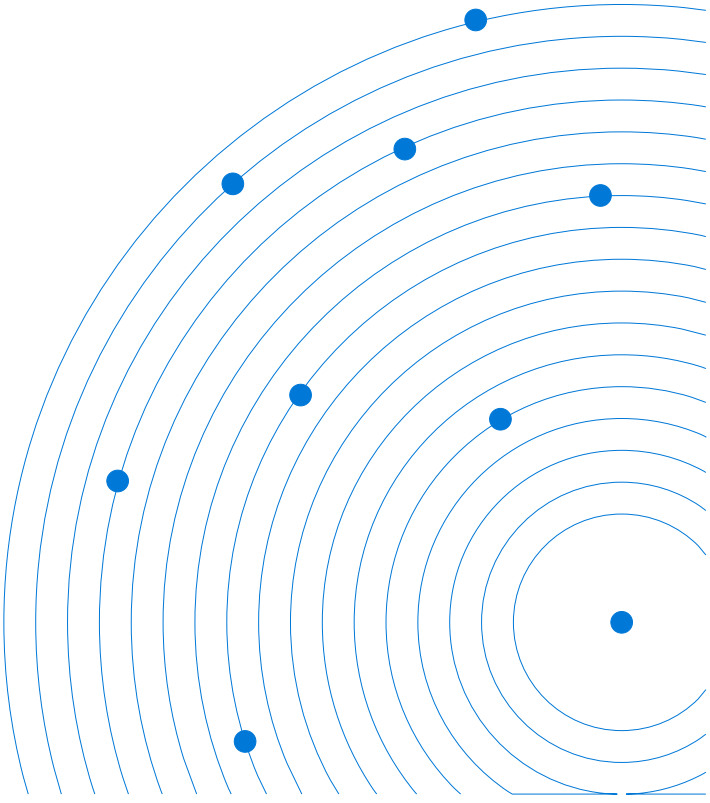
Preemption-Threshold Scheduling Enables Real-Time Systems to Achieve Higher Performance

Innovative real-time scheduling mode permits responsiveness, while protecting critical sections and reducing overhead

January 2020

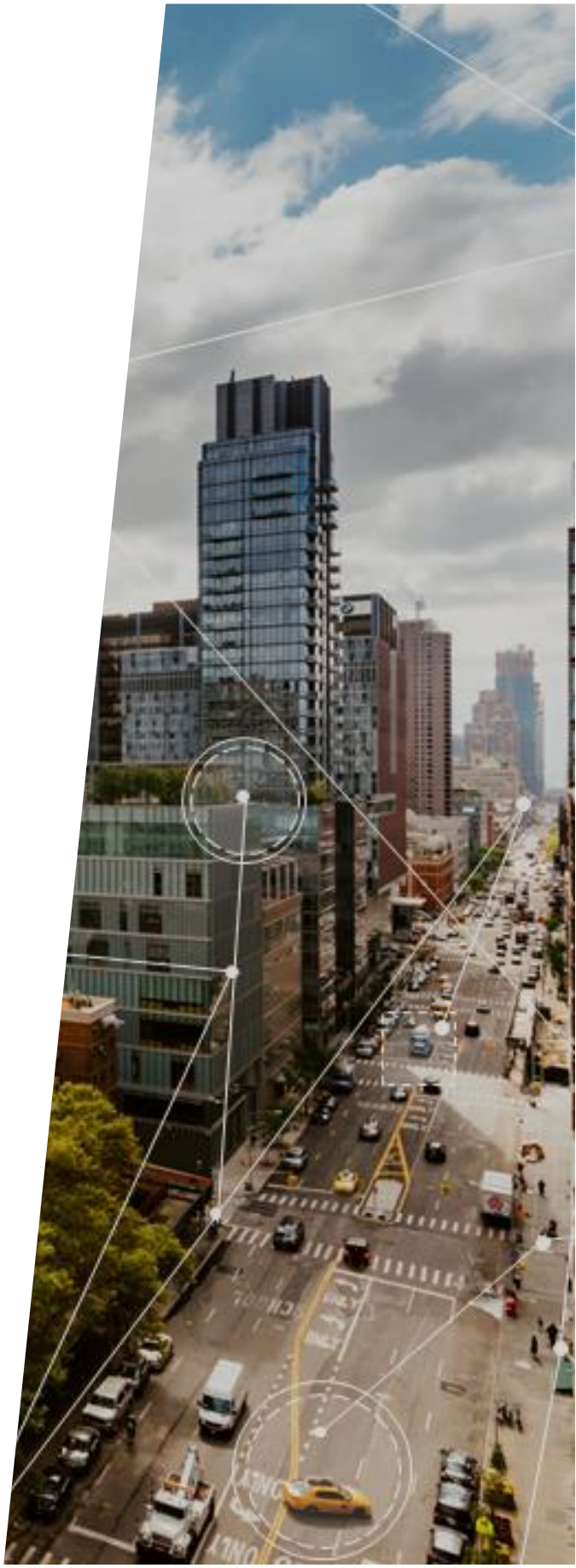


Microsoft

Contents

Introduction

In this paper, we will explore a technology called, “Preemption-Threshold Scheduling,” and how it can be used to reduce preemption overhead, while still enabling applications to meet real-time deadlines. Real- time embedded systems typically employ a collection of application tasks or threads that must complete their work before a certain deadline. Hard real-time systems demand that all deadlines be met with absolute certainty, even under worst case conditions. Real-time systems generally employ preemptive scheduling to guarantee that the most critical threads get immediate attention, enabling them to meet their deadlines. However, preemptive scheduling can result in significant context switch overhead under certain conditions. In this paper, we discuss a scheduling technology that reduces overhead while maintaining the ability to meet all deadlines, even under worst case conditions



Real-Time Operating System (RTOS) Concepts

Before addressing Preemption-Threshold Scheduling, it is important to understand some basic and advanced RTOS technology concepts, ranging from threads, priorities, multithreading, and preemption to Preemption Threshold Scheduling, an interesting technology that is the focus of this paper.

In this paper, we use the term "thread," but some RTOSes might use the term "task." In most cases, these are synonymous. A thread is a function or sort of a main program, that executes until it ends, or until it is interrupted by something else. Sometimes, threads can be blocked or stalled while waiting for an event of some sort. Threads share an address space, and run pseudo-concurrently, sharing the CPU. In multicore systems, threads might run truly concurrently, one on each core, and then share that core with other threads. Threads are used to structure a program in modular pieces, making the application easier to manage, and able to be developed by multiple team members.

Threads can have a number of states:

1. **READY** – means the thread is ready to run, not waiting for, nor needing anything else, but it is not currently executing instructions;
2. **RUNNING** – means the thread is executing instructions
3. **SUSPENDED** – means the thread is not ready to run, because it is waiting for something – perhaps a message in a queue, a semaphore, a timer to expire, etc.
4. **TERMINATED** – means the thread has completed its processing and is not eligible to run.

Thread Priorities

Threads are assigned priorities, which indicate their relative importance and the order in which they will get access to the CPU if more than one is READY to run. Generally, priorities are integer values, 0-to-N, either 0-high or 0-low. In ThreadX, and in this paper, 0 is the highest priority, while higher numbers indicate lower priorities. Each thread is assigned a priority when it is created, but the priority can be changed dynamically. Multiple threads can be assigned the same priority, or they each can be assigned a unique priority.

Multithreading is a term that indicates that the CPU is being shared by more than one thread. In such a system, when a running thread reaches a roadblock, rather than continuing to check for the ability to continue (polling), it can yield the CPU to another thread that is READY (not waiting for anything), thereby making more efficient use of otherwise wasted CPU cycles. For example, in a simple system with 2 threads, thread_a and thread_b, suppose thread_a is running and initiates an I/O operation that may take many hundreds of cycles to complete. Rather than just wait – in a polling loop – thread_a can be suspended until the I/O is complete, while thread_b can be allowed to use the CPU until that time. Once the I/O is complete, thread a is resumed. Multithreading enables more efficient use of CPU resources.

Preemption and Context Switching

In a fully preemptive system, when one thread is executing and a different thread, of higher priority, becomes READY to run (due to the occurrence of some asynchronous event or an explicit action of the running thread), the RTOS

STEP	OPERATION	CYCLES
1	Save the current thread's context (ie: GP and FP register values and PC) on the stack.	20 - 100
2	Save the current stack pointer in the thread's control block.	2 - 20
3	Switch to the system stack pointer.	2 - 20
4	Return to the scheduler.	2 - 20
5	Find the highest priority thread that is ready to run.	2 - 50
6	Switch to the new thread's stack.	2 - 50
7	Recover the new thread's context.	20 - 100
8	Return to the new thread at its previous PC.	2 - 40
9	Other processing	0 - 100
	TOTAL	50 - 500

interrupts, or "preempts" the running thread and replaces it with the higher-priority thread. The process of doing this is termed a "context switch." In a context switch, the RTOS saves the context of the executing thread on that thread's stack, and retrieves the context of the new thread, from the new thread's stack, and loads it into the CPU registers and program counter.

Thus, the executing thread context is switched from one to the other.

The context switch operation is reasonably complex, and can take from 50 to 500 cycles, depending on the RTOS and the processor. It is for this reason that care must be taken to optimize context switching operations in an RTOS, and to minimize the need for such operations in the application. This is one of the goals of Preemption-Threshold Scheduling.

Schedulers

Applications that do not use an RTOS but which are comprised of more than one operation or function – essentially a single task or thread – must provide a mechanism for running whichever function needs to run at any point in time. A simple sequential loop ("Big Loop") might be used, or a more sophisticated loop that checks status to determine whether a particular function has work to do, skipping those that don't and running those that do. Such loops are forms of schedulers, but they tend to be inefficient and unresponsive, especially as the number of threads or functions grows larger. In contrast, an RTOS scheduler keeps track of, or quickly determines, which activity to run at any point in time.

RTOS Schedulers

Generally, RTOS schedulers are preemptive – that means they make sure that the highest-priority thread that is ready to run is the one they let run, and the others wait. RTOS schedulers also can perform round-robin scheduling, which is similar to the Big Loop, or a more sophisticated form of round-robin whereby individual threads are granted a certain percentage of CPU time rather than allowed to run to completion or voluntary suspension. The RTOS scheduler performs context switches when required, and enables threads to sleep, to relinquish their CPU use, or to terminate and leave the pool of threads awaiting the CPU.

[sidebar] **Types of RTOS Schedulers**

1. Preemptive
2. Round-Robin
3. Round-Robin With Time Slicing
4. Preemption-Threshold [end sidebar]

The Perils of Preemption

Preemption generally delivers the fastest response to system events, and is the preferred scheduling method for real-time systems. However, preemption carries with it some potential problems, which the developer must avoid or properly handle.

First is thread **starvation**, where a thread never gets to execute because a higher priority thread never finishes. In a fully preemptive system, there is no way to run any thread if a higher priority thread is READY. Developers should avoid any condition whereby a high-priority thread might end up in an endless loop, or a situation where the thread consumes an undesirable amount of CPU time, preventing other threads from getting access to the processor.

Second, in situations with lots of context switching, **overhead** can add up. In an example to follow, we will examine just such a system with some tools that will enable us to see and measure the overhead.

Third, priority inversion can occur. Priority inversion is the situation whereby a high-priority thread is waiting for a shared resource, but the resource is held by a low priority thread which cannot get time to finish its use of the resource due to preemption by an intermediate priority thread.

A third problem is the preemption of a thread that is executing within a **critical section** of code. That's a section of code that must be completed once entered, so that no intervening activity can interfere with it without the running thread's awareness. If a thread is preempted while it is running in a critical section, the preempting thread might modify related data, and the first thread, when resumed, might see inconsistent status.

Preemption-Threshold Scheduling (PTS)

To help minimize, or avoid such problems, a modified preemptive scheduling algorithm has been developed and incorporated into ThreadX. This scheduling algorithm is called Preemption-Threshold Scheduling. In Preemption-Threshold Scheduling, we assign one or more threads a second priority, called its "Preemption-Threshold," that must be exceeded by a preempting thread. The thread's priority still is used to determine the scheduling of that thread among other threads that are READY, or to preempt another thread. But by defining a Preemption-Threshold that is a higher priority, preemption by some higher-priority threads can be inhibited.

Benefits

Preemption-Threshold Scheduling offers several benefits. It is a tool that may be used by system designers in many ways, even beyond those mentioned here, to suit various system use cases. Here are a few examples of the benefits of Preemption-Threshold Scheduling:

1. **Reduced Overhead**. Preemption-Threshold Scheduling can prevent certain threads from preempting a thread, when those preemptions might be excessive and detrimental to system performance. In many cases, a group of threads at closely ordered priorities work together better if they are not permitted to preempt each other, or if one of them should not be preempted by the others. Our example below illustrates this case.

2. **Critical Section Protection**. A critical section is a part of a multithreaded program that may not be concurrently executed by more than one of the threads. Typically, the critical section accesses a shared resource, such as a data structure, a peripheral device, or a network connection, that does not allow multiple concurrent accesses. Typically, preemption is disabled during critical section access, protecting the section but delaying response to unrelated events. Preemption-Threshold Scheduling can be used to prevent preemption among a set of threads, all of which interact with the same critical section. Threads dealing with the critical section cannot "step on" each other, but they still can be preempted by higher-priority threads that do not use the critical section, thus achieving better system responsiveness.

3. **Meet Schedulability Requirements**. Real-time systems have timing requirements that must be guaranteed. Scheduling and schedulability analysis enables these guarantees to be provided. Preemption-Threshold Scheduling has been studied by multiple academic researchers, and a number of papers have been published documenting its ability to meet schedulability requirements in real-time systems. For further information, please see the lists at the conclusion of this paper.

How it Works

Here is how Preemption-Threshold Scheduling works. Normally, any thread with a priority higher than the running thread can preempt it. But with Preemption-Threshold Scheduling, a running thread can only be preempted if the preempting thread has a priority higher than the running thread's Preemption- Threshold. In a fully preemptive system, the Preemption-Threshold would be equal to the thread's priority. By introducing a Preemption-

Threshold for a thread, and setting the Preemption-Threshold higher than the thread's priority, preemptions by threads with priorities in between those two values will not be permitted.

Priority	Comment
0	Preemption allowed for threads with priorities from 0 to 14 (inclusive)
:	
14	
15	Thread is assigned **Preemption-Threshold = 15** [this has the effect of disabling preemption for threads with priority values from 15 to 19 (inclusive)]
:	
19	
20	Thread is assigned **Priority = 20**
:	
31	

In this example, suppose we have a thread of priority 20, which would normally be preemptable by a

thread of priority 19, 18, 17, 16, and so on. But, if its Preemption-Threshold were set to 15, for example, only threads higher in priority than 15 (lower in number; ie 14, 13, 12, …), could preempt the thread. So,

threads in between - at priority 19, 18, 17, 16 and 15 - cannot preempt it, but threads at priority 14 and higher (lower numbers) can. Preemption- Threshold is optional, and can be specified for any thread, all threads, or no threads. If not specified for a thread, the thread can be preempted by any thread with a higher priority. But with Preemption-Threshold Scheduling, preemption of a thread can be prevented, up to some limit, above which preemption will be permitted.

Performance Benefits

To illustrate the performance benefits that can be achieved using Preemption-Threshold Scheduling, we will compare a fully-preemptive scheduling approach to one that uses Preemption-Threshold Scheduling, and we'll measure the consequences of each with respect to context switching and throughput. For this

investigation, we'll use a simple producer-consumer application, with one thread (Thread_D) sending 3 messages to each of 3 message queues, and 3 threads (Thread A, Thread B, and Thread C) each retrieving them from one of the queues. We'll log all events so we can see what is going on. Then, we'll view the logged events, count the context switches, measure the performance, and draw our conclusions.

To compare approaches, we'll consider 2 cases:

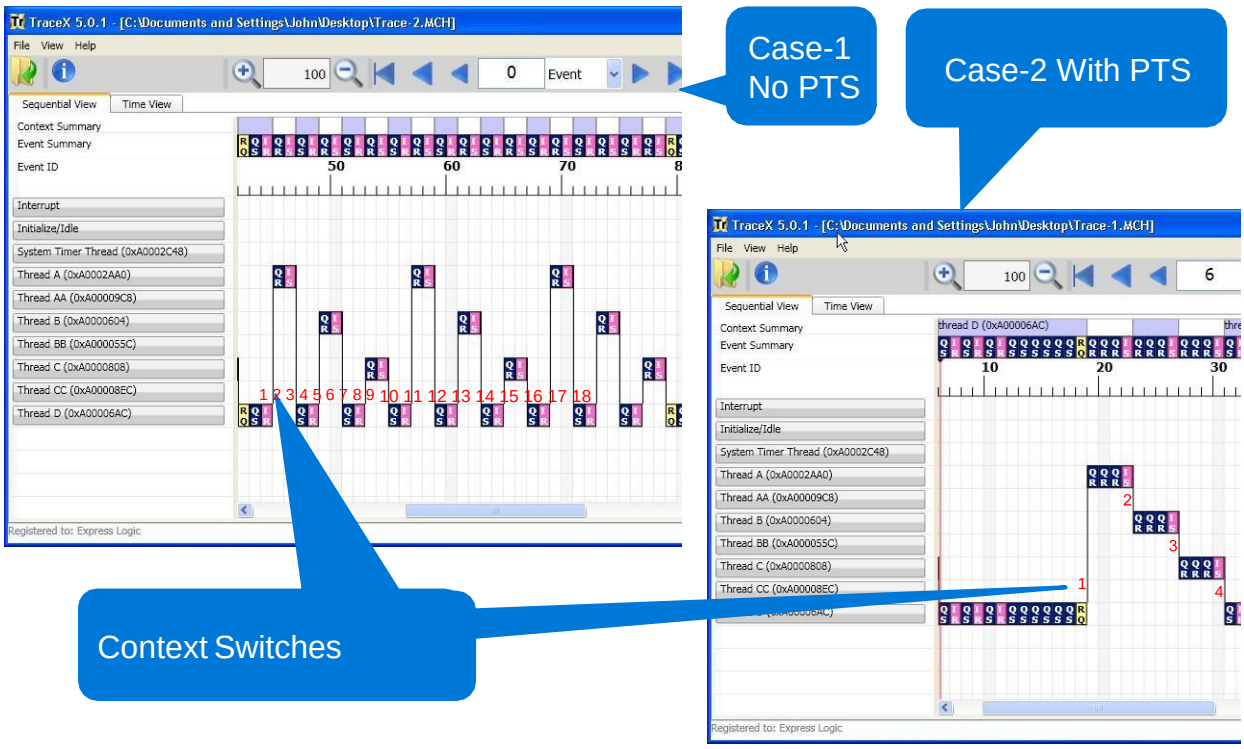
1. Case-1 uses fully-preemptive scheduling, with Threads A, B, C, and D assigned priorities 1, 2, 3, and 4, respectively.
2. In our second case, Case-2, we will use Preemption-Threshold Scheduling, to see how that can be used to reduce context switches. To do so, we assign Thread_D (our "Producer" thread) a preemption-threshold of 1, meaning that it can only be preempted by a thread with priority higher than 1. In this system, no thread has priority higher than 1 (=0), hence, Thread_D will not be preempted by thread A, B, or C

In Case-1, Threads A, B, and C each attempt to read a message from a queue but all are blocked since no messages have yet been sent by Thread D. Accordingly, Threads A, B, and C are SUSPENDED, awaiting a message to appear in the queue from which they are attempting to read. This enables Thread_D to run. Thread_D begins to send its messages to each queue, but as soon as it sends the first message, Thread_A jumps in to retrieve it. Why? Because Thread_A is higher in priority than Thread D, and Thread_A now is READY to run, since the queue from which it was attempting to read now is non-empty. Once Thread_A reads its message, it again is SUSPENED because the queue is again empty. Thread_D is resumed, and Thread_D now sends a message to the queue being read by Thread_B, which makes Thread B READY. Thread B preempts Thread_D, and reads its message, and then is SUSPENDED as the queue becomes empty. Similarly for Thread C, and so on through all 9 messages. This completes a cycle. In this cycle, 9 messages were sent, 9 retrieved, and 18 context switches were recorded.

In Case-2, the code is the same, but Thread_D will not be interrupted while it sends its messages, because Thread D has a Preemption-Threshold of 1, and none of the other threads have the required priority of 0 that would enable them to preempt Thread D. Thread_D will keep sending messages until it encounters a queue that is full, in which case it will be SUSPENDED until the queue becomes non-full. Note that once

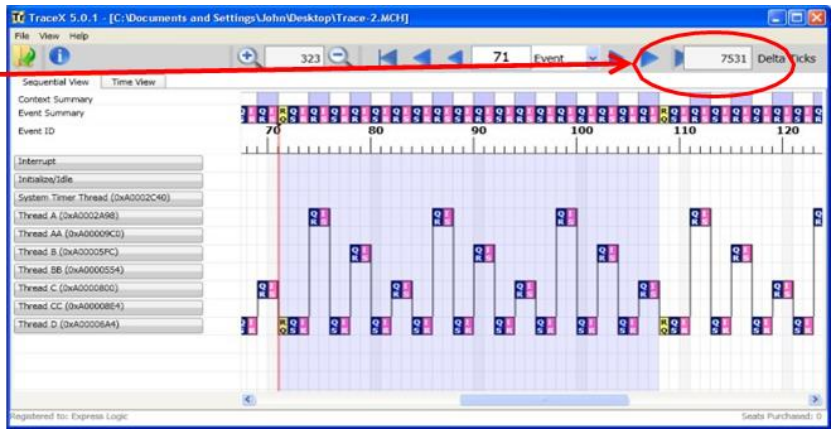
Thread D is SUSPENDED, it will not resume until threads A, B, and C are blocked, since Thread D has priority=4, and thus cannot preempt any other thread. The result is significantly different from Case-1.

While we still see 9 messages sent and 9 received, rather than seeing 18 context switches, we see only 4 context switches. Comparing context switches, we see Case-1 with 18 and Case-2 with 4. See the event trace graphic below:

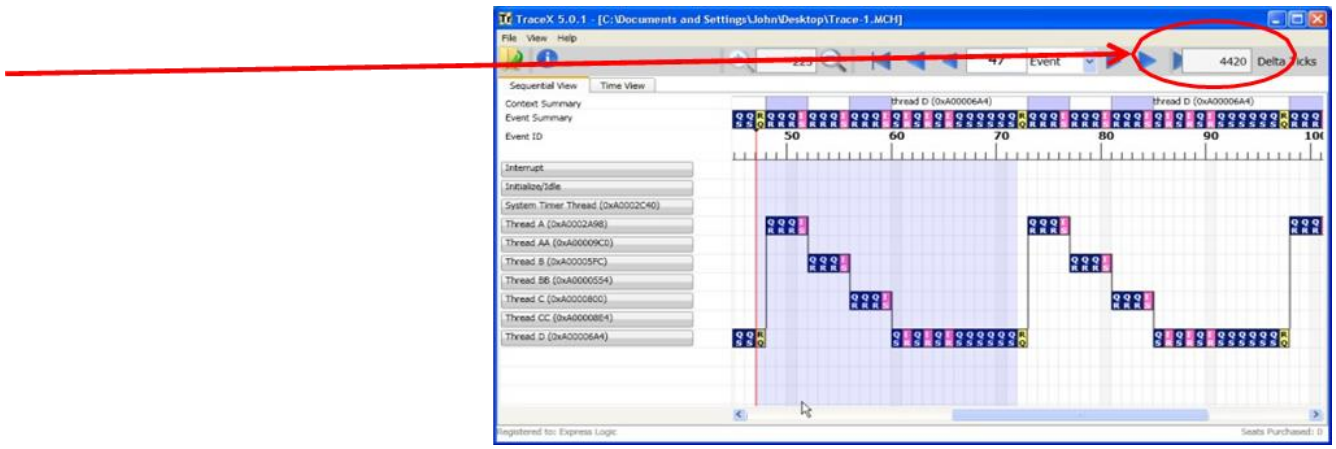


By selecting a complete cycle, we can see the number of timer ticks in that cycle.

Case-1 shows 7,531 ticks in a cycle



Case-2 shows 4,420 ticks in a cycle



Here is a summary of context switches and resulting throughput.

Case	Messages	Context Switches
Case-1: Fully Preemptive Scheduling	9	18
Case-2: Preemption-Threshold Scheduling	9	4

Measurement	Case-1 (Fully Preemptive)	Case-2 (Preemption-Threshold)	Ratio (Case 1 vs Case-2)
Context Switches	18	4	450%
Elapsed Time	7,531 ticks	4,420 ticks	170%
Messages Sent	9	9	No Change
Messages Received	9	9	No Change

If this application were a message-sending system, we'd see a significant improvement in performance and throughput using Preemption-Threshold Scheduling, as compared with the fully-preemptive Case-1.

Summary

We've seen how different types of RTOS schedulers work, and how a fully preemptive scheduler delivers maximum responsiveness. But a fully preemptive scheduler can introduce significant overhead that reduces system efficiency. In cases where system throughput is more critical than individual event responsiveness, Preemption-Threshold Scheduling can reduce context switches, and enable increased performance.

For further Reading (Academic Papers):

- Wang, Concordia University, and Saksena, University of Pittsburgh, on Scheduling Fixed-Priority Tasks with Preemption Threshold
- R. Ghattas and A. G. Dean. Preemption threshold scheduling: Stack optimality, enhancements
- and analysis. In RTAS '07: Proc. of the 13th IEEE Real Time and Embedded Technology and Applications Symposium, 2007.
- G. Yao and G. Buttazzo, Reducing Stack with Intra-Task Threshold Priorities in Real-Time Systems, Proc. of the 10th Int. Conf. on Embedded Software, 2010.

(Industry Publications):

- Embedded Systems Design Magazine, March, 2011, Feature Article: "Lower the Overhead in RTOS Scheduling," by Professor Alexander Dean, Ph.D.
- New Electronics article on Preemption-Threshold Scheduling, September 2013

Microsoft