1. When spinlock is used ?
Ans: In the following situations.

1. The thread that holds the lock is not allowed to sleep.
2. The thread that is waiting for a lock does not sleep, but spins in a tight loop.

When properly used, spinlock can give higher performance than semaphore. Ex: Intrrrupt handler.

2. What are the rules to use spinlocks?
Ans:
Rule - 1: Any code that holds the spinlock, can not relinquish the processor for any reason except to service interrupts (sometimes not even then). So code holding spinlock can not sleep.
Reason: suppose your driver holding spinlock goes to sleep.

Ex: calls function `copy_from_user()` or `copy_to_user()`, or kernel preemption kicks in so higher priority process pushed your code aside. Effectively the process relinquishes the CPU holding spinlock.

Now we do not know when the code will release the lock. If some other thread tries to obtain the same lock, it would spin for very long time. In the worst case it would result in **deedlock.**

Kernel preemption case is handled by the spinlock code itself. Anytime kernel code holds a spinlock, preemption is disabled on the relevant processor. Even uniprocessor system must disable the preemption in this way.

Rule - 2: Disable interrupts on the local CPU, while the spinlock is held.
Reason: Support your driver take a spinlock that control access to the device and then issues an interrupt. This causes the interrupt handler to run. Now the interrupt handler also needs the lock to access the device. If the interrupt handler runs on the same processor, it will start spinning. The driver code also can not run to release the lock. SO the processor will spin for ever.

Rule - 3: Spinlocks must be held for the minimum time possible.
Reason: Long lock hold times also keeps the current processor from scheduling, meaning a higher priority process may have to wait to get the CPU.

So it impacts kernel latency (time a process may have to wait to be scheduled). Typically spinlocks should be held for the time duration, less than that CPU takes to do a contex switch between threads.
Rule -4: if you have semaphores and spinlocks both to be taken. Then take semaphore first and then spinlock

---

```
DEFINE_SPINLOCK(mr_lock);
unsigned long flags;
spin_lock_irqsave(&mr_lock, flags);
/* critical region ... */
spin_unlock_irqrestore(&mr_lock, flags);
```

The routine **spin_lock_irqsave()** saves the current state of interrupts, disables them locally, and then obtains the given lock. Conversely, **spin_unlock_irqrestore()** unlocks the given lock and returns interrupts to their **previous state**

```
The reasons you mustn't use these versions if you have interrupts that
play with the spinlock is that you can get deadlocks:

        spin_lock(&lock);
        ...
                <- interrupt comes in:
```

```
spin_lock(&lock);
```

Top and Bottom Halves

Interrupt handling sometimes needs to perform lengthy tasks.

This problem is resolved by splitting the interrupt handler into two halves:

– Top half responds to the interrupt
• The one registered to request_irq .
• Saves data to device - specific buffer and schedules the bottom half
• Current interrupt disabled, possibly all disabled.
• Runs in interrupt context, not process context. Can't sleep.
• Acknowledges receipt of interrupt.
• Schedules bottom half to run later.

– Bottom half is scheduled by the top half to execute later
• With all interrupts enabled
• Wakes up processes, starts I/O operations, etc.
• Runs in process context with interrupts enabled.
• Performs most work required. Can sleep.
• Ex: copies network data to memory buffers.

Top and Bottom Halves

Three mechanisms may be used to implement bottom halves
– SoftIRQs
• Have strong locking requirements
• Only used of performance sensitive subsystems
–networking, SCSI, etc. *
Reentrant
– Tasklets
• Built on top of SoftIRQs
•Should not sleep
• Cannot run in parallel with itself
• Can run in parallel with other tasklets on SMP systems
• Guaranteed to run on the same CPU that first scheduled them

– Workqueues
• Can sleep
• Cannot copy data to and from user space .

Dos and Don’ts of Interrupt Handlers

-- It's a programming offense if your interrupt context code goes to sleep.
Interrupt handlers cannot relinquish the processor by calling sleepy functions such as schedule_timeout().
-- For protecting critical sections inside interrupt handlers, you can't use mutexes because they may go to sleep. Use spinlocks instead, and use them only if you must.

-- Interrupt handlers are supposed to get out of the way quickly but are expected to get the job done. To circumvent this Catch- 22, interrupt handlers split their work into two halves:
top (slim) and bottom (fat).

-- You do NOT need to design interrupt handlers to be reentrant. When an nterrupt handler s running, the corresponding IRQ is disabled until the handler returns.

-- Interrupt handlers can be interrupted by handlers associated with IRQs that have higher priority. You can prevent this nested interruption by specifically requesting the kernel to treat your interrupt handler as a fast handler.

`static` used as a keyword on file-scope will give the function or object it is used with internal linkage.
That means it lives through the whole program, and dies when the program terminates.

Where objects that have static storage duration are stored isn't specified, but they are usually stored depending on whether they are initialized or not:
- Initialized file-scope variables are usually stored in a section called ".data", while
- Non-initialized file-scope variables are usually stored in a section called ".bss".

Remember that if the variable isn't initialized, it will be zero initialized at the start of the program: The ".bss" section is usually zero initialized by an implementation on program's startup.