它们可能变成瓶颈。而访问共享数据结构时同步的需求，以及支持多处理器的额外功能增加了 CPU 的负担，减少了系统的整体性能的提高。操作系统必须尽量减少这种负担，允许优化使用 CPU。

传统的 UNIX 内核假设为单处理器系统，如要使其运行于多处理器系统需大量修改。三个主要变化为：同步，并行和调度策略，同步包括用于控制对共享数据和资源访问的基本原语。传统的睡眠/唤醒原语加上中断阻塞在多处理环境中是不够的，必须用更强大的方案取代。

并行关心的是有效地使用同步原语宋控制共享资源的访问。这要考虑锁粒度，锁定位、死锁避免等的决策问题，7.10 节讨论其中一些问题。为了优化对处理器的使用，调度策略也相应地进行修改。7.4 节分析与多处理器调度相关的一些问题。

本章首先描述系统 UNIX 系统中的同步机制并分析其局限性,接着对多处理体系结构进行概述。本章最后介绍现代 UNIX 系统中的同步，这些方法同时适于单处理器和多处理器平台。

传统 UNIX 系统中进程是基本调度单位，它有单一的控制线程。正如第 3 章所述，许多现代 UNIX 变体允许每个进程中有多个控制线程，并在内核中全面支持线程。这种多线程系统在单处理器和多处理器体系结构中都使用。在这些系统中，单个线程争用并锁定共享资源，在本章其余部分，我们将以线程为基本调度单位，它是更一般的抽象。对单线程系统而言，线程就是进程的同义词。

7.2　传统 UNIX 内核中的同步

UNIX 内核是可重入的——几个进程可以同时在内核执行，甚至可能执行同一例程。单处理器系统中，任一时刻只可能有一个进程实际运行。然而，系统快速地在进程间切换，提供了一个它们并发执行的假象。这个特征一般称为多道程序(multiprogramming)。由于这些进程共享内核,内核必须同步对它的数据结构的访问,以防破坏它们。2.5 节提供了对传统 UNIX 同步技术的详细讨论。本节中，我们总结重要原则。

传统 UNIX 内核的非抢占特性是第一道保护。任何在内核中执行的线程会一直运行，直到它已准备离开内核或因某种资源而阻塞，而无论其时间段用完与否。这使内核管理许多数据结构无需加锁，因为知道直到当前线程工作完毕或准备离开内核，这些数据结构会保持一致状态而不会有其他线程的访问。

7.2.1 中断屏蔽

非抢占规则提供了一个强大而范围很广的同步工具，但它也有局限性。尽管当前线程不会被抢占，但它可能被中断。中断是系统活动不可缺少的部分，而且通常需要快速服务。中断处理程序可能会访问当前线程正使用的数据结构，造成数据破坏。因此，内核必须对那些被普通内核代码和中断处理程序同时使用的数据进行同步。

UNIX 通过屏蔽中断解决这个问题。每个中断分配一个中断优先级(ipl)，系统维护着当前中断优先级，并在中断发生时检查它，若发生中断比当前中断优先级高，它被立即处理。(抢占当前正在处理的低优先中断)。否则，内核屏蔽中断直到中断优先级降得足够低。在调用中断处理程序之前，系统将当前中断优先级设为这个中断的优先级；当中断处理程序完成后，系统把当前中断优先级恢复为前面的值(所保存的)。当进行了某些关键处理时，内核也可以明确的将当前中断优先级设为任何值以屏蔽中断。

例如，内核例程可能要从缓冲队列中删除一个磁盘块缓冲区，而且这一缓冲队列也可被磁盘中断处理程序访问。管理这一队列的代码称为临界区。进入临界区前，例程将当前中断优先级升得足够高以屏蔽磁盘中断。当完成队列管理，例程将当前中断优先级返回原值，允许磁盘中断服务，因此中断优先级允许内核和中断处理程序对共享的资源进行有效同步。

7.2.2 睡眠和唤醒

尽管线程因某些原因被阻塞，但它经常需要保证独占某一资源。例如，一个线程要将磁盘块读入块缓冲区。它分配缓冲区保存磁盘块后，初始化磁盘动作。这个线程需等待 I/O 完成，即意味着它要放弃 CPU 等待其他线程，若其他线程取得同一缓冲区并用于其他目的，缓冲区的内容可能不确定或破坏。这意味着线程阻塞时需要以某种方式锁定资源。

UNIX 通过为共享的资源分配 locked 和 wanted 标志实现这种功能。当线程需要访问一共享资源，如块缓冲区，它首先检查它的 locked 标志。若标志为空，则线程设置标志并继续使用这个资源。当第二个线程试图访问同一资源时，它发现 locked 标志被设置后就必须阻塞(进入睡眠)直到资源可用。在做上述动作以前，它设置相关的 wanted 标识。进入睡眠的操作包括将线程链入睡眠线程队列，改变它的状态信息，表明它是在这个资源上睡眠的，并将处理器放弃给其他线程。

当第一个线程使用完资源，它清除 locked 标志并检查 wanted 标识。若 wanted 标志已设置，则意味着其他线程正在等待(阻塞于)这个资源。在这种情况下，线程检查睡眠队列，并唤醒所有这类线程。唤醒线程包括从睡眠队列上取下它，改变它的状态为可运行，并放入调度队列。当其中一个线程最终被调度，它再次检查 locked 标志，发现它为空，设置它，并继续使用该资源。

7.2.3 传统方法的局限性

传统同步模型在单处理器系统中工作很正常，但也有一些重要的性能问题将在本小节中讨论。在多处理器环境下，这个同步模型会彻底崩溃，这将在 7.4 节中讨论。

将资源映射到睡眠队列

睡眠队列的组织方式在一些情况下会导致很差的性能。在 UNIX 中，线程因等待资源锁或一个事件而阻塞。每个资源或事件与一个睡眠通道相连。睡眠通道是个 32 位值，通常设置为资源的地址。系统中有一组睡眠队列，同时有一个散列函数将通道(从此处映射资源)映射到这些队列中，如图 7-1 所示。线程把自己放到合适的睡眠队列，并在它的 proc 结构内存好睡眠通道后进入睡眠。

图 7-1　将资源映射到全局睡眠队列上

这种方法有两个后果。首先，多于一个事件可以映射到同一通道上。例如，一个线程锁住一个缓冲区，启动其上的 I/O 活动，并进入睡眠直到 I/O 完成。另一个线程试图访问同一缓冲区，发现它已加锁，则必须阻塞直到它可用，这两个事件映射同一通道，即缓冲区地址。当 I/O 完成后，中断处理程序将唤醒这两个线程，而此时第二个线程所等待的事件还未发生。

其次，散列队列的数目比各种睡眠通道(资源或事件)的数目少得多；即，多个通道映射同一散列队列，因此一个队列中包含等待不同通道的线程。wakeup()例程必须检查队列中的每个线程，并唤醒那些阻塞于对应通道的线程。结果是 wakeup()所用的时间并不决定于在此通道上睡眠的进程数，而是决定于在队列中睡眠的进程总数。这种不可预见的延迟，一般是不希望的，并且对那些支持实时应用，要求有固定分派时延的内核是不可接受(见 5.5.4 节)。

一种解决方法是为每个资源或事件分配一个独立的睡眠队列(见图 7-2)，这种方法以增加这些附加队列的内存负载开销为代价，优化了唤醒(wakeup)算法的时延，典型的队列头包括两个指针(向前和向后)以及其他信息。系统中同步对象的数日很大，为它们每个分配一个睡眠队列可能是十分浪费的。

图 7-2　每个资源阻塞线程队列

Solaris 2.x 提供了一个更节省空间的方法[Eykn 92]。每个同步对象有一个两字节域定位旋转门(turnstile)结构，它包括睡眠队列和其他信息(见图 7-3)。内核只为那些有线程阻塞在资源分配旋转门。为了加快分配，内核维护一个旋转门库，库的大小比活动线程数目大。这种方法以最小存储开销提供了更可预见的实时行为。5.6.7 小节对旋转门进行了详细的介绍。

图 7-3　在旋转门上的阻塞线程队列

共享和互斥访问

睡眠/唤醒机制适用于一次只允许一个线程使用资源。然而，它不能立即用于更复杂的协议如读-写同步问题。它希望能允许多个线程共享一个读的资源，而在修改它之前要求互斥访问。例如文件和目录块可以有效的共享。

7.3　多处理器系统

多处理器系统有三个重要特征。首先是它的内存模型，它决定了处理器共享内存的方式。其次是硬件的同步支持。最后，软件体系结构决定了处理器、内核子系统和用户进程之间的关系。

7.3.1　内存模型

从硬件的角度,多处理器系统根据它们的耦合与内存访问语义的不同可被分为三类(见图 7-4)：

- 统一内存访问(UMA)
- 非统一内存访问(NUMA)
- 非远程内存访问(NORMA)

图 7-4　UMA、NUMA 和 NORMA 系统

最普通的系统是 UMA，或称共事内存多处理器(见图 7-4(a))。这种系统允许所有 CPU 平等的访问主存①和 I/O 设备，而它们一般都是挂在单系统总线上。从操作系统的角度看，这是一个简单的模型。但它的主要问题是可扩展性。UMA 的体系结构只能支持数目很少的处理器。随着处理器数目的增加，总线的竞争也会增加。最大的一个 UMA 系统是 SGI Chal-lenge，它在单总线上支持 36 个处理器。

在 NUMA 系统中(见图 7-4(b))，每个 CPU 都有本地内存，但也可以访问其他处理器的本地内存。远程访问很慢，一般比本地访问慢一个数量级。还有一些混合系统(见图 7-4(c))，在

这种系统中，一组处理器对本地内存进行统一访问，而对其他组中的内存进行慢的访问。不了解 NUMA 模型的硬件体系结构的细节，开发应用程序是很难的。

NORMA 系统中(见图 7-4(d))，每个 CPU 只能直接访问它自己的本地内存，只有通过显式消息传递访问远程内存。硬件提供高性能的交叉连接，增加远程内存访问的带宽。为这种系统构造成功的系统要求 Cache 管理和操作系统中的调度支持，同时，要求编译器能为这种硬件优化代码。

本节中讨论 UMA 系统。

7.3.2 同步支持

一个多处理器的同步本质上依赖于硬件支持。考虑为独占使用而锁定资源的基本操作，需要在共享内存位置中设定 locked 标志。这需要完成下列操作：

1.读标志。

2.若标志为 0(即资源未加锁)，设其为 1，锁住资源。

3.若取得锁，则返回为 TRUE，否则返回为 FALSE。

在多处理器中，两个线程在不同的处理器上可能同时试图实施上述操作过程。如图 7-5 所示，两个线程都认为自己独占资源。为避免这种情况，硬件需要将这三个子任务组合为一个不可分割的操作，以提供更为强大的原语。许多体系结构通过提供原子的 test-and-set 指令或条件存储(Stor-Conditional)指令的方法解决这个问题。

图 7-5 如果 test-and-set 不是原子的指令时造成竞争状态

原子的 Test-and-Set

原子 test-and-set 操作一般针对内存中的一位。它测试该位，设置它为 1，并返回它的原值。因此，当操作完成时，位的值设为 1(加锁)，并通过返回值表示本操作之前是否这位已被设置。这个操作确保为原子的，因此当不同处理器的两个线程对同一位同时发起同一指令，其中一个操作完成后，另一个再开始。进一步，这一操作对于中断也是原子的，因此一个中断只能在该操作完成后出现。

这一原语非常适合于简单锁。当 test-and-set 返回 l，调用线程拥有资源。若返回 0，别的线程锁定资源。开锁只需简单地把位设为 0。test-and-set 指令的例子 VAX-11[Digi 87]的 BBSSI(Branch On Bi Set and Set Interlocked)和 SPARC 的 LDSTUB(LoaD and STore Un-signed Byte)。

Load-Linked 和 Store-Conditional 指令

某些处理器如 MIPS R4000 和 Digital 的 Alpha A XP 使用一对特殊的 load 和 store 指令，提供一个原子的 read-modify-write 操作。load-linked 指令(也称为 load-locked 指令)从内存向寄存器装入一个值，并设定一个标识使硬件监测这个位置。若处理器向这个被监测的位置写数据，硬件清除这个标识。store-conditional 在标识仍设置的情况下，向这个位置写个一个新值。另外它向另一个寄存器中写入这个值以表明存储发生。

这一原语可被用于产生一个原子的增量操作。读该变量用 load-linked，设置新值用 store-conditional。这一序列一直重复直到成功。在 DG/UX[Kell 89]中的事件计数就是基于这个机制。

某些系统如 Motorola MC88100 使用基于 swap-atomic 指令的第三种技术。本章末尾的练习将深入研究这种方法。所有上述硬件机制都是建造强大的同步方法的基石。后续小节的高层软件抽象是在硬件原语之上的。

7.3.3 软件体系结构

从软件的角度来看，共有三种类型的多处理器系统——主从式、功能非对称式和对称式。主从式[Gobl 81]是非对称的：一个处理器充当主处理器的角色，其他的为从处理器。只有主处理器能进行 I/O 操作和接收设备中断，某些情况下，只有主处理器运行内核代码，从处理

器只能运行用户层代码。上述限制可能优化系统设计，但减少了多处理器系统的优势。测试结果[Bach 84]显示出典型的 UNIX 系统有大于 40%的时间是在内核模式运行的，因此希望将内核活动分布到所有处理器上。

在功能非对称式的多处理器系统中，不同的处理器运行不同的系统。例如，一个处理器运行网络，另一个管理 I/O。这种系统更适合于专用系统而不适用于一般目的像 UNIX 这样的操作系统。Auspex NS5000 文件服务器[Hitz 90]是这种模型的成功应用。

对称式多处理(SMP)是更流行的类型。在 SMP 系统中，所有 CPU 是平等的，共享内核代码和数据的单一副本，并竞争系统资源，如设备和内存。每个 CPU 都可运行内核代码，用户进程可被调度到任一处理器上。在没有明确声明的情况下，本章只介绍 SMP 系统。

本章其他部分描述用于单处理器和多处理器系统中的现代同步机制。

7.4 多处理器同步问题

传统同步模型中的基本假设之一是线程保持独占内核(除了中断)直到它准备离开内核或阻塞于某资源。这一假设在多处理器中将不存在，因为同一时刻每个处理器都可能执行内核代码。我们现在需要保护许多在单处理器时无需保护的数据。例如，考虑访问 IPC 资源表(见 6.3.1 小节)，中断处理程序下访问这一数据结构，它也不支持可能引起进程操作阻塞的操作。因此在单处理器中，内核能够无需加锁就管理这个表。在多处理器情况下，不同处理器的两个线程可能同时访问这个表格，因此在使用它之前必须以某种方式锁住它。

加锁原语也必须改变。在传统的 UNIX 系统中，内核只是简单地检查 locked 标志，并设置锁住对象。在多处理器中，不同处理器上的两个线程可能并发的检查同一资源的 locked 标识，二者都发现标识为空，并认为资源可用。然后二者就会设置这个标志，并继续访问资源，产生了不可预料的结果。因此系统必须提供某种形式的 test-and-set 原子操作以确保只有一个线程锁住资源。

另一个例子涉及屏蔽中断。在多处理器系统中，一个线程只能屏蔽它所运行的处理器上的中断，一般不能屏蔽所有处理器上的中断——实际上，其他处理器可能已收到中断。运行在另一个处理器上的处理程序可能破坏这个线程正在访问的数据结构。这一情况因中断处理程序不允许使用睡眠/唤醒同步模型而加剧，同为大多数实现不允许中断处理程序被阻塞。系统应提供某种机制屏蔽其他处理器上的中断。一个可能的解决方法是用软件管理全局的中断优先级。

7.4.1 唤醒丢失问题

在多处理器上，睡眠/唤醒机制不能正确工作，图 7-6 说明了一个潜在的竞争条件。线程 T1 已锁住了资源 R1。在另一个处理器上运行的线程 R2 试图取得这一资源，发现它已被锁住。T2 调用 sleep()等待这个资源。在 T2 发现资源已加锁，并调用 sleep()的这段时间里，T1 释放资源，并唤醒所有阻塞于 R1 资源的进程。由于 T2 尚未被放入睡眠队列，它失去这次被唤醒的机会。最后的结果是资源并未加锁，而 T2 却等待系统资源被解锁。若没有进程再次访问这个资源，T2 会一直阻塞下去。这个问题被称为唤醒丢失问题，解决它需要某种机制把测试资源是否可用和调用 sleep()合为一个单一原子操作。

图 7-6 唤醒丢失问题

非常清楚，我们需要一组新的、能够在多处理器中正确工作的原语。这为我们提供了很好的机会检查传统模型中的其他问题，并设计更好的方案。这些问题大多与性能有关。

7.4.2 巨群问题

当一个线程释放一个资源，它唤醒所有等待它的线程。这时，它们中的一个可能会锁定资源；其他进程发现资源仍加锁，则返回睡眠。这导致了唤醒和环境切换的额外开销。

这一问题在单处理器中并不尖锐，因为随着线程开始运行，锁上的资源一般会被释放。

然而在多处理器上，若几个线程阻塞于一个资源，唤醒所有的进程可能会引起它们在不同处理器上同时被调度，它们又再次争夺同一资源。这通常被称为巨群问题。

尽管只有一个线程阻塞于这个资源，在它被唤醒到实际运行之间仍存在时间延迟，在这段时间内，另一个独立线程也可能取得资源，引起这个被唤醒线程再次睡眠。若这种情况频繁发生，会导致这个线程饿死。

我们已经检查了几个影响正确操作和性能的传统同步措施。本章其他部分将介绍能够同时在单处理器上和多处理器上正确工作的同步机制。

7.5 信 号 灯

早期的 UNIX 多处理器版本几乎全都使用 Dijkstra 的信号灯处理同步[Dijk 65](也称信号灯为计数信号灯)，它是一个支持两个基本操作 P()和 V()的整型变量。p()减少信号灯，并当新值小于 0 时被阻塞，V()增加信号灯，若新值大于或等于 0，则唤醒阻塞于它之上的线程(若存在)，例 7-1 描述了这些函数，并加上了初始化函数 initsem()和 CP()函数，它是 P()的非阻塞版本。

实例 7-1 信号灯操作

```
void initsem(seamphore * sero, int val)
{
    *sem=val;
)
void P(semaphore *sem)    /* acquire the semaphore */
{
    *sem -= 1;
    while(*sem < 0)
        sleep ;
}
void V(semaphore *sem)    /* release the semaphore */
{
    *sem += 1;
    if( *sem <= 0)
        wakeup a thread blocked on sem;
}
boolean_t CP(seamphore *sem)  /* try to acquire semaphore without blocking */
{
    if(*sem>0) {
        *sem -= 1 ;
        return TRUE;
    }else
        return FALSE;
}
```

内核保证信号灯操作是原子的，甚至在多处理器系统中。这样假使两个线程试图操作同一信号灯，只有当一个完成或阻塞另一个才会开始。P()和 V()操作可以同 sleep 和 wakeup 相对应，但有不同的语义。CP()操作允许一种以不会阻塞的方式轮询信号灯，它用于中断处理程序和其他不允许有阻塞的函数中。它也可用于死锁避免，而 P()操作很可能会引发死锁。

7.5.1 提供互斥访问的信号灯

例 7-2 显示了如何使用信号灯对于一个资源进行互斥访问。一个信号灯可以同一个共享资源链表相联，并初始化为 1。每个线程用 P()操作锁住资源，用 V()操作释放它。第一个 P()将值设为 0，引起后续的 P()操作阻塞。当 V()完成，值增加，其中一个阻塞线程被唤醒。

实例 7-2　用于锁定资源的信号灯

```
/* During initialization */
semaphore sem；
initsem(&sem,1);
/* On each use */
P(&sem);
Use resource;
V(&sem);
```

7.5.2　使用的信号灯的事件等待

例 7-3 显示如何通过将信号灯初始为 0，使其用于事件等待，线程调用 P()将会阻塞。当事件发生时，每个阻塞线程的 V()均结束。这个事件可以这样完成，当事件发生时调用一个 V()，并让每个被唤醒的线程再调用一个 V()，如例 7-3 所示。

实例 7-3　用于等待一个事件的信号灯

```
/* during initialization */
semaphore event;
initsem(&event, 0)；     /* probably at boot time */
/* Code executed by thread that must wait on event */
P(&event);    /* Blocks if event has not occurred */
/* Event has occurred */
V(&event);    /* So that another thread may wake up */
/* Continue processing */
/* Code executed when event occurs */
V(&event);    /* Wake up one thread */
```

7.5.3　用于控制可计数资源的信号灯

信号灯也可用于分配可计数资源，例如 STREAMS 实现中的消息块的头。如例 7-4 所示，信号灯被初始化为该资源可用实例数。线程调用 P()取得资源的的一个实例，而 V()释放它，这样，信号灯的值表示当前可用资源实例数。若值为负，那么它的绝对值表示挂起等待这个资源的线程(阻塞线程)数。这是经典的生产-消费问题的一个自然解决方法。

实例 7-4　用于计数可用资源实例的信号灯

```
/* During initialization  */
semaphore counter；
initsem(&counter, resourceCount);
/* Code executed to use the resource */
P(&counter);    /* Blocks until resource is available */
Use resource;   /* Guaranteed to be available now */
V(&counter);    /* Release the resource */
```

7.5.4　信号灯的缺点

尽管信号灯提供了一个灵活的用于处理几类同步问题的统一抽象，但它也存在一些缺点而无法用于某些情况下。首先，信号灯是一个基于提供原子性和阻塞机制的底层原语之上的高层次抽象，为了实现 P()和 V()操作在多处理器系统中的原子性，必须有更低层次的操作保证对信号灯变量自身的互斥访问。阻塞和重新运行要求环境切换以及睡眠和调度队列的管

理，所有这些使操作很慢。这种代价对于那些需要使用很长时间的资源也许可以容忍，但对于只需使用很少时间的资源是不可接受的。

信号灯的抽象也隐藏了线程是否必须阻塞于 P()操作中的信息。这些信息一般是无关紧要的，但对于某些情况却至关重要。例如，UNIX 的缓冲区 cache 使用 getblk()函数在缓冲区 cache 中寻找某块特定的磁盘块。若在 cache 中找到希望的块，getblk 调用 P()试图锁上它。若 P()因这块缓冲区已经加锁而睡眠，那么不会保证当它醒来时，缓冲区中会包含同样的块，已经对缓冲区加锁的块可能已将这块缓冲区分配给其他块。这样当 P()返回时，线程可能锁上了并非所需的缓冲区，这个问题在信号灯框架内可以解决，但方法笨重而低效，这意味着其他抽象可能更适合[Ruan go]。

7.5.5 护卫

同传统的睡眠/唤醒机制相比，信号灯的优点在于进程不会被凭空唤醒。当线程在 P()中被唤醒，可以保证获得所希望的资源。这种语义保证信号灯的所有权传给被唤醒的进程先于进程的实际运行。若另一进程试图同时取得这个信号灯，它是无法得到的。而这个事实会导致称为信号灯 Convoy 的性能问题[Lee 87]，Convoy 是当信号灯在经常竞争时产生的。尽管这也会降低锁性能，但信号灯的特殊语义加重了问题。

图 7-7 显示了 Convoy 的形式。R1 是受信号灯保护的临界区。在时刻(a)线程 T2 取得信号灯，T3 线程阻塞于它。T1 是运行于另一个处理器的线程，T4 等待调度，现假设 T2 离开临界区，并释放了信号灯，它唤醒 T3，并将其放入调度队列。T3 现持有信号灯如(b)所示。

现假设 T1 需进入这个临界区。因为 T3 占有信号灯，T1 将被阻塞，释放处理器 P1。系统调度线程 T4 运行于 Pl。因此，在(c)中，T3 占有信号灯，丁 1 阻塞于它，二者却将不能运行直到 T2 或 T4 放弃它们的处理器，

图 7-7 Convoy 格式

问题发生在(c)。尽管信号灯是赋予了 T3，而 T3 并未运行，因此并未在临界区中。结果是 T1 要阻塞于这个信号灯而临界区中却没有线程。信号灯的语义强迫资源分配以一种先来先服务的顺序②。这会造成一系列不必要的环境切换。假设信号灯被替换为互斥锁或互斥量，那么在步骤(b)，T2 会释放锁并唤醒 T3，但 T3 在这里不能拥有锁。结果是在步骤(c)时，Tl 取得锁，减少了环境切换。

注意： Mutex(互斥锁的简写)，是表示任何具有强迫排它访问语义原语的通用词。

一般而言，用一组低开销的较低层次的原语替代一个单一的高层次抽象是所希望的。这也是现代多处理内核的趋势，后续小节介绍这些低层次机制，它们组合在一起提供了丰富多彩的同步方法。

7.6 自 旋 锁

最简单的锁原语是自旋锁，也称为简单锁或简单互斥锁。若一个资源被自旋锁保护，一个试图取得资源的线程将一直保持忙等(循环或自旋)直到资源被解锁。它通常是一个标量，0 可用，1 为加锁。变量的管理是通过围绕原语的 test-and-set 或机器上可用的相似指令所构造的循环实现的。实例 7-5 显示了自旋锁的一个实现。它假设 test-and-set()返回对象的原值。

实例 7-5 自旋锁的实现

```
void spin_lock(spinlock_t *s)    {
    while(test_and_set(s) != 0)   /* already locked */
       ;    /* loop until successful */
}
```

```
void spin_unlock(spinlock_t *s)    {*s = 0;}
```

即使这么简单的算法也是有缺陷的。有许多处理罪中，test_and_lock 工作是通过锁住内存总线，所以循环占据了总线并极大的降低了系统性能。一个较好的的方法是使用两个循环——若测试失败，内层循环将等待变量变为 o。内层循环的简单测试不要求锁住总线。实例 7-6 显示了这种改进的实现。

实例 7-6　改进的自旋锁实现

```
void spin_lock(spintock_t *s)
{
    while(test_and_set(s)!=0)    /* already locked */
    while(*s != 0)
        ;    /* wait until unlocked */
}
void spin_unlock(spinlock_t *s)    {*s = 0;}
```

7.6.1　自旋锁的使用

自旋锁最重要的特点是线程在等待锁被释放时一直占据 CPU。本质上讲，只是在极短的操作过程中才使用自旋锁。特别是决不能在阻塞操作中持有自旋锁。甚至为了保证尽量短时间地占有锁，可在取得自旋锁以前阻塞当前处理器的中断。

自旋锁的基本前提是线程在某处理器上忙等一个资源，而另一个线程在不同处理器上正使甩这个资源。这只有在多处理器上才可能。在单处理器上，若一个系统试图获取一个已被持有的自旋锁，就会进入死循环。然而多处理器算法对于任何数目的处理器都要适用，即它也必须适用于单处理器情况。这就要求线程必须严格遵守规则，即当它持有自旋锁时，决不放弃对 CPU 的控制。在单处理器情况下，这会保证线程永不需对自旋锁忙等。

自旋锁的主要优点在于它们的花销少。当对锁的使用没有竞争时，典型的 lock 和 unlock 操作每个只需单指令。对于那些只需简短访问数据结构的操作，如从双向链表删除一个项目，或对某变量进行 load-modify-store 类型酌操作是非常理想的。因此，它们常用于保护那些在单处理器上不要保护的数据结构。它们也被扩展地用于保护更复杂的锁，如在后续节中所示。例如，信号灯使用自旋锁保证它们操作的原子性，如实例 7-7 所示。

实例 7-7　使用自旋锁访问双向链表

```
spinlock_t list;
spin_lock(&list);
item->forw->back = item->back;
item->back->forw = item->forw;
spin_unlock(&list);
```

7.7　条件变量

条件变量是一个与某些共享数据的谓词 predicate 相关的更复杂的机制，(结果为 TRUE 和 FALSE 的逻辑表达式)，它允许线程因其阻塞，并在谓词结果改变时，提供唤醒一个或全部线程的手段，相对于等待资源锁定，它更适合于等待事件。

例如，考虑一个或多个服务器线程等待客户请求。进来的请求被传递给等待线程或当没有服务的线程时放入队列中，当有服务器线程准备处理下一个请求时，它首先检查队列。若其中有挂起的消息，线程从队列中取出它，并为其服务。若队列为空，线程阻塞直到有请求到达。这可以通过将一条件变量同队列相关联实现。共享数据是消息队列本身，谓词是队列为非空。

条件变量与睡眠通道相似，因为线程阻塞于条件上而进来的消息唤醒它们。然而，在多

处理器情况下，我们要对某些竞争条件进行保护，如唤醒丢失问题；假设一个消息在线程检查消息队列之后，线程阻塞之前到达。尽管消息已到达线程也会阻塞。因此我们需要一个原子操作完成测试谓词并在必要时阻塞线程。

条件变量通过使用一个额外的互斥量(通常为自旋锁)提供这种原子性。互斥量保护共享数据，避免如唤醒丢失问题。服务线程在消息队列中取得互斥量，然后检查队列是否为空。若为空，则它以获取的自旋锁为参数调用条件变量的 wait()函数。wait()函数以互斥量为参数，原子地阻塞线程，并释放互斥量。当消息到达队列，线程被唤醒，wait()调用在返回前重新取得自旋锁。实例 7-8 提供了条件变量的简单实现。

实例 7-8　条件变量的实现

```
struct condition  {
    proc * next;    /* doubly linked list */
    proc * prev;    /* of blocked threads */
    spinlock_t listLock;    /* protects this list */
};
void wait(condition *c,spinlock_t *s)
{
    spin_lock(&c —>listLock);
    add self to the linked list;
    spin_unlock(&c->listLock);
    spin_unlock(s);    /* release spinlock before blocking */
    swtch();           /* perbrm context switch */
    /* When we return from swtch, the event has occurred */
    spin_lock(s);      /* acquire the spin lock again */
    return;
}
void do_signal(codition *c )
/* Wake up one thread waiting on this condition */
{
    spin_lock(&c->listLock);
    remove one thread from linked list, if it is nonempty;
    spin_unlock(&c->listLock);
    if a thread was removed from the list, make it runnable;
    return;
void do_broadcast(condition *c )
/* Wake up all threads waiting on this condition */
{
    spin_lock(&c->listLock);
    while(linked list is nonempty)    {
       remove a thread from linked list;
       make it runnable;
    }
    spin_unlock(&c->listLock);
}
```

7.7.1　实现问题

有几个关键点需要注意。谓词本身并不是条件变量的一部分，它必须在调用 wait()之前由例程测试。进一步，注意实现中使用两个分开的互斥量。一个是 listLock，它保护阻塞在条件上的线程的双向链表。第二个是用于保护被测试数据自身的互斥量。它不是条件变量的一部分，但被作为参数传递给 wait()函数。swtch()函数和使线程进入就绪状态的代码使用第三个互斥量保护调度队列。

因此，我们会有这种情况发生，一个线程当它占有一个自旋锁时试图获取另一个。这种情况并不是灾难性的，因为自旋锁上的限制要求线程在占有一个自旋锁时不允许阻塞。死锁通过维护一个严格的锁顺序可以避免——谓词上的锁必须先于 listLock 获取。

阻塞线程的队列并非必须成为条件结构的一部分。相反，我们可以拥有一个传统 UNIX 中的一组全局睡眠队列。在这种情况下，条件中的 listLock 被替换为保护相应睡眠队列的互斥量。正如前面所讨论的两种方法各有优点。

条件变量最大的优点之一在于它提供了两种处理事件完成的方式。当事件发生时，唤醒方式的选择或者用 do_signal()只唤醒一个线程，或者用 do_broadcast()唤醒所有线程，二者适用于不同场合。当用于服务器应用程序时，因为每个请求只需用一个线程处理，唤醒一个线程就足够了。然而，考虑几个线程运行同一程序，即分享程序正文的单一拷贝。不止一个程序访问同一个未驻留正文页，将造成它们都产生页失效。失效的第一个线程为此页初始化一个磁盘访问。其他线程注意到 read 已经发出，阻塞等待 I/O 完成。当页面被装入内存，因为对页面的所有访问并不存在冲突，所以可调用 do_broadcast()唤醒所有阻塞的线程。

7.7.2 事件

谓词条件常常是很简单的。线程等待某个特定任务完成，完成通过设置一个全局变量来标志。这种情况最好用称为事件的更高层抽象表示，事件将 done 标志(由自旋锁保护)和条件变量组合成一个单一对象。事件对象有一个简单接口，允许两个基本操作——awaitDone()和 setDone()。awaitDone()阻塞直到事件发生，而 setDone()标识事件已经发生并唤醒所有阻塞在它上的线程。另外，界面也支持非阻塞的 tesrDone()函数和 reset()函数(它将事件标识为未完成)。在某些情况下，布尔值 done 标志也会被事件发生时所能返回的、有更详细信息的变量代替。

7.7.3 阻塞锁

有时，一个资源必须被锁定很长一段时间，而持有这个锁的线程，必须允许阻塞在其他事件上，因此，需要资源的线程在资源可用之前不能使用自旋锁，而必须阻塞。这就需要阻塞锁的原语，它提供两个基本操作——lock()和 unlock()——以及可选的 trylock()。而且有两个对象用于同步——资源的 locked 标志和睡眠队列-.一这意味着我们需要自旋锁保证操作的原子性。这些锁可用条件变量实现，并用谓词实现锁标志的清除。由于性能原因，阻塞锁可以被称为基本原浯。特别地，若每个资源有自己的睡眠队列，单一自旋锁可以同时保护标志和队列。

7.8 读 写 锁

尽管资源的修改要求独占访问,但通常是允许几个线程同时对一个资源进行读操作的(只要没有其他进程同时试图写)，这要求复杂的锁机制允许共享和独占的两种访问方式。而这种手段是建立在简单锁和条件变量基础上的[Birr 89]。在我们分析实现之前，首先考察一下所希望的语义。读写锁可以允许单个写者多个读者。基本操作是 lockShared()，lockExclusive(),unlockShared()和 unlockExclusive()。另外,可能还要 tryLockShared()和 tryLockExclusive()，它们不阻塞而返回 FALSE，同时有 upgrade()和 downgrade()，它们处理共享锁和互斥锁之间的相互转换。loekShared()操作在有排它锁的情况下必须阻塞，而 lockEx-elusive()无论在有排它锁，还是有共享锁的情况下都要阻塞。

7.8.1 设计考虑

一个线程在释放锁时应做些什么呢?传统 UNIX 的方法是唤醒所有等待资源的线程。显然这是低效的——若下面是一个写者取得锁，其他读者和写者都必须返回睡眠，若是读者取得锁，其他写者都将返回睡眠。因此需要找到一个协议避免无意义的唤醒。

若一个读者释放了锁，在有其他活动读者的情况下它将不采取任何行动。若最后一个活动读者释放了它的共享锁，它必须唤醒一个等待写者。

若一个写者解放了它的锁,它必须选择是唤醒另一个写者还是其他读者(假设同时有读者和写者等待资源)。若写者取得优先权，在猛烈竞争下，读者可能会无限制的饥饿下去。所以被选择的解释方案是当释放排它锁后唤醒所有等待的读者。若无读者等待，唤醒单一写者。

而这种方法会导致写者饥饿，若有一个连续的读者流，它们会保持资源的读锁状态，写者将永不能取得锁。为避免这种情况，当写者等待时，lockShared()请求必须阻塞，尽管资源当前处于读锁状态。这种解决方案，在重负担下，会在独立的写者和批量读者之间切换访问。

upgrade()函数必须非常小心以避免死锁。如果实现中不注意，将更新请求优先于等待的写者，死锁可能发生。若两个线程试图同时升级一个锁，由于对方握有共享锁，二者都将阻塞。一种避免的方法是当线程无法立即取得排它锁,upgrade()函数在进入阻塞之前释放共享锁。但这又会给用户造成另外的问题，因为另一个线程在 upgrade()返回之前可能已经改变了对象。另一个解决方法是若有另一个线程挂起升级，upgrade()返回失败并释放共享锁。

7.8.2 实现

实例 7-9 实现一个读写锁的程序

```
struct rwlock  {
     int nActive;    / * num of active readers,or -1 if a writer is active */
     iht nPendingReads;
     int nPendingWrites;
     spinlock_t sl;
     condition canRead;
     condition canWrite;
};
void lockShared (struct rwlock * r)
{
     spin_lock (&r->sl);
     r->nPeningReads++;
     if (r->nPendingWrites > 0)
          wait(&r->canRead,&r->sl);      /* don't starve writers */
     while (r->nActive < 0)                /* someone has exclusive lock */
          wait(&r->canRead,&r->sl);
     r->nActive++;
     r->nPendingReads——;
     spin_unlock (&r->sl);
}
void unlockShared (struct rwlock * r)
{
     spin_lock (&r->sl );
     r-> nActive——;
```

```
     if (r->nActive == 0) {              /* no other readers /
          spin_unlock (&r->sl);
          do_signal (&r->canWrite);
     } else
          spin-unlock (&r->sl);
}
void lockExclusive (struct rwlock * r)
     spin_lock (&r->sl );
     r-> nPendingWrites++ ;
     while (r->nActive);
     wait (r -> canWrite, &r->sl );
     r->nPendingWrites—— ;
     r->nActive = -1;
     spin_unlock (&r->sl);
}
void unlockExclusive (struct rwlock * r)
{
     boolean_t wakeReaders I
     spin-lock (&r->sl);
     r->nActive = 0;
     wakeReaders= (r->nPendingReads ! = 0);
     spin_unlock (&r->sl);
     if (wakeReaders)
        do_broadcast (&r->canRead); /* wake all readers */
     else
        do_signal (&r-canWrite);  /* wake a single writer */
}
void downgrade (struct rwlock * r)
{
     boolean_t wekeReaders;
     spin_lock (&r—— >sl);
     r->nActive = 1;
     wakeReaders = (r->nPendingReads ! = 0);
     spin_unlock (&r->sl);
     if (wakeReaders)
          do_broadcast (&r->canRead); /* wake all readers */
}
void upgrade (struct rwlock * r)
{
      spin_lock ( &r->sl);
      if (r->nActive= = 1) {                  / * no other reader * /
          r->nActlve ——;
      } else {
          r->nPendingWrites ++;
```

```
            r->>nActive——;                    /* release shared lock */
            while (r->nActive)
                  wait (&r->canWrite ,&r->sl);
             r->nPendingWrite——;
             r->nActive= -1;
      }
      spin_unlock (&r->sl );
}
```

7.9 引用计数

尽管锁可用来保护对象内的数据，我们经常需另一种机制保护对象自身。许多内核对象动态的分配和释放。若一个线程释放这样的对象，其他线程没有方法知道它，因此可能试图使用直接指针(它以前取得的)访问对象。同时，内核可能已将内存分配给其他对象，导致严重的系统崩溃。

若一个线程拥有一个指向对象的指针，它希望指针保持可用直到这个线程放弃它。内核可以通过将这种对象和引用计数器相结合保证这点。当初次分配一个对象时(即产生了第一个指针)，内核设置计数为 1。每次产生一个指向对象的新指针，都会增加计数。

这样，当一个线程取得一个指向对象的指针，它实际取得了一个引用。线程有责任在不再使用引用时释放它。同时，内核减少对象引用计数。当计数为 0，即没有线程访问这个对象，内核就可以释放这个对象了。

例如，文件系统维护 v 节点的引用计数(v 节点记录着活动文件信息，见 8.7 节)。当用户打开一个文件，内核返回文件描述符，它构成了 v 节点的引用。用户进行 read 和 write 系统调用中使用这个描述符，允许内核无需重复的进行名字翻译从而提高访问文件的效率。当用户关闭文件时，引用被释放。若几个用户同时打开一个文件，它们引用同一个 v 节点。当最后一个用户关闭文件，内核释放 v 节点。

前面的例子说明引用计数在单处理器系统中也适用。它们本质上更适用于多处理器，因为若没有合适的引用计数,一个线程可能释放一个对象而另一个处理器上的线程正在访问它。

7.10 其他考虑

在设计复杂的锁以及使用这些锁时，还要考虑一些其他因素。本节考察一些重要问题。

7.10.1 死锁避免

一个线程经常需要占有多个资源锁。例如 7.7 节中描述的条件变量实现中使用了两个互斥锁：一个保护数据和条件谓词，而另一个保护阻塞于条件的线程链表。如图 7-8 所示，试图获取多个锁会导致死锁。线程 T1 占有资源 R1，并试图获取资源 R2。同时，线程 T2 可能占有 R2，并试图取得 R1。两个线程都不能前进。

图 7-8 使用自旋锁时可能造成的此锁

两个常用的死锁避免技术是层次锁和随机锁。层次锁在所有有联系的锁上强加一个次序，并要求所有线程以这种次序获取锁。例如，在条件变量中，一个线程必须在对链表加锁之前对条件的谓词加锁。只要严格遵守次序死锁就不会出现。

但也存在必须违反次序的情况。考虑一个高速缓冲区实现，在一个双向链表上以最近最少使用(LRU)顺序储存表维护磁盘缓冲区，所有未处于调用使用的缓冲区在 LRU 链表上。一个自旋锁同时保护队头和队列中缓冲区的向前向后指针。每个缓冲区也有一个自旋锁保护缓冲区中的其他信息。当缓冲区被调用使用时，锁必须能被取得。

当线程需要某个特殊的磁盘块,它确定缓冲区位置(使用散列队列或指针,与本讨论无关)

并锁定缓冲区。然后它锁定 LRU 链表，以便将缓冲区从中移走。因此，这个一般的锁定顺序是“先缓冲区，后队列”。

有时，线程需要一个自由的缓冲区，并试图从 LRU 链表头取得。它首先锁定链表，然后锁定链表上的第一个缓冲区，并从链表中移走它。这样，由于锁定链表先于锁定缓冲区，颠倒了锁定顺序。

因此，可以很容易看出死锁发生的可能。某线程在链表头锁定缓冲区并试图锁定链表。同时，另一个线程已锁定链表并试图锁定缓冲区。二者都等待对方释放锁。

内核使用随机锁处理这种情况。当线程试图取得一个违反层次顺序的锁，它使用 try_lock()操作替代 lock()。这个函数试图取得锁，但当锁已被占用时它返回失败而不是阻塞。在这个例子中，希望取得自由缓冲区的线程将锁定链表，然后进入链表，使用 try_lock()直到它找到一个能够加锁的缓冲区，实例 7-10 描述了自旋锁的 try_lock()的一个实现。

实例 7-10 try_lock()的实现

```
int try_lock(spinlock_t *s)
{
if(test_and_set (s)!= 0)    /* already locked */
    return FAILURE;
else
    return SUCCESS ;
}
```

7.10.2 递归锁

当一个已经拥有锁的线程试图再次获取该锁时，若不阻塞则称锁为递归的。为什么要有这样一个特性呢?为什么一个线程会试图锁定一个它已经锁住的东西呢?典型情况是一个线程锁定一个资源，然后调用一个更低的子例程在它上面操作，而这个低层例程也可能被那些高层的，但并未锁住资源的例程调用。因此，低层例程并不知道资源是否被锁住。若它试图锁住资源，会出现单进程死锁。

当然，这种情况也可借助辅助参数，通过明确通知低层例程关于锁的状态予以避免。然而，这会破坏许多已经存在的接口，并由于有时更低的例程是经过几级调用才得到的，所以这种方法很别扭。导致接口完全没有模块化。一个可行的方法是允许锁递归，但这增加了开销，因为锁必须存储属主 ID 的某个形式，并一般在阻塞或拒绝一个请求时检查它。但更重要的是，它允许函数自己处理自己的加锁请求，而不必关心调用者拥有哪个锁，这样就会有一个原始的模块化接口。

使用这种锁的例子是 BSD 文件系统中(ufs)的目录写操作。例程 usf_write()同时处理向文件和目录写。对文件的许多请求一般通过文件表项访问文件，它直接给出文件 v 节点的指针。这样，v 节点被直接传给 ufs_write()，ufs_write 必须对其加锁。然而，为得到所写目录的 v 节点，需通过路径名遍历例程取得，而它所返回的 v 节点已处于锁定状态。当 ufs_write()调用这个节点，若锁不是递归的将会发生死锁。

7.10.3 阻塞还是自旋

许多复杂锁可用阻塞锁也可用复杂自旋锁实现，而不影响它们的功能或接口。考虑一个被复杂锁保护的对象(例如信号灯或读写锁)。本章中的大多数实现巾，若一个线程试图获取一个对象，并发现它被加锁，线程阻塞直到对象被释放。线程同样可以简单的进入忙等，并仍保持锁的语义。

使用阻塞和忙等的选择是由性能因素决定的。由于忙等占据处理器，一般不受欢迎。然而，某种情况必须使用忙等。若线程已经取得一个简单互斥量，它不允许阻塞。若线程试图取得另一个简单互斥量,它要忙等 ;若它试图取得一个复杂锁,它将释放它所占有的互斥量(例

如条件)。

然而，睡眠和唤醒是花费很高的操作，包括在结束时的环境切换和维护睡眠调度队列，正如忙等很长时间取得锁是荒谬的，在一个很快可以得到的资源上睡眠也是低效的。

此外，一些资源视情形的不同会有短期锁和长期锁。例如，内核会在内存中保留自由磁盘块的一部分链表。当链表变空，它必须从磁盘补充。在大多数情况下，当在内存中添加或删除项时链表需要短时锁定。当磁盘 I/O 被请求，链表则必须锁定很长时间。这样，无论是单独使用自旋锁或阻塞锁都不是好的解决方案。一个可选方法是提供两个锁，阻塞锁只有当链表被填充时使用。然而，这也提出更灵活锁原语的要求。

这个问题可以通过在锁中设置一个能明确说明线程要用自旋锁还是阻塞锁的参数来解决。这个参数由锁的所有者设置，并当试图取得锁的操作没有成功时检查。参数可以是建议的或强制的。

一个可选的解决方法是 Solaris2.x [Eykh 92]中的自适应锁，当线程 T1 试图取得一个被另一个线程 T2 占用的自适应锁，它检查 T2 是否在当前任何处理器上活动。若 T2 活动，T1 执行忙等。若 T2 阻塞，T1 也阻塞。

7.10.4 锁什么

锁可以保护几种对象——数据，谓词，不变量，或操作。例如，读写锁保护数据。条件变量与谓词相关联。不变量与谓词相似，但有一些不同的语义。当一个锁保护不变量，它意味着不变是 TRUE，除非当锁被占用。例如，链表在添加或删除元素时，链表使用单个锁。被这个锁保护的不变量说明这个链表处于一致状态。

最后，锁可以控制对一个操作或函数的访问，这可以限制任一时刻代码只能在最多一个处理器上执行，尽管使用不同的数据结构。同步的监视器模型[Hoar 74]是基于这种方法。许多 UNIX 变体使用主处理器用于内核中的非并行部分(非多处理器安全指令),这样使代码的访问线性化。这种方法通常导致严重的瓶颈 ’，应尽可能避免。

7.10.5 粒度和持续时间

系统性能极大的依赖于锁的粒度，极端情况下，某些非对称处理器在主处理器上运行所有内核代码，这样对于整个内核有单个锁③，另一个极端是系统可以使用很小粒度锁，每个数据变量有一个单独的锁。显然二者都不是理想的解决方案。锁会消耗大量的内存，由于不断取得和释放锁性能会下降。同时，由于有如此多的对象，确定锁定顺序是很困难的，这会导致增加死锁的可能性。

正如通常一样，理想的解决方案是在二者之间，但并没有统一的意见，粗粒度锁的提倡者[Sink 88]建议使用少量锁保护主要系统，并在系统出现瓶颈的地方加入更小粒度的锁。然而，有些系统(例如 Mach)使用小粒度锁结构，并将锁同单个数据对象相连。

锁的持续时间也必须仔细考虑。最好占有锁的时间越短越好，以便减少在锁上的竞争。然而，有时这也会造成额外的加锁和解锁。假设一个线程需要在对象上进行两个操作，都需要对它加锁，在两个操作之间，线程做一些其他事情。它可以在第一个操作之后开锁并在第二个操作的时候再次锁定它。但当其他工作很短时，保持对象在一直加锁会更好。这种决定必须依具体情况而定。

7.11 例子分析

7.5～7.8 节描述的原语构成了一种相当完整的集合，从中一个操作系统可充分混合并匹配以提供一个全面的同步界面。本节介绍 UNIX 主流多处理变体的同步设施。

7.11.1 SVR 4.2/MP

SVR4.2/MP 是 SVR4.2 的多处理器版本，它提供 4 种类型的锁——基本锁，睡眠锁，读写锁和同步变量[UNIX 92]。每个锁必须诵过 xxx_ALLC 和 xxx_DEALLOC 操作明确的分配和释放

(其中 xxx_是类型前缀)。分配操作中使用了调试用参数。

基本锁

基本锁是允许短期锁定资源的非递归互斥锁。它不允许在阻塞期间占用，它以 lock_t 类型变量的形式实现，并用下列操作实现加锁和解锁。

```
pl_t LOCK(lock_t *lockp, pl_t new_ipl);
UNLOCK(lock_t *lockp, pl_t old_ipl);
```

lock 操作在取得锁之前将优先级升为新的中断优先级，并返回原优先级。这个值必须被传给 UNLOCK 操作，以便将中断优先级恢复为原值。

读写锁

读写锁是一个非递归锁，允许短期加锁，并具有单个写者，多个读者的语义。它不允许在阻塞操作中被持有。它以类型 rwlock_t 的变量形式实现，并提供下列操作。

```
pl_t RW_RDLOCK(rwlock_t * lockp, pl_t new_ipl);
pl_t RW_WRLOCK(rwlock_t * lockp, pl_t new_ipl);
void RW UNLOCK(rwlock_t * lockp, pl_t old_ipl);
```

中断优先级的处理同基本锁的处理相同。锁操作把中断优先级提高到指定的优先级，并返回以前的中断优先级。RW_UNLOCK 恢复中断优先级为原值。锁也提供非阻塞操作 RW_TRYRDLOCK 和 RW_TRYWRLOCK。

睡锁

睡锁是一个允许长期锁定资源的非递归互斥锁。它可以在阻塞期间一直被持有，它以类型 sleep_t 的变量实现，并提供下列基本操作：

```
void SLEEP_LOCK(sleep_t * lockp, int pri);
bool_t SLEEP_LOCK_SIG(sleep_t * lockp, int pri);
void SLEEP_UNLOCK(sleep_t * lockp);
```

pri 参数设定进程被唤醒后进程被赋予的调度优先级。若进程在调用 SLEEP_LOCK 时阻塞，它将不会被信号中断。若它在调用 SLEEP_LOCK_SIG 时阻塞，信号会中断进程；若锁被取回，调用返回 TRUE：若睡眠中断返回 FALSE。锁也提供其他操作，如 SLEEP_LOCK_AVAIL(检查锁是否可用)，SLEEP_LOCKOWNED(检查调用者是否拥有锁)，和 SLEEP_TRYLOCK(若锁未得到，返回失败而不是阻塞)。

同步变量

同步变量与 7.7 节讨论的条件变量相同。它以 sv_t 类型的变量实现，而且它的谓词(由用户独立管理)被基本锁保护，它支持下列操作：

```
void SV_WAIT(sv_t *svp, int pri, lock_t * lockp);
bool_t SV_WAIT_SIG(sv_t *svp, int pri, lock_t * lockp);
void SV_SIGNAL(sv_t *svp, int flags);
void SV_BROADCAST(sv_t *svp, int flags);
```

和睡锁一样，pri 参数指定调度在进程被唤醒后赋予进程优先级，而 SV_WAIT_SIG 允许被一个信号中断。lockp 参数用于传递一个保护谓词的基本锁指针。调用者必须在调用 SV_WAIT 或 SV_WAlT_SIG 之前取得 lockp，内核自动的阻塞调用者并释放 lockp。当调用者从 SV_WAIT 或 SV_WAIT_SIG 返回，lockp 不被占用，渭词在调用者阻塞后不保证为真。因此，对 SV_WAIT 或 SV_WAIT_SIG 的调用应被包含在一个 while 循环中，并每次都检查谓词。

7.11.2 Digital UNIX

Digital UNIX 的同步原语派生于 Mach。共有两种类型的锁——简单锁和复杂锁[Denh 94]。简单锁是基本自旋锁，使用机器的原于指令 test-and-set 实现。在使用之前，它必须被声明和初始化。它必须初始化成解锁状态，并且保证在阻塞期间或环境切换中不能被持有。

复杂锁是支持许多特性的高层次抽象，这些特性包括共享和独立访问，阻塞和递归锁。它是一个读写锁，并提供两种选择——睡眠和递归。睡眠选项可以在初始化后任何一个时刻启动或关闭。设置后，若无法保证立即取得锁，内核将阻塞请求。此外，若希望在阻塞期间占用锁，睡眠选项必须可用。递归选项只能被那些已经取得互斥锁的同一线程设置；也只能由设置它的同一线程清除。

该接口提供各种例程的非阻塞版本，即若锁无法立即取得，它返回失败。同时也包括用于升级(共事到独占)或降级(独占到共享)的函数。若有其他挂起的升级请求，升级例程将释放共享锁，并返回失败。非阻塞版本的升级返回失败，但并不放弃共享锁。

睡眠和唤醒

由于有大量的代码是从 4BSD 移植过来的，因此非常希望保留基本函数 sleep()和 wakeup()。因此，阻塞线程是放在全局队列上而不是每个锁一个队列。算法需要改变以便在多处理器中运行，这里的主要问题是丢失唤醒问题。为了修正这个问题同时又保留原有的线程状态框架，sleep()函数用两个更低级的原语——assert_wait()和 thread_block()重写，如图 7-9 所示。

图 7-9 Digital UNIX 中的睡眠实现

假想一个线程需等待一个事件，它由谓词和自旋锁保护。线程取得自旋锁后，再测试谓词。若线程需要阻塞，它调用 assert_wait()，把自己放到合适的睡眠队列。然后它释放自旋锁并调用 thread_block()初始化一个环境切换。若事件发生在释放自旋锁和环境切换之间，内核将线程从睡眠队列中移走，并将它放在调度队列上。因此，线程不会发生丢失唤醒。

7.11.3 其他实现

SVR 的最初多处理器版本是在 NCR 开发的[Camp 9l]。这个版本引入了建议处理器锁(APL，advisory processor l ocks)的概念，它是一个包含有关竞争线程提示的递归锁。提示指定当线程竞争时使用自旋还是睡眠，以及这个选择是建议还是强制。拥有锁的线程可将指示从睡眠变为自旋，或相反。这种 APL 不允许在调用睡眠期间占用，并且主要用于访问较长时间锁的单个线程，APL 突出的特点是它们在环境切换中自动释放和重新获取。这意味着传统的睡眠/唤醒接口无需改变即可使用。此外，同种类型的锁可通过睡眠避免死锁，因为睡眠释放以前拥有的全部锁，实现也提供非递归自旋锁和读写 APL。

NCR 版本由 Intel Multiprocessor Conmrtium(它是由一组致力于开发 SVR4 的正式多处理器版本的公司组建的)改写[Peac 92]。一个重要的改变包括取得 APL 锁的函数。这个函数现在也用一个中断优先级作为参数。这允许在占有锁期间提高处理器优先级。若这个锁无法立即取得，在原来(较低的)优先级忙等。函数返回原优先级，它可传给未阻塞函数。

最低层的原语是一组算术和逻辑操作。算术操作允许对引用汁数增加和减少。逻辑函数用于对标识域进行位操作的小颗粒线程。它们都返回它们所操作变量的原始值。在紧接着的稍高层次是简单自旋锁，它们在环境切换时不自动释放。这些被用厂如队列插入或移出的简单操作。最高层的锁是资源锁，资源锁是长期锁，具有单个写者、多个读者的浯义，它们可在阻塞期间占用。实际上也提供同步和异步跨处理器中断，此中断用于诸如分布式时标信号(Clock tick)和地址转换高速缓存相关(见 15.9 节)。

solaris 2.x 使用自适应锁(见 7.10.3 节)和旋转门(见 7.2.3 节)提高性能。它提供信号灯，读-写锁和条件变量作为高层次同步对象。它也使用内核线程处理中断，以便中断处理程序使用内核中其他部分相同的同步原晤，必要时阻塞。3.6，5 节中对这个特性有更细致的讨论。

所有已知的多处理器实现都使用某种形式的自旋锁用于低层次的短期同步。为了避免重写大量的机制，睡眠/唤醒一般被保留，但可能有些改变。主要的区别表现在对高层抽象的选

择上。早期 IBM/370 和 AT&T 3B20A [Bach 84]上的实现几乎完全依赖信号灯，Ultrix[Sink 88]使用阻塞互斥锁。Amdahl 的 UTS 内核[Ruan 90]基于条件变量。DG/UX[Kell89]使用不可分事件计数器实现顺序锁，它提供了一种在某种程度上完全不同的方法唤醒进程。

7.12 小　结

多处理器上的同步问题与单处理器上的同步问题有本质的不同，而且更复杂。有一系列不同的解决方案，例如睡眠/唤醒，条件变量，事件，读写锁，和信号灯。这些原语的大同小异，例如，用条件变量实现信号灯，或相反都是可能的。其中有些方案不仅限于多处理器，也可被用于单处理器和松耦合的分布式系统。由于大多数 UNIX 的多处理器系统是基于已有的单处理器的变体，对于移植的考虑对采用何种抽象有很大的影响，Mach 和基厂 Mack 系统不受这些考虑的约束，这在它们对同步原语的选择中可略见一斑。

7.13 练　习

1.许多系统中有一个原子交换指令，将寄存器的值和某个内存位置的值交换，请说明如何使用这个指令实现原子 test-and-set。

2.如何使用 load-linked 和 store-conditlonal 在机器上实现原子 test-and-set。

3.假设 cotlvoy 的形式是由于在一个被信号灯保护的临界区的严重竞争。若这个临界区被分为两个临界区，每个由单独的信号灯保护，这会减少 convoy 问题吗?

4.减少 convoy 的一种方法是用别的锁机制替代信号灯机制。这个有饿死线程的危险吗?

5.引用计数与共享锁有何不同?

6.实现资源上的阻塞锁，使用自旋锁和条件变量，并用锁标识为谓词(见 7.7.3 小节)。

7.在练习 6 中，当清除标识时用自旋锁保护渭词是否必要?[Ruan 90]讨论 waitlock 操作以改善这个算法。

8.条件变量是如何避免失去唤醒问题的?

9.实现一个事件抽象，当事件完成时，它返回一个状态值等待线程。

10.假设一个对象受到读或写访问。在什么情况下，最好用简单互斥量，而不是读写锁。

11.读写锁是否必须被阻塞?实现一个读写锁，在线程资源己加锁时使线程忙等。

12.描述一种情况，在此情况下锁粒度细化可避免死锁。

13.试举一例，通过降低锁的粒度为粗粒度来避免死锁。

14.对于多处理器内核来说，是否有必要对每个变量或资源都需要加锁?试列举线程无需加锁就可以访问或修改对象的例子。

15.管程[Uoar 74]是一种通过编程语言来支持的对一段代码提供互斥操作的方法。在什么情况下这样做是非常自然的?

16.实现 7.8.2 节中读写锁的 upgrade()和 downgrade()函数。

① 然而，数据，指令和地址转换 cachc 是局部于每个处理器的。

② 某些实现可以根据优先级来选择要唤醒的线程。与在这个例子中这样做的效果相同。

③ 有时，即便是 SMP 系统也会使用主处理器来执行一段非多处理器安全的指令。这个方法就称为漏斗。