

QNX[®] Neutrino[®] RTOS

Adaptive Partitioning

User's Guide

For QNX[®] Neutrino[®] 6.3.2

© 2005–2007, QNX Software Systems GmbH & Co. KG. All rights reserved.

Published under license by:

QNX Software Systems International Corporation

175 Terence Matthews Crescent

Kanata, Ontario

K2M 1W8

Canada

Voice: +1 613 591-0931

Fax: +1 613 591-3579

Email: info@qnx.com

Web: <http://www.qnx.com/>

Publishing history

February 2006

First edition

Electronic edition published 2007

QNX, Neutrino, Photon, Photon microGUI, Momentics, and Aviage are trademarks, registered in certain jurisdictions, of QNX Software Systems GmbH & Co. KG. and are used under license by QNX Software Systems International Corporation. All other trademarks belong to their respective owners.

Printed in Canada.

Part Number: 002534

About This Guide	vii
What you'll find in this guide	ix
Typographical conventions	ix
Note to Windows users	x
Technical support	x
1 What is Adaptive Partitioning?	1
What are partitions?	3
System requirements	3
Adaptive partitioning scheduler	3
2 A Quick Introduction to the Adaptive Partitioning Scheduler	5
3 Adaptive Partitioning Scheduling Details	11
Introduction	13
Keeping track of CPU time	13
How CPU time is divided between partitions	14
Underload	15
Free time	16
Full Load	17
Summary of scheduling behavior	18
Partition inheritance	18
Critical threads	19
Bankruptcy	21
Adaptive partitioning and other schedulers	22
A caveat about FIFO scheduling	23
Using adaptive partitioning and multicore together	23
Adaptive partitioning and BMP	24
4 System Considerations	27
Determining the number of adaptive partitions and their contents	29
Choosing the percentage CPU for each partition	30

Setting budgets to zero	30	
Setting budgets for resource managers	31	
Choosing the window size	32	
Accuracy	32	
Delays compared to priority scheduling	32	
Practical limits	34	
Uncontrolled interactions between adaptive partitions	34	
Security	35	
Security and critical threads	37	
5	Setting Up and Using the Adaptive Partitioning Scheduler	39
Building an image	41	
Creating partitions	41	
In a buildfile	41	
From the command line	41	
From a program	42	
Launching a process in a partition	42	
In a buildfile	42	
From the command line	43	
From a program	43	
Viewing partition use	43	
6	Testing and Debugging	45
Instrumented kernel trace events	47	
IDE (trace events)	47	
Other methods	47	
Emergency access to the system	47	
A	Sample Buildfile	49
	Glossary	53
	Index	57

List of Figures

The averaging window moves forward as time advances.	14
The adaptive partitioning scheduler's behavior when underloaded.	15
The adaptive partitioning scheduler's behavior under a full load.	17
Server threads temporarily join the partition of the threads they work for.	19

About This Guide

What you'll find in this guide

The Adaptive Partitioning *User's Guide* will help you set up and use adaptive partitioning to divide system resources in a flexible way between competing processes. This guide is intended for software developers of individual applications, as well as for developers who are responsible for the overall time or throughput behavior of the entire system. In general, you need to consider the entire system when you set partition budgets, window sizes, and other parameters.

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?
Getting started with the adaptive partitioning scheduler	A Quick Introduction to the Adaptive Partitioning Scheduler
How the adaptive partitioning scheduler works	Adaptive Partitioning Scheduling Details
Determining how many partitions to have, and what to put in them	System Considerations
Setting up and using adaptive partitioning	Setting Up and Using the Adaptive Partitioning Scheduler
Checking for and fixing any problems	Testing and Debugging
Modifying your buildfile	Sample Buildfile
Terminology used in this guide	Glossary

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	<code>if(stream == NULL)</code>
Command options	<code>-lR</code>
Commands	<code>make</code>
Environment variables	<code>PATH</code>
File and pathnames	<code>/dev/null</code>

continued...

Reference	Example
Function names	<i>exit()</i>
Keyboard chords	Ctrl-Alt-Delete
Keyboard input	something you type
Keyboard keys	Enter
Program output	login:
Programming constants	NULL
Programming data types	unsigned short
Programming literals	0xFF, "message string"
Variable names	<i>stdin</i>
User-interface components	Cancel

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.



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



WARNING: 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 use a forward slash (/) as a delimiter in *all* pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.

Technical support

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

What is Adaptive Partitioning?

In this chapter...

What are partitions?	3
System requirements	3
Adaptive partitioning scheduler	3

What are partitions?

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

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).



The initial version of Adaptive Partitioning implements the budgeting of CPU time, via the *adaptive partitioning scheduler*. Future releases will address budgeting memory and other resources.

Traditional partitions are *static* and work best if 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 realtime scheduler under normal load, but can still provide minimal interrupt latencies when the system is fully loaded
- they distribute a partition's unused budget among partitions that require extra 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.

System requirements

For adaptive partitioning scheduling to operate properly, your system should meet some 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 scheduler still works correctly, but CPU percentages apportioned to partitions will be inaccurate. A simpler reason for preventing SMM is that it introduces interrupt latencies of about 100 microseconds at unpredictable intervals.
- Adaptive partitioning isn't supported on 386 and 486 processors, because they don't have a timebase counter, which the scheduler needs in order to do microbilling.

Adaptive partitioning scheduler

The *adaptive partitioning 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 adaptive partitioning scheduler has been designed on top of the core QNX Neutrino architecture primarily to solve these problems in embedded systems design:

- guaranteeing proper working when the system is fully loaded
- preventing unimportant or untrusted applications from monopolizing the system

Before looking at the details of the scheduler, let's give it a try.

Chapter 2

A Quick Introduction to the Adaptive Partitioning Scheduler

This chapter gives you a quick hands-on introduction to the adaptive partitioning scheduler. It assumes that you're running the Photon microGUI on your Neutrino system.

- 1 Log in as `root`.
- 2 Go to the directory that holds the buildfile for your system's boot image (e.g. `/boot/build`).

- 3 Create a copy of the buildfile:

```
cp qnxbasedma.build apsdma.build
```

- 4 Edit the copy (e.g. `apsdma.build`).

- 5 Search for `procnto`. The line might look like this:

```
PATH=/proc/boot:/bin:/usr/bin:/opt/bin \  
LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib \  
procnto-instr
```



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 command easier to read.

- 6 Add `[module=aps]` to the beginning of the line:

```
[module=aps] PATH=/proc/boot:/bin:/usr/bin:/opt/bin \  
LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib \  
procnto-instr
```

You can add commands to your buildfile to create partitions and start programs in them, but when you're experimenting with adaptive partitioning, it's better to do it at runtime, so that you can easily change things. For more information, see the Setting Up and Using the Adaptive Partitioning Scheduler chapter.

- 7 Save your changes to the buildfile.

- 8 Generate a new boot image:

```
mkifs apsdma.build apsdma.ifs
```

- 9 Put the new image in place. We recommend that you copy your current boot image to `/.altboot` as a backup in case something goes wrong:

```
cp /.altboot /.old_altboot  
cp /.boot /.altboot  
cp apsdma.ifs /.boot
```

- 10 Reboot your system.

11 Log in as a normal user.

You don't have to be `root` to manipulate the partitions because the security options are initially off. If you use adaptive partitioning in a product, you should choose the level of security that's appropriate. For more information, see the System Considerations chapter in this guide, and the documentation for `SchedCtl()` in the Neutrino *Library Reference*.

12 The adaptive partitioning scheduler automatically creates one partition, called `System`. Use the `aps` utility to create another partition:

```
aps create -b20 partitionA
```

Note that the new partition's budget is subtracted from its parent partition's budget (the System partition in this case).

13 Use the `aps` utility to list the partitions on your system:

```
$ aps show -l
```

Partition name	id	CPU Time		Critical Time	
		Budget	Used	Budget	Used
System	0	80%	14.14%	100ms	0.000ms
partitionA	1	20%	0.00%	0ms	0.000ms
Total		100%	14.14%		



The `-l` option makes this command loop until you terminate it (e.g. by pressing Ctrl-C). For this demonstration, leave it running, and start another terminal window to work in.

14 Use the `on` command to start a process in the partition:

```
on -Xaps=partitionA rebound &
```



Because `rebound` is a graphical application, it makes `io-graphics` (which runs in the System partition) use some CPU time. Don't set `rebound` to run at its highest speed, or `io-graphics` will starve the shells that are also running in the System partition.

15 Create another partition and run `rebound` in it:

```
aps create -b20 partitionB
on -Xaps=partitionB rebound &
```

16 To determine which partitions your processes are running in, use the `pidin sched` command. For adaptive partitioning, the `ExtSched` column displays the partition name.**17** Create a program called `greedy.c` that simply loops forever:

```
#include <stdlib.h>

int main( void )
```

```

{
    while (1) {
    }

    return EXIT_SUCCESS;
}

```

- 18** Compile it, and then run it in one of the partitions, where it will use as much CPU as possible:

```

gcc -o greedy greedy.c
on -Xaps=partitionB ./greedy &

```

Note that the instance of **rebound** that's running in **partitionB** no longer gets much (if any) CPU time, but the one in **partitionA** still does, because the adaptive partitioning scheduler guarantees that partition 20% of the CPU time.

- 19** Look at the output from **aps**. It will look something like this:

Partition name	id	CPU Time		Critical Time	
		Budget	Used	Budget	Used
System	0	60%	11.34%	100ms	0.000ms
partitionA	1	20%	2.12%	0ms	0.000ms
partitionB	2	20%	86.50%	0ms	0.000ms
Total		100%	99.96%		

Note that **partitionB** is using more than its budget of 20%. This happens because the other partitions aren't using their budgets; instead of running an idle thread in the other partitions, the adaptive partitioning scheduler gives their unused time to the partitions that need it.

- 20** Start another instance of **greedy** in **partitionA**:

```

on -Xaps=partitionA ./greedy &

```

The instance of **rebound** in that partition grinds to a halt. The output of **aps** looks something like this:

Partition name	id	CPU Time		Critical Time	
		Budget	Used	Budget	Used
System	0	60%	1.73%	100ms	0.000ms
partitionA	1	20%	48.91%	0ms	0.000ms
partitionB	2	20%	49.32%	0ms	0.000ms
Total		100%	99.96%		

The System partition's unused time is divided between the other two partitions.

- 21** Start another instance of **greedy** in the System partition:

```

on -Xaps=System ./greedy &

```

There's now no free time left in the system, so each partition gets only its minimum guaranteed CPU time. The output of **aps** looks something like this:

Partition name	id	CPU Time		Critical Time	
		Budget	Used	Budget	Used

System	0		60%		59.99%		100ms		0.000ms
partitionA	1		20%		19.97%		0ms		0.000ms
partitionB	2		20%		19.99%		0ms		0.000ms

Total			100%		99.96%				

22 If you **slay** the instances of **greedy**, the instances of **rebound** come back to life, and the consumption of CPU time drops.

Because you created the partitions at runtime instead of in your OS image, the new partitions will disappear when you restart the system.

Adaptive Partitioning Scheduling Details

In this chapter...

Introduction	13
Keeping track of CPU time	13
How CPU time is divided between partitions	14
Partition inheritance	18
Critical threads	19
Adaptive partitioning and other schedulers	22
Using adaptive partitioning and multicore together	23

Introduction

The adaptive partitioning 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 adaptive partitioning scheduler has been designed on top of the core QNX Neutrino architecture to solve primarily two problems in embedded systems design:

- proper working in fully loaded conditions
- preventing unimportant or untrusted applications from monopolizing the system

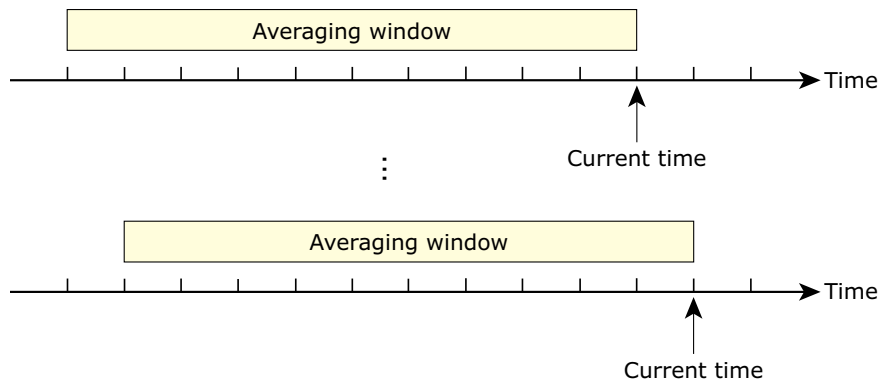
We call our partitions “adaptive” because their contents are dynamic:

- 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 Neutrino send-receive-reply messaging, message receivers automatically run in the partition of the message sender while they're processing that message. That 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 scheduler throttles CPU usage by measuring the average CPU usage of each partition. The average is computed over an averaging window, normally 100 ms, a value that is configurable.

The adaptive partitioning scheduler, however, doesn't wait 100 ms to compute the average. In fact it calculates it very often. As soon as 1 ms passes, the usage for this 1 ms is added to the usage for previous 99 ms to compute the total CPU usage over the averaging window (i.e. 100 ms).



The averaging window moves forward as time advances.

The window size defines the averaging time over which the adaptive partitioning scheduler tries to balance the partitions to their guaranteed CPU limits. You can set the averaging window size to any value from 8 ms to 400 ms.

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 System Considerations chapter.

Because the averaging window slides, it can be difficult for you to keep statistics over a longer period, so the scheduler keeps 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 windows, use the `show -v` or `show -vv` option to the `aps` command.

The 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 call this *microbilling*.

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

How CPU time is divided between partitions

The adaptive partitioning scheduler is a fair-share scheduler. That means it guarantees that partitions receive a defined minimum amount of CPU time, their *budget*, whenever they demand it. The adaptive partitioning scheduler is also a realtime scheduler. That means it's a preemptive, priority-based scheduler. These two requirements appear to conflict.

The scheduler satisfies both requirements by scheduling by priority at all times that it doesn't need to limit a partition in order to guarantee some other partition its budget. Here are the details.

Underload

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

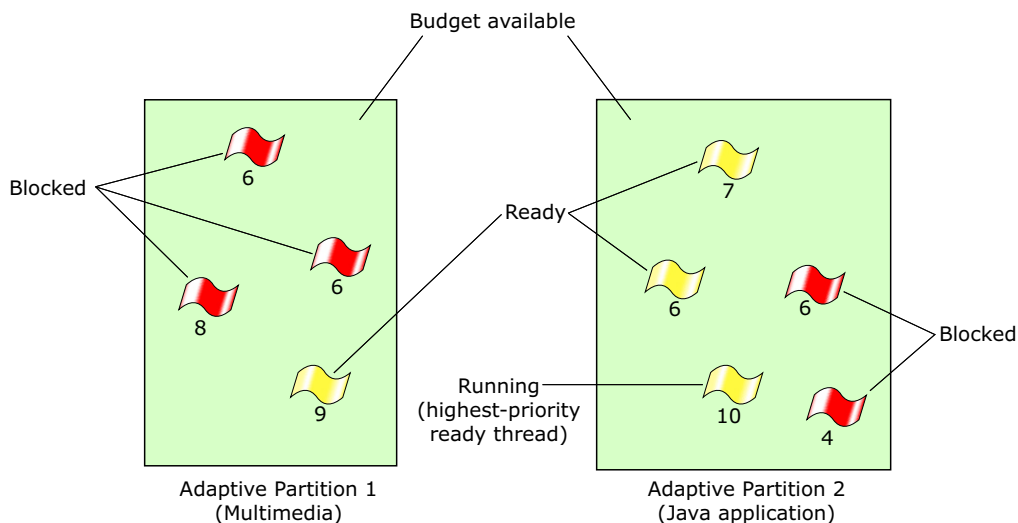
- System partition, with a 70% budget
- partition **Pa**, with a 20% budget
- partition **Pb**, with 10% budget

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

Partition name	id	+---- CPU Time ----+		--- Critical Time ---	
		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 are demanding less than their budgets.

Whenever there are partitions demanding less than their budgets, the scheduler chooses between them by picking the highest-priority running thread. In other words, when underloaded, the adaptive partitioning scheduler is a strict realtime scheduler. This is simply normal Neutrino scheduling.



The adaptive partitioning scheduler's behavior when underloaded.

Free time

Free time occurs when one partition isn't running. The 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 is demanding no time, the scheduler hands out the remaining time to the 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:

Partition name	id	CPU Time		Critical Time	
		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** is getting its guaranteed minimum of 20%, but all the free time is given to partition **Pb**.

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

However, there may be circumstances when you don't want partition **Pb** to get all the free time just because it has the highest-priority thread. That may occur when, say, you choose to use **Pb** to encapsulate an untrusted or third-party application, especially if you can't control its code.

In that case, you may wish to configure the scheduler to run a more restrictive algorithm that divides free time by the budgets of the busy partitions (rather than giving it all to the highest-priority thread). To do so, set the `SCHED_APS_FREETIME_BY_RATIO` flag with `SchedCtl()` (see the *Neutrino 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:

Partition name	id	CPU Time		Critical Time	
		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%		

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 scheduler divides free time between partitions Pa and Pb in roughly a 2:1 ratio, which is the ratio of their budgets.

Full Load

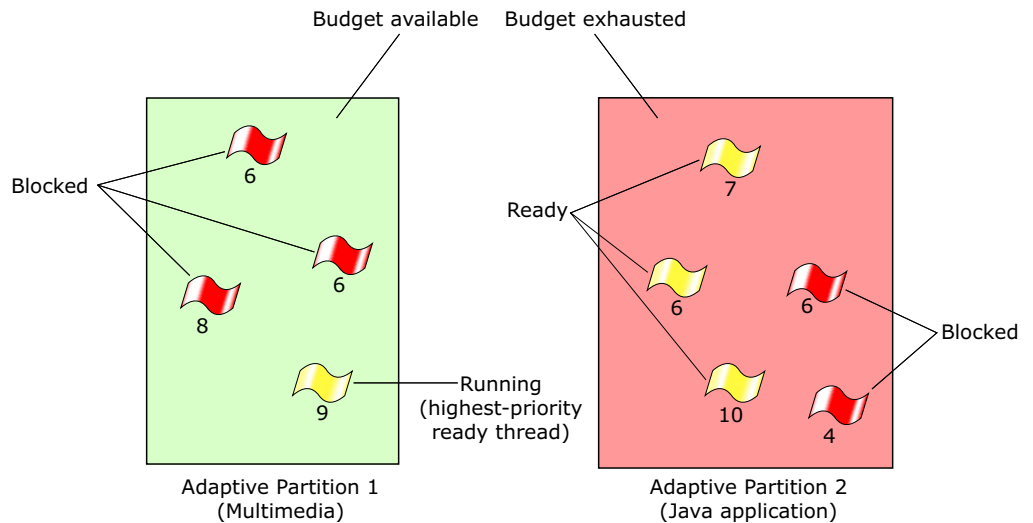
Full load occurs when all partitions are demanding their full budget. A simple way to demonstrate this is to run `while (1)` loops in all of the sample partitions. In this case, the `aps show` command might display:

Partition name	id	+---- CPU Time ----+		----- Critical Time --	
		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 case, the requirement to meet the partitions' guaranteed budgets takes precedence over priority.

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

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



The adaptive partitioning scheduler's behavior under a full load.

Summary of scheduling behavior

The following table summarizes how the 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 adaptive partitioning 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 Neutrino send-receive-reply messaging scheme receives a message from a client, Neutrino considers the server thread to be working on behalf of the the client. So 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 adaptive partitioning 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 adaptive partition.

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

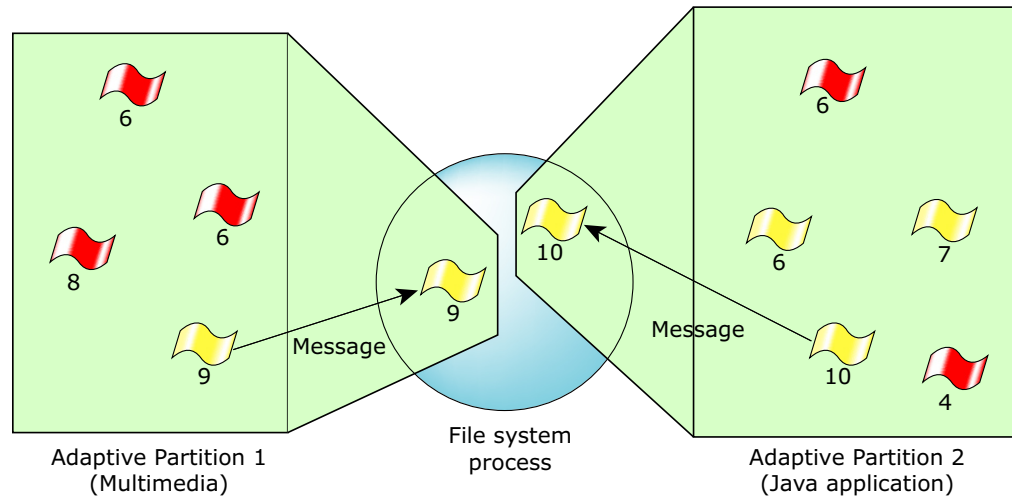
New threads If you receive a message from another partition, and you create a new thread in response, the child thread runs in the sender's partition until the child thread becomes receive-blocked. At that point, the child thread's partition is reset to be its creator's partition.

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 *Neutrino Library Reference*.

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



Server threads temporarily join the partition of the threads they work for.

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, 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. See "Security" in the System Considerations 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 behind this is that a critical thread is allowed to violate the budget rules of

its partition and run immediately, thereby getting the realtime response it needs. 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 whatever thread receives your event as being critical:

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

before you send the 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 ever 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 adaptive partitions as critical and give 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 adaptive partition is out of budget, as long as its partition still has critical budget.

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 be at least one other partition that is 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," below).

In order to be useful, the number of critical threads in the system must be few. If they are the majority, the adaptive partitioning scheduler will rarely be able to guarantee all partitions' their minimum CPU budgets. In other words, the system degrades to a priority-based scheduler when there are too many critical threads.

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



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 adaptive partitioning scheduling. If you don't want this behavior, 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 "Security" in the System Considerations 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 full load to make sure that everything works correctly, especially to ensure that you've chosen the correct critical budgets. An easy way to do this is to start a **while (1)** thread in each partition to use up 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. Neutrino lets you set a recovery policy. The options are:

Default	Do the minimum. When a partition runs out of critical budget, it's not allowed to run again until it gets 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. That means that, at the very least, 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 adaptive partitioning scheduler will deliver that **sigevent** when the corresponding partition becomes bankrupt. To prevent a possible flood of **sigevents**, the adaptive partitioning scheduler will deliver at most one **sigevent** per registration. If you want another notification, use the *SchedCtl()* again and reattach the event to get the next notification. This way 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 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 *Neutrino Library Reference*).

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



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 whole system to stabilize after such an event, the scheduler gives *all* partitions a two-window grace period against declaring bankruptcy when one partition gets its budget canceled.

Adaptive partitioning and other schedulers

Priorities and scheduler policies are relative to one adaptive partition only; priority order is respected within a partition, but not between partitions if the adaptive partitioning scheduler needs to balance budgets.

You can use the adaptive partitioning scheduler with the existing FIFO, round-robin and sporadic scheduling policies. The scheduler, however, may:

- stop a thread 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 scheduler doesn't wait for the end of a thread's timeslice to decide to run a thread from a different partition. The scheduler takes that decision every tick (where a tick is 1 ms on most machines). There are 4 ticks per timeslice.

Round-robin threads that are 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 trick for getting mutual exclusion between a set of threads that are 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 trick into different partitions (or let them get 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 should not 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, you should make sure your applications don't depend on FIFO scheduling or the length of the timeslice for mutual exclusion.

Using adaptive partitioning and multicore together

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

Note the following points:

- On an SMP machine, the adaptive partitioning scheduler considers the time to be 100%, not (say) 400% for a four-processor machine
- The adaptive partitioning scheduler first tries to keep every processor busy; only then does it apply budgets. For example, if you have a four-processor machine, and partitions divided 70%, 10%, 10%, and 10%, but there is only one thread running in each partition, the adaptive partitioning scheduler runs all four threads all the time. The 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 many-threaded 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 are not 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 scheduler correctly guarantees each partition's minimum budget.



On a hyperthreaded machine, actual throughput of partitions may not match the percentage CPU time usage reported by the adaptive partitioning scheduler. This is because on a hyperthreaded machine, throughput isn't always proportional to time, regardless of what kind of scheduler is being used. This 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.

Adaptive partitioning 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%

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 scheduler interprets this to mean that you intend `Pa` to monopolize CPU 1. That's because CPU 1 can provide only 50% of the whole machine's processing time.

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

The 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 scheduler observes that both partitions have available budget. When partitions have available budget, the scheduler schedules in realtime mode, which is strict priority scheduling. So partition **Pa** runs. But because CPU 1 can never satisfy the budget of partition **Pa**, **Pa** never runs out of budget. Therefore the 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 is getting 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.

To simplify the usage of runmasks with the adaptive partitioning scheduler, you may configure the scheduler to follow a more restrictive algorithm that prefers to meet budgets in some circumstances rather than schedule by priority.

To do so, set the flag `SCHED_APS_SCHEDPOL_BMP_SAFETY` with the `SchedCtl()` function that's declared in `<sys/aps_sched.h>` (see the *Neutrino Library Reference*), or use the `aps modify -s bmp_safety` command (see the entry for `aps` in the *Utilities Reference*).

The following table shows how time is divided in normal mode (with its risk of monopolization) and BMP-safety mode on a 2-CPU machine:

Partition state	Normal	BMP-safety
Usage < budget / 2	By priority	By priority
Usage < budget	By priority	By ratio of budgets
Usage > budget, but there's free time	By priority ^a	By ratio of budgets
Full load	By ratio of budgets	By ratio of budgets

^a When `SCHED_APS_SCHEDPOL_FREETIME_BY_PRIORITY` isn't set. For more information, see the `SCHED_APS_SET_PARMS` command in the entry for `SchedCtl()` in the *Neutrino Library Reference*.

In the example above, but with BMP-safety turned on, not only does the scheduler run both the System partition and partition **Pa**, it fairly divides time on CPU 1 by the ratio of the partitions' budgets. The **aps show** command displays usage something like this:

```
+----- CPU Time -----+--- Critical Time ---
Partition name  id | Budget |   Used | Budget |   Used
-----+-----+-----+-----+-----+-----
System         0 |   50% | 25.03% | 200ms | 0.000ms
Pa             1 |   50% | 24.99% |   0ms | 0.000ms
-----+-----+-----+-----+-----+-----
Total          |  100% | 50.02% |
```

The BMP-safety mode provides an easier-to-analyze scheduling mode at the cost of reducing the circumstances when the scheduler will schedule strictly by priority.

In this chapter...

Determining the number of adaptive partitions and their contents	29
Choosing the percentage CPU for each partition	30
Choosing the window size	32
Practical limits	34
Uncontrolled interactions between adaptive partitions	34
Security	35

You typically use the adaptive partitioning 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 adaptive partitioning scheduler with the whole system in mind. The basic decisions are:

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

Determining the number of adaptive partitions and their contents

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

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 adaptive 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 still 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 CPU for each partition

The amount of CPU time that each adaptive 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 CPU consumed.

In general, the key to getting the right combination of partition budgets is to try them:

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

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 zero-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?

- 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 cause free time (see “Summary of scheduling behavior” in the

Adaptive Partitioning Scheduling Details chapter of this guide). A partition with a nonzero budget, that never runs threads, will put the scheduler permanently in free-time mode, which may not be the desired behavior.

- When you want noncritical code to run only when some other partition is idle.
- When the partition is populated by resource managers, or other software, that runs only in response to receiving messages. Because 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.

But in general, setting a partition's budget to zero is risky. (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 hence not run in the sender's partition. So, when the system is loaded (i.e. there's no free time), those threads may simply not run; they might hang, or things might happen 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.



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, sometimes device drivers find out too late which client a particular thread has been doing work for. 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 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.
-

Choosing the window size

You can set the size of the time-averaging window to be from 8 to 400 ms. This is the time over which the scheduler tries to balance adaptive 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 $\pm(\text{tick_size} / \text{window_size})$. For example, if the window size is 10 ms, the accuracy is about 10 percentage points.
- If a partition opportunistically goes over budget (because other partitions are using less than their guaranteed 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).

A small window size means that an adaptive 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 $1 \text{ tick} / \text{window_size_in_ticks}$ 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 scheduler doesn't delay ready-to-run threads, but the highest-priority thread might not run if the adaptive partitioning scheduler is balancing budgets.

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

Case 1

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

$$\text{window_size} - \text{smallest_budget} + \text{largest_budget}$$

This upper bound is only ever reached when low-budget and low-priority adaptive partitions interact with two other adaptive 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, let's suppose we have these adaptive 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 if the following happens:

- Let B and C sleep for a long time. A will run opportunistically and eventually run for 100 ms (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 T_a , since it's the last time partition A ran. Since C is still sleeping, B runs opportunistically.
- At $T_a + 90$ ms, 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 ms, 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 ms, which is when it runs out of budget.
- Only now, at $T_a + 90$ ms + 80 ms, or 170 ms later, does A get to run again.

Note this scenario can't happen 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 adaptive partitions with, on average, priorities equal to other partitions.

However, with a typical mix of thread priorities, each adaptive partition typically experiences a maximum delay, when ready to run, of much less than $window_size$ milliseconds.

For example, let's suppose we have these adaptive 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 ms, it sleeps for 50 ms.

This delay happens if the following happens:

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

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 adaptive partitioning scheduler's window size. That's because Neutrino converts the window size from milliseconds to clock ticks for internal use.

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

Practical limits

The API allows a window size as short as 8 ms. However 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 ms.

Uncontrolled interactions between adaptive partitions

There are cases where an adaptive 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 adaptive partition. This means that faulty hardware or software that causes too many interrupts can effectively limit the time available to other applications.

However, time spent in interrupt threads (e.g. those that use `InterruptAttachEvent()`) is correctly charged to those threads' adaptive partitions.

Security

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 modify` command to specify the level of security that suits your system.

Here are the main security options, in increasing order of security. This list shows the `aps` command and the corresponding `SchedCtl()` flag:

none or the `APS_SCHED_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 may change 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. But `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 budgets must not be zero.

recommended or the `SCHED_APS_SEC_RECOMMENDED` flag

Only `root` from the System partition may 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, may join its own thread to partitions. The percentage budgets must not be zero.

Unless you're testing the partitioning and want to change all parameters without needing to restart, you should set at least **basic** security.

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

```

sched_aps_security_parms p;

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

```



Before you call `SchedCtl()`, make sure 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 to.

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 hole. That's because a thread marked as critical has no effect on the scheduler unless the thread is in a partition that has a critical budget. The adaptive partitioning 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:

- give 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 5

Setting Up and Using the Adaptive Partitioning Scheduler

In this chapter...

Building an image	41
Creating partitions	41
Launching a process in a partition	42
Viewing partition use	43

Building an image

In order to use the adaptive partitioning 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 an example, see the A Quick Introduction to the Adaptive Partitioning Scheduler chapter in this guide; for details, see *Building Embedded Systems*.



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

Creating 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.

In 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
```

From 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.

From 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 up 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 already running threads into another partition.

In 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 into 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.

From the command line

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

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

launches a shell into the `DebugReserve` partition.

From 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. In order to list partitions and the CPU time they're consuming, use the `aps show` command:

```
$ aps show
```

Partition name	id	CPU Time		Critical Time	
		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
```

Partition name	id	Budget	CPU Time			Budget	Critical Time
			Used	10.0s	100.0s		
			0.100s	1.00s	10.0s		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**, **aps** also shows you the critical budget usage over the longer windows.

If you want to display the output of **aps show** every 5 seconds, use the **aps show -1** command. You can use the **-d** option to change the length of the delay.

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

In this chapter...

Instrumented kernel trace events	47
IDE (trace events)	47
Other methods	47
Emergency access to the system	47

Instrumented kernel trace events

The instrumented kernel emits trace events when:

- a partition's budget changes (including when the partition is created)
- a 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 partition becomes bankrupt

In addition, all events include the 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.

IDE (trace events)

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

Other methods

The simplest way to test a system that uses adaptive partitioning 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 “hog” program that loops forever, and start it in each partition. 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 *Neutrino Library Reference*.
- Make sure that all latencies are acceptable for your system.
- Use the `aps modify` 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-net` 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-net` and `inetd` in it. If something goes wrong, you can `telnet` into the system.

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

Appendix A

Sample Buildfile

The following buildfile shows you how to add the adaptive partitioning module to `procnto`. It also includes commented-out commands to 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.

```
#
# The build file for QNX Neutrino booting on a PC
#
[virtual=x86,bios +compress] boot = {
    # Reserve 64k of video memory to handle multiple video cards
    startup-bios -s64k

    # PATH is the *safe* path for executables (confstr(_CS_PATH...))
    # LD_LIBRARY_PATH is the *safe* path for libraries
    # (confstr(_CS_LIBPATH))
    # i.e. This is 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:/opt/bin \
LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll:/opt/lib \
procnto-instr
}

[+script] startup-script = {
    # To save memory make everyone use the libc in the boot image!
    # For speed (less symbolic lookups) we point to libc.so.2 instead
    # of libc.so
    procmgr_symlink ../../proc/boot/libc.so.2 /usr/lib/ldqnx.so.2

    # Default user programs to priority 10, other scheduler (pri=10o)
    # Tell "diskboot" this is a hard disk boot (-b1)
    # Tell "diskboot" to use DMA on IDE drives (-D1)
    # Start 4 text consoles by passing "-n4" to "devc-con" (-o)
    # By adding "-e" linux ext2 filesystem will be mounted as well.
    [pri=10o] PATH=/proc/boot diskboot -b1 -D1 -odevc-con,-n4

    # Create an adaptive partition named "Debugging" with a budget
    # of 20%:
    #sched_aps Debugging 20

    # Start qconn in the Debugging partition:
    #[sched_aps=Debugging]/usr/sbin/qconn

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

# 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.1) add it before the this line.
libc.so

# Include all the files for the default filesystems
libcam.so
io-blk.so
cam-disk.so
```

```
fs-qnx4.so
fs-dos.so
fs-ext2.so
cam-cdrom.so
fs-cd.so

# These programs only need to be run once from the boot image.
# "data=uip" will waste less memory as the ram from the boot
# image will be used directly without making a copy of the data
# (i.e. as the default "data=cpy" does). When they have been
# run once, they will be unlinked from /proc/boot.
[data=copy]
seedres
pci-bios
devb-eide
devb-amd
devb-aha2
devb-aha4
devb-aha7
devb-aha8
devb-adpu320
devb-ncr8
diskboot
slogger
fesh
devc-con
```

A sample buildfile is installed on your system as
`/boot/build/qnxbasedmaaps.build`.

Glossary

averaging window

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

The 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 happens 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.

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 are demanding is less than their CPU budgets.

!

`_NTO_CHF_FIXED_PRIORITY` 19, 23
`[module=aps]` 7, 41, 51
`[sched_aps=...]` 42

A

adaptive partitioning
 debugging with 47
adaptive partitioning scheduler
 about 3
 quick tour 7
adaptive partitions 3
aps
 testing with 47
`APS_INIT_DATA()` 35
`APS_SCHED_SEC_OFF` 35
averaging window 13
 size
 choosing 32
 clock ticks, converting to 34
 limits 34

B

bankruptcy 21
 testing for 47
 trace events 47
borrowed time, repaying 16, 32
bound multiprocessing (BMP) 24
budgets

changing 47
 effect on bankruptcy 22
 trace events 47
defined 3, 13
determining 30
in trace events 47
resource managers 31
setting 41
zero
 preventing 35, 36
 setting 30
 when to avoid 31
buildfiles, modifying for adaptive partitioning
 7, 41, 51

C

`ChannelCreate()` 19, 23
conventions
 typographical ix
CPU usage
 accuracy in controlling 32
 budgeting 3
 interrupt handlers, billing for 34
critical threads 19
 security 37
 trace events 47

D

debugging, using adaptive partitions for 47
delays, avoiding 32

F

FIFO scheduling 22
 file space, budgeting (not implemented) 3
 free time, sharing 9, 16
 full-load situations 17

H

hyperthreaded systems 24

I

IDE
 latencies, checking 47
 trace events 47
 instrumented kernel 47
 Integrated Development Environment *See* IDE
 interrupt handlers 34
 automatically marked as critical 21
InterruptAttachEvent() 34
io-net 48

K

kernel
 trace events 47

L

latencies
 checking for acceptable 30, 47
 critical threads 19
 interrupt 3
 System Management Mode (SMM) 3

M

MAP_LAZY 31

memory, budgeting (not implemented) 3
 microbilling 14
module=aps 7, 41, 51
 multicore systems 23

O

on 8, 43
 OS images, using adaptive partitioning in 7, 41,
 51

P

partitions
 adaptive 3
 budgets
 defined 13
 zero, preventing 35, 36
 zero, setting to 30
 zero, when to avoid 31
 contents of, determining 29
 creating 41
 defined 3
 free time, sharing 9, 16
 IDs
 in trace events 47
 inheritance 18
 preventing 19
 interactions between 34
 locking 36
 number of, determining 29
 processes
 displaying current 8, 43
 starting in 8, 42
 security, setting 8
 static 3
 System 41
 threads
 joining 36
 pathname delimiter in QNX Momentics
 documentation x
pidin 8
 processes

- partitions
 - arranging into 29
 - displaying current 8, 43
 - starting in 8, 42
 - procnto**
 - budget of zero and 31
 - loading adaptive partitioning module 41
 - preventing interrupt handlers from being marked as critical 21
 - pulses, partition processed in 19
- Q**
- qconn** 48
- R**
- realtime behavior with adaptive partitioning 15, 17, 19
 - resource managers
 - budgets, setting 31
 - round-robin scheduling 22
- S**
- SCHED_APS_ADD_SECURITY 30
 - SCHED_APS_BNKR_* 47
 - sched_aps_create_parms** 42
 - SCHED_APS_CREATE_PARTITION 42, 43
 - sched_aps_join_parms** 43
 - SCHED_APS_JOIN_PARTITION 19
 - SCHED_APS_SCHEDPOL_BMP_SAFETY 25
 - SCHED_APS_SCHEDPOL_FREETIME_BY_PRIORITY 25
 - SCHED_APS_SEC_BASIC 35
 - SCHED_APS_SEC_FLEXIBLE 35
 - SCHED_APS_SEC_JOIN_SELF_ONLY 36
 - SCHED_APS_SEC_NONZERO_BUDGETS 30, 36
 - SCHED_APS_SEC_PARENT_JOINS 36
 - SCHED_APS_SEC_PARENT_MODIFIES 36
 - SCHED_APS_SEC_PARTITIONS_LOCKED 36
 - SCHED_APS_SEC_RECOMMENDED 31, 35
 - SCHED_APS_SEC_ROOT_JOINS 36
 - SCHED_APS_SEC_ROOT_MAKES_CRITICAL 36
 - SCHED_APS_SEC_ROOT_MAKES_PARTITIONS 36
 - SCHED_APS_SEC_ROOT0_OVERALL 36
 - SCHED_APS_SEC_SYS_JOINS 36
 - SCHED_APS_SEC_SYS_MAKES_CRITICAL 36
 - SCHED_APS_SEC_SYS_MAKES_PARTITIONS 36
 - SchedCtl()*
 - data structures, initializing 35
 - SchedCtl()*, *SchedCtl_r()* 42, 43, 47
 - scheduling policies 22
 - security, setting 8, 35
 - SIGEV_CLEAR_CRITICAL()* 20
 - SIGEV_GET_TYPE()* 20
 - SIGEV_MAKE_CRITICAL()* 20
 - sigevent**, marking a thread a critical or noncritical 20
 - sporadic scheduling 22
 - static partitions 3
 - symmetric multiprocessing (SMP) 23
 - system
 - emergency access to 47
 - requirements 3
 - System Management Mode (SMM), turning off 3
 - System partition 41
 - infinite critical budget 21
- T**
- telnet** 48
 - threads
 - critical 19
 - security 37
 - trace events 47
 - partitions
 - arranging into 29
 - joining 36
 - tick size, changing 34
 - trace events

IDE, viewing with 47
traceprinter 47
typographical conventions ix

U

underload situations 15

W

window, averaging 13
size
 choosing 32
 clock ticks, converting to 34
 limits 34

Z

zero-budget partitions
 preventing 35, 36
 setting 30
 when to avoid 31