Adaptive Partitioning User's Guide

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: Thursday, October 16, 2014

Table of Contents

About This Guide

The Adaptive Partitioning *User's Guide* will help you configure adaptive partitioning to divide system resources between competing processes.

The following table may help you find information quickly in this guide:

For information on:	Go to:
Adaptive partitioning in general	*What is Adaptive Partitioning?* (p. 9)
How the thread scheduler works	*Using the Thread Scheduler* (p. 15)
Setting up and using the thread scheduler	*Setting Up and Using the Adaptive Partitioning Thread Scheduler* (p. 31)
Using the adaptive partitioning architecture to solve different facets of the problem of controlling the consumption of resources in a system	*Controlling Resources Using the Thread Scheduler* (p. 37)
Knowing when and how to use the thread scheduler	*Considerations for the Thread Scheduler* (p. 39)
Security considerations when partitioning	*Security for Scheduler Partitions* (p. 49)
Checking for and fixing problems	*Testing and Debugging* (p. 53)
Setting up an OS image	*Sample Buildfile* (p. 59)
Frequently Asked Questions about the thread scheduler	*Frequently Asked Questions: Adaptive Partitioning Thread Scheduler* (p. 63)
Terminology used in this guide	*Glossary* (p. 93)

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL)`
Command options	`-lR`
Commands	`make`
Constants	`NULL`
Data types	`unsigned short`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective** **Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
What is Adaptive Partitioning?

As described in the Adaptive Partitioning chapter of the *System Architecture* guide, a *partition* is a virtual wall that separates competing processes or threads.

What are partitions and what is adaptive partitioning?

Partitions let the system designer allocate minimum amounts of system resources to each set of processes or threads. The primary resource considered is CPU time, but can also include any shared resource, such as memory and file space (disk or flash).

QNX Neutrino adaptive partitioning currently supports only the allocation of CPU time via the *thread scheduler* (p. 13).

Traditional partitions are *static* and work optimally when there's little or no dynamic deployment of software; in dynamic systems, static partitions can be inefficient.

Adaptive partitions are more flexible because:

- You can dynamically add and configure them.
- They behave as a global hard real time scheduler under normal load, but can continue to provide minimal interrupt latencies when the system is fully loaded.
- They distribute a partition's unused resources among partitions that require additional resources when the system isn't loaded.

You can introduce adaptive partitioning without changing — or even recompiling — your application code, although you do have to rebuild your system's OS image.

Are partitions box-like? No, they're better. Many competing resource partitioning systems take their model from CPU virtualization, where the objective is to try to divide a computer into a set of smaller computers that interact as little as possible (into a number of boxes). This approach isn't very flexible; every thread, process, and byte of memory is in exactly one box, and it can never move. Adaptive partitioning takes a much more flexible view.

To begin, QNX Neutrino partitions are adaptive because:

- You can change configurations at run time.
- They are typically fixed at one configuration time.
- The partition behavior auto-adapts to conditions at run time. For example:
  - Free time is redistributed to other scheduler partitions.
  - Filesystems can bill time to clients with a mechanism that temporarily moves threads between time partitions.
- Time partitions aren't required to have the same members.

As a result, adaptive partitions aren't boxes; they are much more powerful. In addition to being adaptive, time partitions allow you to easily model the fundamentally different behavior of CPU time when viewed as a resource.

Use a partition to encapsulate wild processes. You can create an application partition for one or more processes that potentially run on high priorities. Then, start the potentially problematic application(s) inside that partition so that when a situation like this arises, the rest of the system will continue to function.

Since adaptive partitions are not “boxes” what are they?

An adaptive partition is a named set of rules. The rules are selected to control the global resource behavior of the system. When a process or thread is associated with a particular partition (scheduler), then its actions are governed by the rules of that partition at that time.

For example, adaptive partitioning is similar to people who belong to clubs. Each person can join several different clubs. They can even move from one club to another club at times. However, while they are at a particular club, they agree to abide by the rules of that specific club.

System and user requirements

For adaptive partitioning to operate properly, your system should meet these requirements:

- On x86 systems, turn off any BIOS configuration that may cause the processor to enter System Management Mode (SMM). A typical example is USB legacy support. If the processor enters SMM, the adaptive partitioning thread scheduler continues to function correctly, but CPU percentages apportioned to partitions will be inaccurate.

  A typical reason for preventing SMM is that it introduces interrupt latencies of about 100 microseconds at unpredictable intervals.
- Adaptive partitioning isn't supported on 486 processors, because they don't have a timebase counter, which the adaptive partitioning thread scheduler needs in order to perform microbilling.

The thread scheduler

The *adaptive partitioning thread scheduler* is an optional thread scheduler that lets you guarantee minimum percentages of the CPU's throughput to groups of threads, processes, or applications. The percentage of the CPU time allotted to a partition is called a *budget*.

The thread scheduler was designed on top of the core QNX Neutrino architecture primarily to solve the following problems encountered in embedded systems design:

- to guarantee proper function when the system is fully loaded
- to prevent unimportant or untrusted applications from monopolizing the system

Chapter 2
Using the Thread Scheduler

The adaptive partitioning thread scheduler is an optional thread scheduler that lets you guarantee minimum percentages of the CPU's throughput to groups of threads, processes, or applications. The percentage of the CPU allotted to a partition is called a *budget*.

The thread scheduler was designed on top of the core QNX Neutrino RTOS architecture to primarily solve two problems in embedded systems design:

- to function properly under fully loaded conditions
- to prevent unimportant or untrusted applications from monopolizing the system

We call our partitions *adaptive* because their contents are dynamic:

- You can dynamically launch an application into a partition.
- Child threads and child processes automatically run in the same partition as their parent.
- By default, when you use the standard QNX Neutrino send-receive-reply messaging, message receivers automatically run in the partition of the message sender while they're processing that message. This means that all resource managers, such as drivers and filesystems, automatically bill CPU time (except overhead) to the budget of their clients.

Keeping track of CPU time

The adaptive partitioning thread scheduler throttles CPU usage by measuring the average CPU usage of each partition. The average is computed over an averaging window (typically 100 milliseconds), a value that is configurable.

However, the thread scheduler doesn't wait 100 milliseconds to compute the average. In fact, it calculates it very often. As soon as 1 millisecond passes, the usage for this 1 millisecond is added to the usage of the previous 99 milliseconds to compute the total CPU usage over the averaging window (i.e. 100 milliseconds).

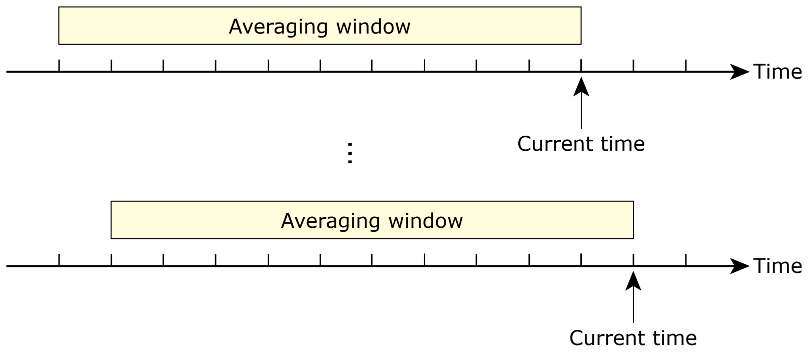

Figure 1: The averaging window moves forward as time advances.

The window size defines the averaging time over which the thread scheduler attempts to balance the partitions to their guaranteed CPU limits. You can set the averaging window size to any value from 8 milliseconds to 400 milliseconds.

Different choices of the window size affect both the accuracy of load balancing and, in extreme cases, the maximum delays seen by ready-to-run threads. For more information, see the *Considerations for the Thread Scheduler* (p. 39) chapter.

Because the averaging window slides, it can be difficult for you to keep statistics over a longer period, so the scheduler keeps track of two other windows:

- Window 2 — typically 10 times the window size.
- Window 3 — typically 100 times the window size.

To view the statistics for these additional windows, use the `show -v` or `show -vv` option with the `aps` command.

The thread scheduler accounts for time spent to the actual fraction of clock ticks used, and accounts for the time spent in interrupt threads and in the kernel on behalf of user threads. We refer to this as *microbilling*.

Microbilling may be approximated on ARM targets if the board can't provide a micro clock.

How is CPU time divided between partitions?

The thread scheduler is a fair-share scheduler. This means that it guarantees that partitions receive a defined minimum amount of CPU time (their *budget*) whenever they demand it. The thread scheduler is also a real time scheduler. That means it's a preemptive, priority-based scheduler. These two requirements appear to conflict, but the thread scheduler satisfies both of these requirements by scheduling through priority at all times so that it doesn't need to limit a partition in order to guarantee some other partition its budget.

Underload

Underload occurs when partitions demand less CPU time than their defined budgets, over the averaging window. For example, if we have three partitions:

- System partition, with a 70% budget
- partition `Pa`, with a 20% budget
- partition `Pb`, with 10% budget

with light demand in all three partitions, the output of the `aps show` command might be something like this:

```
                    +---- CPU Time ----+-- Critical Time --
Partition name   id | Budget |    Used | Budget |      Used
--------------------+------------------+-------------------
System            0 |    70% |   6.23% |  200ms |   0.000ms
Pa                1 |    20% |  15.56% |    0ms |   0.000ms
Pb                2 |    10% |   5.23% |    0ms |   0.000ms
--------------------+------------------+-------------------
Total               |   100% |  27.02% |
```

In this case, all three partitions demand less than their budgets.

Whenever partitions demand less than their budgets, the thread scheduler chooses between them by picking the highest-priority running thread. In other words, when underloaded, the thread scheduler is a strict real time scheduler. This is simply typical of QNX Neutrino scheduling.

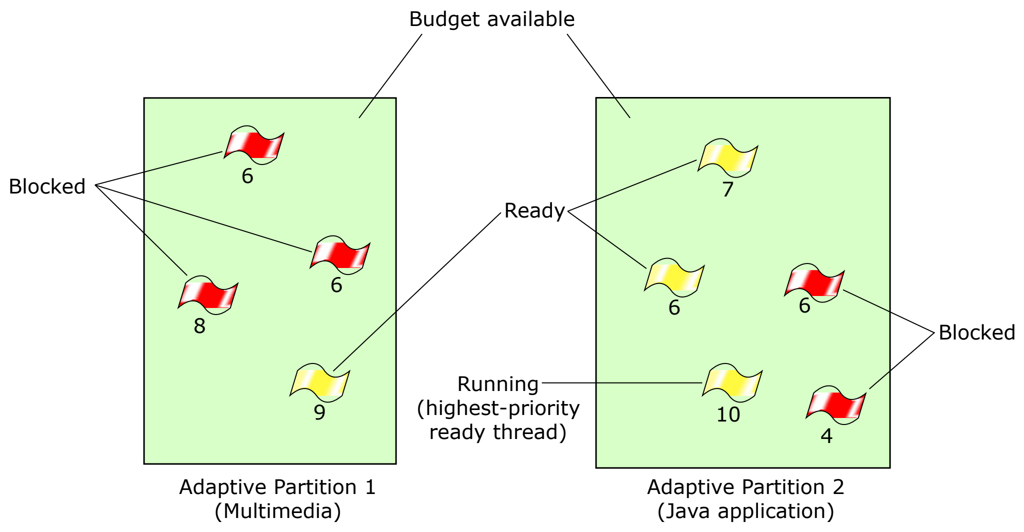

Figure 2: The thread scheduler's behavior when underloaded.

Free time

Free time occurs when one partition isn't running. The thread scheduler then gives that partition's time to other running partitions. If the other running partitions demand enough time, they're allowed to run over budget.

If a partition opportunistically goes over budget, it must pay back the borrowed time, but only as much as the scheduler remembers (i.e. only the borrowing that occurred in the last window).

For example, suppose we have these partitions:

- System partition, with a 70% budget, but running no threads
- partition Pa, with a 20% budget, running an infinite loop at priority 9
- partition Pb, with a 10% budget, running an infinite loop at priority 10

Because the System partition demands no time, the thread scheduler distributes the remaining time to the highest-priority thread in the system. In this case, that's the infinite-loop thread in partition Pb. So the output of the `aps show` command may be:

```
                    +---- CPU Time ----+-- Critical Time --
Partition name   id | Budget |    Used | Budget |      Used
--------------------+------------------+-------------------
System            0 |    70% |   0.11% |  200ms |   0.000ms
Pa                1 |    20% |  20.02% |    0ms |   0.000ms
Pb                2 |    10% |  79.83% |    0ms |   0.000ms
--------------------+------------------+-------------------
Total               |   100% |  99.95% |
```

In this example, partition Pa receives its guaranteed minimum of 20%, but all of the free time is given to partition Pb.

This is a consequence of the principle that the thread scheduler chooses between partitions strictly by priority, as long as no partition is being limited to its budget. This strategy ensures the most real time behavior.

But, there may be circumstances when you don't want partition `Pb` to receive all of the free time just because it has the highest-priority thread. That may occur when, say, when you choose to use `Pb` to encapsulate an untrusted or third-party application, particularly when you are unable control its code.

In that case, you may want to configure the thread scheduler to run a more restrictive algorithm that divides free time by the budgets of the busy partitions (rather than assigning all of it to the highest-priority thread). To do so, set the `SCHED_APS_FREETIME_BY_RATIO` flag (see “Scheduling policies” in the entry for *SchedCtl()* in the QNX Neutrino *C Library Reference*), or use the `aps modify -S freetime_by_ratio` command (see the *Utilities Reference*).

In our example, freetime-by-ratio mode might cause the `aps show` command to display:

```
                    +---- CPU Time ----+-- Critical Time --
Partition name   id | Budget |    Used | Budget |      Used
--------------------+------------------+-------------------
System            0 |    70% |   0.04% |  200ms |   0.000ms
Pa                1 |    20% |  65.96% |    0ms |   0.000ms
Pb                2 |    10% |  33.96% |    0ms |   0.000ms
--------------------+------------------+-------------------
Total               |   100% |  99.96% |
```

which indicates that in freetime-by-ratio mode, the thread scheduler divides free time between partitions `Pa` and `Pb` in roughly a 2:1 ratio, which is the ratio of their budgets.

Full Load

Full load occurs when all partitions demand their full budget. A simple way to demonstrate this is to run `while(1)` loops in all of the sample partitions. In this case, the `aps show` command might display:

```
                    +---- CPU Time ----+-- Critical Time --
Partition name   id | Budget |    Used | Budget |      Used
--------------------+------------------+-------------------
System            0 |    70% |  69.80% |  200ms |   0.000ms
Pa                1 |    20% |  19.99% |    0ms |   0.000ms
Pb                2 |    10% |   9.81% |    0ms |   0.000ms
--------------------+------------------+-------------------
Total               |   100% |  99.61% |
```

In this example, the requirement to meet the partitions' guaranteed budgets takes precedence over priority.

In general, when partitions are at or over their budget, the thread scheduler divides time between them by the ratios of their budgets, and balances usage to a few percentage points of the partitions' budgets. (For more information on budget accuracy, see “ *Choosing the window size* (p. 43) ” in the Considerations for Scheduling chapter of this guide.)

Even at full load, the thread scheduler can provide real time latencies to an engineerable set of critical threads (see *Critical threads* (p. 23) later in this chapter). However, in that case, the scheduling of critical threads takes precedence over meeting budgets.

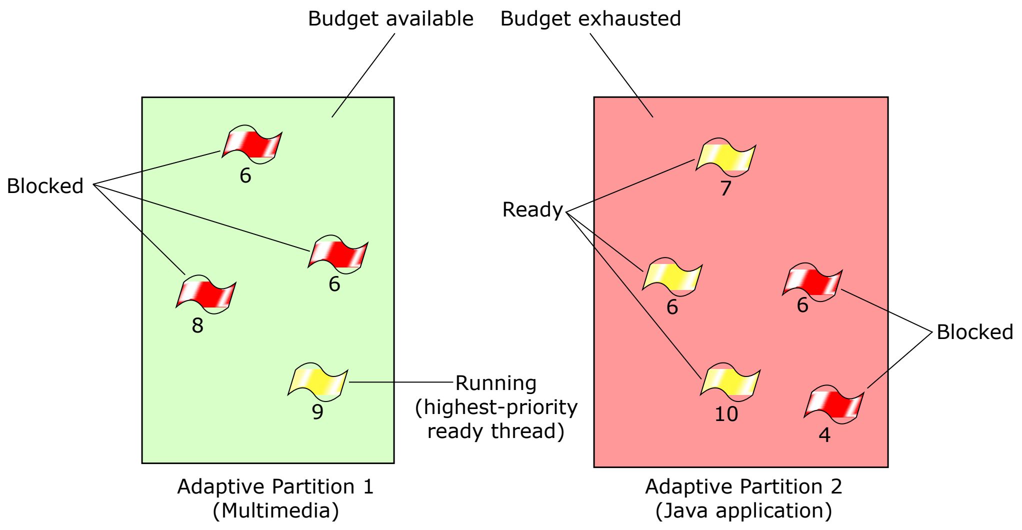
Budget available
Budget exhausted
Blocked
6
6
8
9
Running (highest-priority ready thread)
Ready
7
6
6
10
4
Blocked
Adaptive Partition 1 (Multimedia)
Adaptive Partition 2 (Java application)

Figure 3: The thread scheduler's behavior under a full load.

Summary of scheduling behavior

The following table summarizes how the thread scheduler divides time in normal and freetime-by-ratio mode:

Partition state	Normal	Freetime-by-ratio
Usage < budget	By priority	By priority
Usage > budget and there's free time	By priority	By ratio of budgets
Full load	By ratio of budgets	By ratio of budgets
Partitions running a critical thread at any load	By priority	By priority

The scheduler's overhead doesn't increase with the number of threads; but, it may increase with the number of partitions, so you should use as few partitions as possible.

Partition inheritance

Whenever a server thread in the standard QNX Neutrino send-receive-reply messaging scheme receives a message from a client, QNX Neutrino considers the server thread to be working on behalf of the client. So QNX Neutrino runs the server thread at the priority of the client. In other words, threads receiving messages inherit the priority of their sender.

With the thread scheduler, this concept is extended; we run server threads in the partition of their client thread while the server is working on behalf of that client. So the receiver's time is billed to the sender's scheduler partition.

What about any threads or processes that the server creates? Which partition do they run in?

- New threads — If you receive a message from another partition, and you create a new thread in response, the child thread runs in the sender's partition until the child thread becomes receive-blocked. At that point, the child thread's partition is reset to be its creator's partition.
- New processes — If you receive a message from another partition, and create a process in response, the process is created in the sender's partition. Any threads that the child process creates also run in the sender's partition.

If you don't want the server or any threads or processes it creates to run in the client's partition, set the `_NTO_CHF_FIXED_PRIORITY` flag when the server creates its channel. For more information, see *ChannelCreate* in the QNX Neutrino *C Library Reference*.

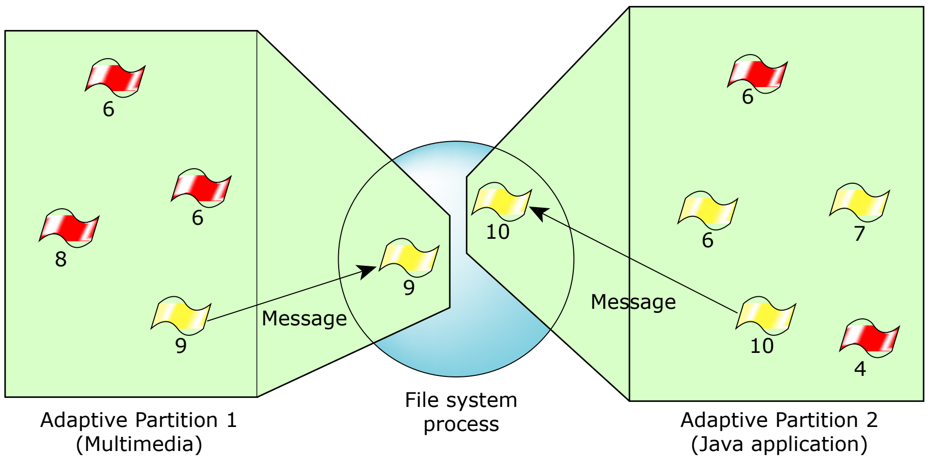
6
6
8
9
Message
9
10
Message
6
6
7
10
4
Adaptive Partition 1 (Multimedia)
File system process
Adaptive Partition 2 (Java application)

Figure 4: Server threads temporarily join the partition of the threads they work for.

Send-receive-reply message-passing is the only form of thread communication that automatically makes the server inherit the client's partition.

Pulses don't inherit the sender's partition. Instead, their handlers run in the process's pulse-processing partition, which by default is the partition that the process was initially created in. You can change the pulse-processing partition with the `SCHED_APS_JOIN_PARTITION` command to *SchedCtl()*, specifying the process ID, along with a thread ID of -1.

Critical threads

A *critical thread* is one that's allowed to run even if its partition is over budget (provided that partition has a critical time budget). By default, QNX Neutrino automatically identifies all threads that are initiated by an I/O interrupt as critical. However, you can use *SchedCtl()* to mark selected threads as critical.

The ability to mark any thread as critical may require security control to prevent its abuse as a DOS attack. For information about security, see *Managing security for the thread scheduler* (p. 50) in the Security for Scheduler Partitions chapter of this guide.

Critical threads always see realtime latencies, even when the system is fully loaded, or any time other threads in the same partition are being limited to meet budgets. The basic idea is that a critical thread is allowed to violate the budget rules of its partition and run immediately, thereby obtaining the realtime response it requires. For this to work properly, there must not be many critical threads in the system.

You can use a `sigevent` to make a thread run as critical:

1. Define and initialize the `sigevent` as normal. For example:

   ```
   struct sigevent my_event;
   SIGEV_PULSE_INIT (&my_event, coid, 10,
                     MY_SIGNAL_CODE, 6969);
   ```

2. Set the flag that will mark the thread that receives your event as a critical event before you send the event:

   ```
   SIGEV_MAKE_CRITICAL(&my_event);
   ```

   This has an effect only if the thread receiving your event runs in a partition with a critical-time budget.

3. Process the `sigevent` as normal in the thread that receives it. This thread doesn't have to do anything to make itself a critical thread; the kernel does that automatically.

To make a thread noncritical, you can use the `SIGEV_CLEAR_CRITICAL` macro when you set up a `sigevent`.

The `SIGEV_CLEAR_CRITICAL` and `SIGEV_MAKE_CRITICAL` macros set a hidden bit in the *sigev_notify* field. If you test the value of the *sigev_notify* field of your `sigevent` after creating it, and if you've ever used the *SIGEV_MAKE_CRITICAL()* macro, then use code like this:

```
switch (SIGEV_GET_TYPE(&my_event) ) {
```

instead of this:

```
switch (my_event.sigev_notify) {
```

A thread that receives a message from a critical thread automatically becomes critical as well.

You may mark selected scheduler partitions as critical and assign each partition a critical time budget. Critical time is specifically intended to allow critical interrupt threads to run over budget.

The critical time budget is specified in milliseconds. It's the amount of time all critical threads may use during an averaging window. A critical thread will run even if its scheduler partition is out of budget, as long as its partition has critical budget remaining.

Critical time is billed against a partition while all these conditions are met:

- The running partition has a critical budget greater than zero.
- The top thread in the partition is marked as running critical, or has received the critical state from receiving a *SIG_INTR()*, a `sigevent` marked as critical, or has just received a message from a critical thread.
- The running partition must be out of percentage-CPU budget.
- There must be at least one other partition competing for CPU time.

Otherwise, the critical time isn't billed. The critical threads run whether or not the time is billed as critical. The only time critical threads won't run is when their partition has exhausted its critical budget (see “ *Bankruptcy* (p. 25) ”).

In order to be useful and effective, the number of critical threads in the system must be few, and it's also ideal to give them high and unique priorities. Consequently, if critical threads are the majority, the thread scheduler will rarely be able to guarantee all of the partitions their minimum CPU budgets. In other words, the system degrades to a priority-based thread scheduler when there are too many critical threads.

To gain benefit from being critical, a critical thread must be the highest priority thread in the system, and not share its priority with other threads. If a ready-to-run critical thread is behind other noncritical threads (either because others have a higher priority, or are at the same priority and were made ready before your critical thread), then the critical thread may stall if the partition is out of budget.

Although your thread is critical, it must wait for a higher priority, and earlier threads sharing its partition to run first. However, if those other threads are noncritical, and if the partition is out of budget, your critical thread won't run until the averaging window rotates so that the partition once again has a budget.

A critical thread remains critical until it becomes receive-blocked. A critical thread that's being billed for critical time won't be round-robin-timesliced (even if its scheduling policy is round robin).

QNX Neutrino marks all `sigevent` structures that are returned from a user's interrupt-event handler functions as critical. This makes all I/O handling threads automatically critical. This is done to minimize code changes that you would need to do for basic use of partition thread scheduling. If you don't want this behavior to occur, specify the `-c` option to `procnto` in your buildfile.

To make a partition's critical budget infinite, set it to the number of processors times the size of the averaging window. Do this with caution, as it can cause security problems; see “ *Managing security for the thread scheduler* (p. 50) ” in the Considerations for The Thread Scheduler chapter of this guide.

Bankruptcy

Bankruptcy occurs when the critical CPU time billed to a partition exceeds its critical budget.

The `System` partition's critical budget is infinite; this partition can never become bankrupt.

It's very important that you test your system under a full load to ensure that everything works correctly, in particular to ensure that you've chosen the correct critical budgets. One method to verify this is to start a `while(1)` thread in each partition to consume all available time.

Bankruptcy is always considered to be a design error on the part of the application, but the system's response is configurable. QNX Neutrino lets you set a recovery policy. The options are:

Default

Do the minimum. When a partition runs out of critical budget, isn't allowed to run again until it receives more budget, i.e. the sliding-averaging window recalculates that partition's average CPU consumption to be a bit less than its configured CPU budget. After bankruptcy, enough time must pass for the calculated average CPU time of the partition to fall to its configured budget. At the very least, this means that a number of milliseconds equal to the critical budget must pass before that partition is scheduled again.

Force a reboot

This is intended for your regression testing. It's a good way of making sure that code causing an unintended bankruptcy is never accidentally shipped to your customers. *We recommend that you turn off this option before you ship.*

Notify

The *SchedCtl()* function lets you attach a `sigevent` to each partition. The thread scheduler delivers that `sigevent` when the corresponding partition becomes bankrupt. To prevent a possible flood of `sigevents`, the thread scheduler will deliver at most one `sigevent` per registration. If you want another notification, use *SchedCtl()* again and reattach the event to obtain the next notification. As a result, QNX Neutrino arranges the rate of delivery of bankruptcy notification to never exceed the application's ability to receive them.

Cancel

The cancel option causes the bankrupt partition's critical-time budget to be set to zero. That prevents it from running as critical until you restore its critical-time budget, either through the `SCHED_APS_MODIFY_PARTITION` command to the *SchedCtl()* function, or through the `-B` option to the `aps modify` command.

You can set the bankruptcy policy with the `aps` utility (see the *Utilities Reference*) or the SCHED_APS_SET_PARMS command to *SchedCtl()* (see the QNX Neutrino *C Library Reference*).

Whenever a critical or normal budget is changed for any reason, there is an effect on bankruptcy notification: it delays bankruptcy handing by two windows to prevent a false bankruptcy detection if a partition's budget suddenly changes, for example, from 90% to 1%.

Canceling the budget on bankruptcy changes the partition's critical budget, causing further bankruptcy detections to be suppressed for two window sizes.

In order to cause the entire system to stabilize after such an event, the thread scheduler gives *all* partitions a two-window grace period against declaring bankruptcy when one partition has its budget canceled.

Adaptive partitioning thread scheduler and other thread schedulers

Priorities and thread scheduler policies are relative to one adaptive partition only; priority order is respected within a partition, but not between partitions if the thread scheduler needs to balance budgets. You can use the thread scheduler with the existing FIFO, round robin, and sporadic scheduling policies. The scheduler, however, may stop a thread from running before the end of its timeslice (round robin case), or before the thread has run to completion (FIFO case).

This occurs when the thread's partition runs out of budget and some other partition has budget, i.e. the thread scheduler doesn't wait for the end of a thread's timeslice to determine whether to run a thread from a different partition. The scheduler takes that decision every clock tick (where a tick is 1 millisecond on most machines). There are 4 clock ticks per timeslice.

Round robin threads being billed for critical time aren't timesliced with threads of equal priority.

A caveat about FIFO scheduling

Be careful not to misuse the FIFO scheduling policy. There's a technique for obtaining mutual exclusion between a set of threads reading and writing shared data without using a mutex: you can make all threads vying for the same shared data run at the same priority.

Since only one thread can run at a time (at least, on a uniprocessor system), and with FIFO scheduling, one thread never interrupts another, each has a monopoly on the shared data while it runs. This is bad because any accidental change to the scheduler policy or priority will likely cause one thread to interrupt the other in the middle of its critical section. So it may lead to a code breakdown. If you accidentally put the threads using this technique into different partitions (or let them receive messages from different partitions), their critical sections will be broken.

If your application's threads use their priorities to control the order in which they run, you should always place the threads in the same partition, and you shouldn't send messages to them from other partitions.

Pairs of threads written to depend on executing in a particular order based on their priorities should always be placed in the same partition, and you shouldn't send them messages from other partitions.

In order to solve this problem, you must use mutexes, barriers, or pulses to control thread order. This will also prevent problems if you run your application on a multicore system. As a workaround, you can specify the `_NTO_CHF_FIXED_PRIORITY` flag to *ChannelCreate()*, which prevents the receiving thread from running in the sending thread's partition.

In general, for mutual exclusion, you should ensure that your applications don't depend on FIFO scheduling, or on the length of the timeslice.

Using the thread scheduler and multicore together

On a multicore system, you can use scheduler partitions and symmetric multiprocessing (SMP) to reap the rewards of both. For more information, see the Multicore Processing *User's Guide*.

Note the following facts:

- On an SMP machine, the thread scheduler considers the time to be 100%, not (say) 400% for a four-processor machine
- The thread scheduler first attempts to keep every processor busy; only then does it apply budgets. For example, when you have a four-processor machine, and if partitions are divided into 70%, 10%, 10%, and 10%, if there's only one thread running in each partition, the thread scheduler runs all four threads all the time. The thread scheduler and the `aps` command report the partition's consumed time as 25%, 25%, 25%, and 25%.

It may seem unlikely to have only one thread per partition, since most systems have many threads. However, there is a way this situation will occur on a multithreaded system.

The *runmask* controls which CPUs a thread is allowed to run on. With careful (or foolish) use of the runmask, it's possible to arrange things so that there aren't enough threads that are permitted to run on a particular processor for the scheduler to meet its budgets.

If there are several threads that are ready to run, and they're permitted to run on each CPU, then the thread scheduler correctly guarantees each partition's minimum budget.

On a hyperthreaded machine, actual throughput of partitions may not match the percentage of CPU time usage reported by the thread scheduler. This discrepancy occurs because on a hyperthreaded machine, throughput isn't always proportional to time, regardless of what kind of scheduler is being used. This scenario is most likely to occur when a partition doesn't contain enough ready threads to occupy all of the pseudo-processors on a hyperthreaded machine.

Scheduler partitions and BMP

Certain combinations of runmasks and partition budgets can have surprising results.

For example, suppose we have a two-CPU SMP machine, with these partitions:

- `Pa`, with a budget of 50%
- `System`, with a budget of 50%

Now, suppose the system is idle. If you run a priority-10 thread that's locked to CPU 1 and is in an infinite loop in partition `Pa`, the thread scheduler interprets this to mean that you intend `Pa` to monopolize CPU 1. That's because CPU 1 can provide only 50% of the entire machine's processing time.

If you run another thread at priority 9, also locked to CPU 1, but in the System partition, the thread scheduler interprets that to mean you also want the System partition to monopolize CPU 1.

The thread scheduler has a dilemma: it can't satisfy the requirements of both partitions. What it actually does is allow partition `Pa` to monopolize CPU 1.

This is why: from an idle start, the thread scheduler observes that both partitions have available budget. When partitions have available budget, the thread scheduler schedules in realtime mode, which is strict priority scheduling. So partition `Pa` runs. However, because CPU 1 can never satisfy the budget of partition `Pa`; `Pa` never runs out of budget. Therefore, the thread scheduler remains in realtime mode and the lower-priority `System` partition never runs.

For this example, the `aps show` command might display:

```
                    +---- CPU Time ----+-- Critical Time --
Partition name   id | Budget |    Used | Budget |      Used
--------------------+------------------+-------------------
System            0 |    50% |   0.09% |  200ms |   0.000ms
Pa                1 |    50% |  49.93% |    0ms |   0.000ms
--------------------+------------------+-------------------
Total               |   100% |  50.02% |
```

The System partition receives no CPU time even though it contains a thread that is ready to run.

Similar situations can occur when there are several partitions, each having a budget less than 50%, but whose budgets sum to 50% or more.

Avoiding infinite loops is a good way to avoid these situations. However, if you're running third-party software, you may not have control over the code.

Chapter 3
Setting Up and Using the Adaptive Partitioning Thread Scheduler

To use the thread scheduler, you'll need to first build an image, and then create scheduler partitions to launch processes in partitions.

Building an image

To use the thread scheduler, you must add the `[module=aps]` attribute to the command that launches `procnto` in your OS image's buildfile. For example:

```
[module=aps] PATH=/proc/boot procnto -vv
```

Once you've added this line, use `mkifs` to rebuild your OS image, and then put the image in `/.boot`. For details, see *Building Embedded Systems*.

You don't need to recompile your applications in order to run them in partitions.

Creating scheduler partitions

On boot up, the system creates the initial partition, number 0, called `System`. The System partition initially has a budget of 100%. You can create partitions and set their budgets in your buildfile, with command-line utilities, or dynamically through the API defined in `sys/sched_aps.h`. When you create a partition, its budget is subtracted from its parent partition's budget.

To see which partitions you've created, use the `aps show` command. For more information about the `aps` utility, see `aps`.

Using a buildfile

To create a partition in your buildfile, add a line like this to the startup script:

```
sched_aps name budget
```

You can also use the `aps` in your startup script to set security options. For example, to create a partition called `Drivers` with a CPU budget of 20% and then use our recommended security option, add these lines to your buildfile's startup script:

```
sched_aps Drivers 20
aps modify -s recommended
```

Using the command line

To create a partition from the command line, use the `aps` utility's create option. For example:

```
aps create -b15 DebugReserve
```

creates a partition named `DebugReserve` with a budget of 15%.

When you create a partition, its budget is taken from its parent partition's budget. The parent partition is usually the system partition.

Using a program

To create a partition from a program, use the `SCHED_APS_CREATE_PARTITION` command to *SchedCtl()*. For example:

```
sched_aps_create_parms creation_data;

memset(&creation_data, 0, sizeof(creation_data));
creation_data.budget_percent = 15;
creation_data.critical_budget_ms = 0;
creation_data.name = "DebugReserve";

ret = SchedCtl( SCHED_APS_CREATE_PARTITION, &creation_data,
                sizeof(creation_data));
if (ret != EOK) {
   printf("Couldn't create partition \"%s\": %s (%d).\n",
            creation_data.name, strerror(errno), errno);
} else {
   printf ("The new partition's ID is %d.\n", creation_data.id);
}
```

Note that *SchedCtl()* puts the partition's ID in the `sched_aps_create_parms` structure.

Launching a process in a partition

You can use options in your buildfile to launch applications at boot time. In general, you need to launch only the command that starts a multiprocess application, since child processes of your initial command — including shells and commands run from those shells — run in the same partition.

You can also launch a process into a partition at the command line. The interface defined in `sys/sched_aps.h` lets you launch individual threads into a partition and move currently running threads into another partition.

Using a buildfile

To launch a command into a partition, use the `[sched_aps=`*`partition_name`*`]` attribute in your buildfile's startup script. For example:

```
[+session pri=35 sched_aps=DebugReserve] ksh &
```

launches a high-priority shell in the `DebugReserve` partition.

The statements you use to start a command in a partition may appear anywhere in the startup script after you've created the partition.

Using the command line

To launch a program in a partition from the command line, use the `-Xaps=`*partition_name* option of the `on` command. (The `X` refers to an external scheduler, the thread scheduler in this case.) For example:

```
on -Xaps=DebugReserve ksh
```

launches a shell into the `DebugReserve` partition.

Using a program

To launch a program into a partition from a program, start the program (e.g by calling `spawn`), and then use the `SCHED_APS_JOIN_PARTITION` command to *SchedCtl()* to make the program run in the appropriate partition. For example, this code makes the current process join a given partition:

```
sched_aps_join_parms join_data;

memset(&join_data, 0, sizeof(join_data));
join_data.id = partition_ID;
join_data.pid = 0;
join_data.tid = 0;

ret = SchedCtl( SCHED_APS_JOIN_PARTITION, &join_data,
                sizeof(join_data));
if (ret != EOK) {
   printf("Couldn't join partition %d: %s (%d).\n",
            join_data.id, strerror(errno), errno);
} else {
   printf ("Process is now in partition %d.\n", join_data.id);
}
```

Viewing partition use

The most common use of aps is to list the partitions and the CPU time they're using. To list partitions and the CPU time they're consuming, use the aps show command:

```
$ aps show
                    +---- CPU Time ----+-- Critical Time --
Partition name   id | Budget |    Used | Budget |      Used
--------------------+------------------+-------------------
System            0 |    60% |  36.24% |  100ms |   0.000ms
partitionA        1 |    20% |   2.11% |    0ms |   0.000ms
partitionB        2 |    20% |   1.98% |    0ms |   0.000ms
--------------------+------------------+-------------------
Total               |   100% |  40.33% |
```

To display CPU usage over the longer windows (typically 10 times and 100 times the length of the averaging window), add the -v option:

```
$ aps show -v
                    +----------- CPU Time ------------+-- Critical Time --
                    |        |          Used          |        |
Partition name   id | Budget | 0.100s   1.00s   10.0s | Budget |      Used
--------------------+---------------------------------+-------------------
System            0 |    60% | 20.91%   3.23%   4.33% |  100ms |   0.000ms
partitionA        1 |    20% |  1.78%   2.09%   2.09% |    0ms |   0.000ms
partitionB        2 |    20% |  1.71%   2.03%   2.03% |    0ms |   0.000ms
--------------------+---------------------------------+-------------------
Total               |   100% | 24.40%   7.34%   8.44% |
```

If you specify more than one v option, the aps utility's output results also shows you the critical budget usage over the longer windows.

If you want to display the output of the aps show command every 5 seconds, use the -l option in the command as in aps show -l. You can also use the -d option to change the length of the delay of the output results.

For more information about the aps utility, see the *Utilities Reference*.

Chapter 4
Controlling Resources Using the Thread Scheduler

The thread scheduler is a component of the QNX Neutrino adaptive partitioning architecture. The thread scheduler solves the problem of controlling the consumption of resources in a system. For example, we might want to control these resources to:

- prevent an application from consuming too many resources, such that it starves another application
- maintain a reserve of resources for emergency purposes, such as a disaster-recovery system, or a field-debugging shell
- limit well-behaved applications to a set share of the resource allocation for the system. For example, when a QNX Neutrino user builds a system that serves several end users, the QNX Neutrino user might want to bill their end users by the amount of throughput or capacity they are allocated on the shared system.

However, the details for controlling a resource are very different depending on the type of resource.

There are some usage and implementation questions to answer about thread scheduler partitions:

Question	Answer for the Thread scheduler
When do you get more of the resource?	When more time appears.
How much history of resource consumption does the adaptive partitioning system use to make decisions?	Time usage over the last 100 milliseconds (a rolling window). The 100ms is configurable; however, it is typically short.
Hierarchy of partitions: Does the partition size limit of a parent limit the size of the child partitions?	Yes. A child partition can never be given a larger CPU share than its parent partition. When a child scheduler partition is created, we subtract the child's budget (partition size) from the size of its parent so that a child is separate from its parent. Why: The hierarchical accounting rules needed for a child partition to be a component of a parent partition are too CPU-time intensive for scheduling because scheduling operations occur thousands of times every second, and continue forever.
Is there a limit to the number of partitions?	Yes. There is a maximum of eight scheduler partitions. Why: For every scheduling operation, the thread scheduler must examine every partition before it can pick a thread on which to run. That may occur 50000 times per second on a 700MHz x86 (i.e. a slow machine). So it's important to limit the number of scheduler partitions to keep the scheduler overhead to a minimum.

Question	Answer for the Thread scheduler
Is the hierarchy of partitions represented in a path namespace?	No. Scheduler partitions are named in a small flat namespace that is unique to the thread scheduler.
In what units are partitions sized?	The percentage of CPU time.
What do the terms *guaranteed*, *minimum size*, and *maximum size* mean for partitions?	The *size*, or budget of a scheduler partition is the guaranteed minimum amount of CPU time that threads (in partitions), will be allowed to consume over the next 100ms rolling window. Scheduler partitions don't have a maximum size (i.e. an amount of consumption that would cause the thread scheduler to stop running threads in a partition) because they were using too much of the system's resources. Instead, the thread scheduler allows a partition to overrun or exceed its budget when other partitions are not using their guaranteed minimums. This behavior is specific to scheduling. It's designed to make the most possible use of the CPU at all times (i.e. keep the CPU busy if at least one thread is ready to run).
What mechanism enforces partition consumption rules? When are these rules applied?	Every timer interrupt (typically, every millisecond), every message/pulse send, receive or reply, every signal, every mutex operation, and on every stack fault, and including many times for process manager operations (creation/destruction of processes or threads and *open()* operations on elements of the path namespace). Enforcement mechanism: If a partition is over budget (meaning that the consumption of CPU time over the last 100 milliseconds exceeds the partition's size, and other partitions are also demanding time) and a thread wants to run, the thread scheduler doesn't run the thread; it runs some other thread. Only when enough time has elapsed, so that the average CPU time use of that partition (over the last 100 milliseconds) falls below the partition's size, will the scheduler run the thread. However, the thread is guaranteed to eventually run.
Can we say that a partition has members? What is the member?	Yes, threads are members of scheduler partitions. We say they're running in a scheduler partition. However, a mechanism designed to avoid priority-inversion problems means that occasionally threads can temporarily move to other partitions. The different threads of a process may be in different scheduler partitions.
What utility-commands are used to configure partitions?	The `aps` command, using options for scheduler partitions only.

Chapter 5
Considerations for the Thread Scheduler

You typically use the thread scheduler to:

- engineer a system to work in a predictable or defined manner when it's fully loaded
- prevent unimportant or untrusted applications from monopolizing the system

In either case, you need to configure the parameters for the thread scheduler with the entire system in mind. The basic decisions are:

- How many scheduler partitions should you create, and what software should go into each?
- What guaranteed CPU percentage should each scheduler partition receive?
- What should be the critical budget, if any, of each scheduler partition?
- What size, in milliseconds, should the time-averaging window be?

Determining the number of scheduler partitions and their contents

It seems reasonable to put functionally-related software into the same scheduler partition, and frequently that's the right choice. However, adaptive partitioning thread scheduling is a structured way of deciding when *not* to run software. So the actual method is to separate the software into different scheduler partitions if it should be starved of CPU time under different circumstances.

The maximum number of partitions you can create is eight.

For example, if the system is a packet router that:

- routes packets
- collects and logs statistics for packet routing
- handles route-topology protocols with peer routers
- collects and logs route-topology metrics

it may seem reasonable to have two scheduler partitions: one for routing, and one for topology. Certainly logging routing metrics is functionally related to packet routing.

However, when the system is overloaded, meaning there's more outstanding work than the machine can possibly accomplish, you need to decide what work to do slowly. In this example, when the router is overloaded with incoming packets, it's still important to route them. But you may decide that if you can't do everything, you'd rather route packets than collect the routing metrics. By the same analysis, you might conclude that route-topology protocols should continue to run, using much less of the machine than routing itself, but run quickly when they need to.

Such an analysis leads to three partitions:

- a partition for routing packets, with a large share, say 80%
- a partition for topology protocols, say 15%, but with maximum thread priorities that are higher than those for packet routing
- a partition for logging both the routing metrics and topology-protocol metrics

In this case, we chose to separate the functionally-related components of routing and logging the routing metrics because we prefer to starve just one if we're forced to starve something. Similarly, we chose to group two functionally-unrelated components, the logging of routing metrics and the logging of topology metrics, because we want to starve them under the same circumstances.

Choosing the percentage of CPU for each partition

The amount of CPU time that each scheduler partition tends to use under unloaded conditions is a good indication of the budget you should assign to it. If your application is a transaction processor, it may be useful to measure CPU consumption under a few different loads and construct a graph of offered load versus the CPU consumed.

Typically, the key to obtaining the right combination of partition budgets is to try them:

1. Leave security turned off.
2. Load a test machine with realistic loads.
3. Examine the latencies of your time-sensitive threads with the IDE System Profiler tool.
4. Try different patterns of budgets, which you can easily change at run time with the `aps` command.

You can't delete partitions, but, you can remove all of its corresponding processes, and then change that specific partition's budget to 0%.

Setting budgets to zero

It's possible to set the budget of a partition to zero as long as the `SCHED_APS_SEC_NONZERO_BUDGETS` security flag *isn't* set—see the `SCHED_APS_ADD_SECURITY` command for *SchedCtl()*.

Threads in a zero-budget partition run only in these cases:

- All other nonzero-budget partitions are idle.
- The zero-budget partition has a nonzero critical budget, in which case its critical threads run.
- A thread receives a message from a partition with a nonzero budget, in which case the receiving thread runs temporarily in the sender's partition.

When is it useful to set the budget of a partition to zero?

It useful to set the budget of a partition to zero when:

- A partition is permanently empty of running threads; you can set its budget to zero to effectively turn it off. When a zero-budget partition is idle, it isn't considered to produce free time (see "*Summary of scheduling behavior* (p. 20)" in the Using the Thread Scheduler chapter of this guide). A partition with a nonzero budget that never runs threads puts the thread scheduler permanently in free-time mode, which may not be the desired behavior.
- You want noncritical code to run only when some other partition is idle.

- The partition is populated by resource managers, or other software, that runs only in response to receiving messages. Putting them in a zero-budget partition means you don't have to separately engineer a partition budget for them. (Those resource managers automatically bill their time to the partitions of their clients.)

Typically, setting a partition's budget to zero is not recommended. (This is why the `SCHED_APS_SEC_RECOMMENDED` security setting doesn't permit partition budgets to be zero.) The main risk in placing code into a zero-budget partition is that it may run in response to a pulse or event (i.e. not a message), and therefore, not run in the sender's partition. As a result, when the system is loaded (i.e. there's no free time), those threads may simply not run; they might hang, or things might occur in the wrong order.

For example, it's hazardous to set the System partition's budget to zero. On a loaded machine with a System partition of zero, requests to `procnto` to create processes and threads may hang, for example, when `MAP_LAZY` is used. In addition, if your system uses zero-budget partitions, you should carefully test it with all other partitions fully loaded with `while(1)` loops.

Setting budgets for resource managers

Ideally we'd like resource managers, such as filesystems, to run with a budget of zero. That way they'd always be billing time to their clients. However, some device drivers realize too late which client a particular thread was doing work for. Consequently, some device drivers may have background threads for audits or maintenance that require CPU time that can't be attributed to a particular client. In those cases, you should measure the resource manager's background and unattributable loads, and then add that amount to its partition's budget.

- If your server has maintenance threads that never serve clients, then it should be in a partition with a nonzero budget.
- If your server communicates with its clients by sending messages, or by using mutexes or shared memory (i.e. anything other than receiving messages), then your server should be in a partition with a nonzero budget.

You can set the size of the time-averaging window to be from 8 to 400 milliseconds. This is the time over which the thread scheduler attempts to balance scheduler partitions to their guaranteed CPU limits. Different choices of window sizes affect both the accuracy of load balancing and, in extreme cases, the maximum delays seen by ready-to-run threads.

Choosing the window size

You can set the size of the time-averaging window to be from 8 to 400 milliseconds. This is the time over which the thread scheduler attempts to balance scheduler partitions to their guaranteed CPU limits. Different choices of window sizes affect both the accuracy of load balancing and, in extreme cases, the maximum delays seen by ready-to-run threads.

Accuracy

Some things to consider:

- A small window size reduces the accuracy of CPU time balancing. The error is `+/-(`***`tick_size / window_size`***`)`. For example, if the window size is 10 milliseconds, the accuracy is about 10 percentage points.
- If a partition opportunistically goes over budget (because other partitions are using less than their guaranteed budgets), it must pay back the borrowed time, but only as much as the thread scheduler remembers (i.e. only the borrowing that occurred in the last window).

  A small window size means that a scheduler partition that opportunistically goes over budget might not have to pay the time back. If a partition sleeps for longer than the window size, it won't get the time back later. So load balancing won't be accurate over the long term if both the system is loaded, and some partitions sleep for longer than the window size.
- If the window size is small enough that some partition's percentage budget becomes less than a tick, the partition will get to run for at least 1 tick during each window, giving it 1 tick / *window_size_in_ticks* per cent of the CPU time, which may be considerably larger than the partition's actual budget. As a result, other partitions may not get their CPU budgets.

Delays compared to priority scheduling

In an underload situation, the thread scheduler doesn't delay ready-to-run threads, but the highest-priority thread might not run if the thread scheduler is balancing budgets.

In very unlikely cases, a large window size can cause some scheduler partitions to experience runtime delays, but these delays are always less than what would occur without adaptive partitioning thread scheduling. There are two cases where this can occur.

Case 1

If a scheduler partition's budget is *budget* milliseconds, then the delay is never longer than:

window_size – *smallest_budget* + *largest_budget*

This upper bound is only ever reached when low-budget and low-priority scheduler partitions interact with two other scheduler partitions in a specific way, and then only when all threads in the system are ready to run for very long intervals. This maximum possible delay has an extremely low chance of occurring.

For example, given these scheduler partitions:

- Partition A: 10% share; always ready to run at priority 10
- Partition B: 10% share; when it runs, it runs at priority 20
- Partition C: 80% share; when it runs, it runs at priority 30

This delay happens when the following occurs:

- Let B and C sleep for a long time. A will run opportunistically and eventually run for 100 milliseconds (the size of the averaging window).
- Then B wakes up. It has both available budget and a higher priority, so it runs. Let's call this time *Ta*, since it's the last time partition A ran. Since C continues to sleep, B runs opportunistically.
- At *Ta* + 90 milliseconds, partition A has just paid back all the time it opportunistically used (the window size minus partition A's budget of 10%). Normally, it would run on the very next tick because that's when it would next have a budget of 1 millisecond, and B is over budget.
- But let's say that, by coincidence, C chooses to wake at that exact time. Because it has budget and a higher priority than A, it runs. It proceeds to run for another 80 milliseconds, which is when it runs out of budget.
- Only now, at *Ta* + 90 ms + 80 ms, or 170 milliseconds later, does A get to run again.

This scenario can't occur unless a high-priority partition wakes up exactly when a lower-priority partition just finishes paying back its opportunistic run time.

Case 2

Still rare, but more common, is a delay of:

window_size – *budget*

milliseconds, which may occur to low-budget scheduler partitions with, on average, priorities equal to other partitions.

With a typical mix of thread priorities, when ready to run, each scheduler partition typically experiences a maximum delay of much less than the *window_size* milliseconds.

For example, let's suppose we have these scheduler partitions:

- partition A: 10% share, always ready to run at priority 10
- partition B: 90% share, always ready to run at priority 20, except that every 150 milliseconds, it sleeps for 50 milliseconds.

This delay occurs when the following happens:

- When partition B sleeps, partition A is already at its budget limit of 10 milliseconds (10% of the window size).
- But then A runs opportunistically for 50 milliseconds, which is when B wakes up. Let's call that time *Ta*, the last time partition A ran.
- B runs continuously for 90 milliseconds, which is when it exhausts its budget. Only then does A run again; this is 90 milliseconds after *Ta*.

However, this pattern occurs only if the 10% application never suspends (which is exceedingly unlikely), and if there are no threads of other priorities (also exceedingly unlikely).

Approximating the delays

Because these scenarios are complicated, and the maximum delay time is a function of the partition shares, we approximate this rule by saying that the maximum ready-queue delay time is twice the window size.

If you change the tick size of the system at runtime, do so before defining the windows size of the partition thread scheduler, because QNX Neutrino converts the window size from milliseconds to clock ticks for internal use.

The practical way to verify that your scheduling delays are correct is to load your system with stress loads, and use the System Profiler tool from the IDE to monitor the delays. The `aps` command lets you change budgets dynamically, so you can quickly confirm that you have the correct configuration of budgets.

Practical limits

If you use adaptive partitions, you need to be aware of the following limitations:

- The API allows a window size as short as 8 milliseconds, but practical window sizes may need to be larger. For example, in an eight-partition system, with all partitions busy, to reasonably expect all eight to run during every window, the window size needs to be at least 8 timeslices long, which for most systems is 32 milliseconds.
- Overloads aren't reported to users. The Adaptive Partition scheduler does detect overload and acts to limit some partitions to guarantee the percentage shares of others, but it doesn't inform anything outside of the kernel that an overload was detected. The problem is that an overload might occur (or might not occur) on every scheduling operation, which can occur at the rate of 50000 per second on a 200mhz machine (an older, slower machine).
- `SCHED_RR` threads might not round robin in partitions whose portion of the averaging window is smaller than one timeslice. For example, when the timeslice is 4 ms (the default) and the adaptive partitioning scheduler's window size is 100 ms (the default), then `SCHED_RR` threads in a 4% partition may not round-robin correctly.
- If you use adaptive partitioning and bound multiprocessing (BMP), some combinations of budgets might not be met. Threads in a zero-budget partition should run only when all other nonzero-budget partitions are idle. On SMP machines, zero-budget partitions may incorrectly run when some other partitions are demanding time. At all times, all partitions' minimum budgets are still guaranteed, and zero-budget partitions won't run if all nonzero-budget partitions are ready to run. For detailed information, see *Using the thread scheduler and multicore together* (p. 29).
- To calculate the total microcycle used in a window size, the product of $P \times W \times N$ must be less than 2,147,483,648, where:

  - P is the processor clock rate (in Hz)
  - W is the APS window size (in seconds)
  - N is the number of processors on the SMP device

  The default value of W is 0.1 (100 milliseconds) and, given this value, the following constraints apply:

  - 1 processor: maximum clock rate 21.5 GHz
  - 2 processors: maximum clock rate 10.7 GHz
  - 4 processors: maximum clock rate 5.4 GHz
  - 8 processors: maximum clock rate 2.7 GHz

- As reported by the `aps show -v` command on ARM targets, the 10 and 100 window averages occasionally give incorrect information, but this has no effect on scheduling.

Uncontrolled interactions between scheduler partitions

There are cases where a scheduler partition can prevent other applications from being given their guaranteed percentage CPU:

Interrupt handlers

The time used in interrupt handlers is never throttled. That is, we always choose to execute the globally highest-priority interrupt handler, independent of its scheduler partition. This means that faulty hardware or software that causes too many interrupts can effectively limit the time available to other applications.

Time spent in interrupt threads (e.g. those that use *InterruptAttachEvent()*) is correctly charged to those threads' partitions.

Chapter 6
Security for Scheduler Partitions

By default, anyone on the system can add partitions and modify their attributes. We recommend that you use the `SCHED_APS_ADD_SECURITY` command to *SchedCtl()*, or the `aps` command to specify the level of security that suits your system.

Managing security for the thread scheduler

The following list shows the main security options (including both the option for the `aps` command and the corresponding *SchedCtl()* flag), in increasing order of security:

none or the SCHED_APS_SEC_OFF flag

Anyone on the system can add partitions and modify their attributes.

`basic` or the SCHED_APS_SEC_BASIC flag

Only `root` in the System partition can change the overall scheduling parameters and set critical budgets.

`flexible` or the SCHED_APS_SEC_FLEXIBLE flag

Only `root` in the `System` partition can change scheduling parameters or change critical budgets. However, `root` running in any partition can create subpartitions, join threads into its own subpartitions and modify subpartitions. This lets applications create their own local subpartitions out of their own budgets. The percentage for budgets must not be zero.

`recommended` or the SCHED_APS_SEC_RECOMMENDED flag

Only `root` from the System partition can create partitions or change parameters. This arranges a 2-level hierarchy of partitions: the System partition and its children. Only `root`, running in the System partition, can join its own thread to partitions. The percentage for budgets must not be zero.

Unless you're testing the partitioning aspects and want to change all of the parameters without restarting, you should set at least `basic` security.

After setting up the scheduler partitions, you can use `SCHED_APS_SEC_PARTITIONS_LOCKED` to prevent further unauthorized changes. For example:

```
sched_aps_security_parms p;

APS_INIT_DATA( &p );
p.sec_flags = SCHED_APS_SEC_PARTITIONS_LOCKED;
SchedCtl( SCHED_APS_ADD_SECURITY, &p, sizeof(p));
```

Before you call *SchedCtl()*, ensure that you initialize *all* the members of the data structure associated with the command. You can use the *APS_INIT_DATA()* macro to do this.

The security options listed above are composed of the following options (but it's more convenient to use the compound options):

`root0_overall` or the `SCHED_APS_SEC_ROOT0_OVERALL` flag

You must be `root` running in the System partition in order to change the overall scheduling parameters, such as the averaging window size.

`root_makes_partitions` or the `SCHED_APS_SEC_ROOT_MAKES_PARTITIONS` flag

You must be `root` in order to create or modify partitions.

`sys_makes_partitions` or the `SCHED_APS_SEC_SYS_MAKES_PARTITIONS` flag

You must be running in the System partition in order to create or modify partitions.

`parent_modifies` or the `SCHED_APS_SEC_PARENT_MODIFIES` flag

Allows partitions to be modified (`SCHED_APS_MODIFY_PARTITION`), but you must be running in the parent partition of the partition being modified. Modify means to change a partition's percentage or critical budget, or attach events with the SCHED_APS_ATTACH_EVENTS command.

`nonzero_budgets` or the `SCHED_APS_SEC_NONZERO_BUDGETS` flag

A partition may not be created with, or modified to have, a zero budget. Unless you know your partition needs to run only in response to client requests, i.e. receipt of messages, you should set this option.

`root_makes_critical` or the `SCHED_APS_SEC_ROOT_MAKES_CRITICAL` flag

You have to be `root` in order to create a nonzero critical budget or change an existing critical budget.

`sys_makes_critical` or the `SCHED_APS_SEC_SYS_MAKES_CRITICAL` flag

You must be running in the System partition to create a nonzero critical budget or change an existing critical budget.

`root_joins` or the `SCHED_APS_SEC_ROOT_JOINS` flag

You must be `root` in order to join a thread to a partition.

`sys_joins` or the `SCHED_APS_SEC_SYS_JOINS` flag

You must be running in the System partition in order to join a thread.

`parent_joins` or the `SCHED_APS_SEC_PARENT_JOINS` flag

You must be running in the parent partition of the partition you wish to join.

`join_self_only` or the `SCHED_APS_SEC_JOIN_SELF_ONLY` flag

A process may join only itself to a partition.

`partitions_locked` or the `SCHED_APS_SEC_PARTITIONS_LOCKED` flag

Prevent further changes to any partition's budget, or overall scheduling parameters, such as the window size. Set this after you've set up your partitions.

Security and critical threads

Any thread can make itself critical, and any designer can make any `sigevent` critical (meaning that it will cause the eventual receiver to run as critical), but this isn't a security issue. That's because a thread marked as critical has no effect on the thread scheduler unless the thread is in a partition that has a critical budget. The thread scheduler has security options that control who may set or change a partition's critical budget.

For the system to be secure against possible critical thread abuse, it's important to:

- assign a critical budget only to the partitions that need one
- move as much application software as possible out of the `System` partition (which has an infinite critical budget)

Chapter 7
Testing and Debugging

Adaptive partitioning can even make debugging an embedded system easier during development or deployment by providing an “emergency door” into the system. Simply create a partition that you can run diagnostic tools in; if you don't need to use the partition, the thread scheduler allocates its budget among the other partitions. This provides you with access to the system without compromising its performance.

Instrumented kernel trace events

The instrumented kernel emits trace events when:

- a scheduler partition's budget changes (including when the partition is created)
- a scheduler partition's name changes (i.e. when the partition is created)
- a thread runs —in wide mode, these events include the partition ID and indicate whether or not the thread is running as critical
- a scheduler partition becomes bankrupt

In addition, all events include the scheduler partition ID and its budget. You can use `traceprinter` to display the contents of the trace file. You can also use the IDE to parse and display a trace file.

Using the IDE (trace events)

You can—and should—use the System Profiler tool from the IDE to check your system's latencies. For more information about using this tool and the IDE, see the IDE *User's Guide*.

Using other methods

The easiest method to test a system that uses the thread scheduler is from the command line.

Be sure to test your system in a fully loaded state, because that's where problems are likely to occur. Create a program that consumes resources by looping forever, run it in each partition, and then do the following:

- Watch for bankruptcies, which you should consider to be programming errors. You can use `SchedCtl` with the `SCHED_APS_BNKR_*` flags to control what happens when a partition exhausts its critical budget. This can range from delivering an event to rebooting the system. For more information, see the QNX Neutrino *Library Reference*.
- Ensure that all latencies are acceptable for your system.
- Use the `aps` command to change your partitions' budgets. The new budgets come into effect at the beginning of the next averaging window. Since the window size is typically 100 ms, you can quickly try many different combinations of budgets.

Emergency access to the system

You can use adaptive partitioning to make it easier to debug an embedded system by providing emergency access to it:

- during development — create a partition and start `io-pkt` and `qconn` in it. Then, if a runaway process ties up the system, you can use the IDE to debug and query the system.
- during deployment — create a partition and start `io-pkt` and `inetd` in it. If you encounter a problem, you can `telnet` into the system.

In either case, if you don't need to use this partition, the thread scheduler allocates its budget among the other partitions. This provides you with emergency access to the system without compromising performance.

A different approach is to create a separate partition in which to run any questionable applications; if they run wild, the rest of the system should continue to work normally.

Appendix A
Sample Buildfile

The following buildfile shows you how to add the adaptive partitioning module to `procnto`. It also includes commands that create partitions, start a program in the partition, and set the security level.

In a real buildfile, you can't use a backslash (\) to break a long line into shorter pieces, but we've done that here, just to make the buildfile easier to read.

```
[compress=3]
[virtual=x86,bios] .bootstrap = {
    startup-bios

    # PATH is the *safe* path for executables (confstr(_CS_PATH...))
    # LD_LIBRARY_PATH is the *safe* path for libraries
    # (confstr(_CS_LIBPATH)). That is, it's the path searched for libs
    # in setuid/setgid executables.

    # The module=aps enables the adaptive partitioning scheduler.

    [module=aps] PATH=/proc/boot:/bin:/usr/bin:/sbin:/usr/sbin \
        LD_LIBRARY_PATH=/proc/boot:/lib:/lib/dll:/usr/lib \
        procnto-smp-instr
}

# Start-up script

[+script] .script = {
    # Programs require the runtime linker (ldqnx.so) to be at a fixed
    # location. To save memory, make everyone use the libc in the boot
    # image. For speed (fewer symbolic lookups), we point to libc.so.3
    # instead of libc.so.
    procmgr_symlink ../../proc/boot/libc.so.3 /usr/lib/ldqnx.so.2

    # Create some adaptive partitions during system startup:
    #   - IOPKT with a 20% budget
    #   - QCONN with a 20% budget
    # NOTE: To specify a critical budget, use the "aps modify" command
    #       when the filesystem on the disk is available.
    sched_aps IOPKT 20
    sched_aps QCONN 20

    # Start the system logger.
    slogger &
    slogger2 &
    dumper -d /var/dumps

    # Start the PCI server.
    seedres
    pci-bios &
    waitfor /dev/pci

    display_msg .
    display_msg Welcome to QNX Neutrino 6.6.0 on an PC-compatible
    display_msg BIOS system with APS enabled

    # Get the disk up and running.
    devb-eide blk automount=hd0t179:/ &

    waitfor /bin
    # Further commands can now be run from disk.

    # USB services
    display_msg "Start USB services..."
    io-usb -dehci &
    waitfor /dev/io-usb/io-usb 4
    waitfor /dev/io-usb/devu-ehci.so 4

    display_msg "Starting Input services..."
    io-hid -d ps2ser kbd:kbddev:ps2mouse:mousedev \
       -d usb /dev/io-usb/io-usb &
    waitfor /dev/io-hid/io-hid 10

    # Start up some consoles.
    display_msg Starting consoles
    devc-pty &
    devc-con-hid -n4 &
    reopen /dev/con1

    display_msg Starting serial port driver
    devc-ser8250 -b115200 &

    # Start the networking manager in the IOPKT partition. Don't
    # forget devnp.shim.so for compatibility with io-net drivers.
    on -X aps=IOPKT io-pkt-v4-hc -d /lib/dll/devn-pcnet.so &

    # Start some services.
    pipe &
    random -t &

    waitfor /dev/io-net/en0
    dhcp.client -i en0 -m -u -t1 -A 0

    #waitfor /dev/dbgmem

    # Create an additional parition with services:
    aps create -b10 INETD
    on -X aps=INETD inetd &
    on -X aps=QCONN qconn &

    # Specify a critical budget for a partition created during startup:
    aps modify -B 10 IOPKT

    # Use the "recommended" security level for the partitions:
    aps modify -s recommended

    # These env variables are inherited by all the programs that follow:
    SYSNAME=nto
    TERM=qansi

    # Start some extra shells on other consoles:
    reopen /dev/con2
    [+session] sh &
    reopen /dev/con3
    [+session] sh &
    reopen /dev/con4
    [+session] sh &

    # Start the main shell
    reopen /dev/con1
    [+session] sh &
}

[perms=0777]
# Include the current "libc.so". It will be created as a real file
# using its internal "SONAME", with "libc.so" being a symlink to it.
# The symlink will point to the last "libc.so.*" so if an earlier
# libc is needed (e.g. libc.so.3) add it before the this line.
libc.so
libelfcore.so.1
```

```
libslog2.so


###########################################################################
## uncomment for USB driver
###########################################################################
#libusbdi.so
devu-ehci.so

fs-qnx6.so
fs-qnx4.so
cam-disk.so
io-blk.so
libcam.so

#devn-pcnet.so
#devnp-shim.so
#libsocket.so

[data=copy]

devc-con
pci-bios
seedres
devc-ser8250
dumper
devb-eide
mount
slogger
slogger2
sh
cat
ls
pidin
less
```

Appendix B
Frequently Asked Questions: Adaptive Partitioning Thread Scheduler

This appendix contains a set of frequently asked questions (FAQ) and answers. It covers the implementation details of the scheduling extensions contained in QNX Neutrino adaptive partitioning, as well as any common questions related to partitions in general.

The information contained in this Frequently Asked Questions section is subject to change at any time without notice. QSS makes no representation or warranty regarding the information and is not responsible whatsoever for reliance on the information contained herein.

Scheduling behavior

How does the thread scheduler guarantee a partition's minimum CPU budget?

The thread scheduler guarantees a minimum CPU budget by ensuring that other partitions don't overrun their budget. This determination is made every clock tick.

The clock interrupt handler invokes the thread scheduler. That means it runs a miniimum of every clock period (typically every millisecond). On each clock tick:

- On a uniprocessor, it examines the partition of the currently running thread to see if it should keep running. The adaptive partition (AP) thread scheduler will decide that a thread should stop running if its partition has less available time (budget-cycles minus used-cycles during this averaging window) than what is necessary to pay for the duration of the next clock period. If the currently running partition fails this test, then the AP portion of the clock handler sets a "must examine all partitions" flag.
- On an SMP processor, the AP scheduler's portion of the clock interrupt handler always sets the "must examine all partitions" flag.
- If the currently running partition fails this test, then the adaptive partitioning portion of the clock handler sets a "must examine all partitions" flag.

On exit from the QNX Neutrino clock interrupt handler, the handler examines the flag. If set, it causes the system to immediately enter the kernel and invoke the full scheduling algorithm.

The full thread scheduler algorithm examines all partitions. It stops running the current partition if it's about to run out of budget (i.e. it no longer has enough to pay for another quarter clock period, in addition to one clock period for each additional CPU — if the system is multicore). In other words, the thread scheduler guarantees that budgets are met by forcing a partition to temporarily stop running if it will run over its budget before the next time the scheduler is in control of the system. This also requires that some other partition has budget and threads that are ready to run.

When does the scheduler guarantee that a partition gets its budget?

The thread scheduler guarantees that budgets are met by forcing a partition to temporarily stop running if it runs over its budget before the next time when the scheduler is in control of the system.

The thread scheduler makes sure that a partition gets at least its budget in the current averaging window when:

- The partition becomes ready to run often enough to consume at least its budget worth of time.
- On multicore machines:

    - let `B(p)` be the budget, in percent of partition `p`
    - let `R(p)` be the number of ready to run threads in our partition, and
    - let `N` be the number of CPUs

    Then the thread scheduler guarantees that partition `p` gets `B(p)` percent of CPU time over the last averaging window if:

    ```
    R(p) >= N * B(p)/100
    ```

    In other words, it means that *when* the partition has enough ready to run threads to occupy the processors in the system.

- The scheduler didn't bill any critical time to any partition.

In other words, budgets are guaranteed if the system is busy enough and no partition has used its critical budget.

Does a 100-ms window mean a CPU time-averaging occurs only once in every 100 ms?

See the next answer.

How often does the algorithm enforce partition budgets?

A 100-ms averaging window stores a detailed history of CPU usage for each of the last 100 millisecond intervals. Rather, it stores a history of CPU usage, with detail for each of the last 100 millisecond intervals. The window rotates, or slides forward in time, for every clock tick. So the window provides precise information about the average CPU consumption every millisecond (or clock period).

Between clock ticks, when the thread scheduler algorithm is called, CPU usage of each partition is approximated with the assumption that each partition will likely run continuously at least until the next clock tick.

In other words, the thread scheduler computes the used CPU time and enforces the budgets, many times per millisecond.

What system assumptions does the design of thread scheduler make?

In order to guarantee that the partitions get their guaranteed minimum CPU budgets, the design assumes:

- The clock interrupt handler runs periodically. In other words, the users don't inhibit clock interrupts.
- The *ClockCycles()* function is monotonic, except for 64-bit wraparound.
- The *ClockCycles()* function increases at a constant rate.
- Useful work done by the processor is proportional to *ClockCycles()*.
- The *ClockCycles()* functions, as seen by each CPU on an SMP. machine, increment at the same rate (though there may be a constant offset between each processor's *ClockCycles()*).
- Each CPU works at the same rate on SMP machines.

- The resolution of *ClockCycles()* is at least 1/200th of the clock period between timer ticks.
- The user doesn't change the size of the averaging window often.

When does the thread scheduler calculate percentage CPU usage?

Never. It avoids doing division in order to execute quickly.

The scheduler only compares a partition's CPU usage with its budget, expressed as a total time over the last averaging window rather than as a percentage. To make a quick comparison, both usage and budgets are treated internally as counts of *ClockCycles()*, not as percentages.

How often does the thread scheduler compute CPU usage?

At least once every clock period (typically every millisecond). However, it also does it on kernel calls, such as message and pulse sending or mutex releases. For example, on a 733MHz x86 machine that performs a lot of I/O, the scheduler computes CPU usage around 50 times every millisecond.

When is the scheduler's behavior realtime?

Within a single partition, the thread scheduler always follows POSIX scheduling rules, i.e. preemptive priority-based scheduling with FIFO and sporadic policies. So a partition looks somewhat like a complete system in POSIX.

However the CPU time, seen by a partition, may be sliced by threads running in other partitions.

So the question remains: when does a partition get continuous realtime? Since our definition of realtime is to schedule strictly by priority, the answer is the thread scheduler schedules strictly by priority whenever a set of partitions has used less than their budgets over the last averaging window. This implies that all threads run by priority-preemption rules as long as their partitions have not exhausted their budget in the current averaging window. In brief, it's realtime when a partition is using less than its budget.

What is free-time mode?

See the next answer.

What is free time?

Free-time mode is a specific budget situation when at least one partition with a nonzero budget isn't using all of its budget. Free-time mode means other partitions may use up the free time even if they exceed their own budgets. This is one of the reasons why adaptive partitioning is *adaptive*.

The extra time a partition gets in free time mode is called free time, but it isn't always free; sometimes it must be paid back.

Do you have to repay free time?

Partly. In general, only the free time during the last averaging window needs to be paid back.

For example, suppose that partition p1 has exhausted its budget, and another partition p2 has available budget. Therefore partition p2 is running. Now assume that partition p2 becomes idle (i.e. goes to sleep) for 10 milliseconds. Because partition p2 has no competition and is in free-time mode, partition p1 begins running and exceeds its budget by 10 milliseconds.

Now, say partition p2 wakes up. The partition p2 won't run until the averaging window rotates enough to carry the history of its CPU over-usage past 100 milliseconds into the past. So, p2 may not run until *window-size – budget* milliseconds passes. This interval, where p2, is suspended is effectively paying back the free time.

In general, when free time is less than window size — budget must be paid back.

In a different example, suppose partition p2 goes to sleep for a minute. In this situation, partition p1 runs opportunistically and subsequently consumes 100% of the CPU. When partition p2 wakes up, it will have available budget, and partition p1 will be over budget, so partition p1 will run.

The partition p2 won't run again until window rotation removes history of its CPU usage past 100 milliseconds in the past. So in this case, partition p2 needs to pay back only *window-size – budget* milliseconds of the minute of CPU time that ran because partition p1 was asleep.

While the partition is over budget (because of the free time it received) — it won't run at all until enough time has passed to cause the total usage (recorded in the averaging window) to fall below budget. It implies that the partition has stopped running until its stopped time compensates for the free time it took earlier.

An exception is free time that occurred just before a call to `SchedCtl(SCHED_APS_SET_PARMS,...)` to change the window size. Changing the window size wipes the scheduler's memory so free time just before a change in window size isn't paid back.

How does the thread scheduler behave on HyperThreaded (HT) processors?

Adaptive partitioning treats a two-headed HT processor as a multicore system with two CPUs. It assumes that each virtual processor has equal and constant throughput. Whereas this is true for SMP machines, it's true on HT machines only when the system is sufficiently loaded to keep both pseudo-CPUs busy. Adaptive partitioning requires that a system's throughput be proportional to the *ClockCycles()* function.

How long can a round-robin thread run with the thread scheduler?

Without the thread scheduler (i.e. using classic QNX Neutrino scheduling), a round-robin thread:

- may be preempted at any time by a higher-priority thread

- if not preempted or if there is no other thread at the same priority, it runs until it gives up control voluntarily
- if not preempted and there is another thread at equal priority, it runs for 4 ticks (nominally 4 milliseconds) before being time-sliced with other thread

With the thread scheduler, a round-robin thread:

- may be preempted at any time by a higher priority thread in the same scheduler partition
- runs until it gives up control or its partition runs out of budget — if not preempted, and if there is no other thread of the same priority in that partition
- may start running if its partition gets more budget on the next clock tick. This happens for a ready to run round-robin thread (in a partition that is out of budget). This also happens since the rotation of the window gives that available partition budget back.
- runs for 4 ticks (nominally 4 milliseconds), before being time-sliced with the other thread — if its partition has
    - at least 4 milliseconds of available budget
    - not been preempted
    - another thread of equal priority

The scheduler overrides the time slice for a round-robin thread. When a partition has more than 4 ticks of available time left in its budget, thread scheduler behavior is the same as the classic QNX Neutrino scheduling. However on a loaded system, it's best to assume that a Round-Robin thread may be sliced every tick.

When a round-robin thread is preempted by the scheduler, it will be able to run a thread in a different partition. In other words, round-robin behavior is unchanged relative to the other threads in the same partition.

How long can a FIFO thread run with the thread scheduler?

Without the thread scheduler, if not preempted by a higher priority thread, a FIFO thread runs until it gives up control voluntarily.

With the thread scheduler, a FIFO thread runs if not preempted by a higher priority thread in the same partition until it gives up control voluntarily, or its partition runs out of budget.

FIFO behavior is unchanged as long as your partition has budget. On a loaded system, it's best to assume that a FIFO thread may be time sliced every millisecond with threads in other partitions. However, relative to all other threads in the same partition, FIFO behavior is the same as in classic QNX Neutrino scheduling.

How long can a sporadic (SS) thread run with the thread scheduler?

Without the thread scheduler, if not preempted by a higher priority thread, an SS thread runs until it gives up control voluntarily. Since the priority of an SS thread alternates between normal and low priorities, it's likely to be preempted when running at its low priority.

With the thread scheduler, the SS thread runs if not preempted by a higher priority thread in the same partition until it gives up control voluntarily or its partition runs out of budget.

Some developers set the higher priority of a sporadic-scheduled thread to be the highest priority in the system, in order to make the thread nonpreemptible during its high-priority mode. With the thread scheduler, the thread is non-preemptible only as long as its partition hasn't exhausted its budget.

Sporadic scheduling behavior is unchanged as long as your partition has budget. On a loaded system, it's best to assume that an `SS` thread may be time-sliced every millisecond with threads in other partitions. However, relative to all other threads in the same partition, SS behavior is the same as in classic QNX Neutrino scheduling.

How often does the thread scheduler algorithm run?

See the next answer.

How often does the thread scheduler enforce budgets?

The thread scheduler runs and enforces budgets:

- every clock tick
- every time a thread sleeps or blocks for a mutex
- whenever a thread becomes ready after it has received an event, pulse, or message.

The frequency depends on how often messaging occurs.

How do power-saving modes affect scheduling?

If the system suspends, scheduler is unaware of the interruption. Upon resumption, partitions will have the same percentage consumption they had at suspension.

If the system varies the processor speed to conserve power, scheduler is unaware of the variation. Although the scheduler guarantees that all partitions get their budget percentages, it assumes that each millisecond has the same throughput. This means that partition budget enforcement is effectively inaccurate for the 100 milliseconds (or window size) after the CPU changes speed. Thereafter, it's inaccurate.

How does changing the clock period (using ClockPeriod()) affect scheduling?

If you change the clock period, the thread scheduler can't schedule accurately because it's unaware of the change in the size of the tick. However, calling `SchedCtl(SET_APS_PARMS,...)` with the existing window size causes the scheduler to recalculate all internal parameters that depend on the size of the clock period. Correspondingly this calling restores accuracy.

As described in the Adaptive Partitioning `User's Guide`, you should set the window size after changing the clock period.

Microbilling

How does microbilling work?

Microbilling refers to the accounting for the CPU time that is used by a thread to a much finer resolution than the clock period between tick interrupts.

The thread scheduler has been implemented where threads send or receive messages many times (as opposed to a single time) per clock period. Adaptive partitioning scheduling would not be possible if we were limited to counting integer ticks of CPU time. That's because most threads send or receive messages, or otherwise block, many times per clock period.

Microbilling works by taking a fine-resolution timestamp every time a thread changes state from ready to not-ready, and charging differences between sequential timestamps against that partition's used CPU cycles count.

Microbilling uses the system call *ClockCycles()* to get that fine-resolution timestamp.

How often does thread scheduler microbill?

The thread scheduler microbills each time that:

- one thread stops running and another starts running
- a clock tick occurs

How does ClockCycles() work?

The thread scheduler always depends on the processor being used. On x86 processors, QNX Neutrino uses a free-running counter that is implemented on the CPU chip itself. This counter is read with a single instruction.

On x86 processors, *ClockCyles()* increase by about 1 billion counts every second on a 1 GHz processor.

On processors that don't have a free-running counter for the purpose of being a fine-grained clock, QNX Neutrino emulates *ClockCyles()*. For example, on ARM processors, QNX Neutrino reads the intermediate value of the countdown timer that's used to trigger the clock interrupts. This value tells how far you're into the current clock tick. QNX Neutrino further adds a scaled version of how far you're into the current clock tick to a constant determined at the last clock tick to get an emulated *ClockCycles()* value.

On some processors, such as ARM, the countdown timer used for emulating *ClockCycles()* is located off-chip and requires slow I/O operations to read it. On other processors, the countdown timer is located on-chip, and can be quickly read.

How accurate is microbilling?

See the next answer.

How accurate is ClockCycles()?

The accuracy of microbilling or *ClockCycles()* is determined by the accuracy of the clock oscillator source used in the CPU. However, since the scheduling is relative between partitions, it doesn't require *ClockCycles()* be equal to the absolute time; it requires only that *ClockCycles()* be proportional to the work done by CPU. In fact, a wrongly calibrated *ClockCycles()* has no effect on the accuracy of the thread scheduler.

What is the resolution of thread timing?

It's the resolution of the *ClockCycles()* function. The resolution of clock cycles varies from platform to platform. In most cases, the resolution is much finer.

The thread scheduler requires `1/200` of a tick to meet its specification for accuracy. In some platforms, such as x86, the resolution is on the order of nanoseconds.

Averaging window

How does the averaging window work?

The averaging window consists of tables. There are two tables per partition, one for the CPU time spent while critical, and another for any CPU time spent. The tables have one slot per timer tick. So a 100-ms averaging window, with a 1-ms clock period, has 100 slots. Each slot is used to hold the CPU time spent during a particular tick interval. For example:

```
[99ms ago][98 ms ago][97 ms ago]....[1 ms ago][current ms]
```

The slots hold the total CPU times of all threads in that partition as measured by consecutive calls to *ClockCycles()*. Note that total CPU times are then scaled by a carefully chosen factor so that all numbers fit into a 32-bit unsigned integer register.

At any time, the sum of the elements of a table represents the total CPU time used by that partition over the averaging period.

When the scheduler stops a thread running, it adds the time spent by that thread since when it started, or since the last tick, into the `current ms` slot of the table. If the thread was running as critical, the scheduler also adds the time to the `current ms` slot of that partition's critical time table. The scheduler also does this when a clock tick occurs.

However, on a clock tick, after billing the current thread to its partition's [`current ms`] slot, the scheduler also rotates the table. To rotate the table, it does the following:

- deletes the 99ms-ago slot
- shifts the entire table to the left by one slot, moving the time in the 98ms-ago slot to the 99ms-ago slot etc.
- creates a new current-ms slot, which the scheduler initializes to zero

This is called *window rotation*. Each rotation effectively provides available budget back to the partition that ran 99 ms ago. Window rotation is implemented without summing the entire table, shifting the table, or calls to the *malloc()* or *free()* functions.

What is the window-rotation algorithm?

The averaging window isn't physically rotated. It's logically rotated:

- A separate field, `used_cycles`, is always maintained to contain the total of every slot in the table.
- An integer, `cur_hist_index`, is an index into the table and points to the slot for the `current ms`.

- On microbilling, the scheduler adds the CPU time of the current thread to both the current slot in the table, and also to the total field. For example:

  ```
  usage_hist[cur_hist_index] += delta_time;
  used_cycles += delta_time;
  ```

- On window rotation, the scheduler does the following:
  - subtracts the oldest slot from the total:

    ```
    used_cycles -= usage_hist[(cur_hist_index +1) MOD 100]
    ```

  - increments the table index, modulo its table size (say 100):

    ```
    cur_hist_index = (cur_hist_index+1) MOD 100
    ```

  This is done for every partition, for both normal and critical CPU time.

Can I change the window size?

See the next answer.

How does changing the window size affect scheduling?

You can change the window size with the `SchedCtl(SCHED_APS_SET_PARMS,...)` on the fly. The scheduler doesn't *malloc()* new tables, but it does zero the history in all tables, zeros all the totals, and zeros the table indexes.

The effect is to wipe the memory of the scheduler. Here the scheduler assumes that no partition has run in the last *x* ms, where *x* is the new window size.

We recommend you leave the window size at the default, or set it during startup. Also, you shouldn't change the window size often.

How do maximum latencies relate to the averaging window size?

In general, the longer the averaging window, the longer the partition has to wait before it gets the CPU time.

For example, with a 100 milliseconds averaging window and a partition p with a 10% budget, the partition p will exhaust its budget if it runs continuously for 10 milliseconds. It has to wait another 90 milliseconds before window rotations cause the averaging window to lose memory of its past execution. So, it will be 90 milliseconds before the partition p gets some available budget back and runs again.

However, in most real systems that engage in inter-partition interaction, partition p's 10 milliseconds of running time is likely to get spread out in the averaging window. So even if p exhausts the budget soon, it will most likely get available budget back in much less than 90 milliseconds.

The Adaptive Partitioning *User's Guide* describes an unlikely scenario where two interacting partitions result in a larger latency than the window size budget.

Scheduling algorithm

How does the thread scheduler pick a thread to run?

See the next answer.

How does the scheduling algorithm work?

The thread scheduler evaluates a merit function on each partition and chooses the partition with the highest merit. It then picks the highest-priority thread in that partition. A partition with budget has more merit than a partition that has exhausted its budget.

First, let's look at a few helper functions. The details are provided below:

- The `COMPETING(p)` function is a boolean function of partition `p`. It returns `True` if:
    - partition `p` is currently running a thread of priority greater than zero

      Or:
    - partition `p` contains a thread that is ready to run, and has a priority greater than zero
- The `HAS_BUDGET(p)` function is a boolean function of partition `p`. It returns `True` if `cycles_used(p) + cycles_left_in_current_tick <= budget_cycles(p)`

  where:

  `cycles_used(p)`

  The CPU time that the partition has used during the current averaging window.

  `budget_cycles(p)`

  The size of the averaging window, expressed in *ClockCycles()* (not milliseconds - ms) multiplied by the percentage budget of `p`.
- The `MAY_RUN_CRITICAL(p)` function is a boolean function of partition `p`. It returns `True` if:
    - partition `p` is configured with a critical budget that's greater than zero
    - partition `p` has used, during the last averaging window, a critical time that is less than its critical budget minus 1/32 of a tick
    - the highest-priority thread that's ready to run or is currently running in partition `p` is allowed to run as critical.

- The `HIGHEST_PRIO(p)` function is the numerical priority of the highest priority thread that is either running or ready to run in partition `p`.
- If the partition has a nonzero budget, then the relative function free (`RFF(p)`) function is: `1 - used_cycles(p)/budget_cycles(p)`

  If the partition has a zero budget, then `RFF(p)` is defined to be a constant smaller than the smallest possible value of *RFF()* for all other nonzero partitions.

Some operating modes, defined by these boolean expressions, are also defined:

`underload`

When `COMPETING(p) && (HAS_BUDGET(p)||MAY_RUN_CRITICAL(p)) == True` for at least one partition `p`.

`all_at_load`

When `COMPETING(p) == True` for all `p`, and `HAS_BUDGET(p)||MAY_RUN_CRITICAL(p) == False`, for all partitions `p`.

`free_time`

When `COMPETING(p) == False` for at least one partition `p` that has a nonzero budget.

`idle`

when `COMPETING(p) == False`, for all partitions `p`.

The scheduler picks up one of the merit functions, depending on the operating mode:

`underload`

```
merit(p) = (COMPETING(p), HAS_BUD
GET(p)||MAY_RUN_CRITICAL(p), HIGHEST_PRIO(p),
RFF(p) )
```

`all_at_limit`

```
merit(p) = (COMPETING(p), RFF(p))
```

`free_time`, `default`

```
merit(p) = (COMPETING(p), HAS_BUD
GET(p)||MAY_RUN_CRITICAL(p), HIGHEST_PRIO(p),
RFF(p) )
```

`free_time`, `SCHEDPOL_RATIO`

```
merit(p) = (COMPETING(p), HAS_BUD
GET(p)||MAY_RUN_CRITICAL(p), RFF(p) )
```

`idle`

Undefined.

If the mode is idle, the scheduler chooses to run the idle thread in the System partition.

Otherwise, the scheduler chooses to run the highest-priority thread that has a compatible runmask for the CPU on which the scheduler was invoked from the partition `p` such that:

```
merit(p) > merit(p')
```

for all `p'` not equal to `p`.

Merit functions return tuples, and are compared as tuples. For example:

```
(a,b) < (c,d) if (a<c) || ( (a=c) && (b<d) )
```

How does the scheduler find the highest-priority thread in a partition?

It does it very quickly. Each partition has a bitmap that tracks the priority levels (between 0 to 255) that are in use by some ready to run thread in that partition.

Each time the scheduler makes a thread ready to run, it sets the bit corresponding to that thread's priority. When the scheduler runs a thread (its state changes from ready to run), the scheduler examines the queue of threads in that partition that are ready-to-run and at the same priority. If there are no other threads of that priority, the scheduler clears the bit for that thread's priority.

When the scheduler needs to know the highest priority thread that is ready to run in a partition, it uses the bitmap to index a table that maps integers to the number of their highest `1` bit. This is done with a set of tables to avoid the need for 2^{255} table elements.

The same mechanism is also used in classic QNX Neutrino scheduling. The macros are:

- *DISPATCH_SET()*
- *DISPATCH_CLEAR()*
- *DISPATCH_HIGHEST_PRI()*

How are RFFs (relative fraction free) computed?

For the scheduling algorithm, the computation of *RFF()* requires floating-point division. However, QNX Neutrino doesn't perform floating-point operation inside the kernel or even fixed-point division; these operations are very slow on some platforms.

QNX Neutrino computes a function equivalent to *RFF()* that requires only addition and multiplication.

How does the scheduler algorithm avoid division and floating-point mathematics?

For the scheduling algorithm, the computation of *RFF()* requires floating-point division. However, QNX Neutrino doesn't need the absolute values of *RFF()*; it needs to know only the relative ordering of `RFF(p1)`, `RFF(p2)`, ... `RFF(pn)`.

Therefore, QNX Neutrino computes a different function that has the same ordering properties as *RFF()*. This function is computable with only addition and `16×16` bit multiplication.

The idea is:

1. `relative_fraction_free(P)`, or `RFF(P) = 1 - cycles_used/budget_cycles`

   However, instead of finding partition `p`, such that `RFF(p) > RFF(p')` for `p'` not equal `p`, define `relative_fraction_used(p) = RFU(p) = cycles_used/budget_cycles` , and find partition *p* such that `RFU(p) < RFU(p')` for `p'` not equal to `p`.
2. Then find a function that has the same ordering properties as *RFU()*:

   - Find:

     `used_cycles(p0)/budget_cycles(p0) < used_cycles(p1)/budget_cycles(p2)< .... < used_cycles(pn)/budget_cycles(pn)`
   - let

     `k = budget_cycles(p0) * budget_cycles(p1) * ... * budget_cycles(pn)`, then
   - `k/budget_cycles(p0)*used_cycles(p0) < k/budget_cycles(p1)*used_cycles(p1) < ... < k/budget_cycles(pn)*used_cycles(pn)`, as long as all numbers are `>0`.
   - Values of `c(p)=K/budget_cycles(p)`, for all `p`, are computed once, or whenever any partition's percentage budget is changed. The values are stored and aren't recalculated during scheduling
   - At scheduling time, QNX Neutrino computes `f(p) = used_cycles(p) * c(p)`

     and compare `f(p)` to `f(p')` to find which has the better *RFF()*.

However, there are two complications:

Running out of bits

So far, `f(p) = used_cycles(p) * c(p)` requires 64-bit multiplication. However, since the accuracy specification is 0.2%, QNX Neutrino scales all values of `c(p)` by a common factor, until the largest value fits in 16 bits.

QNX Neutrino also shifts `used_cycles(p)` until its largest possible value fits in 16 bits. Therefore, at scheduling time, QNX Neutrino computes only:

```
f(p) = (used_cycles(p)>>scaling_factor) * scaled_c(p)
```

Zero-budget partitions

The above algorithms nominally require QNX Neutrino to multiply and divide everything by zero. However *RFF()* of a zero-budget partition is defined to be a constant that is smaller than any nonzero partition's possible value of *RFF()*. QNX Neutrino defines `RFU(p)` for a zero budget partition as a constant that is greater than *RFU()*. The largest value of *f()* is `window size * c(pm)` where `c(pm) > c(p')` for all `p'` not equal to `pm`.

Therefore, QNX Neutrino can set *f()* for a zero-budget partition as:

```
f_zero = 1 + window size*c(pm)
```

and then scale it as described for running out of bits.

The window size is expressed in cycles.

How does the scheduler algorithm determine if a thread that's allowed to run as critical, should actually run as critical?

See the next answer.

How does the scheduler algorithm decide when to bill critical time?

When the scheduler algorithm picks a thread that is allowed to run as critical to run, it doesn't always charge its CPU time to its partition's critical budget. A thread `t` charges its CPU time to the critical budget of its partition `p` only when the following are true when the scheduler algorithm is invoked:

- thread `t` has the highest priority in the system
- thread `t` is allowed to run as critical now
- partition `p` has a critical budget configured to be greater than zero
- CPU cycles used by all threads in partition `p` during the last averaging window are less than the critical budget of partition `p`
- partition `p` has exhausted its normal CPU budget
- at least one partition, `p'` not equal to `p`, has
  ```
  COMPETING(p') &&(HAS_BUDGET(p')||MAY_RUN_CRITICAL(p')) == True
  ```

For definitions of *COMPETING()*, *HAS_BUDGET()*, and *MAY_RUN_CRITICAL()*, see “How does the scheduling algorithm work?” in Scheduling behaviour.

What are the algorithm's size limitations?

The mathematics of the algorithm are extendable to any number of partitions. However, these are the limitations of the current implementation:

- It has <= 32 partitions, because of use of bit sets and 32-bit integers.
- It has <= 16 partitions, because of an internal step of RFF calculation limited to `16×16` bit multiplication.
- It has <= 8 partitions, a practical limit to prevent the scheduler from consuming too much memory or CPU time.
- You must specify budgets, in integer percentages, e.g. 30% or 31%, but not 30.5%.
- There's no limit on the number of threads per partition.

What are the algorithm's accuracy limitations?

Accuracy refers to the closeness of the scheduler's guarantee or limit that a partition can consume only its budget on a loaded system. For QNX Neutrino, the accuracy is measured based on whichever is greater:

- 0.5%

  Or

- Tick size (in ms) or window size (in ms). For a 100 milliseconds window with a default tick, this is 1%.

When you changes the averaging window size to *x* ms, the accuracy is undefined for the next *x* ms.

The first limitation comes from the accuracy in which the *RFF()* calculation is carried out. The accuracy of *RFF()* is calculated to a limited number of bits, specifically to speed up the scheduling algorithm.

The second limitation comes from the uncertainty in predicting how long a thread runs before it voluntarily blocks, is preempted by a higher-priority thread, or when the next tick interrupt occurs. This limitation comes from the fact that the thread scheduler has guaranteed control of the system only every tick (but may run more often).

In practice, the last limitation implies that when a window size is changed, the scheduler clears its history of used CPU time. So the partition (`p`) with the highest priority thread runs for `budget(p)*window size` (ms) before another partition runs. After the window size (in milliseconds) has elapsed, all budgets are again guaranteed. So a partition, configured for a budget of 40%, with a 100 milliseconds averaging window, is considered to be scheduled accurately when its usage over the last 100 ms was 39 to 41 ms. This happens when the window size hasn't changed in the last 100 milliseconds. In practice, the scheduling accuracy is usually much better.

When is the scheduling algorithm approximated?

In order to save overhead, a very short version of the scheduling algorithm is used on some paths involved in message passing. This short version is implemented with the internal scheduler functions, such as *ready_ppg()*, *block_and_ready_ppg()* and *adjust_priority_ppg()*.

Overhead

Which partition is the overhead associated with scheduling charged to?

Let's consider all kernel calls, such as messaging and mutexting, that switch threads to be overhead. Call the initial running thread `t1`, and the next thread `t2`. Let's consider the kernel calls that are initiated by `t1` and cause `t1` to stop running and `t2` to start running.

The overhead is split between `t1` and `t2`, but mostly to `t1` with the following details:

Time to:	Is charged to the partition of:
Enter the kernel	`t1`
Run the scheduling algorithm	`t1`
Do a context switch	`t2`
Exit the kernel	`t2`

Which partition is the overhead for processing interrupts charged to?

There are two parts of interrupt servicing: the interrupt handler and the interrupt thread.

If you service interrupts with an interrupt thread, most of the time spent servicing the interrupt is the thread's time, and only a small part of the time is spent in the interrupt handler. The interrupt handler determines the thread to which the interrupt event should be delivered.

If you service interrupts with an interrupt handler, all of the time spent servicing the interrupt is in the handler.

The time spent in an interrupt thread is charged against the partition of that thread.

The time spent in an interrupt handler is charged against the partition that's running at that time.

Since the interrupts occur in random, time spent in interrupt handler is spread evenly over all running partitions.

What is the CPU overhead with the thread scheduler?

The QNX Neutrino results indicate that heavy compiling benchmarks that involve a lot of filesystem-related messaging are about 1% slower on x86 platforms when using the thread scheduler.

What is the memory overhead with the thread scheduler?

Data

A few kilobytes of fixed overhead along with 2 KB per partition.

Code

About 18 KB.

Both of these are in the kernel space.

What factors increase the overhead for the thread scheduler?

In approximate order of importance, the cost of the thread scheduler increases with:

- the number of scheduling operations, such as sending messages, events and signals sent, mutex operations, and sleeps
- the platform — in particular, ARM is noticeably slower because of the I/O needed to implement *ClockCycles()*
- the frequency of clock ticks
- the number of partitions
- the use of runmasks

In all the above cases, the increase is approximately linear.

The following factors don't affect the cost of scheduling at all:

- the number of threads
- the length of the averaging window (except for a very small effect when changing the window size)
- the choice of percentage budgets
- the choice of thread priorities
- the choice of FIFO, round-robin, or sporadic thread policies

Critical threads and bankruptcy

How does the scheduler mark a thread as critical?

See the next answer.

How does the thread scheduler know that a thread is critical?

QNX Neutrino maintains a data block, the `thread_entry`, representing the *state* of each thread. It contains three state bits for controlling the critical threads that indicate whether or not the thread is:

- always allowed to run as critical
- allowed to run as critical until it blocks
- currently running as critical (and is consuming its partition's critical budget).

These state bits are turned on as follows:

Always allowed	**When the user calls *SchedCtl()* with the `SCHED_APS_MARK_CRITICAL` command on that thread.**
Until blocked	When the thread receives an event from an interrupt handler, a message from another thread marked either always allowed to run critical, or allow critical until it blocks an event, on which the user has previously called the macro, *SIGEV_MAKE_CRITICAL()*
Currently running as critical	When the scheduler algorithm decides that thread would not have been eligible to run if it hadn't been allowed to run as critical.

Do critical threads expose security?

No.

You can set your own thread to be critical, or receive a critically tagged event or message. This way, the thread gets the property of the allowed to run critical flag. You must configure the partition with a nonzero critical budget to:

- affect the critical budget, and
- cause its partition to run when it's out of budget (thereby taking time from some other partition).

Setting a nonzero critical budget on a partition is controlled. For the recommended scheduler partition security settings, only `root`, running in the parent partition of a target partition, can set a nonzero critical budget.

When does the scheduler check for bankruptcy?

To save time, the thread scheduler only polls partitions for bankruptcy only on each clock tick (rather than every scheduling operation). So typically, bankruptcy is detected one millisecond (or clock period) after a partition's critical budget has been exhausted.

How does the scheduler detect bankruptcy?

QNX Neutrino compares the total critical time (over the last averaging window) to the partition's configured maximum critical time budget. Each partition maintains a separate rotating window for tracking critical time usage. The history window for this critical time identifies, for each millisecond of the last 100 milliseconds, which part of the total CPU time was considered to be critical time.

Inheritance

What is partition inheritance?

Partition inheritance occurs when the scheduler bills the CPU time of a thread not to its own partition, but to the partition of a different thread. This feature makes the thread scheduler *adaptive*.

When does partition inheritance occur?

Partition inheritance occurs under two situations:

- when one thread is working on behalf of another

  When a client thread sends a message to a server thread, that server thread is considered to be working on the client thread's behalf. In this case, QNX Neutrino charges the execution time of the receiving thread, from the time it receives the message and up to the time it waits for the next message, to the partition of the sending thread.

  This means that resource managers, such as filesystems, automatically bill their time to their appropriate clients. This implies that partitions containing only resource managers don't need to be reengineered every time a new client is added to the system.
- when not inheriting would cause excessive delays (in a special case of mutex inheritance)

How does mutex partition and inheritance work?

When threads line up for access to a mutex, QNX Neutrino doesn't consider the thread holding the mutex to be waiting on behalf of the threads waiting for the mutex. So, there is no inheritance of partitions.

However, there is a special scenario when the thread holding the mutex is in a partition that ran out of available budget. In this scenario, the thread can't run and release the mutex. All the threads waiting for that mutex are stalled until enough window rotations have occurred for mutex-holding partitions to regain some available budget. This is particularly nasty if the user has configured that partition to have a zero budget.

So, when a thread `t1` holds a mutex in a partition that has exhausted its budget, and another thread `t2` attempts to seize the mutex, QNX Neutrino puts thread `t2` to sleep until thread `t1` releases the mutex (which is classic mutex handling), and then changes the partition of `p1` to be `p2` until it releases the mutex, provided the budget of partition `p2` is nonzero. This prevents extended delays, should the current mutex holder run out of budget.

How fast is partition inheritance?

Very fast.

The data block that QNX Neutrino keeps for each thread, the `thread_entry`, has a pointer to its containing partition. So inheritance is simply a matter of swapping the pointer. Often, QNX Neutrino doesn't even need to update the microbilling because the same partition is executing before and after the inheritance.

Why is partition inheritance for message passing secure?

Sending a message to a process effectively gives the sender's partition budget to the receiver thread (temporarily). However, to receive threads in that manner, the receiver process must have been started under the root user.

Budgets

Can I change the budgets dynamically?

You can change a partition's budget any time.

How does a budget change affect scheduling?

See the next answer.

How quickly does a budget change take effect?

The operation is quick and doesn't reset the scheduler or cause any change to the partition's history of CPU usage that is stored in the averaging window.

However, if you change the budget of a partition from 90% to 10%, the partition could suddenly become over budget. In this situation, the partition may not run again until enough window rotations have occurred to lower the partition's used cycles below its budget.

When does a change in budgets take effect?

A change in budget takes effect at the next tick interrupt or next scheduling operation i.e. typically, in less than one millisecond.

What is a partition with zero budget?

Threads in a partition with a defined budget of zero runs if all nonzero partitions are sleeping. These threads also run if they inherit the partition of thread that sends a message. Zero-budget partitions are most useful for resource managers with no internal daemon threads. They're also useful for turning off unused partitions.

How does the scheduler guarantee that the sum of all partitions' budgets is 100%?

At startup, QNX Neutrino creates the first partition (the System partition) with a budget of 100%. Thereafter, when a thread running in a partition creates a new partition, the current partition is considered as the parent and the new partition is the child. The budget of the child is always taken from the budget of the parent, and may never reduce the parent's budget below zero. So creating partitions produces a hierarchy of partitions that subdivide the System's original budget of 100%.

How does the scheduler prevent an untrusted thread from increasing its partition's budget?

For any change to occur, the scheduler partition security would have to be:

- unlocked to permit budget changes
- set to permit non-*root* users to modify budgets
- set to permit a partition to modify its own budget (usually only the parent can modify a partition's budget)

Note that a thread in a partition can't increase its budget more than the budget of its parent partition.

How can I cheat to exceed my partition's budget?

You can:

- change the window size often
- provide your partition an infinite critical budget and set yourself to run as critical

  As the root user, unlock the scheduler partition configuration and turn off the scheduler partition security.

In order to do either of these, you must be the *root* user, unlock the scheduler partition configuration and turn off the scheduler partition security.

The following ideas look promising, but:

- Giving your own partition more budget (it can't exceed its parent's, even if security is off).
- Setting your thread priority to 255 (you can starve everything else in your partition, but not another partition).
- Setting your thread policy to FIFO and loop (you can starve everything else in your partition, but not another partition.)
- Creating your own partition (the child partition's budget can't be greater than your own).

Joining a partition

How does joining a thread to a partition work?

See the next answer.

How fast is joining a thread to a partition?

Each `thread_entry` (the control block that QNX Neutrino maintains for each thread) has a pointer to its containing partition. Joining a thread means only changing this pointer. The act of joining is very fast. Most of the time is spent in entering the kernel in order to swap the pointer.

QNX Neutrino system considerations

Why doesn't QNX Neutrino allow a partition to be deleted?

It's safer and much more efficient not to delete a partition. A suggested alternative is to set the partition's budget to zero.

To delete a partition, QNX Neutrino would have to locate all threads (or assert that there are none) in a partition and move them to some other partition.

Threads are mapped to their partitions with a single pointer. There is no back pointer, as it would require a linked list to implement a many-to-one mapping to chain together all threads.

In addition, QNX Neutrino would require additional kernel memory for a two-way queue through all `thread_entries`. In addition, QNX Neutrino also have to do two-way queue extractions every time it (QNX Neutrino) inherited partitions (e.g. message sending) while evading the simultaneous destruction of other threads.

How does the thread scheduler plug into procnto?

See the next answer.

Is the classic scheduler still present when the thread scheduler is active?

Adaptive partitioning scheduler is part of the kernel.

It is shipped as a library module (`libmod`) that is built into the image along with `procnto`. The `procnto` also contains the code for the classic QNX Neutrino scheduler when the thread scheduler module is not present. However, when the thread scheduler module is present, `procnto` initializes the thread scheduler instead of the classic scheduler. The thread scheduler then directs a set of function pointers, one for each primitive scheduling operation (such as *ready()*, *block()*, etc.), to its own function constants. Subsequently, it creates the system partition, which it returns to `procnto`.

Does the thread scheduler inhibit I/O interrupts?

Yes. The thread scheduler calls *InterruptDisable()* for slightly longer than the time required to call *ClockCycles()* each time it must microbill. That includes not inhibiting interrupts to get mutual exclusion between the clock interrupt handler, scheduling algorithm, getting partition statistics, or changing budgets.

SchedCtl(SCHED_APS_PARTITION_STATS,...)

Is there a performance limitation on how often I can call SchedCtl(SCHED_APS_PARTITION_STATS,...) to get statistics?

Other than the cost of the *SchedCtl()* kernel call, the answer is no.

Getting statistics doesn't inhibit interrupts, or delay window rotations or the scheduling algorithm (on other SMP processors.) Consistent retrieval of statistics is accomplished by detecting collisions and having the API withdraw and retry. Note that the call to `SchedCtl(`*`SCHED_APS_PARTITION_STATS API`*`..)` fails with `EINTR` only in the unlikely case of three consecutive collisions. In general, this can occur only if the user has set the clock period to such a short value that it's likely unsafe for the rest of the system.

Appendix C Glossary

averaging window

A sliding window, 100 ms long by default, over which the thread scheduler calculates the CPU percentage usage.

the thread scheduler also keeps track of the usage over longer windows, strictly for reporting purposes. window 2 is typically 10 times the length of the averaging window, and window 3 is typically 100 times the length of the averaging window.

bankruptcy

What happens when critical threads exhaust their partition's critical time budget.

budget

The CPU time, expressed as a fraction of 100%, that a partition is guaranteed to receive when it demands it.

CPU share

Another word for *budget*.

critical budget

A time, in milliseconds, that critical threads are allowed to run even if their partition is out of CPU budget.

critical thread

A thread that's allowed to run, even if its partition is out of CPU budget, provided its partition has a nonzero critical budget.

fair-share schedulers

A class of thread schedulers that consider dynamic processor loads, rather than only fixed thread priorities, in order to guarantee groups of threads some kind of minimum service.

free time

A time period when some partitions aren't demanding their guaranteed CPU percentage.

inheritance

What happens when one thread, usually a message receiver, temporarily adopts the properties of another thread, usually the message sender.

inheritance of partition

What occurs when a message-receiving thread runs in the partition of its message-sender.

microbilling

Calculating the small fraction of a clock tick used by threads that block frequently, and counting this time against the threads' partitions.

partition

A division of CPU time, memory, file resources, or kernel resources with some policy of minimum guaranteed usage.

scheduler partition

A named group of threads with a minimum guaranteed CPU budget.

thread scheduler

Lets you guarantee minimum percentages of the CPU's throughput (using budgets) to groups of threads, processes, or applications.

throttling

Not running threads in one partition, in favor of running threads in another partition, in order to guarantee each their minimum CPU budgets.

underload

The situation when the CPU time that the partitions demand is less than their CPU budgets.

Index