Zephyr™ Project
Developer Summit
June 8-10, 2021 • @ZephyrIoT

Real Time in the Real World

SCHEDULER DETAILS FOR PRACTICAL PROBLEMS

Zephyr™ Project
Developer Summit
June 8-10, 2021 • @ZephyrIoT

Andy Ross

SOFTWARE ENGINEER – INTEL

ANDREW.J.ROSS@INTEL.COM
ANDYROSS @ZEPHYRPROJECT.SLACK.COM
ANDYROSS @GITHUB

"Real" Time Zephyr

- What is an RTOS, really?
- Mostly about doing the right thing at the right time
- But that's an application decision, right?
- Zephyr is an RTOS
  - Because it gives an application tools to run the right code at the right time
  - Some of that is out of scope
    - Interrupt facilities
    - Driver models
    - Timing subsystem
- Today we want to talk about how the kernel decides what code to run

Scheduling!

- Scheduling is a decision made by the OS about which code to run
- It's deciding what to do NOW, on this CPU
  - Not deciding what to do LATER, that's timing
  - Not deciding what to do WHEN, that's interrupts
  - Not deciding what NOT to do UNTIL, that's interprocess communication
- Core scheduling metaphor:
  - Application presents a bunch of things that can be run
  - Application provides rules about which things are important
  - OS picks a winner
- So what is this "code" we are scheduling?

Threads!

- Basic unit of runnable Zephyr code
- Contains essential state
  - A stack to run on
    - Multiple kilobytes!
  - Space somewhere to save state
- Also data for the scheduler to use
  - Priority (we'll come back to this)
  - Is it runnable?
  - If not, sleeping? (has a timeout active)
  - Or pended? (waiting on a wait_q)
  - Or suspended? (none of the above)

```
static struct k_thread thread;
k_tid_t t = k_thread_create(&thread, thread_stack, thread_stack_size,
                            thread_fn, arg1, arg2, arg3,
                            thread_priority, 0, K_NO_WAIT);
```

Thread Runnability

- Lots of ways to switch between runnable and not
  - `k_sleep()` puts current thread to sleep on a timeout
    - `k_wakeup()` wakes it up from a different thread
  - `k_thread_suspend()` puts a **different** thread to sleep
  - `k_thread_resume()` wakes it up again
  - `k_yield()` gives up the CPU, but stays runnable
  - Almost every IPC primitive has its own access to these states
    - Semaphores, mutexes, queues, poll, etc...
    - Generally all based on an internal "wait queue" utility
- All these states are just chrome on top of the scheduler
  - All it cares is "is it runnable or not"

Scheduling: When?

- So you have runnable threads, when does "scheduling" happen?
- Simplest answer: any time the set of runnable threads or their state changes
  - If you go to sleep, someone else has to run
  - If you resume a thread, maybe it was higher priority
  - If you change your own priority, maybe it's lower than someone else's
- Complicated answer: depends on IPC implementation
  - Scheduler itself doesn't define schedule points, provides `reschedule()` primitive
  - Different utilities make different decisions
- May happen in a call out of a regular thread
  - Can cause a context switch via `z_swap()`
- May happen in an interrupt
  - ISR returns to a different thread than it interrupted
  - This is "preemption"

Scheduling: Why?

- How to decide which thread is more important?
  - Zephyr's decision is 100% deterministic
  - No dependence on budget/interactivity/fairness/history
- Priority!
  - Priority is just an integer
  - Lower numbers are higher
  - No special limit to the number of priorities available (kconfig defines max number)
    - (Except multiqueue backend)
  - Priorities can be changed at runtime via `k_thread_priority_set()`
- Two kinds of priority
  - Preemptible
    - "Normal" threads
    - Non-negative priority values
  - Cooperative...

Cooperative Priorities

- Priorities with negative values are special
  - They cannot be preempted
    - They can be interrupted, but the interrupt is guaranteed to return back to the interrupted thread
  - They will not context switch because of changes to scheduler state
    - Even if you make a higher priority thread runnable, they keep running
  - They run until they willfully sleep, pend, suspend or yield
    - Thus the name: these threads have to "cooperate" to share the CPU
    - Term comes from ancient OSes that had no ability to preempt code
- Why? Locking!
  - If you can't be preempted, you don't need to lock yourself against other threads
  - Higher performance
  - Easier reasoning about synchronization (fewer bugs)
  - (Doesn't work against interrupts!)
  - (Doesn't work in SMP either!)
- Corrolary: cooperative priorities are never just for "important" code
  - Use preemptible priorities unless you know exactly why you want a cooperative design

Scheduler Locking

- Cooperative priorities act like locks vs. other threads
  - Priorities can be changed
  - So why not use it as a lock?
  - Complicated by `k_thread_priority_set()` being a scheduling point
- We have `k_sched_lock()` / `k_sched_unlock()`
  - Effectively elevates current thread to a cooperative priority
  - Even when there are no cooperative priorities configured
  - Very cheap, even faster than spinlocks
  - Doesn't impact interrupt latency
  - Same warning: doesn't work on SMP!

Another kind of priority

- What if two threads have the same priority?
  - Scheduler runs the first one to have been made runnable
- Alternatively: Earliest Deadline First scheduling
  - Optional feature (`CONFIG_SCHED_DEADLINE`)
  - Associates a "deadline" time with each thread
    - `k_thread_deadline_set()`
    - Same units as `k_cycle_get_32()`, as a delta from "now"
  - Runs the thread with the soonest deadline within its priority
  - Popular for realtime compute workloads
    - It's provably optimal
    - Requires resource management layer that knows how long tasks take!

Yet another: MetaIRQ

- Sometimes interrupts leave work to be done ("bottom half")
  - Hardware can be "re-armed" to do more work
  - But processing needs to happen before we run another app thread
  - Ex.: network device: packet has been received and hardware ready for another
    - But need to parse that packet immediately, might have data for the highest priority thread!
- Another layer of priorities, higher than cooperative: MetaIRQ
  - They will preempt even cooperative threads that "cannot be preempted"
  - Having done so, always return to the **specific** cooperative thread they preempted
  - Otherwise act like preemptible threads
    - Can have more than one level (`CONFIG_NUM_METAIRQ_PRIORITIES`)
    - They can preempt each other
    - If we didn't have cooperative threads, we wouldn't need MetaIRQ
  - Intended for driver layer workloads, not applications

Notes about SMP

- No rules change with more than one CPU
  - But rules apply on **all** CPUs!
  - Changes to scheduler state on one CPU affect the others
    - (Requires an IPI be provided by the arch layer, not always present)
  - As mentioned: breaks the idea of "cooperative" as "locking"
- CPU affinity / "pinning" is possible (`CONFIG_SCHED_CPU_MASK`)
  - `k_thread_cpu_mask` functions manage a per-CPU bitmask for every thread
  - Threads must have a 1 in the mask to run on a given CPU
  - Note: with careful design, this can recover cooperative/sched_lock synchronization
  - Optional feature, and when enabled default is threads run on any CPU
    - Persistent argument about this, other RTOS designs do it differently

Scheduler Backends

- By default, runnable threads live in a simple ("dumb") linked list
  - Small code, O(N), but fast for small lists, generally very few runnable threads
  - No limit on number of priorities (though the data type is a byte!)
  - Required by `cpu_mask` APIs
  - Use this one
- `CONFIG_SCHED_MULTIQ` uses one list per priority
  - Limit of 32 priorities (non-empty lists found via bitmask)
    - No EDF
  - Fastest backend (no list searching needed, O(1) in all cases)
- `CONFIG_SCHED_SCALABLE` uses a red/black tree
  - log(N) behavior always, no matter how many runnable threads
  - Significant overhead, use only if you have a **LOT** of threads

Scheduling other code

- Stacks are really big! Context switching is really expensive!
- Must we schedule only threads?
  - Well, yes, if you want preemptibility
    - Saved state needs to live somewhere
  - What if we give up preemptibility?
- Work queues
  - Work items are just a callback + data pointer (and a timeout)
  - No stack, they share a single thread in the queue
  - First-in/first-out, no priority between items
  - So not really "scheduling", but still an important tool
- What if we want work items **and** preemptibility? ...

P4 Work Queues

- Pooled, Parallel, Preemptible, Priority-based Work Queues
  - New gadget, still maturing
  - Work items are still a callback + data
  - But they also have a scheduler priority and EDF deadline
  - Executed by a pool of threads across multiple SMP CPUs
  - Those threads can be preempted by higher priority items running in different threads
  - No context switch needed in the common case where item runs to completion
    - Just pull the next out of the queue
  - Always runs the highest priority work item
    - As long as you don't run out of threads in the pool! (No free lunch)
  - Best of all worlds?

Scheduler Takeaways

- Zephyr has a bewildering variety of tools, but the basic rules are simple
  - Threads can be runnable or not
  - Scheduler always runs the highest priority runnable thread when it switches
  - Threads can switch whenever state changes (except cooperative (except MetaIRQ))
  - Don't use cooperative priorities to mean "important"
    - Mostly, don't use cooperative priorities unless you really need the locking optimization
    - I'm serious. Don't use cooperative priorities.
    - Yes, this means your app too.
  - Use the complicated tools where they make sense, but leave the defaults alone

Zephyr™ Project
Developer Summit

"There is a saying that every nice piece of work needs the right person in the right place at the right time."

- Benoit Mandelbrot


Zephyr™ Project
Developer Summit
June 8-10, 2021 ▪ @ZephyrIoT