Semaphore Pattern

The semaphore pattern is one step above the spinlock pattern and is in fact implemented using spinlocks. But it has one key difference: instead of targeting mutual exclusion, the semaphore pattern is designed to solve the problem of signaling between application multiple application level threads or application level threads and interrupt handlers.

The semaphore pattern thus provides the much needed link on top of the spinlock which allows us to decide where our CPU is going to jump next after a piece of code has executed (and "given" a semaphore).

Defining Characteristics

- **Give/Take**: these are two main operations on a semaphore. **`Give`** increments the internal counter and marks the semaphore as free. **`Take`** decrements the counter or queues current thread in the semaphore’s internal queue until it becomes free again.
- **Maintains counter**: the semaphore is usually implemented as a counting semaphore - meaning that it can be taken multiple times without delay and then once the counter reaches zero, the semaphore will queue the calling thread until another thread gives the semaphore. For simple signalling, the semaphore usually has a count of 1 or 0 meaning that it can only be either free or taken.
- **Thread aware**: the semaphore pattern is thread aware from scheduler perspective - meaning that it can instruct the OS scheduler to jump to a new piece of code when one other piece of code gives the semaphore. This is different from simple spinlock because we are now able to call **`k_sem_give()`** in one thread (or interrupt) and come out of **`k_sem_take()`** in another thread at first valid opportunity. A semaphore thus almost works like a teleporter for our CPU core by teleporting it to different places in code which are market by a call to **`k_sem_take()`**.
- **Maintains thread queue**: multiple threads can wait on the same semaphore and these are maintained in a priority queue of waiting threads inside the semaphore data structure. This allows the semaphore logic to wake up the highest priority thread when a semaphore is given and the scheduler to then jump to that thread immediately. This allows multiple threads to try to lock the same semaphore without spinning on a flag like with spinlock - instead the threads are swapped out and not returned to until the semaphore is given.

Use Cases

- **Signalling from interrupt handler**: a semaphore (if it provides interrupt safe API) can be used to wake up (unblock) an application thread when an interrupt handler runs. For this, we try to take the semaphore from an application thread, and if the semaphore is busy, our thread will then be swapped out to the semaphore thread queue. Then, inside an interrupt handler, we can give the semaphore and the semaphore will mark the first waiting thread as ready and pend a context switch. Once the interrupt handler is finished, the scheduler interrupt will execute and jump the CPU to the place where the other thread was trying to acquire the semaphore. Thus our code can continue in the place where it was waiting for the interrupt event.
- **Signalling between threads**: just like we can signal between interrupts, we can also use

semaphores to signal between threads. We can have one thread attempt to take a semaphore and be placed to sleep if this was not successful. Then another thread can give the semaphore and the semaphore will wake up the waiting thread and pend a context switch. The scheduler will then run the waiting thread at the first opportunity where it makes sense - ie if the waiting thread is lower priority than the currently running thread then it will have to wait - otherwise it will be waken up right away.
- **Waking up a thread pool**: we can have multiple threads waiting on a counting semaphore and once we give that semaphore, we would be waking up one thread. Thus we can give the semaphore multiple times to wake up more than one thread. The worker thread would then perform some action after taking the semaphore and then attempt to take it again. If there is more work to do then it will successfully proceed or otherwise be put to sleep. Note however, that all threads are not woken up atomically - only one at a time. For atomic wakeup of multiple threads you can use broadcasting feature of condition variable pattern.

The semaphore is not designed as a mutual exclusion mechanism. It does not have priority inheritance. It does not track what thread is holding it - like a mutex would do. Thus the primary class of use cases for a semaphore is signalling.

Benefits

- **Very lightweight**: the semaphore is very lightweight compared to other thread aware synchronization primitives. It's main purpose is to pend current thread until semaphore becomes available and then wake up the highest priority thread in response to the semaphore becoming available. There is no complex logic beyond that.
- **Usable from interrupt handlers**: semaphores are the simplest and most lightweight way of sending a signal from an interrupt handler to an application thread. Mutexes can not be used in interrupt handlers and work queues are more memory intensive when used to defer work from interrupts.

Drawbacks

- **No priority boosting**: if a high priority thread tries to take a semaphore that has been taken by a low priority thread, the low priority thread will not be boosted in priority in order to complete faster. Generally a semaphore should therefore never be used for mutual exclusion between threads - only for signalling.
- **Not suitable for mutual exclusion**: design decisions of a semaphore are not compatible with design decisions of a mutex. Use mutex instead for mutual exclusion.

Implementation

Let's consider a common scenario:

Suppose we have a lengthy hardware operation that expects data to be loaded into a FIFO and then transmitted over a wire link. When transmission is done, hardware generates an interrupt which gives us indication that hardware operation is completed.

We would like to have a method of loading data into hardware, then relinquishing control of the CPU while the operation is in progress and then returning from our transmission function only once the hardware has signalled that transmission was completed.

```
void some_device_transmit(
    struct some_device *self,
    const void *data,
    size_t size){
    // (1) load device registers
    // (2) suspend this function until operation is done
    // (3) continue and return only once the operation is done
}
```

We have a hard requirement that we can not sit and poll the device while we are waiting for it. Instead, we would like the CPU to either be put to low power mode or for it to jump to some other code that needs to run, while we are waiting for the resource to become available again.

The solution to this problem is a semaphore.

The semaphore must work closely with the RTOS scheduler to allow one operation to wait for a signal from another operation. In the above case we would like to wait for an interrupt from the device signaling to us that transmission has completed.

Semaphores must be implemented in such a way that they work from an interrupt meaning that we must be able to unblock a task waiting for a semaphore from an interrupt handler to tell the RTOS that it needs to resume the function that has been waiting for the semaphore in question.

A semaphore in software works just like a railroad semaphore - the code will be paused (while other tasks can still run) until the semaphore can be taken. When another task gives the semaphore then our task that is waiting to take is able to acquire it. We can then continue.

File: some_device.c

```
struct some_device {
    struct k_sem done;
};

// this will be called asynchronously by hardware
static void some_device_isr(void){
    struct some_device *self = &_instance_of_device;
    // check whether transmission is completed
    // ...
    // signal that operation is done
    k_sem_give(&self->done)
}

void some_device_transmit(
    struct some_device *self,
    const void *data,
    size_t size){
    // (1) load device registers
    // ...
    // (2) suspend this function until operation is done
    k_sem_take(&self->done);
    // (3) continue and return only once the operation is done
    // ...
}
```

Giving a semaphore

Here is the actual implementation of the semaphore give function:

Giving a semaphore

```
void z_impl_k_sem_give(struct k_sem *sem)
{
    k_spinlock_key_t key = k_spin_lock(&lock);
    struct k_thread *thread;

    thread = z_unpend_first_thread(&sem->wait_q);

    if (thread != NULL) {
        arch_thread_return_value_set(thread, 0);
        z_ready_thread(thread);
    } else {
        sem->count += (sem->count != sem->limit) ? 1U : 0U;
        handle_poll_events(sem);
    }

    z_reschedule(&lock, key);
}
```

Here we first acquire a spinlock which is used to maintain mutual exclusion up until the point that we have successfully rescheduled to another task (important because giving a semaphore must always result in the code jumping to the place where another piece of code is trying to acquire it).

Once we have acquired the spinlock, we then unpend the first highest priority thread from the semaphore thread queue. The queue is already sorted by priority so when we get one thread from it, it will always be the highest priority thread.

If there is a thread waiting, then we set its status to 'ready' and call reschedule with the lock we have acquired (this lock will be released once rescheduling has completed ensuring that we will not be interrupted until then.

If there are no threads waiting on the semaphore then we bump its 'free' count within limit and also call reschedule - except that in this case reschedule will simply release the spinlock since most likely no additional threads have changed status to ready.

The reason why we do not bump the count when there are threads in the queue will become apparent shortly.

Taking a semaphore

Now let’s have a look at the function that takes the semaphore:

Taking a semaphore

```
int z_impl_k_sem_take(struct k_sem *sem, k_timeout_t timeout)
{
    int ret = 0;

    __ASSERT(((arch_is_in_isr() = false) ||
          K_TIMEOUT_EQ(timeout, K_NO_WAIT)), "");

    k_spinlock_key_t key = k_spin_lock(&lock);

    if (likely(sem->count > 0U)) {
        sem->count--;
        k_spin_unlock(&lock, key);
        ret = 0;
        goto out;
    }

    if (K_TIMEOUT_EQ(timeout, K_NO_WAIT)) {
        k_spin_unlock(&lock, key);
        ret = -EBUSY;
        goto out;
    }

    ret = z_pend_curr(&lock, key, &sem->wait_q, timeout);
out:
    return ret;
}
```

The first thing we do is assert on the timeout which can not be set if we are in an interrupt handler (it will panic if this is the case).

Then we acquire the spinlock again and check if the semaphore count has free slots (ie is larger than zero). If it does, then we simply decrease the count, unlock the spinlock and return. This is because we can freely grab the free slots without any thread scheduling operations.

If however there are no free slots (ie the count is zero), then we check the timeout and pend the current thread on the semaphore wait queue for the specified time. If the timeout is set to **`K_NO_WAIT`** we simply return **`EBUSY`** without pending the thread.

Additional details

What does it actually mean?

It means that our **`count`** variable represents the number of resources that are currently free.

- If we initialize a semaphore to 0 with a maximum count of 1. This means that we have zero resources available and at most one resource is available. If we then attempt to take it from a thread, then our thread is put to sleep on the queue. If we then give that semaphore, our pending thread is woken up and **the count still remains zero**. This is correct, because we have unblocked one resource and we consider it such that the thread that just woke up is then "using" this virtual resource.

- If we start with the same state - semaphore at zero with maximum count of one and we then give that semaphore, then we now have one slot available. If we now try to take it then our thread instantly returns with the semaphore taken and the count is at zero.
- Suppose we initialize the semaphore to count 2 with a maximum count of 2. This means that we have two slots available and both of them are free. If we now give the semaphore then nothing changes. We can also take it twice without any issues - the thread that takes it will simply return with success. If we however try to take it the third time, then the thread is put on the queue. If we then give it once, then the count is still zero because we have one thread waiting. If we give it again with no threads waiting then the count of "free" slots goes to one - meaning that now one other thread can easily grab it without waiting.

Thus you see that the logic in the source code is in fact correct - even though it may not seem obvious when looking at it for the first time.

Best Practices

- **Give in one thread, take in another**: one of the best guiding factors for use of semaphore is asking the question: will I always take it in one thread and give it in another? if this is not the case and you need to both take and give as part of the same sequence then it is a signal to you that you are trying to do mutual exclusion and should therefore consider a spinlock or a mutex depending on whether it should be thread aware or not. However, if you are signalling between different context/threads then the favor falls on the semaphore instead.
- **Use for resource management**: you can use spinlock for mutual exclusion to a data variable (like a queue) and a semaphore to signal threads that there is work to be done. Thus giving a semaphore from a critical section guarded by a spinlock will defer the time when a thread will wake up in response to the semaphore until the critical section ends (ie spinlock is unlocked).
- **Avoid circular dependencies**: you must avoid a scenario where two threads may end up waiting on each other’s semaphores. If this happens then both threads will lock up - this situation is referred to as a "deadlock".

Common Pitfalls

- **Priority inversion**: This is a situation that occurs when a low-priority thread holds a resource busy that a high-priority thread needs - causing the high priority thread to be blocked. Mutexes avoid this through priority inheritance but semaphores do not have this ability due to simple design and different use case.
- **Deadlock**: This situation can occur when multiple semaphores are used to control access to a shared resource and there is a circular dependency between them. For example, if two threads need to access resources A and B and each acquires a semaphore for the resource it needs, a deadlock can occur if the first thread acquires the semaphore for resource A and the second thread acquires the semaphore for resource B - and then

both threads try to acquire the second semaphore to the other resource that each needs. The best way to avoid this is to always give and take multiple semaphores in the same sequence order.
- **Semaphore never given**: this can occur for example if a semaphore is given from an interrupt - but is only given if some condition is true. Suppose an interrupt only signals if transfer was successful but does not signal if transfer had an error. Thus a thread waiting for an operation to complete will never be unblocked and will never continue because the semaphore was never given. Such problems are sometimes not easy to debug - the best way to prevent them is to fully unit test all code paths through your interrupt handlers making sure that the semaphores are given correctly along all paths.

Alternatives

- **Condition variables**: these are more complex patterns which can be used to signal many threads at once (ie they are all woken up before picking which one should run next).
- **Mutexes**: this pattern is used primarily for mutually exclusive access to a resource between threads. A mutex must be given in the same thread that has acquired it - thus mutexes can not be used for signalling between threads.

Conclusion

In this module we looked at the semaphore implementation and how semaphores help us send signals one level above the spinlock mechanism - between interrupt handlers and threads - and between threads as well.

Semaphores are very lightweight and we have seen that taking a semaphore that is free, only involves a simple spinlock acquisition and decrease in the count of free slots.

We have also seen that the semaphore maintains a thread queue of threads that are waiting for the resource slots maintained by the count variable. These threads are placed on a queue and do not consume any resources until the semaphore becomes available again.

We have also seen that the rescheduling happens atomically and if the highest priority waiting thread is ready to run then a call to **`k_sem_give`** continues directly at the end of **`k_sem_take`** inside the other thread that is ready to run. This is a powerful insight which you can directly use to structure your embedded firmware source code.

Quiz

- How does the semaphore ensure that giving a semaphore in one thread, directly makes the processor end up at the end of the call to semaphore take function of the thread that is waiting for the semaphore? When does this occur and when does this not occur?
  1. When semaphore is given, then the scheduler function is executed from the semaphore give function and swaps context to the new thread right away.
  2. When semaphore is given, the scheduler context switch is pended. If it is currently not possible to do the switch (such as inside an interrupt) then the switch is pended until it is possible to do it. If newly unblocked thread is higher priority then

we switch to it right away.
    3. When the semaphore is given, the context switch is triggered but happens first when the system tick timer expires and it is time to run another thread.
- Why is the semaphore count not incremented if there are threads waiting for the semaphore to become available? What does the count actually represent?
    1. Because the semaphore count doesn’t represent how many threads are waiting.
    2. Because the semaphore count is only decremented when the semaphore is acquired.
    3. Because then the semaphore count would have to be incremented and decremented again which would be a waste operation.
- How does a semaphore differ from a spinlock?
    1. The semaphore is aware of threads and is used to wake up one thread at a time, while the spinlock is used to maintain mutual exclusion between interrupts on the same CPU and code executed by other CPUs.
    2. The semaphore can be used in interrupts.
    3. The semaphore can be used in place of spinlock.
- How does a semaphore differ from a mutex? What functionality does a mutex have that the semaphore doesn’t?
    1. The semaphore is just simpler implementation of a mutex.
    2. The semaphore is used for signalling, while mutex is used for mutual exclusion between threads.
    3. The semaphore is used for mutual exclusion between interrupts while the mutex is used only for threads.
- Why should a semaphore never be used for mutual exclusion in place of a mutex?
    1. Because it does not keep a queue of waiting threads.
    2. Because it does not prevent starvation like the mutex does through priority inheritance.
    3. Because it is not aware of interrupts.
- What thread will be scheduled to run first when multiple threads are waiting for a semaphore and the semaphore is given by another thread or an interrupt?
    1. The highest priority thread.
    2. The lowest priority thread.
    3. The first thread that started waiting on the semaphore and has not run yet.
- When does the CPU switch to the awakened thread when a semaphore is given from an interrupt handler?
    1. At first opportunity after returning from the current interrupt handler.
    2. When the system tick timer expires.
    3. Right away.