# Kernel Synchronization

Two main aspects of OS synchronization design are, *(i)* provide synchronization constructs for the user space processes and threads, *(ii)* meet the synchronization requirements of the OS software itself. In this course, we will deal with the latter aspect. However, user space synchronization is an interesting design question: what kind of kernel support (if at all any) required for user-space synchronization? System call APIs for user-space synchronization support is provided by `futex` construct in Linux kernel (See man pages).

From the synchronization perspective of kernel code itself, following questions arise.

1. When and why locking is necessary?
2. Is it required in a uni-processor system?
3. How kernel synchronization different than user-space synchronization?

Resources can be shared across multiple contexts like process context (in system call handlers), interrupt or bottom half contexts and kernel threads. In such scenarios, synchronization requirements can be different. Let us analyze the different setups and scenarios one by one.

### Uni-processor synchronization

- A system with single CPU can execute one context at a time. So, why use locks? Consider a scenario when P1 executing a system call S1 in kernel context gets switched by another process P2 who is also executing in kernel context and access shared state used in the implementation of S1.
- Can we avoid locks? If we disable preemption when executing system calls (a.k.a. non-preemptable kernels), the context switch is avoided.
- What about interrupts arising during the system call execution? If there is a shared state between the system call handler and interrupt handler, should we use locks? Note that, interrupt handlers can not be switched out by process contexts. As a result _____ situation could arise. What should be the solution? Oops ... we have to disable interrupts
- What if a shared state is accessed by two different interrupt handlers? Locking would not help... interrupt disabling is required.

To summarize, depending on the entities involved in access of shared state, a synchronization technique should be used. It is advisable to use the lowest possible level of serialization for better responsiveness of the system. For example, disabling interrupts implicitly disables preemption, but results in possible delay in interrupt processing and should be avoided if possible.

### Multi-processor synchronization

Synchronization requirements in a multi-processor system becomes tricky.

- With shared state between process contexts (syscall handlers), can preemption disabling work? Obviously not, because another process can execute on another processor potentially accessing a shared resource. Using locks becomes inevitable.
- What about shared state between process context and an interrupt context? Does interrupt disabling solve the problem? Why or why not? A processor can disable interrupts on the local processor which means interrupts can be serviced on other processors. So, the solution is: locking + local interrupt disabling

- Similarly, protecting shared state between multiple interrupt handlers requires combining interrupt disabling with locking.

There is a basic bottleneck in accessing the synchronization construct from different processors. The cache coherency overheads cause performance degradation. From the OS design point of view, minimizing lock contention from different processors is always desirable (but may not be always possible). An example alternate design is per-CPU data structures where every CPU has the local copy and the OS takes care of the consistency requirements across the different copies.

## Synchronization techniques

There could be many ways to approach the problem of meeting synchronization requirement on shared resources.

1. Every execution entity checks for availability of an entry token in a *tight loop* till the time of acquiring the token. Example: spinlocks, rwlocks etc.

2. As a modification to the tight loop behavior, an execution entity may sleep if the resource is busy and woken up latter when the resource is free. Example: mutex, semaphore etc.

3. What about not waiting for the 'entry token' at all? Example: RCU, RLU and transactional memory.

While using different type of locks, one should bear in mind the tradeoffs of using different types of locks. Possible tradeoffs: lock acquisition overhead vs. lock acquisition latency, lock granularity vs. code complexity and CPU cycle wastage due to lock wait vs. context switch overhead. Most operating systems (including Linux) use architecture support as a building block to build synchronization constructs. However, in theory it is possible to implement locks without assuming any architecture support. (Refer standard OS books: Galvin, Tanenbom ...).

## Spinlocks

1. An execution context "spins" around the lock till it grabs a hold of it.

2. Which of the synchronization requirements are not met by spinlocks?

3. Useful when lock to be held for short durations, why?

A wrong implementation!

```
lock (spinlock sl)
{
  1.  while(sl != 0);
  2.  sl = 1;
}

unlock(spinlock sl)
{
    sl = 0;
}
```

Line #1 and #2 in `lock()` procedure can be interleaved, resulting in multiple entities holding the lock. One way to avoid interleaving is to execute the two operations (checking for the lock availability and setting the lock value) in an atomic manner. An atomic operation is an *uninterruptible and indivisible* operation. Should be careful about atomicity while writing code which is executed simultaneously (a.k.a. critical section) by more than one execution contexts. If a critical section consists of many instructions, atomicity can not be guaranteed because an interrupt might leave the machine in an intermediate state. Similarly, an operation which can be further divided into many sub-operations while a state resulting due to a sub-operation is visible from another CPU can be a non-atomic operation. For example, an atomic increment of a memory location should ensure that other CPUs either access the memory location before increment operation or after completion of the increment operation. A critical section consisting of

1. Multiple C instructions → non-atomic
2. Single C statement → if multiple assembly instructions → non-atomic
3. Single assembly instruction → can be non-atomic!

What kind of single instructions can be non-atomic? Answer: Read-Modify-Write instructions leave a memory state inconsistent with the processor state and could lead to consistency issues. Most arithmetic operations are non-atomic. For example, `inc (Mem)` is non-atomic unless other processors are denied reading the memory variable during the instruction execution. Prefixing an instruction with `lock` assembly prefix prohibits other processor accessing memory at the same time (memory bus is locked). An example implementation of atomic increment is shown below.

```
atomic_inc( atomic_t *var)
{
   lock; inc (var)
}
```

A plethora of atomic operations are supported by Linux. `atomic_add, atomic_sub, atomic_inc_and_test` etc. are some examples of atomic API of Linux.
**Homework:** Refer Linux kernel source to understand implementation of various atomic operations.

Coming back to our wrong spinlock implementation, if line#1 and #2 can be atomically executed, it can help the cause. One of the basic hardware instruction used to implement locks is `cmpxchng or cmpandswap`. A `cmpxchng` instruction compares the content of a register with the memory content and swap them if they are equal. The logic of `cmpxchng` instruction is shown below.
**cmpxchng**

```
cmpxchng destination [Mem/Reg] source[Reg]
implicit registers :- eax and eflags

if eax == val(destination)
then
      zeroflag = 1
      destination = source
else
      zeroflag = 0
      eax = destination
```

The destination operand is compared with `eax` register and if equal, the content of the destination is copied onto the source operand. Additionally, the zero flag bit of `eflags` register is set. Otherwise, the content of destination is copied onto the `eax` register. Can we rewrite the spinlock code using cmpxchng instruction to implement a spinlock? Note that, the `cmpxchng` instruction is non-atomic and should be prefixed by a `lock` prefix to ensure atomicity.
A spinlock can be easily implemented using this basic construct as shown below.

```
lock (spinlock *sl)
{
   1: mov ecx, 1

   2: xor eax, eax
      lock; cmpxchng (sl), ecx
      jnz 2b
      ret
}

unlock (spinlock *sl)
{
    mov (sl) $0
}
```

There are other instructions like `xchng` which can be used to implement spinlocks.
**Homework:** Understand the syntax of `xchng` instruction and implement a spinlock using this instruction.
Notes

1. Read and write operations require same type of locking
2. Context holding the spinlock should not (can not?) sleep
3. A context holding spinlock should not ideally be switched, Linux does not employ any techniques for *deadlock avoidance*, it is up to the programmers like us!
4. So, many a times a spinlock is accompanied by *disabling interrupts*

### Spinlock with interrupt disabling

Many a times, the context intending to hold the lock would require disabling the interrupts at the same time. Linux provides APIs like `spinlock_irqsave` and `spinlock_irqrestore` to achieve the above objective. Sometimes, bottom-halves require disabling instead of interrupts. Similar APIs are provided for the bottom halves.

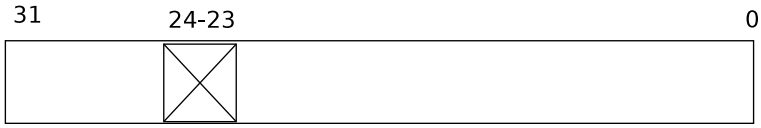
```
spinlock_irqsave (spinlock *sl)
{
    local_irq_disable();
    spin_lock(sl);
}

spinlock_irqrestore (spinlock *sl)
{
    spin_unlock(sl);
    local_irq_enable();
}
```

The ordering of interrupt disabling and lock acquire/release matters.

## Read-write locks

1. Multiple readers are allowed but only one writer allowed.
2. Useful when a significant percentage of access does not modify any shared data.
3. Some form of reader-count need to be maintained.




32-bit integer based R/W lock implementation

```
Value   0x01000000      -->       unlocked
        0x00000000      -->       locked for writing
        0x00FFFFFF      -->       one reader
        0x00FFFFFE      -->       two readers
                                   .....
```

Logic: A reader wants to acquire lock decrements lock value. If the lock value is negative implies ____ or ____. Otherwise, read-lock is granted. A write lock is granted only when the lock value is 0x01000000. So, how many readers can hold the lock simultaneously?

So, Is everything great about read-write locks? What could be the issues? The readers can dominate the lock acquisition causing delays in write operations. Additionally, it requires more operations to implement this locking mechanism.

## Read-Copy-Update (RCU)

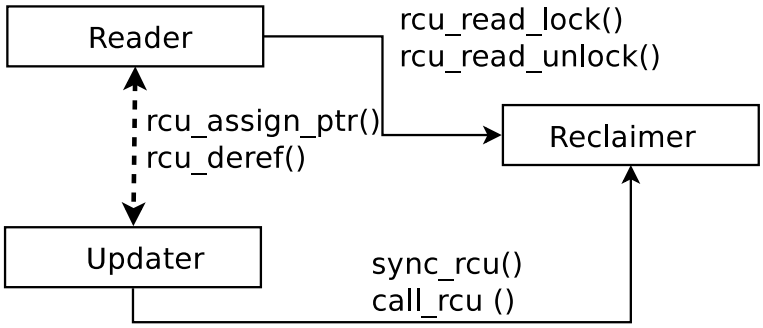



Figure 1: Logical view of RCU operations.

- A reader can access to the shared object without holding any locks albeit with the condition that it will disable preemption and will not sleep.
- A writer creates a separate copy on which update is performed. After updation, it updates the old memory location pointer using an atomic operation.
- So what happens to the old copy? When should it be freed? Only after all references taken by the readers are released.
- What is the maximum number of readers that would hold reference to the old value?

## Semaphores

- Semaphores are waiting locks. When an execution context wants to acquire a lock held by another execution entity, it puts itself into a *sleep-wait* state.
- On a lock release, one of the waiting entity for the lock is woken up.
- In Linux, wait queues are extensively used to implement semaphores.
- In practice linux kernel extensively use the following special semaphore realizations: mutexes or binary semaphore and read-write semaphores.

An example semaphore data structure (similar to one used in the kernel) is shown here.

```
struct semaphore{
                    spinlock semlock;
                    int count;
                    struct list_head waitq;
}
```

For a mutex implementation, the count variable (`count`) is initialized to one. For a generic counting semaphore, it can be set to a value which indicates maximum number of execution contexts allowed in a critical section. A positive value of `count` implies a free lock while any other value indicates that lock is being used. There are two main operations on semaphore—`down()` (lock) and `up()` (unlock). Note that `down()` and `up()` operations require atomicity as they can be executed by multiple execution entities in a concurrent manner. A spinlock (`semlock`) is used to ensure atomicity of the semaphore operations. Further, a wait queue is associated with every semaphore to account for execution entities waiting (sleeping) for the semaphore. Both `down()` and `up()` procedures acquire the `semlock` and simultenously disable interrupts while manipulating the elements in the semaphore structure.

`down()`: If `count` value is greater than zero, the semaphore is granted after decrementing the count value. Otherwise, the context self-sleeps—adds itself to `waitq`, releases `semlock` and relinquishes the CPU. When the context is woken up (may be by a signal or genuinely by release of the semaphore), first it acquires the semaphore lock (`semlock`), checks if it is woken up by semaphore release. If the context is woken up by a semaphore release, semaphore is granted to this execution context, otherwise it re-enters the self-sleep mode.

`up()`: After acquiring `semlock`, the releasing context checks if the wait queue (`waitq`) is empty. In this case, simply the `count` variable is incremented before returning from the function. Otherwise, the first process in the wait queue is woken up before finishing the `up()` routine by releasing the `semlock`.