1.5 自旋鎖 spinlock 的實現

自旋鎖，顧名思義：自己在原地打轉，等待資源可用，一旦可用就上鎖霸佔它。

問題來了，假設別人已經上鎖了，你原地打轉會占住 CPU 資源了，別的程式怎麼運行？它沒有 CPU 怎麼解鎖？

這個問題，有 2 個答案：

① 原地打轉的是 CPU x，以後 CPU y 會解鎖：這涉及多個 CPU，適用於 SMP 系統；

② 對於單 CPU 系統，自旋鎖的“自旋”功能就去掉了：只剩下禁止搶佔、禁止中斷

我先禁止別的執行緒來打斷我(preempt_disable)，我慢慢享用臨界資源，用完再使能系統搶佔(preempt_enable)，這樣別人就可以來搶資源了。

注意：SMP 就是 Symmetric Multi-Processors，對稱多處理器；UP 即 Uni-Processor，系統只有一個單核 CPU。

要理解 spinlock，要通過 2 個情景來分析：

① 一開始，怎麼爭搶資源？不能 2 個程式都搶到。

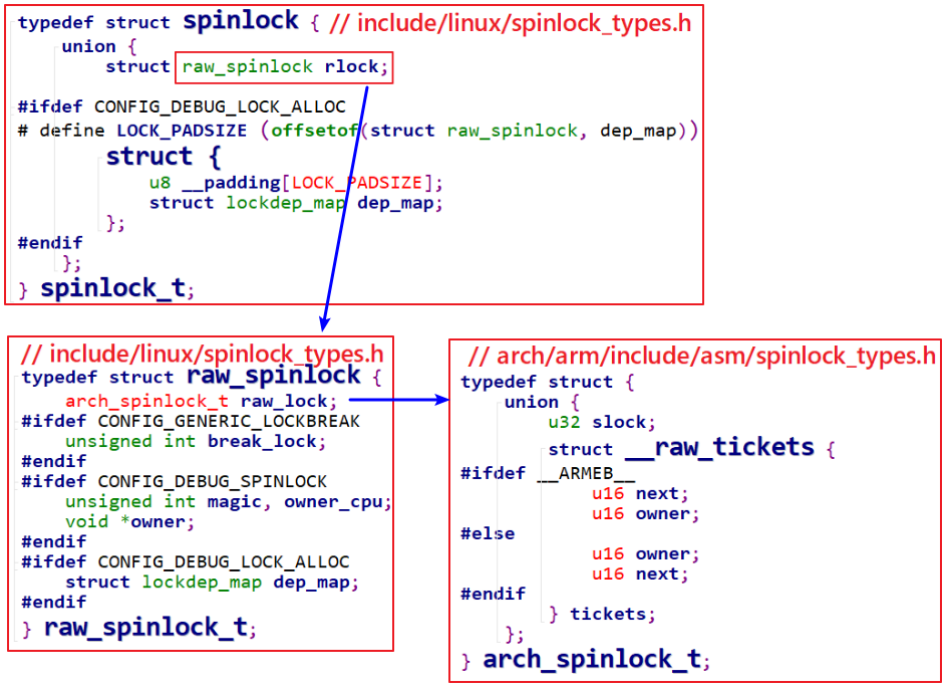
這挺好解決，使用原子變數就可以實現。

② 某個程式已經獲得資源，怎麼防止別人來同時使用這個資源。

這是使用 spinlock 時要注意的地方，對應會有不同的衍生函數(_bh/_irq/_irqsave/_restore)。

1.5.1 自旋鎖的內核結構體

spinlock 對應的結構體如下定義，不同的架構可能有不同的實現：



上述__raw_tickets 結構體中有 owner、next 兩個成員，這是在 SMP 系統中實現 spinlock 的關鍵。

1.5.2 spinlock 在 UP 系統中的實現

對於“自旋鎖”，它的本意是：如果還沒獲得鎖，我就原地打轉等待。等待誰釋放鎖？

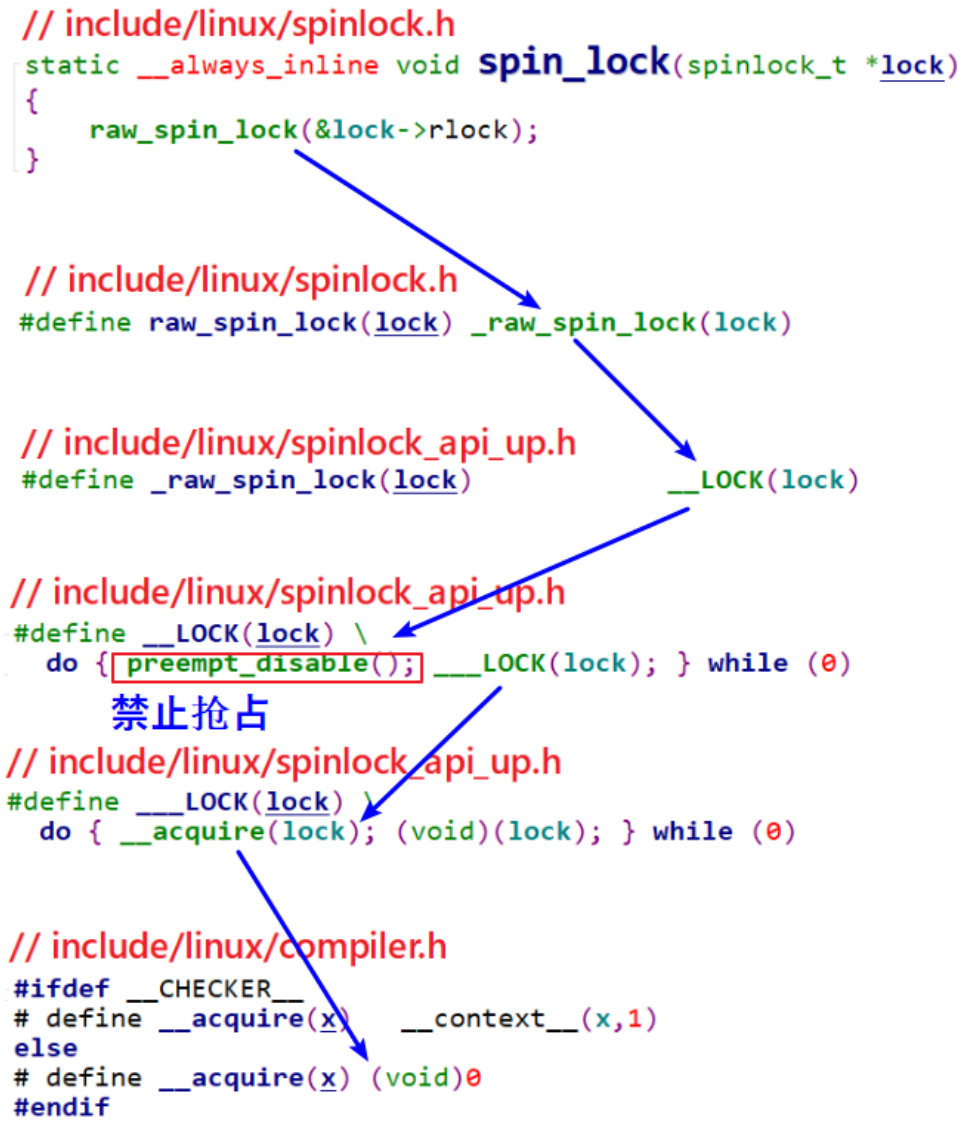
① 其他 CPU

② 其他進程/執行緒

對於單 CPU 系統，沒有“其他 CPU”；如果內核不支持 preempt，當前在內核態執行的執行緒也不可能被其他執行緒搶佔，也就“沒有其他進程/執行緒”。所以，對於不支援 preempt 的單 CPU 系統，spin_lock 是空函數，不需要做其他事情。

如果單 CPU 系統的內核支援 preempt，即當前執行緒正在執行內核態函數時，它是有可能被別的執行緒搶佔的。這時 spin_lock 的實現就是調用“preempt_disable()”：你想搶我，我乾脆禁止你運行。

在 UP 系統中，spin_lock 函式定義如下：



從以上代碼可知，在 UP 系統中 spin_lock()就退化為 preempt_disable()，如果用的內核不支持 preempt，那麼 spin_lock()什麼事都不用做。

對於 spin_lock_irq()，在 UP 系統中就退化為 local_irq_disable()和 preempt_disable()，如下圖所示：

```
// include/linux/spinlock_api_up.h
#define __LOCK(lock) \
  do { preempt_disable(); ___LOCK(lock); } while (0)

#define __LOCK_IRQ(lock) \
  do { local_irq_disable(); __LOCK(lock); } while (0)
```

假設程式 A 要訪問臨界資源，可能會有中斷也來訪問臨界資源，可能會有程式 B 也來訪問臨界資源，那麼使用 spin_lock_irq()來保護臨界資源：先禁止中斷防止中斷來搶，再禁止 preempt 防止其他進程來搶。

對於 spin_lock_bh()，在 UP 系統中就退化為禁止軟體插斷和 preempt_disable()，如下圖所示：

```
// include/linux/spinlock_api_up.h
#define __LOCK_BH(lock) \
  do { __local_bh_disable_ip(_THIS_IP_, SOFTIRQ_LOCK_OFFSET); ___LOCK(lock); } while (0)

// include/linux/preempt.h
/*
 * The preempt_count offset needed for things like:
 *
 *  spin_lock_bh()
 *
 * Which need to disable both preemption (CONFIG_PREEMPT_COUNT) and
 * softirqs, such that unlock sequences of:
 *
 *  spin_unlock();
 *  local_bh_enable();
 *
 * Work as expected.
 */
#define SOFTIRQ_LOCK_OFFSET (SOFTIRQ_DISABLE_OFFSET + PREEMPT_LOCK_OFFSET)
```

取个巧，
只看这个宏就知道它是：
既禁止preempt，又禁止softirq

對於 spin_lock_irqsave，它跟 spin_lock_irq 類似，只不過它是先保存中斷狀態再禁止中斷，如下：

```
// include/linux/spinlock_api_up.h
#define __LOCK(lock) \
  do { preempt_disable(); ___LOCK(lock); } while (0)

#define __LOCK_IRQSAVE(lock, flags) \
    do { local_irq_save(flags); __LOCK(lock); } while (0)
```

對應的 spin_unlock 函數就不再講解。

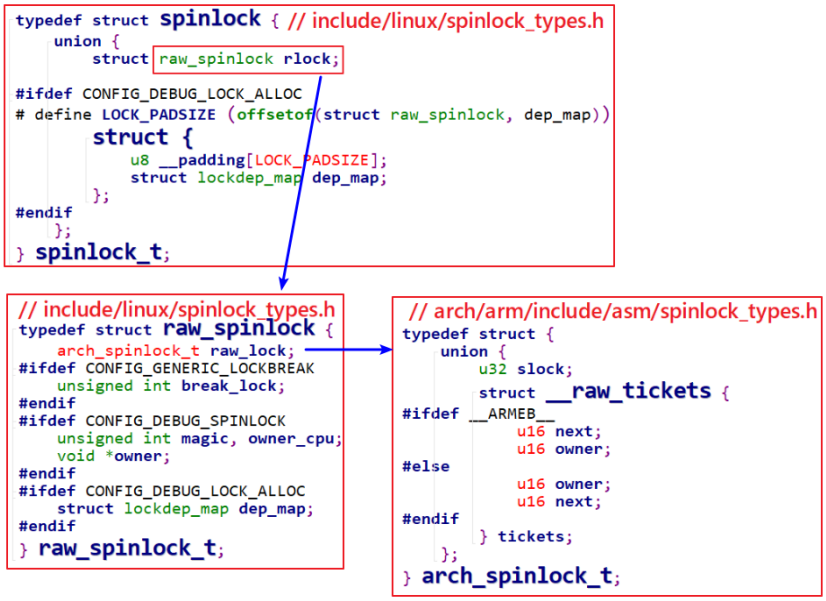
1.5.3 spinlock 在 SMP 系統中的實現

要讓多 CPU 中只能有一個獲得臨界資源，使用原子變數就可以實現。但是還要保證公平，先到先得。比如有 CPU0、CPU1、CPU2 都調用 spin_lock 想獲得臨界資源，誰先申請誰先獲得。

要想理解 SMP 系統中 spinlock 的實現，得舉一個例子。感謝這篇文章：
Linux 內核同步機制之（四）：spin lock
http://www.wowotech.net/kernel_synchronization/spinlock.html
wowotech 真是一個神奇的網站，裡面 Linux 文章的作者統一標為“linuxer”，牛！

我借用這篇文章的例子講解，餐廳裡只有一個座位，去吃飯的人都得先取號、等叫號。注意，有 2 個動作：顧客從取號機取號，電子叫號牌叫號。
① 一開始取號機待取號碼為 0
② 顧客 A 從取號機得到號碼 0，電子叫號牌顯示 0，顧客 A 上座；
取號機顯示下一個待取號碼為 1。
③ 顧客 B 從取號機得到號碼 1，電子叫號牌還顯示為 0，顧客 B 等待；
取號機顯示下一個待取號碼為 2。
④ 顧客 C 從取號機得到號碼 2，電子叫號牌還顯示為 0，顧客 C 等待；
取號機顯示下一個待取號碼為 3。
⑤ 顧客 A 吃完離座，電子叫號牌顯示為 1，顧客 B 的號碼等於 1，他上座；
⑥ 顧客 B 吃完離座，電子叫號牌顯示為 2，顧客 C 的號碼等於 2，他上座；
在這個例子中有 2 個號碼：取號機顯示的“下一個號碼”，顧客取號後它會自動加 1；電子叫號牌顯示“當前號碼”，顧客離座後它會自動加 1。某個客戶手上拿到的號碼等於電子叫號牌的號碼時，該客戶上座。
在這個過程中，即使顧客 B、C 同時到店，只要保證他們從取號機上得到的號碼不同，他們就不會打架。
所以，關鍵點在於：取號機的號碼發放，必須互斥，保證客戶的號碼互不相同。而電子叫號牌上號碼的變動不需要保護，只有顧客離開後它才會變化，沒人爭搶它。

在 ARMv6 及以上的 ARM 架構中，支援 SMP 系統。它的 spinlock 結構體定義如下：



owner 就相當於電子叫號牌，現在誰在吃飯。next 就當于於取號機，下一個號碼是什麼。每一個 CPU 從取號機上取到的號碼保存在 spin_lock 函數中的區域變數裡。

spin_lock 函式呼叫關係如下，核心是 arch_spin_lock：

```
// include/linux/spinlock.h
#define raw_spin_lock(lock) _raw_spin_lock(lock)
static __always_inline void spin_lock(spinlock_t *lock)
{
    raw_spin_lock(&lock->rlock);
}

// kernel/locking/spinlock.c
void __lockfunc _raw_spin_lock(raw_spinlock_t *lock)
{
    __raw_spin_lock(lock);
}

// include/linux/spinlock_api_smp.h
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
    LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}

// include/linux/spinlock.h
static inline void do_raw_spin_lock(raw_spinlock_t *lock) __acquires(lock)
{
    __acquire(lock);
    arch_spin_lock(&lock->raw_lock);
}
```

arch_spin_lock 代碼如下：

```
// arch/arm/include/asm/spinlock.h
static inline void arch_spin_lock(arch_spinlock_t *lock)
{
    unsigned long tmp;
    u32 newval;
    arch_spinlock_t lockval;

    prefetchw(&lock->slock);
    __asm__ __volatile__(
"1: ldrex   %0, [%3]\n"
"   add %1, %0, %4\n"
"   strex   %2, %1, [%3]\n"
"   teq %2, #0\n"
"   bne 1b"
    : "=&r" (lockval), "=&r" (newval), "=&r" (tmp)
    : "r" (&lock->slock), "I" (1 << TICKET_SHIFT)
    : "cc");

    while (lockval.tickets.next != lockval.tickets.owner) {
        wfe();
        lockval.tickets.owner = ACCESS_ONCE(lock->tickets.owner);
    }
    smp_mb();
} « end arch_spin_lock »
```

1.取号, lockval = lock->slock, next和owner都读出来了
2.取号机的号码要加1, newval = lockval + (1<<TICKET_SHIFT), 就是next++
3.新号码写回取号机, lock->slock=newval, 不一定能写成功
4.如果123过程中被别人先取号了，我的"写回"操作失败，那重新取号吧
%0 %1 %2 %3 %4
5.我手上的号码不等于电子叫号牌的话
6.就原地休息一会, cpu低功耗运行
7.时不时看一眼电子叫号牌
8.能运行到这里，表示我手上的号码等于叫号牌了，座位肯定是我的了

圖中的注釋把原理講得非常清楚了,即使不同的個體去同時取號，也可以保證取到的號碼各不相同。

假設第 1 個程式取到了號碼，它訪問了臨界資源後，調用 spin_unlock，代碼如下：

```
// arch/arm/include/asm/spinlock.h
static inline void arch_spin_unlock(arch_spinlock_t *lock)
{
    smp_mb();
    lock->tickets.owner++;
    dsb_sev();
}
```

不可能有多个程序同时修改owner
所以不用加什么互斥措施

假如有其他程式正在 spin_lock 函數中迴圈等待，它就會立刻判斷自己手上的 next 是否等於 lock->tickets.owner，如果相等就表示輪到它獲得了鎖。

深入分析_linux_spinlock_實現機制
https://blog.csdn.net/electrombile/article/details/51289813

深入分析 Linux 自旋鎖
http://blog.chinaunix.net/uid-20543672-id-3252604.html

Linux 內核同步機制之（四）：spin lock
http://www.wowotech.net/kernel_synchronization/spinlock.html