Concurrency Pattern

Concurrency is a design tool that we use to enable our software to make the most use of CPU resources so that it can meet deadlines without having to worry about missing events while executing complex tasks.

Concurrency is a way to manage what the processor is doing at any given time without having to make decisions about this everywhere in the code.

Threading, scheduling etc are all abstractions that take the decision making out of the code and place it on the hardware and software specifically designed to make these decisions - ie the RTOS scheduler driven by hardware timers and interrupts.

This section is an introduction a preparation covering various forms of concurrency so that higher level patterns related to concurrency will be easier to understand as you go through this training.

Defining Characteristics

Concurrency is the ability to respond to multiple incoming external and internal events quickly, with minimal delay and in some cases in parallel if the hardware allows it. On systems with multiple physical CPUs it also means ability to do multiple things at once and the software patterns that enable one to do it safely.

On single processor systems concurrency simply ensures that we can manage the CPU as a resource and load it with appropriate code at any given time so that it can do the actions necessary to satisfy all timing constraints.

We then structure our software in such a way that we can give hints to the system scheduler about which tasks should be run when. We do so using task priorities and synchronization primitives.

A system that uses concurrency will always have the following parts:

- **Scheduler**: this is the code that has a higher level view over what the CPU is doing and is able to direct the CPU to run any other portion of the software.
- **Hardware Support**: the scheduler almost always relies on hardware timers to drive it. The hardware is a very important aspect in concurrency which must be taken into account since at the core it is always the hardware that makes scheduling possible.
- **Synchronization**: this consists of software constructs that can be used in the application to control what the scheduler will do next. These consist of spinlocks, semaphores, mutexes, conditional variables and other higher level concepts.

Use Cases

Hardware, by its nature, is always concurrent - so is the rest of the physical world. However, when we are writing instructions for one or more calculators (processors) we are forced to work in a linear fashion - one calculating unit is only able to run things in a sequence.

All concurrency that we implement in software is therefore always a way to mimic the concurrency of the real world and we do it to make the most use of the resources available to us.

Examples include:

- **Parking CPU while waiting for IO**: if our calculation requires an IO value that is not available yet (availability can be signalled using an electrical signal). In such a scenario it is a good idea to load the CPU with a different state and let other calculations happen while we are waiting.
- **Concurrent hardware operations**: each peripheral on a SoC is an independent hardware unit driven by a shared clock. This hardware unit can do things independently of other hardware units. However, to make use of this functionality we need a way to jump between multiple code paths that execute operations in sequence on each piece of hardware. We need concurrency in order to do this.
- **Multiple CPU cores**: we need concurrency in software when we have multiple cores that can access the same peripherals (or memory) and we need ways to synchronize this access in software.
- **Real-time timing constraints**: we need our software to respond to events very quickly and we want to be able to start complex chains of software events when a hardware event occurs. We need concurrency in order to switch contexts quickly in response to hardware events without the need of running some code to completion.

Benefits

The opposite of concurrency is a single, main loop, polling application. This application does not use interrupts, does not use task scheduling and is simply running one main loop over and over again at maximum speed to try to keep up with what is happening. This is horrible.

Concurrency solves these issues.

- **Low power consumption with low CPU load**: CPU load is measured by looking at how much time the CPU spends in 'wait for event' state. This state is entered by executing a CPU instruction (WFE/WFI) which halts the CPU and waits for an interrupt or some kind of event. Many CPUs also have power management functionality where you configure what low power mode the CPU can enter into once it executes the WFE instruction. This is impossible to achieve when your application must poll for changes. Concurrency eliminates need for polling and your application can be event based.
- **Separation of tasks by time domains**: applications often consist of sequential tasks which involve a lot of waiting for IO. We can of course make our code into a state machine (making the sequential task much less obvious) or we can decide to keep our code sequential but separate the different sequences of operations into multiple threads. Concurrency shows us how to do it properly. A time domain is simply a sequence of actions with delays in between that must be executed as a single block. Delays typically occur naturally by having to wait for IO to complete.
- **Basis for many other patterns**: if we want to implement an event processor or async/await pattern in our code, this becomes a lot easier if we have a scheduler that we can use for separating one state machine completely from another and running these step machines as two separate threads.

- **Timeouts and repetitive tasks**: hardware only has a limited set of timers. We need concurrency to have support for flexible and unlimited delayed tasks. We need concurrency patterns in order to implement these things properly along side of interrupts.

Drawbacks

Concurrency has the main drawback of complexity followed by memory usage. Both of which are very easy to deal with.

- **Complexity**: yes it is complex to have to deal with synchronization primitives, threads and interrupts on top of that. But this is how our hardware works and this is also how our physical world works. This complexity is absolutely necessary at system level to be able to transcend hardware limitations in software (such as having the ability to instantiate infinite number of software timers using only one hardware timer).
- **Memory**: a realtime scheduler can be implemented using 10-20KB of flash memory. If your application has 128KB of flash or more then you should be using concurrency because your code will be simpler and more responsive. If you have less than that then concurrency will come at a price of leaving too little space for the actual logic of your application. Unless you are trying to write firmware for devices with ridiculously little space, you should always use an RTOS and use concurrency where appropriate in your application.
- **Performance**: context switching takes time and if we have a lot of threads then our software will be slower due to excessive context switching. It is important to manage thread priorities to keep context switching to a minimum.

Implementation

Concurrency comes in several main forms:

- **Hardware concurrency**: this concurrency is a direct consequence of the hardware being able to toggle unlimited number of signals on a single clock cycle. Things do happen in parallel in the processor and your job as a programmer is to make the best use of it. Hardware concurrency is mostly invisible for you but if you can understand where it is happening and can use it in your software (for example by starting multiple DMA transfers at once or doing transmissions in parallel) you can tremendously improve the performance of your application. Hardware concurrency also applies to multiple CPU cores executing code instructions in parallel.
- **Interrupt concurrency**: this level of concurrency is tied to the hardware support for interrupts. When an interrupt signal (electrical level) arrives at an interrupt input of the CPU, the CPU can be configured to save all main registers to memory (to stack of currently running code) and branch off to a separate location (an interrupt handler). You can configure which function will run by placing it into hardware interrupt vector table and then handle the event in that function (in C).
- **Software concurrency**: this level is supported by special interrupts (PendSV and SVC on arm) and is able to switch software contexts (threads) when instructed to do so either by

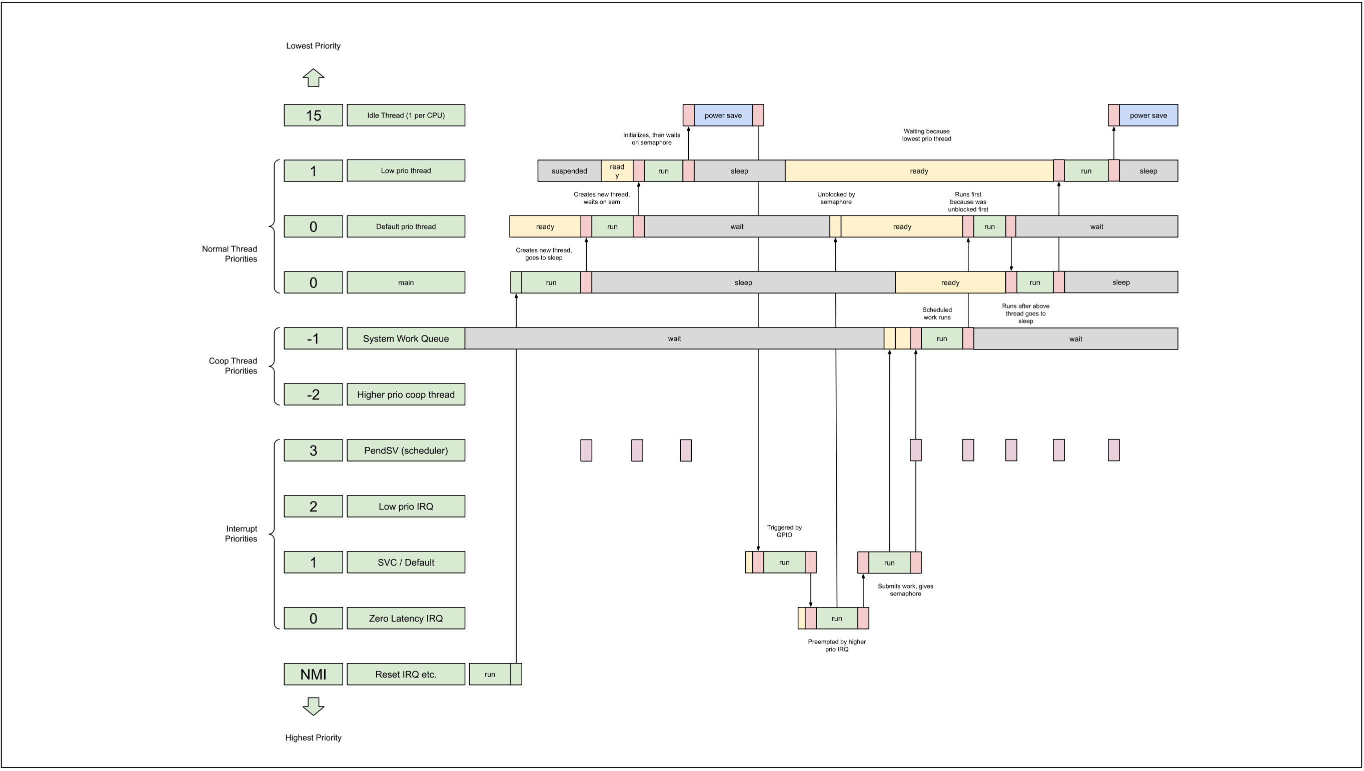
a timer interrupt or by an explicit CPU instruction (you 'pend' the switch and then it happens when CPU considers it safe to do it). This kind of concurrency extends into the realm of threads, synchronization primitives, async/await idioms, work queues and other software only concepts.

When you deal with software development outside of embedded field, you often do not have to deal with threads at all. They are still there, it’s just that at the level of abstraction where you are coding (a standalone linux process) you don’t need to worry about threads. You just create multiple processes and have them communicate over network or something else. However, when we deal with embedded systems, we are the people that make this kind of simplicity possible. So we have to deal with concurrency almost every day and everything we are doing must maximize efficiency using concurrency. We must always write efficient and safe code that can then be used to drive higher level concepts like async/await or state machines.

Concurrency always starts at the hardware level, then goes through interrupts (which connect hardware level to code level through explicit hardware support - the vector table) and then continue into software based abstractions until it is completely abstracted away behind language concepts.

To understand concepts like threads and mutexes we therefore must first understand what we are trying to do (deal with natural parallelism in the physical world - hardware) and why we need these concepts (to break away form the limitations of interrupt concurrency). We must therefore understand hardware concurrency and interrupt concurrency first.

Multiple Priorities



Architecture Simplified

Your hardware consists basically of a CPU core connected to memory that fetches and executes instructions from memory and reads/writes other memory locations which can be connected to other peripherals.

If you care to try setting up a new architecture in Renode, you will find that you will start by creating a cpu and then continue by creating an interrupt controller followed by other peripherals:

```
cpu: CPU.CortexM @ sysbus
    cpuType: "cortex-m3"
    nvic: nvic

nvic: IRQControllers.NVIC @ sysbus 0xE000E000
    priorityMask: 0xF0
    systickFrequency: 72000000
    IRQ -> cpu@0

exti: IRQControllers.EXTI @ sysbus 0x40010400
    [0-6] -> nvic@[6-10, 23, 40]

gpioPortA: GPIOPort.STM32F1GPIOPort @ sysbus <0x40010800, +0x400>
    [0-15] -> exti@[0-15]
```

The CPU is just a state machine that fetches instructions from memory and runs them. All other peripherals are connected to the memory bus and respond to memory read/write requests done by the CPU. In a simple CPU without separate clock domains, all peripherals are clocked by the same clock and run completely in parallel with each other and with the CPU.

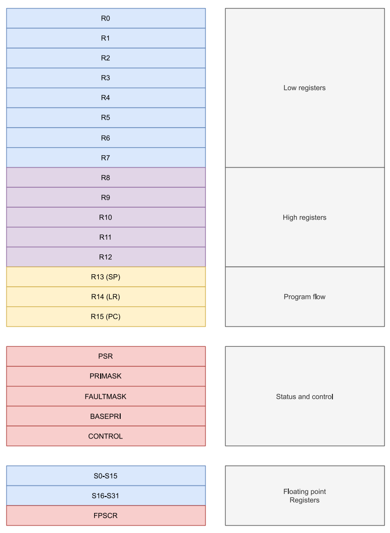
When a peripheral needs to send an event to the CPU, it does so by signaling an interrupt line. The CPU state machine then jumps to the interrupt code and runs your C code.

When we interact with peripherals from our software, we simply read/write memory locations and respond to interrupts.

To make full use of the natural hardware concurrency, we must have a way to structure our code such that we can easily start multiple hardware operations concurrently and have a way to respond to results being ready concurrently as well.

To make this possible, we must have a way of building on the existing interrupt functionality so that our software can accommodate for things happening out of sequence.

CPU Registers



Each CPU core has a set of registers that define current state of the CPU core and which instruction it is going to fetch next. Context switching means to save and restore this state to control where the CPU will execute the next instruction.

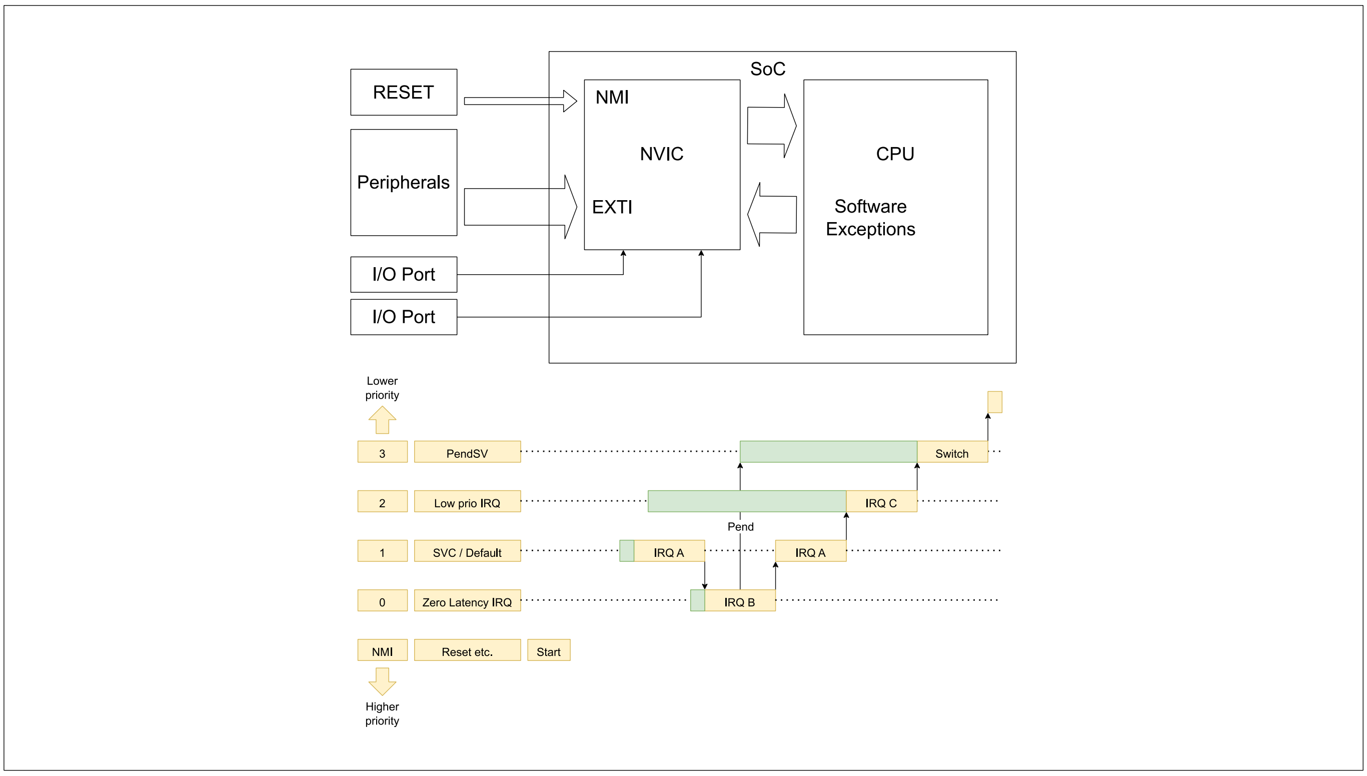
All concurrency uses this type of context switching. Next we are going to look at interrupt concurrency where this context switching is done in hardware and after that we will look at software concurrency where this context switching is done in software.

Interrupt Concurrency

The interrupt controller is a separate peripheral that listens on many incoming interrupt signals and directs the CPU to jump to an interrupt handler when an interrupt occurs. Interrupt controllers often support nested interrupts where preemption is controlled by interrupt priorities - interrupts with higher priority can interrupt other interrupts of lower priority and so on.

Interrupts provide a very crude way of multitasking where you can easily respond to events from the hardware but you can not easily implement complex operations. You can not execute sequences that must sleep from inside an interrupt - this means that any work that may need to wait for IO or sleep must be 'deferred' and executed once the CPU returns from the interrupt handler.

We can deal with this to a degree by for example implementing a low priority timer driven interrupt handler that executes queued tasks and then defer work to that interrupt from higher priority interrupts. The problem with this approach is that we are very limited in how we can jump in the code. We can not for instance jump dynamically after an interrupt is finished to another location where more work needs to be done with the result obtained in the interrupt. We just don’t have this functionality with hardware interrupt support. To obtain it we must implement a software scheduler.



Software Concurrency

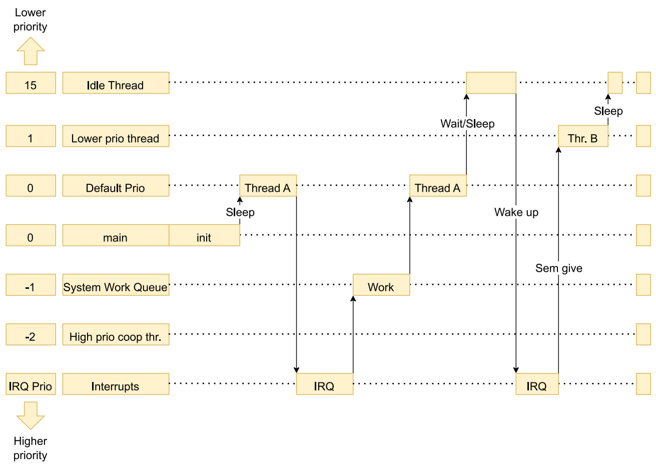
Software concurrency builds upon the interrupt concurrency and adds the ability to jump dynamically to any location in the software (switch context) in response to hardware events. Which location to jump to is controlled using synchronization primitives like mutexes and semaphores. To the programmer, this abstraction provides a clean and simple way to write code while also retaining the power to respond effectively to any kind of events.

On top of software concurrency we can then implement any other concept that we have in our operating systems and computer software. Things like async/await are based internally on a work queue and the work queue is executed by a thread and the thread is a context which can be switched in and out.

When one thread needs to wait for a signal, it waits for a semaphore. When another thread or interrupt code unblocks this semaphore - the thread that was waiting can be switched to immediately. We just don’t get this kind of flexibility with hardware interrupts alone.

The reason we need software concurrency is that we need to be able to implement higher level concepts like software timers and threads. We can only have a limited number of interrupts but an application should be able to create any number of execution threads or software timers without being limited. We have to implement this in ways that do not require the application code to be aware of the existence of other threads at all. Preferably it should not be aware of existence of threads at all unless it explicitly needs to create threads.

When we have software concurrency, implementing things like power saving modes becomes a simple matter of registering generic handlers that are called whenever no thread is ready to run and before the system executes WFE instruction. From application perspective we can now offload all of these details to the RTOS code and this code in turn runs in response to hardware interrupts so it becomes completely transparent to our application.



Best Practices

- **Separate by 'Time Domains'**: when thinking about which tasks you want to split into multiple software threads, think of it in terms of which sequences of operations you would like to interleave. An interleaved sequence means that in code you write the code sequentially, but when you code is waiting for IO then other code can run (be interleaved) with your code. Threading is about optimizing the utilization of the CPU and the splitting of code into tasks should be primarily driven by the necessity to separate sequences of tasks into isolated blocks of code that can be maintained separately.
- **Interrupts only for loading hardware**: use interrupts only for operating on prepared data that needs to either be pumped out through the hardware or data needs to be received from hardware and placed into a prepared buffer. Interrupts are not for application logic. Interrupts must only operate on the premise of responding to a hardware event and quickly preparing next operation if appropriate. If your task is above the lowest level of abstraction (ie a GPIO operating over an I2C gpio expander) then you need to defer this work into the system work queue or another thread.
- **Utilize system work queue**: the system work queue provides a way of queuing cooperative blocks of operations which are then executed in sequence as part of the work queue thread loop. You can queue work from interrupts or from other threads. This is an excellent way to avoid creating unnecessary threads in your application.

Common Pitfalls

- **Do not reinvent the scheduler**: You should be using an existing scheduler/RTOS like Zephyr. Do not try to invent your own scheduling. This problem has been thoroughly solved many many times since the concept first appeared in 1980s. Any solution that you implement yourself will be a nightmare to maintain.

- **Do not use threads as level of abstraction**: one of the worst misuses of software concurrency is using threads as ways of "layering abstractions". For example, you have a uart 'low level thread' which reads characters and places lines into a queue. Then you have your main application thread reading from that 'line queue'. This is abstraction layering and it is almost always a mistake. The intermediate thread is completely unnecessary and just wastes resources. Threads are for 'time domain separation' - not for layering abstractions.

Alternatives

- **Interrupt driven concurrency**: you have seen how the interrupt controller already provides us with a way of nesting tasks of different proorities. Simple concurrency can thus be achieved using only the interrupt controller. However, such concurrency is extremely limited because it does not gives us a way to jump to an arbitrary location at any given time.
- **State machines**: Run to completion state machines are another approach. This approach can be used together with conventional threading in order to create more reliable software because a state machine can be mathematically proven. The biggest drawback is that state machines need to run to completion. Preemption is only possible if you have a conventional scheduler as well.

Conclusion

This module has been a brief introduction to concurrency and the layers below it that connect concurrency to the hardware. It is designed to pave the way towards understanding higher level software concurrency concepts such as spinlocks, semaphores, mutexes and conditional variables.

Quiz

- What is the main reason that interrupt based concurrency is not enough for having full flexibility when writing higher level applications? What limitations does it have?
    1. Interrupt based concurrency can only jump to another location upon receiving an event. It can not jump with a software instruction.
    2. Interrupt based concurrency is only suitable for interrupts.
    3. Interrupts can not be prioritized.
- How does software concurrency enable you to make the most use of your CPU without requiring major changes to your code structure and software architecture?
    1. It enables the code to trigger interrupts at any given time and do work in the interrupt handler.
    2. It enables us to jump each CPU to any location in code as needed without having to wait for completion.
    3. It prevents the CPU from wasting cycles by going to sleep.
- How would you typically structure your priorities for maximum flexibility?
    1. Main thread at the highest priority, system work queue at lowest priority and interrupts in the middle.

2. Interrupts at highest priority, system work queue at highest thread priority, then main thread, then all other threads.
3. System work queue at highest priority, then interrupts, then main thread and then all other threads.

- What happens to an interrupt handler if another lower prority interrupt is triggered?
  1. Current interrupt is preempted, the other interrupt executes and then execution goes to the current interrupt.
  2. Current interrupt handler executes to completion, then the new pending interrupt executes.
  3. If the other interrupt is deasserted before current interrupt handler completes then the other interrupt never runs.