Adaptive Partitioning: the How it Works FAQ

Table of contents

About this FAQ

This faq documents the implemtation of APS as of June 2006.

The audience for this faq is kernel developers interested in the implentation details. It's also been of some use to users who where interested in security issues.

Scheduling Behavior#

How does AP guarantee a partition's minimum cpu budget?#

The adaptive partitioning scheduler guarantees a minimum CPU budget by ensuring that other partitions do not overrun their budget. This determination is made every clock tick.

The AP scheduler is invoked by the clock interrupt handler. That means it runs a minimum of every clock period (typically every millisecond). On each clock tick:

- On a uni-processor, it examines the partition of the currently running thread to see if it should keep running. The AP scheduler will decide if 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".

On exit from the Neutrino clock interrupt handler, the handler examines the flag. If set, it causes the system to immediately enter the kernel and invoke the full AP scheduling algorithm. The AP scheduling algorithm will examine all partitions, and it will stop running the current partition if it is about to go out of budget (no longer has enough to pay for the rest of the current tick) and some other partition has budget. In other words, the AP scheduler guarantees that budgets are met by forcing a partititon to temporarily stop running if it will run over it's budget before the next time the AP scheduler is in control of the system.

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

The AP scheduler will make sure a partition gets at least it's budget in the current averaging window when:

- When the partition is ready-to-run often enough to consume at least at least it's budget's worth of time.
- On SMP machines:
  - let B(p) be the budget, in percent of partition p.
  - let R(p) = "number of ready to run threads in our partition", and
  - let N = "number of CPUs"
  - then the scheduler will guarantee partition p gets B(p) percent cpu over the last averaging window if R(p) >= N * B(p)/100
  - In other words, when the partition has enough ready-to-run threads to occupy the processors in the system.
- No partitition has been billed any critical time.

In other words, budgets are guaranteed, if the system is busy enough, and if no one has used their overdraft protection (critical budget)

Does a 100ms window mean that cpu time averaging occurs only once every 100ms?#

See next answer.

How often does the algorithm enforce partition budgets?#

A 100ms averaging window does not produce information about cpu usage only once every 100ms. 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, every clock tick. So the window provides precise information about the average cpu consumption every millisecond (or clock period).

Between clock ticks, when the AP scheduling 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, AP scheduling computes cpu time used, and enforces budgets, many times per millisecond.

What system assumptions does the design of the AP scheduler make?#

To be able to guarantee partitions get their guaranteed minimum cpu budgets, we assume:

- The clock interrupt handler runs periodically. In other words, the users do not inhibit clock interrupts.
- ClockCycles() is monotonic, except for 64 bit wraparound.
- ClockCycles() increases at a constant rate.
- Useful work done by the processor is proportional to ClockCycles()
- On an SMP machine, each processor sees ClockCycles() incrementing at the same rate (though there may be a constant offset between each processor's ClockCycles()).
- On SMP machines, each CPU does work at the same rate.
- The resolution of ClockCycles() is at least 1/200th of the clock period between timer ticks.
- The user is not changing the size of the averaging window often.

When does the ap scheduler calculate percentage cpu usage?#

Actually, never. It avoids doing division in order to execute quickly. It only ever compares the cpu usage of the partition, over the last averaging window, with that partition's budget, expressed as a total time over the last window rather than a percentage. To make comparisons quick, both usage and budgets are treated internally counts of ClockCycles(), not percentages.

How often does the AP scheduler compute cpu usage?#

At least once very clock period (typically every millisecond). However, it also does it on kernel calls, like message and pulse sending or mutex releases. The scheduler could easily be called to examine the budgets of partitions 50 times per millisecond on a 700mhz x86.

When is the scheduler's behavior realtime?

Within a single partition, the AP scheduler always follows POSIX scheduling rules: 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 is, when does a partition get continuous real time? Since our definition of realtime is "schedule strictly by priority", the answer is:

- The AP scheduler schedules strictly by priority whenever a set of partitions has used less than their budgets over the last averaging window. That means that all threads in will 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, provided you're using less than your budget.

What is free time mode?#

See next answer.

What is free time?#

Free time mode is when there is at is least one partition, with a non-zero budget, that is not using up all of it's budget. Free time mode means that other partitions may use up the free time even going over their own budgets. This is one of the reasons why AP is "adaptive". The extra time a partition gets in free time mode is called "free time", but it's not always free. Sometimes it must be paid back.

Do you have to re-pay "free time"?

Partly. In general, only the free time during the last averaging window need be paid back. For example:

Let partition Pf be one that has exhausted its budget. Let Pa be a partition with available budget. So Pa is running. Let Pa become idle, (goto sleep), for 10 milliseconds. Because it has no competition, i.e. free time mode, Pf will begin running and will run over its budget by 10 milliseconds. Then Pa wakes up. Because Pa has budget, and Pf is over budget, Pa runs. Pf will not run again until the averaging window rotates enough to

carry the history of it's cpu over-usage past 100ms into the past. So Pf might not run until windowsize - budget milliseconds passes. This interval, where Pf, is suspended is effectively paying back the free time.

In general, free time less than windowsize - budget must be paid back.

In a different example, suppose Pa goes to sleep for a minute. Pf will run opportunistically, and consume 100% of the cpu. When Pa wakes up, it will have budget, and Pf will over budget, so Pa will run. Pf will not run again until window rotation move history of it's cpu usage past 100ms in the past. So in this case, Pf need pay back only windowsize - budget milliseconds of the minute of cpu time that ran because Pa was asleep.

An exception is free time that occurred just before a call to SchedCtl(SCHED_APS_SET_PARMS,...) to change the windowsize. Changing the windowsize wipes the scheduler's memory so free time just before a windowsize change is not paid back. So we recommend that users change their windowsize infrequently.

How does AP behave on HyperThreaded Processors?

AP treats a two-headed HT processor as two cpus in SMP. It assumes that each virtual processors has equal and constant throughput. This is true on true SMP machines, but is true on HT machines only when the system is sufficiently loaded to keep both pseudo-CPUs busy. This is a consequence of AP requiring a system's throughput to be proportional to ClockCycles()

How long can a Round-Robin thread run with AP scheduling?

First non-AP scheduling (i.e. classic Neutrino scheduling): A FIFO thread:

- May be preempted any time by a higher priority thread.
- If not preempted, and if there is no other thread at the same priority, a Round-Robin thread will run until it voluntarily gives up control, or forever.
- If un-preempted, but if there is another thread at equal priority, a FIFO thread will run for 4 ticks (nominally 4ms) before being timesliced with the other thread.

With APs scheduling, a Round-Robin thread:

- May be preempted any time by a higher priority thread in the same partition.
- if not preempted, and if there is no other thread of the same priority in that partition, a Round-Robin thread will run until it gives up control or it's partition runs out of budget.
- A ready-to-run Round-Robin thread, in a partition that is out of budget, may start running if it's partition gets more budget on the next clock tick.(As the rotation of the window gives that partition available budget back.)
- If not preempted, and if there is another thread of equal priority, a Round-Robin thread will run for 4 ticks (nominally 4ms), before being timesliced with the other thread, if it's partition has at least 4 milliseconds of available budget.

In other words, AP's actions to guaranteeing budgets overrides a Round-Robin's thread's timeslice. When a partition has more than 4 ticks of available time left in it's budget, behavior is the same as classic Neutrino scheduling. However on a loaded system, it is best to assume that a Round-Robin thread may be sliced every tick. However, when a Round-Robin thread is preempted by AP, it will be 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 AP Scheduling?#

First non_AP (i.e. classic Neutrino) scheduling:

- If not preemepted by a higher priority thread, a SS thread will run until it voluntarily gives up control.

With AP scheduling, an FIFO thread will:

- If not preempted by a higher priority thread in the same patrition, an SS thread will run until it voluntarily gives up control, or it's partition runs out of budget.

In other words, FIFO behavior is unchanged as long as your partition has budget. On a loaded system it is best to assume that an FIFO thread may be timesliced 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 Neutrino scheduling.

How long can a Sporadic (SS) thread run with AP Scheduling?#

First non_AP (i.e. classic Neutrino) scheduling:

- If not preemempted by a higher priority thread, a SS thread will run until it voluntarily gives up control.
- Because the priority of an SS thread changes from a normal priority to a low priority, it is much more likely to be preempted when running at it's low priority.

With AP scheduling, an SS thread will:

- If not preempted by a higher priority thread in the same partition, an SS thread will run until it voluntarily gives up control, or it's partition runs out of budget.
- Some developers set at the higher priority of an SS thread to be the highest in the system in order to make that thread un-preemptable during it's high-priority mode. With AP scheduling the thread is unpreemptable only as long as it's partition has not exhausted its budget.

In other words, SS behavior is unchanged as long as your partition has budget. On a loaded system it is best to assume that an SS thread may be timesliced 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 Neutrino scheduling.

How often does the AP scheduling algorithm run?

See next answer

How often does the AP scheduler enforce budgets?#

The AP scheduler runs and enforces budgets Whenever any of these events occur:

- Every tick.
- Every time a thread sleeps or blocks for a mutex.
- If a thread becomes ready, say because it received an event, pulse or message.

Mostly, the frequency depends on how often messaging occurs.

How do power-saving modes affect scheduling?#

If the system suspends, and resumes, AP is unaware of the interruption. Upon resumptions, partitions will have the percentage consumptions they had at suspensions. If the system varies processor speed to conserve power, AP is unaware of the variation. AP will continue to guarantee all partitions their budget percentages, but will assume that each millisecond has the same throughput. This means that scheduling will be effectively inaccurate for the 100ms (or windowsize) after a cpu changes speed. Thereafter it will be accurate.

On SMP systems, APS assumes all CPUs are in the same power saving mode.

How does changing ClockPeriod() affect scheduling?#

Changing clock period will cause AP to schedule inacurately as it is unaware of the change in the size of the tick. However, calling SchedCtl(SET_APS_PARMS,...) with the exiting windowsize will cause AP to recalculate all internal parameters that depend on the size of the clock period and therefore will restore accuracy. (This is why the user guide recommends setting the AP windowsize after any changes to clock period.)

Microbilling#

How does microbilling work?#

Micirobilling means accounting for the cpu time used by a thread to much finer resolution than the clock period between tick interrupts. AP 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 thread's partition's used cpu cycles count. Microbilling uses the system call ClockCycles() to get that fine-resolution timestamp.

How often does AP Microbill?#

Each time any of these events occur:

- One thread stops running and another starts running
- A clock tick occurs

How does ClockCycles() work?#

That depends on the processor being used.

On x86 processors, we use a free-running counter which is implemented on the cpu chip. We read it with a single instruction.

On PowerPC targets we read a similar free-running counter with just a few instructions. In those cases, ClockCycles() increments typically at about the processors clock rates(i.e. ClockCycles() increases by 3 billion counts every second on a 3Ghz machine.) On processors which do not have a free-running counter for the purpose of being a fine-grained clock, Neutrino emulates ClockCyles(). For example, on the ARM processors, we read the intermediate value of the count-down timer use to trigger the clock interrupts. That value tells us

how far we are into the current clock tick. We add a scaled version of how far we are into the current clock tick, to a constant determined at the last clock tick to get an emulated ClockCycles() value. On some processors, line ARM, the count-down timer used for emulating ClockCycles() is located off-chip and requires slow io-operations to read. On other processors, like MIPS, the count-time timer is located on chip, and is quick to read.

How accurate is microbilling?#

See next answer.

How accurate is ClockCycles()?

The accuracy of microbilling, or ClockCycles() is determined by the accuracy of the clock oscillator source used to provide the cpu with it's clock signal. However, since AP schedules relatively between partitions, it does not require ClockCycles() to be equal to absolute time. It only requires ~Clockcycles() to be proportional to work done by the cpu. Deliberately miss-calibrating ClockCycles(), assuming that's possible, would have no effect on the accuracy of AP scheduling.

What is the resolution of thread timing?#

It is the resolution of ClockCycles(). The resolution of clock cycles varies from platform to platform. However, in all cases is is much finer that the 1/200 of a tick that AP requires to be able to meet it's spec for accuracy. (In some platforms, like x86 the resolution is on the order of nanoseconds.)

Averaging Window

How does the averaging window work?#

The averaging window is composed of a bunch of tables. There are two tables per partition, one for cpu time spent while critical, and another for any cpu time spent. The tables have one slot per timer tick. So a 100ms averaging window, with a 1ms clock period would have 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() (which are then scaled by a factor carefully chosen so that all numbers will fit into a 32 bit unsigned int.) At any time, the sum of the elements of a table represents the total cpu time used but that partition over the averaging period. When the scheduler stops running a thread, 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 critical, it also ads 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 it's partition's `[current ms]` slot. It also rotates the table. To rotate the table, it:

- Deletes the `[99ms ago]` slot
- Shifts the entire table to the left by one slot, moving time in the "98ms ago" slot to the `"99ms ago slot"` etc,
- and creates a new "current ms" slot which it initializes to zero.

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

What is the window rotation algorithm?

To avoid taking ridiculous amounts of cpu time every clock tick, the averaging window is not physically rotated. It is logically rotated this way:

- A separate field, used_cycles, is maintained to always contain the total of every slot in the table
- A integer, cur_hist_index, is an index into the table and points to the slot which is "`[current ms]`"
- On microbilling we add the cpu time of the current thread to both the current slot in the table, and also to the total field. ex
  - `usage_hist[cur_hist_index] += delta_time; used_cycles += delta_time;`
- On window rotation, we:
  - subtract the oldest slot from the total:
    - `used_cycles -= usage_hist[(cur_hist_index +1) MOD 100]`
  - and increment the table index, modulo it's table size, say 100:
    - cur_hist_index = (cur_hist_index+1) MOD 100

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

Can I change windowsize?#

See next answer.

How does changing the windowise affect scheduling?

You can change the windowsize with the SchedCtl(SCHED_APS_SET_PARMS,...) on the fly. The scheduler does not malloc() new tables, but it does zero the history in all tables, zeros all totals, and zeros table indexes. The effect is to wipe the memory of the scheduler. That means that the scheduler assumes no partition has run in the last x milliseconds where x is the new windowsize. This is why the windowsize should not be changed often. Leaving it at the default or setting it once during startup is recommended.

How does maximum latencies related to the averaging windowsize?#

In general, the longer the averaging window, the longer a partition might have to wait before it gets cpu time. For example, with a 100ms averaging window and a partition (p) with a 10% budget, if (p) runs continuously for 10ms, it will exhaust it's budget. It will then be another 90ms before window rotations will cause the averaging window to loose memory of it's past execution. Another way of saying this is that it will be 90ms before (p) gets some available budget back and runs again.

However, in most real system that engage in inter-partition interaction, (p)'s 10 ms of running time is likely to get spread out in the averaging window. So even if it exhausts it's available budget in 100ms, it will most likely get available budget back in much less than 90ms.

There is a very unlikely scenario where two interacting partitions can result in a latency of larger than windowsize-budget. It is documented on page 42 of the Adaptive Partitioning Technology Development Kit User's guide.

Scheduling Algorithm#

How does AP pick a thread to run?#

See next answer

How does the AP scheduling algorithm work?#

Basically it evaluates a merit function on each partition, and chooses the partition with highest merit. It then picks the highest priority thread in that partition. The short version of this is that a partition with budget has more merit than a partition that has exhausted it's budget. The details: First a few helper functions:

- let COMPETING(P) be a boolean function of partition p. It returns True if:
  - parition p is currently running a thread of priority greater than zero, or
  - partition p contains a thread, which is ready to run, and has a priority greater than zero.
- let HAS_BUDGET(p) be 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) is the cpu time that partition has used during the current averaging window, and budget_cycles() is the size of the averaging window, expressed in ClockCycles() (not milliseconds) multiplied by the percentage budget of p.
- let MAY_RUN_CRITICAL(p) be a boolean function of partition p. It returns true if:
  - partition p is configured with a critical budget greater than zero
  - partition p has used, during the last averaging window, critical-time is less than it's critical budget minus 1/32 of a tick.
  - the highest-priority thread, which is ready to run, or is currently running, in partition p is marked "allowed to run critical".
- let HIGHEST_PRIO(p) be the numerical priority of the highest priority thread which is either running or ready to run in partition p.
- let RFF(p), "relative fraction free" be 1 - used_cycles(p)/budget_cycles(p) if the partition has a non-zero budget. If the partition has a budget of zero, RFF(p) is defined to be a constant smaller than the smallest possible value of RFF() for all other non-zero partitions.

Some operating modes, defined by these boolean expressions:

underload
: when COMPETING(p) && (HAS_BUDGET(p)||MAY_RUN_CRITICAL(p)) == True, for at least one p.

all_at_load
: when COMPETING(p) == True for all p, and HAS_BUDGET(p)||MAY_RUN_CRITICAL(p) == False, for all p.

free_time
: when COMPETING(p) == False for at least one p which has a non-zero budget.

idle
: when COMPETING(p) == False, for all p.

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

underload
: merit(p) = (COMPETING(p), HAS_BUDGET(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_BUDGET(p)||MAY_RUN_CRITICAL(p), HIGHEST_PRIO(p), RFF(p))

free time SCHEDPOL_RATIO
merit(p) = (COMPETING(p), HAS_BUDGET(p)||MAY_RUN_CRITICAL(p), RFF(p))

idle
merit(p) is 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, which 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 like tuples. I.e. (a,b) < (c,d) if (a<c) || ((a=c) && (b<d))

How does the scheduler find the highest prio thread in a partition?#

It does it very quickly.

Each partition has a bit map which tracks which of each of the 0 to 255 priority levels is 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 we run a thread, meaning we have just changed it's state from ready-to-run, we examine the queue of threads in that partition which are ready to run and at the same priority. If there are no other threads of that priority, we clear the bit for that thread's priority.

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

The same mechanism is used in classic Neutrino scheduling. The macros are DISTPATCH_SET(), DISPATCH_CLEAR() and DISPATCH_HIGHEST_PRI().

How are RFFs (relative fraction free) computed?#

Computing the RFF() function of the scheduling algorithm nominally requires floating point divides. However we can not do floating point inside the kernel and even fixed-point division is very slow on some platforms. So we compute a function equivalent to RFF() which requires only addition and multiplication.

How do you avoid division and floating point math?#

Computing the RFF() function of the scheduling algorithm nominally requires floating point divides. However, we don't need the absolute values of RFF(), we need only to know the relative ordering of RFF(p1), RFF(p2), ... RFF(pn).

So, instead we compute, a different function which has the same ordering properties as RFF(). This function is chosen to be computable with only addition and 16x16 bit multiplies.

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 which has the same ordering properties as RFU():
   - we want: 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
   - 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.
   - the 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 are not recalculated during scheduling
   - at scheduling time, we compute 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:

1. Running out of bits: So far f(p) = used_cycles(p) * c(p) requires 64 bit multiplies. However, since our accuracy spec it 0.2%, we scale all values of c(p) by a common factor, until the largest fits in 16 bits. We also shift used_cycles(p) until it's largest possible value fits in 16 bits. Therefore, at scheduling time, we need only compute f(p) = (used_cycles(p)>>scaling_factor) * scaled_c(p).
2. Zero budget partitions: The above algorithms would nominally require us to multiply and divide everything by zero. However RFF() of a zero budget partition is defined to be a constant smaller than any non-zero partition's possible value of RFF(). So we can define RFU(p) for a zero budget partition to be a constant greater than the RFU() than any partition. The largest value of f() is windowsize_in_cycles * c(pm) where c(pm) > c(p') for all p' not equal to pm. Therefore we can set f() for a zero budget partition, = f_zero = 1 + windowsize_in_cycles*c(pm), and then scale it as per "running out bits".

How does the AP algorithm determine if a thread which is allowed to run critical, should actually run critical?#

How does the AP algorithm decide when to bill critical time?#

When the AP algorithm picks a thread allowed to run critical to run, it does not always charge its cpu time to its partition's critical budget. A thread, (t), charges it's cpu time to the critical budget of it's partition, (p), only if all of these are true when the AP scheduling algorithm is invoked.

1. (t) has the highest priority in the system.
2. (t) is allowed to run critical now
3. (p) has a critical budget configured to be greater than zero.
4. the cpu cycles used by all threads in (p) during the last averaging window is less than the critical budget of (p).
5. (p) has exhausted it's normal cpu budget
6. at one partition, p' not equal to p, has COMPETING(p') &&(HAS_BUDGET(p')|| MAY_RUN_CRITICAL(p')) == True. (See scheduling algorithm for definitions of COMPETING(), HAS_BUDGET(), and MAY_RUN_CRITICAL().)

What are the algorithm's size limitations?

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

- <= 32 partitions, because of use of bit sets and 32 bit integers.
- <= 16 partitions, because of an internal step of RFF calculation limited to 16x16 bit multiplies
- <= 8 partitions, a practical limit to prevent too much memory or cpu time consumed by the scheduler.
- Budgets, in percent, must be specified as integers. i.e. 30%, or 31%, but not 30.5%.
- no limit on the number of threads per partition

What are the algorithm's accuracy limitations?#

By accuracy, we mean, how closely the scheduler can guarantee/limit a partition to be consuming only it's budget on a loaded system. The accuracy limit is whichever of these is greater:

1. 0.5%, or
2. tick size (in milliseconds) / windowsize (in milliseconds). For a 100ms window, with a default tick, this is 1%.
3. When the user changes the averaging windowsize to x milliseconds, the accuracy is undefined for the next x milliseconds.

Limitation 1 comes from the accuracy to which the RFF() calculation is carried out. The accuracy of RFF() is calculated to a limited number of bit's specificially to speed up the scheduling algorithm. Limitation 2 comes from the uncertainty in predicting how long a thread will run before it voluntarily blocks, is preempted by a higher priority thread, or when the next tick interrupt occurs. In other words, this limitation comes from the fact that the AP scheduler is guaranteed control of the system only every tick (but may run more often). In practice, limitation 3 means that when a the windowsize is changed, the scheduler clears its history of used cpu time. So the partition,(p), with the highest priority thread will run for budget(p)*windowsize milliseconds before another partition will run. After windowsize milliseconds have elapsed, all budgets will again be guaranteed. So a partition, configured for a budget of 40%, with a 100ms averaging window, would be considered to be scheduled accurately when it's usage over the last 100ms was 39 milliseconds to 41 milliseconds -- provided the windowsize was not changed in the last 100ms. In practice, the scheduling accuracy is usually much better.

When is the scheduling algorithm approximated?#

To save overhead, a very short version of the scheduling algorithm is use on some paths involved in message passing. These are in the internal scheduler functions ready_ppg(), block_and_ready_ppg(), adjust_priority_ppg()

Overhead#

To which partition is the overhead of scheduling charged?#

Let's consider to be overhead of all kernel calls that switch threads, like messaging and mutexing. Let the initially running thread be called t1, let the next thread that runs be called t2. We consider kernel calls that are initiated by t1, which cause t1 to stop running and t2 to start running. The short answer is that is is split between t1 and t2, but mostly to t1. The details:

time to do	Charged to partition of
Entering kernel	t1
Scheduling algorithm	t1
Context switch	t2
Exiting kernel	t2

To which partition is interrupt processing charged?

There are two portions of interrupt servicing: the interrupt handler and the interrupt thread. If the user is servicing interrupts with an interrupt thread, most of the time spent servicing the interrupt will be the thread's time, and only a small portion will be spent in the interrupt handler which determines to which thread the interrupt event should be delivered. If the user is servicing interrupts with an interrupt handler, all of the time spent servicing the interrupt in the handler. The time spent in the interrupt thread is charged against the partition of that thread. The time spend in an interrupt handler is charged against the what ever partition happend to be running at the time. Because the occurrence of interrupts is essentially random, time spent in interrupt handler is spread evenly over all running partitions.

How much cpu overhead with AP scheduling?

Heavy compile benchmarks, which involve a lot of filesystem related messaging, are about 1% slower on x86 platforms than without AP scheduling.

How much memory overhead is there with AP scheduling?

Data: a few Kb of fixed overhead, plus about 2Kb per partition.

Code: about 18Kb

Both of which are in the kernel space.

What factors increase AP scheduling overhead?

In approximate order of importance, the cost of AP scheduling increases with:

- number of scheduling operations. such as message, event and signal sending, mutex operations, and sleeps
- platform. In particular ARM is noticeably slower because of the IO needed to implement ClockCycles()
- frequency of clock ticks
- number of partitions
- use of runmasks

In all cases, the increase is approximately linear. Also these factors do not affect the cost of scheduling at all: *number of threads *length of averaging window, (except for very small effect when changing windowsize) *choice of percentage budgets *choice of thread priorities *choice of FIFO, Round-Robin, or Sporadic thread policies

---

Critical Threads and Bankruptcy

How is a thread marked critical?

See next answer How does the AP scheduler know a thread is critical? Neutrino maintains a data block representing the state of each thread: the thread_entry. It contains 3 state bits for controlling AP scheduling's critical threads. They are:

1. this thread is always allowed to run critical
2. this thread is allowed to run critical until it blocks.
3. this thread is currently running critical (and is consuming it's partitions critical budget).

These state bits are turned on as follows:

always allowed#

When the user calls the SchedCtl() with the SCHED_APS_MARK_CRITICAL command on that thread.

until blocked#

When the thread receives any of:

- an event from an interrupt handler
- a message from another thread marked either "always allowed to run critical", or "allowed critical until it blocks".
- an event, on which the user has previously called the macro, siginfo.h::SIGEV_MAKE_CRITICAL()

currently running critical#

When the AP scheduling algorithm decides that thread would not have been eligible to run if it had not been allowed to run critical.

Are critical threads a security exposure?

No.

Anyone can set their own thread to be critical, or receive a critically tagged event or message. However, that only gives the thread the property of `"allowed to run critical"`. To affect the critical budget of its partition, and cause its partition to run when it's out of budget (thereby taking time from some other partition), its partition must have been configured with a non-zero critical budget. Setting a non-zero critical budget on a partition is controlled. For the recommended AP security settings, only root running in the parent partition of a target partition can set a non-zero critical budget.

When do we check for bankruptcy?#

To save time, AP only polls partitions for bankruptcy on each clock tick. (Rather than every scheduling operation.) So typically, bankruptcy may be detected a millisecond (or clock period) after a partition's critical budget has been exhausted.

How do we detect bankruptcy?#

We compare the total critical time used by a partition, used over the last averaging window, to the partitions configured maximum critical time budget. Each partition maintains a separate rotating window for tracking critical time usage. The critical time history window identifies for each ms of the last 100ms, which portion of that millisecond's total cpu time was considered to be critical time.

Inheritance#

What is partition inheritance?#

When the scheduler bills the cpu time of a thread, not to it's own partition, but to the partition of a different thread, we call this partition inheritance. It is another reason why AP is adaptive.

When does partition inheritance occur?

The scheduler inherits partitions under two circumstances:

1. When one thread is working on behalf of another
2. Under special circumstances when not inheriting might cause excessive delays

Case 1: When a client thread sends a message to a server thread, that server thread is considered to be working on behalf of the client thread. In that case, we will charge the execution time of the receiving thread, from the time it receives the message and up to the time it next waits for a message, to the partition of the sending thread.

This means that resource managers, like filesystems, automatically bill their time to their appropriate clients. This usually means that partitions containing only resource managers no not need to be re-engineered every time a new client is added to the system.

Case 2 occurs in a special case of mutex inheritance

How does mutex partition-inheritance work?#

When threads line up for access to a mutex, we don't consider the thread holding the mutex to be waiting on behalf of the threads waiting for the mutex. So we do not inherit partitions.

However, there is a special case when the thread holding the mutex is in a partition which ran out of available budget. That means it cannot run and release the mutex. That means that all the threads waiting for that mutex are stalled until enough window rotations have occurred to return the partition of the mutex-holding some available budget. This is particularly nasty if the user has configured that partition to have a zero budget.

So, when: *a mutex is being held by a thread, (p1), in a partition which has exhausted its budget, and *another thread (p2) attempts to seize the mutext

we: *put (p2) to sleep, until (p1) releases the mutext (which is classic mutex handling) *we change the partition of (p1) to be the partition of (p2) until it releases the mutext, provided the budget of (p2) is non-zero This prevents extended delays should the current mutex holder run out of budget.

Someone has suggested this is analogy: A lineup of people are waiting for access to a vending machine. If they guy at the head of the line is fumbling ineffectively for change, the guy at the back of the line give him coins so he'll hurry up. Less polite human interactions are also possible, but we implement only this one.

How fast is inheriting a partition?#

Very fast.

The data block that Neutrino keeps for each thread, the thread_entry, has a pointer to its containing partition. So inheritance is swapping the pointer. Often, we don't even need to update the microbilling because the same partition will be executing before and after the inheritance.

Why is partition inheritance for message passing secure?

Sending a message to a process, effective 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. (Usual comments about to err is human, but to really mess up you need the root password.)

Budgets#

Can I change budgets dynamically?#

See next answer

How does changing a budget affect scheduling?#

How quickly does a budget change take effect?#

You can change a partition's budget any time. The operation is quick and does not reset the scheduler or cause any change to the partition's history of cpu usage stored in the averaging window. However, if you change the budget of a partition from 90% to 10% it will quite likely suddenly be come under budget and may not run again until enough window rotations have occurred to lower the partition's used cycles to below its budget.

When does a change in budgets take effect?#

At the next tick interrupt or next scheduling operation. i.e. in typically less than a millisecond.

What does a partition with a zero budget mean?

Threads in a partition with a defined budget of zero will run if all non-zero partitions are sleeping. They will also run if they inherit the partition of thread which sends a message. Zero budget partitions are most useful to contain resmgrs with no internal daemon threads. They're also useful for turning off unused partitions.

How do we guarantee that the sum of all partition's budgets is 100%#

At startup Neutrino creates the first partition, System, having 100% budget. Thereafter, when a thread running in a partition creates a new partition, the current partition is considered to be the parent and the new partition is the child. The budget of the child is always take 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 System's original budget of 100%.

How do we prevent an untrusted thread from increasing it's partition's budget?

For any change to occur, first:

- AP security would have to be unlocked to permit budget changes.
- AP security would have to be set to permit non-root users to modify budgets

- AP security would have to be set to permit a partition to modify it's own budget (Rather than being modifiable only by it's parent.)

And even then, a thread in a partition cannot increase its budget to be larger than the the budget of it's parent partition.

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

How	But first
Change windowsizes often	You must be root, AP configuration must be unlocked, or AP security must be turned off
Give your partition an infinite critical budget, and set yourself to run critical	You must be root, AP configuration must be unlocked, or AP security must be turned off

Good looking ways of cheating that don't work are:

- Giving your own partition more budget. (Can't exceed parent, even if security if 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 looping. (You can starve everything else in your partition, but not another partition.)
- Create you 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 next answer

How fast is joining a thread to a partition?

Each thread_entry (the control block Neutrino maintains for each thread) has a pointer to its containing partition. Joining a thread means only changing that pointer. The join itself is very fast. Most of the time is spent simply entering the kernel so we can swap the pointer.

QNX System Considerations#

Why don't we allow a partition to be deleted?#

It's much more efficient and much safer to not delete a partition. Setting a patrition's budget to zero is the suggested alternative. To delete a partition, we would have to locate all threads which are members of the given partition, and either assert that there are none, or move them all to some other partition. However, threads are mapped to their partitions with a single pointer. There is no back pointer as that would be a many-to-one mapping. It would require yet another linked-list to chain together all threads in a partition. We would require additional kernel memory for a 2-way queue through all thread_entries. Plus we would have to do two-way queue extractions every time we inherited partitions (I.e. message sending.) while evading the simultaneous destruction of other threads.

How does AP plug into procnto?#

See next answer

Is the classic scheduler still present when AP is active#

AP is effectively part of the kernel. It is shipped as a libmod which is built into the image along with procnto. Procnto still contains the code for the classic scheduler when the AP libmod is present. However, when the AP libmod is present, procnto initializes AP instead of the classic scheduler. AP then directs a set of function pointers, one for each primitive scheduling operation (like ready(), block(), etc), to it's own function constants. It then creates the system partition which it returns to procnto.

Does the AP scheduler inhibit IO interupts?

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

Is there a performance limitation how often one can call SchedCtl(SCHED_APS_PARTITION_STATS,...) to get statistics.#

Other than the cost of the SchedCtl kernel call, no. Getting statistics does not inhibit interrupts and does not delay the window rotation or the scheduling algorithm on other SMP processors. Consitant retrieval of statistics is accomplished by detecting collisions and having the API back off and retry. Only in the incredibly unlikely case of 3 consecutive collisions will the SCHED_APS_PARTITION_STATS API fail with EINTR. In general, that can only happen if the user has set the clock period to a value so short it is likely to be unsafe for the rest of the system too.