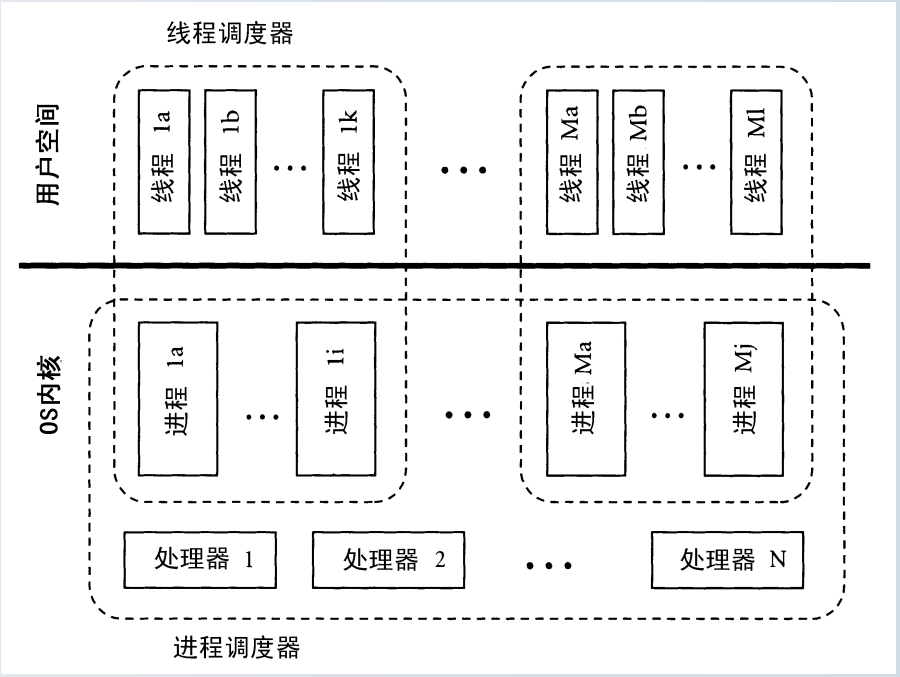
线程的实现

OS 提供一组（不太多）并发执行的进程，作为物理处理器结构的抽象

并发的应用程序的线程通常在一个或几个 OS 进程上实现

- 一种极端情况是每个线程独立占用一个 OS 进程
- 另一极端是所有线程都在唯一的一个 OS 进程上实现
- 常见折中实现方案：一个（或多个）程序里的所有线程（可能非常多）在不多的一组 OS 进程上运行



线程的实现

- 每个线程专用一个进程的主要问题是 OS 进程代价太高
  - OS 进程在内核实现，任何操作都要通过系统调用（需转换程序模式）
  - OS 进程是通用机制，提供了许多线程不需要的特征（独立地址空间、优先级、簿记信息、信号和 I/O 接口等），这些机制代价很大
- 将一个并行程序的所有线程放在一个进程上，也有很多缺点：
  - 排除了在多处理器上并行执行这一程序的可能性
  - 如果当前线程做系统调用时进入阻塞（例如等待I/O），将使唯一的运行进程被 OS 挂起，本程序的其他线程也都不可能运行了

采用两层组织（用户线程在内核进程上实现），两层代码可能类似：语言的运行系统在几个进程上实现线程，就像 OS 在几个处理器上实现进程

为减少同步引起的延迟（同步延迟），多处理器 OS 可能采用一些策略，如：把同一应用的不同线程安排到不同处理器上同时运行（成组调度）；把一组处理器分配给一个特定应用（处理器划分）

线程的实现

可以通过对协程的改造来实现线程，现在介绍相关情况

协程实现顺序控制，一组协程在一个 **OS** 进程上实现，同时处于执行中，但每个时刻只有一个协程正在活动，其余休眠（停在执行中的某状态）

- 标志性操作：当前活动协程将控制显式 **transfer** 给另一协程，本操作的参数是被唤醒协程引用（在实现层，就是该协程的上下文块指针）
- 当协程 **A** 把控制 **transfer** 给协程 **B**，**A** 保持当时状态并进入休眠，而 **B** 从当时休眠状态被唤醒并继续执行

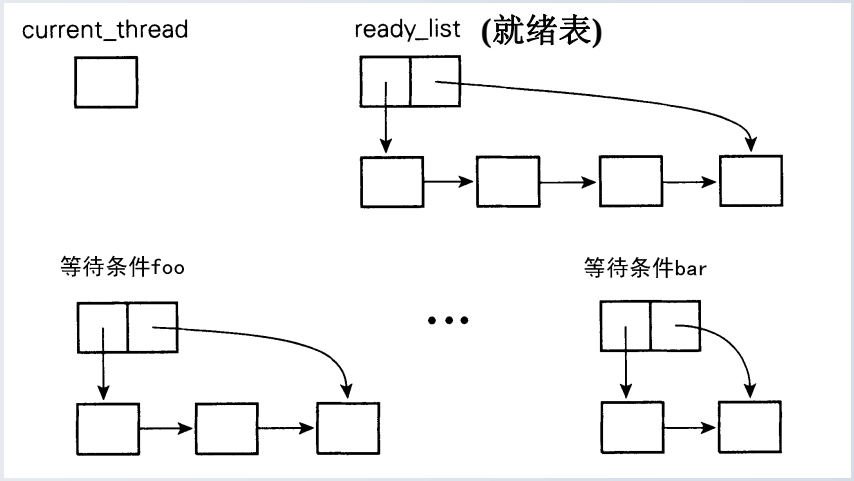
通过三个步骤就能把协程改造为线程：

1. 实现一个**调度器**，以隐藏 **transfer** 的参数。当前活动线程简单地交出对处理器的控制，由调度器去选择下一个进入运行的线程
2. 实现**强占**以常规地（如定时）挂起当前线程，给其他线程运行机会
3. 提供一种支持多个 **OS** 进程共享线程集合的数据结构，使线程可运行在任何（一个或多个）进程上，而这些进程又能运行在不同处理器上

线程的实现：单处理器调度

一个线程可能处于几种状态之一：

- 可运行（就绪）。某（几）个可运行线程正在处理器上实际运行；其他线程在等待运行机会，其上下文块位于一个就绪表队列里
- 被阻塞（等待同步或等待资源）。由于同步需求而阻塞的线程的上下文块驻留在与其等待条件关联的数据结构里（通常也是队列）



线程的实现：单处理器调度

当前线程需要交出处理器时，调用调度器操作 reschedule

```
procedure reschedule
    t : thread := dequeue(ready_list)
    transfer(t)
```

如果线程希望将来再次运行，就在进入调度器前把自己的上下文块放入相关数据结构。如果只是让其他线程有运行机会，就放入就绪表

```
procedure yield
    enqueue(ready_list, current_thread)
    reschedule
```

如果是因为同步阻塞交出处理器，就放入与等待条件相关的队列

```
procedure sleep_on(ref Q : queue of thread)
    enqueue(Q, current_thread)
    reschedule
```

如果当前线程执行的操作使某个条件变真，就会取出相关等待队列等待的一个或多个线程，并将它们加入就绪表

线程的实现：单处理器调度

如果一个线程运行时还有其他可运行线程，就出现了公平性问题

- 单处理器系统要表现“并行的感觉”，保证可运行线程能经常实际运行

在合作模式中，长时运行的线程必须不时交出处理器（如在自己代码中循环每次开始之前），使其他线程能运行

- 不良线程可能独占处理器。即使没有不良线程，不同线程交出处理器的操作间隔也可能不同，也会产生实际的非公平性

更好的办法是按某种小时间粒度（如每秒若干次）公平安排处理器，不需要线程显式调用 **yield**。许多系统利用时钟信号实现强占式多线程：

- 时钟信号导致线程切换，它导致控制进入 **OS**
- 中断处理程序将当前进程上下文（寄存器和 **pc**）入栈，然后调用一个特殊处理程序去修改相应线程的状态，使其就像刚调了 **yield** 操作
- 该处理程序“返回”到“前面”**yield** 操作，由该操作把控制传给另一线程，就像前一线程自愿把对处理器的控制交给后一线程

线程的实现：单处理器调度

时钟信号可能在任何时刻发生，因此可能出现对调度器的自愿调用（如要求等待IO）和由强占触发的自动调用之间的竞态。例如：

- 假定时钟信号到达时当前线程刚好在 **yield** 里把自己放入就绪队列，正要调用 **reschedule**；信号处理器完成工作“返回”进入 **yield** 将把当前线程第二次放入就绪表。如果后来该线程由于同步而阻塞，就绪表里它的第二个上下文块可能使它又投入运行（实际上这时它应该等待）

  如果信号出现在 **enqueue** 操作执行期间，就可能破坏就绪队列

- 为消除竞态保护就绪表，线程包通常禁止在调度器调用期间发信号

```
procedure yield
    disable_signals
    enqueue(ready_list, current_thread)
    reschedule
    reenable_signals
```

调用 **reschedule** 的**每处**代码都必须在调用前禁止信号，调用后再允许信号

线程的实现：单处理器调度

`sleep_on` 操作也要求调用者禁止和允许信号。出问题的场景：

- 假定线程 **A** 检查某个条件并发现其为假，就准备调用 **sleep_on** 把自己挂入该条件的队列；假定在条件检查和调用 **sleep_on** 之间出现时钟信号；再假定信号后被允许运行的线程 **B** 把该条件变为真
- **A** 还没把自己放入队列，因此 **B** 不可能找到 **A** 并将它重变成可运行的。**A** 重新运行时将立即挂起自己，而且可能永远不会再被唤醒

为保证不会出这种问题，线程必须在检查条件前禁止信号

```
disable_signals
if not desired_condition
    sleep_on(condition_queue)
reenable_signals
```

在单处理器系统，禁止信号能保证检查和睡眠动作成为一个**原子**操作，也就是说，从其他线程的角度看这种操作总是“一下子”做完

线程的实现：多处理器调度

- 多数并发语言都允许线程的真并行运行

  从程序员角度看，真并行与基于时钟中断在不同线程间切换的“准并行”没有实质差别。线程都需要显式同步，以正确处理各种竞态条件

- 扩充单处理器的强占式并行包，使之能运行在多个 OS 进程上并不困难

  这时需要安排多个进程共享一个就绪表和其他相关数据结构（条件队列等）。每个进程需要有一个独立的 current_thread 变量

  如果是共享存储器的多处理器系统，就可能让多个线程同时运行。只要应用程序有一个进程没在 OS 里阻塞，该程序就可能进展

- 可运行线程都在就绪表里，负责该应用程序的任一进程都可执行它们

  某进程调用 reschedule 时，可以把就绪表里等待时间最长的线程给它

  更巧妙的调度器可以根据需要选择线程，如：1，选择交互式线程；2，优先级最高的线程；3，选择时间攸关的线程；4，选择上次在这个处理器上运行的线程（其数据可能还在缓存或内存里）；等等

共享存储器模型

对共享存储器并发程序，最主要语义问题是同步。常见同步有两种：

- 互斥：保证在任何时刻至多一个线程执行临界区（关键代码段）代码
- 条件同步：保证除非满足某些条件，否则给定线程不会继续执行（例如，直至某变量有了某个特定值等）

虽然互斥也可以看作条件同步（等到没有其他线程位于临界区），但

- 互斥时的同步需要知道当前所有线程的状况
- 条件同步通常不会提供这么强的功能（以有效处理一类常见情况）

并行线程实现中实例：

- 某个进程修改就绪表时，其他进程不能去读/写就绪表（互斥）
- 某进程需要运行一个线程时，它必须等到就绪表不空（条件同步）

注意：我们不希望过度同步/互斥，因为这样必然减少并行的可能性

需要关注必要的同步，消除“坏的”竞态条件（保证程序产生正确结果）

共享存储器：忙等待

忙等待条件同步比较简单：

- 把等待的条件改造为“位置*X*包含值*Y*”，在等待这个条件的线程（或进程）里写一个循环，其中不断读位置 *X* 直到得到值 *Y*
- 为此要求硬件的装载和保存指令是原子操作，存储器和/或总线能把处理器和设备的并发访问顺序化（几乎所有计算机都支持）

忙等待互斥更为困难，下面讨论的**自旋锁**用于实现互斥

Dekker 提出了一个两线程互斥算法，其中只要求装载/保存指令是原子操作

Dijkstra 在 **1965** 年发表了 ***n*** 个线程算法

Peterson 在 **1981** 年发表了一个大大简化的两线程互斥算法。可用它构造一种层次的 ***n*** 线程锁。线程进入临界区需要 ***O*(*n* log *n*)** 空间和 ***O*(log *n*)** 时间

Lamport 在 **1987** 年发表了一个 ***n*** 线程算法，没有锁竞争时需要 ***O*(*n*)** 空间和 ***O*(1)** 时间。如果多个线程都想进入临界区，算法需要 ***O*(*n*)** 时间

共享存储器：忙等待

要实现常量时间互斥，需要更强大的原子指令。从 **1960** 年代起处理器开始提供这种指令，它们能在一个原子操作中读、修改或写一个存储单元

最简单的读-修改-写指令是 **test_and_set**：将一个布尔变量置为 **true** 并返回该变量原来是否**false**的指示。用 **test_and_set** 很容易实现自旋锁

```
while not test_and_set(L)
    -- nothing -- spin
```

如果多个处理器都想获取锁，循环里的 **test_and_set** 就可能造成巨大的总线信息流。这种对硬件资源的过度需求称为**争用**，是大型机的主要性能障碍

用 **test-and-test_and_set** 锁可减少争用：

循环里用常规读操作直至看到锁已空闲

当某个线程释放锁时，正在等待的线程都会去执行 **test_and_set** 指令，这将会导致一阵总线活动。但影响比较局部

```
type lock = Boolean := false;

procedure acquire_lock(ref L : lock)
    while not test_and_set(L)
        while L
            -- nothing -- spin
procedure release_lock(ref L : lock)
    L := false
```

忙等待：读者写者锁

互斥的一种重要变形是**读者写者锁**。用于保护共享数据，其使用分为：

- 读者，这种使用中只是提取数据，不修改
- 写者，这种使用是修改数据

如果有几个线程同时希望去读同一个数据结构，读者写者锁允许这些线程同时做，因为它们不会相互干扰

- 允许多个线程同时执行读数据操作（允许与其他线程同时读）

当某线程要**写**这个数据结构时，必须防止其他线程在此同时去读或写

- 写操作必须获得对于数据的独占权（只允许一个写操作执行，它需要与其他读操作或写操作互斥）

多数忙等待互斥锁都可以扩充为允许读者同时访问

- 很容易基于它们实现读者写者锁

共享存储器：调度器的实现

为实现用户线程，**OS** 进程访问就绪表和条件队列时需要同步，常用自旋

右图是个简单的**可重入**线程调度器：（可重入：一线程没有返回前，其他线程就可安全“进入”）

进程共用 **scheduler_lock** 和 **ready_list**，各有一个 **current_thread**

如果进程指定了处理器，**low_level_lock** 可用常规自旋锁，否则用“自旋后交出”锁。**reschedule** 里的循环忙等待到就绪表不空

```
shared scheduler_lock : low_level_lock
shared ready_list : queue of thread
per-process private current_thread : thread

procedure reschedule
    -- assume that scheduler_lock is already held
    -- and that timer signals are disabled
    t : thread
    loop
        t := dequeue(ready_list)
        if t ≠ nil
            exit
        -- else wait for a thread to become runnable
        release_lock(scheduler_lock)
        -- window allows another thread to access ready_list
        -- (no point in reenabling signals;
        -- we're already trying to switch to a different thread)
        acquire_lock(scheduler_lock)
    transfer(t)
    -- caller must release scheduler_lock
    -- and reenable timer signals after we return
```

共享存储器：调度器实现

进调度器前要禁止信号，保护就绪表和条件队列不被进程和信号处理器并发访问

schedule_lock 保护调度器（在 **yield** 或**sleep_on** 里）

将上下文块存入队列前必须首先获取锁，从 **reschedule** 返回后必须释放锁

reschedule 调用 **transfer**，因此锁 **schedule_lock** 通常由一线程获取（禁止时钟信号的线程），另一线程释放（新进入运行的线程，它允许时钟信号）

yield 的代码实现同步

```
procedure yield
    disable_signals
    acquire_lock(scheduler_lock)
    enqueue(ready_list, current_thread)
    reschedule
    release_lock(scheduler_lock)
    reenable_signals

procedure sleep_on(ref Q : queue of thread)
    -- assume that caller has already disabled timer signals
    -- and acquired scheduler_lock, and will reverse
    -- these actions when we return
    enqueue(Q, current_thread)
    reschedule
```

不能在 **sleep_on** 代码内部实现同步，其代码不能禁止时钟信号并获取调度器锁。因为线程通常需要用一个原子动作检查条件，根据检查的情况看是否需要阻塞自己

共享存储器：调度器实现

线程调用 **sleep_on** 的代码（右）

把信号和锁操作移到**sleep_on**内就会出现竞态，例如线程 ***A*** 检查条件并发现它为假；线程 ***B*** 使条件变真并发现当时的条件队列为空；这就会使线程 ***A*** 在条件队列上永远睡眠下去

```
disable_signals
acquire_lock(scheduler_lock)
if not desired_condition
    sleep_on(condition_queue)
release_lock(scheduler_lock)
reenable_signals
```

对单一处理器，调度器不需要锁，只需禁止信号

如果每个进程运行在一个处理器上，那么就只需一个自旋锁保护就绪表和条件队列

如果可能在单处理器运行，也可能在比进程个数少的多处理器上运行，无法获得锁时就需要放弃处理器，可以用“自旋后交出”锁

```
type lock = Boolean := false;

procedure acquire_lock(ref L : lock)
    while not test_and_set(L)
        count := TIMEOUT
        while L
            count -:= 1
            if count = 0
                OS_yield            -- relinquish processor
                count := TIMEOUT

procedure release_lock(ref L : lock)
    L := false
```

基于调度器的同步

忙等待同步的进程将一直占用处理器，一些情况下可以这样做：

- 当前处理器没有更有价值的事情可做
- 等待的期望时间小于切换上下文转到其他线程后再切换回来的时间

为得到更好的性能，许多并发语言里采用的是基于调度器的同步机制，在一个线程阻塞时立即切换到另一个线程

- 基于调度器同步，总把等待线程移出就绪表，直至等待条件变为真（或可能为真）时才将它放回。而自旋方式在交出锁后仍执行忙等待：交出处理器后相关线程仍留在就绪表，得到机会就去做循环检测
- 下面考虑三种基于调度器同步的形式：信号量、管程和条件临界区

下面一种用一个**有界缓冲区**实例，说明各种同步机制的语义

有界缓冲区是容量有限的并发队列，**生产者**线程将数据加入，**消费者**线程删除数据。这种缓冲区用于缓和两类线程的相对速度波动，提高系统通量

信号量（semaphone）

信号量是最老的同步机制，由 **Dijkstra** 在 **1960** 年代提出。该机制出现在 **Algol 68**，**SR**，**Modula-3** 和各种库里

信号量有两个操作：

P：计数器减 **1** 后等待直到计数器非负

V：计数器加 **1** 并唤醒一个等待线程

P/V 来自荷兰语的通过和释放。**Algol 68** 称 **down** 和 **up**

通常假定信号量公平：线程完成 **P** 操作的顺序与其继续执行的顺序相同

```
type semaphore = record
    N : integer -- usually initialized to something nonnegative
    Q : queue of threads

procedure P(ref S : semaphore)
    disable_signals
    acquire_lock(scheduler_lock)
    S.N -:= 1
    if S.N < 0
        sleep_on(S.Q)
    release_lock(scheduler_lock)
    reenable_signals

procedure V(ref S : semaphore)
    disable_signals
    acquire_lock(scheduler_lock)
    S.N +:= 1
    if N ≤ 0
        -- at least one thread is waiting
        enqueue(ready_list, dequeue(S.Q))
    release_lock(scheduler_lock)
    reenable_signals
```

基于前面调度器代码的信号量实现

信号量：实例

二值信号量：初值为**1**，其中的 **P** 和 **V** 操作总成对出现

这种信号量可作为互斥锁，**P** 操作获取锁，**V** 操作释放它

通用信号量：初值为某个***k***，用于管理 ***k*** 个副本的资源访问

任何时刻其值总比已发生 **P** 操作次数（***#P***）与已发生 **V** 操作次数（***#V***）之差大 ***k***。调用 **P** 操作的线程阻塞到 ***#P*** < ***#V*** + ***k***

两者描述能力相同，可相互实现

右边代码是有界缓冲区实现，互斥用一个二值信号量，条件同步用两个通用信号量（**计数信号量**）

```
shared buf : array [1..SIZE] of bdata
shared next_full, next_empty : integer := 1, 1
shared mutex : semaphore := 1
shared empty_slots, full_slots : semaphore := SIZE, 0

procedure insert(d : bdata)
        P(empty_slots)
        P(mutex)
        buf[next_empty] := d
        next_empty := next_empty mod SIZE + 1
        V(mutex)
        V(full_slots)

function remove : bdata
        P(full_slots)
        P(mutex)
        d : bdata := buf[next_full]
        next_full := next_full mod SIZE + 1
        V(mutex)
        V(empty_slots)
        return d
```


管程（monitor）

信号量功能强大、使用广泛。其主要问题是太低级：

- **P** 和 **V** 是简单子程序调用，不明显，容易被忽视（控制流复杂时）
- 编程时应该把 **P/V** 操作隐蔽在某种抽象里，否则使用一个信号量的操作会散布在整个程序里的各处，维护时很难追踪

管程由 **Dijkstra** 提出的一种结构化的线程间同步方案，后由 **Brinch Hansen** 深入开发，并由 **Hoare** 做了形式化研究。管程机制出现许多语言里，包括 **Concurrent Pascal**，**Modula(1)**，**Mesa**，**Java** 等

管程就是一个模块或对象，提供一些操作、内部状态和条件变量。性质：

- 任意时刻只允许激活一个操作。如果管程忙，调用其操作的线程阻塞到它空闲。管程操作里通过调用 **wait** 使线程在条件变量上**等待**，通过给条件变量发**信号**唤醒等待线程（如唤醒最早等待的线程）
- 管程操作（**入口**）自动实现互斥，比 **P** 和 **V** 操作更容易使用

管程是一种抽象，将针对封装数据的所有操作汇集在一处，包括同步操作

管程：实例

基于管程的有界缓冲区代码见右，**insert**和**remove**是入口子程序，要求对管程数据进行互斥访问

管程中的条件与存储无关，**insert** 和 **remove** 都在最后产生一个信号

注意：管程的条件变量无“记忆”（与信号量不同）

- 如果执行 **signal** 时没有正等待的线程，该 **signal** 就没有作用
- 执行 **V** 操作总将信号量计数器加 **1**，如当时无等待进程，下面第一个 **P** 操作总成功

```
monitor bounded_buf
imports bdata, SIZE
exports insert, remove

    buf : array [1..SIZE] of data
    next_full, next_empty : integer := 1, 1
    full_slots : integer := 0
    full_slot, empty_slot : condition

    entry insert(d : bdata)
        if full_slots = SIZE
            wait(empty_slot)
        buf[next_empty] := d
        next_empty := next_empty mod SIZE + 1
        full_slots +:= 1
        signal(full_slot)

    entry remove : bdata
        if full_slots = 0
            wait(full_slot)
        d : bdata := buf[next_full]
        next_full := next_full mod SIZE + 1
        full_slots −:= 1
        signal(empty_slot)
        return d
```


管程：定义

Hoare 给出了管程的形式化定义，在其模型里，管程的每个条件变量附有一个线程队列，整个管程有一个**入口队列**和一个**紧急队列**

- 调用忙管程的线程在入口队列等待
- 如果线程在管程里执行 **signal** 操作的时刻有线程在相应条件上等待，发 **s** 信号的线程进入紧急队列，条件队列第一个线程获得控制。如果被 **signal** 的条件上无等待线程，这个 **signal** 就相当于空操作
- 线程离开管程时（无论是完成还是到某个条件等待），首先考虑释放紧急队列里第一个线程，紧急队列为空时释放入口队列里第一个线程

管程的许多实现中去掉了紧急队列，可能还有其他修改。后面讨论

管程的正确实现需要**管程不变式**的概念。不变式描述“本管程状态完整”的逻辑条件。管程初始时和退出时不变式都必须为真，每个 **wait** 语句处也为真。对 **Hoare** 模型的管程，还要求在每个 **signal** 处不变式为真

对于有界缓冲区实例，不变式应断言 **full_slots** 是缓冲区里的数据项数，而且这些项位于编号 **next_full** 到 **next_empty – 1**（取模 **SIZE**）的位置

管程：信号语义

假设 **signal** 某条件时有等待线程。要保证被唤醒线程唤醒后的时刻该条件仍为真，必须立即把控制切换过去，为此就需要有紧急队列暂时保存发出 **signal** 的线程，还要保证管程不变式在 **signal** 操作中维持不变

- 在 **signal** 时切换上下文会带来调度器开销，另一方面，发 **signal** 线程的后续操作通常不会改变与被 **signal** 变量相关联的条件
- 为减小开销并消除 **signal** 处的不变式要求，一些语言采用另一种定义：**signal** 只是**提示**，只要求把一个等待线程移到就绪表，而发 **signal** 的线程继续保存对管程的控制。被唤醒线程真正运行时应重新检查条件

在**Hoare**管程里的标准惯用形式

```
if not desired_condition
    wait(condition_variable)
```

这种情况下需要写成：

```
while not desired_condition
    wait(condition_variable)
```

Mesa/Modula-3 都只把 **signal** 看作提示（**Java/C#** 里的情况也都类似）

它们都不立即执行从 **signal** 线程到被唤醒的等待线程的切换

管程：嵌套调用问题

在大部分管程语言里，如线程在管程操作中成功进入其他管程的操作（嵌套管程调用），后来在嵌套内层执行 **wait**，线程只释放对最内层管程的互斥，但维持外层管程的锁定状态（仍被它占用）

如其他线程到达相应 **signal** 方式都要通过同样外层管程，就会出现**死锁**（死锁是并发程序运行时可能出现的一种状态，其中一组线程相互等待都不能前进）。在这里，先进入外层管程的线程在内层等其他线程的 **signal** 操作，而其他线程则在等它离开（释放）外层管程

让内层管程在等待时释放外层管程互斥，同样可能导致死锁。一个场景：

- 等待线程被唤醒时需要重获内层和外层管程的互斥
- 内层管程自然可用（刚被 **signal**），但外层管程未必空闲。可能某个正用着外层管程的线程需访问内层管程，完成后才能释放外层管程。如果被唤醒线程在 **signal** 后立即进入内层管程，就会出现死锁

避免这个问题的一种方案是要求线程在程序里的所有管程上互斥。这一方案在单处理器上还可以接受，但会严重影响多处理器实现的并发性