Synchronization becomes necessary when multiple threads works on shared memory to manipulate the data. To do so kernel provides various methods to achieve synchronization which is given in the below list:

1. [Spin locks](https://notes.eddyerburgh.me/operating-systems/linux/kernel-synchronization#spin-locks)
2. [Semaphores](https://notes.eddyerburgh.me/operating-systems/linux/kernel-synchronization#semaphores)
3. [Mutexes](https://notes.eddyerburgh.me/operating-systems/linux/kernel-synchronization#mutexes)
4. [Sequential locks](https://notes.eddyerburgh.me/operating-systems/linux/kernel-synchronization#sequential-locks)
5. [Barriers](https://notes.eddyerburgh.me/operating-systems/linux/kernel-synchronization#barriers)

Let's look into the various methods given above one by one.

Spin locks:

A spin lock is a lock that can be held by at most one thread of execution. If a thread of execution attempts to acquire a spin lock while it is already held, which is called contended, the thread busy loops spins waiting for the lock to become available. If the lock is not contended, the thread can immediately acquire the lock and continue. The spinning prevents more than one thread of execution from entering the critical region at any one time. The same lock can be used in multiple locations, so all access to a given data structure, for example, can be protected and synchronized. Spin locks are architecture-dependent and implemented in assembly. The architecture dependent code is defined in <asm/spinlock.h> .The actual usable interfaces are defined in <linux/spinlock.h> . The lock can be held simultaneously by at most only one thread of execution. Consequently, only one thread is allowed in the critical region at a time. This provides the needed protection from concurrency on multiprocessing machines. On uniprocessor machines, the locks compile away and do not exist; they simply act as markers to disable and enable kernel preemption. If kernel preempt is turned off, the locks compile away entirely. One has to be careful while using spin locks and the reason for that is, they are not recursive. Unlike spin lock implementations in other operating systems and threading libraries, the Linux kernel’s spin locks are not recursive. This means that if you attempt to acquire a lock you already hold, you will spin, waiting for yourself to release the lock. But because you are busy spinning, you will never release the lock and you will deadlock.

Semaphores:

Semaphores in Linux are sleeping locks. When a task attempts to acquire a semaphore that is unavailable, the semaphore places the task onto a wait queue and puts the task to sleep. The processor is then free to execute other code. When the semaphore becomes available, one of the tasks on the wait queue is awakened so that it can then acquire the Semaphore. Because the contending tasks sleep while waiting for the lock to become available, semaphores are well suited to locks that are held for a long time. Conversely, semaphores are not optimal for locks that are held for short periods because the overhead of sleeping, maintaining the wait queue, and waking back up can easily outweigh the total lock hold time. Because a thread of execution sleeps on lock contention, semaphores must be obtained only in process context because interrupt context is not schedulable. You can (although you might not want to) sleep while holding a semaphore because you will not deadlock when another process acquires the same semaphore. (It will just go to sleep and eventually let you continue.) You cannot hold a spin lock while you acquire a semaphore, because you might have to sleep while waiting for the semaphore, and you cannot sleep while holding a spin lock.

Mutexes:

A mutex is a sleeping lock that enforces mutual exclusion. Only one task can hold the mutex at a time. That is, the usage count on a mutex is always one. Whoever locked a mutex must unlock it. That is, you cannot lock a mutex in one context and then unlock it in another. This means that the mutex isn’t suitable for more complicated synchronizations between kernel and user-space. Most use cases, however, cleanly lock and unlock from the same context. Recursive locks and unlocks are not allowed. That is, you cannot recursively acquire the same mutex, and you cannot unlock an unlocked mutex. A process cannot exit while holding a mutex. A mutex cannot be acquired by an interrupt handler or bottom half, even with mutex\_trylock() . A mutex can be managed only via the official API: It must be initialized via the methods described in this section and cannot be copied, hand initialized, or reinitialized. Mutexes and semaphores are similar. Having both in the kernel is confusing. Thankfully, the formula dictating which to use is quite simple: Unless one of mutex’s additional constraints prevent you from using them, prefer the new mutex type to semaphores. When writing new code, only specific, often low-level, uses need a semaphore. Start with a mutex and move to a semaphore only if you run into one of their constraints and have no other alternative.

Sequential locks:

Sequential locks (seq locks) work by maintaining a sequence counter. When shared data is written to, a sequential lock is acquired and the counter is incremented. Readers will check the sequence number before and after reading. If the values are equal then a write did not occur, if the value is even then a write has succeeded. The sequential lock, generally shortened to seq lock, is a newer type of lock introduced in the 2.6 kernel. It provides a simple mechanism for reading and writing shared data. 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. Further, if the values are even, a write is not underway. (Grabbing the write lock makes the value odd, whereas releasing it makes it even because the lock starts at zero.) Seq locks are ideal when your locking needs meet most or all these requirements, your data has a lot of readers. Your data has few writers. Although few in number, you want to favor writers over readers and never allow readers to starve writers. Your data is simple, such as a simple structure or even a single integer that, for whatever reason, cannot be made atomic.

Barriers:

Barriers are a way to ensure that load and read instructions aren’t reordered by either the compiler or the processor. When dealing with synchronization between multiple processors or with hardware devices, it is sometimes a requirement that memory-reads (loads) and memory-writes (stores) issue in the order specified in your program code. When talking with hardware, you often need to ensure that a given read occurs before another read or write.Additionally, on symmetrical multiprocessing systems, it might be important for writes to appear in the order that your code issues them (usually to ensure subsequent reads see the data in the same order). Complicating these issues is the fact that both the compiler and the processor can reorder reads and writes 4 for performance reasons. Thankfully, all processors that do reorder reads or writes provide machine instructions to enforce ordering requirements. It is also possible to instruct the compiler not to reorder instructions around a given point. These instructions are called barriers. The rmb() method provides a read memory barrier. It ensures that no loads are reordered across the rmb() call. That is, no loads prior to the call will be reordered to after the call, and no loads after the call will be reordered to before the call. The wmb() method provides a write barrier. It functions in the same manner as rmb() , but with respect to stores instead of loads—it ensures no stores are reordered across the barrier. The mb()  call provides both a read barrier and a write barrier. No loads or stores will be reordered across a call to mb() . It is provided because a single instruction (often the same instruction used by rmb() ) can provide both the load and store barrier.

**References:**

Robert Love, Linux Kernel Development (3rd Edition), Addison-Wesley Professional, 2010