Linux Kernel Training. Lectures 11-12

Concurrency and race conditions. Part 1

Oleksandr Redchuk
<oleksandr.redchuk@gmail.com>

April 27, 30, 2020. GlobalLogic

A bit of theory before we begin

- We need to define and describe certain number of terms and concepts before continue with Linux Kernel synchronization primitives and methods
- These terms include, but not limited to
  - Parallelism
  - Concurrency
  - Serialized execution
  - Race condition
  - Critical sections
  - Reentrancy
  - Thread-safety
- We will talk about them briefly and cover various issues represented by parallel/concurrent computing and what Linux Kernel provide to address them

Parallel computing

- **Parallelism** is a term in computer science used to describe algorithm's parallel execution degree.
- Parallel execution is used to split complex, single task algorithm into multiple, independent from each other at certain degree, most of which can execute in parallel to each other and thus grow computing power in horizontal dimension.
- It is dominant paradigm in computing architecture for last decade since 2004 when frequency scaling to grow computing power vertically reaches practical limits.
- An good example of moving to parallelism paradigm is cancellation by Intel theirs Tejas/Jayhawk cores releases and abandoning NetBurst architecture.

Concurrent computing

- **Concurrency** is a term to describe process where computations done in some time overlapping interval.
- In contrast to sequential computations it is not known at which time slice interleave happens and thus special techniques might be required to synchronize execution flow in case of reentrancy.
- It is used to simulate parallel computations to certain degree on systems where implementing true parallel execution isn’t possible (i.e. single-core systems).
- Few concurrent computation processes might benefit when running in parallel on systems with true parallelism support (e.g. multi-core systems).

Parallelism vs Concurrency

- Parallelism
  - Computing power grows in horizontal dimension.
  - Special design for data is required to achieve computing power growth (e.g. split vs sharing data across computing units).
  - Special mechanisms required for synchronization and execution with shared data usage that affect efficiency in degree proportional to methods being used.
  - Latency increase observed in degree depending on synchronization mechanisms being used
- Concurrency
  - Better computing resources utilization by reducing idle times
  - Multi-tasking by executing computations in given time slices
  - Adds overhead to overall computation process by switching tasks
  - Mechanisms for synchronization is still required, however might be simpler than when executing in parallel.

Serialized execution

- In contrast to parallel/concurrent computing there is serialized execution
  - No parallelism: one task executes at each given time (i.e. serially)
  - No concurrency: strict order of task execution, new task scheduled only after current ends (i.e. sequentially)
- Since there is no cases where execution context or data can be corrupted due to parallel/concurrent access
- It is quite often to require certain order of access to resource (serialization) even in case of parallel or concurrent execution.
- For example two tasks prepare data (producers) for third task (consumer) that expects data in certain order: at first data from the first task, then from second. Thus consumer must serialize against their data producers.

Race condition

- This is situation where two or more instances tries to perform simultaneously conflicting operations on the same data.
- There are examples of such conflicting operations:
  - Writing in one control flow and reading from another one to the same memory area
  - Writing to the same memory area from two control flows
- In general, results of such operations are undefined.
- An example can be modifying integer variable from two threads executing instructions flow in parallel.
- Note that for ISO C11/C++11 memory model defines conflicting access to same memory area as undefined. This helps compiler in decisions for code optimizations.

Code reentrancy

- It is a feature that allows execution flow within reentrant code being interrupted in the middle of the operation due to some event (e.g. hardware interrupt or exception) and entered for execution from the beginning for different data instance without corruption of previous execution context.
- Later, interrupted execution flow can resume from the interruption point and continue without any notice.
- This places several restrictions on reentrant code implementation:
  - It must not share storage of its current state in static/global variables.
  - It must not move their own code at runtime to retain behaviour.
  - It should not call non-reentrant code.
- Simple example of reentrant code is one that called recursively.

Thread-safe code

- It is a type of code that either manipulates private local copy of data or employ some mutual exclusion mechanisms to manipulate shared data.
- Thus following approaches can be taken when for making implementation thread-safe
  - Make routines reentrant where possible so that each of them operate on private local copy
  - Use Thread Local Storage (TLS) variables when code routines cross boundary is desired and reentrancy can’t be implemented.
  - Operate on immutable objects. For example on constants or data in read-only section.
  - Implement operations on shared data using atomic operations. For example when operating on machine sized data structures (e.g. pointers, machine word sized integers, etc).
  - Use a kind of mutual exclusion when thread safety needed to be implemented for operations on shared data.

Critical section

- This is a part of the code where access to the data, that would otherwise be subject to *race condition*, is protected either via disabling context interruption or, which is more preferred, some mutual exclusion mechanism.
- While this is part of the code it's main intend is to protect access to **data** shared between two or more execution flows.
- As for mutual exclusion mechanism, it must meet following requirements:
  - It should guarantee that only one execution flow can enter critical section until
  - Be hardware independent (e.g. CPU speed, model, memory type, etc), while still might use architecture specific constructions (e.g. atomic operations) in their implementation
- Depending on implementation it might be important for performance reasons to keep critical section as short as possible.

Mutual exclusion

- It is one of the way to control execution flow concurrency to prevent race conditions when accessing shared resource (e.g. memory area).
- Defined as requirement that only one execution flow can access resource within critical section at given time line.
- While main intent of it is to solve one problem, improper use might create another one: **deadlock**.
- Also overall system performance might be impacted by lock **contention** — a case where are multiple execution flows trying to enter busy critical section.
- There is also chance of **starvation** when execution flow woken up to access resource after release but without a chance to acquire lock.

Mutual exclusion (cont.)

- A classical example of deadlock is when there are two tasks A and B, executing *concurrently* and need to share access to resources RA and RB each protected with it's own (LA and LB) mutual exclusion primitive:
  - Task A acquires LA to access RA
  - Task B acquires LB to access RB
  - Then task A, holding RA via LA **blocks** in attempt to acquire LB to access RB
  - On the other hand, task B, holding RB via LB blocks in attempt to acquire LA to access RA
- Contention happened when execution flow attempts to enter critical section by lock acquisition where another execution flow already executes within same critical section with lock held. More flows attempting to acquire lock more contention is.

Mutual exclusion (cont.)

- Typical execution flow with access to shared resource protected via mutual exclusion might look as following:
  - Non-critical section path. Not attempting to access shared, protected resource.
  - Try to acquire access to protected resource by entering critical section.
  - Work with resource in critical section.
  - Leave critical section and give others chance to access critical section.
- Accessing to resource in critical section might be quite expensive. Therefore in most cases it is preferred to distinguish type of access:
  - **Shared** read access. There might be multiple execution flows accessing resource in read-only mode and thus sharing access to it.
  - **Exclusive** read-write access. There might be only one execution flow that can update resource with no readers present at the same time.

Race condition by example

- Let’s look at race condition problem by example. Implement kernel module that schedules two dedicated works on different CPUs. This gives us true parallel execution.

```
static void
allocfree_func(struct work_struct *w)
{
    if (shared_ptr) {
            kfree(shared_ptr);
            shared_ptr = NULL;
    } else {
            shared_ptr = kmalloc(1024,
GFP_KERNEL);
    }

    schedule_work_on(0, w);
}
```

```
static void
test_func(struct work_struct *w)
{
    if (shared_ptr) {
            pr_debug("ksize(%p) == %zu\n",
                    shared_ptr,
ksize(shared_ptr));
    }

    schedule_work_on(1, w);
}
```

Kernel context switching

- There are few execution flow contexts within kernel:
  - Hardware interrupt context, where ISR (Interrupt Service Routine) is executed
  - Software interrupt context, where softirq, tasklet and timer handler functions executed.
  - User context, where syscalls from user space process, kthread, workqueues are executed.
- By priority from highest to lowest context is:
  - hardirq -> softirq -> timer -> tasklet -> user context
- From synchronization perspective this can be seen as following:
  - Atomic context. In this context sleeping (blocking, scheduling) isn't allowed. In this context only atomic operations on integers or bitwise or busy-wait style locking can be used. This context includes hardware/software interrupt context, tasklets and timers.
  - Non-atomic context.This is context of either user space process, kthread or work queue. Sleeping (blocking, scheduling) is allowed here as well as atomic operations and busy-wait style locking.

Kernel context switching (cont.)

- Sometimes it is necessary to synchronize access to shared data from multiple contexts. This task requires special attention.
- For example it is quite common to share data between timer handler function executed in **atomic** context where sleeping isn't allowed and workqueue work running in **non-atomic** context.
- In general we must choose *busy-wait,* as most generic (but not best) style locking to implement correct synchronization.
- It is not problem when your user context work and timer handler function running on different CPUs.
- But it is when they run on some CPU (e.g. quite common for UP case).

Kernel context switching (cont.)

- In last case, since timer context has greater priority than workqueue user context following situation could happen when running on same CPU:
  - In user context (non-atomic, interruptible)
    - Acquire busy-wait lock to access resource (L1)
      - Hardware interrupt
        - …
          - Timer handler function (atomic, interruptible)
            - Attempt to acquire busy-wait lock (L1) that already held!!!
              - Iterate indefinitely in busy-wait loop: softlockup
- Since lock is acquired in user context, then external event (interrupt) triggers timer handler function that need L1 to work with shared data which is already held in user context and **never** can be released due to interrupted context.

Kernel context switching (cont.)

- To prevent soft lockup conditions caused by interrupted context busy-wait locking primitives may provide corresponding variants. We will talk about them next when introducing busy-wait locking primitives.
- However there are generic routines that can help with control of kernel context switch. They are divided into two pieces:
  - Disable/enable hardware interrupt handling.
  - Disable/enable software interrupt handling (softirqs, or previously bottom-half).
- While disable/enable hardware interrupt is most generic way to prevent kernel execution context switching, in may situations disabling/enabling softirqs is more than enough.
- Of course it is not good idea to disable hard/soft interrupt handling for long.

Kernel context switching (cont.)

- Here are architecture independent, defined in `<linux/irqflags.h>`, API that can be used to control hardware interrupt processing on local CPU:

```
void local_irq_save(unsigned long flags);
void local_irq_restore(unsigned long flags);
```

- Here are API defined in `<linux/bottom_half.h>` that can be used to control software interrupt (softirq) processing on local CPU:

```
void local_bh_disable(void);
void local_bh_enable(void);
```

- There is also kernel preemption that is tied together with softirqs, atomic context and other stuff. This would be described in one of the next lessons.

Kernel context switching (cont.)

- Backing to our example with user context and timer handler function sharing some data we can fix deadlock condition by simply disabling softirq handling before acquiring lock for critical section:
  - In user context (non-atomic, interruptible)
    - local_bh_disable()
    - Acquire busy-wait lock to access resource (L1)
      - Hardware interrupts are still ok, but softirqs are off on current CPU
    - Release busy-wait lock to access resource (L1)
    - local_bh_enable()
- There are lot more cases to consider. All of them described in detail in Documentation/DocBook/kernel-hacking.tmpl. Use make {html,pdf}docs to make it in either HTML or PDF form (more options available, see make help).

Linux Kernel primitives for synchronization

- Linux provide wide variety of synchronization primitives. We will cover following ones in this lecture:
  - Spinlocks
  - Reader/Writer spinlocks
  - Semaphores
  - Reader/Writer semaphores
  - Mutexes
  - Completions
- There little more advanced techniques we will talk later
  - Memory barriers
  - Read-Copy-Update (RCU) semantics
  - Per-CPU variables
  - Sequential locks

Before synchronization: atomic operations

- As said previously we can operate on shared data either within *critical section* or atomically.
- An operation called atomic if it’s action can’t be interrupted nor divided by multiple sub operations and result of it appears as either all or nothing.
- Often they implemented architecture-dependent hardware facilities, like “lock” instruction prefix on x86 that prevents any operation on CPU memory bus until instruction prefixed with “lock” completes.
- However on some architectures that lack generic support for atomic operations, or their support is limited or buggy, implement these operations using `cmpxchg()` routine.

Before synchronization: atomic operations (cont.)

- There is `cmpxchg()` routine, used not only to implement support for rest of atomic operations, but also to implement `atomic_cmpxchg()` that is used in various high level mutual exclusion primitives implementation.
- In general, arch specific code **must** provide it's implementation. Otherwise generic one from `<asm-generic/cmpxchg.h>` would be used that does not work on SMP and therefore limiting architecture support to UP implementation only that disables interrupts before operations to gain atomicity. It's prototype:

```
/* @ptr to retrieve/store result, @old and @new values */
#define cmpxchg(ptr, old, new)
```

- There are number of variations of `cmpxchg()` routines: `cmpxchg64()`, `cmpxchg_local()`, `cmpxchg64_local()` etc.

Before synchronization: atomic operations (cont.)

- There is special type, called `atomic_t` is used to distinguish atomic operations on atomic variables. This type is nothing else as structure wrapping single integer variable and defined in `<linux/types.h>`.

```
typedef struct {
    int counter;
} atomic_t;
```

```
#ifdef CONFIG_64BIT
typedef struct {
    long counter;
} atomic64_t;
#endif
```

- There is also machine word sized `atomic_long_t` type that defined together with corresponding operations in `<asm-generic/atomic-long.h>` as either `atomic_t` for 32-bit machines or as `atomic64_t` for 64-bit.

Before synchronization: atomic operations (cont.)

- There are following operations defined on atomic type(s):
  - ATOMIC_INIT() - define atomic variable at compile time
  - int atomic_read(const atomic_t *v) - retrieve atomic variable value from @v atomically
  - atomic_set(atomic_t *v, int i) - set atomic variable @v to given integer @i atomically
- This is basic arithmetic/bitwise on atomic variables
  - atomic_add(int i, atomic_t *v) - add integer @i to atomic variable @v atomically
  - atomic_sub(int i, atomic_t *v) - sub integer @i from atomic variable @v atomically
  - atomic_inc(atomic_t *v) - increment atomic variable @v by 1 atomically
  - atomic_dec(atomic_t *v) -decrement atomic variable @v by 1 atomically
  - atomic_and(int i, atomic_t *v) - perform bitwise AND on @i and @v atomically
  - atomic_or(int i, atomic_t *v) - perform bitwise OR on @i and @v atomically
  - atomic_xor(int i, atomic_t *v) - perform bitwise XOR on @i and @v atomically

Before synchronization: atomic operations (cont.)

- There is additional helpers that may either return result from operation or test that certain condition is true:
  - atomic_add_return(int i, atomic_t *v) - same as atomic_add(), but return result of addition
  - atomic_sub_return(int i, atomic_t *v) - same as atomic_sub(), but return result of subtraction
  - atomic_inc_return(atomic_t *v) - same as atomic_inc(), but return result of increment
  - atomic_dec_return(atomic_t *v) - same as atomic_dec(), but return result of decrement
  - atomic_sub_and_test(int i, atomic_t *v) - subtracts @i from @v and returns true if @v is zero
  - atomic_dec_and_test(atomic_t *v) - decrements @v by 1 and returns true if @v is zero
  - atomic_inc_and_test(atomic_t *v) - increments @v by 1 and returns true if @v is zero.
- And following “high” level atomic operation(s) defined in `<linux/atomic.h>`
  - atomic_inc_not_zero(atomic_t *v) - increment @v by 1 unless it is zero

Before synchronization: bit operations

- It is quite common to operate on bits instead of bytes or words. In Linux Kernel bit operations are atomic, unless otherwise stated explicitly.
- Since they atomic, they need architecture support for their implementation to work correctly.
- However there is generic implementation using high level synchronization primitives (irq safe spinlocks) for architectures where architecture code does not provide custom implementation.
- Note that generic implementation for both atomic ops and bit ops helps to port Linux to various architectures, possibly reducing initial port time by eliminating the need to implement all architecture specific parts at once.

Before synchronization: bit operations (cont.)

- There are commonly used routines to operate on bits provided by Linux Kernel in either architecture header files or in `<asm-generic/bitops/atomic.h>`
  - set_bit(int nr, volatile unsigned long *addr) - set bit @nr in array of unsigned longs @addr atomically. Note that @nr isn't limited to sizeof(long) * BITS_PER_BYTE.
  - clear_bit(int nr, volatile unsigned long *addr) - clear bit @nr in array of unsigned longs @addr atomically.
  - change_bit(int nr, volatile unsigned long *addr) - set bit @nr it it does not already set or clears it otherwise.
  - test_and_set_bit(int nr, volatile unsigned long *addr) - like set_bit(), but also returns previous value of the bit @nr.
  - test_and_clear_bit(int nr, volatile unsigned long *addr) - like clear_bit(), but also returns previous value of bit @nr.
  - test_and_change_bit(int nr, volatile unsigned long *addr) - change bit, and return previous val

Before synchronization: bit operations (cont.)

- There are non-atomic variant of the bit ops functions above. They should only used in conjunction with other synchronization methods. In most cases generic implementation in <asm-generic/bitops/non-atomic.h> is enough and architecture does not provide overrides. Names of non-atomic functions differ from atomic ones by “__” prefix:
  - __set_bit(int nr, volatile unsigned long *addr)
  - __clear_bit(int nr, volatile unsigned long *addr)
  - __change_bit(int nr, volatile unsigned long *addr)
  - __test_and_set_bit(int nr, volatile unsigned long *addr)
  - __test_and_clear_bit(int nr, volatile unsigned long *addr)
  - __test_and_change_bit(int nr, volatile unsigned long *addr)

Synchronization: spinlocks

- Most well known, generic and widely used synchronization mechanism in kernel.
- It is a busy-wait type synchronization mechanism. That means if lock already acquired by current execution flow all further attempts to acquire lock from other flows would block, but not sleep/scheduled to other tasks, in busy-wait loop until current flow releases lock.
- Spinlocks are simple and lightweight in their implementation, but designed to protect small and lightweight critical sections due to their busy-wait nature.
- They place no restrictions on context where they can be used: they mainly used to protect data in atomic context, but can be used in non-atomic as well.

Synchronization: spinlocks (cont.)

- However there are few restrictions on spinlock usage are still exist:
  - Locking isn’t reentrant: once in critical section acquiring same lock leads to deadlock.
  - Multiple unlocks lead to locking imbalance which leads to kernel execution context imbalance making system unusable.
  - Active (held) spinlock must not be reinitialized.
  - Memory dedicated to spinlock must not be freed until lock is active.
- There is another important features to note:
  - Access type in critical section is exclusive. There is no way to differentiate acquisitions for read and for read-write access with classical spinlock implementation.
  - Critical section under spinlocks is **atomic**. No sleep/scheduling is allowed. This also mean if you need some nested locking under spinlocks you need to use busy-wait type locking (another spinlock?).
  - Lock acquisition mechanism is serialized. This means FIFO access to protected resources.

Synchronization: spinlocks (cont.)

- As said before spinlock implementation itself is very light. However they should be used with extra caution regarding to performance.
- This happens due to exclusive locking nature of spinlocks.
- If critical section is large and/or lock contention is high, parallelism degree and as such performance would degrade linearly to the contention level.
- This is due to busy-wait nature of spinlocks: all contented flows, executed in parallel would spent huge amount of time in busy-wait iterations without switching away to other tasks.
- That creates unnecessary load on processor, leading to increases in heat production and raised voltage consumptions.

Synchronization: spinlocks (cont.)

- It should be noted, that spinlocks are optimized on kernels build as non preemptive UP or when running SMP on uniprocessor machine (thanks to alternatives system).
- Such optimization completely removes actual spinlock usage on UP, while leaving certain facilities presented by spinlock API variants like soft/hard interrupt and preemption disable/enable.
- Spinlock implementation is split to architecture dependent, which uses atomic operations (like atomic_cmpxchg() on never kernels or xadd() on older) and generic that wraps around architecture specific implementation and adds certain spinlock API variants. Let’s look at them closely.

Synchronization: spinlocks (cont.)

- Spinlock type is defined as following in `<linux/spinlock_types.h>`:

```
typedef struct spinlock {
    union {
          struct raw_spinlock rlock;

#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
          struct {
                  u8 __padding[LOCK_PADSIZE];
                  struct lockdep_map dep_map;
          };
#endif
    };
} spinlock_t;
```

Synchronization: spinlocks (cont.)

- and initializers are defined as following in `<linux/spinlock_types.h>`:

```
/* This can be used to initialize spinlock as part of data structure */
#define __SPIN_LOCK_UNLOCKED(lockname) \
    (spinlock_t ) __SPIN_LOCK_INITIALIZER(lockname)

/* This is to define spinlock in global scope */
#define DEFINE_SPINLOCK(x)    spinlock_t x = __SPIN_LOCK_UNLOCKED(x)

/* This is to initialize spinlock at runtime */
#define spin_lock_init(_lock)                          \
do {                                                   \
    spinlock_check(_lock);                             \
    raw_spin_lock_init(&(_lock)->rlock);               \
} while (0)
```

Synchronization: spinlocks (cont.)

- Here is code API to work with spinlocks:

```
/* Try to acquire lock, return nonzero (true) if succeeded and zero (false) in
 * case lock is held.  Do not busy-wait for lock to be released by holder for
 * reacquire it.
 */
int spin_trylock(spinlock_t *lock);

/* Acquire lock, if not locked. Busy-wait for lock release and then reacquire it */
void spin_lock(spinlock_t *lock);

/* Release previously acquired lock via spin_trylock() or spin_lock() */
void spin_unlock(spinlock_t *lock);
```

Synchronization: spinlocks (cont.)

- and there are alternative versions of API tied together with various kernel execution context control primitives:

```
int  spin_trylock_bh(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);

int  spin_trylock_irq(spinlock_t *lock);
void spin_lock_irq(spinlock_t *lock);
void spin_unlock_irq(spinlock_t *lock);

int  spin_trylock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
```

Synchronization: spinlocks (cont.)

- All these alternative versions are nearly equivalent of corresponding sequence of disable/enable hard/soft interrupt context switching and then lock acquisition:

```
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)
{
local_irq_save(flags);
spin_lock(lock);
}

void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)
{
        spin_unlock(lock);
        local_irq_restore(flags);
}
```

Synchronization: spinlocks (cont.)

- There is additional, less frequently used, spinlock API available:

```
/* Returns nonzero (true) if lock held, and 0 (false) otherwise */
int spin_is_locked(spinlock_t *lock);

/* Returns nonzero (true) if lock is contented,and 0 (false) otherwise */
int spin_is_contended(spinlock_t *lock);

/* helper routine that is similar to atomic_dec_and_test(), but
 * instead of returning true if result is zero acquire lock.
 */
int atomic_dec_and_lock(atomic_t *atomic, spinlock_t *lock);
```

Synchronization: spinlocks (cont.)

- Debugging spinlocks. Since they are widespread and most generic and safe synchronization mechanism, used even in generic implementation of some atomic operations there should be kernel build-in advanced debugging mechanisms:

```
# Enable basic checks of spinlocks (e.g. using uninitialized locks)
CONFIG_DEBUG_SPINLOCK=y

# Live lock reinitialization or release, statistics, etc.
CONFIG_DEBUG_LOCK_ALLOC=y
CONFIG_PROVE_LOCKING=y
CONFIG_LOCK_STAT=y
# Activate runtime locking dependency engine to debug locking at runtime
CONFIG_DEBUG_LOCKDEP=y
```

Synchronization: rwlock spinlocks

- While spinlock critical section is exclusive for both read and read-write access there is variant of **shared** read locking with **exclusive** read-write support.
- This shared read lock gives much better performance on workloads with infrequent modifications (producer) but frequent accesses for read/search (consumer).
- While read performance might be improved dramatically for correct work loads doing large or frequent modifications may lead to readers starvation thus degrading rwlock parallelism degree to pure spinlocks.
- Internally implementation is based on spinlocks, but structure and use semantics changed completely.

Synchronization: rwlock spinlocks (cont.)

- First of all one need to decide if your workload is applicable to rwlock semantics. Basically only one question must be answered:
  - Is amount of modifications is sufficiently lesser than access to unmodified data?
- If so, it is very likely that rwlocks are applicable to your workload.
- Since rwlock implementation is based of spinlocks it inherits all strengths and weakness of them. Of course rwlock is a busy-wait lock type and thus can be used in atomic context as well as critical section created by this lock type is of atomic context too.
- As with spinlocks API are split to architecture dependent and independent parts. Independent part is available via `<linux/rwlock.h>` header file.

Synchronization: rwlock spinlocks (cont.)

- Rwlock type is defined as following in `<linux/rwlock_types.h>`:

```
typedef struct {
    arch_rwlock_t raw_lock;
#ifdef CONFIG_GENERIC_LOCKBREAK
    unsigned int break_lock;
#endif
#ifdef CONFIG_DEBUG_SPINLOCK
    unsigned int magic, owner_cpu;
    void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
} rwlock_t;
```

Synchronization: rwlock spinlocks (cont.)

- and initializers are defined as following in `<linux/rwlock_types.h>`:

```
/* This can be used to initialize rwlock as part of data structure */
#define __RW_LOCK_UNLOCKED(lockname) \
    (rwlock_t)    {    .raw_lock = __ARCH_RW_LOCK_UNLOCKED,    \
                           RW_DEP_MAP_INIT(lockname) }

/* This is to define rwlock in global scope */
#define DEFINE_RWLOCK(x)    rwlock_t x = __RW_LOCK_UNLOCKED(x)

/* This is to initialize spinlock at runtime */
#define rwlock_init(lock)                                       \
    do { *(lock) = __RW_LOCK_UNLOCKED(lock); } while (0)
```

Synchronization: rwlock spinlocks (cont.)

- Here is code API to work with rwlocks:

```
/* Acquire/release read lock (shared) */
int read_trylock(rwlock_t *lock);
void read_lock(rwlock_t *lock);
void read_unlock(rwlock_t *lock);

/* Acquire/release write lock (exclusive) */
int write_trylock(rwlock_t *lock);
void write_lock(rwlock_t *lock);
void write_unlock(rwlock_t *lock);
```

- Note that API mostly the same as for spinlocks. Except there is a split for reader and writer cases.

Synchronization: rwlock spinlocks (cont.)

- and there are alternative versions of API tied together with various kernel execution context control primitives:

```
void read_lock_bh(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);

void read_lock_irq(rwlock_t *lock);
void read_unlock_irq(rwlock_t *lock);
void write_lock_irq(rwlock_t *lock);
void write_unlock_irq(rwlock_t *lock);

void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
```

Synchronization: rwlock spinlocks (cont.)

- There are same considerations against alternative versions of API for rwlocks as for spinlocks.
- Debugging rwlocks. There are the same options as for debugging spinlocks:

```
# Enable basic checks of rwlocks (e.g. using uninitialized locks)
CONFIG_DEBUG_SPINLOCK=y
# Live lock reinitialization or release, statistics, etc.
CONFIG_DEBUG_LOCK_ALLOC=y
CONFIG_PROVE_LOCKING=y
CONFIG_LOCK_STAT=y
# Activate runtime locking dependency engine to debug locking at runtime
CONFIG_DEBUG_LOCKDEP=y
```

Synchronization: semaphores

- It is integer variable with atomic access type that controls access by execution control flow to critical section it protects.
- It is invented by Edsger Dijkstra in 1965 as part of concurrent programming problem solving work.
- Their principles are very simple:
  - Semaphore variable contains non-zero value that describes how many control flows can be in critical section simultaneously.
  - If variable is zero, then next and further control flows should sleep (e.g. via yielding scheduling time to another task) until variable becomes non-zero.
  - When flow enters critical section it decrements variable by one atomically via function down().
  - When flow exits critical section it must increment variable via function up() and possibly wake up sleeping flows waiting for semaphore.
- This means only specified at semaphore initialization time number of control flows can enter critical section.

Synchronization: semaphores (cont.)

- In Linux Kernel semaphores counter is protected using raw spinlock. FIFO serialization is guaranteed at point of acquisition when semaphore is non-zero.
- When semaphore is zero, a task, that requests access to semaphore protected critical section) is added at the tail of semaphore waiters list head. This ensures FIFO serialization of waiters during process wake up when mutex released.
- There is special case of semaphores: binary semaphore defined as semaphore with maximum number of flows that can enter critical section equal to 1. That's mean only one control flow can reach critical section at given time.
- Until 2006 when mutexes being merged into kernel binary semaphores are the only interface to provide sleeping lock mutual exclusion.
- Of course nowadays binary semaphores still could be used in place of mutexes, but refer to https://lwn.net/Articles/164802/ scribing top 10 facts explaining why it is better to use mutexes instead of binary semaphores.

Synchronization: semaphores (cont.)

- Here is semaphore data structure in Linux Kernel

```
struct semaphore {
    raw_spinlock_t              lock;      /* to protect ops on @count */
    unsigned int                count;     /* number of flows */
    struct list_head            wait_list; /* head of waiters list */
};
```

- And here is API to initialize this data structure

```
#define __SEMAPHORE_INITIALIZER(name, n) /* initialize structure */

#define DEFINE_SEMAPHORE(name)    \
    struct semaphore name = __SEMAPHORE_INITIALIZER(name, 1)

void sema_init(struct semaphore *sem, int val);
```

Synchronization: semaphores (cont.)

- Following API can be used to acquire/release semaphores:

```
/* acquire semaphore and decrement counter: deprecated, should be
 * avoided where possible, blocks process in kernel
 */
void down(struct semaphore *sem);

/* acquire semaphore and decrement counter with support for process
 * interruption by fatal (killable), non-fatal (interruptible) or by
 * timeout
 */
int __must_check down_interruptible(struct semaphore *sem);
int __must_check down_killable(struct semaphore *sem);

/* acquire semaphore when possible, return when @jiffies timed out
 * without acquisition when semaphore is busy.
 */
int __must_check down_timeout(struct semaphore *sem, long jiffies);
```

Synchronization: semaphores (cont.)

- Following API can be used to acquire/release semaphores (cont.)

```
/* acquire semaphore when possible, but do not block when semaphore is
 * busy and process should sleep.
 */
int __must_check down_trylock(struct semaphore *sem);

/* release semaphore, waking up first waiting process if any or just
 * incrementing counter
 */
void up(struct semaphore *sem);
```

- There is another nice feature of semaphores: some of the routines might be used in interrupt context, when sleeping (scheduling) in general isn't allowed. Following routines are atomic and would no sleep:

```
int __must_check down_trylock(struct semaphore *sem);
void up(struct semaphore *sem);
```

Synchronization: semaphores (cont.)

- It is possible to use other `down()` routines from interrupt context when it is known that they would not sleep (i.e. successfully acquire semaphore by decrementing counter only).
- Debugging semaphores. Kernel does not provide special config options to debug semaphores, however, since implementation depends on raw spinlock and scheduler to sleep/wake processes, following configuration options might be handy when dealing with semaphore related problems:

```
CONFIG_LOCKUP_DETECTOR=y
CONFIG_DETECT_HUNG_TASK=y
CONFIG_SCHED_STACK_END_CHECK=y
```

- Also sparse(1) semantics preprocessor might be handy to detect mismatched acquire/release of semaphore.

Synchronization: semaphores (cont.)

- Implementation of the Semaphore API (a good example of a synchronization primitive template).

```
void down(struct semaphore *sem)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->lock, flags);

    if (likely(sem->count > 0))
        sem->count--;
    else
        __down(sem);

    raw_spin_unlock_irqrestore(&sem->lock, flags);

}
```

Synchronization: semaphores (cont.)

```
void down(struct semaphore *sem)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->lock,
                          flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        __down(sem);

    raw_spin_unlock_irqrestore(&sem->lock,
                               flags);
}

void __down(struct semaphore *sem)
{
          __down_common(sem,
                  TASK_UNINTERRUPTIBLE,
                  MAX_SCHEDULE_TIMEOUT);
}
```

```
int __down_common(struct semaphore *sem,
                  long state, long timeout)
{
    struct semaphore_waiter waiter;
    list_add_tail(&waiter.list, &sem->wait_list);
    waiter.task = current;
    waiter.up = false;
    for (;;) {
        if (signal_pending_state(state, current))
            goto interrupted;
        if (unlikely(timeout <= 0))
            goto timed_out;
        __set_current_state(state);
        raw_spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout);
        raw_spin_lock_irq(&sem->lock);
        if (waiter.up)
            return 0;
    }
  // ... interrupted: and timed_out: processing
}
```

Synchronization: read-write semaphores

- Similar to rwlock spinlocks providing **shared** read and **exclusive** write locking support there is read-write semaphores implementation called rwsem.
- Performance considerations for rwsem are the same as for rwlocks:
  - There is performance benefit in workloads where shared read operations prevail over exclusive write operations
  - Readers may experience starvation in case of large amount of data modifications or they expensive to perform
- In last case parallelism degree of read-write semaphores degrades to ordinary semaphores.
- Similarly to rwlocks read-write semaphores might be split to architecture dependent, optimized, part and generic. API is defined in `<linux/rwsem.h>`.

Synchronization: read-write semaphores (cont.)

- To spit read vs write access to critical section semaphore definition must be updated incompatibly with ordinary semaphores way. While exact definition is architecture dependant for generic implementation we have something like following (taken from `<linux/rwsem-spinlock.h>`):
  - If counter is 0 there is no activity on semaphore (both read and write)
  - If counter is -1 semaphore is locked exclusively for write
  - Counter greater than 0 indicates number of readers in critical section
  - If wait_list in semaphore definition isn't empty there are process(es) waiting for semaphore
- Assuming above generic rwsem implementation does not allow to specify number of concurrent readers that can come into critical section unlike to ordinary semaphores.

Synchronization: read-write semaphores (cont.)

- Generic implementation using spinlocks is in `asm-generic/rwsem.h` activated by architecture dependent Kconfig files via
  CONFIG_RWSEM_GENERIC_SPINLOCK
  option which is default to “yes” on most architectures, except ones listed in Documentation/features/locking/rwsem-optimized/arch-support.txt.
- Architecture specific implementation is stored in `arch/$ARCH/include/asm/rwsem.h`
- As with architecture spinlocks and described later in the lecture mutexes spinning by waiters from wait_list for architecture optimized rwsem implementation is implemented using MCS spinlocks activated via
  CONFIG_RWSEM_SPIN_ON_OWNER.

Synchronization: read-write semaphores (cont.)

- Here is semaphore data structure declaration:

```
/* Generic read-write semaphore data structure */
struct rw_semaphore {
__s32              count;
raw_spinlock_t     wait_lock;
struct list_head   wait_list;
#ifdef CONFIG_DEBUG_LOCK_ALLOC
struct lockdep_map         dep_map;
#endif
};
```

- It is easy to note that read-write semaphore data structure looks very similar to ordinary semaphore data structure (note that `count` is of signed type).

Synchronization: read-write semaphores (cont.)

- Here are semaphore static and run-time initializers:

```
/* Initialize rwsem statically as part of container data structure */
#define __RWSEM_INITIALIZER(name)                              \
{ .count = RWSEM_UNLOCKED_VALUE,                               \
  .wait_list = LIST_HEAD_INIT((name).wait_list),               \
  .wait_lock = __RAW_SPIN_LOCK_UNLOCKED(name.wait_lock)        \
  __RWSEM_OPT_INIT(name)                                       \
  __RWSEM_DEP_MAP_INIT(name) }

/* Initialize rwsem statically either in global or local scope */
#define DECLARE_RWSEM(name) \
    struct rw_semaphore name = __RWSEM_INITIALIZER(name)

/* Initialize at runtime using custom storage for lockdep class key */
extern void __init_rwsem(struct rw_semaphore *sem, const char *name,
                         struct lock_class_key *key);
#define init_rwsem(sem)                                        \
do {                                                           \
    static struct lock_class_key __key;                        \
    __init_rwsem((sem), #sem, &__key);                         \
} while (0)
```

Synchronization: read-write semaphores (cont.)

- Here is code API to work down/up semaphores:

```
/* Acquire/release lock for reading (shared) */
void down_read(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);

/* Acquire/release lock for writing (exclusive) */
void down_write(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);

/* Try-Lock for reading/writing -- returns 1 if successful, 0 if contention */
int down_read_trylock(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);

int rwsem_is_contended(struct rw_semaphore *sem);

/* Convert write lock to read lock when finishing with data
 * structure update but want to continue as reader
 */
void downgrade_write(struct rw_semaphore *sem);
```

Synchronization: read-write semaphores (cont.)

- Similar considerations regarding to kernel execution context where ordinary semaphores can be applied also true for read-write semaphores as well.
- Here are primitives that wouldn't sleep/block and thus are safe in atomic context:

```
int down_read_trylock(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);
```

- Of course you can use rest of primitives in atomic context in case you absolutely sure that lock will be acquired without putting flow into sleep.

Synchronization: read-write semaphores (cont.)

- There is additional API class that should be avoided whenever possible, except for the cases when used with completions (we will talk about them on next lecture):

```
# define down_read_non_owner(sem)        down_read(sem)
# define up_read_non_owner(sem)          up_read(sem)
```

- They are just preprocessor defines, unless lock allocation debugging is on, that use ordinary down_read()/up_read(). When debugging is on these routines instruct lock inspection code to treat such down/up specially.
- This is to show that semaphores (both ordinary and read-write) can be acquired/released non-owning execution flow.

Synchronization: read-write semaphores (cont.)

- Debugging read-write semaphores. Unlike ordinary semaphores rwsem support for extended debugging facilities, also available for other lock types (e.g. spinlocks, rwlocks, etc).
- These include following Kconfig options available via “Kernel Hacking” menu:

```
# Live lock reinitialization or release, statistics, etc.
CONFIG_DEBUG_LOCK_ALLOC=y
CONFIG_PROVE_LOCKING=y
CONFIG_LOCK_STAT=y

# Activate runtime locking dependency engine to debug locking at runtime
CONFIG_DEBUG_LOCKDEP=y
```

Synchronization: mutexes

- While described earlier semaphores can be used to implement mutual exclusion when accessing critical section there is special interface in Linux Kernel to do exactly this task.
- There are number of benefits of using mutexes in kernels:
  - Simpler, cleaner and less error prone API
  - Lighter implementation compared to binary semaphores
  - Built-in debugging facilities in kernel
- By the analogy with semaphore mutexes provide sleepable locking thus other execution flows waiting for lock would sleep.
- Mutex data structures and API defined in `<linux/mutex.h>`
- Refer to Documentation/locking/mutex-design.txt for more information

Synchronization: mutexes (cont.)

- While at first look mutexes is the same as binary semaphores this isn't true when looking at their implementation, which also covers why using mutexes is more preferred over binary semaphores.
- There are three states of mutex:
  - Lock count is equal to 1 (like binary semaphore)
  - Single locked (no other flows waiting to acquire), set to 0 (also like binary semaphore)
  - Less than zero (there are flows waiting to acquire)
- According to these three states there are three different code paths in mutex implementation:
  - Fastpath, mutex is unlocked, atomically (either via atomic_dec_return() or atomic_xchg())
  - Midpath, mutex is locked and lock owning task is in TASK_RUNNING state.
  - Slowpath, mutex is locked owned task is not running: put it to wait queue

Synchronization: mutexes (cont.)

- To implement semantics correctly following restrictions assumed for mutexes:
  - Only one task can hold the mutex at a time
  - Only the owner can unlock the mutex
  - Multiple unlocks are not permitted
  - Recursive locking/unlocking is not permitted
  - A mutex must only be initialized via the API (see below)
  - A task may not exit with a mutex held
  - Memory areas where held locks reside must not be freed
  - Held mutexes must not be reinitialized
  - Mutexes may not be used in hardware or software interrupt, contexts such as tasklets and timers
- All of them are enforced when CONFIG_DEBUG_MUTEXES is in effect.

Synchronization: mutexes (cont.)

- Let’s look at mutex structure definition and find differences and similarity with semaphore data structure:

```
struct_mutex {
    atomic_long_t       owner; /* pointer to owner’s task_struct and flags */
    spinlock_t          wait_lock;
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
    struct_optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
    struct list_head    wait_list;
#ifdef CONFIG_DEBUG_MUTEXES
    Void                *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map  dep_map;
#endif
};
```

Synchronization: mutexes (cont.)

- Another difference between mutexes and binary semaphores, that becomes clear after describing mutex restrictions is that mutexes locks are owned by the task acquires it.
- That means contrary to semaphores mutexes should only be released by task that acquires them.
- This is enforced when CONFIG_DEBUG_MUTEXES or CONFIG_MUTEX_SPIN_ON_OWNER (that is configuration for mid path or optimistics locking) options is in effect.
- However there is common place between mutexes and binary semaphores: this is list of waiters and it’s spinlock protector.
- Initializing mutexes is similar to semaphores, see documentation

Synchronization: mutexes (cont.)

- There are few routines defined in `<linux/mutex.h>` to work with mutex:

```
void mutex_lock(struct mutex *lock);

int __must_check mutex_lock_interruptible(struct mutex *lock);
int __must_check mutex_lock_killable(struct mutex *lock);

int mutex_trylock(struct mutex *lock);

int atomic_dec_and_mutex_lock(atomic_t *cnt, struct mutex *lock);

void mutex_unlock(struct mutex *lock);

int mutex_is_locked(struct mutex *lock);
```

Synchronization: mutexes (cont.)

- Debugging mutexes. Unlike semaphores mutexes provide integrated debugging facilities activated via CONFIG_DEBUG_MUTEXES option in “Kernel Hacking” -> “Lock debugging (spinlocks, mutexes, etc…)”.
- There is other, generic options, available to debug not only mutexes problems:

```
# Debug whenever live lock is re initialized (e.g. via mutex_init())
# or memory is released (e.g. via kfree() while lock is alive).
CONFIG_DEBUG_LOCK_ALLOC=y

CONFIG_PROVE_LOCKING=y
CONFIG_LOCK_STAT=y
# Activate runtime locking dependency engine to debug locking at runtime
CONFIG_DEBUG_LOCKDEP=y
```

Synchronization: completions

Lightweight API to wait for some process to have reached a specific state.

```
struct completion {
        unsigned int done;
        wait_queue_head_t wait;
};
```

- Static declarations in file scope

  ```
  static DECLARE_COMPLETION(setup_done);
  ```

- Dynamically allocated

  ```
  void init_completion(struct completion *x);
  ```

- Within functions the static initialization should always use:

  ```
  DECLARE_COMPLETION_ONSTACK(setup_done);
  ```

- Reuse

  ```
  void reinit_completion(struct completion *x);
  ```

Synchronization: completions (cont.)

- Waiting functions (see `linux/completion.h` for full list)

```
void wait_for_completion(struct completion *);
int wait_for_completion_interruptible(struct completion *x);
int wait_for_completion_killable(struct completion *x);
unsigned long wait_for_completion_timeout(struct completion *x,
                                          unsigned long timeout);
long wait_for_completion_interruptible_timeout(struct completion *x,
                                          unsigned long timeout);
long wait_for_completion_killable_timeout(struct completion *x,
                                          unsigned long timeout);

bool try_wait_for_completion(struct completion *x);
bool completion_done(struct completion *x);
```

- Signaling completions

```
void complete(struct completion *);
void complete_all(struct completion *);
```

Synchronization: completions (cont.)

Completions in kthreads, works, etc

- Bad (race condition):

```
static int thread_fn(void *data)
{
    struct my_job *mjob = data;

    /* … */

    while ( /* … */ ) {
        /* … */
    }

    /* … */

    complete(&mjob->done, 0);
        /* preemption! */
    do_exit(0); // return 0;
}
```

```
struct my_job *mjob = alloc_my_job();

start_my_job(mjob);

wait_for_completion(&mjob->done);

free_my_job(mjob);
```

Synchronization: completions (cont.)

Completions in kthreads

- Right way:

```
static int thread_fn(void *data)
{
    struct my_job *mjob = data;

    /* … */

    while ( /* … */ ) {
        /* … */
    }

    /* … */

    complete_and_exit(&mjob->done, 0);
}
```

```
// In kernel/exit.c

void complete_and_exit(
        struct completion *comp,
        long code)
{
        if (comp)
                complete(comp);

        do_exit(code);
}
```

Things we talk in this session

- Cover theoretical aspects and problems of horizontal scalability
  - Describe differences between parallelism and concurrency.
  - Race conditions and how to avoid them.
  - Code reentrancy, thread-safety, critical sections.
  - Synchronization, serialization and when they required.
  - Show why is better to avoid shared data access when possible.
- Describe basic synchronization primitives available in kernel such as:
  - atomic operations on integers and bits
  - spinlocks/read-write spinlocks (rwlock)
  - semaphores/read-write semaphores (rwsem)
  - mutexes
  - completions
- Show options provided by the kernel to debug synchronization issues.

References

- Linux Device Drivers, Third Edition. Chapter 5: *Concurrency and Race Conditions*
- Documentation/locking/spinlocks.txt
- Generic Mutex Subsystem (LWN.net)
- Documentation/locking/mutex-design.txt
- Documentation/scheduler/completion.txt

That’s all for now. Thank you for viewing.

Authors:

- Serhii Popovych, Software Engineer
  - Cisco-IOSXE Linux SDK, GlobalLogic Ukraine LLC
  - Email: serhii.popovych@globallogic.com
- Oleksandr Redchuk, Software Engineer
  - GlobalLogic Ukraine
  - Email: oleksandr.redchuk@gmail.com