

The synchronization/locking in the Linux kernel on x86 architecture.

(Notes on resource synchronization mechanisms in the Linux kernel, in a concurrent processing environment.)

Peter Chacko

peter@netdiox.com

NetDiox computing systems

(Linux kernel research/training center)

Jayanagar 4th T Block, Bangalore, India

www.netdiox.com

Key words: semaphore, spin lock, RCU, Linux, kernel, SMP, CMP, mutex, multi-core, synchronization, dead lock, race conditions, interrupts, tasklets, softirqs

Table of Contents

1. About this document-----	3
2. The need for concurrency control in a distributed system.-----	3
3. Atomic functions and the memory barriers -----	4
4. hardware multi threading , Chip level multiprocessing and SMP platforms-----	5
5. Locking introduction - Asynchronous events that affect execution flow-----	6
6. Volatile variables, SMP/CMP/SMT platforms and locks-----	6
7. Various locking primitives.-----	7
7.1 semaphores & mutexes-----	7
7.2 spin locks. -----	7
7.3 Reader-Write locks & seqLocks-----	8
7.4 RCU – new highly optimized locks-----	8
7.5 Big kernel lock(BKL)-----	9
7.6 condition variables-----	9
8. sharing your process context data with interrupts/softirqs/tasklets/timers -----	9
9. dead-locks, live locks-----	10
10. Lockless coding – pragmatic approach -----	11
11. Appendix --Kernel APIs for locking (please read kernel documentation for more details)-	12
12. Conclusion.-----	15

1) About this document

This is primarily meant for those who wish to master Linux kernel synchronization, and to have a solid understanding on the various forms of locking facilities in the kernel. This is not a quick tutorial for people who want to familiarize themselves with the APIs and how to use it for locking, but is designed to provide a comprehensive perspective of the fundamental concepts and is to motivate the reader to take a fresh approach with locking. At the end of the document there is a partial list of locking APIs, but a motivated reader is advised to refer other sources for detailed information on how to use various kernel primitives, as the kernel interfaces undergo rapid changes all the time and the best place to get that information is the Linux documentation which is part of the kernel tree itself.

2) The need for concurrency control in a distributed system.

Many use the word **concurrency** for **parallelism**, which is incorrect. While both mean high throughput computing, **concurrency** means the ability of your application to execute as a sum of various independent parts, utilizing an infinite amount of compute/storage resources while some parts are not executed by CPU at a specific time, and **parallelism** means the ability to run various threads of execution in parallel using the available CPUs. In both ways, we have the better throughput experience, while only parallelism means concurrent execution. That said, let us focus on the need of control to synchronize distributed access of various participating execution threads. Consider this example of two threads trying to test and set a flag to 1, which is the basis of a typical lock implementation. The assumption is that if a thread is able to change the value from 0, to 1 it acquires the lock. If it sees 1, it should block until the value becomes 0. To unlock, the thread should change the value from 1, to 0. Assume also that A is initially set to 0.

thread 1	
read A(0)	thread 2
..... this thread was interrupted	
for an instruction cycle time in between	read A(0)
A=1	A=1
store A (1)	store A(1)

Here, after the thread 1 reads A, but before it stores the value back to memory, thread 2 running in another processor reads the previous value and erroneously thinks that it has obtained the lock (as it found the value of A is zero), by being able to set it to 1, causing chaos as both threads now own the

lock!. The solution for such problems is to make the process of reading and writing an atomic execution. If there is a way to lock the **system bus**, (preventing other CPUs from driving the **bus to access the same variable**) during the whole process, the second thread would not be able to read the value. On SMP/CMP systems, 'LOCK' prefixed assembly instructions on x86 architecture do just that. We can think of many other situations where different execution threads can execute the same functions, manipulating shared global data structures (like linked lists, buffer cache, memory structures etc.), causing unexpected results as different threads are 'racing' to execute the same code aka critical sections. These types of issues, which occur because of the relative timing of different parallel-running execution threads, are collectively called **race conditions**. The region of code that causes this kind of behavior when applied to parallel execution is called a **critical section**. Other than SMP/CMP, we face such issues in a UP environment as well, because threads can be preempted by another high priority thread or can be interrupted. Hence we need to control different threads working on shared data by proper synchronization primitives. For this, Linux kernel code primarily use semaphores, mutexes, spin locks, rw locks, and the all-new RCU mechanisms. We will take a detailed look at the various mechanisms in the subsequent sections. Before that, let us take a look at **atomic operations/memory barriers** which are the fundamental lock primitives and set the stage for the study of synchronization/locking.

3) Atomic functions and memory barriers

As you already have guessed, all implementations of locks invariably depend on testing and setting a shared variable across all threads/logical processors. This necessitates a mechanism for atomic operations that would let you load the variable, modify it and then store it back, before another process accesses the same memory location. Intel processors export this feature to software (operating systems) through the instruction with 'LOCK' prefix. Different processors have different levels of atomicity at instruction level (Please check the system architecture manual of your processor for finer details).

Any Linux kernel book will provide you a list of atomic operations. Also see `<linux/atomic.h>` to see various macros/data types for this purpose. All atomic operations in the Linux kernel are executed uninterrupted. All operations dealing with integers work on the data type **atomic_t**.

Examples are `atomic_dec()`, `atomic_inc()`, `atomic_inc_and_test()`, `atomic_dec_and_test()` etc. See Linux docs for details.

Memory barriers are used to prevent unexpected results arising from instruction re-ordering, at a compiler level (static) and at an instruction execution level (dynamic). As instructions (both load and store) which operate on different data can be re-ordered at architecture level, leveraging the super scalar, pipelined executions can be also be optimized by the compiler (Declaring 'volatile' will solve this problem). To explicitly tell the CPU not to do such re-ordering, **memory barrier operations** are used. The reason behind the use of these routines is: re-ordering would affect the application logic beyond the knowledge of the processor. When we deal with memory-mapped IO memory regions for Input/Output, usage of memory barriers is very important for reading consistent data. Please note that no processor does a re-ordering if there are dependencies that can be understood by the machine.

Consider the following sequence:

```
a = 2;  
b = a;
```

Any processor will execute this, as there are dependencies understood by the machine.

But if you have a condition like this:

```
a=2;  
b=3;
```

Hardware can re-order this in any way. Problems arise when you have application-level dependencies which are not understood by the compiler or processor.

Imagine a linked list addition example:

```
new->next = current->next;  
current->next = new;
```

In the above example, if the second line is executed first, you get a corrupted list. This happens because of the instruction re-ordering due to the pipelined execution. (x86 architecture never does this sort of re-ordering, but Alpha, having a weak memory consistency model, does). The solution is to put a write memory barrier after the first one.

i.e.

```
new->next = current->next;  
wmb()  
current->next = new;
```

You can use `rmb()` for preserving read ordering and `mb()` for both. There are `smp` variants `smp_rmb()/smp_wmb()/smp_mb()` as well for `smp` environments. (That will essentially cause an instruction pipeline stall in the processor). As this works against super scalar, pipelined computing, it has its own toll on the CPU performance. Always remember that the common locking primitives like spin locks, semaphores, RCU's (which are covered later in the document) all have barriers as part of the implementation. Another fundamental presumption you can have about memory re-ordering is, such issues arise only if one CPU observes the memory operations of another CPU in a multi-processing environment. A single CPU sees its own memory operations in program-order, unless compiler optimization/outside-program-code modifications (like device registers) are involved, for which you use volatile variables from a C program.

4) Hardware multi threading , Chip level multiprocessing and SMP platforms

The latest technology developments put the famous ‘Moor's law’ under a strain test. The difficulty in increasing the clock cycles/CPU and the silicon miniaturization paved the way for the incorporation of multiple cores in a single die. Intel, AMD and all the other chip manufactures have entered this multi-core business. Multi-core and shared memory symmetric multiprocessing essentially means the same thing except that multi-core CPUs are sharing the last level cache (typically L2) and the FSB (front side system bus). On SMPs, all CPUs have their own L2 caches (while all hardware registers/L1 caches are separate in both the cases). Multi-core is also known as Chip multiprocessing (CMP). Hardware multi threading (Also known as Simultaneous Multithreading (SMT)) is the ability of multiple sets of CPU registers to maintain multiple execution states. And all hardware threads share all other resources, except the minimal hardware executional state. From a locking perspective, all such technologies are identical as the enablers of parallel execution of the same code path causing concurrent access to shared data structures. Always remember that the locks are actually shared variables that need memory, which are loaded in fastest caches (L1 level) of the cores. This means we should always group related lock variables together and let it be aligned on a cache line boundary. That way, your program incurs less cache line re-fill costs. Distributed cache coherence protocol (MESI) does CPU-to-CPU communication/BUS snooping to synchronize with other CPUs’ cached data when you change its value cached in another processor. When you bring all related lock structures into a single cache line, hardware cache line algorithms synchronize all caches faster.

5) Locking introduction - Asynchronous events that affect execution flow

The understanding of the following cases is imperative when you work with kernel locks:

1. A thread acquires the lock, and then the scheduler preempts it. New thread tries to get the lock, and the old thread never can release the lock as it is not running now. This has been the case with many old preemptive /RTOS kernels which are now solved by adding preemption disable points when a thread is the owner of a lock.
2. When you deal with locks shared by softirqs/timers/bottom halves/tasklets/ hard interrupts, use only variants of spin locks which are suitable for the purpose, *e.g.* `spin_lock_bh`, `spin_lock_irqsave` etc.
3. If you are defining your own locking mechanisms, design it in such a way that it is recursive – this will save a lot of code-debugging time. It is very hard to develop low level APIs that need to work with shared resources without a recursive lock, as these sets of APIs have to be called from other high level primitives which already exist with the assumption that lower level APIs don't operate on locks. Therefore, follow the principle of “hide the policy, and export the interface”, design locks which are robust, self-monitoring and recursive.

4. Wherever possible, maintain an order of lock acquisition calls to avoid deadlocks.
5. Understand kernel functions that sleep (But printk doesn't).

6) Volatile variables, SMP/CMP platforms and locks

When we declare a variable as volatile, it has nothing to do with concurrency synchronization- We are just informing the compiler to read the memory each time it is used and act as a memory barrier.

Consider the following example:

```
int A ;
A=100;
void call-func() {
    some code...
}

for (i=0;i <1000;i++) {
    if (A == 100) call-func(); else break;
}
```

When an optimized compiler will avoid reading the memory address of A, as it will now use a CPU register to hold the value of A, it will execute this loop 1000 times because A is not modified in the code path. We cannot blame the compiler for this, as reduction in the number of memory-reads is a part of optimization. Everything works until we modify this global variable from another thread, in another function, or from a signal handler. If this variable is a pointer, even if it is a local variable, it can cause harm as such variables can hold the memory addresses which are part of IO memory (memory mapped IO port addresses). Devices can modify these locations, outside the scope of a code path of the software (Yes, *memory-mapped IO ports declared as volatile variables are the actual bridges of hardware and software*). Clear enough? Now 2 questions to the reader ,

- 1) Can we get away with volatile declaration if we use locks?
- 2) Can we get away with locks if we declare as volatile?

Think for some time about this before you continue.

It is important to understand that *both are different*. You need to use locks if you want your data to be modified in an ordered, expected manner by multiple threads. Volatile declaration doesn't help there. If you want to tell the compiler to re-read the memory each time it sees it as an operand, you must use volatile declarations, locking doesn't help there. This is not different in SMP/CMP environment. Remember that in a UP environment, with a preemptive scheduler, we have to deal with the same issues as we would face in the SMP/CMP environment.

volatile variables and the Linux kernel:

There is an interesting fact about volatile variables in Linux: they are highly unwanted and behave almost like bugs if you use them; why? The simple answer is that Linux kernel code use memory barriers explicitly and implicitly as part of the implementation of other locking primitives (like spin_locks etc. Hence, GCC compiler has no special optimization work to do on most cases which you want to prevent using volatile key words. However, due to legacy issues, there are still exceptions, like the declaration of the variable jiffies...that are modified by the timer interrupt. Prepare for a fight with other Linux kernel developers, if you have added a volatile variable to the kernel.

7) Various locking primitives

7.1) Semaphores and mutexes

These are the traditional locking primitives which are meant to be used for long durations, putting the current process to sleep state if necessary (if the lock is unavailable). Essentially, a semaphore controls a counted resource, using a 'P' operation to take it and a 'V' operation to release it. If the resource counter is less than zero after a 'P' operation, it will block until it is greater than zero. It acquires the lock if the counter is greater or equal to zero after the 'V' operation, by the current owner of the resource. All semaphores/mutexes are based on this fundamental idea.

Linux kernel APIs that use these operations are shown in the appendix.

A **mutex** is actually a semaphore with just 1 counted resource. It is used to enforce mutual exclusion of the code path, hence the name mutex.

7.2) spin locks

spin locks are the fastest mutexes in the kernel that will *busy-wait* ('spin') in the kernel. It is guaranteed that all memory stores before entering the lock are visible inside the lock (implicit memory ordering). A thread cannot sleep while holding the spin lock and the spin lock is best used if the wait time (if spin lock is owned by some other thread at the time) is less than twice the context switch time. spin lock can be called from the interrupt handlers. Various spin lock primitives are available which depend on whether you are sharing data across process context, interrupt context, or bottom half context or any combination of these. Though this is the fastest of the non-sleepy locks, it would cause severe memory bus contention when multiple processors try to read the global variable (that is cached in all per-CPU caches in all processors) that control the lock - so much so that even the CPU wishing to release the spin lock may not be able to do so as it may not win the bus arbitration (to reset the spinlock-defining variable) on a high priority basis. Perhaps we can hope that the future micro-architectures of the SMP processors may be aware of the load/store operations to memory locations that implement various low latency locks. Kernel preemption is disabled while lock is held.

7.3) Reader-writer locks & seqLocks

Reader writer locks (it exists in the form of spin locks as well as semaphores) are the optimized variants of spin locks. It is designed for the common case that in practice, one can encounter read mostly data structures than write-centric access. As a read operation is idempotent, we can have concurrent readers. We need to get an actual lock only when a writer enters the party, for which case we wait for all readers exit and then get the lock for exclusive operation. During this time, no reader can access the shared data structure. The main disadvantage of this type of lock is that writer is starved, for most occasions. SeqLock is a solution to this problem.

A seqLock consists of storage for saving a sequence number in addition to a lock. The lock is used as the synchronization primitive between two writers and the counter is used as a data consistency primitive across readers. In addition to updating the shared data, the writer increments the sequence number, both after acquiring the lock and before releasing the lock. Now what happens is, reader reads the counter twice before, and if it reads an even value, uses it for the application. Once it is done with the data, it goes back and reads the counter again. If it is still the same even number, reader is done. If it is an odd number, a writer is now updating; if it is another even number, a writer has updated; reader should re-read the counter and re-use the data for the application logic. So this sequence lock is ideal for read-mostly data structures which do not involve any memory re-claim etc. In this mechanism, readers do not have to take a lock, and the writers do not have to wait for other readers to get the lock (like in read-write locking methods).

7.4) RCU – new highly optimized locks

RCU, acronym for Read Copy Update, was invented by Paul McKenny et al. at Sequent computers who first implemented on Dynix/Ptx and then guided others to get it implemented on other operating systems like Linux of late(In future any computer science book about SMP synchronization/operating systems has to dedicate one chapter for his work around RCU technology). It works on the principle that most workloads follow a *read mostly IO pattern* and can tolerate stale data. A perfect example is IP forwarding code. It can tolerate some stale data (if it mis forwarded an IP packet with wrong route entry, higher level protocols will take care of it anyway), and it is a *read mostly data structure*. The frequency of IP lookup is much higher than route update. And we can find such structures very frequently in other domains. Initially all read-side critical sections are executed concurrently as there is no lock required, when a writer enters the world, it will take a copy of old value, and then will depend on heavy weight write-side logic that does locking across other writers (this locking logic is outside the scope of RCU), and does an implicit synchronization by waiting for all CPUs to execute at least 1 context switches for a quiescent state, to do a deferred deconstruction of the old data and updating with the modified version. In RCU readers enjoy a free ride (no locking is required, just a preemption-disable marker), write side RCU framework implements synchronization in 2 stages:

- 1) Do the partial update necessary(like exchanging pointers making the resource invisible to future readers) to make the application logic visible to other readers who entered the critical section after this semi-update is complete. From this point onwards, new readers see this data(and may be different versions as multiple writers can play in succession).
- 2) Do the rest of the update (garbagecollection, memory deallocation etc.) after all the readers stop referencing the old values, by waiting for a sufficient time, called *grace-period*. The grace period can be calculated in different ways. One pessimistic approach is to wait for all CPUs to go through a context switch after the writer has finished the step1. Once this point is reached, all the rest of the update can be done as the resources will not be referenced by any reader any more. Writer will never be able to touch this as we have various lock-out mechanisms among writers.

(RCU APIs are explained in the appendix.)

7.5 The Big Kernel Lock (BKL)

The legacy global kernel code lock is called the **Big Kernel Lock** or **BKL**. It is a recursive, spinning lock, and can sleep while owning the lock (in which case it will disown the lock when it goes to the sleep queue and will reacquire on scheduled again). All such flexibility was provided to BKL so that SMP migration of the Linux kernel would be painless. Today, BKL is not used for any new code.

```
lock_kernel();
```

```
.....critical region....
```

```
unlock_kernel();
```

Use `kernel_locked()` to check if the kernel is locked already.

7.6 Completion variables :

Use these primitives to implement simple event-based synchronization mechanism. Conventional posix threads use this mechanism in user space, in most situations. When a thread wants to wait on a predicate (i.e. to wait for a particular value to be stored in a variable), it will call **wait_for_completion()** and will be woken up when another thread does a `complete()` function. See `<linux/completion.h>` for details.

8. Sharing your process context data with interrupts/softirqs/tasklets/timers

The key concepts that you have to keep in mind with other softirqs/tasklets/timers are ,

- a) if you share your process context data with bottom half(tasklet/timers or softirq), disable

bottom half locally, and get the spin lock using `spin_lock_bh` and unlock it using `spin_unlock_bh()`. From the bottomhalf code, just use `spin_lock()`.

b) if you share your process context data with interrupts, use `spin_lock_irq` or `irqsave` variants in your process context code. (from interrupt context code, just `spin_lock()`)

c) if two different tasklets share data, use just a `spin_lock`. If same tasklets share data, you don't need to use `spinlock`, even to disable preemption of the tasklet by another hardware interrupt, as same tasklets don't run on 2 CPUs concurrently. (consider an example: one tasklet in the middle of a linked list update is interrupted, and at the same time another tasklet (of the same type) is NOT scheduled to run on another CPU because the kernel remembers as the first tasklet as still in a running state). Another caveat is that, in future tasklets may be re-implemented as concurrently executable code at which point you need to change the way you synchronize data sharing across same tasklets. You don't need to call “_bh” variants of spin lock(to disable bottom halves) under these cases.

d) if same or different softirqs share data, use just a `spin_lock()`. No need to disable bottom halves, as one softirq cannot pre-empt another softirq.

So, what i should use now ?

if you are sharing your data with either interrupts/softirqs/tasklets/timers, you have no other option but to use spin lock variants. If you share data only with other process context threads, then use spin locks for short range locking, otherwise use a simple mutex (if you can sleep). If your application neatly divided as read side and write side threads, then use reader write variants of spin locks and semaphores.... Use RCU for read mostly data structures.... use atomic operations simply to play with global counters etc... and always check the Linux kernel locking APIs for fine grained/best suitable locking for your job. Never use coarse-grained locking. (When you go to bath room for a shower you can keep your biological privacy by closing the entire house, or just your room or just the bath room, or just your bath-tub..... you now understand what is fine grained locking....)

9) Dead locks, live locks

Dead locks and live locks are common incidents we experience in real life (the real life ones are harder to solve!). Before that we will understand the simplest, recursive locking. Recursive locking means that the lock requester himself tries to acquire the same lock again. This occurs when a lower layer function tries to acquire a resource lock that is owned by the same process/thread from a higher layer function. Another reason is to call a lock from a recursive function (crime!). The solution is obvious:

handle your lock requests with discipline. Another way to solve such issues is by containing locks in well designed abstractions, which will have some state *to keep track of who owns the lock currently, and to be silent when a further lock request from the same process arrives.*

Dead locks, in a simple definition means that two or more processes are unable to make any progress as each one needs more than one lock, and each one has locked at least 1 resource which is needed by at least another process/thread in the lock chain. Another non-computer science example: Imagine a bridge that has the width of only 1 car, and 2 cars approaching each other from opposite sides. Both cars cannot proceed; because one needs the space occupied by the other (“AAA” cannot help you if these cars do not have a reverse gear!). So a dead lock doesn't mean that a resource contention, but lock contention. (In a dead lock situation, resources are up for grabs, having the consumers are locked out by each other. The solution to this dead lock is not easy. As a developer in any distributed system project, you will have to face several bugs around dead lock. But there are some guide lines to stay away from this undesirable situation.

- 1) Always maintain a protocol governing the sequence of lock requests. And let all parts of the code follow that. But this will not work in a case where one thread needs to acquire the full sequence of all locks, and another thread needs only a subset of this. When it starts, it may start from , say, 2nd or 3rd lock in the lock chain, while another orthodox thread has just acquired the first one and proceeds to acquire the 2nd one.. dead locking each other.
- 2) Allow all locks to be resilient of locking out. This means that when you add a new lock which may be related to another lock-fenced resource, tries to keep some state, invariants checking that if this falls in a lock chain, and that the previous lock in the chain is not owned by the current thread, fail the lock request.
- 3) Use reference counting in the resources. Always check the reference count, if it is less than the maximum limit and you are not getting the resource lock, leave all the acquired locks. Stochastic locking works in a similar way. It has try_lock() variants and instead of blocking, it returns with proper values that can help the caller make other decisions.
- 4) Know all parts of the code before you add locking code, and talk to all the developers who implemented other locks.
- 5) Pray!!

Live locks:

Another interesting, common case with locking systems. Imagine that you were talking to your girl friend, and the phone just gets disconnected. What will happen? Both of you will try connecting again

and again at the same time, getting only an 'engaged tone'. (If there is an asymmetry in the love, then you will get the connection back sooner as you/she will try dialing less frequently than the other party). Having experienced this, what could you do to get around this situation? Wait for a random time and dial, and then again wait for a different random time – repeat this until you succeed. Now coming back to science, Ethernet is famous and infamous for its CSMA/CD algorithm for packet injection into the LAN. The CD algorithm is actually used to avoid the situation that multiple nodes inject frames at the same time after collision detection using randomized back off technique. . Now you know what a live lock is - Live lock always happens as a by-product of the dead-lock avoidance protocol, as all parties leave the locks and the scheduler schedules all such process in such a manner that they get into the same situation again! Use randomized waits and algorithms to solve such situation. You cannot actually prevent this situation. You can only deal with it.

10) Lockless coding – a pragmatic approach

Yes, we can get away with all forms of locks in many situations, if we make enough software designs, at a higher level, that work with each other, in a shared-nothing model. It's a myth that concurrent execution always produces better performance, and a fact-learned-from-the-field is that using a single thread for all update operations always produces simpler, faster code. Imagine N threads are simultaneously reading or writing from/to memory/persistent storage medium. Do we get N* times performance or N/1 performance? In such situation, the fact is that we get worse performance than N/1 as memory bus and the I/O bus are congested, and we don't gain any advantage of SMT/CMP/SMP here. Always remember this: Use the power of N-Core for parallel processing, not for parallel memory/disk I/O.

If we engage one dedicate thread for I/O (say 1 for read operations and the other for write operations), and then have other threads process different non-overlapping data concurrently, we do not need any lock at all. If it's a traversal of a linked structure with concurrent update, we can use atomic operations cleverly to avoid locks. Before you let other threads share data, always ask yourself – "Can I re-design this module in such a way that the different threads work on different parts of the same shared data? Or can I find a data structure that lends itself to be portioned and entered from multiple CPUs concurrently?"

In theory, it is not possible 100%, but in practice, you can achieve this to a great extent. If we apply deep motivation not to use lock based primitives, we can still add concurrency/synchronization to the code as part of the design or the way different application modules work together. This calls for an all-new look at the way you design the data sharing, rather than just implementing your 'complex' code that can be used only by the smartest people with the locks. Motivate yourself to find better ways.

Or be a lateral thinker - Imagine you don't acquire the lock. What will happen then?

Can the readers tolerate stale data?

Can you recover the damage?

Is the percentage of actual update very low?

If your answer is 'YES' to all these questions, try to do without a lock.

Always remember: Using too many locks in your code indicates that you are lazy enough to design it well and sloppy enough to be happy with a hack that just works for the moment but will be a nightmare to others who want to build a complex code on top of your hack. We can build complexity only on top of simplicity. Kernel software has to be simple and should be obviously simple so that the lesser-experienced application programmers can build complex applications on top of it.

11) [Appendix ----- Kernel APIs for locking \(please read kernel documentation for more details\)](#)

NOTE : Always remember that , you enable CONFIG_DEBUG_SPINLOCK option when you test your new locking code, to help you debug faster..

A. atomic operations

```
atomic_t atomicVar ; /* define the atomic variable */
atomicVar = ATOMIC_INIT(0);
atomic_read(atomic_t *atomicVar) ;
/* To read the integer value from the variable , atomically */
atomic_set(atomic_t * atomicVar, int i) ;
atomic_add(int i, atomic_t *atomicVar);
atomic_sub(int i , atomic_t * atomicVar);
atomic_inc(atomic_t *atomicVar);
atomic_dec(atomic_t * atomicVar);
```

atomic bitwise operations :

```
set_bit(int pos, void *word) ; /* set the bit(bit position is specified by pos; pos =0 is the least significant bit) */
```

```
clear_bit(int pos, void *word); /* clears the bit atomically */
```

etc.... take a look at Linux kernel docs for more API information.

B. Spin locks

```
#include <linux/spinlock.h>
```

```
spinlock_t lockVar = SPIN_LOCK_UNLOCKED; /* initialized */
```

```
.....
```

```
..... /* some code */
```

```
/* You are about to work on some thing that you expect that other CPUs can execute this portion concurrently , hence wish to guard the section, aka , critical section */
```

```
spin_lock(&lockVar); /* If you are the first one to call this, you will get the lock, other wise you will do a "spin-wait" until the current lock owner release the lock */
```

```
.....
```

```
.....
```

```
spin_unlock (&lockVar); /* you are done and release the lock */
```

you can use `spin_lock_init()` to dynamically initialize a `spinlock_t` variable through a pointer.

`Spin_try_lock()` attempts to obtain a given spin lock, and returns zero if it is contented, rather than block-spin.

similarly, you can use the spinlock variants `spin_lock_irq`, `spin_lock_irqsave` , `spin_lock_bh` etc.. to avoid race conditions when you share data structures with interrupt handlers/softirqs/tasklets etc. (`spin_lock_irqsave/spin_unlock_irqrestore` is preferred over the unconditional variants `spin_lock_irq/spin_lock_spinunlock_irq`).

Please refer any Linux kernel book for details.

C mutexes

```
#include <linux/mutex.h>
```

```
static DEFINE_MUTEX(muextVar); /* statically define a mutex; for dynamic initialization, use mutex_init(mutexVar) variant */
```

```
mutex_lock(&mutexVar); /* Will sleep if its currently owned by any other thread */
```

```
.....
```

```
..... critical section
```

```
mutex_unlock(&mutexVar);
```

D semaphores

You use `down_interruptible()` function to do an interruptible blocking . (if you get the signal instead of lock, this call returns with `-EINTR`. Essentially this call decrement the usage counter and would block if it is less than zero. (P operation). The `up()` function, used to release a semaphore, increments the usage count. If the new value is greater than or equal to zero, one or more tasks on the wait queue will be woken up:

```
struct semaphore semVar;
sema_init(&semVar, 1); /* usage count is set to 1 */
if (down_interruptible(&semVar))
/* semaphore not acquired; received a signal ... */
else /* protected critical region starts here ... */
up(&semVar);
```

If you use `down()` function, you will put the thread into an uninterruptible sleep. Finally, Linux provides the `down_trylock()` function, which attempts to acquire the given semaphore. If the call fails, `down_trylock()` will return nonzero instead of blocking.

E reader writer locks

```
rwlock_t rwlockVar = RW_LOCK_UNLOCKED;
```

to protect read side critical sections,

```
read_lock(&rwlockVar);
```

.....
.....

```
read_unlock(&rwlockVar);
```

for writers,

```
write_lock(&rwlockVar);
```

.....
.....

```
write_unlock(&rwlockVar);
```

For a detailed kernel APIs for semaphore variants of reader writer lock, please refer any Linux documentation.

F RCU (The next big thing in the Linux kernel locking !!)

- `rcu_read_lock/rcu_read_unlock`

This delimits an RCU read-side critical section that allows writers to detect concurrent readers. This actually has nothing to do with the locking, but sets the stage for the RCU framework. On a non-preemptive kernel, this is a no-op. On a preemptive kernel, just the kernel pre-emption is disabled. This has no variants for writers. (They have to use other primitives like spin_lock variants.

- synchronize_rcu

This is the defining API of RCU that implements the deferred destruction that waits for a grace period (until all CPUs have undergone a context switch for example) and which in turn guarantees that all current readers have finished.

You call this API after removing item and updating the modified version, for future readers but before freeing item concurrent readers could still access

- call_rcu

The job of this API is same as above, but an asynchronous variant. In fact synchronize_rcu may be implemented using call_rcu.

- rcu_barrier

Blocks until all RCU callbacks on all CPUs have completed
typical usage scenario: module unloading

- rcu_assign_pointer

This primitive is used to make an update visible (after the initialization) to future readers. preventing write reordering, using write memory barriers.

- rcu_dereference

This primitive is used to get a copy of an RCU-protected pointer object to dereference preventing read reordering, using read barriers.

12) Conclusion

We have learned that locking is not a good thing for a scalable, robust kernel, but we use it to synchronize access to shared data structures as we have no other option. And can always design software in such a way that it shares global data minimally. Before getting excited about using your own lock, always ask a question to yourself, “can I re-design the software so that I don’t need to use the lock?” Always

remember that “Free ways” will be useless if one is to build traffic signals across it. When you add a new lock, or use an existing lock, do it with real care. And apply locking at the lowest layer of the stack of function calls that may wind up on top of it, and encapsulate in clever abstractions. Always shy away from changing the working code that handles synchronization well...We can expect safe ways of locking invented by smart and experienced kernel developers, defer developing your code that use heavy locking for now :-).

[About NetDiox Computing Systems](#)

NetDiox is a Bangalore(India) based center of research and education focusing around OS technologies, networking, distributed storage systems, virtualization and cloud computing. It has been changing the way computer science is taught and applied to distributed data centers. NetDiox provides high quality training and R&D project opportunity on system software, Linux kernels, virtualization, and networking technologies to brilliant BE/M.Techs/MCA/MScs who are passionate about system/storage/networking software.