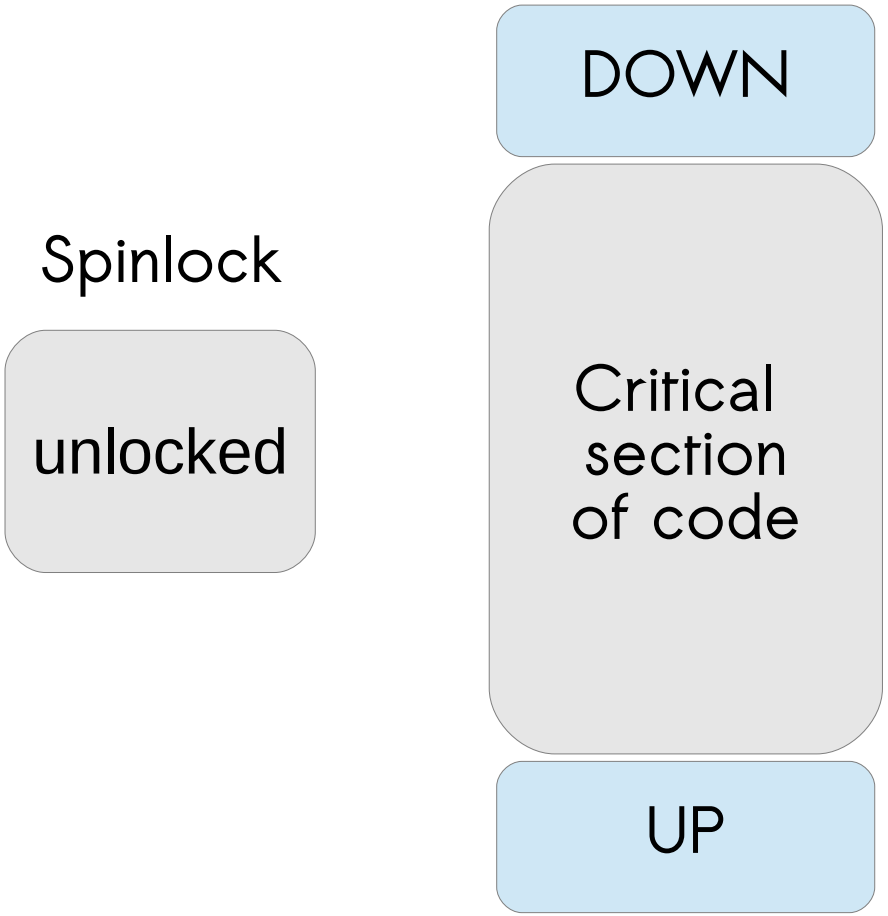
Spinlocks

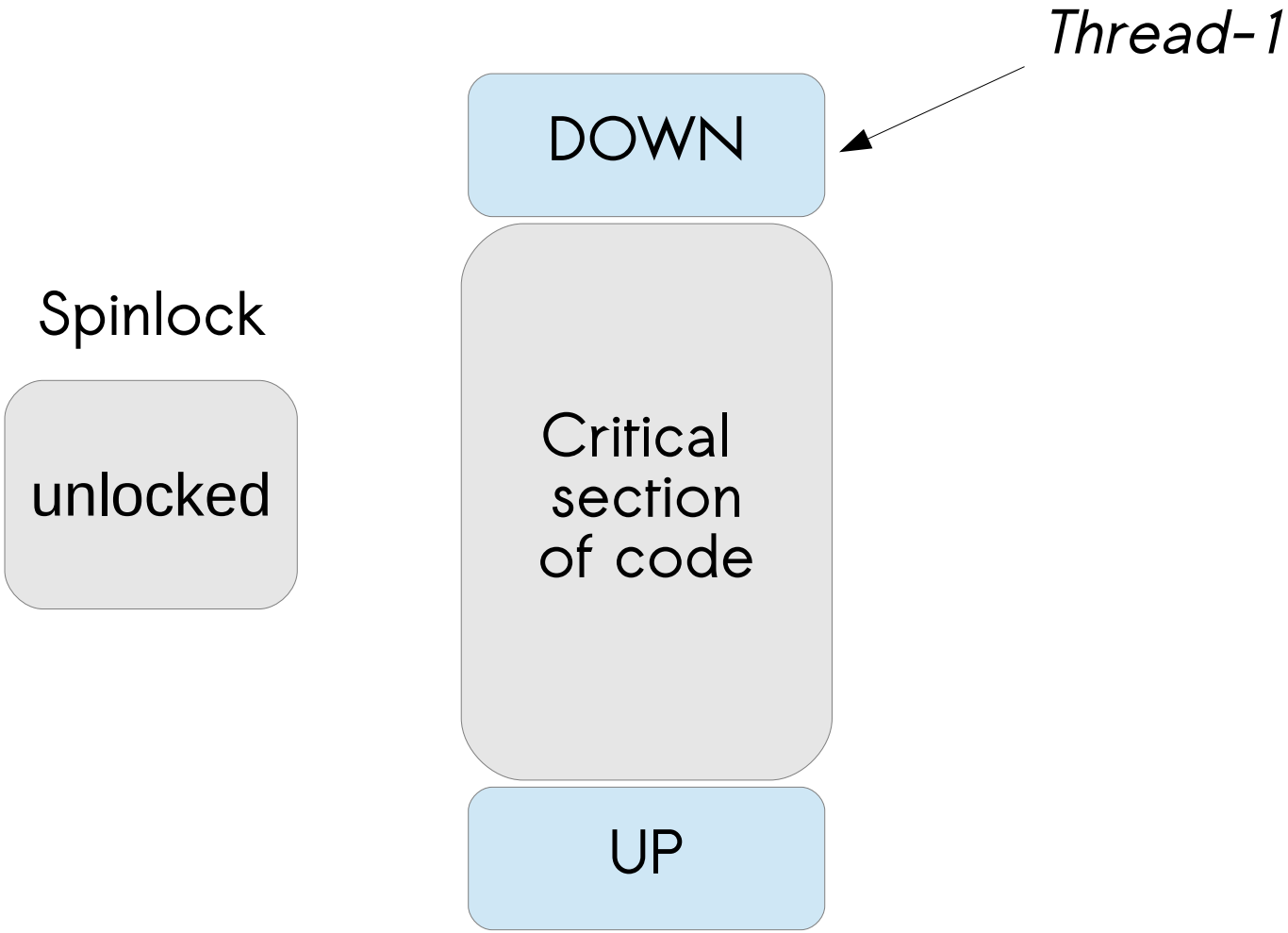
Spinlocks in action...

- Spinlocks and critical sections are setup



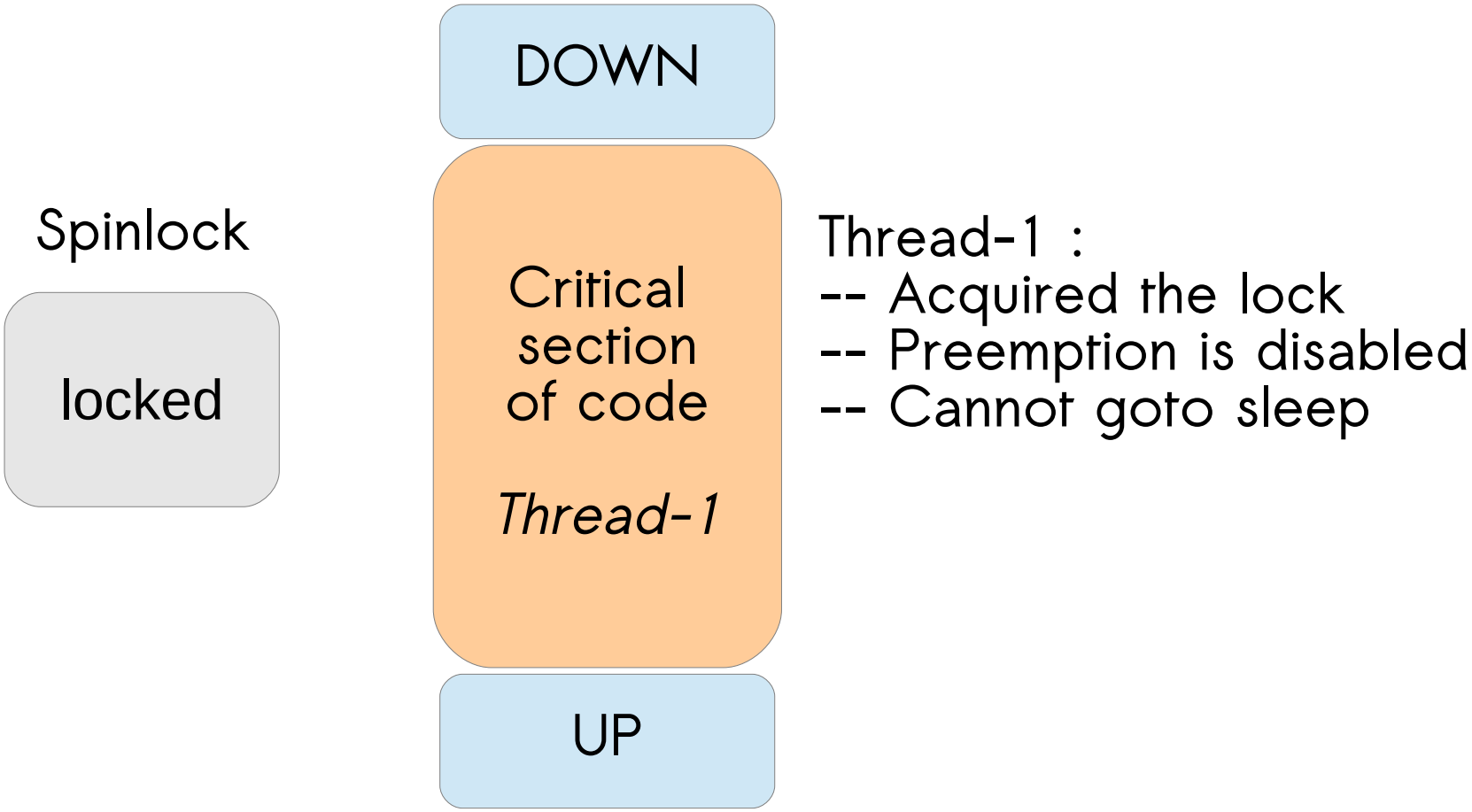
Spinlocks in action...

- Thread-1 tries to acquire the spinlock



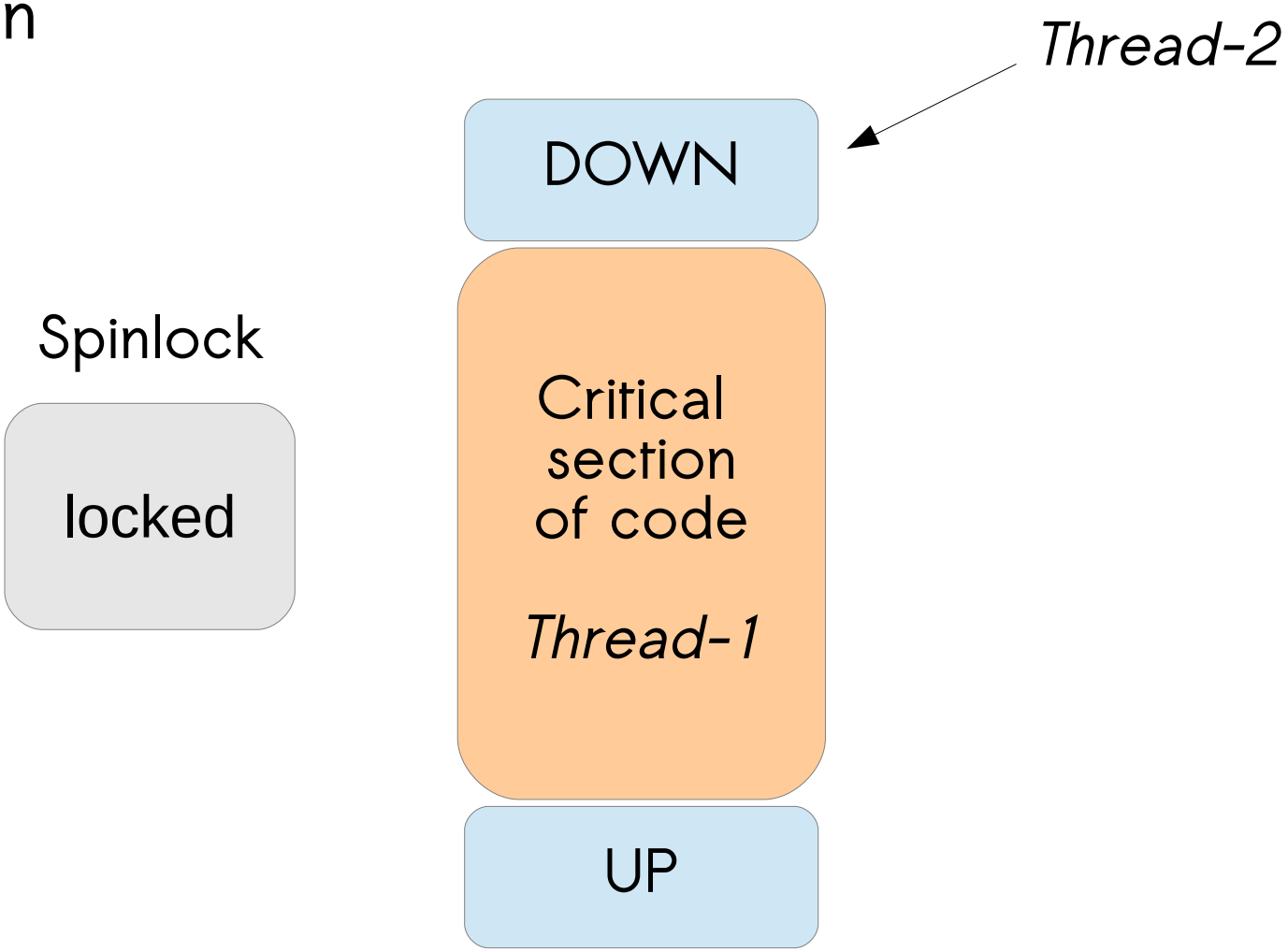
Spinlocks in action...

- Thread-1 enters the critical section by acquiring the spinlock



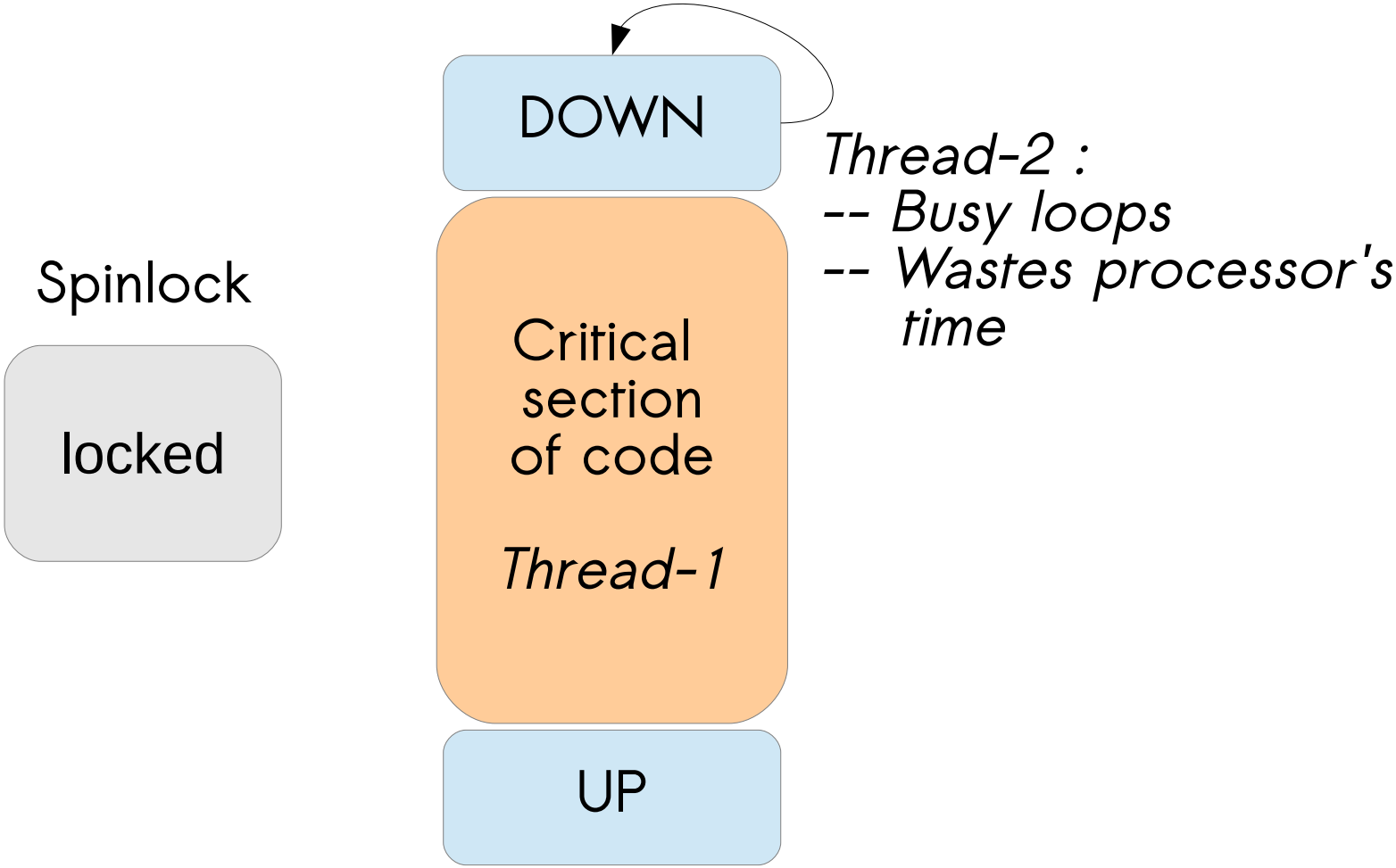
Spinlocks in action...

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



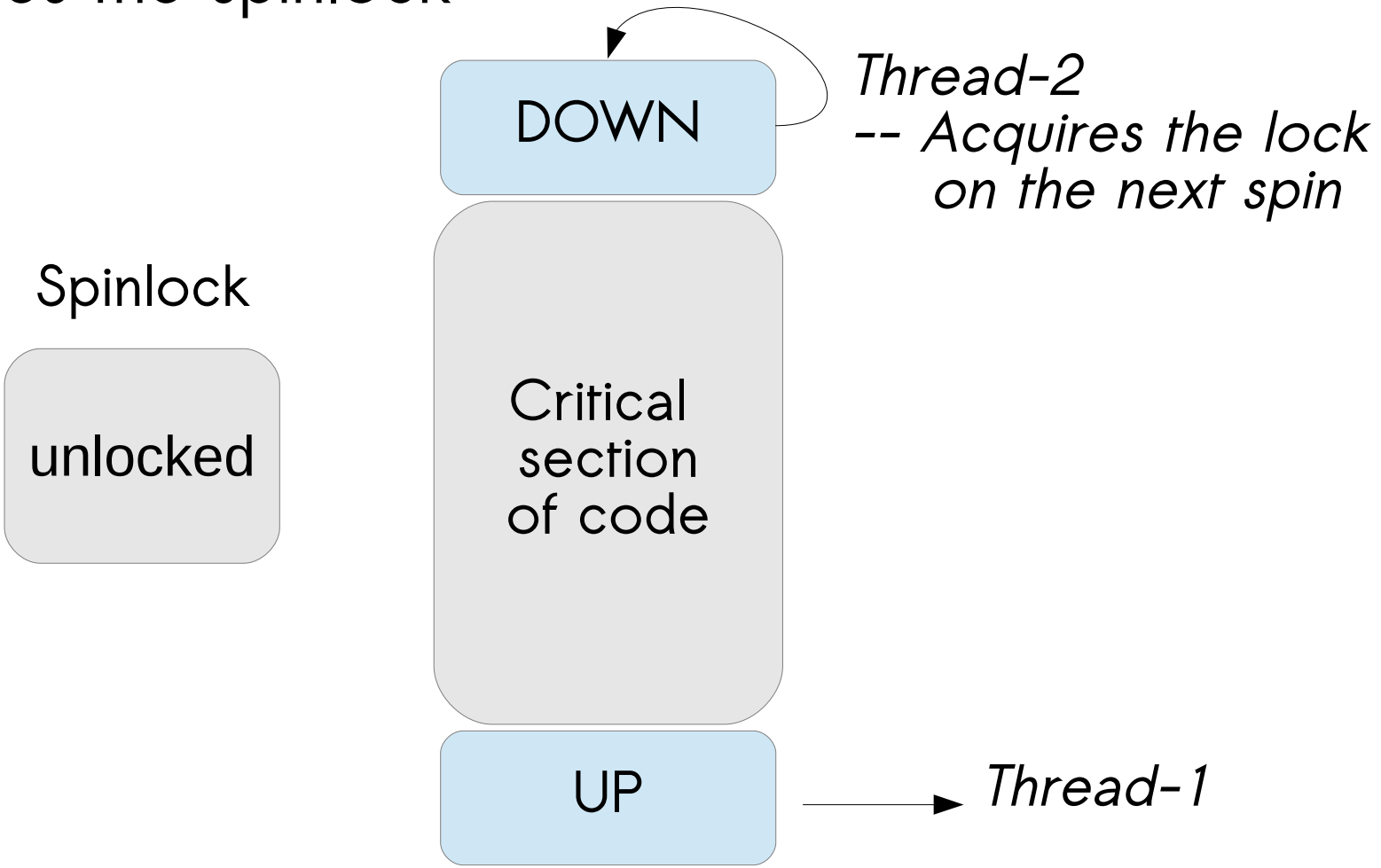
Spinlocks in action...

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



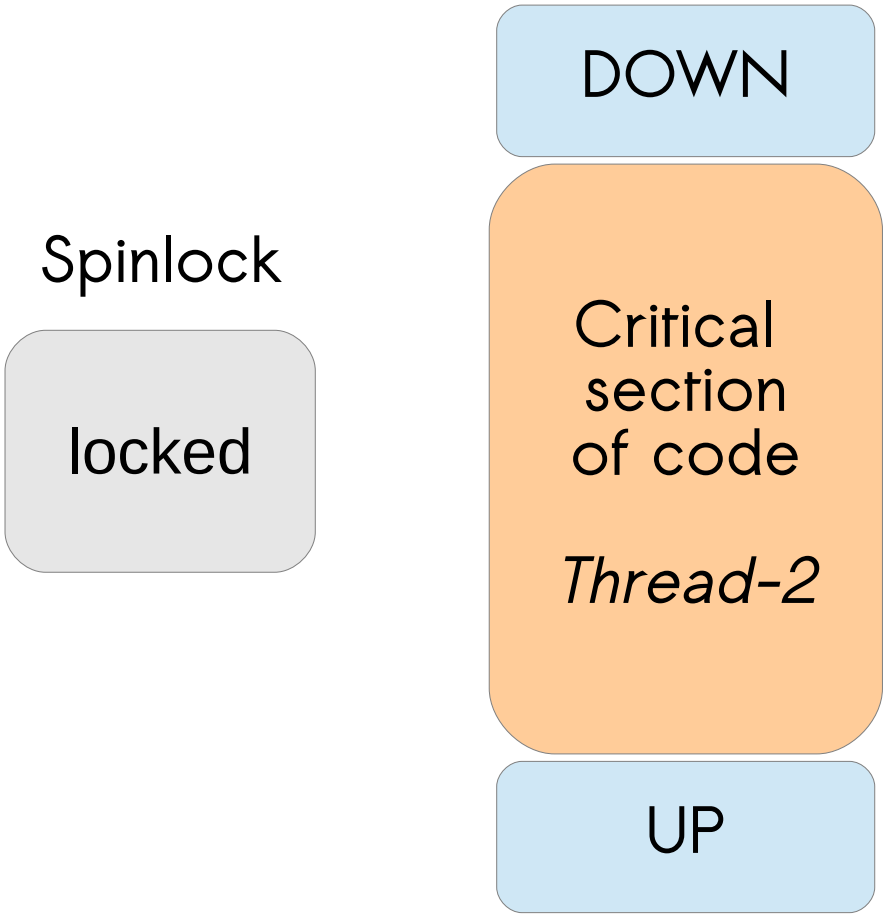
Spinlocks in action...

- 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.

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 erroneously 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_bh (spinlock_t *lock);
  - void spin_unlock_bh (spinlock_t *lock);
- Trylock variants :
  - 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.

Reader/Writer Spinlocks

- <linux/spinlock.h>
  rwlock_t;
- Reader locks :
  - void read_lock(rwlock_t *lock);
  - void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
  - void read_lock_irq(rwlock_t *lock);
  - void read_lock_bh(rwlock_t *lock);
  - void read_unlock(rwlock_t *lock);
  - void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
  - void read_unlock_irq(rwlock_t *lock);
  - void read_unlock_bh(rwlock_t *lock);

Reader/Writer Spinlocks

- Writer locks :
  - void write_lock(rwlock_t *lock);
  - void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
  - void write_lock_irq(rwlock_t *lock);
  - void write_lock_bh(rwlock_t *lock);
  - int write_trylock(rwlock_t *lock);
  - void write_unlock(rwlock_t *lock);
  - void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
  - void write_unlock_irq(rwlock_t *lock);
  - void write_unlock_bh(rwlock_t *lock);

Semaphore Vs Spinlocks

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

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