并发

第二十八章——锁

刘 玉峰

Fx_163.com 湖南

大学

标签： The Basic Idea

```
1   lock_t mutex; // some globally-allocated lock 'mutex'
2   ...
3   lock(&mutex);
4   balance = balance + 1;
5   unlock(&mutex);
```

- 锁 只是 一个 变量， 因此 要 使用 它， 必须 声明某种类型的 **锁 变量** （ 例如 上面的mutex）。
- 这个 锁 变量 （或者 简称为 “lock” ）在任何时刻都保持锁的 状态 。

标签： The Basic Idea

- 变量保存锁的状态。
  - 可用 （或未锁定或 免费）
    - 没有 线程持有 锁。
  - 获得 （或锁定或 持有）
    - 只有 一个线程持有锁， 并且可能处于临界 区。

lock （）的语义

- lock（）
  - 尝试 获取 锁。
  - 如果 没有 其他 线程 持有 该 锁，则该 线程 将获取该 锁。
  - 进入 *临界 区*。
    - 这个 线程 被 认为 是锁的 所有者 。
- 当持有锁的第一个线程在临界区时，其他线程被阻止进入临界区。

Pthread 锁- 互斥

- POSIX 库 用于 锁的名称。
  - 用于提供线程之间的互斥。

```
1    pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
2
3    Pthread_mutex_lock（&lock）;//pthread_mutex_lock（）的包装器
4    int count =count+ 1;
5    Pthread_mutex_unlock（&lock）;
```

- POSIX 的 lock 和 unlock 函数会传入一个变量，因为我们可能用不同的锁来保护不同的变量。这样可以增加并发：不同于任何临界区都使用同一个大锁（粗粒度的锁策略），通常大家会用不同的锁保护不同的数据和结构，从而允许更多的线程进入 临界区（细粒度的方案）

关键问题：怎样实现一个锁

如何构建一个高效的锁？高效的锁能够以低成本提供互斥，同时能够实现一些特性，我们下面会讨论。需要什么硬件支持？什么操作系统支持？

评估 锁

- 第一是锁是否能完成它的基本任务，即提供互斥（mutual eXclusion）。最基本的，锁是否有效，能够阻止多个线程 进入临界区？

- 第二是公平性（fairness）。当锁可用时，是否每一个竞争线程有公平的机会抢到锁？用另一个方式来看这个问题是检查更极端的情况：是否有竞争锁的线程会饿死（starve），一直无法获得锁？

- 最后是性能（performance），具体来说，是使用锁之后增加的时间开销。

补充：DEKKER 算法和 PETERSON 算法

20 世纪 60 年代，Dijkstra 向他的朋友们提出了并发问题，他的数学家朋友 Theodorus Jozef Dekker 想出了一个解决方法。不同于我们讨论的需要硬件指令和操作系统支持的方法，Dekker 的算法（Dekker's algorithm）只使用了 load 和 store（早期的硬件上，它们是原子的）。

Peterson 后来改进了 Dekker 的方法[P81]。同样只使用 load 和 store，保证不会有两个线程同时进入临界区。以下是 Peterson 算法（Peterson's algorithm，针对两个线程），读者可以尝试理解这些代码吗？flag 和 turn 变量是用来做什么的？

```
int flag[2];
int turn;

void init() {
    flag[0] = flag[1] = 0;      // 1->thread wants to grab lock
    turn = 0;                   // whose turn? (thread 0 or 1?)
}
void lock() {
    flag[self] = 1;             // self: thread ID of caller
    turn = 1 - self;            // make it other thread's turn
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait
}
void unlock() {
    flag[self] = 0;             // simply undo your intent
}
```

一段时间以来，出于某种原因，大家都热衷于研究不依赖硬件支持的锁机制。后来这些工作都没有太多意义，因为只需要很少的硬件支持，实现锁就会容易很多（实际在多处理器的早期，就有这些硬件支持）。而且上面提到的方法无法运行在现代硬件（应为松散内存一致性模型），导致它们更加没有用处。更多的相关研究也湮没在历史中……

Algorithm 1（单标志算法）

核心思想：设置一个公共整形变量turn,用于指示被允许进入临界区的进程编号。若turn = 0，表示允许P0进入临界区。

- 共享 变量：
  - int turn;
    初始 匝数= 0
  - turn ==ι P_i可以进入其临界 区
- 过程 P_i

```
做 什么
    while（转！ = i) ; //不是自己的标志，就空转
        临界 截面
    turn = j; (对于两个进程来说，j=1−i)
        reminder section
} while （1）;
```

- 有什么问题？

Algorithm 1（单标志算法）

核心思想：设置一个公共整形变量turn,用于指示被允许进入临界区的进程编号。若turn = 0，表示允许P0进入临界区。

- 共享 变量：
  - int turn;
    初始匝数 =0
  - turn == i Πι 可以 进入 临界 区
- 工艺 Pi
- 做 什么
- while（转！ = i)；//不是自己的标志，就空转
- 临界 截面
- turn = j; (对于两个进程来说，j=1-i)
- 提醒 段
- } while （1）；
- 满足了相互 排斥， 但不是进步
- 问题在于两个进程必须轮流执行，否则一个进程进入了一次ctitical section之后就无法再进入了。

Algorithm 2（双标志、先检查算法）

设立一个标志数组**flag[]**：描述进程是否在临界区，初值均为**FALSE**。
先检查，后修改：在进入区检查另一个进程是否在临界区，不在时修改本进程在临界区的标志;在退出区修改本进程在临界区的标志;

- 共享 变量
  - booleanflag[2]；
    初始 标志[0]=标志[1]=假。
  - flag[i] = true P_i准备 进入其临界 区

- // Pi 进程
- while
  （flag[j]）;//①
- return TRUE; //③
- 临界 区
- return [i]；
- 剩余 部分;

// Pj 进程

while(flag[i]); // ② 进入区

flag[j]=TRUE; // ④ 进入区

critical section; // 临界区

flag[j] = FALSE; // 退出区

remainder section; // 剩余区

优点：不用交替进入，可连续使用；

有什么问题？

Algorithm 2（双标志、先检查算法）

- 共享 变量
  - booleanflag[2];
    初始标志[0] = 标志[1]=假。
  - flag[i] = true *Pi*准备 进入其临界 区

设立一个标志数组**flag[]**：描述进程是否在临界区，初值均为**FALSE**。
先检查，后修改：在进入区检查另一个进程是否在临界区，不在时修改本进程在临界区的标志;在退出区修改本进程在临界区的标志;

- // Pi 进程
- while
  （flag[j]）;//①
- return TRUE; //③
- 临界 区
- return [i];
- 剩余 部分;

// Pj 进程
while(flag[i]); // ② 进入区
flag[j] =TRUE; // ④ 进入区
critical section; // 临界区
flag[j] = FALSE; // 退出区
remainder section; // 剩余区

优点：不用交替进入，可连续使用;

缺点：**Pi**和**Pj**可能同时进入临界区。按序列①②③④ 执行时，会同时进入临界区。即 在检查对方**flag**之后和切换自己**flag** 之前有一段时间，结果都检查通过。这里的问 题出在检查和修改操作不能一次进行。

Algorithm 3（双标志、后检查算法）

- 共享 变量
  - boolean flag[2];
    初始 标志[0] =标志[1] =假。
  - flag [i] = τρυε Pi 准备进入 临界 区

```
// Pi进程  flag[i]
=TRUE;
while(flag[j]);
critical section;
flag[i] =FLASE;
剩余 部分;
```

```
// Pj进程
flag[j] =TRUE;  // 进入区
while(flag[i]); // 进入区
critical section; // 临界区
flag [j] =FLASE;   // 退出区
remainder section; // 剩余区
```

有什么问题？

Algorithm 3（双标志、后检查算法）

- 共享 变量
  - boolean flag[2];
    初始 标志[0] =标志[1] =假。
  - flag [i] = τρυε Pi 准备进入 临界　区

```
// Pi进程  flag[i]
=TRUE;
while(flag[j]);
    critical section;
flag[i] =FLASE;
剩余 部分;
```

```
// Pj进程
flag[j] =TRUE; // 进入区
while(flag[i]); // 进入区
    critical section; // 临界区
flag [j] =FLASE;   // 退出区
remainder section; // 剩余区
```

当两个进程几乎同时都想进入临界区时，它们分别将自己的标志值flag设置为TRUE，并且同时检测对方的状态（执行while语句），发现对方也要进入临界区，于是双方互相谦让，结果谁也进不了临界区，从而导致“饥饿”现象。

Algorithm 4（Peterson算法）

- 组合 算法1 和 2的共享变量。
- 工艺 Pi

做 什么

return true;

return j;

while（flag [j]and turn ==j）;

临界 截面

return false;

剩余 部分

} while （1）;

满足 所有三个要求:解决了两个过程的临界截面问题。

基本思想是算法1和算法2的结合。标志flag[I]表示进程Pi想要进入临界区，标志Turn 表示要在进入区等待的进程标识。我的天

Algorithm 4 （Peterson算法）

Flag[1]为false表示p1没有想进临界区

P0
做
什
么
```
int [0] = true;
int[1]=true;
while（flag [1] and turn== 1）;
    临界 截面
return [0];
    剩余 部分
} while （1）;
```

P1
做 什么
```
int [1] = true;
while（flag [0]andturn == 0）;
    临界 截面
return false;
    剩余 部分
} while （1）;
```

进程执行完临界区程序后，修改flag状态使等待进入临界区的进程可在有限时间内进入。

能否满足：

- 互斥（思路：**turn**可能同时等于**0**和**1**吗？
不 可能，同时进入临界区的条件是
flag[0]==false且**flag[1]==false,**但这种情况也不可能）
- 有空让进
- 有限等待

用一个变量来标志锁是否被某些线程占用。第一个线程进入临界区，调用 lock()，检查标志是否为 1（这里不是 1），然后设置标志为 1， 表明线程持有该锁。结束临界区时，线程调用 unlock()，清除标志，表示锁未被持有

```
1   typedef struct   lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 -> lock is available, 1 -> held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)  // TEST the flag
10          ; // spin-wait (do nothing)
11      mutex->flag = 1;          // now SET it!
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```

图 28.1　第一次尝试：简单标志

```
1    typedef struct   lock_t { int flag; } lock_t;
2
3    void init(lock_t *mutex) {
4        // 0 -> lock is available, 1 -> held
5        mutex->flag = 0;
6    }
7
8    void lock(lock_t *mutex) {
9        while (mutex->flag == 1)  // TEST the flag
10           ; // spin-wait (do nothing)
11       mutex->flag = 1;          // now SET it!
12   }
13
14   void unlock(lock_t *mutex) {
15       mutex->flag = 0;
16   }
```

图 28.1　第一次尝试：简单标志

当第一个线程正处于临界区时，如果另一个线程调用 lock()，它会在 while 循环中自旋等待（spin-wait)，直到第一个线程调用 unlock()清空标志。然后等待的线程会退出 while 循环，设置标志，执行临界区代码。

表 28.1　　追踪：没有互斥

Thread 1	Thread 2
call lock() while (flag == 1) interrupt: switch to Thread 2	

	call lock()
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1

flag = 1; // set flag to 1 (too!)	

- 通过适时的（不合时宜的？）中断，我们很容易构造出两 个线程都将标志设置为 1，都能进入临界区的场景。显然没有满足最基本的要求：互斥.

- 性能问题（稍后会有更多讨论）主要是线程在等待已经被持有的锁时，采用了自旋等 待（spin-waiting）的技术，就是不停地检查标志的值。

控制 中断

- 禁用关键 区段的中断
  - 最早提供的互斥解决方案之一，就是在临界区关闭中断。这个解决方案是为单处理器系统开发的。没有中断，线程可以确信它的代码会继续执行下去，不会被其他线程干扰。

```
1          public void run
2                ()   {
         public void run () ;
3   }
4   public void run ()   {
5             int n () ;
6   }
```

- 问题：
  - 第一，一个贪婪的程序可能在它开始时就 调用 lock()，从而独占处理器。更糟的情况是，恶意程序调用 lock()后，一直死循环。后一 种情况，系统无法重新获得控制，只能重启系统。
  - 第二，这种方案不支持多处理器。如果多个线程运行在不同的 CPU 上，每个线程都试 图进入同一个临界区，关闭中断也没有作用。线程可以运行在其他处理器上，因此能够进 入临界区。
  - 第三，关闭中断导致中断丢失，可能会导致严重的系统问题。假如磁盘设备完成了读取请求，但 CPU 错失了这一事实，那么，操作系统如何知道去唤醒等待读取的进程？ 最后一个不太重要的原因就是效率低。

在某些情况下操作系统本身会采用屏蔽中断的方式，保证访问自己数据结构的原子性，或至少避免某 些复杂的中断处理情况。因为在操作系统内部不存在信任问题，它总是信任自己可以执行特权操作

用Test-And-Set构造 工作自旋 锁

- 关闭中断的方法无法工作在多处理器上，所以系统设计者开始让硬件支持锁. 最早的多处理器系统，像 20 世纪 60 年代早期的 Burroughs B5000[M82]，已经有这些支持。今天所有的系统都支持，甚至包括单 CPU的系统。

- 最简单的硬件支持是测试并设置指令（test-and-set instruction），也叫作原子交换（atomic eXchange）

- 在 SPARC 上，这个指令叫 ldstub（load/store unsigned byte，加载/保存无符号字节）;在 x86 上，是 xchg（atomic eXchange，原子交换）指令。但它们基本上在不同的平台上做同样的事，通常称为测试并设置指令（test-and-set）

```
1  int TestAndSet(int *old_ptr, int new) {
2      int old = *old_ptr; // fetch old value at old_ptr
3      *old_ptr = new;     // store 'new' into old_ptr
4      return old;         // return the old value
5  }
```

- 测试并设置指令做了下述事情。它返回 old_ptr 指向的旧值，同时更新为 new 的新值。 当然，关键是这些代码是原子地（atomically）执行。因为既可以测试旧值，又可以设置新 值，所以我们把这条指令叫作“测试并设置”。

```
1    typedef struct   lock_t {
2        int flag;
3    } lock_t;
4
5    void init(lock_t *lock) {
6        // 0 indicates that lock is available, 1 that it is held
7        lock->flag = 0;
8    }
9
10   void lock(lock_t *lock) {
11       while (TestAndSet(&lock->flag, 1) == 1)
12           ; // spin-wait (do nothing)
13   }
14
15   void unlock(lock_t *lock) {
16       lock->flag = 0;
17   }
```

图 28.2　利用测试并设置的简单自旋锁

讨论两种情况：
第一，首先假设一个线程在运行，调用 lock()，没有其他线程持有锁，所以 flag 是 0。当调用 TestAndSet(flag, 1)方法，返回 0，线程会跳出 while 循环，获取锁。同时也会原子的设置 flag 为 1，标志锁已经被持有。当线程离开临界区，调 用 unlock()将 flag 清理为 0。
第二，第二种场景是，当某一个线程已经持有锁（即 flag 为 1）。本线程调用 lock()，然后调用 TestAndSet(flag, 1)，这一次返回 1。只要另一个线程一直持有锁，TestAndSet()会重复返回 1， 本线程会一直自旋。当 flag 终于被改为 0，本线程会调用 TestAndSet()，返回 0 并且原子地 设置为 1，从而获得锁，进入临界区。

强调两个线程同时执行TestAndSet时，两个必须按顺序执行，获取锁的线程必然是设置了flag为1，返回0，没有获取锁的线程也设置了flag为1，但返回1;它们不可能同时返回0又设置了flag为1。
将测试（test 旧的锁值）和设置（set 新的值）合并为一个原子操作之后，我们保证了 只有一个线程能获取锁。这就实现了一个有效的互斥原语

- 它是最简单的锁类型，只需使用CPU周期旋转，直到锁变得可用。

- 要在单个处理器上正确工作，它**需要抢占式调度程序**。(如果不能被抢先，线程会一直自旋下去)

评估 自旋 锁

- **正确性：** 旋转锁一次只允许一个线程 进入临界区 。

- **公平**。简单的自旋锁（到目前为止已经讨论过）是不 公平的，可能会导致饥饿。

- **性能：** 对于旋转锁，在**单CPU**的情况下， 性能开销 可能 会 非常 痛苦。 在多个CPU上，旋转锁工作得相当好。

比较和交换

```
1  int CompareAndSwap(int *ptr, int expected, int new) {
2      int original = *ptr;
3      if (original == expected)
4          *ptr = new;
5      return original;
6  }
```

Figure 28.4: **Compare-and-swap**

- 另一个 硬件原语是 比较和交换
  指令 （在SPARC上）或比较和交换（在x86上）。

- 比较并交换的基本思路是检测 ptr 指向的值是否和 expected 相等;如果是，更新 ptr 所 指的值为新值。否则，什么也不做。不论哪种情况，都返回该内存地址的实际值，让调用者能够知道执行是否成功。

- 在 任何一种情况下， 都要**返回该内存位置的实际值**， 从而允许调用compare-and-swap的代码知道它是否成功 。

```
1  void lock(lock_t *lock) {
2      while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3          ; // spin
4  }
```

检查标志是否为0，如果是，原子地交换为1，从而获得锁。

compare-and-swap 是一个比test-and-set更强大的指令， 可以用来 实现 无锁 同步

```
1   char CompareAndSwap(int *ptr, int old, int new) {
2       unsigned char ret;
3
4       // Note that sete sets a 'byte' not the word
5       __asm__ __volatile__ (
6             "  lock\n"
7             "  cmpxchgl %2,%1\n"
8             "  sete %0\n"
9             : "=q" (ret), "=m" (*ptr)
10            : "r" (new), "m" (*ptr), "a" (old)
11            : "memory");
12      return ret;
13  }
```

负载关联 和 存储条件

一些平台提供了实现临界区的一对指令。例如 MIPS 架构[H93]中，链接的加载 （load-linked，LL）和条件式存储（store-conditional，SC）可以用来配合使用，实现其他的并发结构。

```
1  int LoadLinked(int *ptr) {
2      return *ptr;
3  }
4
5  int StoreConditional(int *ptr, int value) {
6      if (no update to *ptr since LoadLinked to this address) {
7          *ptr = value;
8          return 1; // success!
9      } else {
10         return 0; // failed to update
11     }
12 }
```

Figure 28.5: Load-linked And Store-conditional

LL/SC 指令的独特之处在于，它们不是一个简单的内存读取/写入的函数，当使用 LL 指令从内存中读取一个字之后，比如 LL d, off(b)，处理器会记住 LL 指令的这次操作（会在 CPU 的寄存器中设置一个不可见的 bit 位），同时 LL 指令读取的地址 off(b) 也会保存在处理器的寄存器中。

接下来的 SC 指令，比如 SC t, off(b)，会检查上次 LL 指令执行后的 RMW（Read-Modify-Write）操作是否是原子操作（即不存在其它对这个地址的操作），如果是原子操作，则 t 的值将会被更新至内存中，同时 t 的值也会变为1，表示操作成功;反之，如果 RMW的操作不是原子操作（即存在其它对这个地址的访问冲突），则 t 的值不会被更新至内存中，且 t 的值也会变为0，表示操作失败。

SC 指令执行失败的原因有两种：

- 在 LL/SC 操作过程中，发生了一个异常（或中断），这些异常（或中断）可能会打乱 RMW 操作的原子性。
- 在多核处理器中，一个核在进行 RMW 操作时，别的核试图对同样的地址也进行操作，这会导致 SC 指令执行的失败。

最开始，lock->flag设置为0，表示没有线程获得锁，LL指令返回0，跳出while循环

```
1  void lock(lock_t *lock) {
2      while (1) {
3          while (LoadLinked(&lock->flag) == 1)
4              ; // spin until it's zero
5          if (StoreConditional(&lock->flag, 1) == 1)
6              return; // if set-it-to-1 was a success: all done
7                      // otherwise: try it all over again
8      }
9  }
10
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }
```

如果直到SC执行，lock->flag的值没有被修改过，则SC指令把lock->flag的值修改为1，并返回1. 返回值1==1，执行return

如果SC执行时，发现lock->flag的值修改过，则SC返回0，所以 不执行return，则需要重新进入while（1），再次尝试获取锁

Figure 28.6: **Using LL/SC To Build A Lock**

获取并添加

```
1  int FetchAndAdd(int *ptr) {
2      int old = *ptr;
3      *ptr = old + 1;
4      return old;
5  }
```

- 获取并增加（fetch-and-add）指令，它能原子地返回特定地址的旧值，并且让该值自增一。

```
1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn   = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15   }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19 }
```

返回lock->ticket的值，并对lock->ticket的值加1

获取进入临界区的turn

Figure 28.7: Ticket Locks

使用了ticket和turn变量作为组合来构造锁。当一个线程希望获得锁时，它先对ticket值做一次原子地fetch-and-add操作;此时这个值就作为这个线程的”turn”（myturn）。全局共享变量lock->turn用来决定轮到了哪个线程; 当对于某个线程myturn等于turn时，那就轮到了这个线程进入临界区。unlock简单地将turn值加1，由此下一个等 待线程（如果存在的话）就可以进入临界区了。注意这个方法相对于前面的几种方式的一个重要不同：它保证 了所有线程的执行。一旦某个线程得到了他自己的ticket值，在将来的某一时刻肯定会被调度执行（一旦前面的 那些线程执行完临界区并释放锁）。在先前的方案中，并没有这一保证;比如，某个自旋在test-and-set的线程 可能会一直自旋下去，即使其他的线程获得、释放锁。

```
1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn   = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15  }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19 }
```

返回lock->ticket的值，并对lock->ticket的值加1

Figure 28.7: **Ticket Locks**

第一个线程： turn= 0; myturn=0; ticket=1;
第二个线程： 当第一个线程未调用unlock时， turn=0; myturn=1; ticket=2;
第三个线程： 当第一个线程未调用unlock时， turn=0; myturn=2; ticket=3;
当第一个线程unlock后， turn=1， 第二个线程获得锁， 第二个线程unlock后， turn=2， 第三个线程获得锁

最开始， ticket=0， turn=0。
第一个线程，执行faa， myturn=0， lock->ticket=1,while中turn=0且myturn=0，所以跳出while，假设，第一个线程刚执行完faa，第二个线程来了，执行faa，则myturn=1，ticket变为2，但是第二个线程执行while时lock->trun 还是0，mytrun为1，所以自旋等待，等到第一个线程从临界区出来，使用unlock把lock->turn变为1，才能从while中出来。

太多的旋转：现在怎么办？

关键问题：怎样避免自旋

如何让锁不会不必要地自旋，浪费 CPU 时间？

一个线程会 一直自旋检查一个不会改变的值，浪费掉整个时间片!如果有 N 个线程去竞争一个锁，情 况会更糟糕。同样的场景下，会浪费 N−1 个时间片

- 单靠硬件 支持**无法**解决 问题。 我们也需要**操作系统的支持** ！

一个 简单的方法：JustYield，Baby

- 假设 有一个操作系统原语**yield ()**，当一个线程 想要 放弃 CPU 并 让另一个线程运行时，它可以调用该原语。

- Yield 将 调用方 从 运行 状态移动到 就绪状态，从而将另一个线程提升到运行状态。

- 屈服过程基本上是自行分解 的。

```
1  void init() {
2      flag = 0;
3  }
4
5  void lock() {
6      while (TestAndSet(&flag, 1) == 1)
7          yield(); // give up the CPU
8  }
9
10 void unlock() {
11     flag = 0;
12 }
```

Figure 28.8: Lock With Test-and-set And Yield

两个线程的例子中，基于 yield 的方法十分有效。一个线程 调用 lock()，发现锁被占用时，让出 CPU，另外一个线程运行，完成临界区。在这个简单的 例子中，让出方法工作得非常好。

许多线程（例如 100 个）反复竞争一把锁的情况。在这种情况下，一个线程持有锁，在释放锁之前被抢占，其他 99 个线程分别调用 lock()，发现锁被抢占，然后让 出 CPU。假定采用某种轮转调度程序，这 99 个线程会一直处于运行—让出这种模式，直到 持有锁的线程再次运行。虽然比原来的浪费 99 个时间片的自旋方案要好，但这种方法仍然 成本很高，上下文切换的成本是实实在在的，因此浪费很大。

使用 队列： 睡眠而不是 旋转

- 前面一些方法的真正问题是存在太多的偶然性。调度程序决定如何调度。如果调度不合理，线程或者一直自旋（第一种方法），或者立刻让出 CPU（第二种方法）。无论哪种方 法，都可能造成浪费，也能防止饿死。

- 必须显式地施加某种控制，决定锁释放时，谁能抢到锁。

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag  = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock
33                                     // (for next thread!)
34     m->guard = 0;
35 }
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

Solaris 提供的支持，它提供了两个调用：park()能够让调用线程休眠，unpark(threadID)则会唤醒 threadID 标识的线程。可以用这两个调用来实现锁，让调用 者在获取不到锁时睡眠，在锁可用时被唤醒。

该方法并没有完全避免自旋等待。线程在获取锁或者释放锁时可能被中断，从而导致其他线程自旋等待。但是，这个自旋等待时间是很有限的（不是用户定义的临界区，只是在 lock 和 unlock 代码中的几个指令，m->guard会很快被设置为0，因此自旋等待时间有限）

获得锁的条件是guard==0 且flag==0

多个线程竞争lock时，第一个抢得guard并进入if分支获得flag（ guard==0且flag==1 ），其它的进入else分支调用park()。

18行是抢得锁的线程释放guard，让其它线程有机会进入等待队列，21行类似

如果21行放在22行后面会出现什么情况？后面调用lock的线程会一直spinning下去

park调用之前，线程被切换，另一个拥有锁的线程执行（随后释放了锁），这时被切换出去的线程随后park时，可能会永久睡眠（因为没有线程从临界区出来执行unlock从而执行unpark）。

unlock中TestAndSet的必要性，获取guard，这样后面可以设置flag值，并操作队列，类似于lock中的实现

通过队列来控制谁会获得锁，避免饿死

线程被唤醒时(调用unpark)，就像是从 park()调用返回，直接把锁从释放的线程传递给下一个获得锁的线程，期间 flag 不必设置为 0，因为flag==1表示有线程在临界区

park调用之前，线程被切换，另一个拥有锁的线程执行（随后释放了锁），这时被切换出去的线程随后park时，可能会永久睡眠（因为没有线程从临界区出来后执行unlock中的unpark）。

Solaris 通过增加了第三个系统调用 separk()来解决这一问题。通过 setpark()，一个线程 表明自己马上要 park。如果刚好另一个线程被调度，并且调用了 unpark，那么后续的 park 调用就会直接返回，而不是一直睡眠。

```
1              return (m-〉 q, getId () ) ;
2              setpark () ;// newcode
3              =   0;
4              return () ;
```

两相 锁

- 两阶段锁意识到自旋可能很有用，尤其是在很快就要释放锁的场景。
- 因此，两阶段锁的第一阶段会先自旋 一段时间，希望它可以获取锁。 但是，如果第一个自旋阶段没有获得锁
- 第二阶段调用者会睡眠，直到锁可用。
- 两阶段锁是又一个杂合（hybrid）方案的例子，即结合两种好想法得到更好的想法。

结束