lesson:5

Introduction

What is "scheduling"?

The scheduler determines which thread is allowed to execute at any point in time; this thread is known as the current thread.

There are various points in time when the scheduler is given an opportunity to change the identity of the current thread. These points are called reschedule points. Some potential reschedule points are:

- transition of a thread from running state to a suspended or waiting state, for example by k_sem_take() or k_sleep().
- transition of a thread to the ready state, for example by k_sem_give() or k_thread_start()
- return to thread context after processing an interrupt
- when a running thread invokes k_yield()

A thread sleeps when it volutarily initiates an operation that transitions itself to a suspended or waiting state.

Whenever the scheduler changes the identity of the current thread, or when execution of the current thread is replaced by an ISR, the kernel first saves the current thread's CPU register values. These register values get restored when the thread later resumes execution.

How does the scheduler work?

The kernel's scheduler selects the highest priority ready thread to be the current thread. When multiple ready threads of the same priority exist, the scheduler chooses the one that has been waiting the longest.

Note: execution of ISRs takes precedence over thread execution. Unless interrupts have been masked, the current thread can be replaced by an ISR at any time.

The kernel can be built with one of several choices for the ready queue implementation. The choice is a trade-off between:

- code size
- constant factor runtime overhead
- performance scaling when many threads are involved

Your Kconfig-file (`prj.conf`) should contain one of the following (or it will default to simple linked-list).
The queue types:

- Simple linked-list ready queue (CONFIG_SCHED_DUMB)
    - simple unordered list
    - very fast constant time performance for single threads
    - very low code size
    - useful for systems with:
        - constrained code size
        - small number of threads (<=3) at any given time
- Red/black tree ready queue (CONFIG_SCHED_SCALABLE)
    - red/black tree (wiki)
    - slower constant time insertion and removal overhead
    - requires extra 2Kb code
    - scales cleanly and quickly into many thousands of threads
    - Useful for systems with:
        - many concurrent runnable threads (>20 or so)
- Traditional multi-queue ready queue (CONFIG_SCHED_MULTIQ)
    - classic array of lists, one per priority (max 32 priorities)
    - tine code size overhead vs. the "dumb" scheduler
    - runs in 0(1) time with very low constant factor
    - requires fairly large RAM budget to store list heads
    - incompatible with deadline scheduling and SMP affinity
    - systems with small # of threads (but usually DUMB is good enough)
- Scalable wait_q implementation (CONFIG_WAITQ_SCALABLE)
- Simple linked-list wait_q (CONFIG_WAITQ_DUMB)

How do thread priorities work?

A thread's priority is an integer value, and can be either negative or non-negative. Numerically lower priorities take precedence over higher values (-5>6).

The scheduler distinguishes between two classes of threads, based on each thread's priority.

- A cooperative thread has a negative priority value. Once it becomes the current thread, a cooperative thread remains the current thread until it performs an action that makes it unready.
- A preemptible thread has a non-negative priority value. Once it becomes the current thread, a preemptible thread may be supplanted at any time if a cooperative thread, or a preemptible thread of higher or equal priority, becomes ready.

A thread's initial priority value can be altered up or down after the thread has been started. Thus it is possible for a preemptible thread to become a cooperative thread, and vice versa.

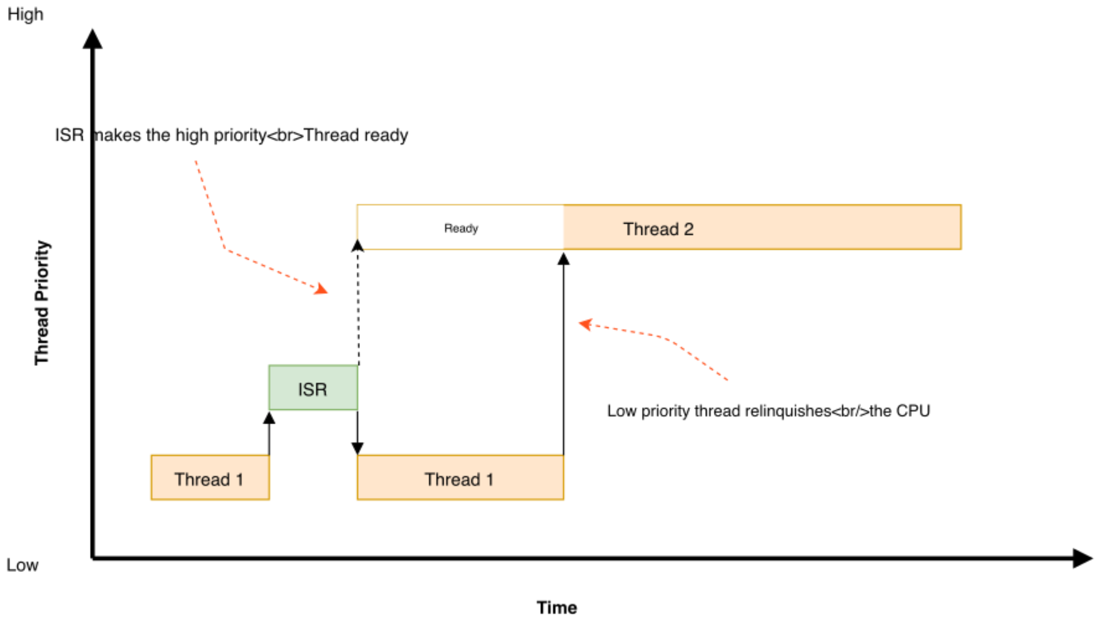
The kernel supports a virtually unlimited number of thread priority levels. The configuration options CONFIG_NUM_COOP_PRIORITIES and CONFIG_NUM_PREEMPT_PRIORITIES specify the number of priority levels for each class of thread, resulting in the following usable priority ranges:

- cooperative threads: (-CONFIG_NUM_COOP_PRIORITIES) to -1

- preemptive threads: 0 to (CONFIG_NUM_PREEMPT_PRIOTIES-1)

What is cooperative time slicing?

Once a cooperative thread becomes the current thread, it remains the current thread until it performs an action that makes it unready. Consequently, if a cooperative thread performs lengthy computations, it may cause an unacceptable delay in the scheduling of other threads, including those of higher priority.

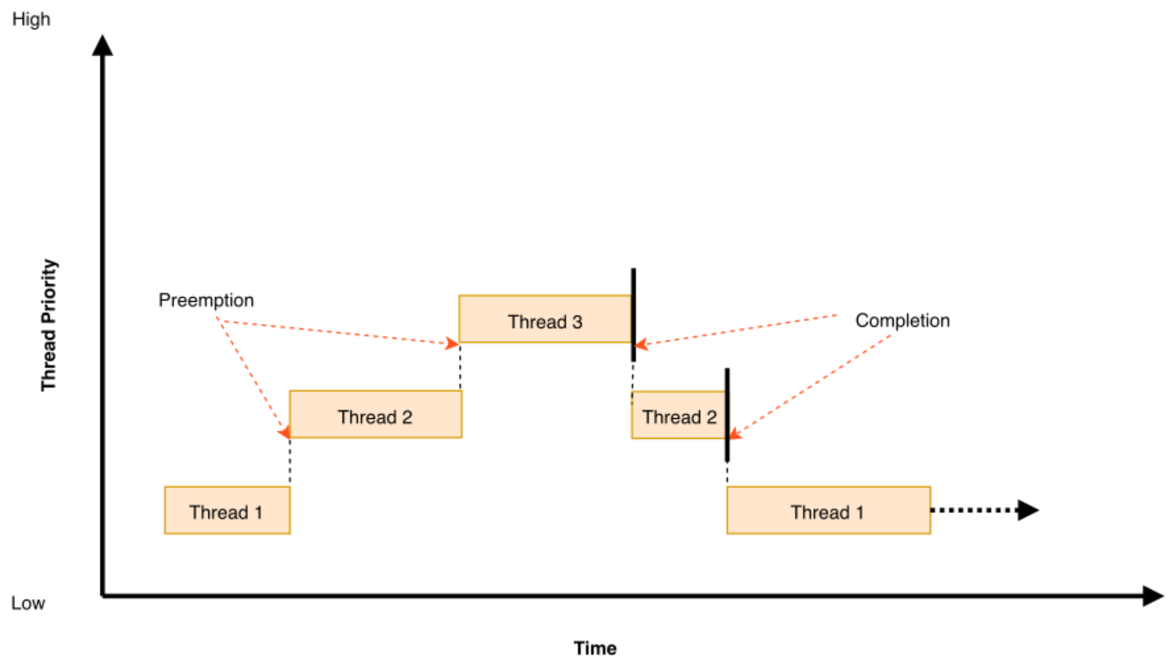


To overcome such problems, a cooperative thread can voluntarily relinquish the CPU from time to time to permit other threads to execute. A thread can relinquish the CPU in two ways:

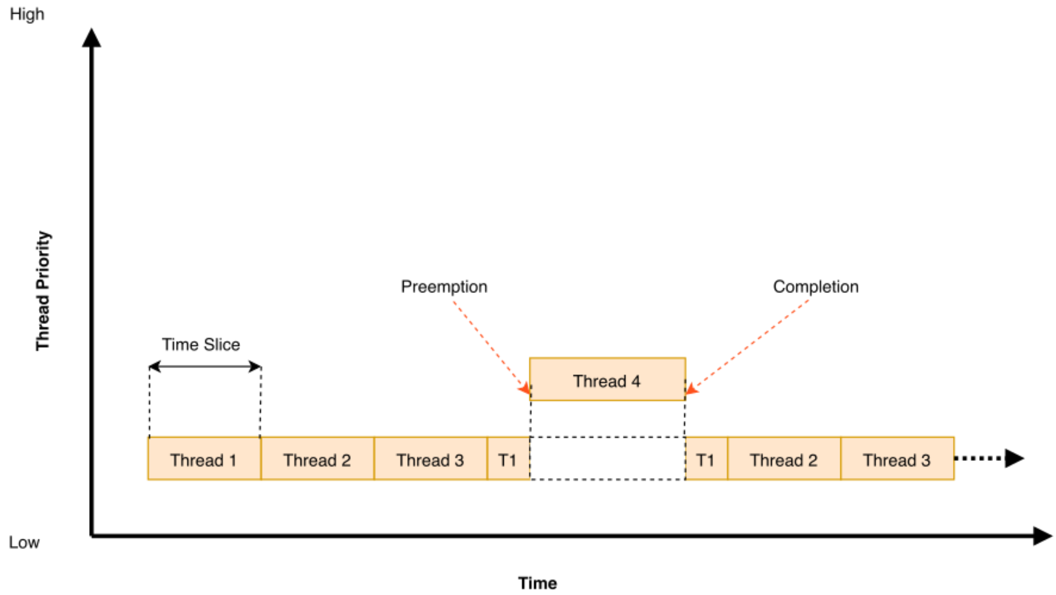
- Calling k_yield() puts the thread at the back of the scheduler's prioritized list of ready threads, and then invokes the scheduler. All ready threads whose priority is higher or equal to that of the yielding thread are then allowed to execute before the yielding thread is rescheduled. If no such threads exist, the scheduler immediately reschedules the yielding thread without context switching.
- Calling k_sleep() makes the thread unready for a specified time period. Ready threads of *all* priorities are then allowed to execute; however, there is no guarantee that threads whose priority is lower than that of the sleeping thread will actually be scheduled before the sleeping thread becomes ready once again.

What is preemptive time slicing?

Once a preemptive thread becomes the current thread, it remains the current thread until a higher priority thread becomes ready, or until the thread performs an action that makes it unready. Consequently, if a preemptive thread performs lengthy computations, it may cause an unacceptable delay in the scheduling of other threads, including those of equal priority.



To overcome such problems, a preemptive thread can perform cooperative time slicing (as described above), or the scheduler's time slicing capability can be used to allow other threads of the same priority to execute.



The scheduler divides time into a series of time slices, where slices are measured in system clock ticks. The time slice size is configurable, but this size can be changed while the application is running.

At the end of every time slice, the scheduler checks to see if the current thread is preemptible and, if so, implicitly invokes k_yield() on behalf of the thread. This gives other ready threads of the same priority the opportunity to execute before the current thread is scheduled again. If no threads of equal priority are ready, the current thread remains.

Threads with a priority higher than the specified limit are exempt from preemptive time slicing, and are never preempted by a thread of equal priority. This allows an application to use preemptive time slicing only when dealing with lower priority threads that are less time-sensitive.

Note: the kernel's time slicing algorithm does not ensure that a set of equal-priority threads receive an equitable amount of CPU time, since it does not measure the amount of time a thread actually gets to execute. However, the algorithm does ensure that a thread never executes for longer than a single time slice without being required to yield.

A preemtible thread that does not wish to be preempted while performing a critical operation can instruct the scheduler to temporarily treat it as a cooperative thread by calling k_sched_lock(). This prevents other threads from interfering while the critical operation is being performed.

Once the critical operation is complete the preemptible thread must call k_sched_unlock() to restore its normal, preemptible status.

If a thread calls k_sched_lock() and subsequently performs an action that makes it unready, the scheduler will switch the locking thread out and allow other threads to execute. When the locking thread again becomes the current thread, its non-preemptible status is maintained.

Note: Locking out the scheduler is a more efficient way for a preemptible thread to prevent preemption than changing its priority level to a negative value.

What is thread sleeping?

A thread can call k_sleep() to delay its processing for a specified time period. During the time the thread is sleeping the CPU is relinquished to allow other ready threads to execute. Once the specified delay has elapsed the thread becomes ready and is eligible to be scheduled once again.

A sleeping thread can be woken up prematurely by another thread using k_wakeup(). This technique can sometimes be used to permit the secondary thread to signal the sleeping thread that something has occured without requiring the thread to define a kernel synchronization object, such as a semaphore. Waking up a thread that is not sleeping is allowed, but has no effect.

What is busy waiting?

A thread can call k_busy_wait() to perform a busy wait that delays its processing for a specified time period without relinquishing the CPU to another ready thread.

A busy wait is typically used instead of thread sleeping when the required delay is too short to warrant having the scheduler context switch from the current thread to another thread and then back again.

Cooperative or preemptive?

- Device drivers and other performance-critical work -> cooperative threads

- Use cooperative threads to implement mutual exclusion without the need for a kernel object, such as a mutex
- Use preemptive threads to give priority to time-sensitive processing over less time-sensitive processing

Commands

A couple of commands we haven't discussed in threads:

- k_yield: Yield the current thread. This routine causes the current thread to yield execution to another thread of the same or higher priority. If there are no other ready threads of the same or higher priority, the routine returns immediately.
- k_wakeup: Wake up a sleeping thread. This routine prematurely wakes up thread from sleeping.
- k_current_get: Get thread ID of the current thread.
- k_sched_lock: A preemptible thread that does not wish to be preempted while performing a critical operation can instruct the scheduler to temporarily treat it as a cooperative thread by calling k_sched_lock(). This prevents other threads from interfering while the critical operation is being performed.
- k_sched_unlock: Once the critical operation is complete the preemptible thread must call k_sched_unlock() to restore its normal, preemptible status.
- k_busy_wait: A thread can call k_busy_wait() to perform a busy wait that delays its processing for a specified time period without relinquishing the CPU to another ready thread.

Scheduler configuration

CONFIG	Description
CONFIG_SCHED_DUMB	The scheduler ready queue will be implemented as a simple unordered list, with very fast constant time performance for single threads and very low code size. This implementation should be selected on systems with constrained code size that will never see more than a small number (3, maybe) of runnable threads in the queue at any given time. On

CONFIG	Description
	most platforms (that are not otherwise using the red/black tree) this results in a savings of ~2k of code size.
CONFIG_SCHED_SCALABLE	The scheduler ready queue will be implemented as a red/black tree. This has rather slower constant-time insertion and removal overhead, and on most platforms (that are not otherwise using the red/black tree somewhere) requires an extra ~2kb of code. The resulting behavior will scale cleanly and quickly into the many thousands of threads. Use this for applications needing many concurrent runnable threads (> 20 or so). Most applications won't need this ready queue implementation.
CONFIG_SCHED_MULTIQ	When selected, the scheduler ready queue will be implemented as the classic/textbook array of lists, one per priority (max 32 priorities). This corresponds to the scheduler algorithm used in Zephyr versions prior to 1.12. It incurs only a tiny code size overhead vs. the "dumb" scheduler and runs in O(1) time in almost all circumstances with very low constant factor. But it requires a fairly large RAM budget to store those list heads, and the limited features make it incompatible with features like deadline scheduling that need to sort threads more finely, and SMP affinity which need to traverse the list of threads. Typical applications with small numbers of runnable threads probably want the DUMB scheduler.
CONFIG_WAITQ_SCALABLE	When selected, the wait_q will be implemented with a balanced tree. Choose this if you expect to have many threads waiting on individual primitives. There is a ~2kb code size increase over CONFIG_WAITQ_DUMB (which may be shared with CONFIG_SCHED_SCALABLE) if the red/black tree is not used elsewhere in the application, and pend/unpend operations on "small" queues will be somewhat slower (though this is not generally a performance path).
CONFIG_WAITQ_DUMB	When selected, the wait_q will be implemented with a doubly-linked list. Choose this if you expect to

CONFIG	Description
	have only a few threads blocked on any single IPC primitive.

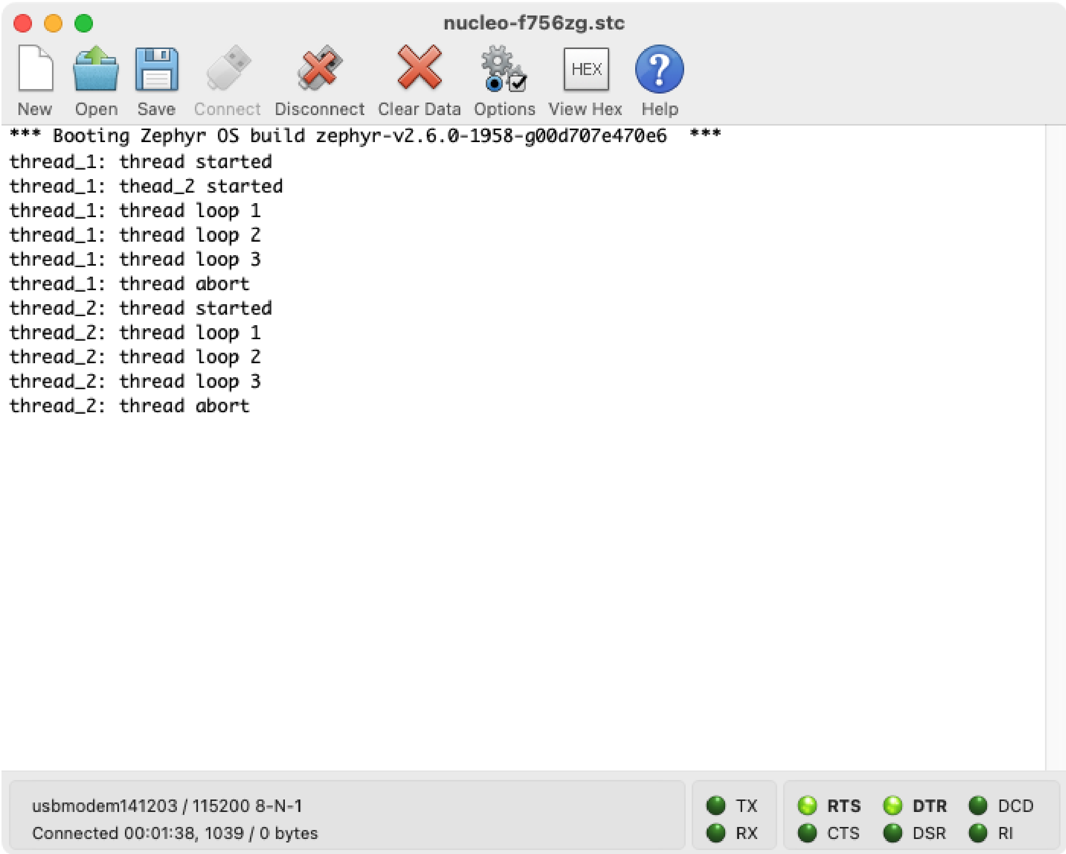
Thread priorities configuration

CONFIG	Description
CONFIG_NUM_COOP_PRIORITIES	Number of coop priorities
CONFIG_NUM_PREEMPT_PRIORITIES	Number of preempt priorities

Exercises

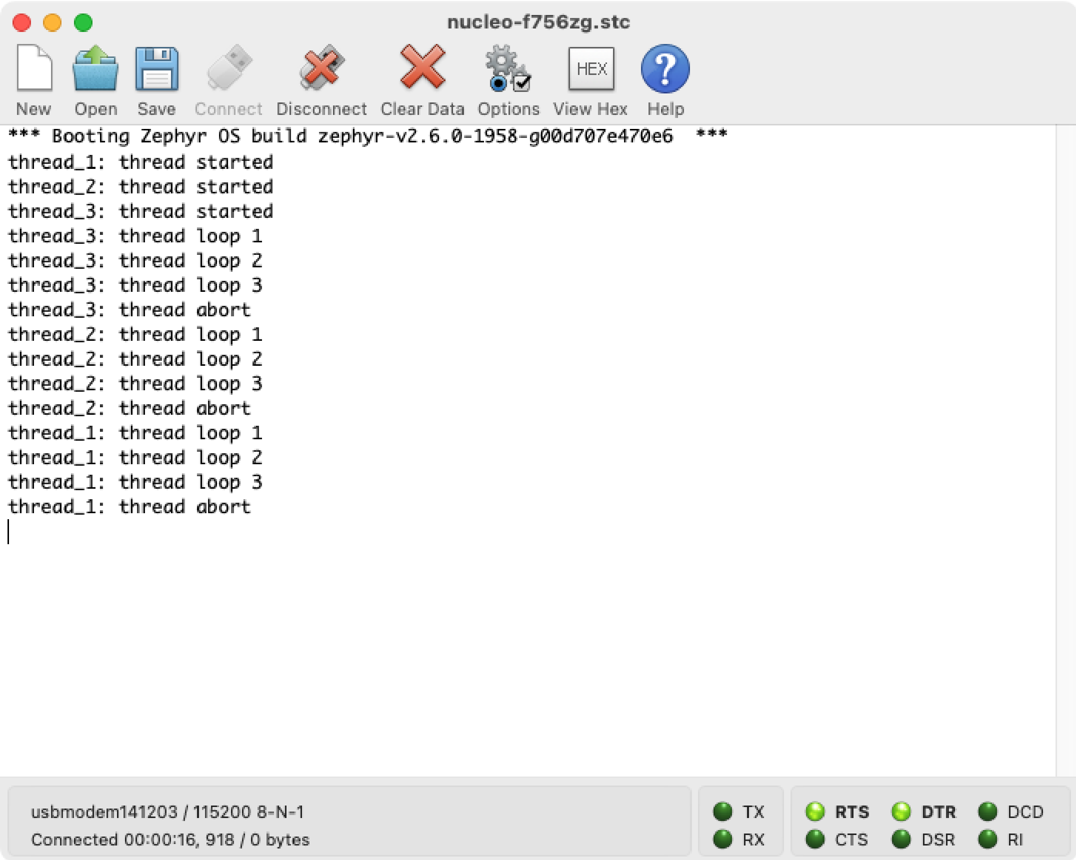
Cooperative Time Slicing

Implement cooperative time slicing that puts out the following serial output (using `printk()`)
figure showing that thread_1, even though lower priority, doesn't yield to thread_2 (until finished)



Preemptive Time Slicing

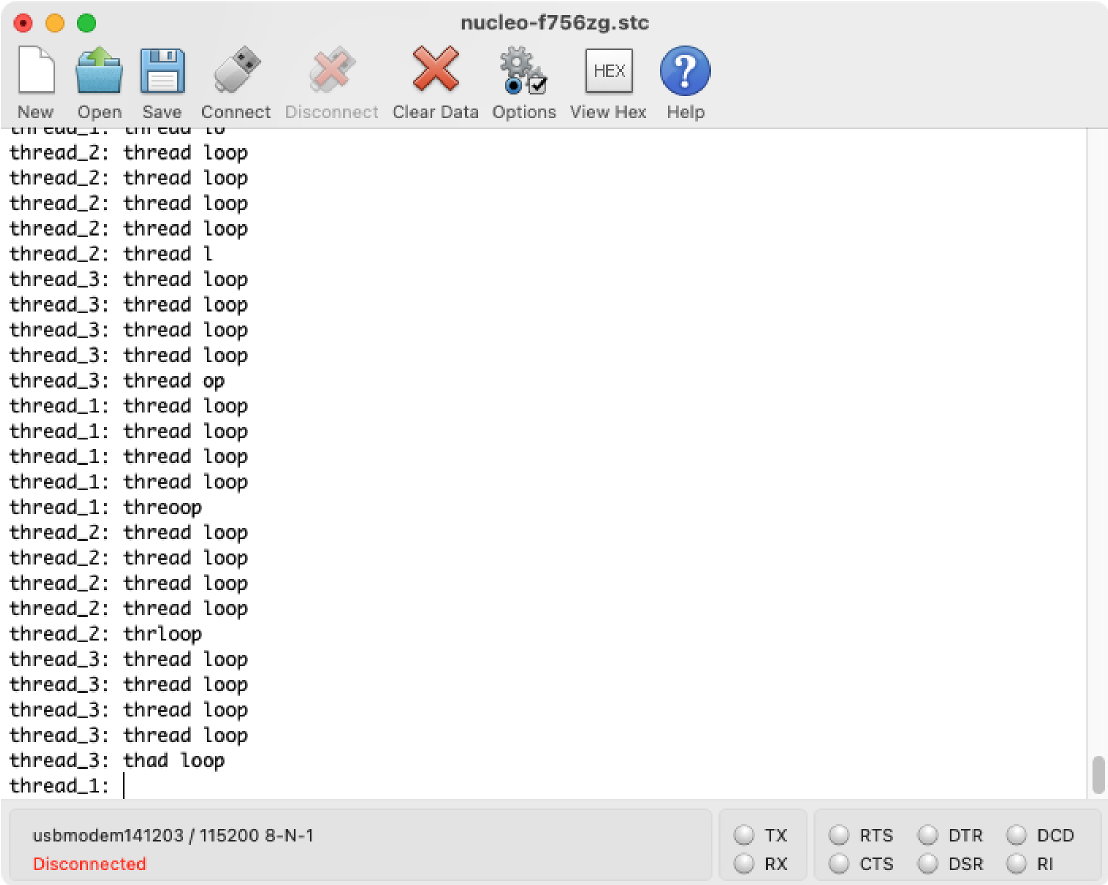
Implement Preemptive time slicing that puts out the serial (using `printk()`)
figure showing the increase in thread priority and preemption and completion of each thread



Time Slicing (with 3 threads)

Implement time slicing with three threads (of equal priority)

figure showing the equal priority threads preempting each other


nucleo-f756zg.stc
New
Open
Save
Connect
Disconnect
Clear Data
Options
HEX
View Hex
Help
thread_2: thread loop
thread_2: thread loop
thread_2: thread loop
thread_2: thread loop
thread_2: thread l
thread_3: thread loop
thread_3: thread loop
thread_3: thread loop
thread_3: thread loop
thread_3: thread op
thread_1: thread loop
thread_1: thread loop
thread_1: thread loop
thread_1: thread loop
thread_1: threoop
thread_2: thread loop
thread_2: thread loop
thread_2: thread loop
thread_2: thread loop
thread_2: thrloop
thread_3: thread loop
thread_3: thread loop
thread_3: thread loop
thread_3: thread loop
thread_3: thad loop
thread_1:
usbmodem141203 / 115200 8-N-1
Disconnected
TX
RX
RTS
CTS
DTR
DSR
DCD
RI