Spinlock Pattern

The most basic form of concurrency that is above the pure concurrency in hardware is the interrupt driven concurrency.

This is the first place where an action happening in hardware is forwarded to software. This is also the first place where we need to deal with concurrency in our software.

The interrupt masking pattern (or spinlock) is a way of safely managing concurrency at the level of interrupts. It also extends to concurrency between processor cores - from which the name "spinlock" originates (a core spins in a tight loop while another core keeps a resource busy).

In single core systems, the spinlock simply works as a mechanism for momentarily disabling interrupts in order to avoid concurrency conflicts in code.

In this section we are going to cover this mechanism, how it works, and also why it is the foundation for higher levels of concurrency.

Defining Characteristics

A spinlock consists of two parts:

- **Atomic variable**: this variable is used together with a set of special atomic operations to ensure that it can only be set by a single core at any given time.
- **Tight loop while locked**: this is only used when we want our code to support safe execution on a multiprocessor system. This mechanism loops on an atomic variable. We loop until we are able to atomically set the variable when it was previously zero (atomic compare and swap).
- **Interrupt masking**: we then disable interrupts and store the previous state of the interrupts in a variable called 'key'. This allows spinlocks to be locked recursively without any issues (interrupts are only enabled when the first spinlock is unlocked and the key is restored - we will look at how this works in this module).

Use Cases

Spinlocks are used in all major operating systems at the lowest level to ensure that we are able to synchronize data access between our normal mode code and the code that runs as part of interrupt handler.

- **Data access synchronization with interrupts**: any code that needs to access the same data that is also accessed by an interrupt handler must disable at least that interrupt handler before accessing the data. Spinlock provides a generic mechanism to do this.
- **Multi-core synchronization**: In a multi-core system, an interrupt can be executed by any one of the cores. Atomic operations together with a memory barrier are the only way to ensure that a certain memory location is only modified by a single core and no other core. Spinlock provides a standard mechanism for achieving this kind of synchronization.

Note that the spinlock is used at the very core (in kernel code only) and is a foundation for semaphores, mutexes and all other primitives. You would never use spinlock in application code which does not directly interface with interrupts. Spinlock is therefore almost never mentioned in materials on multi-threaded programming - even though it is **always** there. On a linux system, spinlocks exist (and make any kind of sense) only in the kernel. In application code, you should only use higher level synchronization primitives instead.

Benefits of this pattern

The spinlock is a generalization of the standard `irq_lock/irq_unlock` approach where we would just disable interrupts.

- **Generalized approach**: this minimizes potential bugs specially in multi-core systems and ensures that all locking is agnostic with regards to number of cores in the system.
- **Extremely lightweight**: the spinlock algorithm is very lightweight. In fact, on a single core system you can simply omit the looping part and then it simply translates to locking/unlocking of interrupts (but you still retain the possibility of building your code for a multi-core system without having to chase down every place where you disable interrupts).
- **Usable in interrupt handlers**: the spinlock can be used in interrupts to disable higher priority interrupts and also to synchronize between more than one core - making it useful for synchronization even between interrupt handlers as well (although interrupts should not share data).

Drawbacks

- **No concept of threads**: a spinlock is a very simple mechanism which has no concept of threads. Unlike mutex, a spinlock has no possibility to redirect execution to another thread once it is unlocked. It is simply a mechanism to synchronize data access between code and actions called directly by hardware (the interrupt handlers).
- **Disables interrupts**: this is both a feature and also a potential problem because when a single spinlock is held, NO other code can interrupt the critical section. This is because all interrupts are disabled and all events are held back until the spinlock is released. This means that under no circumstances should you ever hold a spinlock longer than necessary to synchronize data access (lock data - not code).

Spinlocks are not for synchronization between threads - but they are essential to implementing such synchronization safely.

Implementation

In order to understand how the spinlock is implemented and why, let’s first refresh some knowledge about interrupts (exceptions).

A modern CPU is able to automatically push registers to currently active stack and jump to the address specified in its interrupt vector table to allow software to handle the exception.

Additionally we almost always have following hardware features available to us:

- **Global interrupt mask**: On ARM architecture this is called PRIMASK and it’s a signal that controls whether exceptions are enabled at all. We can disable exception handling at any time using special CPU instructions. Any exceptions that occur while exceptions are disabled will be handled once exceptions are enabled again if they are still pending by then.
- **Interrupt priorities**: each interrupt can be configured with a priority. Value of zero is highest priority and then as the priority number increases we get the "turn" at which a particular interrupt will be handled. If an exception has priority number 3 then its turn to run is only after exceptions with priority numbers lower than 3 have been handled.
- **Base priority**: on ARM this is called BASEPRI. This is a setting that allows us to disable exception handling of exceptions with the same or lower priority than the base priority. This allows disabling of non-system interrupts while allowing other time critical interrupts to occur (with certain limitations of course as to what the interrupt handler is allowed to do in such scenario). Setting base priority to zero value disables it.

Normal application code can be thought to run at lowest interrupt priority. It will only run if no exceptions are pending or interrupts are disabled. Naturally, if we set base priority to a value other than zero from application code then what we are doing is disabling the CPU from preempting our code when any exception between base priority and the priority of our code occurs. This is a safe way to synchronize between our code and say "uart" interrupt while still retaining the ability to quickly respond to some other interrupt that needs to be handled immediately.

Since both our application code and even our exception code can be preempted by a higher priority exception at any time, problems arise if exception code needs to share data with lower priority application code. This is true for any operation on data that requires more than one step - for example reading a memory location, adding a number to it and writing back the result.

Sharing data with ISR

```
struct data {
        int foo;
};

static struct data data;

void interrupt_routine(void){
        data.foo += 5;
}

void data_user_code(struct data *self){
        self->foo += 8;
}

void main(void){
        data_user_code(&data);
}
```

Disassembly of interrupt

We can compile this code and then disassemble it to see exactly why this creates a problem:

```
Disassembly of section .text:

00000000 <interrupt_routine>:
        int foo;
};

static struct data data;

void interrupt_routine(void){
   0:   e52db004        push    {fp}             ; (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
        ; Code blow implements: data.foo += 5;
   8:   e59f301c        ldr     r3, [pc, #28]    ; 2c <interrupt_routine+0x2c>
   c:   e5933000        ldr     r3, [r3]
  10:   e2833005        add     r3, r3, #5
  14:   e59f2010        ldr     r2, [pc, #16]    ; 2c <interrupt_routine+0x2c>
  18:   e5823000        str     r3, [r2]
}
  1c:   e1a00000        nop                      ; (mov r0, r0)
  20:   e28bd000        add     sp, fp, #0
  24:   e49db004        pop     {fp}             ; (ldr fp, [sp], #4)
  28:   e12fff1e        bx      lr
  2c:   00000000        .word   0x00000000
```

Disassembly of user code

```
00000030 <data_user_code>:

void data_user_code(struct data *self){
  30:   e52db004        push    {fp}             ; (str fp, [sp, #-4]!)
  34:   e28db000        add     fp, sp, #0
  38:   e24dd00c        sub     sp, sp, #12
  3c:   e50b0008        str     r0, [fp, #-8]
        self->foo += 8;
  40:   e51b3008        ldr     r3, [fp, #-8]
  44:   e5933000        ldr     r3, [r3]
  48:   e2832008        add     r2, r3, #8
  4c:   e51b3008        ldr     r3, [fp, #-8]
  50:   e5832000        str     r2, [r3]
}
  54:   e1a00000        nop                      ; (mov r0, r0)
  58:   e28bd000        add     sp, fp, #0
  5c:   e49db004        pop     {fp}             ; (ldr fp, [sp], #4)
  60:   e12fff1e        bx      lr
```

We can see that the code that adds the numbers to the variable always involves at least three instructions: load, add and store.

Assuming that the interrupt can occur at any time, the data_user_code function can be interrupted anywhere during the "data.foo += 8" code (lines 40-50). If this happens, and the exception does something to the value as well then the resulting value at the memory location of that variable will be wrong.

We therefore need to protect lines 40-50 so that interrupt handler that also accesses the same data can never occur during the time while we are accessing that variable and doing operations with it.

The simple solution is to momentarily disable exceptions that also access the same data so that they will remain pending until we are done with this data.

Disabling interrupts

On a single processor system, we can achieve sufficient protection here by disabling interrupts while we are accessing data that is shared with the interrupt routine. We can accomplish this using "irq_lock" function that returns a value that we can then use during unlocking. This ensures that we can disable interrupts repeatedly and maintain correctness in the code, we save the value of whether interrupts are disabled or enabled into a stack variable called 'key'.

The implementation of irq_lock would typically look like this:

```
static ALWAYS_INLINE unsigned int arch_irq_lock(void){
    unsigned int key;

    __asm__ volatile("mrs %0, PRIMASK;"
      "cpsid i"
      : "=r" (key)
      :
      : "memory");
    return key;
}
```

This can also be achieved by setting BASEPRI to default interrupt priority and saving the original value. The end result is as intended because highest "application level" interrupt priority will be the default. In the case of using BASEPRI we will retain the ability to run higher level interrupts but synchronization will not work for these interrupts so they must not share data with application code or any of the application level interrupts (application level interrupts simply means that these exceptions are considered "good citizens that are configured in such a way that they obey by interrupt masking rules).

Regardless of which way we choose, the end result is that all exceptions that may share data with application code are disabled. Keep in mind that this means that while interrupts are disabled, any events that arrive will be deferred until interrupts are enabled again. This is important and should be handled by only locking minimum memory access necessary using this approach.

If we now update our original example with interrupt locking, our code will look like this:

```
void interrupt_routine(void){
    data.foo += 5;
}
```

```
void data_user_code(struct data *self){
    int tag = irq_lock();
    self->foo += 8;
    irq_unlock(tag);
}
```

At this point, once we have locked the interrupts, the addition of 5 to the variable will never occur (if interrupt is executing then by definition interrupts have not been locked otherwise it will not be able to execute in the first place). So we can now safely add the 8 to the memory location and then enable interrupts again.

If the exception does occur while interrupts are locked, then 5 will be added to the variable as a separate operation after interrupts are unlocked again (or before interrupts are locked). Either way, the result will be correct.

The biggest problem with this approach is that interrupts are only disabled on a per-core basis. This means that if our code is ever to work correctly on a system with more than one core then we need a better approach. The solution is using atomic operations and using memory as a way to synchronize our cores and ensure that we maintain our requirement of "only one thread on one cpu returning from locking operation and proceeding into the critical section at a time".

Spinlock

The generic solution to synchronizing our code with exceptions is a spinlock. It is called a spinlock because on a multiprocessor system it will loop until it is able to set a special atomic variable successfully.

A spinlock basically combines the interrupt locking we have seen so far with atomic "compare-and-swap" to ensure that we can synchronize our cores.

- **Locking interrupts.** This solves the concurrent data access problem between exceptions and lower priority code just like before.
- **Atomic compare-and-swap (CAS).** This operation is used inside an infinite loop to compare a variable to an expected value and set it only if the comparison is successful. If comparison is unsuccessful then the code loops and retries the same operation until it succeeds.

The atomic operation executes the following logic but does so atomically (ie the instruction can not be interrupted):

- Store an atomic value as part of the spinlock initially set to a known state like zero.
- To synchronize with other CPUs, compare this value to expected value - ie zero - and set it to one if and only if it was still zero.
- Return the new value if comparison was successful or the comparison value if the comparison was unsuccessful.

This means that if we were to do this operation exactly at the same time on two CPU cores against the same memory location then on one CPU the operation would return new value stored and on the other it would return the value we are comparing against.

The result is that we can now loop this operation until the variable has the expected value and once the instruction is executed, the variable will have a new value so that all other attempts will fail until the value has been reset (ie unlocked).

Rules

There are a few rules to adhere to when you use spinlocks.

Locking a spinlock is valid if:

- **We are not trying to acquire the same lock again**: while locking multiple locks is fine, we can not acquire the same lock recursively because that would result in a deadlock.
- **The lock we are trying to acquire is not already held by current CPU**: because acquiring the same lock by the same CPU twice would result in a deadlock.

Unlocking is valid only if:

- **The lock is held by current CPU**: unlocking an already held lock from another CPU is entirely wrong.

Code

A spinlock is implemented using this concept as follows:

```
struct k_spinlock {
    // only needed on multiprocessor systems!
    atomic_t locked;
};

static ALWAYS_INLINE k_spinlock_key_t k_spin_lock(struct k_spinlock *l)
{
    k_spinlock_key_t k;

    // lock interrupts and return current mask (so that nested locking works)
    k.key = arch_irq_lock();

    // only needed on multiprocessor (SMP) systems
    // compare "locked" to zero and if true set to 1 and return previous value (0)
    // this compare and swap is implemented using atomic CPU instruction
    while (!atomic_cas(&l->locked, 0, 1)) {}

    return k;
}

static ALWAYS_INLINE void k_spin_unlock(struct k_spinlock *l,
                    k_spinlock_key_t key)
{
    // clear the lock (only needed for SMP!)
    atomic_clear(&l->locked);

    // unlock interrupts
    arch_irq_unlock(key.key);
}
```

That’s it.

We can now use this spinlock to secure exclusive access to any data that we share with interrupt handlers on our system.

We can also use this spinlock to build higher level synchronization primitives that are meant to be used in RTOS context - such as semaphores, mutexes and other even higher level primitives.

Best Practices

- **Use spinlocks even on bare-metal**: even if you are doing bare metal development without an RTOS, spinlock is a valuable abstraction that makes your code more robust. You can easily port spinlock-based code to a system with multiple cores or to a system that uses an rtos - as long as your spinlock implementation itself is portable.
- **Lock data, not code**: this is a central rule that is especially important with spinlocks because they delay handling of all incoming events. The critical section locked by a spinlock should be very small and only contain memory operations and preferably no calls to other functions.

Common Pitfalls

- **Locking across functions**: when you hold a spinlock across function call, you are always introducing a risk. The risk is that the function you are calling is later modified and would take longer time than necessary. Since you are holding a spinlock, ALL incoming events are being delayed while you are holding it. This means that you can easily degrade the realtime performance of your system.
- **Often little sense in multiple instances**: since a spinlock is so strict at protecting the critical section that nothing else can interrupt it, it is often a waste of memory to have a separate spinlock for multiple instances of an object of the same class. This is why spinlocks in Zephyr often are implemented as static global variables and shared among instances of the same type.

Alternatives

- **Raw interrupt disable**: this works but only on single core systems. On multi-core system this must be implemented as a spinlock to guard against access to data by multiple CPUs because disabling interrupts is always only local to one CPU.
- **Scheduler locking**: a slightly different approach is to lock the whole scheduler. While this protects against being interrupted by other threads, it does not protect against interruptions by interrupt handlers which is the kind of protection that the spinlock is explicitly designed to provide.

For the purpose of synchronizing data access at the level closest to hardware between CPUs and interrupt handlers, the spinlock doesn’t really have any other alternative that is as simple and lightweight.

Conclusion

In this section we have looked at the interface between software and hardware and ways we can synchronize access to variables between code that is directly invoked by hardware and the code that is part of our software.

On top of a spinlock we can now implement all other patterns of synchronization and threading. Even an RTOS scheduler must use spinlocks to ensure that it is fully thread safe.

Quiz

- What main problem is the spinlock abstraction designed to solve?
    1. It is designed to prevent concurrent access to data by a hardware interruption - be it another CPU running any code or current CPU running an interrupt handler.
    2. It is designed only for synchronized access between threads.
    3. It is designed only for multiprocessor systems.
- Why does modern implementation of spinlock always disable interrupts on current CPU before trying to acquire the lock?
    1. Because not doing so would stall the CPU
    2. Because not doing so would allow the critical section to be interrupted by local interrupt handlers.
    3. Because not doing so would allow the critical section to be interrupted by local interrupts as well as other CPUs.
- Why is it important to save original interrupt state (when disabling interrupts) into a 'key' variable on stack when you acquire a spinlock and then to restore interrupt state to the saved state after releasing the spinlock? What does it ensure?
    1. Because it is important to enable interrupts after spinlock is unlocked.
    2. Because we don’t know if interrupts were already disabled and we want to make sure we don’t enable them incorrectly. and because we want to allow multiple spinlocks to be held at the same time
    3. Because we want to restore the interrupt state to exactly the same state it was in before so that we can easily allow recursive locking of the same spinlock as well as making sure that we don’t enable interrupts again incorrectly.
- Why is it important not to hold spinlocks across function calls at all?
    1. Because functions can enable interrupts and our code will not function correctly.
    2. Because we don’t know what the other function is going to do and implementation can evolve without awareness that the other function was called with the spinlock held, causing all kinds of potential issues.
    3. Because spinlocks do not allow any function to be called with the spinlock held.
- What code/operations does locking and unlocking a spinlock essentially translate into on a single core system?
    1. It disables/enables interrupts and spins on an atomic variable.
    2. On single core systems it simply translates to interrupt disable/restore operation.
    3. It spins on an atomic variable.
- Is it fine to attempt to lock another spinlock while holding a different lock?
    1. No. This is strictly forbidden.
    2. Yes it is fine, as long as it’s not the same spinlock.

3. It depends on what we want to accomplish.

- Why should you always use spinlock pattern instead of low level interrupt enable/disable even if you don’t have direct plans to support multi-core systems?
    1. Because it prevents deadlocks.
    2. Because all code in an embedded firmware must always be concurrency safe.
    3. Because it ensures that your code will remain safe on multi-core systems just the same without any modifications to the code itself.
- Why is it completely wrong to compare spinlocks to any other "thread aware" synchronization primitives?
    1. Because spinlocks have no concept of threads. In fact spinlocks can be used without any scheduler.
    2. Because spinlocks protect against concurrent access between CPUs and not other code.
    3. Because spinlocks allow other threads to be interrupted by interrupt handlers if they run on a second CPU core.