Adaptive Partitioning Scheduler: Requirements#

Table of Contents

What is it?#

It's a fair-share thread scheduler which guarantees groups of threads a user-specified percentage of cpu time when the system is overloaded. When the system is sufficently underloaded, it chooses to schedule threads based strictly on priority and therefore maintians realtime behavior. Even when overloaded it provides realtime latencies to an engineerable set of **critical** threads.

We call a group of threads working for the same purpose a **purpose group**. Nominally, a purpose group is composed of the threads in a user-specificed set of processes. But during the time a server thread does work on behalf of a client thread, the server thread is considered to be a temporary member of the client's purpose group. Cpu usage, and guarantees, are tracked by purpose group.

Because competing schedulers use the word "partition" to mean "group of threads" being scheduled together, the external name for our scheduler is the adaptive partition scheduler to emphasise that the variable membership of our partitions. **Purpose Group** and **Adaptive partition** are interchangeable terms.

High Level Requirements#

System Considerations

- The adaptive partition scheduler is optional and has minimal cpu and memory costs when not installed.
  - q: what are the measured overheads when the ap scheduler is not installed?

- The overhead of adaptive scheduling may not reduce the throughput of the sytem as load increases. That means the overhead of ap scheduling may not increase with the number of threads. But it may increase with the number of partitions.
  - This is to make sure the scheduler itself does not trigger overload or causes overload to be self-perpetuating.

- The scheduler will try to minimise the ready-queue delay times whether or not the system is overloaded.

- The adaptive scheduler will be retrofittable on an existing application design and work with the existing choice of thread priorities.

The user's interface#

- The user may specify the guaranteed-percentage cpu, and all other properties, of a set of adaptive partitionss at boot time. The user may not add adaptive partitions or change percentages at run time.The user may specify partitions' cpu percentages to 1%.

- The user may specify an averaging windowsize. This is the time over which the scheduler will try to keep adaptive partitions at their guarateeded cpu percentages, when the system is overloaded. A typical time is 100 milliseconds. The window size is specified only at boot time and is the same for all partitions.
  - For consistancy with other apis, the window size will be specfied in milliseconds, but will be approximated internally as a number of timeslices.
  - See the Users' Guide for how choice of window size affect accounting accuracy and latencies in extreem cases.

- The user may specify up to 4 operating modes. A mode is a set of adaptive-partition parameters. All modes must be specified at boot time. However, a run-time api will allow the user to switch modes as needed.
  - example: A separate set of adaptive-partition cpu-percentages may be needed during startup versus normal operations. To do that, setup two different modes have the application switch from restart-mode to running-mode when it's intialization is complete.
  - Note: modes may not be delivered in qnx 6.4.

- At boot time, the user may label selected adaptive partitions as critical and give to each a critical time budget. The critcal time budget is specified in milliseconds (actually in as nanoseconds rounded to the nearest ms). It is 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 it's partition still has critical budget.

-
  - A critical thread is one which is intiated by an IO interrupt. Also, the user may specify a set of timer interrupts make their associated handler threads critical. An api will also allow users to mark selected threads as critical.
  - The user may specify the system's response to an adaptive partition exceeding it's critial time budget. The choices are:
    - force a reboot
    - notify the system watchdog, which may be CRM (critical resource monitoring) or supplied by the user.
    - drop a signal on a designated member process

-
  - The system will always log running out of critical budget. In general we consider an adaptive partition exceeding it's critial time budget to be an application design error.

-
  - See the User's Guide for security issues.

- A process may register to be informed when the system enters and leaves overload. The intention is that applications use overload notification to gracefully degrade their service. For example by skipping less important functions, or by reducing the precision of computation.

- Users may give a partition a name (up to 15 chars) at creation time. If a name is not provided, the scheduler will assign a name in the rage "Pa" to "Pz". All names must be unique.

Operating principles#

- a thread is in exactly one adaptive partition at any time.

- When the system is overloaded, the scheduler will balance adaptive partitions to within 1% of their guaranteed cpu percentages, or +/- timeslice/windowsize, whichever is greater.

- Interrupt handler priorities continue to have global meaning. If the system in underloaded, thread priorities have global meaning. In overload, the scheduler will maintain priority ordering of threads only within their adaptive partitions.

- The scheduler will account for time spent to the actual fraction of timeslices used and account for the time spent in interrupt threads and in the kernel on behalf of user threads. (aka *'micro billing*')
    - Microbilling may be approximated on sh and arm targets if the board cannot provide a micro clock.

- Time spent in interrupt threads is allowed to run an adaptive partition over budget and is not counted as critical time.
    - q: bug? critical time is specifically intended to allow interrupt threads to run over budget.

- Time spent in the kernel at interrupt level, including user's interupt handlers, is not accounted.
- Whenever a thread inherits priority from another thread (ex: message passing, or mutex contention or anywhere we implement priority-inversion avoidance), the inheriting thread temporarily inherits the adaptive partition of the sender. So the receiver's time, until it become associate with a different client, will be billed to the sender's adaptive partition.

- A critical thread remains critical until it becomes receive blocked. The criticality of a thread, or billing to it's ap's critial time budget, is inherited along with adaptive partition during message passing and mutexing.

- A critical thread, which is being billed for critical time, will not be round-robin timesliced (even if it's sched policy is round robin).

- If only one partition is defined, then the scheduler will never declare it to nave run out of critical time.

- When the system is not overloaded, and when one adaptive partition chooses to sleep, the scheduler will give the cpu time to other partitions.
    - This happens even if the other partitions are over budget.
    - The scheduler will hand out the 'free' time:
        1. if one ap has the uniquely highest priority, to that highest priority thread
        2. if two or more ap has the same highest priority: in proprotion to the other ap's percentages.

-
    - This is necessary to prevent long ready-queue delay times in the case where two adaptive partitions have the same priority. (See the No InterWiki reference defined in properties for Wiki called "Adaptive partitioning"!)
    - For example: There are three aps with 70%, 20% and 10% guarantees. The system is overloaded. When the the 70% ap goes to sleep, the scheduler will hand cpu time out to the 20% and 10% adaptive partitions in a 2 to 1 ratio.

- If the sleeping adaptive partition sleeps for a *short time* then the scheduler will make sure that the formerly-sleeping ap will later get cpu time up to it's guaranteed limit.
    - That means that the 20% and 10% adaptive partitions, of our example, must pay back the time they oppourtunistically grabbed.
    - a short time is less than than *windowsize - percentage*windowsize* milliseconds within one averaging window.
- If the sleeping partition sleeps for a long time, then some of all of the time given to other partitions becomes free.

- When we schedule a critical thread, we always count its run time against it's adaptive partition. But we only count it's time against the ap's critical budget if we would not have otherwise run it. That means that time spent in a critical thread time is not counted as critical when:
    1. the system is underloaded.
    2. the system is loaded but one or more aps are not using up their guaranteed cpu percentages.
- In general, cpu time charged against an adaptive partition's critical budget if an only if that ap would not have run had it not been critical.

- When the parameters of the set of adaptive partitions is changed at run time, say because of a mode switch, we may take up to *windowsize* time for it to take effect. Also, the scheduler will never immediately declare any adaptive partition to be bankrupt (being over critical-budget) because the partition's critical budget suddenly shrank as a result of a parameter change

- for measurement purposes, the scheduler will track time spent in idle, averaged over the same *windowsize* as partitions.
    - We could infer idle time from 100 ap_time, once we're confident in the sched algorithm's accuracy. - ad

---

Limitations, Design considerations and Questions

Membership and inheritance#

- There may be up to 16 adaptive partitions. One must be the system partition, where at least the idle thread and proc threads live. The sum of the cpu percentages of all adaptive partitions must always be 100%.
    - q: what % should the system adaptive partition group be given?

- Spawned and forked children appear in the adaptive partition of their parents. However, an api will be provided that will allow spawning threads into other adaptive partitions. This api will be avalable only to the system (root) and is intended to be used to implement a application launcher that launches processes at startup into their respective adaptive partitions.

- The system starts up in mode 0. I.e. the mode for initialization is mode 0.

- Message passing and mutexes must inherit the originator's adaptive partition. Pulses and sigenvents should allow specifying which adaptive partition the receiver should bill to. Criticality of a thread is inherited with adaptive-partition inheritance.

- Busy-server priority-inversion avoidance: When a client messages a resource manager which has all of it's server threads busy with other clients, we currently raise the priorities of all server threads that waited

on the same coid as our client. This is an attempt to get the server to hurry up so they can serve our client. When this occurs, for all the server threads whose priorities we are about to raise, we we must now begin billing the new client.
    - q: is it worth splitting the time spend in server thread so far and billing that to the previous clients?
    - q: should we bill all the time of the server threads who's priority were raising *'only*' if the server is out of budget?

- Adaptive partition parameters should be specified at boot time. The window size should be specified as a command line parameter to procnto. Other parameters should be settable in a runtime api. However, only an in-house tool should be allowed to run the api at any time. User loads should be restricted to running the api once, to specify the parameters of all adaptive partitions, per startup.
    - This is necessary to allow the same load to customize itself to different boards.
    - The parameters are:
        - for each adaptive partition and mode, the guaranteed cpu time in percent
        - for each adaptive partition and mode, the maximum critical time budget
        - for each adaptive partition and mode, the policy for how the system should respond should the partition exhaust it's critial-time budget
        - for each adaptive partition and mode, the length of the averaging windowsize in milliseconds
        - for each adaptive partition and mode, the list of processes which are initially part of each adaptive partition
        - the list of modes

Scheduling

- **Throttling**, or choosing to run threads not based purely on priority, must happen when at least one ap is over budget. (A workable impmenation would appear to have the same code for throtling and not throttling.)

- If all adaptive partitions are at their cpu limit, then we must run the highest-priority thread anyway. If two partitions have the same highest priority, then we run the partition that has used the least fraction of it's budget. This is needed to prevent long ready-queue delays which would otherwise occur. (See [Adaptive partitioning:User Guide|User Guide]].)
    - Example. If the windowsize is 100ms, ap1 is alloted 80% and has used 40ms, and ap2 is alloted 20% and has used 5ms, and both ap1 and ap2 are at prio 10, then we run ap2. That's because ap2's relative fraction free is 5ms/20ms, or 0.25, while ap1's relative fraction free is 40ms/80ms or 0.50.

- In the event a critial ap exceeds it critial budget and takes no corrective action (ex: masks notification or avoids restart), and if critical process monitoring is installed as the system watchdog, then CPM should set that ap's critical budget to zero. Then at least the mis-behaving ap will become fully throttled and will cease stealing time from other aps.

- When checking if an ap has enough budget to run a thread, it must have at least 1/2 a tick's worth.

- Threads triggered by interrupts in critical aps may will still run if their ap is over budget provided the max deficit is not exceeded. But the time spent in critical mode is still charged against the thread's adaptive partition, so other threads in the same ap may not run for longer as a consequence.

- High-priority threads, running with the fifo scheduling, which run a long time, will be pre-empted when their adaptive partition runs out of budget.

- If a thread exceeds bugdet in the middle of its timeslice we will allow it to run to the end of it's time slice. This causes the ap to briefly run over bugdet.

Accounting

- Windowsize choices should be limited from a minimum of two time slices to 255 milliseconds, i.e. 8ms to 255ms.

- Guaranteed CPU budget and critical time budgets are averaged by the same windowsize.

- An smp machine is treated as producing 100 percentage points of cpu time. Time spent spinlocked on one processor, while waiting to enter the kernel, is charged to the thread that is trying to enter the kernel.

- Time spent in kernel calls is charged to the adaptive partition of the invoking thread.

- On startup, set all averaging window slots to "nothing ran". This means that application will not be informed of overload for at least the first windowsize milliseconds.

- Overload or underload is property of the whole system, not a single adaptive partition. Also partitions may legitimately go over budget when some other partition is sleeping. Therefore one partition going over budget is not by itself considered to be overload. (And therefore will not trigger the overload notification api.)

- We will not provide a general interface to allow any thread to bill an arbitrary amount to a different adaptive partition. (Too many security risks otherwise). Instead resource manages, which discover late to which client they should have been billing, should simply have their own overhead budget. (See the[Adaptive partitioning:User Guide|User Guide]].)

- Time spend by idle should never be billed.

- Overload detection and notification of users:
  - q: what is the exact definition of overload. Possible alternatives may be:
    - system has not run idle during the the averaging window
    - the highest priority threads of an adaptive partition see a ready-queue delay time that exceeds a user-specified time.
  - q: what api do we need? Possibilities include:
    - delay notification until the system has been in overload for a specified number of milliseconds. This is to prevent too many notifications when the system is fluttering near overload.
    - require the user to re-register for the next notificaiton after each notification is delivered. This produces a natural throttling at whatever rate the client can accept.
  - Overload notification may not be implemented in 6.4

- Time spent in the kernel at interrupt level, including user's interrupt handlers, is not accounted.
  - q: Would it be worth counting at least the user's interrupt handler time against theirr adaptive partition? "Probably not worth the cost. -ad"

- Server threads will bill their time to the adaptive partition of the thread from which they received their message until they once again receive block or call message connect to associate themselves with another client.

Security and Logging

- The api will allow anyone to mark any thread critical. Similarly, there is no restriction on who may return a critical event from an interrupt handler. However, the critical state of a thread will be a no-op unless it is running in a partition with a non-zero critical budget. There will be security to limit who may set critical budgets.

- Loggable events, for example a adaptive partition exceeding critical budget, are detected by the kernel. But they will be logged by a thread in the Critical Process Monitor.
    - q: or the user may log the event in his own system watchdog process

Bankrupcy Handling#

It's pretty much described by the Handling Bankrupcy section of sys/sched_aps.h:

```
/* Handling Bankruptcy
* ===================
*
* Bankruptcy is when critical cpu time billed to a partition exceeds it's critical budget. Bankruptcy is
* always considered to be a design error on the part of the application, but the system's response is configurable.
*
*
* If the system is not declaring bankruptcy when you expect it, note that bankruptcy can only be declared if critical
* time is billed to your partition. Critical time is billed on those timesclices when these four conditions are
* all met:
*      1. the running partition has a critical budget greater than zero
*      2. 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.
*      3. the running partition must be out of percentage-cpu budget
*      4. there be at least one other partition that is competing for cpu time.
*
*      And then only if the billed critical time exceeds a partitions critical budget will the system declare bankrupcty.
*
*
* When the system detects bankruptcy it will always:
*
* 1. cause that partition to be out of budget for the remainder of the current scheduling window.
* 2. If the user has set a sigevent for notify_bankrupcty with SCHED_APS_ATTACH_EVENTS, deliver the event.
*    This occurs at most once per calling SCHED_APS_ATTACH_EVENTS.
*
* In addition the following responses are configurable. QNX recommends using SCHED_APS_BNKR_RECOMMENDED.
*/
#define SCHED_APS_BNKR_BASIC            0x0000000
/* This causes delivery of bankruptcy notification events and makes the partition out-of-budget for the rest of
* the scheduling window (nominally 100ms). This is the default.
*/

#define SCHED_APS_BNKR_CANCEL_BUDGET 0x00000001
/* Causes the system to set the offending partitions budget to zero, which forces the partition to be
 * be scheduled by it's percentage cpu budget only. That also means that a second bankruptcy cannot occcur.
 * This persists until a restart or SCHED_APS_MODIFY_PARTITION is called to set a new critical budget.
 */
```

```
#define SCHED_APS_BNKR_LOG                  0x000000002
/* Causes the system to log the occurence of bankruptcy. To prevent causing a flood of logs, contiguous bankruptcies
 * which occur while the same process is running, will be logged once.
 *
 * NOTE: For now, this is logged only to the system console output. Output to slogger is scheduled for a later
 * release.
 */

#define SCHED_APS_BNKR_REBOOT          0X000000004
/* The most severe response, suggested for use while testing a product to make sure bankrupcies will never be
 * ingored. Causes the system to crash with a brief message identifying the offending partition. Not recommended
 * for field use.
 */
```

Bankruptcy notifications are throttled. There is only one notiftation per partition per registration. I.e. users must re-register to get another notitification. If an offending partition in in a tight loop, the maximum rate of bankruptcy notification is determined by how quickly the thread-to-be-informed can received notififation and register for a nother notification.

An expection is if the user selects the SCHED_APS_BNKR_CANCEL policy. This causes the bankrupt partitions critical budget to be set to zero on its first bankrupcy. Because the critical budget is being set to zero, no furthter bankrupcies can occur until the user uses SCHED_APS_MODIFY_PARTITION to set a new non-zero critical budget. However, in order to cause the whole system to stabilize after such and event, the scheduler will give *all* partitions a two-window grace against declaring bankrupcy when one partition get's it's budget cancelled.

Billing execptions#

There are certain operations where the work will not be billed to the client.

- time spent mapping memory will always be charged to the partition of procnto, not the client. This means that threads that access very large tables, for the frist time, will appear to be using very little budget. This also means that the System partition should always have some budget to deal with mmap requests.