Linux Device Drivers

– Kernel Synchronization

By

Anurag Mondal
Project Engineer, Embedded Team
CDAC - Hyderabad

Agenda

- Concurrencies
- Concurrency Management Techniques

Where we are???

1. Module Program
2. Understand the flow from application layer to driver
3. Various driver functionality
4. Timers and wait queues

Concurrencies

Concurrency

- The inconsistency caused due to accessing a *shared resource* parallely may lead to a situation known as *concurrency* or *race condition*.
- As an example, consider the following function :

```
int temp;
void swap(int *a, int *b)
{
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Concurrency : Example

- The example shown in the previous slide is logically correct but it leads to race conditions, if accessed by more than one thread at the same time.
- The variable 'temp' is shared among all the threads accessing the functions, which might corrupt its value.
- The function 'swap' thus can be said as a non-reentrant function.

Solution..

Make 'temp' local

```
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Acquire a lock

```
int temp;
void swap(int *a, int *b)
  {
    acquire_lock;
    temp = *a;
    *a = *b;
    *b = temp;
    release_lock;
  }
```

Sources of concurrencies in the Kernel

- **Multiple user-space processes are running, which can access our code in surprising combination of ways.**
- **Device Interrupts**
- **Asynchronous kernel events : workqueues, timers, tasklets, etc.**

Concurrency Management Techniques

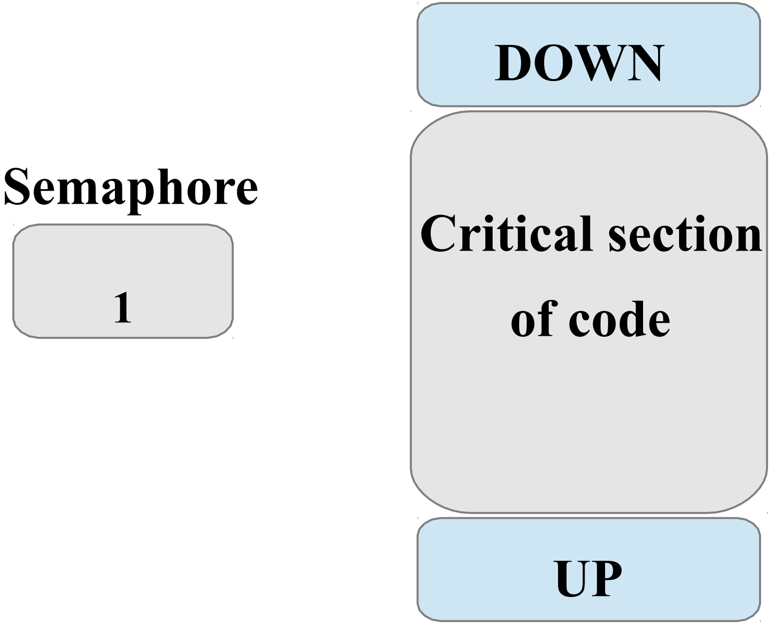
Concurrency Management Techniques

- Semaphores
- Spinlocks
- Completions
- Sequential lock
- Atomic Variables

Semaphores

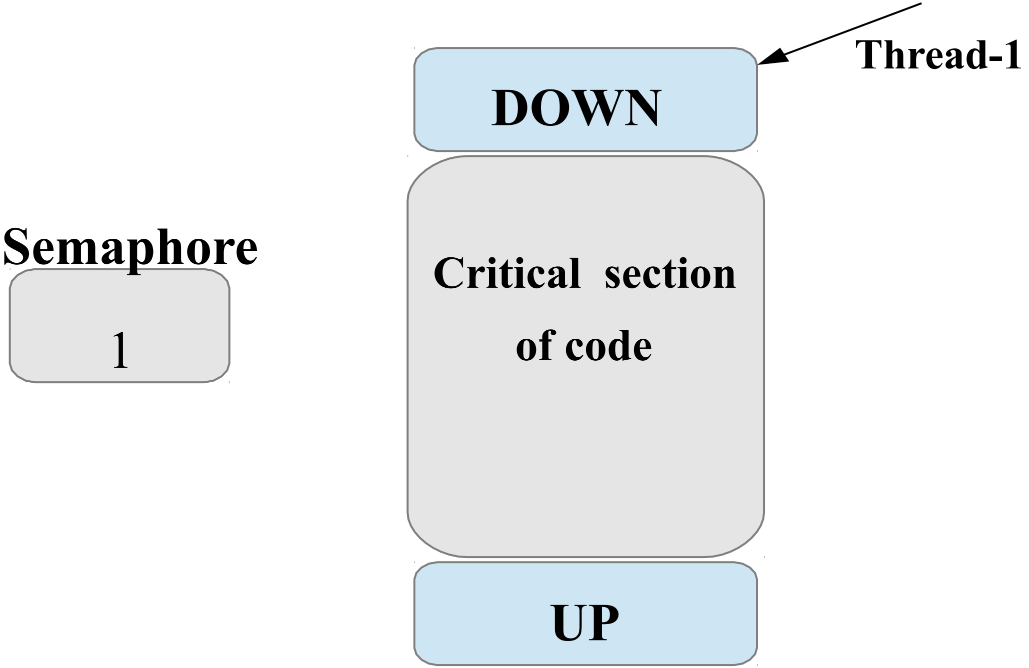
Semaphores in action... Case -1

- Semaphore and critical sections are setup



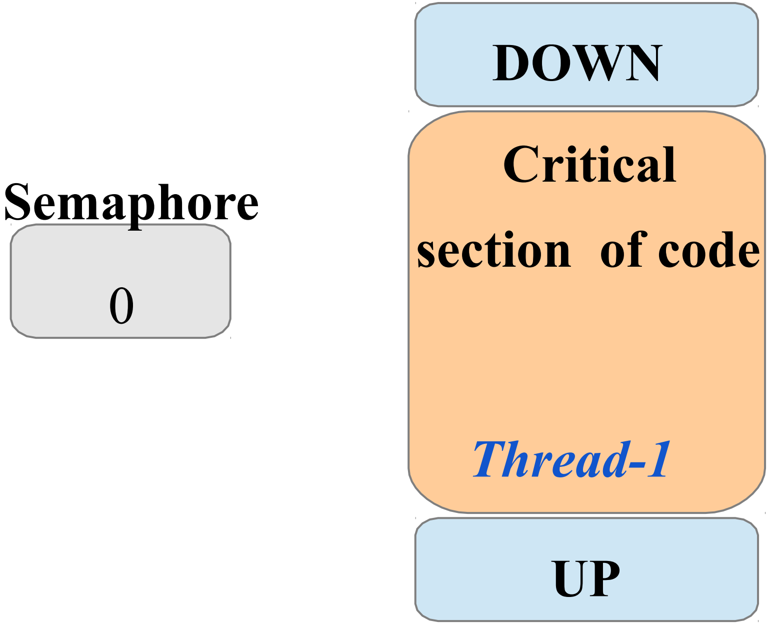
Semaphores in action... Case -1

Thread-1 arrives and tries to acquire the semaphore



Semaphores in action... Case -1

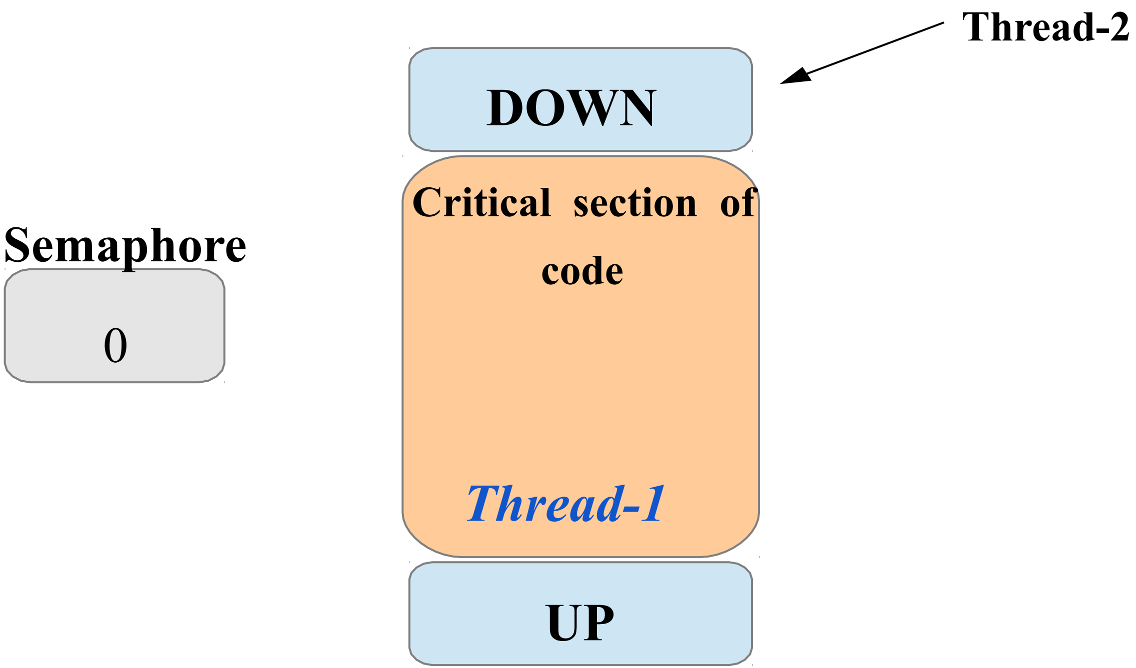
Thread-1 enters the critical section by acquiring the semaphore



Thread-1 can goto sleep even after acquiring the lock

Semaphores in action... Case -1

Now Thread-2 appears while Thread-1 is still in critical section



Semaphores in action... Case -1

Thread-2 finds that the semaphore is taken and thus goes to sleep

Semaphore

0

DOWN

Critical section of code

Thread-1

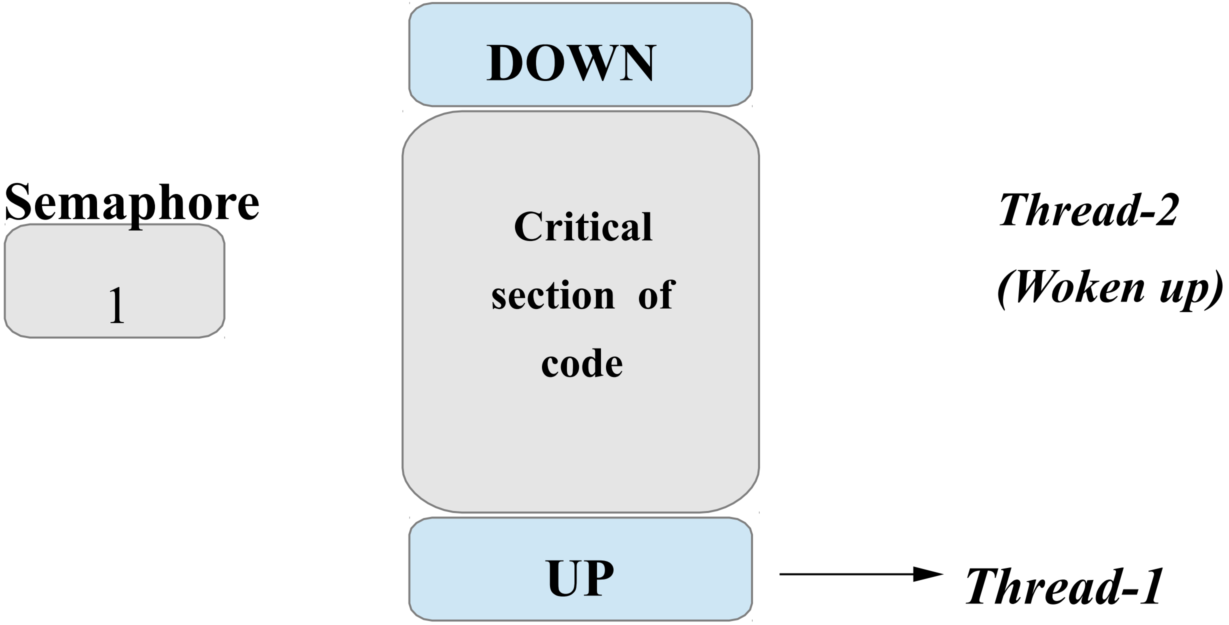
UP

Thread-2 :

-- Pushed onto the wait-queue

Semaphores in action... Case -1

Thread-1 is now out of the critical section and releases the semaphore



Semaphores in action... Case -1

Thread-2 tries again later and finds that the semaphore is now available and enters the critical section

Semaphore

0

DOWN

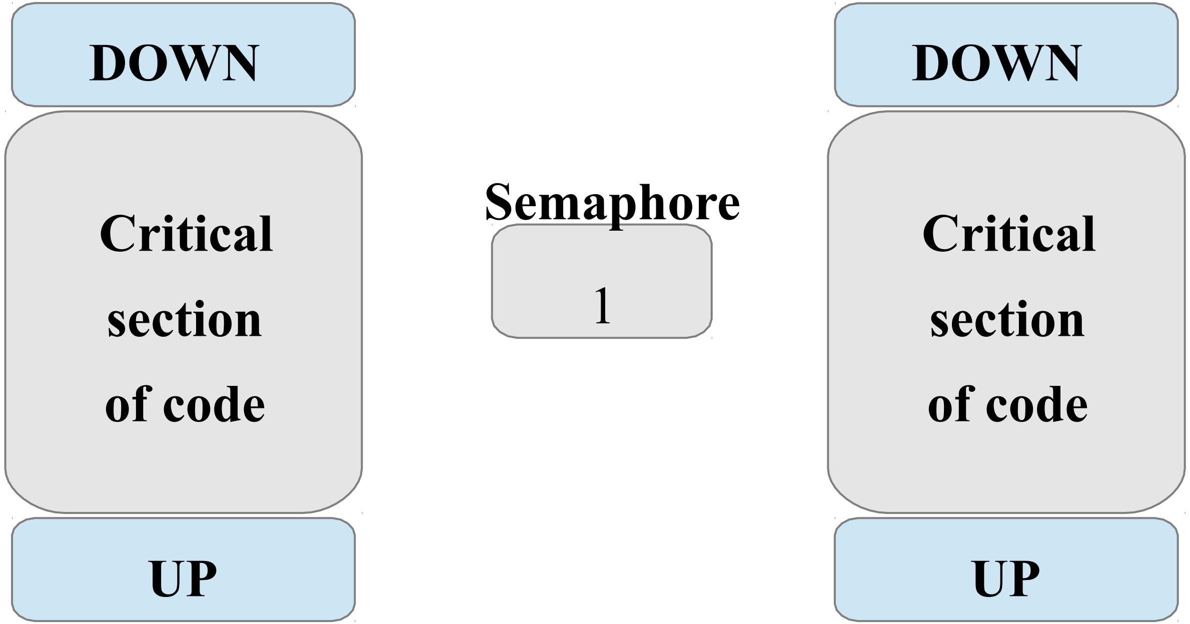
Critical section of code

Thread-2

UP

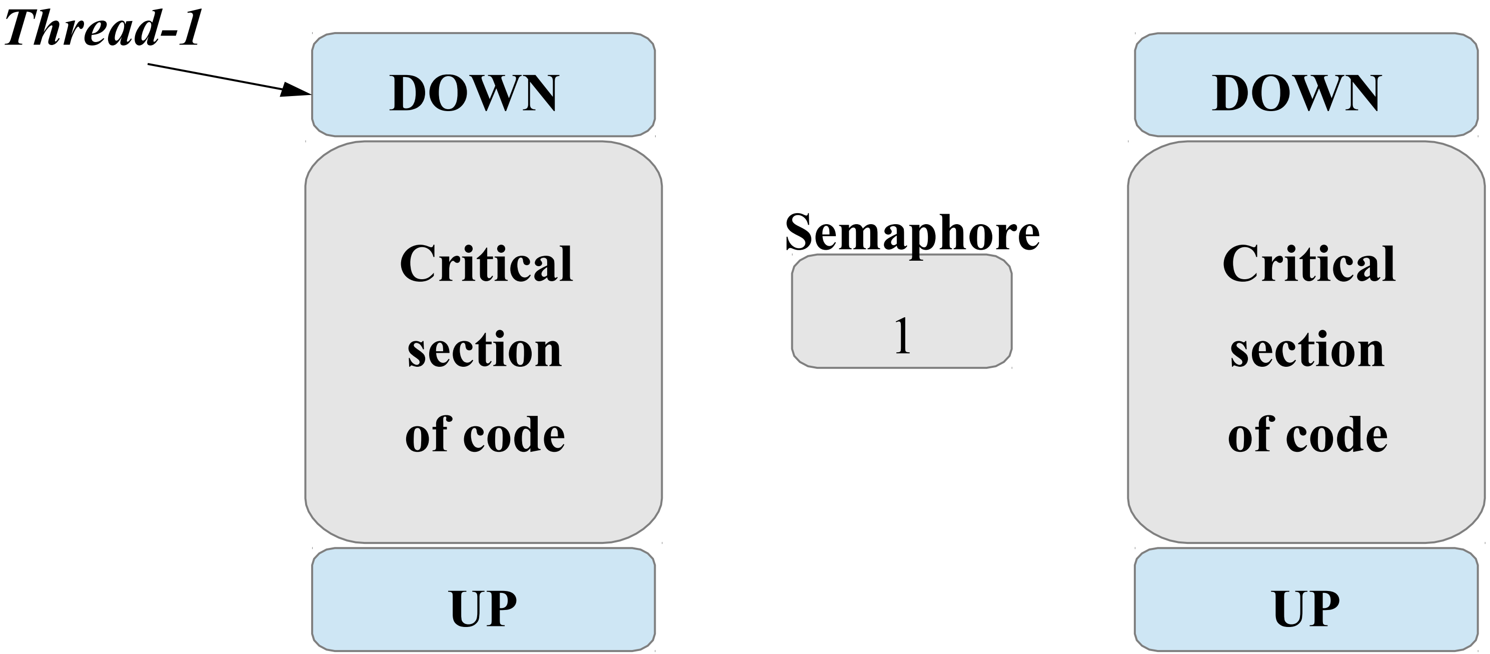
Semaphores in action... Case -2

Semaphore and critical sections are setup



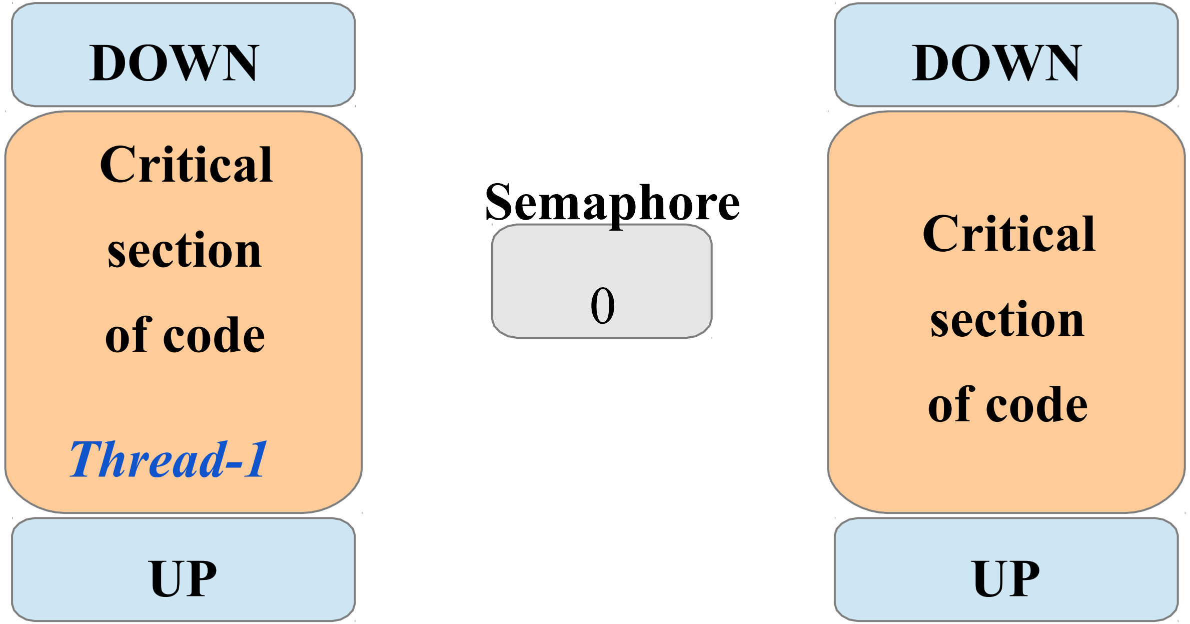
Semaphores in action... Case -2

Thread-1 tries to acquire the semaphore



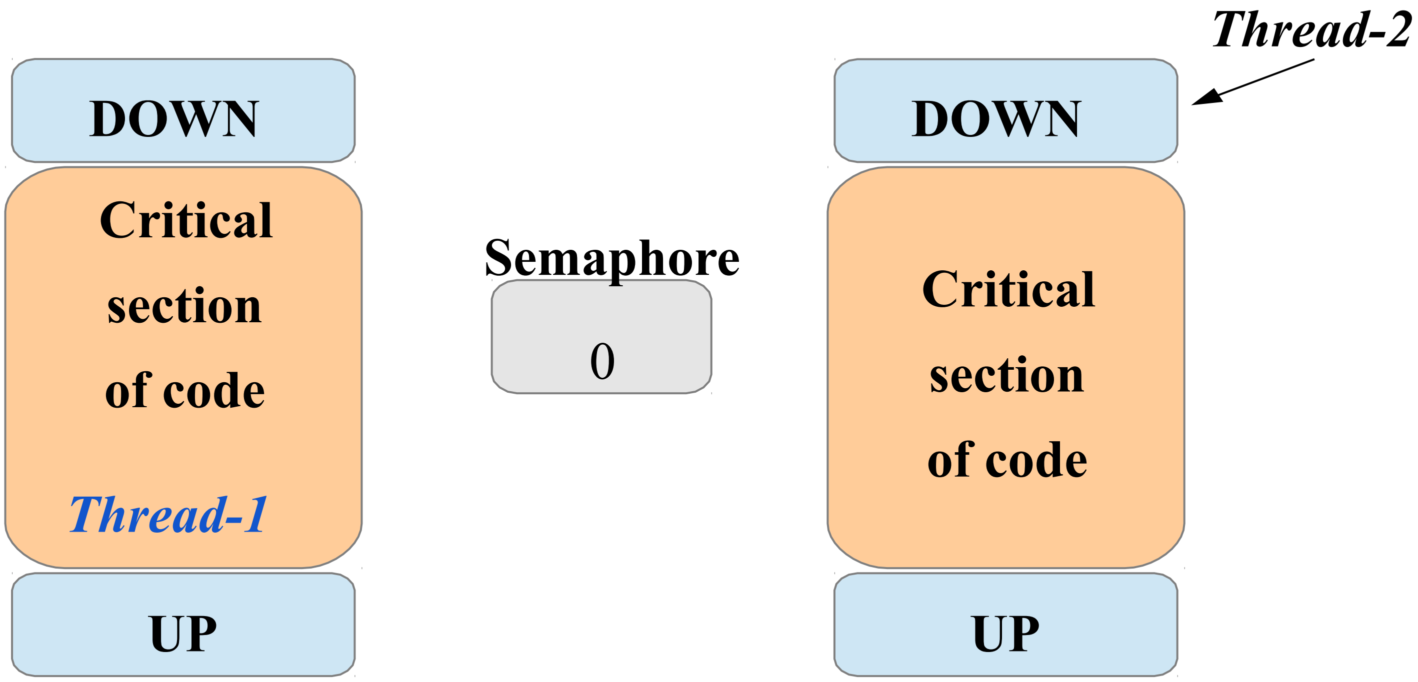
Semaphores in action... Case -2

Thread-1 is now in its critical section



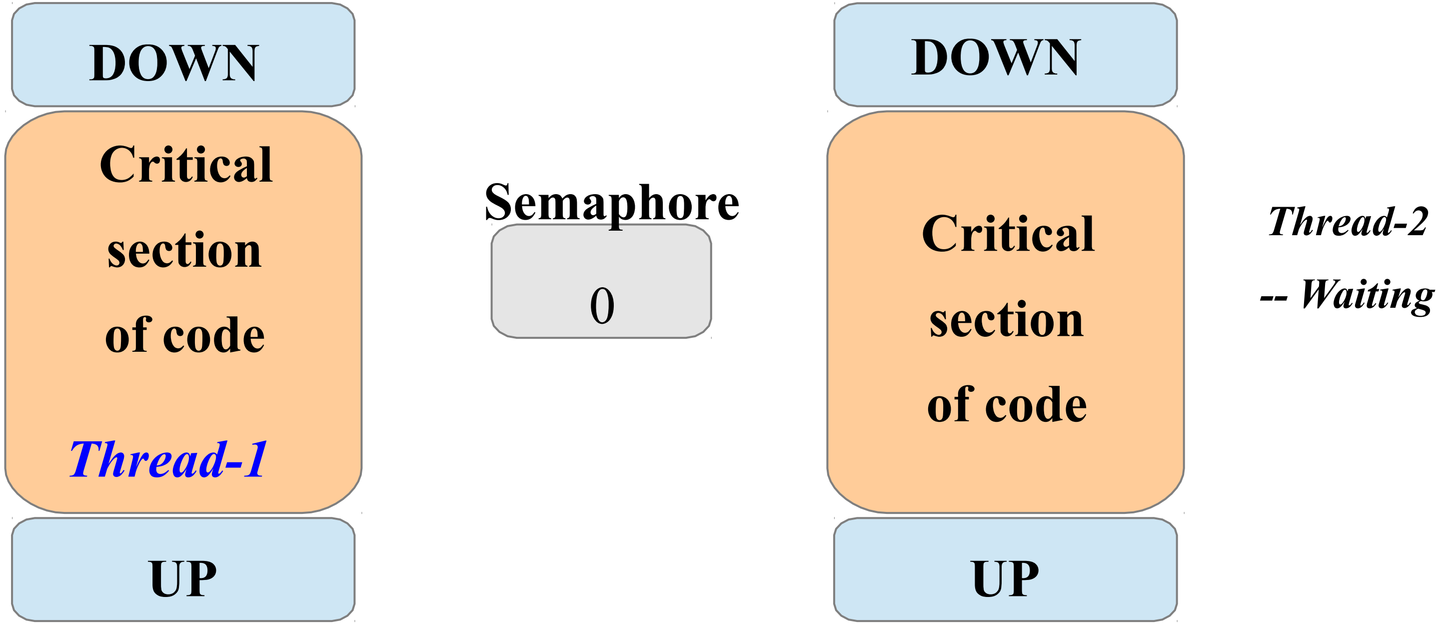
Semaphores in action... Case -2

Thread-2 arrives and tries to access another instance protected by the same semaphore



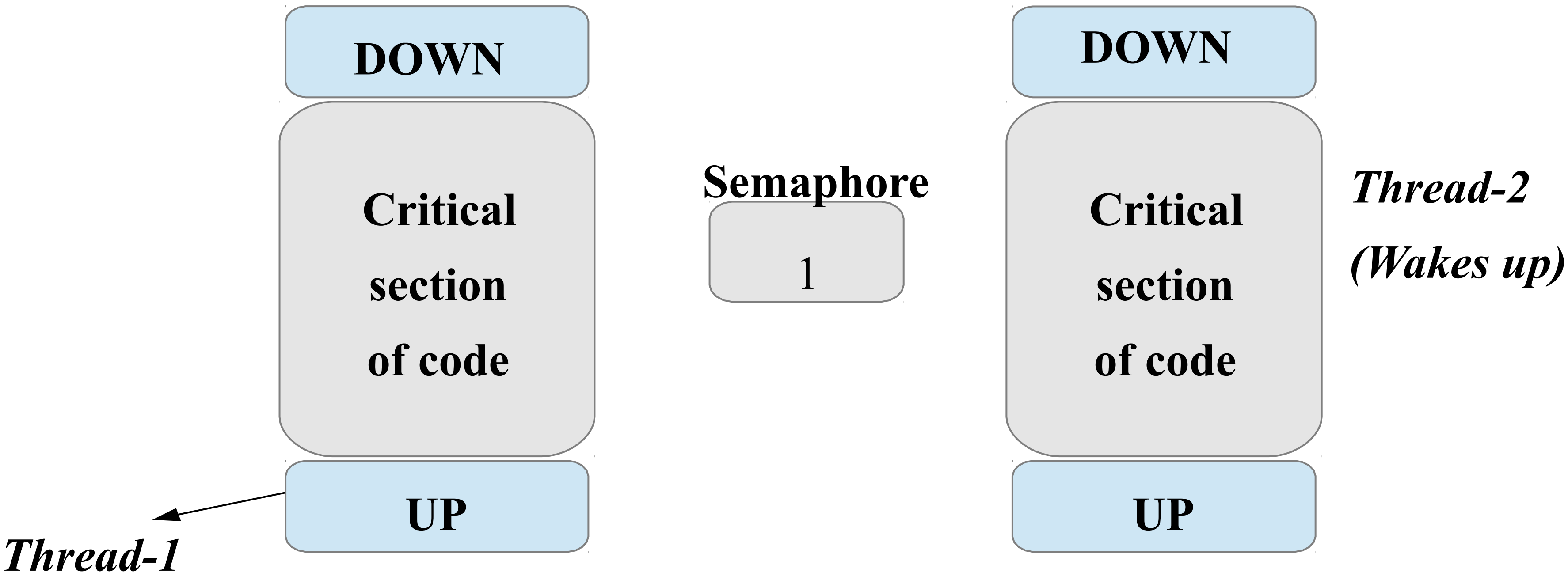
Semaphores in action... Case -2

- Thread-2 arrives and tries to access another instance protected by the same semaphore



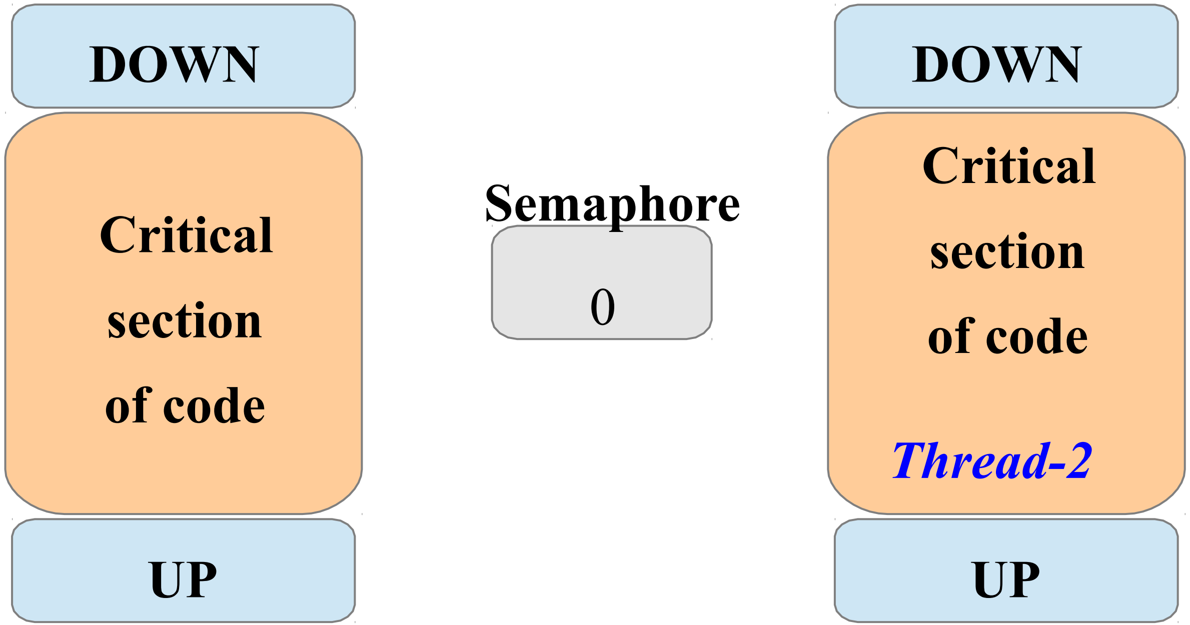
Semaphores in action... Case -2

Thread-1 releases the semaphore thus waking up Thread- 2



Semaphores in action... Case -2

Thread-2 now succeeds in acquiring the semaphore



Semaphores : Theory

- "Go to sleep" is a well-defined term in this context.
- The process can go to sleep while waiting for its turn. Thread that owns the lock can sleep.
- Suitable for locking in process context.
- Should be avoided in interrupt context as it is non-schedulable.
- Not optimal for locks that are held for short periods because the overhead of sleeping, maintaining the wait queue, and waking up.

Kernel APIs : Initialisations

<linux/semaphore.h>

struct semaphore;

```
struct semaphore {
        raw_spinlock_t          lock;
        unsigned int            count;
        struct list_head        wait_list;
};
```

Setup Semaphore:

Dynamically : void sema_init(struct semaphore *, int count);

count : initial value to assign a semaphore

In place of semaphore we can use mutex if count value is 1. To declare mutex kernel provide suitable macros

void init_MUTEX(struct semaphore *sem);

void init_MUTEX_LOCKED(struct semaphore *sem);

Statically :

DEFINE_SEMAPHORE(name);

DECLARE_MUTEX(name); - semaphore variable name is initialized to 1

DECLARE_MUTEX_LOCKED(name); - semaphore variable name initialize to 0

Kernel APIs : Semaphore Operations

Acquire the semaphore :

- **void down(struct semaphore *);**

- **int down_interruptible(struct semaphore *);**
- **This allows the process that is waiting on a semaphore to be interrupted by the user.**
- **If the operation is interrupted, the function returns a non-zero value and the caller does not hold the semaphore.**

- **int down_trylock(struct semaphore *);**
  This function never sleeps

Release the semaphore :

void up(struct semaphore *);

Self Study…..

1. **Read/Write Semaphore and its details for all the API.**

2. **Try to implement Mutex.**

3. **Try Read/Write Semaphore implementation in an application**

Spinlocks

Spinlocks in action...

Spinlocks and critical sections are setup

Spinlock

unlocked

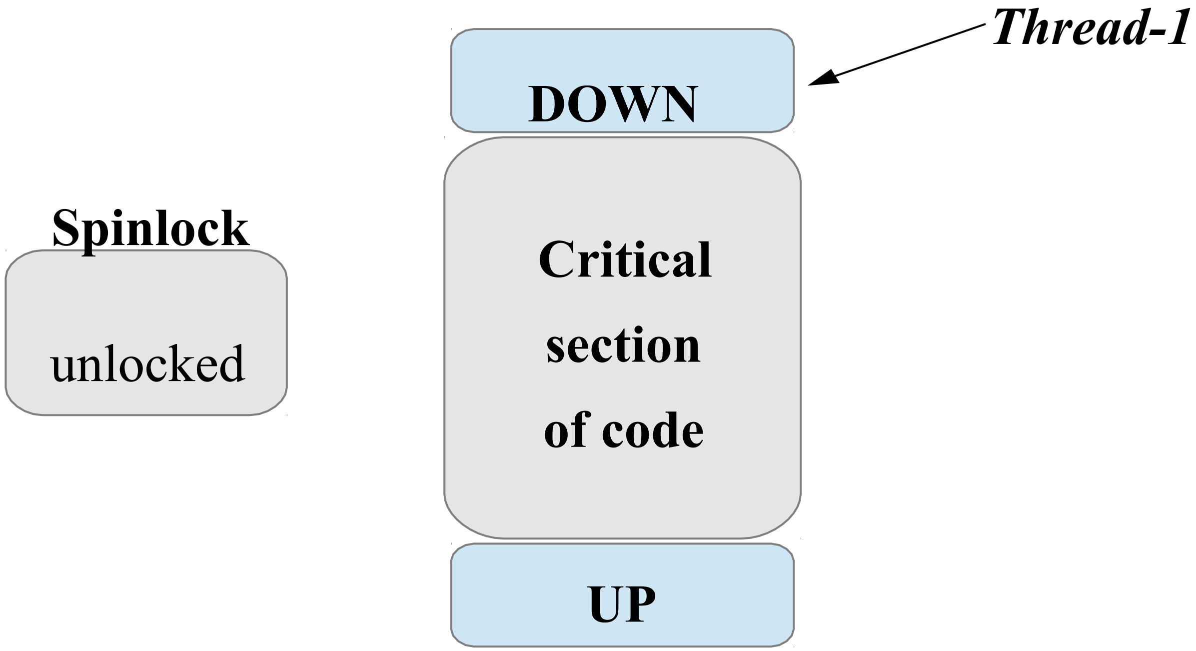
DOWN

Critical section of code

UP

Spinlocks in action...

Thread-1 tries to acquire the spinlock



Spinlocks in action...

Thread-1 enters the critical section by acquiring the spinlock

Spinlock

locked

DOWN

Critical section of code

Thread-1

UP

Thread-1 :

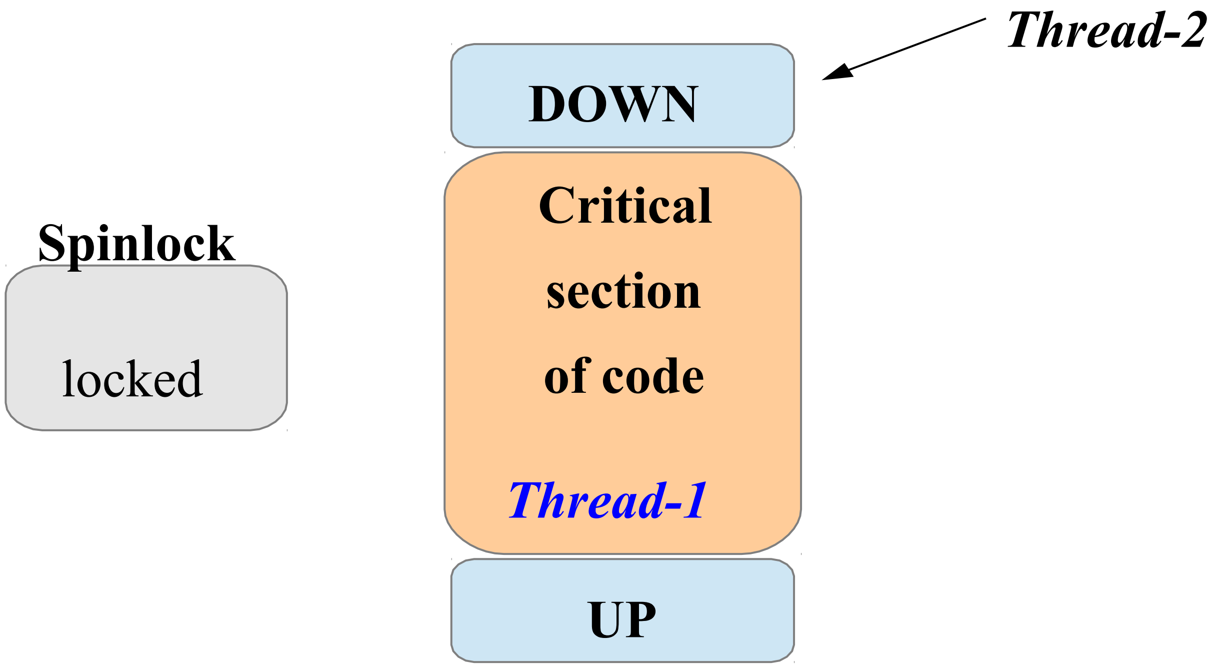
-- Acquired the lock

-- Preemption is disabled

-- Cannot goto sleep

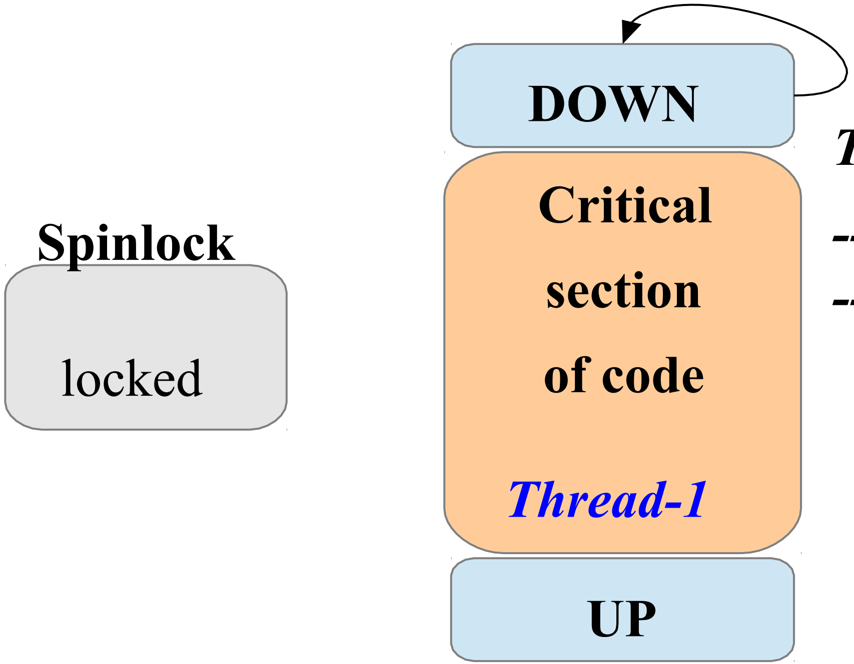
Spinlocks in action...

Now Thread-2 appears while Thread-1 is still in critical section



Spinlocks in action...

Thread-2 finds that the spinlock and forms a tight loop until the lock is free



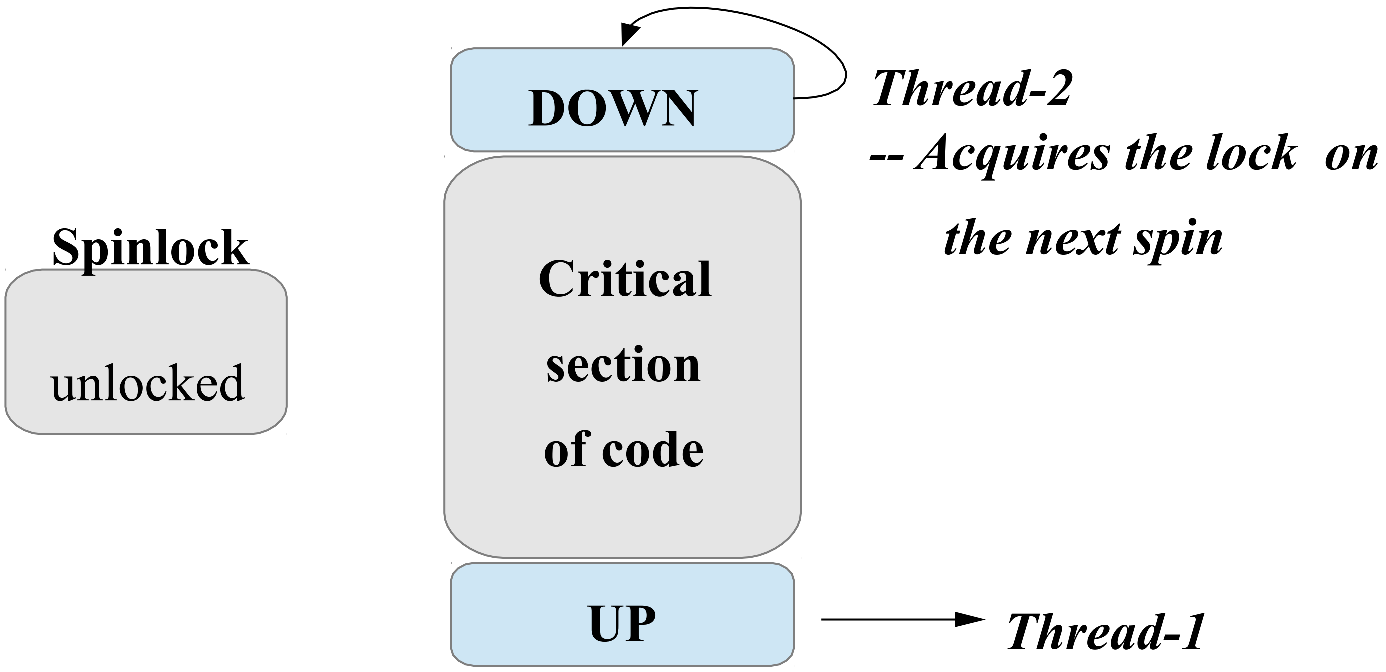
Thread-2 :

-- Busy loops

-- Wastes processor's time

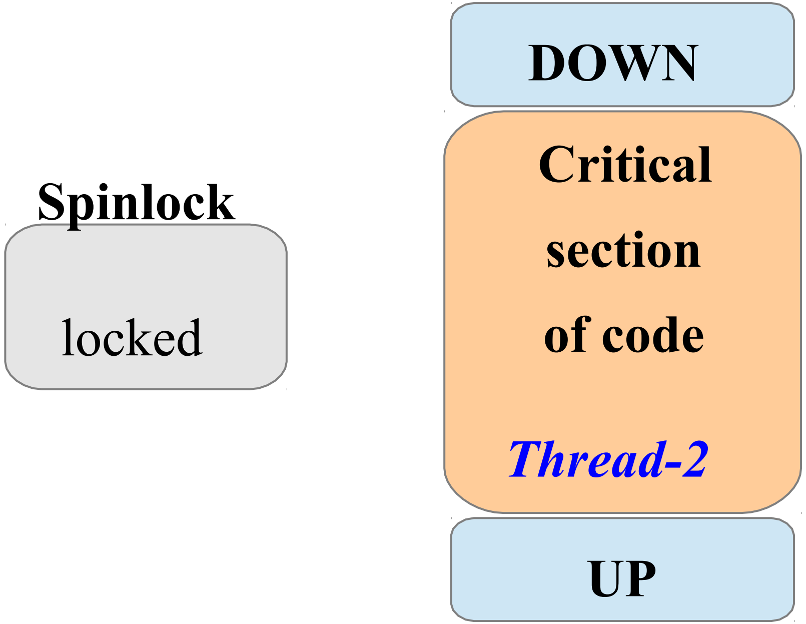
Spinlocks in action...

Thread-1 is now out of the critical section and releases the spinlock



Spinlocks in action...

Thread-2 finally acquires the lock and continues with the critical section



Spinlocks : Theory

- A spinlock is a mutual exclusion device that can have only two values: “locked” and “unlocked.”
- If the lock is available, the “locked” bit is set and the code continues into the critical section.
- If, instead, the lock has been taken by somebody else, the code goes into a tight loop where it repeatedly checks the lock until it becomes available
- Unlike semaphores, spinlocks may be used in code that cannot sleep, such as interrupt handlers.
- Spinlocks offer higher performance than semaphores in general

Spinlocks : Theory cont...

- The preemption is disabled on the current processor when the lock is taken.
- Hence, spinlocks are, by their nature, intended for use on multiprocessor systems.
- As the preemption is disabled, the code that has taken the lock must not sleep as it wastes the current processor's time or might lead to deadlock, in an uniprocessor system
- Spinlocks must be held for as minimum time as possible as it might make the other process to spin or make a high priority process wait as preemption is disabled.
- Spinlock waits are, by their nature, uninterruptible

Kernel APIs

<linux/spinlock.h>

spinlock_t;

Initialisation :

- **Dynamically :** void spin_lock_init(spinlock_t *);
- **Statically :** DEFINE_SPINLOCK(name);

Locking : void spin_lock(spinlock_t *);

Unlocking : void spin_unlock(spinlock_t *);

Kernel APIs : Other locking variants

void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);

It disable interrupts on the local processor before acquiring the lock and the previous interrupt state is stored in *flags.*

void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);

Unlocks the given lock and returns interrupts to its previous state. This way, if interrupts were initially disabled, your code would not enable them, but instead keep them disabled

Kernel APIs : Other locking variants

If you always know before the fact that interrupts are initially enabled, there is no need to restore their previous state.You can unconditionally enable them on unlock

```
void spin_lock_irq (spinlock_t *lock);

void spin_unlock_irq (spinlock_t *lock);
```

Kernel APIs : Other locking variants

- The following versions disables software interrupts before taking the lock, but leaves hardware interrupts enabled.

  - void spin_lock_irq_bh (spinlock_t *lock);
  - void spin_unlock_irq_bh (spinlock_t *lock);

- Trylock variants : (nonblocking spinlock operations)

  - int spin_trylock(spinlock_t *lock);
  - int spin_trylock_bh(spinlock_t *lock);
  - These functions return nonzero on success (the lock was obtained), 0 otherwise.

Semaphore Vs Spinlocks

Requirement	Recommended lock
• Low overhead locking	Spinlock
• Short lock hold time	Spinlock
• Long lock hold time	Semaphore
• Need to lock in interrupt context	Spinlock
• Need to sleep while holding lock	Semaphore

Self Study….

1. **Reader/Writer Spinlocks working and different API**
2. **Try different variants of spinlock in different applications**
3. **Try Reader/Writer Spinlock in application**

Completions

Completions

Completions are a lightweight mechanism allowing one thread to tell another that the job is done.

Completion variable :
<linux/completion.h>

struct completion;

Initialisation :

- Statically : DECLARE_COMPLETION(name);
- Dynamically : void init_completion(struct completion *);

Operations :

- void wait_for_completion(struct completion *);
- void complete(struct completion *);

Self Study…

1. **Sequential lock details and implementation**

2. **Implementation Atomic Variables and apply on an application**

Reference…..

1. LINUX DEVICE DRIVERS - 3rd Edition, Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

Thank You :)