Operating Systems 2

Synchronization in Linux Kernel

Arkadiusz Chrobot

Department of Information Systems

April 22, 2024

Outline

Introduction

In the kernel-space the concurrent execution of code is common. So are shared resources. To protect them from racing conditions and other similar problems Linux kernel uses a collection of synchronization methods collectively named *locks*. In this lecture the majority of these locks is described and their applications are discussed.

Atomic Operations

The simplest shared resources are bits and variables of primitive types. Most modern CPUs provide instructions that perform *atomic* operations on such resources. *Atomicity* means, that they cannot be suspended, they are performed in one step and sequentially. The problem is that Linux kernel supports many types of CPUs. Some of them provide such atomic operations as add, subtract, read, write, etc. Others have instructions that just block the data bus in multiprocessor hardware, when one of the processors accesses memory. There are also CPUs (most notably the 32-bits SPARC processors) that do not provide atomic operations at all. To unify and assure such operations for all supported CPUs, Linux kernel programmers created an abstract type called `atomic_t`. Variables of this type store integer numbers. Initially, these numbers could be only 24-bits wide, although the size of a single `atomic_t` variable is always 32-bits. The least significant 8 bits (or in other worlds one byte) were used to implement a lock, which was necessary for such CPUs as SPARCs.

Atomic Operations

In newer versions of the kernel the lock is implemented in other way and all 32-bits are now available for storing numbers. The operations for the `atomic_t` type are implemented as macros and inline functions. Some of them are specific for a given CPU, other are common for all of them. These include:

`ATOMIC_INIT()` it initializes the `atomic_t` variable in place of its declaration; as argument the macro takes an integer number,

`int atomic_read(const atomic_t *v)` atomically reads the value of the `atomic_t` variable,

`void atomic_set(atomic_t *v, int i)` atomically stores the number given as its second argument in the `atomic_t` variable,

`void atomic_add(int i, atomic_t *v)` atomically adds the number given as its first argument to the `atomic_t` variable,

Atomic Operations

`void atomic_sub(int i, atomic_t *v)` atomically subtracts the number given as its first argument from the `atomic_t` variable,

`void atomic_inc(atomic_t *v)` atomically increments the value of the `atomic_t` variable,

`void atomic_dec(atomic_t *v)` atomically decrements the value of the `atomic_t` variable,

`int atomic_sub_and_test(int i, atomic_t *v)` atomically subtract the integer number given as its first argument from the `atomic_t` variable and returns true (a nonzero value) if the result is 0; otherwise false,

`int atomic_add_negative(int i, atomic_t *v)` atomically adds the integer number given as its first argument to the `atomic_t` variable and returns true (a nonzero value) if the result is negative; otherwise false,

Atomic Operations

`int atomic_dec_and_test(atomic_t *v)` atomically decrements the value of the `atomic_t` variable and returns true (a nonzero value) if the resulting value of the variable is 0; otherwise false,

`int atomic_inc_and_test(atomic_t *v)` atomically increments the value of the `atomic_t` variable and returns true (a nonzero value) if the resulting value of the variable is 0; otherwise false.

Please note, that the `atomic_t` variables are passed to the functions by pointer, so it is not possible to perform an atomic operation on a variable of the `int` type. When 64-bit CPUs became more common the kernel developers added another type, for atomic operations on 64-bit variables, called `atomic64_t`. The macros and functions that implement operations for this type have names that start with `ATOMIC64_` or `atomic64_` prefix.

Atomic Operations

Linux kernel also provides atomic operations for bits. The macros and functions that implement these operations do not require a special data type. They usually take as arguments the number of the bit in a binary word and the address of this word, which is passed by the `void *` pointer. In theory it is possible to specify any bit in the memory using the first argument. Bit numbers start from zero for the least-significant bit. These functions and macros include:

`void set_bit(int nr, volatile void *addr)` atomically sets (to one) the nr^{th} bit in a word at the address specified by the `addr` pointer,

`void clear_bit(int nr, volatile void *addr)` atomically clears (sets to zero) the nr^{th} bit in a word specified by the address stored in the `addr` pointer,

`int test_and_set_bit(int nr, volatile void *addr)` atomically set the nr^{th} bit in a word specified by the address stored in the `addr` pointer and returns its previous value,

Atomic Operations

`int test_and_clear_bit(int nr, volatile void *addr)` atomically clears the nr^{th} bit in a word at the address specified by the `addr` pointer and returns its previous value,

`int test_and_change_bit(int nr, volatile void *addr)` atomically flips the nr^{th} bit in a word at the address stored in the `addr` pointer and returns its previous value,

`test_bit(nr, addr)` atomically returns the value of the nr^{th} bit in a word at the address stored in the `addr` pointer.

There are also non-atomic version of these functions, which names starts with `__` (double underscore) prefix. The kernel also provides two function that search for the first set or cleared bit starting at a given address. These are called respectively `find_first_bit()` and `find_first_zero_bit()`. If only one word (32 or 64 bits) should be searched, then the faster functions `__ffs()` and `__ffz()` can be applied.

Spin Locks

Spin locks (sometimes spelled *spinlocks*) are similar to semaphores, but they force the code that tries to acquire an already locked spin lock, to *busy wait*. It means that the code checks in a loop if the spin lock is unlocked. Spin locks are used for protecting bigger than a single variable resources, like queues, lists and other data structures. They make the operations on such resources *indivisible*, which means that when one such operation is performed on a given shared resource, no other *indivisible* operation on this resource can be started, until the first one is finished. In other words these operations are performed as *critical sections*. In case of spin locks these sections has to be shorter than the time needed for context switching. Although they are prevailing type of locks in the kernel source code, they are only present in the binary kernel images compiled for multiprocessors. In kernels compiled for uniprocessors and with enabled kernel thread preemption spin locks are replaced with preemption switches and in kernels compiled for uniprocessors and without enabled preemption they are not used at all.

Spin Locks

Spin locks are only useful in multiprocessor environments, due to the busy waiting mechanism they provide. These locks can be applied inside interrupt handlers, but only when the interrupt system or IRQ line is disabled, to avoid deadlocks. Spin locks are also not recursive, which means that if a code that already possesses a spin lock tries to acquire the same spin lock then it will result in a (self) deadlock. Spin locks are variables of the `spinlock_t` type (it is a structure). The API of spin locks consists of several functions and macros:

`DEFINE_SPINLOCK(NAME)` declares and initializes spin lock of a specified name,

`spin_lock()` acquires (locks) the spin lock,

`spin_lock_irq()` switches off local interrupts system and acquires the spin lock,

`spin_lock_irqsave()` saves the state of the local interrupts system, disables interrupts and acquires the spin lock,

`spin_lock_bh()` turns off the bottom halves (softirqs and tasklets) and acquires the spin lock,

Spin Locks

`spin_unlock()` releases (unlocks) the spin lock,

`spin_unlock_irq()` releases the spin lock and switches on the local interrupts system,

`spin_unlock_irqrestore()` releases the spin lock and restores the state of the local interrupts system,

`spin_unlock_bh()` turns on the bottom halves (softirqs and tasklets) and releases the spin lock,

`spin_trylock()` tries to acquire the spin lock; if it is available then locks it, otherwise returns 0,

`spin_is_locked()` returns a nonzero value if the spin lock is currently acquired; otherwise returns 0.

`spin_lock_init()` initializes the spin lock.

Spin Locks

Many resource sharing issues in the kernel can be expressed in the terms of the readers-writers problem. Linux provides a special version of spin locks for solving the *first readers-writers problem* (also called the *readers-preference*). They are called *Reader-Writer Spin Locks* or simply *R-W* Spin Locks. The API of these locks has separate functions and macros for readers and for writers. The R-W spin lock may be acquired by more than one reader or even multiple times by the same reader (in that case the R-W spin lock is recursive). The spin lock can be acquired at a given time only by one writer. Moreover, the writer is allowed to acquire the spin lock only when no reader or writer possesses this lock. The API of spin locks consists of following macros and functions:

`DEFINE_RWLOCK` declares and initializes the R-W lock (a variable of the `rwlock_t` type),

`read_lock()` acquires the R-W spin lock for a reader,

`read_lock_irq()` turns off local interrupts system and acquires the R-W spin lock for a reader,

Spin Locks

`read_lock_irqsave()` saves the state of the local interrupts system, disables interrupts, and acquires the R-W spin lock for a reader,

`read_lock_bh()` turns off the bottom halves (tasklets and softirqs) and acquires the R-W spin lock for a reader,

`read_unlock()` releases the R-W spin lock for a reader,

`read_unlock_irq()` releases the R-W spin lock for a reader and turns on the local interrupts system,

`read_unlock_irqrestore()` releases the R-W spin lock for a reader and restores the state of the local interrupts system,

`read_unlock_bh()` releases the R-W spin lock for a reader and turns on the bottom halves (tasklets and softirqs),

`write_lock()` acquires the R-W spin lock for a writer,

`write_lock_irq()` turns off the local interrupts system and acquires the R-W spin lock for a writer,

Spin Locks

`write_lock_irqsave()` saves the state of the local interrupts system, disables the interrupts and acquires the R-W spin lock for a writer,

`write_lock_bh()` switches off the bottom halves (tasklets and softirqs) and acquires the R-W spin lock for a writer,

`write_unlock()` releases the R-W spin lock for a writer,

`write_unlock_irq()` releases the R-W spin lock for a writer and turns on the interrupts,

`write_unlock_irqrestore()` releases the R-W spin lock for a writer and restores the state of the local interrupts system,

`write_unlock_bh()` releases the R-W spin lock for a writer and turns on the bottom halves (tasklets and softirqs),

`write_trylock()` tries to acquire the R-W spin lock for a writer; returns nonzero if fails,

`rw_lock_init()` initializes the R-W spin lock,

Spin Locks

`rw_is_locked()` if the R-W spin lock is already acquired it returns true (nonzero value).

Please note, that there is no `read_trylock()` function. It wouldn't have a point, because the R-W spin lock can be acquired by many readers.

Semaphores

Unlike the spin lock the *semaphore* puts the code that tries to acquire it to sleep if it is already locked. The code (a kernel thread, or any other *execution thread*) is added to a wait queue associated with the semaphore. An execution thread that possesses a spin lock cannot try to acquire a semaphore because it would cause a deadlock. Also semaphores unlike spin locks do not switch off kernel threads preemption. They are used for protecting shared resources that require critical sections that last longer than the time needed for context switching. The number of the execution threads that may simultaneously possess the same semaphore is defined by the *semaphore counter*. The semaphore is a variable of the `struct semaphore` type. It is an abstract type. Its operations are implemented by the following macros and functions:

`sema_init(struct semaphore *, int)` initializes a semaphore; its second argument is the initial value of the semaphore counter,

Semaphores

`down_interruptible(struct semaphore *)` tries to acquire the semaphore; if it fails it changes the state of the execution thread to `TASK_INTERRUPTIBLE`,

`down(struct semaphore *)` tries to acquire the semaphore; if the semaphore is not available it changes the state of the execution thread to `TASK_UNINTERRUPTIBLE`,

`down_killable(struct semaphore *)` tries to acquire the semaphore; if it fails it changes the state of the execution thread to `TASK_KILLABLE`; this function is available since the 2.6.26 version of the kernel,

`down_timeout(struct semaphore *, long)` it is available since the 2.6.16 version of the kernel; it allows the execution thread to set the maximum time of waiting for acquiring the semaphore,

`down_trylock()` tries to acquire the semaphore; if it is not possible at the time, it just returns a nonzero value,

Semaphores

`up(struct semaphore *)` it releases the semaphore and wakes one of the execution threads that awaits this event.

The semaphore can be declared and initialized with the use of the `DEFINE_SEMAPHORE` macro. Just like in the case of spin locks, there are special semaphores for the first readers-writers problem. These semaphores are variables of the `struct rw_semaphore` type. Such a semaphore can be created and initialized with the help of the `DECLARE_RWSEM` macro, or it can be just initialized with the use of the `init_rwsem()` function. The functions that handle R-W semaphores for readers have similar names to the functions that handle regular semaphores, but they end with `read` postfix. Likewise, the writers' functions handling R-W semaphores have similar names, but they end with `write` postfix. There are also `down_read_trylock()` and `down_write_trylock()` functions but unlike the `down_trylock()` function they return zero, if the semaphore is unavailable at the time they try to acquire it. Finally, there is `downgrade_writer()` function which converts a writer lock into a reader lock.

Mutexes

The kernel programmers noticed, that in most cases, semaphores that take only two values (binary semaphores) are used. They decided to add a new type of lock, to replace them. The lock is called *mutex*[1]. It is an abstract type, based on the following structure[2]:

```
struct mutex {
        atomic_t count;
        spinlock_t wait_lock;
        struct list_head wait_list;
};
```

Listing 1: Simplified definition of the `struct mutex` type.

[1]The name comes from the expression *mutual exclusion*.

[2]The actual definition of this structure has some additional members useful for debugging.

Mutexes

Unlike the `struct semaphore`, the `struct mutex` type is CPU independent. So are most of the mutex operations (some of the are just optimized for some of the CPUs supported by the kernel). The `count` field stores the state of the mutex. If its value is one, the mutex is unlocked, if its zero, the mutex is locked, if its less than zero, the mutex is locked and at least one execution thread waits for the mutex to be unlocked. The negative value of the counter field also means that a thread needs to be woken up when the mutex is available. Just like the semaphores, mutexes can be used only in the process context. They are also not recursive. The API of mutexes consists of the following macros and functions:

`DEFINE_MUTEX(NAME)` defines and initializes a mutex of the given name,

`mutex_init(struct mutex *lock)` initializes the mutex,

`mutex_lock_interruptible(struct mutex *lock)` tries to lock a mutex; if it fails it changes the state of the execution thread to `TASK_INTERRUPTIBLE`,

Mutexes

`mutex_lock(struct mutex *lock)` tries to lock a mutex; if the mutex is unavailable at this time, it changes the state of the execution thread to `TASK_UNINTERRUPTIBLE`,

`mutex_lock_killable(struct mutex *lock)` tries to lock a mutex; if fails it changes the state of the execution thread to `TASK_KILLABLE`,

`mutex_trylock(struct mutex *lock)` tries to acquire the mutex and returns one if it succeed and zero if it failed,

`mutex_unlock(struct mutex *lock)` releases the mutex,

`mutex_is_locked(struct mutex *lock)` returns zero if the mutex is unlocked; otherwise a nonzero value.

There are also real-time mutexes in the Linux kernel, which were shortly discussed in the third lecture. Any execution thread that acquires such a mutex has its priority promoted to the real-time priority. This prevents the *priority inversion* issue to occur. After the thread releases the mutex, its priority returns to what it was before acquiring the mutex.

Completion Variables

Completion Variables are simplified semaphores. They are usually used in scenarios when there are several threads and one of them has to inform the others, that it has completed its task. The completion variables are of the `struct complection` type. It is an abstract type with operations defined as the following macros and functions:

`DECLARE_COMPLETION()` declares and initializes the completion variable,

`init_completion(struct completion *)` initializes the completion variable,

`wait_for_completion(struct completion *)` waits for the signal that the other thread has finished its job,

`complete(struct completion *)` wakes a thread waiting for the signal.

To learn more about the completion variables API please refer to the fifth laboratory instruction.

Big Kernel Lock

The *Big Kernel Lock* or BKL for short was introduced to the 2.2 series of the kernel, to allow more than one CPU to run the kernel code in an SMP environment. It was supposed to be a temporary solution, but unfortunately it stayed in the kernel for much longer than originally intended. The BKL is a global spin lock, with some additional properties. An execution thread starts busy waiting if it tries to acquire the BKL when it is already locked. However the thread that holds the BKL may go to sleep, and in that moment the BKL is unlocked. It is locked up again when the thread wakes up. It is also recursive, disables kernel preempting and can be only used in the process context. Its API consists of three functions:

`lock_kernel()` acquires the BKL,
`unlock_kernel()` releases the BKL,
`kernel_lock()` checks if the BKL is locked.

In the 2.6.39 version of the kernel the programmers managed to finally get rid of the BKL. Subsequent versions of the kernel are free of this lock.

Sequence Locks

The *Sequence Locks* (also known as *sequential locks* or simply *seqlocks*) were introduced in the 2.6 series of Linux kernel. It is a simple synchronization mechanism for the *second readers-writers problem*, also called *writers-preference*. Such a lock is basically a sequence counter. Its initial value is zero. In a scenario where there is one writer and many readers the writer increments the counter before and after it changes the shared resource. Also every reader reads the counter before and after reading the shared resource and then compares the results. If the values are the same but odd then it means that the reader read the resource while the writer was changing it, and thus the value obtained from the resource may be incorrect. If the values of the sequence counter differ, then the reading operation was interleaved with the writing operation. In both cases the reading must be repeated. If there is more than one writer, then the seqlock acts for them as a spin lock.

Sequence Locks

The sequence locks are implemented as an abstract data type named `seqlock_t`. Writers use two functions for incrementing the counter of the sequence lock. The first one is the `wirte_seqlock()` function, which is used before the shared resource is modified and the second one is the `write_sequnlock()` function which is used after the shared resource has been changed. The reader, before reading the shared resource, invokes the `read_seqbegin()` function for the sequence lock and stores its result in a local variable. After reading the shared resources it calls the `read_seqretry()` function which reads the current value of the sequence lock and compares it with its previous value. The reader repeats these operations as long as the last function returns true. Both functions are usually invoked in a `do…while` loop. For more details see the fifth laboratory instruction.

Preemption Disabling

If the shared resources that need protection are local to one of the processors in a multiprocessor environment, which means they are unavailable for other processors, then disabling and enabling kernel preemption for this CPU is sufficient for synchronization. The kernel preemption is switched off when the value of the *preemption counter* (one of the members of the `struct thread_info` structure in case of the majority of hardware platforms supported by Linux) is greater than zero. The `preempt_count()` function returns the current value of that counter. The `preempt_disable()` function increments the value of the counter and the `preempt_enable()` decrements its value. The invocations of these functions can be nested, but the second one has to be called as many times at the first one to re-enable the kernel preemption. The `preempt_enable_no_resched()` function enables the kernel preemption, but does not call the scheduler.

Preemption Disabling

In a multiprocessor environment the `get_cpu()` function can be used for disabling kernel preemption on a specified CPU. It returns the identifier (a number) of that processor. The `put_cpu()` function re-enables kernel preemption on a specified CPU.

Bottom Halves Disabling

The bottom halves (sofirqs and taskles) for a given CPU can be disabled with the use of the `local_bh_disable()` function and re-enabled with the help of the `local_bh_enable()` function.

Barriers

Most of the modern, pipelined CPUs apply the *out-of-order execution* to perform instructions more efficiently. It means that some of the read and write operations can be executed in different order than they are in the source code. If they are independent then it doesn't matter, but if they are not then it may cause incorrect results of these operations. The CPU can recognize some simple patterns of the dependent reads and writes, but not the complex one. Similar issues can occur during compilation time, because most of contemporary compilers optimize the executable code for performance by reordering the instructions. To prevent such an issue the kernel provides a set of locks called *barriers*:

`rmb()` prevents changing the order of any reads that surround it,

`read_barrier_depends()` prevents changing the order of dependent reads that surround it,

`wmb()` prevents changing the order of writes that surround it,

Barriers

- `mb()` prevents changing the order of any reads and writes surrounding it,
- `smp_rmb()` in SMP systems it acts like the `rmb()` and for uniprocessors it behaves like the `barrier()`,
- `smp_read_barrier_depends()` in SMP systems it behaves like the `read_barrier_depends()` and for the uniprocessors it acts like the `barrier()`,
- `smp_wmb()` in SMP systems it acts like the `wmb()` and for the uniprocessors it behaves like the `barrier()`,
- `smp_mb()` in SMP systems it acts like the `mb()` and for the uniprocessors it behaves like the `barrier()`,
- `barrier()` prevents the compiler from optimizing the reads and writes across that barrier.

RCU Mechanism

The RCU *mechanism*[3] is an effective, highly scalable synchronization mechanism that is uses for solving the first readers-writers problem. It requires a usually small memory overhead and a fulfillment of the following requirements to work correctly:

- the reader's code that uses the shared resource cannot sleep,
- writes to the shared resource should be rare, the reads should be frequent,
- the execution thread may access the shared resource only with the help of pointers.

If the reader wants to read the shared resource it has to acquire a pointer to the resource before and perform the operation with the use of the pointer. If the writer wants to modify the resource it makes a copy of it, then changes it and publishes a pointer to this copy. If any reader tries to access the resource after this happens then it receives the pointer to the new copy of the resource.

[3]The name comes from the names of three operations: Read-Copy-Update.

RCU Mechanism

The original is destroyed after the last reader that has access to it finishes its job. The writer uses the `rcu_assign_ptr` macro to publish the pointer to the modified copy of the resource. It can also register the callback function which will destroy the original when the last reader that accesses it finishes its job. To this end the writer can apply the `call_rcu()` function. It can also destroy the original resource after returning from the `sychronize_rcu()` function. The reader uses the `rcu_read_lock()` function to acquire the pointer to the resource and the `rcu_dereference` macro to actually get access to the resource. Finally, it calls the `rcu_read_unlock()` function to release the pointer (but not the memory it points to). The macro can be used only between the two functions calls. The RCU mechanism does not offer any protection from concurrent writes, so it is mostly suitable for scenarios where there is only one writer and many readers. For other scenarios synchronization between writers must be provided.

Questions

?

The End

Thank You for Your attention!