Linux Device Drivers

- Kernel Synchronization

Agenda

- Concurrencies
- Concurrency Management Techniques

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 is prone 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

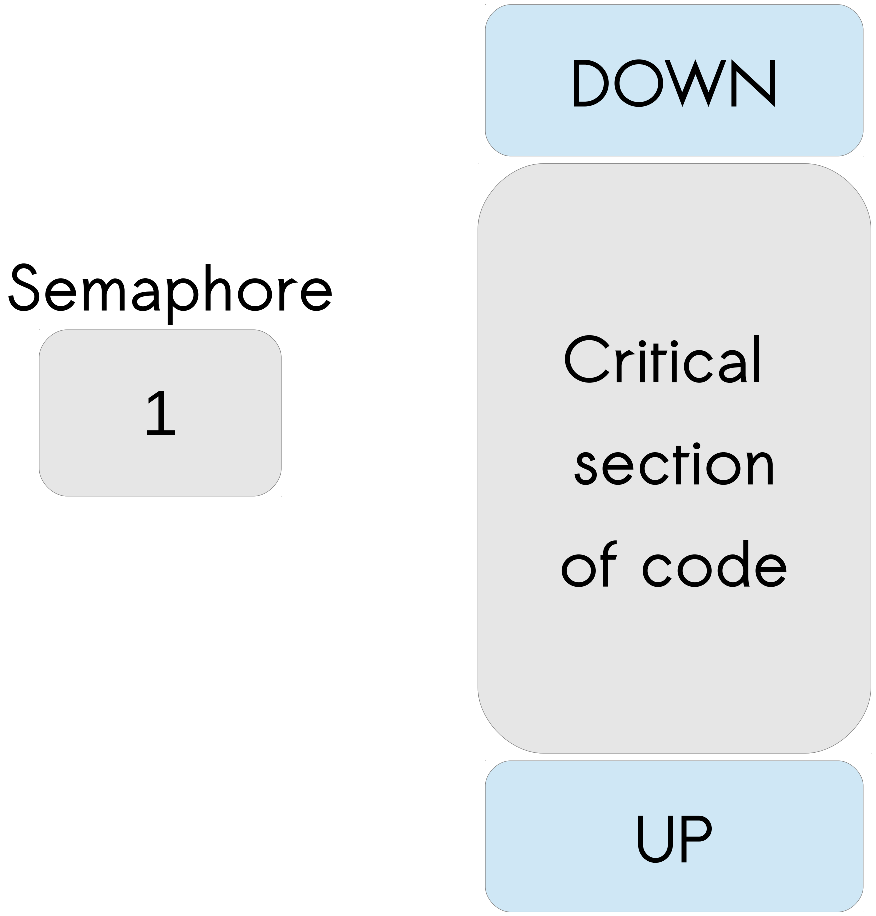
Concurrency Management Techniques

- Semaphores
- Spinlocks
- Completions
- Atomic Operations
- Sequential Locks
- Read-Copy-Update

Semaphores

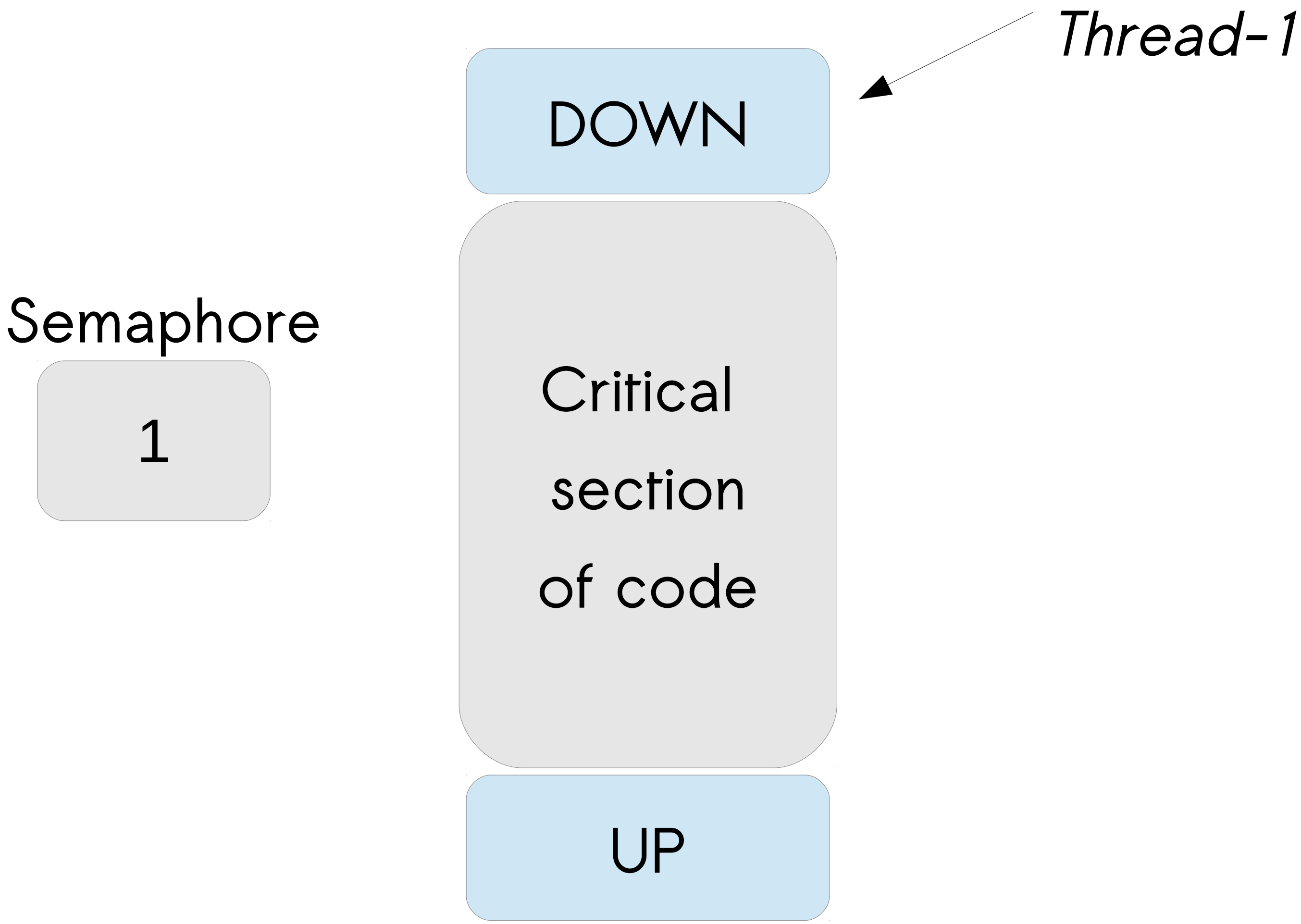
Semaphores in action... Case -1

- Semaphore and critical sections are setup



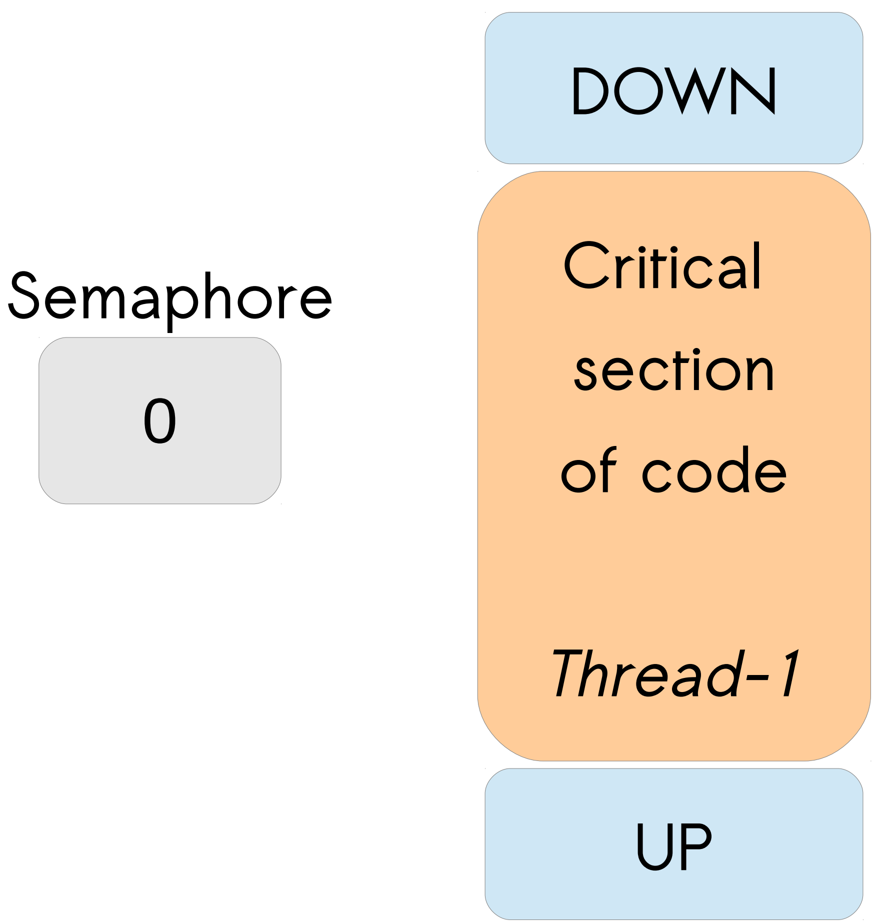
Semaphores in action... Case -1

- Thread-1 arrives and tries to acquire the semaphore



Semaphores in action... Case -1

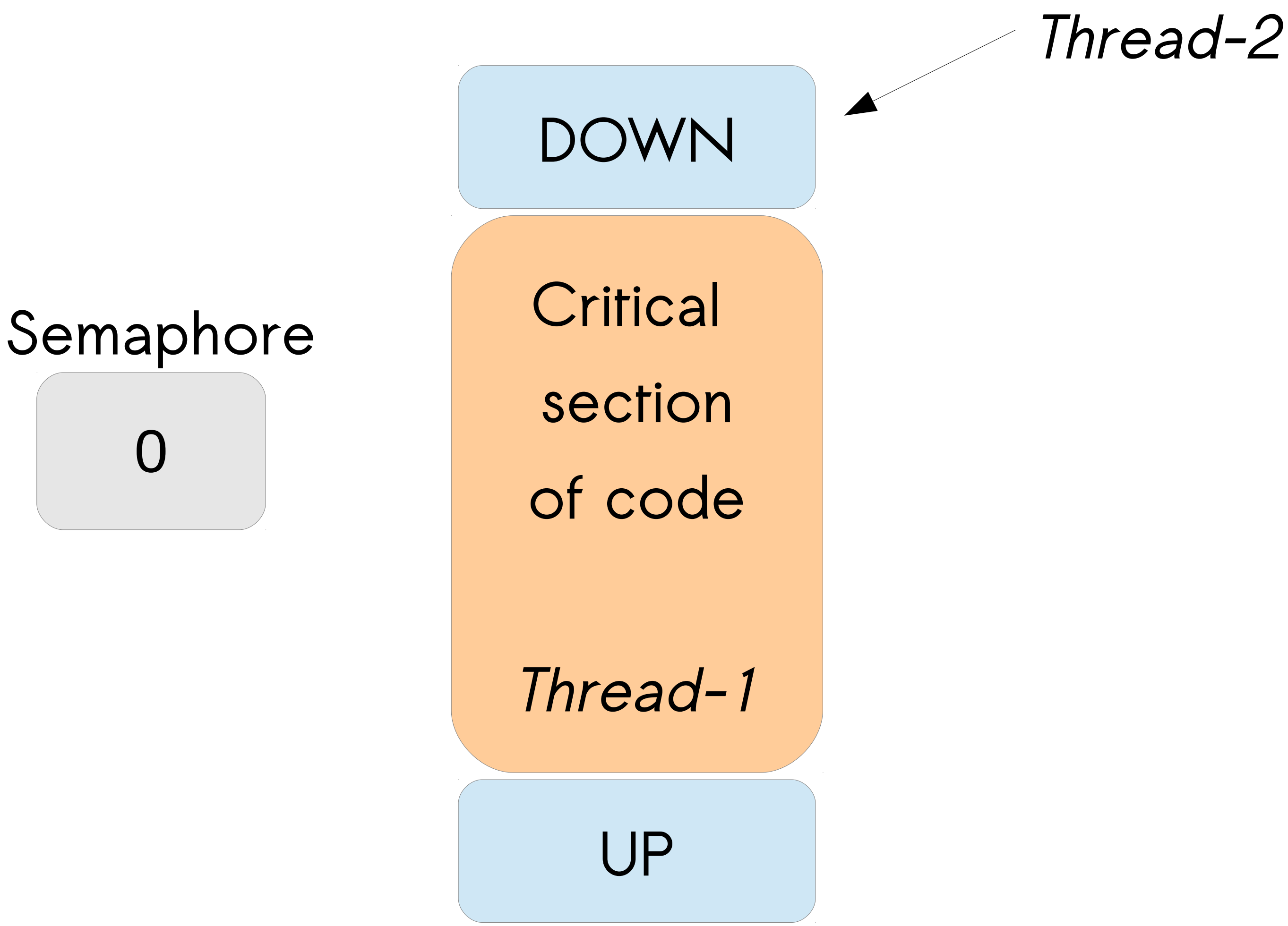
- Thread-1 enters the critical section by acquiring the semaphore



Thread-1 can goto sleep even after acquiring the lock

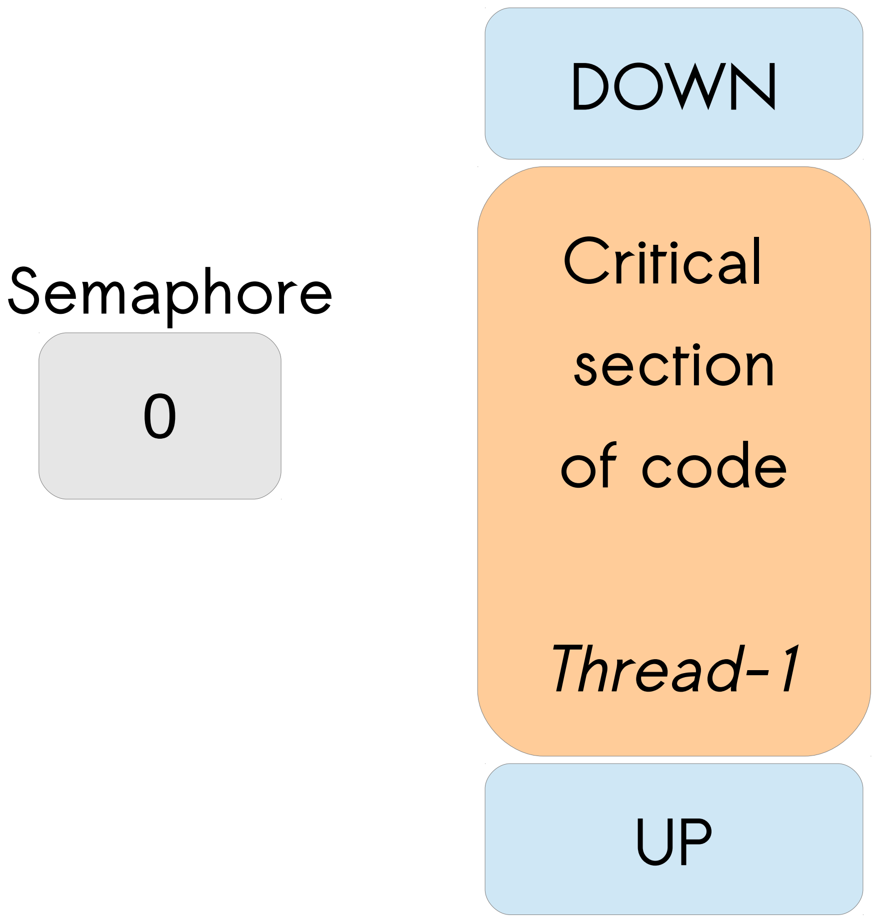
Semaphores in action... Case -1

- 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



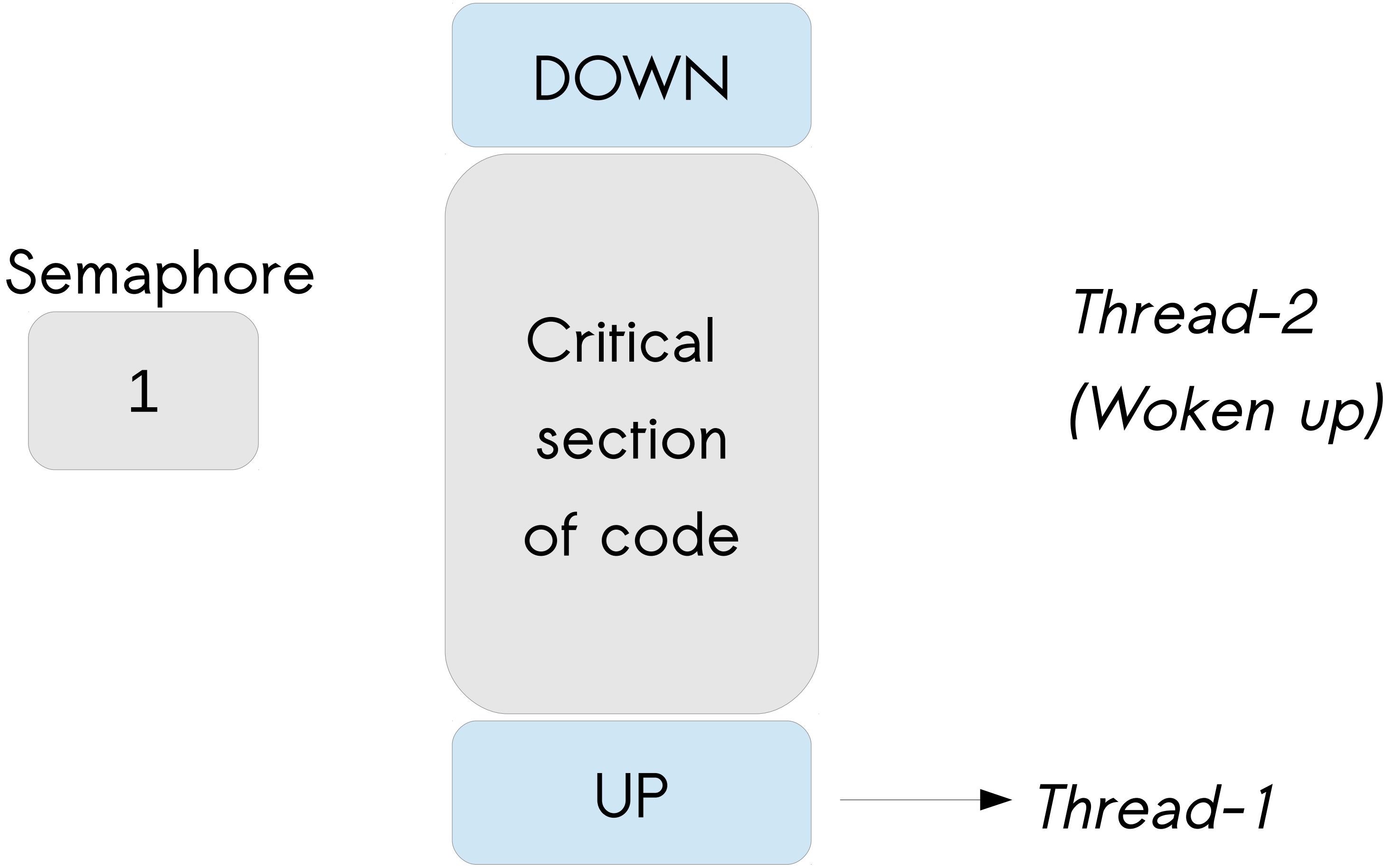
Thread-2 :

-- Pushed onto the wait-queue

-- Yields the processor

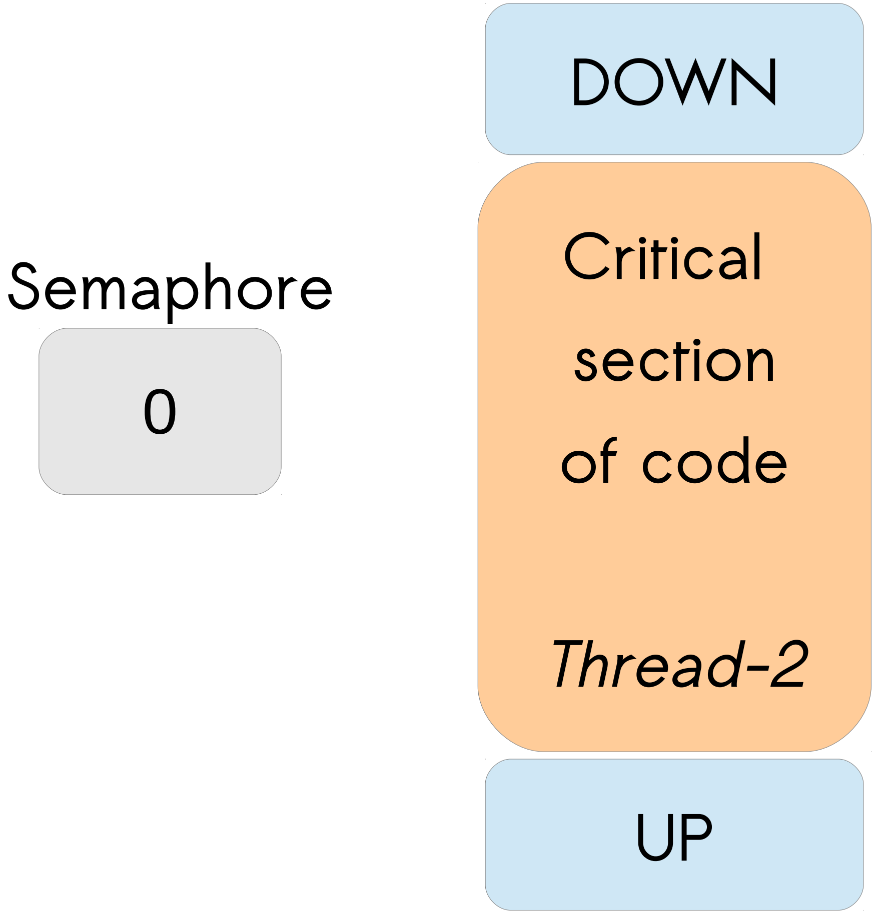
Semaphores in action... Case -1

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



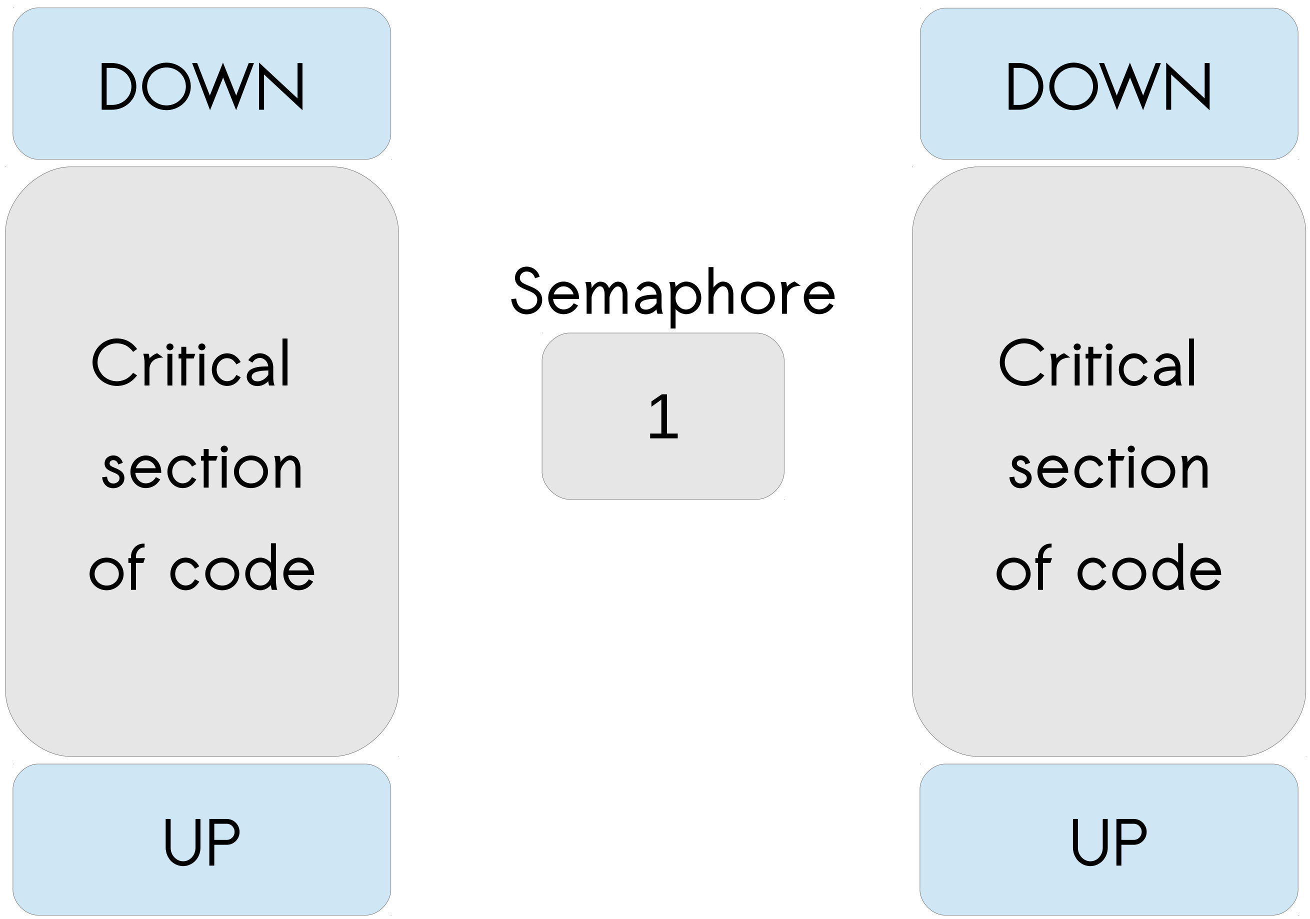
Semaphores in action... Case -1

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



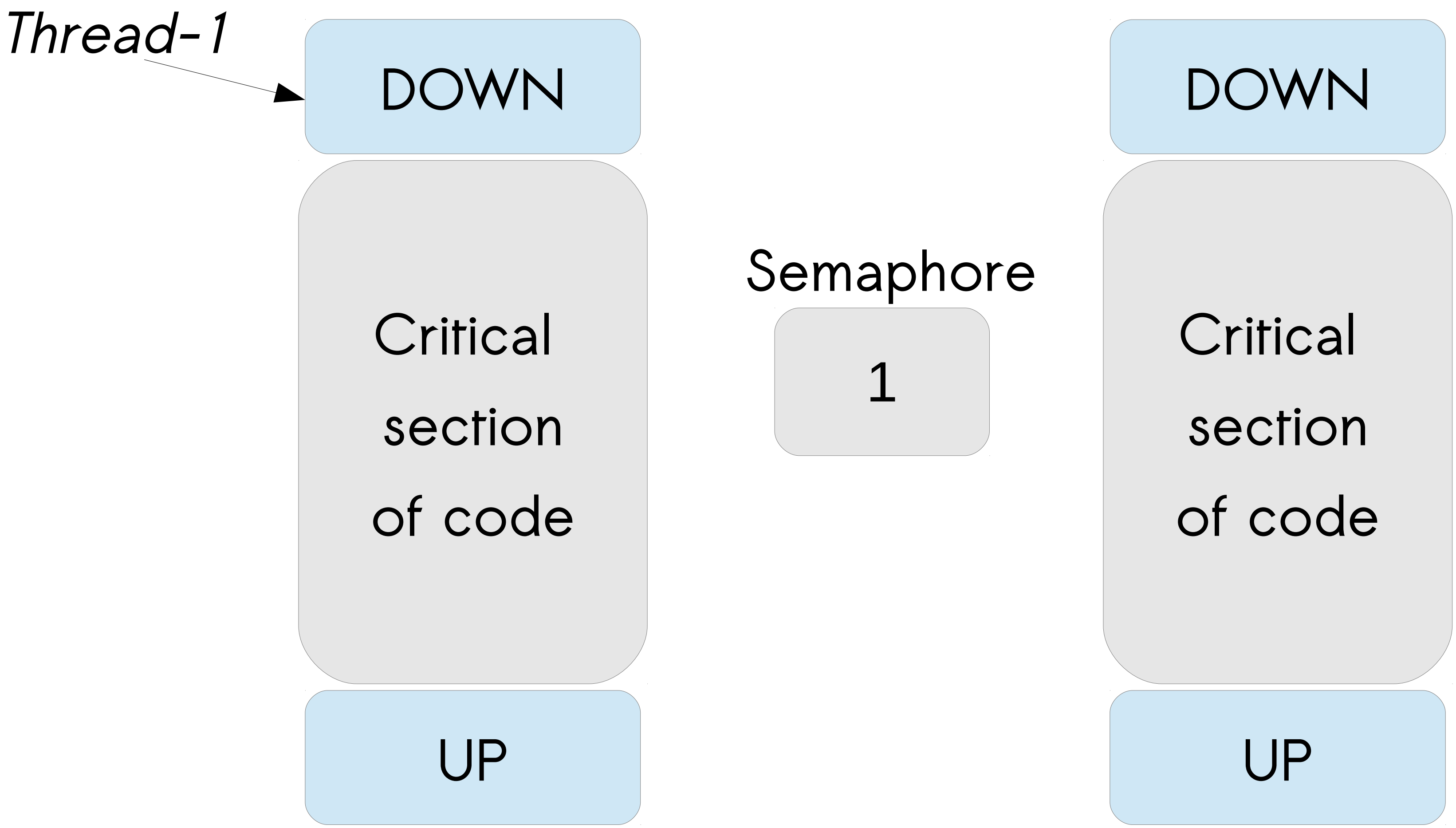
Semaphores in action... Case -2

- Semaphore and critical sections are setup



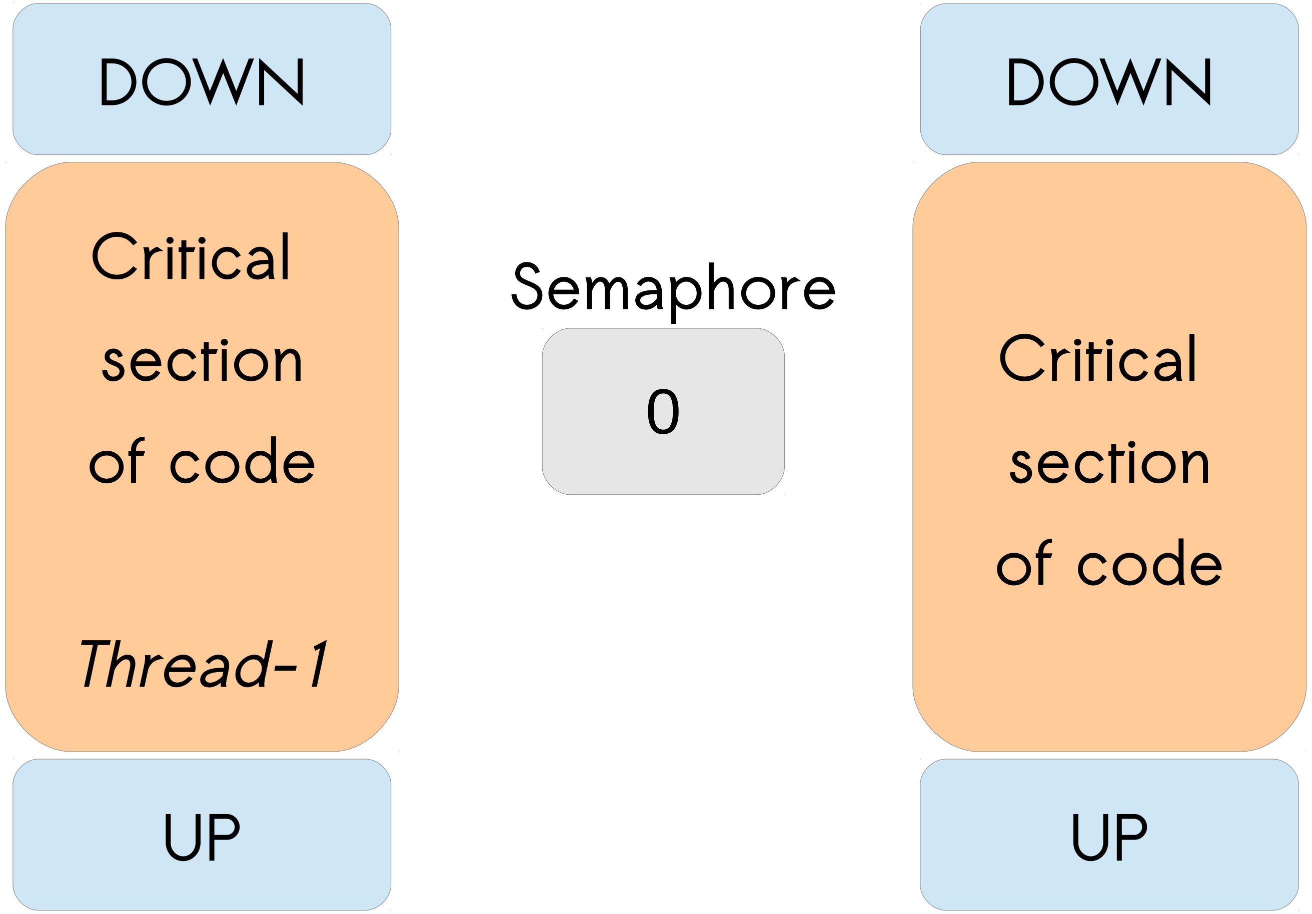
Semaphores in action... Case -2

- Thread-1 tries to acquire the semaphore



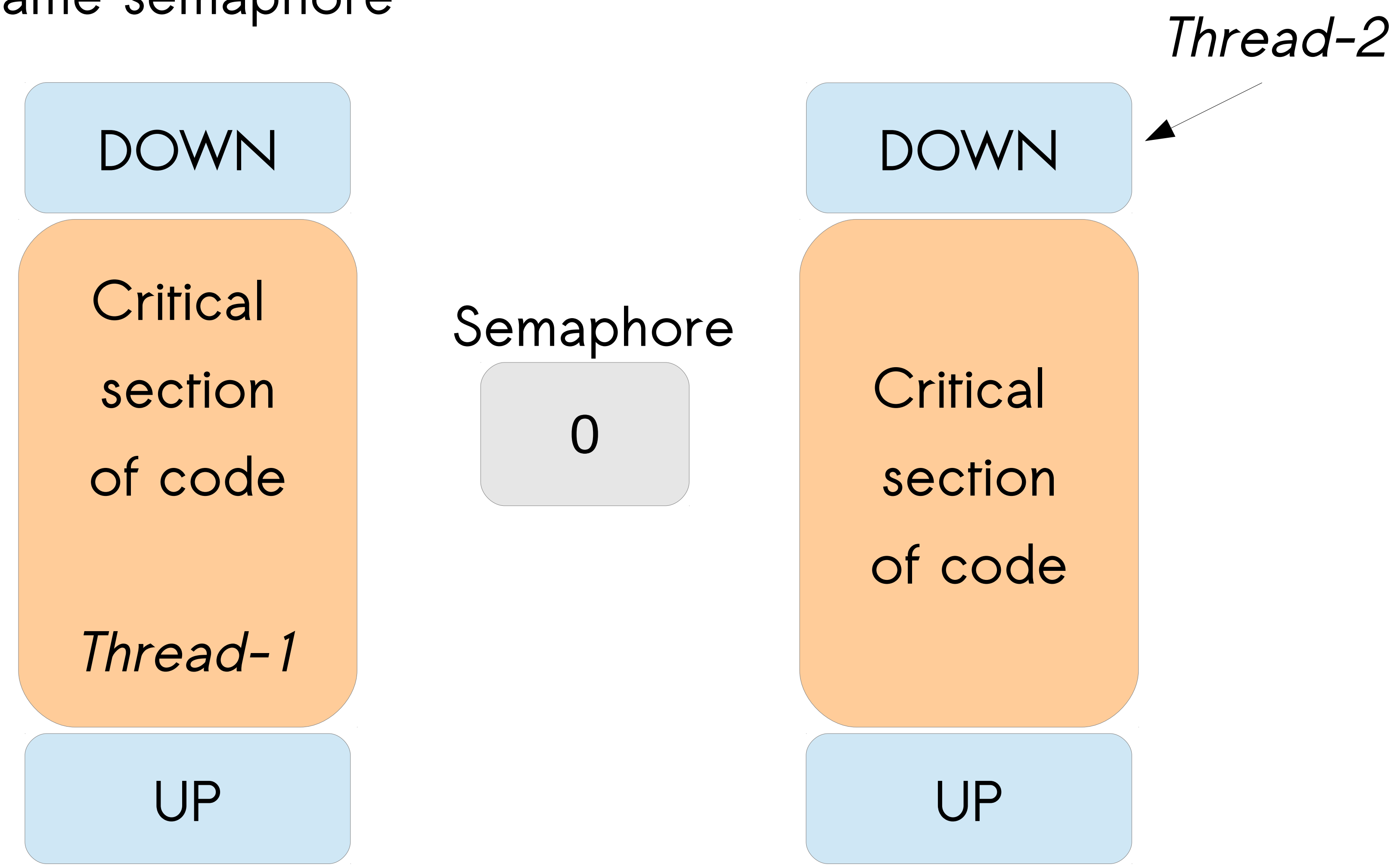
Semaphores in action... Case -2

- Thread-1 is now in its critical section



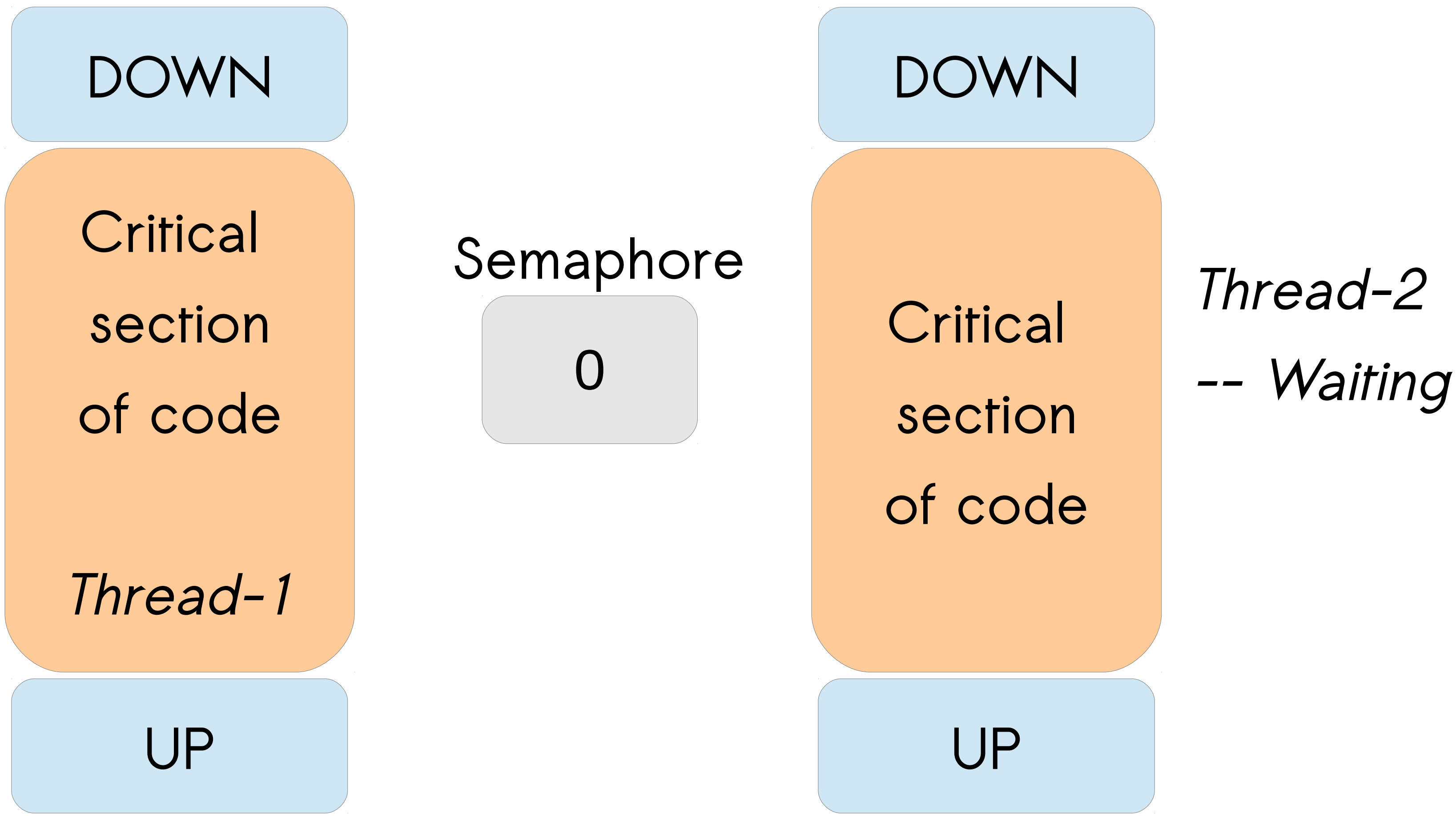
Semaphores in action... Case -2

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



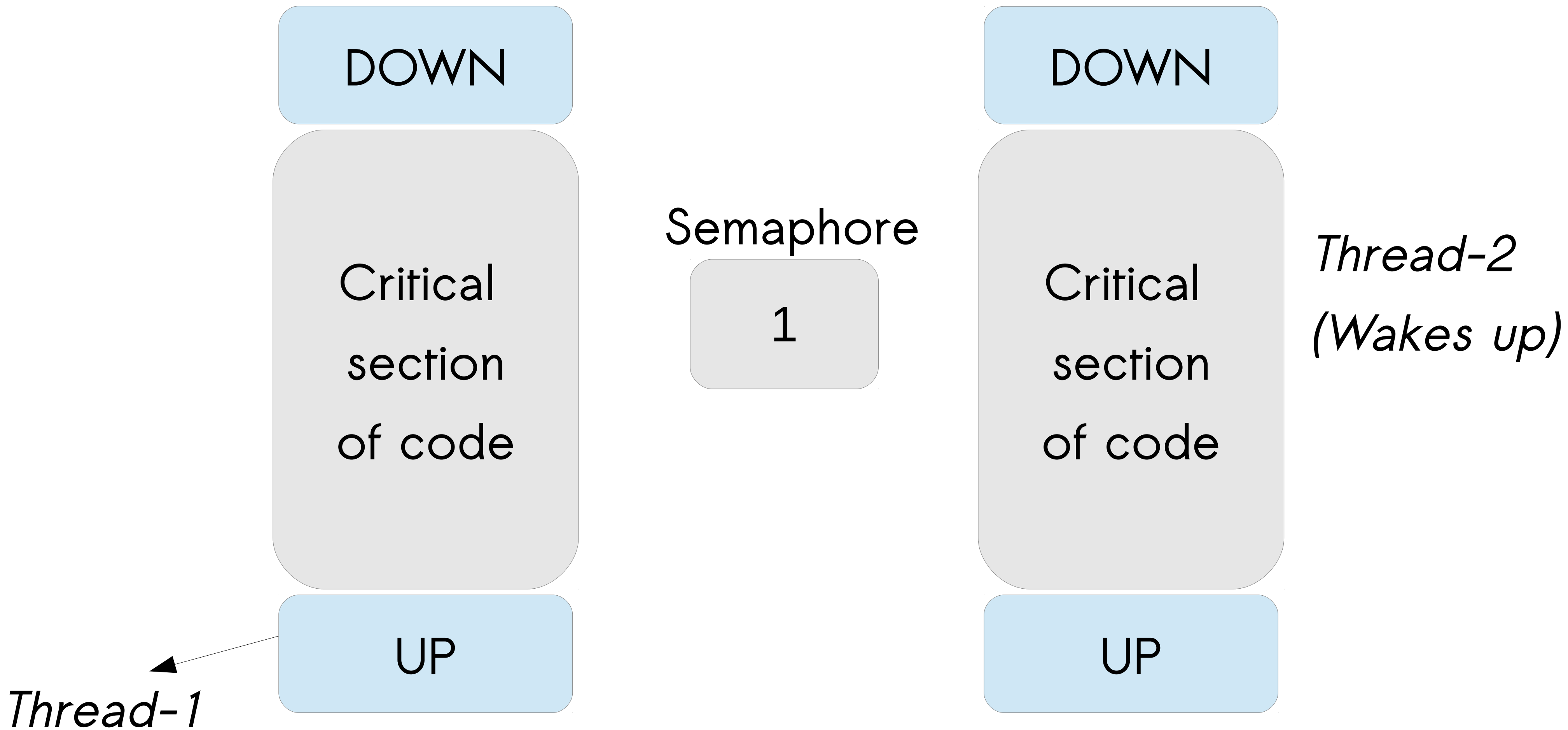
Semaphores in action... Case -2

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



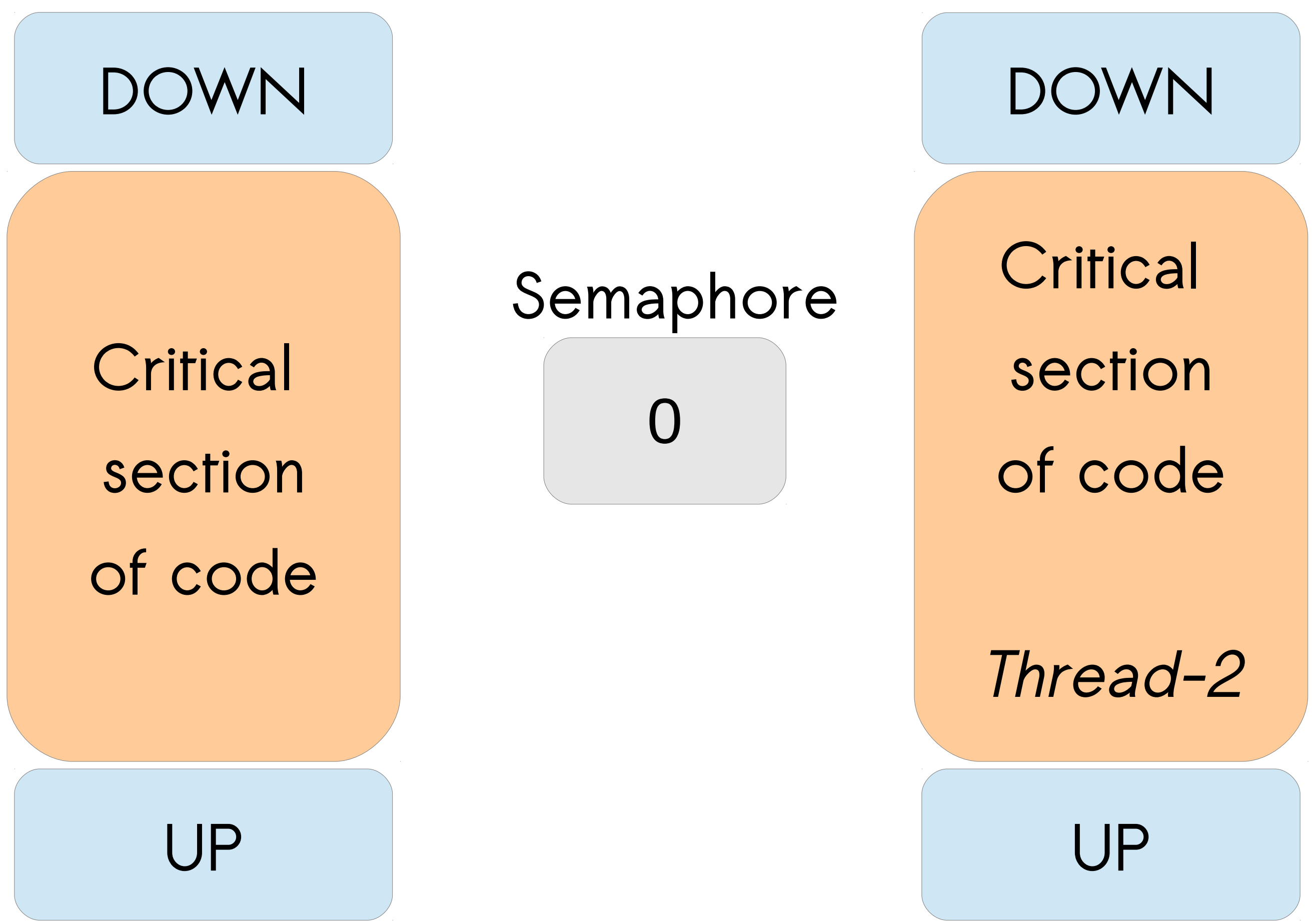
Semaphores in action... Case -2

- 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
- Well suited to locks that are held for a long time.
- Not optimal for locks that are held for short periods because the overhead of sleeping, maintaining the wait queue, and waking back up

Kernel APIs : Initialisations

- <linux/semaphore.h>

  struct semaphore;

- Dynamically : void sema_init(struct semaphore *);
- Statically : DEFINE_SEMAPHORE(name);

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. If the semaphore is not available at the time of call, it returns immediately with a nonzero value
- Release the semaphore : void up(struct semaphore *);

Reader/Writer Semaphores

- It is often possible to allow multiple concurrent readers, as long as nobody is trying to make any changes.
- For handling this kind of situations, the kernel provides the reader/writer semaphores.
- An *rwsem* allows either one writer or an unlimited number of readers to hold the semaphore
- Writers get priority; as soon as a writer tries to enter the critical section, no readers will be allowed in until all writers have completed their work.
- Best suitable for the cases where writers are far less than the readers

Kernel APIs : RWSEM

- <linux/rwsem.h>

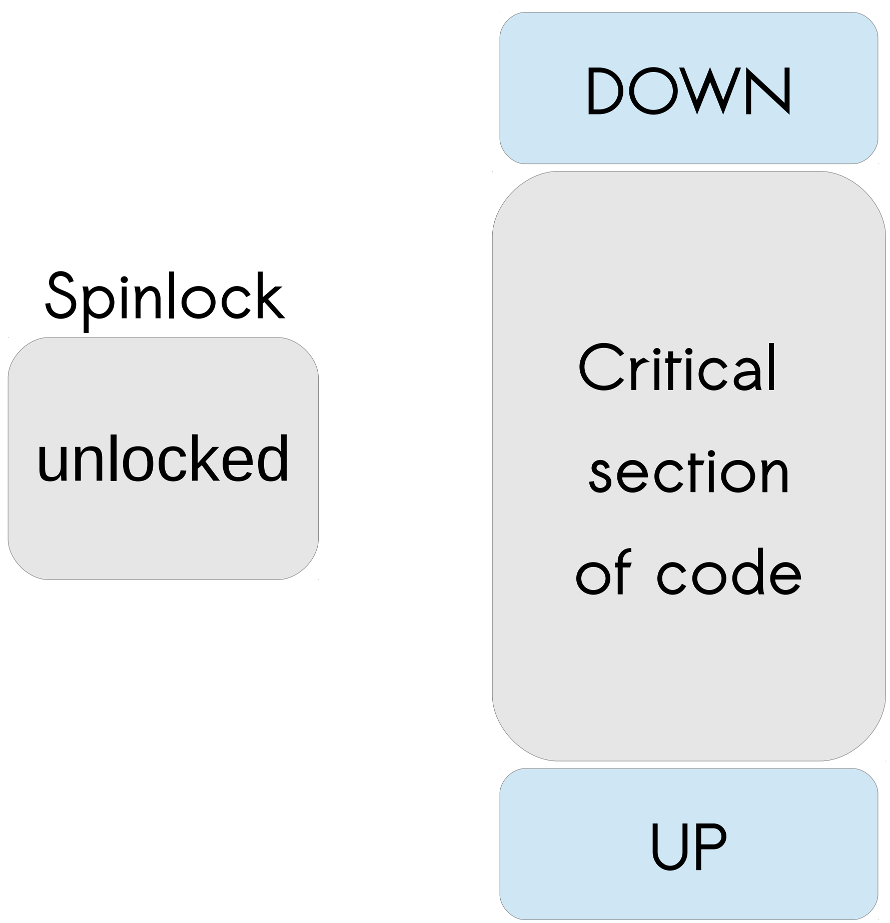
  struct rw_semaphore;

- Initialisation : void init_rwsem(struct rw_semaphore *);
- Reader lock :
  - void down_read(struct rw_semaphore *);
  - int down_read_trylock(struct rw_semaphore *);
  - void up_read(struct rw_semaphore *);
- Writer lock :
  - void down_write(struct rw_semaphore *);
  - int down_write_trylock(struct rw_semaphore *);
  - void up_write(struct rw_semaphore *);

Spinlocks

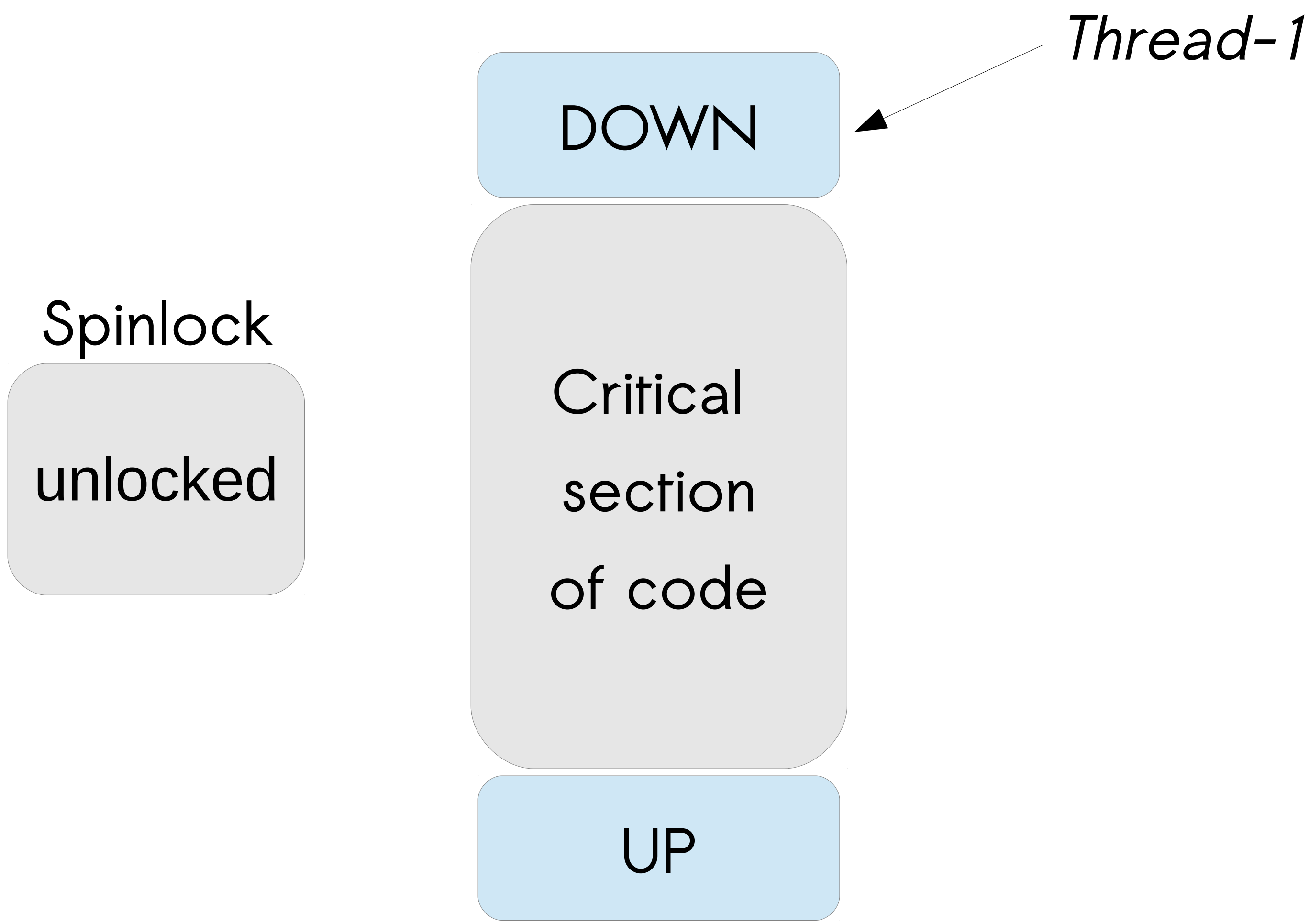
Spinlocks in action...

- Spinlocks and critical sections are setup



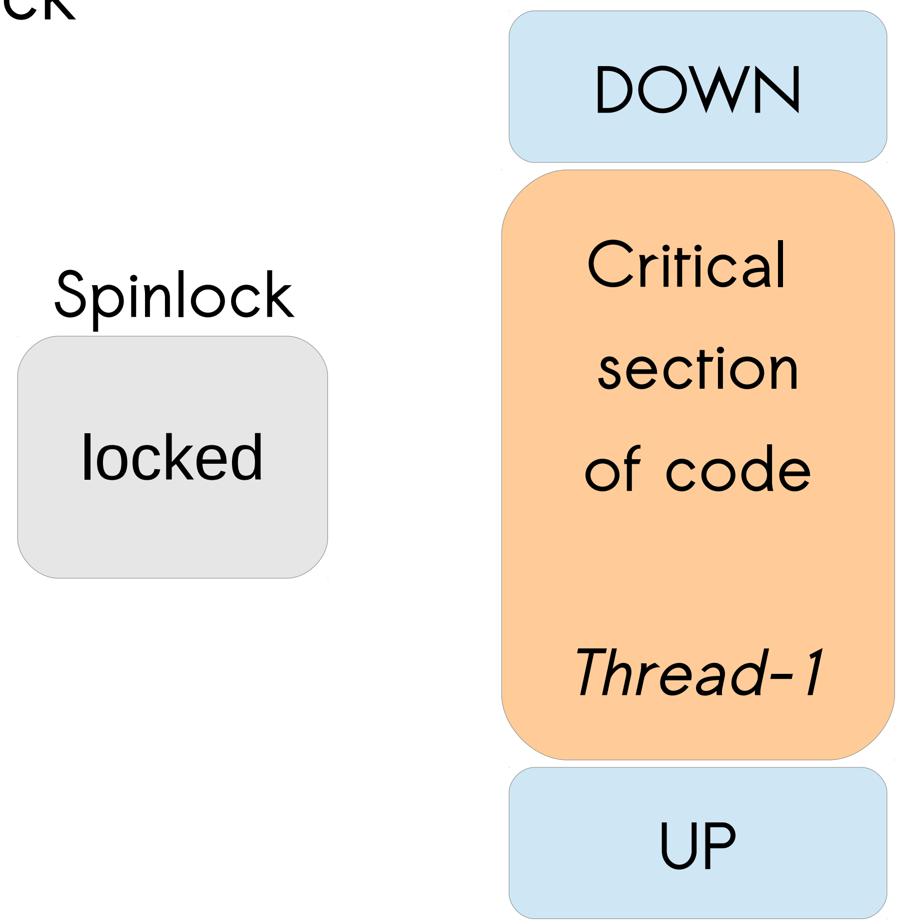
Spinlocks in action...

- Thread-1 tries to acquire the spinlock



Spinlocks in action...

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



Thread-1 :

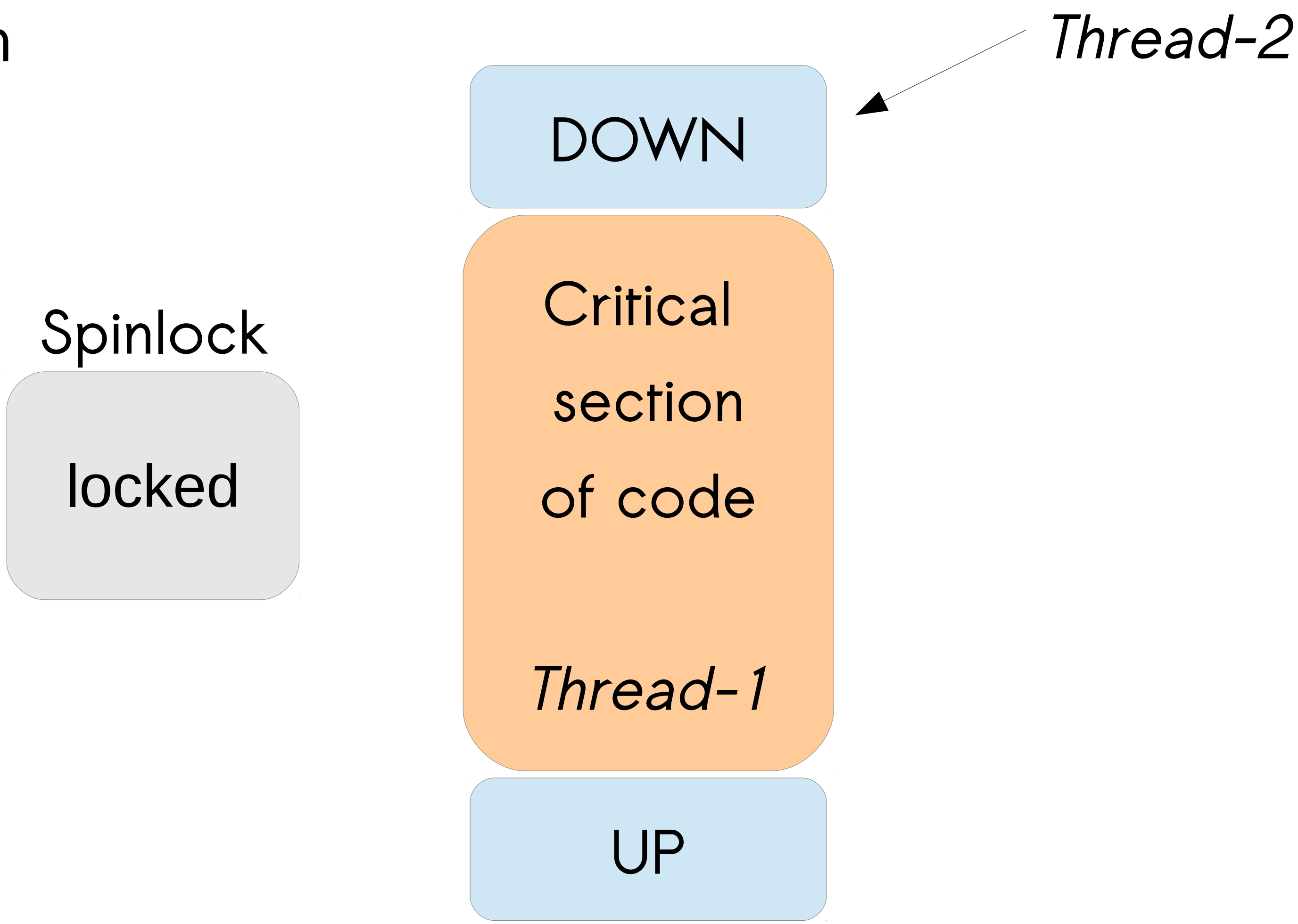
-- Acquired the lock

-- Preemption is disabled

-- Cannot goto sleep

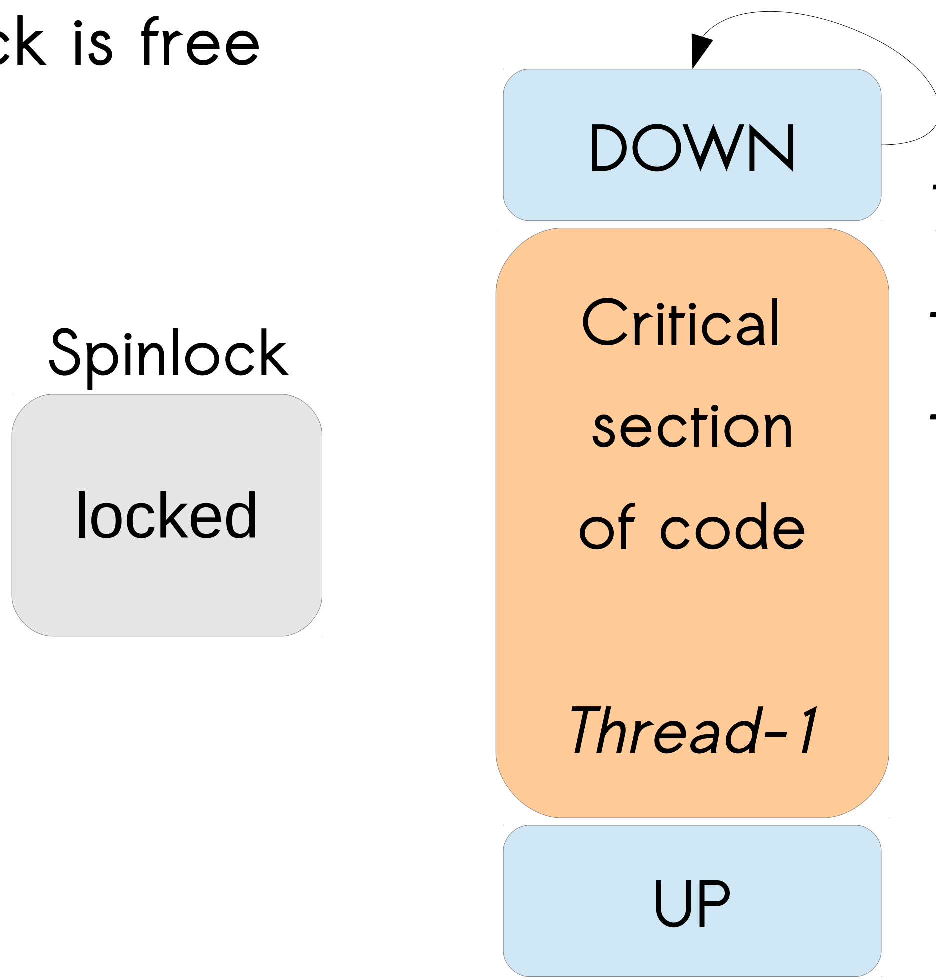
Spinlocks in action...

- 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



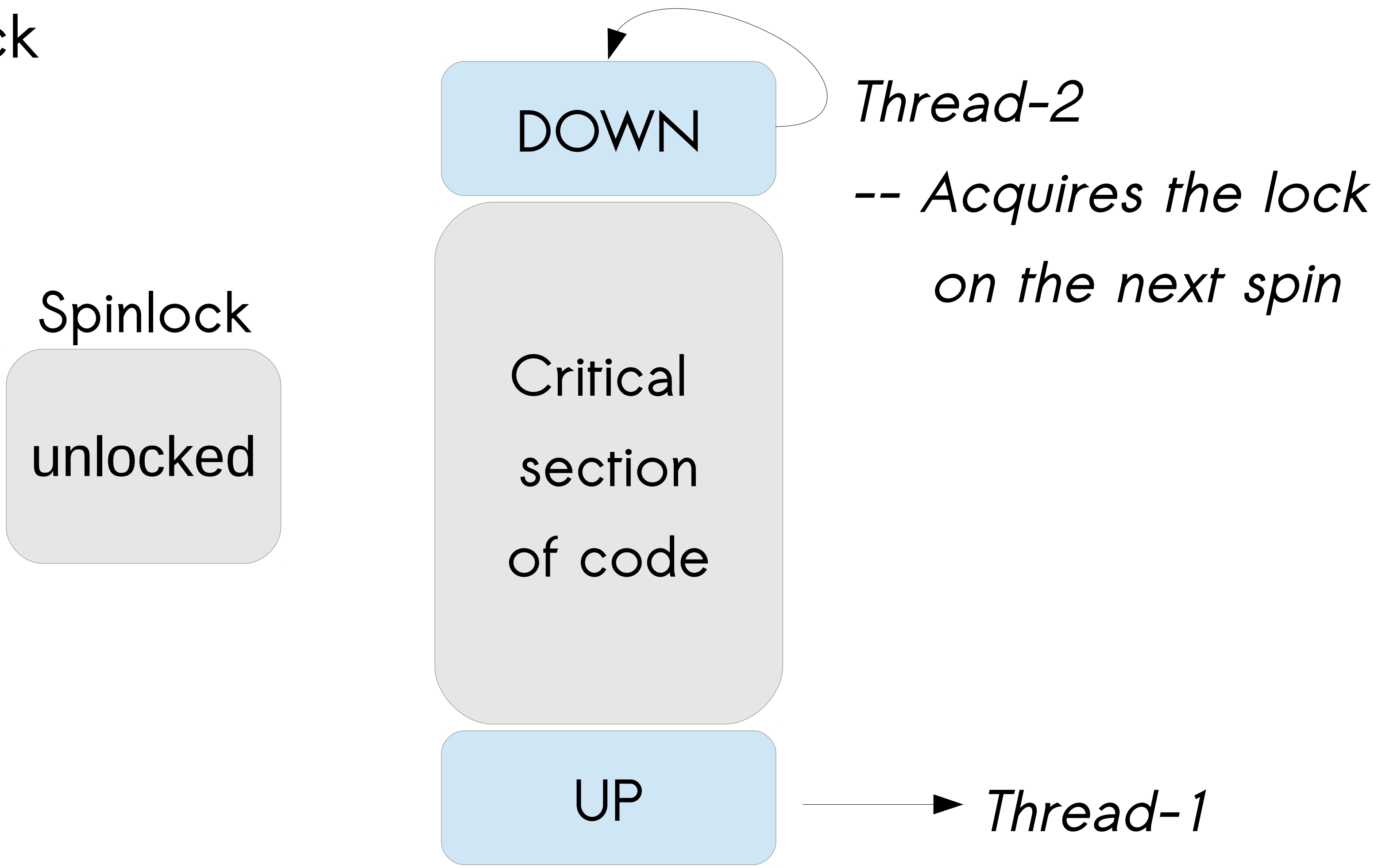
Thread-2 :

-- Busy loops

-- Wastes processor's time

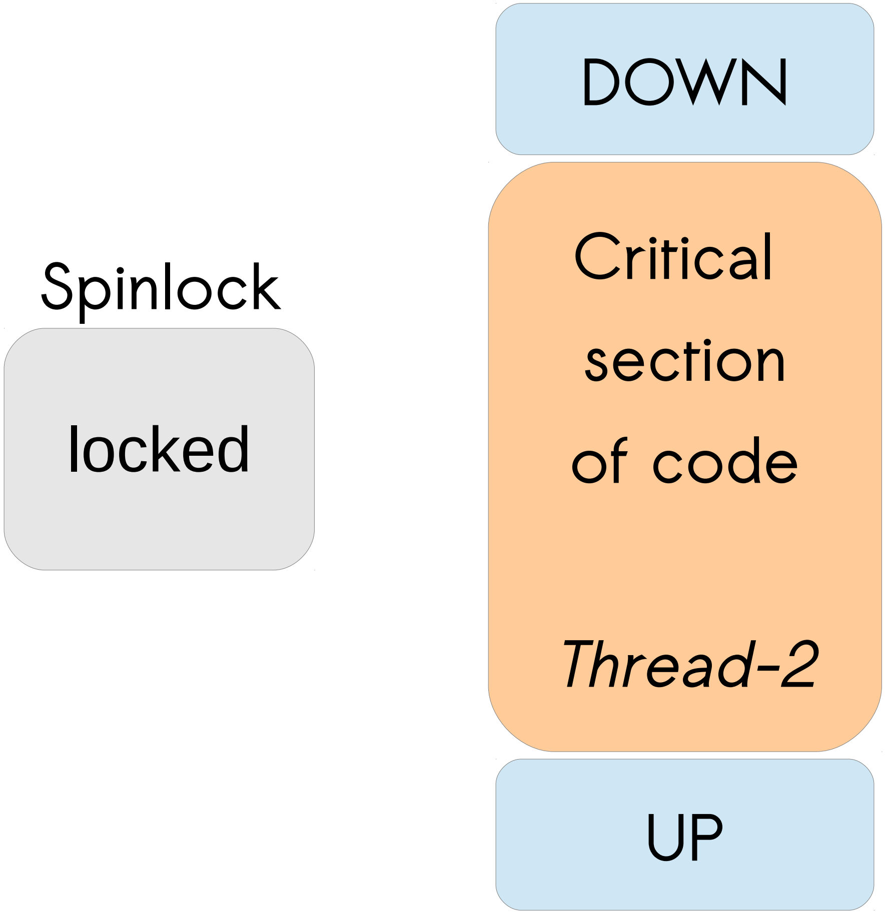
Spinlocks in action...

- 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.

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_irq_bh (spinlock_t *lock);
  - void spin_unlock_irq_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

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 :
  - int wait_for_completion_interruptible(struct completion *);
  - void complete(struct completion *);

Atomic Operations

Atomic Integer Operations

- Sometimes, a shared resource may be a simple integer value.
- In order to protect such variables, we can use *atomic_t* type.
- The following operations may be possible with such type :
  - void atomic_set(atomic_t *v, int i);
  - void atomic_add(int i, atomic_t *v);
  - void atomic_sub(int i, atomic_t *v);
  - void atomic_inc(atomic_t *v);
  - void atomic_dec(atomic_t *v);
- The above operations are implemented directly using assembly code and hence they guarentee that the operation is atomic.

Atomic Integer Operations cont..

- For more such functions, visit <linux/atomic.h>
- Such functions may be atomic as they execute individually. However, if two such functions are called one after the other, then in such cases, locking is the only alternative. Example :

  atomic_sub(amount, &first_atomic);

  /* Some code might interrupt here and messup the value of *amount* */

  atomic_add(amount, &second_atomic);

Atomic Bit operations

- <asm/bitops.h>
- Operations :
  - void set_bit(int nr, unsigned long addr);
  - void clear_bit(int nr, unsigned long addr);
  - void change_bit(int nr, unsigned long addr);
  - int test_bit(int nr, unsigned long addr);
  - int test_and_set_bit(int nr, unsigned long addr);
  - int test_and_clear_bit(int nr, unsigned long addr);
  - int test_and_change_bit(int nr, unsigned long addr);

Sequential Locks

Introduction

- Seq locks are useful to provide a lightweight and scalable lock for use with many readers and a few writers
- It works by maintaining a sequence counter.
- Whenever the data in question is written to, a lock is obtained and a sequence number is incremented
- Prior to and after reading the data, the sequence number is read. If the values are the same, a write did not begin in the middle of the read.
- Cannot be used to protect data structures involving pointers because the reader may be following a pointer that is invalid while the writer may be changing the data structure

Kernel APIs

- Initialisation :
  - <linux/seqlock.h>

    seqlock_t;
  - Statically : DEFINE_SEQLOCK(name);
  - Dynamically : seqlock_init(seqlock_t *);
- Write lock :
  - void write_seqlock(seqlock_t *);
  - void write_sequnlock(seqlock_t *);

Kernel APIs cont..

- The reading part is generally not implemented as locks, but usually we follow this procedure :

```
unsigned long seq;
do {
    seq = read_seq_begin(seqlock_t *);
    /* Read the data here */
} while( read_seqretry(seqlock_t *, seq) );
```

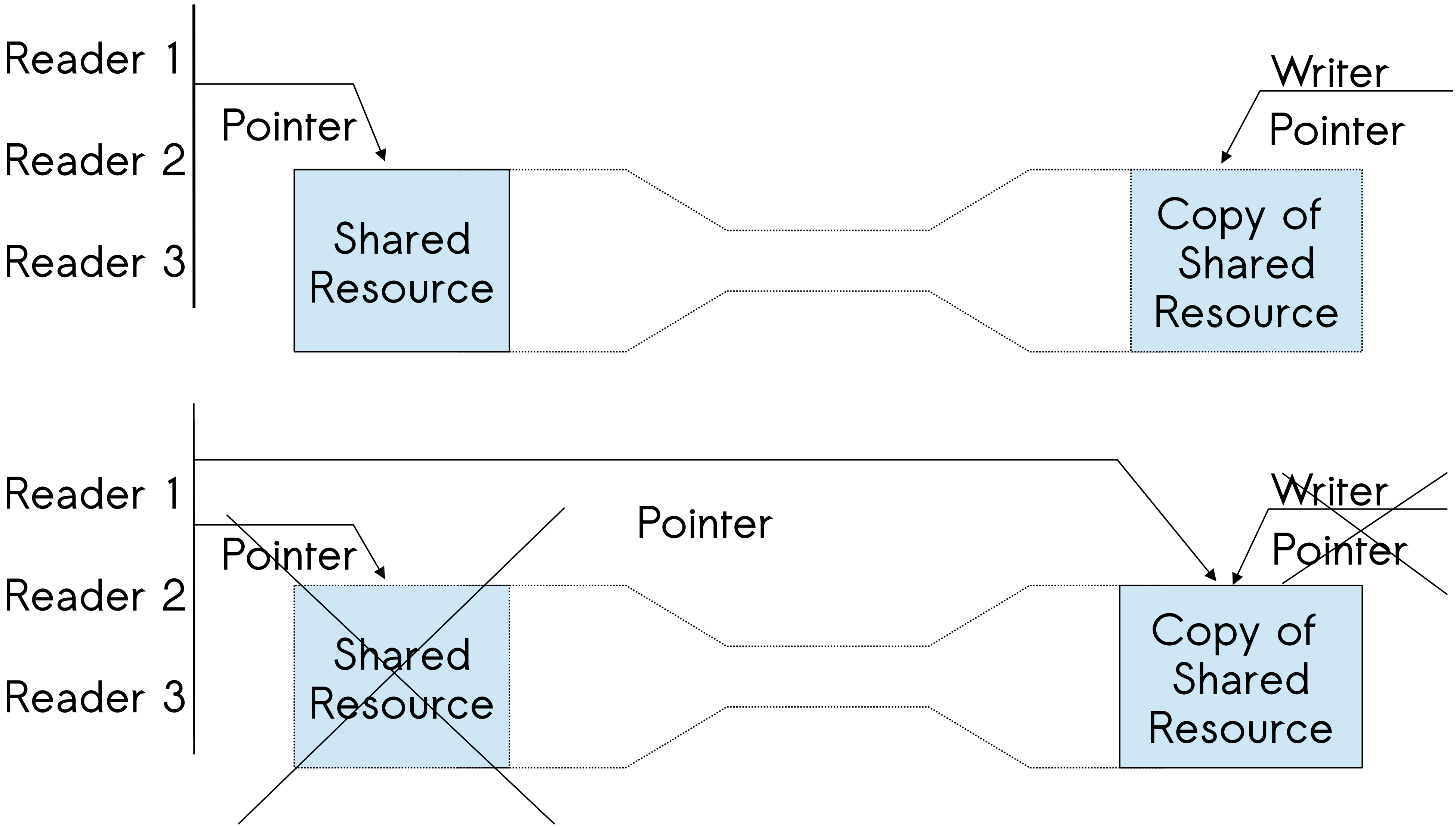
- Read access works by obtaining an (unsigned) integer sequence value on entry into the critical section. On exit, that sequence value is compared with the current value; if there is a mismatch, the read access must be retried.

Read-Copy-Update

Read-Copy-Update

- It is optimized for situations where reads are common and writes are rare.
- The resources being protected should be accessed via pointers.
- When the data structure needs to be changed, the writing thread makes a copy, changes the copy, then aims the relevant pointer at the new version—thus, the name of the algorithm
- When the kernel is sure that no references to the old version remain, it is freed.

Read-Copy-Update



References

- Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman,"Linux Device Drivers",3rd Edition, O'Reilly Publications
- Robert Love, "Linux Kernel Developent", 3rd Edition, Developer's Library

Thank You :)