Mutual Exclusion Pattern

The mutex pattern is primarily designed for mutual exclusion between threads.

Unlike the semaphore and the spinlock, the mutex can not be used from interrupts. It is the first synchronization primitive in our pattern stack that is required to be both locked and unlocked by the same application level thread.

In this module we are going to look at the implementation of a mutex and discuss how the mutex works.

We will also cover how the mutex handles thread priorities and avoids priority inversion.

Defining Characteristics

The key defining characteristic of a mutex is that it must be locked and unlocked by the same application level thread. It is absolutely not usable from an interrupt and is the first in our list of synchronization primitives that operates entirely on OS threads.

- **Lock/Unlock**: these are two main operations that can be performed on a mutex. **`Lock`** attempts to acquire an exclusive lock on the mutex. **`Unlock`** releases the lock and wakes up next thread that was waiting for it.
- **Mutual exclusion**: guaranteed mutual exclusion between threads. One thread on any CPU has is allowed to hold the same mutex at any given time.
- **Does not disable interrupts**: unlike spinlock, the mutex is not aware of interrupts and does not hold interrupts disabled while mutex is locked. The system can still process interrupts and respond to hardware events as normal.
- **Maintains thread queue**: the mutex also maintains a thread queue just like the semaphore, but unlike the semaphore, the mutex also implements priority inheritance for threads that currently hold the mutex so that they can inherit a higher priority if another thread with a higher priority also tries to acquire the mutex.
- **Recursive locking is allowed by same thread**: the same thread can try to lock the same mutex twice without any issues. The mutex is thread aware and knows about the owner that is currently holding the mutex. Thus the same thread can call other functions that also try to lock the mutex while keeping the mutex locked without any issues.

Use Cases

- **Mutually exclusive access**: the mutex is primarily designed to ensure scalable mutually exclusive access to resources that can be shared between OS threads. It ensures that only one thread can have access to a resource while the mutex is locked. Trying to acquire a mutex from an interrupt is invalid.
- **Thread safety**: when we talk about thread safety of an API, we usually refer to the ability of multiple threads to share the object which the API operates on when calling the functions defined by the API. This thread safety is typically implemented as mutual exclusion using a separate mutex for each instance of the object. This allows each instance of an object to be safely accessed in a mutually exclusive fashion by multiple threads.

Benefits

The benefits of using mutex for mutual exclusion compared to lower level primitives include:

- **Priority inversion avoidance**: unlike the semaphore pattern, the mutex implements the much needed priority inheritance making sure that highest priority threads in an application do not need to depend on lower priorities of other threads when they contend with these threads for the same resource.
- **Ease of use**: a mutex explicitly checks whether it is being locked and unlocked by the same thread providing extra level of debugging provided that it is used as intended for simple locking and unlocking around operations that perform data access to shared data.

Drawbacks

- **Single resource only**: compared to a semaphore, a mutex does not provide a counter like the semaphore does. The internal counter of a mutex is only used for keeping track of recursive locks done by the same thread that locked the mutex when it was still unlocked.
- **Threads only**: mutex is by definition unsuitable for use in interrupts and for any kind of synchronization with hardware since it is implemented for thread only usage and expects to keep track of the owner thread and since it needs to know the owner thread of the mutex in order to handle priority inheritance. Therefore mutex can not be used when trying to synchronize access to a variable between a thread and an interrupt. In such a scenario, a spinlock should be used instead.

Implementation

When we separate code paths into threads and make them either run fully parallel in hardware (if we have multiple cores) or contend for resources on a single core (using an RTOS), we have the problem of having to ensure mutually exclusive access to shared data.

While a spinlock will provide safety in such a scenario, it lacks thread awareness and so it becomes way too expensive to use a spinlock for general purpose locking. A spinlock not only blocks thread switching - it also blocks interrupts so we must use spinlocks only for very short periods of time.

To deal with locking on a broader basis, we need a mechanism of mutual exclusion which is fully thread aware and does not need to block interrupts for longer than very short periods of time.

The solution to this problem is the mutex.

Mutex mechanics

A mutex is similar to a semaphore but is also slightly more complex. The main difference is in the way it is intended to be used. While a semaphore is intended to be given in one place and

taken by a completely unrelated piece of code, the mutex has a hard requirement to always be locked and unlocked by the same thread.

Internally, the mutex holds a queue of threads which are waiting for it to become unblocked so that we can track who is able to continue next when the mutex is unlocked. A mutex implementation uses spinlock underneath to synchronize access to the mutex data structure and to this queue of threads. When a new thread tries to lock the mutex one of the following things happens:

- If the mutex is locked then current thread id is stored as owner of the mutex and the locking function returns success.
- If the mutex is already locked by another thread and we try to acquire it, then the current thread is added to a wait queue inside the mutex object and its state is suspended.
- If the mutex is already held by a lower priority thread, the holding thread's priority is adjusted to the priority of the current thread which is trying to acquire the mutex.

Upon unlocking a mutex, one of the following things happen:

- If there are threads in the wait queue of the mutex then the highest priority thread is picked, its state is set to "ready" and the scheduler is called to switch context directly to that thread.

Just like in the case of a semaphore, the mutex unlocking operation may switch context right away during the unlock and before the unlock function returns. It all depends on whether there are higher priority threads waiting for the mutex to be unlocked. If this is the case, then the highest priority thread is allowed to take the mutex next and continue until it is put back into suspended state.

Priority inheritance means that a higher priority thread can never be blocked for longer than necessary if a lower priority thread is holding a mutex that it is trying to lock.

Locking a mutex

Mutex uses spinlock to guard critical sections that can not be interrupted by any other code at all. This is important for mutex count variable as well as the time between trying to acquire a mutex and putting current thread into pending state waiting for the mutex to be unlocked. It is important that no other thread context switch happens in these places because otherwise we would end up with the mutex in an inconsistent state when another thread runs - which would be dangerous.

```
// there is a global lock for all mutexes
static struct k_spinlock lock;

int k_mutex_lock(struct k_mutex *mutex, k_timeout_t timeout)
{
    int new_prio;
    k_spinlock_key_t key;
    bool resched = false;

    __ASSERT(!arch_is_in_isr(), "mutexes cannot be used inside ISRs");
```

```
    key = k_spin_lock(&lock);

    if (likely((mutex->lock_count == 0U) || (mutex->owner == _current))) {
        mutex->owner_orig_prio = (mutex->lock_count == 0U) ?
                    _current->base.prio :
                    mutex->owner_orig_prio;

        mutex->lock_count++;
        mutex->owner = _current;

        k_spin_unlock(&lock, key);
        return 0;
    }
```

```
    ...
    if (unlikely(K_TIMEOUT_EQ(timeout, K_NO_WAIT))) {
        k_spin_unlock(&lock, key);
        return -EBUSY;
    }

    new_prio = new_prio_for_inheritance(_current->base.prio,
                        mutex->owner->base.prio);

    if (z_is_prio_higher(new_prio, mutex->owner->base.prio)) {
        resched = adjust_owner_prio(mutex, new_prio);
    }

    int got_mutex = z_pend_curr(&lock, key, &mutex->wait_q, timeout);

    if (got_mutex == 0) {
        return 0;
    }
```

```
    ...
    /* timed out */
    key = k_spin_lock(&lock);

    struct k_thread *waiter = z_waitq_head(&mutex->wait_q);

    new_prio = (waiter != NULL) ?
        new_prio_for_inheritance(waiter->base.prio, mutex->owner_orig_prio) :
        mutex->owner_orig_prio;

    resched = adjust_owner_prio(mutex, new_prio) || resched;

    if (resched) {
        z_reschedule(&lock, key);
    } else {
        k_spin_unlock(&lock, key);
    }

    return -EAGAIN;
```

We start by checking if we are trying to lock a mutex from an interrupt handler. Mutexes are not interrupt aware and so this is always invalid and must generate a panic.

Next, we check if current thread is trying to lock a mutex that it already holds locked. If this is the case, then we increment internal lock count and return. This special case handling prevents deadlock in a situation where current thread calls another function that tries to lock the same mutex. Having this special case handling in place, simplifies our application code because we do not need to worry about splitting our functions into two functions where one does the operation and another (public one) locks the mutex and then calls the first one.

If mutex is locked and timeout is set to **`K_NO_WAIT`** then we simply return **`EBUSY`**.

If the mutex is already locked and the current thread is not the owner of the mutex, then we check whether the thread that holds the mutex locked needs to have its priority adjusted. We adjust the priority and put current thread into pending state and insert it into the work queue of the mutex. Now our thread is waiting and if we get zero result code from **`z_pend_curr`** then we know that we got the mutex.

If we did get the mutex, then we simply exit from the **`k_mutex_lock`** function and the current thread is now in the mutually exclusive critical section.

If the operation has timed out then we adjust the priority of the next thread on the wait queue back to what it was before and return **`EAGAIN`** code.

Unlocking a mutex

```
int k_mutex_unlock(struct k_mutex *mutex)
{
    struct k_thread *new_owner;

    __ASSERT(!arch_is_in_isr(), "mutexes cannot be used inside ISRs");

    CHECKIF(mutex->owner == NULL) {
        return -EINVAL;
    }
    /*
     * The current thread does not own the mutex.
     */
    CHECKIF(mutex->owner != _current) {
        return -EPERM;
    }

    /*
     * Attempt to unlock a mutex which is unlocked. mutex->lock_count
     * cannot be zero if the current thread is equal to mutex->owner,
     * therefore no underflow check is required. Use assert to catch
     * undefined behavior.
     */
    __ASSERT_NO_MSG(mutex->lock_count > 0U);
```

```
    /*
     * If we are the owner and count is greater than 1, then decrement the
     * count and return and keep current thread as the owner.
     */
    if (mutex->lock_count > 1U) {
```

```
        mutex->lock_count--;
        goto k_mutex_unlock_return;
    }
    k_spinlock_key_t key = k_spin_lock(&lock);
    adjust_owner_prio(mutex, mutex->owner_orig_prio);

    /* Get the new owner, if any */
    new_owner = z_unpend_first_thread(&mutex->wait_q);
    mutex->owner = new_owner;

    if (new_owner != NULL) {
        /*
         * new owner is already of higher or equal prio than first waiter since
         * the wait queue is priority-based: no need to adjust its priority
         */
        mutex->owner_orig_prio = new_owner->base.prio;
        arch_thread_return_value_set(new_owner, 0);
        z_ready_thread(new_owner);
        z_reschedule(&lock, key);
    } else {
        mutex->lock_count = 0U;
        k_spin_unlock(&lock, key);
    }
k_mutex_unlock_return:
    return 0;
```

Unlocking operation starts off by checking that unlocking is valid.

- Unlocking is invalid if used from a code that is called from an interrupt handler. Mutexes are not interrupt aware.
- Unlocking is invalid if the mutex was never locked.
- Unlocking is invalid if the mutex is being unlocked by a thread different from the thread that locked it.

The mutex count takes care of keeping track of recursive mutex locks by the same thread. No locking is necessary here when accessing the count since we are guaranteed to be the owner of the mutex at this point in the unlock operation. If the mutex count is still higher than 1 then we simply decrement the count, and return.

Next the mutex locks the spinlock so that we do not end up with a context switch before the unlock function has completed all its internal operations. We unlock the spinlock at the end of the unlock function - at which point the scheduler will switch context if another thread has been unblocked by unlocking the mutex.

If the lock count is one then this can only happen if the current thread was the thread that locked the mutex and the mutex is about to become unlocked. We take the first highest priority thread from the queue of threads that is waiting for the mutex and make it the new owner of the mutex.

If the new owner is not null then we mark the new thread as ready and call reschedule to pend a new reschedule event. Once we call **`z_reschedule`** and it finishes the reschedule, it unlocks our spinlock and the system will continue in the next thread that needs the mutex (where it

called `z_curr_pend`) or simply return from current unlock operation.

Best Practices

- **Object oriented design**: the fastest and most sure way to ensure that it is easy for you to manage concurrent access to resources is to make each shared resource into a well defined object which can also then have a mutex as a member variable making it clear that the lock should be used by all public API functions when accessing the resource. Without object oriented design, things tend to get very messy.
- **Lock data, not code**: just like with spinlock, the mutex should lock only sections that access the shared resource. If possible, data should be copied to stack and the resource should be released before processing the values - provided that this does not create race conditions with other threads (ie. if you need to later write data then you can not release the lock between read and write operation).
- **Reschedule on busy in work queue items**: if you use a mutex in a work queue, it can often make sense to reschedule the work queue item instead of waiting on a locked mutex. In such a scenario, you would try to acquire the mutex using zero timeout and then if unsuccessful, you would reschedule the work item. This ensures that your mutex does not block the whole work queue. More on this in the section on work queues.

Common Pitfalls

- **Deadlocks**: just like with semaphores, this can occur if you lock one mutex and another thread locks another mutex and then you try to lock the other mutex that the other thread holds and the other thread tries to lock your mutex. In such a scenario, the two threads are deadlocked because neither one will be able to continue. This issue is handled by always locking and unlocking multiple mutexes in the same order.
- **Thread Starvation**: holding the mutex for longer than necessary causing other threads to be delayed. This issue is handled by locking only while accessing data.

Alternatives

- **Semaphore**: use a semaphore when your use case requires locking to happen in one thread and unlocking to happen in a different thread. This is a case of signaling and so a semaphore is better suited than a mutex.
- **Condition variable**: condition variable provides a way to unlock a mutex while waiting for a condition to become true. It is useful when you are waiting for one or more events and would like to do so without keeping a resource locked.

Conclusion

In this module we have covered the mutex pattern which we use as the primary way to guarantee mutually exclusive data access to variables that we share between threads.

The mutex is thread aware but not interrupt aware - meaning that it is always invalid to use mutex in any way from an interrupt handler.

Mutex locking and unlocking operation is very inexpensive when the probability of threads trying to lock the same mutex concurrently is low.

Quiz

- Why is it illegal to lock/unlock the mutex from an interrupt handler? What key functionality does this design choice make possible?
    1. This makes it possible to focus the implementation specifically on threads and make the mutex optimized mechanism for mutual exclusion between threads. It would not be possible to ensure locking/unlocking by the same context if had to support interrupts.
    2. It would not allow mutual exclusion between interrupts.
    3. Mutex uses spinlock which can not be used inside interrupt handlers.
- Why is it the case that a mutex can not be locked by one thread and unlocked by another thread? What implications would it have and what key functionality of the mutex would be impossible to implement if this was allowed?
    1. Because mutex must keep track of which thread is locking it so that it can wake up that thread when it becomes available.
    2. If we were to allow unlocking of a mutex by another thread, we would not be able to guarantee mutual exclusion because another thread would then be able to free a mutex while we expect it to be locked.
    3. It would be impossible to implement interrupt concurrency if this was allowed.
- What happens to a lower priority thread if it is holding the mutex and is currently suspended when a higher priority thread tries to acquire the same mutex? When does the lower priority thread get to run again?
    1. The thread is placed on the internal queue of the mutex.
    2. The priority of the thread that is currently holding the mutex is increased until it releases the mutex.
    3. The priority of the thread trying to acquire the mutex is increased.
- If multiple mutexes are locked/unlocked, in what sequence should these mutexes be locked/unlocked to avoid deadlock?
    1. The mutexes can be locked and unlocked in any order.
    2. This is an invalid situation. You should only use one mutex at all times.
    3. The mutexes can be locked in any order, but must be always unlocked in reverse order to the order in which they have been locked.