1.4 Linux 鎖的介紹與使用

本節參考：
https://www.kernel.org/doc/html/latest/locking/index.html
https://mirrors.edge.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/

1.4.1 鎖的類型

Linux 內核提供了很多類型的鎖，它們可以分為兩類：
① 自旋鎖(spinning lock)；
② 睡眠鎖(sleeping lock)。

1.4.1.1 自旋鎖

簡單地說就是無法獲得鎖時，不會休眠，會一直迴圈等待。有這些自旋鎖：

自旋鎖	描述
raw_spinlock_t	原始自旋鎖(後面講解)
bit spinlocks	位自旋鎖(似乎沒什麼意義)

自旋鎖的加鎖、解鎖函數是：spin_lock、spin_unlock，還可以加上各種尾碼，這表示在加鎖或解鎖的同時，還會做額外的事情：

尾碼	描述
_bh()	加鎖時禁止下半部(軟中斷)，解鎖時使能下半部(軟中斷)
_irq()	加鎖時禁止中斷，解鎖時使能中斷
_irqsave/restore()	加鎖時禁止並中斷並記錄狀態，解鎖時恢復中斷為所記錄的狀態

1.4.1.2 睡眠鎖

簡單地說就是無法獲得鎖時，當前執行緒就會休眠。有這些休眠鎖：

休眠鎖	描述
mutex	mutual exclusion，彼此排斥，即互斥鎖(後面講解)
rt_mutex	
semaphore	信號量、旗語(後面講解)
rw_semaphore	讀寫信號量，讀寫互斥，但是可以多人同時讀
ww_mutex	
percpu_rw_semaphore	對 rw_semaphore 的改進，性能更優

1.4.2 鎖的內核函數

1.4.2.1 自旋鎖

spinlock 函數在內核檔 include\linux\spinlock.h 中聲明，如下表：

函數名	作用
spin_lock_init(_lock)	初始化自旋鎖為 unlock 狀態
void spin_lock(spinlock_t *lock)	獲取自旋鎖(加鎖)，返回後肯定獲得了鎖
int spin_trylock(spinlock_t *lock)	嘗試獲得自旋鎖，成功獲得鎖則返回 1，否則返回 0
void spin_unlock(spinlock_t *lock)	釋放自旋鎖，或稱解鎖
int spin_is_locked(spinlock_t *lock)	返回自旋鎖的狀態，已加鎖返回 1，否則返回 0

自旋鎖的加鎖、解鎖函數是：spin_lock、spin_unlock，還可以加上各種尾碼，這表示在加鎖或解鎖的同時，還會做額外的事情：

尾碼	描述
_bh()	加鎖時禁止下半部(軟中斷)，解鎖時使能下半部(軟中斷)
_irq()	加鎖時禁止中斷，解鎖時使能中斷
_irqsave/restore()	加鎖時禁止並中斷並記錄狀態，解鎖時恢復中斷為所記錄的狀態

1.4.2.2 信號量 semaphore

semaphore 函數在內核檔 include\linux\semaphore.h 中聲明，如下表：

函數名	作用
DEFINE_SEMAPHORE(name)	定義一個 struct semaphore name 結構體， count 值設置為 1
void sema_init(struct semaphore *sem, int val)	初始化 semaphore
void down(struct semaphore *sem)	獲得信號量，如果暫時無法獲得就會休眠 返回之後就表示肯定獲得了信號量 在休眠過程中無法被喚醒， 即使有信號發給這個進程也不處理
int down_interruptible(struct semaphore *sem)	獲得信號量，如果暫時無法獲得就會休眠， 休眠過程有可能收到信號而被喚醒， 要判斷返回值： 0：獲得了信號量 -EINTR：被信號打斷
int down_killable(struct semaphore *sem)	跟 down_interruptible 類似， down_interruptible 可以被任意信號喚醒， 但 down_killable 只能被“fatal signal”喚醒， 返回值： 0：獲得了信號量 -EINTR：被信號打斷
int down_trylock(struct semaphore *sem)	嘗試獲得信號量，不會休眠， 返回值：

	0：獲得了信號量 1：沒能獲得信號量
int down_timeout(struct semaphore *sem, long jiffies)	獲得信號量，如果不成功，休眠一段時間 返回值： 0：獲得了信號量 -ETIME：這段時間內沒能獲取信號量，超時返回 down_timeout 休眠過程中，它不會被信號喚醒
void up(struct semaphore *sem)	釋放信號量，喚醒其他等待信號量的進程

1.4.2.3 互斥量 mutex

mutex 函數在內核檔 include\linux\mutex.h 中聲明，如下表：

函數名	作用
mutex_init(mutex)	初始化一個 struct mutex 指標
DEFINE_MUTEX(mutexname)	初始化 struct mutex mutexname
int mutex_is_locked(struct mutex *lock)	判斷 mutex 的狀態 1：被鎖了(locked) 0：沒有被鎖
void mutex_lock(struct mutex *lock)	獲得 mutex，如果暫時無法獲得，休眠 返回之時必定是已經獲得了 mutex
int mutex_lock_interruptible(struct mutex *lock)	獲得 mutex，如果暫時無法獲得，休眠; 休眠過程中可以被信號喚醒， 返回值： 0：成功獲得了 mutex -EINTR：被信號喚醒了
int mutex_lock_killable(struct mutex *lock)	跟 mutex_lock_interruptible 類似， mutex_lock_interruptible 可以被任意信號喚醒， 但 mutex_lock_killable 只能被“fatal signal” 喚醒， 返回值： 0：獲得了 mutex -EINTR：被信號打斷
int mutex_trylock(struct mutex *lock)	嘗試獲取 mutex，如果無法獲得，不會休眠， 返回值： 1：獲得了 mutex， 0：沒有獲得 注意，這個返回值含義跟一般的 mutex 函數相反，
void mutex_unlock(struct mutex *lock)	釋放 mutex，會喚醒其他等待同一個 mutex 的執行緒
int atomic_dec_and_mutex_lock(atomic_t *cnt, struct mutex *lock)	讓原子變數的值減 1， 如果減 1 後等於 0，則獲取 mutex， 返回值：

	1：原子變數等於 0 並且獲得了 mutex 0：原子變數減 1 後並不等於 0，沒有獲得 mutex

1.4.2.4 semaphore 和 mutex 的區別

semaphore 中可以指定 count 為任意值，比如有 10 個廁所，所以 10 個人都可以使用廁所。

而 mutex 的值只能設置為 1 或 0，只有一個廁所。

是不是把 semaphore 的值設置為 1 後，它就跟 mutex 一樣了呢？不是的。

看一下 mutex 的結構體定義，如下：

```
// include/linux/mutex.h
struct mutex {
    /* 1: unlocked, 0: locked, negative: locked, possible waiters */
    atomic_t        count;
    spinlock_t      wait_lock;
    struct list_head    wait_list;
#if defined(CONFIG_DEBUG_MUTEXES) || defined(CONFIG_MUTEX_SPIN_ON_OWNER)
    struct task_struct  *owner;
#endif
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
    struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
#ifdef CONFIG_DEBUG_MUTEXES
    void            *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map  dep_map;
#endif
};
```

指向某个线程

它裡面有一項成員“struct task_struct *owner”，指向某個進程。一個 mutex 只能在進程上下文中使用：誰給 mutex 加鎖，就只能由誰來解鎖。

而 semaphore 並沒有這些限制，它可以用來解決“讀者-寫者”問題：程式 A 在等待資料——想獲得鎖，程式 B 產生資料後釋放鎖，這會喚醒 A 來讀取資料。semaphore 的鎖定與釋放，並不限定為同一個進程。

主要區別列表如下：

	semaphore	mutex
幾把鎖	任意，可設置	1
誰能解鎖	別的程式、中斷等都可以	誰加鎖，就得由誰解鎖
多次解鎖	可以	不可以，因為只有 1 把鎖
迴圈加鎖	可以	不可以，因為只有 1 把鎖
任務在持有鎖的期間可否退出	可以	不建議，容易導致鎖死
硬體中斷、軟體插斷上下文中使用	可以	不可以

1.4.3 何時用何種鎖

本節參考：https://wenku.baidu.com/view/26adb3f5f61fb7360b4c656e.html
英文原文：https://mirrors.edge.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/
你可能看不懂下面這個表格，請學習完後面的章節再回過頭來看這個表格。

	IRQ Handler A	IRQ Handler B	Softirq A	Softirq B	Tasklet A	Tasklet B	Timer A	Timer B	User Context A	User Context B
IRQ Handler A	None									
IRQ Handler B	spin_lock_irqsave()	None								
Softirq A	spin_lock_irq()	spin_lock_irq()	spin_lock()							
Softirq B	spin_lock_irq()	spin_lock_irq()	spin_lock()	spin_lock()						
Tasklet A	spin_lock_irq()	spin_lock_irq()	spin_lock()	spin_lock()	None					
Tasklet B	spin_lock_irq()	spin_lock_irq()	spin_lock()	spin_lock()	spin_lock()	None				
Timer A	spin_lock_irq()	spin_lock_irq()	spin_lock()	spin_lock()	spin_lock()	spin_lock()	None			
Timer B	spin_lock_irq()	spin_lock_irq()	spin_lock()	spin_lock()	spin_lock()	spin_lock()	spin_lock()	None		
User Context A	spin_lock_irq()	spin_lock_irq()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	None	
User Context B	spin_lock_irq()	spin_lock_irq()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	down_interruptible	None

舉例簡單介紹一下，上表中第一行“IRQ Handler A”和第一列“Softirq A”的交叉點是“spin_lock_irq()”，意思就是說如果“IRQ Handler A”和“Softirq A”要競爭臨界資源，那麼需要使用“spin_lock_irq()”函數。為什麼不能用 spin_lock 而要用 spin_lock_irq？也就是為什麼要把中斷給關掉？假設在 Softirq A 中獲得了臨界資源，這時發生了 IRQ A 中斷，IRQ Handler A 去嘗試獲得自旋鎖，這就會導致鎖死：所以需要關中斷。

1.4.4 內核搶佔(preempt)等額外的概念

早期的的 Linux 內核是“不可搶佔”的，假設有 A、B 兩個程式在運行，當前是程式 A 在運行，什麼時候輪到程式 B 運行呢？

① 程式 A 主動放棄 CPU：

比如它調用某個系統調用、調用某個驅動，進入內核態後執行了 schedule()主動啟動一次調度。

② 程式 A 調用系統函數進入內核態，從內核態返回用戶態的前夕：

這時內核會判斷是否應該切換程式。

③ 程式 A 正在使用者態運行，發生了中斷：

內核處理完中斷，繼續執行程式 A 的使用者態指令的前夕，它會判斷是否應該切換程式。

從這個過程可知，對於“不可搶佔”的內核，當程式 A 運行內核態代碼時進程是無法切換的(除非程式 A 主動放棄)，比如執行某個系統調用、執行某個驅動時，進程無法切換。

這會導致 2 個問題：

① 優先順序反轉：

一個低優先順序的程式，因為它正在內核態執行某些很耗時的操作，在這一段時間內更高優先順序的程式也無法運行。

② 在內核態發生的中斷不會導致進程切換

為了讓系統的即時性更佳，Linux 內核引入了“搶佔”(preempt)的功能：進程運行于內核態時，進程調度也是可以發生的。

回到上面的例子，程式 A 調用某個驅動執行耗時的操作，在這一段時間內系統是可以切換去執行更高優先順序的程式。

對於可搶佔的內核，編寫驅動程式時要時刻注意：你的驅動程式隨時可能被打斷、隨時是可以被另一個進程來重新執行。對於可搶佔的內核，在驅動程式中要考慮對臨界資源加鎖。

1.4.5 使用場景

本節參考：https://wenku.baidu.com/view/26adb3f5f61fb7360b4c656e.html
英文原文：https://mirrors.edge.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/

1.4.5.1 只在用戶上下文加鎖

假設只有程式 A、程式 B 會搶佔資源，這 2 個程式都是可以休眠的，所以可以使用信號量，代碼如下：

```
static DEFINE_SPINLOCK(clock_lock); // 或 struct semaphore sem;  sema_init(&sem, 1);
if (down_interruptible(&sem))  // if (down_trylock(&sem))
{
    /* 獲得了信號量 */
}

/* 釋放信號量 */
up(&sem);
```

對於 down_interruptible 函數，如果信號量暫時無法獲得，此函數會令程式進入休眠；別的程式調用 up()函數釋放信號量時會喚醒它。

在 down_interruptible 函數休眠過程中，如果進程收到了信號，則會從 down_interruptible 中返回；對應的有另一個函數 down，在它休眠過程中會忽略任何信號。

注意：“信號量”(semaphore)，不是“信號”(signal)。

也可以使用 mutex，代碼如下：

```
static DEFINE_MUTEX(mutex);  //或 static struct mutex mutex; mutex_init(&mutex);
mutex_lock(&mutex);
/* 臨界區 */
mutex_unlock(&mutex);
```

注意：一般來說在同一個函數裡調用 mutex_lock 或 mutex_unlock，不會長期持有它。這只是慣例，如果你使用 mutex 來實現驅動程式只能由一個進程打開，在 drv_open 中調用 mutex_lock，在 drv_close 中調用 mutex_unlock，這也完全沒問題。

1.4.5.2 在用戶上下文與 Softirqs 之間加鎖

假設這麼一種情況：程式 A 運行到內核態時，正在訪問一個臨界資源；這時發生了某個硬體中斷，在硬體中斷處理完後會處理 Softirq，而某個 Softirq 也會訪問這個臨界資源。

怎麼辦？

在程式 A 訪問臨界資源之前，乾脆禁止 Softirq 好了！

可以使用 spin_lock_bh 函數，它會先禁止本地 CPU 的中斷下半部即 Softirq，這樣本地 Softirq 就不會跟它競爭了；假設別的 CPU 也想獲得這個資源，它也會調用 spin_lock_bh 禁止它自己的 Softirq。這 2 個 CPU 都禁止自己的 Softirq，然後競爭 spinlock，誰搶到誰就先執行。可見，在執行臨界資源的過程中，本地 CPU 的 Softirq、別的 CPU 的 Softirq 都無法來搶佔當前程式的臨界資源。

釋放鎖的函數是 spin_unlock_bh。

spin_lock_bh/spin_unlock_bh 的尾碼是“_bh”，表示“Bottom Halves”，中斷下半部，這是軟體插斷的老名字。這些函數改名為 spin_lock_softirq 也許更恰當，請記住：spin_lock_bh 會禁止 Softirq，而不僅僅是禁止“中斷下半部”(timer、tasklet 裡等都是 Softirq，中斷下半部只是 Softirq 的一種)。

示例代碼如下：

```
static DEFINE_SPINLOCK(lock); // static spinlock_t lock; spin_lock_init(&lock);
spin_lock_bh(&lock);
/* 臨界區 */
spin_unlock_bh(&lock);
```

1.4.5.3 在用戶上下文與 Tasklet 之間加鎖

Tasklet 也是 Softirq 的一種，所以跟前面是“在用戶上下文與 Softirqs 之間加鎖”完全一樣。

1.4.5.4 在用戶上下文與 Timer 之間加鎖

Timer 也是 Softirq 的一種，所以跟前面是“在用戶上下文與 Softirqs 之間加鎖”完全一樣。

1.4.5.5 在 Tasklet 與 Timer 之間加鎖

假設在 Tasklet 中訪問臨界資源，另一個 CPU 會不會同時運行這個 Tasklet？不會的，所以如果只是在某個 Tasklet 中訪問臨界資源，無需上鎖。

假設在 Timer 中訪問臨界資源，另一個 CPU 會不會同時運行這個 timer？不會的，所以如果只是在某個 Timer 中訪問臨界資源，無需上鎖。

如果在有 2 個不同的 Tasklet 或 Timer 都會用到一個臨界資源，那麼可以使用 spin_lock()、spin_unlock()來保護臨界資源。不需要用 spin_lock_bh()，因為一旦當前 CPU 已經處於 Tasklet 或 Timer 中，同一個 CPU 不會同時再執行其他 Tasklet 或 Timer。

1.4.5.6 在 Softirq 之間加鎖

這裡講的 softirq 不含 tasklet、timer。

同一個 Softirq 是有可能在不同 CPU 上同時運行的，所以可以使用 spin_lock()、spin_unlock()來訪問臨界區。如果追求更高的性能，可以使用“per-CPU array”，本章不涉及。

不同的 Softirq 之間，可以使用 spin_lock()、spin_unlock()來訪問臨界區。

總結起來，在 Softirq 之間(含 timer、tasklet、相同的 Softirq、不同的 Softirq)，都可以使用 spin_lock()、spin_unlock()來訪問臨界區。

示例代碼如下：

```
static DEFINE_SPINLOCK(lock); // static spinlock_t lock; spin_lock_init(&lock);
spin_lock(&lock);
/* 臨界區 */
spin_unlock(&lock);
```

1.4.5.7 硬中斷上下文

假設一個硬體插斷服務常式與一個 Softirq 共用資料，需要考慮 2 點：

① Softirq 執行的過程中，可能會被硬體中斷打斷；

② 臨界區可能會被另一個 CPU 上的硬體中斷進入。

怎麼辦？

在 Softirq 獲得鎖之前，禁止當前 CPU 的中斷。

在硬體插斷服務常式中不需要使用 spin_lock_irq()，因為當它在執行的時間 Softirq 是不可能執行的；它可以使用 spin_lock()用來防止別的 CPU 搶佔。

如果硬體中斷 A、硬體中斷 B 都要訪問臨界資源，怎麼辦？這篇文章裡說要使用 spin_lock_irq()：

https://mirrors.edge.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/

但是我認為使用 spin_lock()就足夠了。因為 Linux 不支持中斷嵌套，即當前 CPU 正在處理中斷 A 時，中斷 B 不可能在當前 CPU 上被處理，不需要再次去禁止中斷；當前 CPU 正在處理中斷 A 時，假如有另一個 CPU 正在處理中斷 B，它們使用 spin_lock()實現互斥訪問臨界資源就可以了。

spin_lock_irq()/spin_unlock_irq() 會禁止/使能中斷，另一套函數是 spin_lock_irqsave()/spin_unlock_irqrestore()，spin_lock_irqsave()會先保存當前中斷狀態(使能還是禁止)，再禁止中斷；spin_unlock_irqrestore()會恢復之前的中斷狀態(不一定是使能中斷，而是恢復成之前的狀態)。

示例代碼如下：

```
static DEFINE_SPINLOCK(lock); // static spinlock_t lock; spin_lock_init(&lock);
spin_lock_irq(&lock);
/* 臨界區 */
spin_unlock_irq(&lock);
```

示例代碼如下：

```
unsigned long flags;
static DEFINE_SPINLOCK(lock); // static spinlock_t lock; spin_lock_init(&lock);
spin_lock_irqsave(&lock, flags);
/* 臨界區 */
spin_unlock_irqrestore(&lock, flags);
```

寫在最後：這個連結是一篇很好的文檔，以後我們會完全翻譯出來，現在講的知識暫時夠用了。

https://mirrors.edge.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/