Conditional Pattern

The conditional variable pattern provides a special kind of synchronization primitive that implements a mechanism that allows one or more threads to wait for a certain condition to be met before proceeding.

When a condition is met, one or more threads that have been queued waiting for it can be awakened and then proceed to perform the task that needs to be done after the condition has been met.

A conditional variable is typically used together with a mutex lock to ensure that the condition being waited on is properly protected from concurrent access.

The basic mechanism is that a thread acquires the mutex, checks the condition and if the condition is not met, the thread releases the mutex and waits on the conditional variable.

Another thread, having acquired the mutex, can then change the condition and signal the queued threads by using the conditional variable. The waiting thread then re-acquires the mutex lock and continues execution.

Defining Characteristics

- **Wait/Signal/Broadcast**: these are the three main operations that can be performed on a conditional variable. **`Wait`** is executed by the waiting thread, **`Signal`** unblocks one thread, **`Broadcast`** unblocks all threads waiting.
- **Mutex protection**: a conditional variable is always used in conjunction with a mutex. When a thread is waiting on a conditional variable, it releases the lock, and when the condition is met and conditional variable is signalled, the thread reacquires the lock and continues execution with the mutex locked.
- **Simultaneous wakeup**: a conditional variable can be used to "broadcast" the condition to multiple threads waiting for the condition to be met. This is different from semaphore and mutex where only one thread is picked from the wait queue in that all the waiting threads can be marked as ready before a reschedule is triggered. In such a scenario the threads always proceed in sequence of priority and sometimes in parallel depending on how many CPU cores are available.
- **Checked in a loop**: the condition which is signalled through the conditional variable is typically checked in a loop. This is because the conditional variable can be used to wake up the main thread for multiple different conditions and it is up to the thread waiting on the conditional variable to check exactly which condition has been met.

Use Cases

- **Signaling multiple threads**: The conditional variable is used to wake up one or more threads when a condition is met. The mutual exclusion to all user data belonging to the condition is guarded by the mutex that is used in conjunction with the conditional variable.
- **Waiting for a set of multiple complex conditions to occur**: The conditional variable enables thread safe checking of multiple potentially complex conditions to be met before proceeding. Multiple checks can be performed safely with the mutex locked and the

thread can wait for the condition variable signal with the mutex unlocked - allowing other threads to perform operations that use the same mutex while the current thread is waiting for the condition to be met.
- **Producer-consumer pattern**: the conditional variable can be used in order to repeatedly wake up a consumer thread when a new item is added by the producer. The consumer thread can then decide whether to consume the item directly or whether to wait for complimentary information before consuming an item. Conditional variables enable this kind of flexibility.

Benefits

- **Arbitrary conditions**: this pattern allows checking for arbitrary set of conditions to be true. With mutex and semaphore we can only check for a resource to be available or not. With conditional variables we can introduce arbitrary checks into the mix and perform these checks safely through the use of mutual exclusion against other threads using the accompanying mutex.
- **Mutex unlocked while waiting**: another valuable benefit of a conditional variable is that the synchronization mutex is kept in unlocked state while waiting for signal through the conditional variable. This allows the mutex to be acquired and released by other threads while the waiting thread sleeps. Once it's time to wake up, the conditional variable makes sure that the mutex is locked again before the waiting thread checks the conditions again. Thus to the waiting thread it looks like it had the mutex locked at all times even though the mutex was unlocked while it was asleep.
- **Can be signaled from interrupt handler**: conditional variables can be signaled from interrupt handlers - although you can not wait for a conditional variable inside an interrupt handler for obvious reasons.

Drawbacks

- **Complexity**: a conditional variable must always be checked in a loop and you must follow a few specific rules when waiting for a condition to occur such as always checking condition with the guarding mutex locked and remembering to unlock the mutex.
- **Memory**: waiting for a conditional variable always requires one mutex per thread that will be waiting concurrently. This adds additional memory requirements.

Implementation

So how do we implement the scenario where we would like to unblock one or more threads atomically upon getting a signal?

Here is what we would like to have:

- We want to keep a flag called "done" which is protected by mutex named "lock".
- We would like to have each thread lock the mutex before checking the flag value.
- If the value is false, each thread should unlock the mutex and be put to sleep until a wakeup signal is received.

- Upon being woken up, each thread should once again try to lock the mutex and then check the "done" flag and if done is true then it should perform its work and then unlock the mutex when done.

Furthermore, the whole process should be thread safe so we must make sure for example that if more than one thread is waiting for the signal then all threads should be awakened before the scheduler picks the next thread that needs to run. This is important in order to maintain correct behavior even when we have more than one CPU core.

The solution is what we call a "conditional variable". A conditional variable has three operations:

- **Wait**: this operation waits for the conditional to be signalled.
- **Signal**: this operation unblocks one thread that is waiting in the queue (or is ignored if no threads are currently waiting)
- **Broadcast**: this operation unblocks all threads that are waiting on the queue.

Usage scenario

We need to be able to implement the scenario described above in code as follows:

```
struct context {
    struct k_mutex lock; // mutex for ready
    bool ready; // threads only continue when this is true
    struct k_condvar cv; // condvar for signalling
}

// this will be called from several threads to wait for ready
void wait_for_ready(struct context *self){
    k_mutex_lock(&self->lock);
    while(!self->ready){
        // this will atomically do the following:
        // (1) unlock the mutex
        // (2) put the thread to sleep
        // Upon wakeup from signal:
        // (1) lock the mutex again
        k_condvar_wait(&self->cv, &self->lock, TIMEOUT);
    }
    k_mutex_unlock(&self->lock);
    // our thread continues here
}

// this is called from one thread once condition is ready
void signal_ready(struct context *self){
    k_mutex_lock(&self->lock);
    self->ready = true;
    k_condvar_broadcast(&self->cv);
    k_mutex_unlock(&self->lock);
}
```

The mutex guards the "condition" represented by the ready flag. This does not need to be a single flag, but rather this can also be a set of more complex conditions. The conditional

variable is used to signal waiting threads about the change in the condition - it does not however contain data about what has changed. The checks for the condition must therefore be done each time we get a signal and until we get the condition that we want (in a loop).

Waiting For Events

```
int k_condvar_wait(struct k_condvar *condvar, struct k_mutex *mutex,
                   k_timeout_t timeout)
{
    k_spinlock_key_t key;
    int ret;

    key = k_spin_lock(&lock);
    k_mutex_unlock(mutex);

    ret = z_pend_curr(&lock, key, &condvar->wait_q, timeout);
    k_mutex_lock(mutex, K_FOREVER);

    return ret;
}
```

Waiting on a conditional variable means that we must atomically unlock the supplied mutex first and then place our current thread on the queue that waits for the notification to be sent. To achieve this, we use a spinlock that we lock before unlocking the mutex and we then pass this spinlock to the **`z_pend_curr`** which will pend current thread and only unlock the spinlock after it has completed this action. This ensures atomicity.

Once we later wake up (when the condition has been signalled), we proceed with the **`k_mutex_lock`**. This will only happen when the conditional variable has been signalled and now the running threads will contend for the mutex and continue once they have been able to acquire it.

Note that locking the mutex has timeout set to **`K_FOREVER`**. This is correct because it ensures that we are guaranteed to always hold the mutex when returning from **`k_condvar_wait`** and we should never have the situation where locking the mutex would timeout.

The timeout which is passed as parameter is only used as timeout for receiving a signal (ie when and if it expires, the conditional variable wait operation will try to acquire the mutex and return).

Signaling Events

The signaling operation awakens next highest priority thread and allows it to acquire the mutex.

```
int k_condvar_signal(struct k_condvar *condvar)
{
    k_spinlock_key_t key = k_spin_lock(&lock);

    struct k_thread *thread = z_unpend_first_thread(&condvar->wait_q);
```

```
    if (thread != NULL) {
        arch_thread_return_value_set(thread, 0);
        z_ready_thread(thread);
        z_reschedule(&lock, key);
    } else {
        k_spin_unlock(&lock, key);
    }

    return 0;
}
```

If the new thread is eligible for reschedule then the code will continue at that thread's **`k_condvar_wait`** call and we will attempt to acquire the mutex there. If not, then the thread will run once higher priority tasks have finished.

Broadcasting Events

The "broadcast" operation works just like signal operation except that it marks all threads as ready and then reschedules as many of them as there are CPU cores. This is accomplished by looping through the queue, setting the status of each thread to ready and then calling reschedule:

```
int k_condvar_broadcast(struct k_condvar *condvar)
{
    struct k_thread *pending_thread;
    k_spinlock_key_t key;
    int woken = 0;

    key = k_spin_lock(&lock);

    /* wake up any threads that are waiting to write */
    while ((pending_thread = z_unpend_first_thread(&condvar->wait_q)) !=
           NULL) {
        woken++;
        arch_thread_return_value_set(pending_thread, 0);
        z_ready_thread(pending_thread);
    }

    z_reschedule(&lock, key);

    return woken;
}
```

Note that the whole looping operation is guaranteed to be done atomically and we only exit the critical section after we have rescheduled (which is why we need to pass the spinlock to **`z_reschedule`**.

The return from this function is the number of threads that has been awakened.

Best Practices

- **Use sparingly**: only use conditional variable when you need to either signal threads from

an interrupt with additional data that must have mutual exclusion protection between threads or you explicitly need to wake up several threads at once. Usually you can accomplish the signalling with a semaphore in a much more lightweight fashion.
- **Always check in a loop**: when you are checking for the condition, make sure to do it in a loop where you repeatedly wait on the conditional variable if the condition you expect is not met. Code can evolve in such a way that your conditional variable is signaled even when your expected condition is not met and your code needs to continue to work correctly even when that is the case.

Common Pitfalls

- **Spurious wake-ups**: even if it looks to you like the conditional variable is only signalled when your expected condition is met, it may still be signalled sporadically due to other reasons. On Linux such other reasons could be a signal received by the process which requires the system call to be restarted. This is mitigated by always checking the expected condition in a loop.

Alternatives

- **Semaphore**: a semaphore can be used to signal an event to one thread at a time just like a conditional variable but the drawback is that we do not have the same implementation guarantees with a semaphore such as with a conditional. A conditional is explicitly designed for queuing for an event and then waking up one or more of the queuing threads while guaranteeing that protection is maintained and mutex is kept unlocked while the thread is waiting.
- **Work queue**: it is possible to check a condition by submitting a work queue item. In such a scenario, the action of checking for the condition is deferred to the work queue and this action then notifies a list of callbacks that have been registered as listeners for the action. The biggest problem is that this does not provide a way for threads to queue for a broadcasted event notification and therefore we can not wake up multiple threads simultaneously.
- **Event bus**: one commonly used alternative to conditional variables in the context of broadcasting an event is an event bus. In such a scenario, multiple threads can subscribe to a topic and then any events that other threads send through the event bus to that topic will be broadcasted to all subscriber threads.

Conclusion

In this module we have looked at how we can implement signaling from one thread to one or more other threads.

A conditional variable is a very simple concept to implement and it is also quite light weight besides the necessity to always lock a mutex.

We now have a pattern in our toolbox which allows us to atomically wake up multiple threads from a third thread or an interrupt handler!

Quiz

- Can a conditional variable be signaled from an interrupt handler?
    1. No. Conditional variables can only be signalled from threads.
    2. Yes. This is perfectly fine.
    3. Yes, but only one thread can be awakened at a time.
- When a conditional variable **`broadcast`** method is used, at what point do all the threads wake up and in what order do they continue after that if there is only one CPU?
    1. They continue in the order in which they are able to acquire the mutex.
    2. They continue sequentially one at a time in the order in which they were waiting for the conditional.
    3. They continue all at the same time concurrently.
- If there are multiple CPUs and broadcast is used to signal a conditional variable, can a thread be swapped in and start running before all threads have been woken up?
    1. No. The reschedule will only happen when all waiting threads have been marked as ready to run.
    2. Yes, but only if the new thread is a higher priority thread.
    3. Yes, but only if we use broadcast method.
- What is the main difference between a conditional variable pattern and the semaphore pattern?
    1. The conditional variable pattern explicitly guarantees that the mutex always remains locked while current thread is scheduled in, but is unlocked when current thread is swapped out. Semaphore doesn't provide any such mechanics.
    2. The semaphore can be used form interrupts while the conditional variable can't.
    3. The logic is largely the same.
- If multiple threads are waiting on a conditional variable, which one will be chosen to run first if the conditional variable is signaled (not broadcasted)?
    1. They can all potentially run if there is more than one core but they will only return from 'wait' function after having acquired the mutex.
    2. They will always start running in sequence.
    3. They will continue all at the same time and all of them will return from wait operation so that application code can check for condition.
- Why is it always important to check for the condition itself in a loop even if you know that you are only signaling when the desired condition is true? If you don't do this, when will it cause you trouble?
    1. Because another thread could potentially alter the condition.
    2. Because we have exclusive access to the condition.
    3. Because the 'wait' call can wake up for a variety of reasons such as a timeout and not necessarily because the condition is true.