CODE TIME TECHNOLOGIES

Starvation Protection

Whitepaper

Copyright Information
This document is copyright Code Time Technologies Inc. ©2011. All rights reserved. Code Time Technologies Inc. may have patents or pending applications covering the subject matter in this document. The furnishing of this document does not give you any license to these patents.

Introduction

As processors become increasingly powerful, more and more features are being executed on a single device. Networking, user interface, signal processing; tasks that used to be handled by individual chips now compete for system resources on a single processor. And, in cases of extreme processing demands, the processor can become so heavily loaded by critical tasks that lower priority tasks may never have the chance to run. This is called *resource starvation*.

Many solutions exist to help enforce equitable resource scheduling, although they are usually implemented only on advanced real-time kernels, due to complexity. Furthermore, some of these solutions share the burden between the kernel and the end user, requiring careful analysis at the design phase and often result in suboptimal performance as the product evolves.

This paper examines the problem in more depth and details the available remedies, including those unique to the Abassi real-time kernel.

Priority Window

Traditionally, tasks are assigned a fixed priority at design time and the kernel schedules them accordingly. A higher priority task that needs to run will seize control of the processor from a lower priority task, and maintain control as long as required. However, this hard priority enforcement can be relaxed to allow flexibility on when a lower priority task gets preempted.

By assigning a priority window to each task, the designer has greater control over task preemption. Each task gets assigned both a base priority and a priority delta, below which it will not be preempted. The user is now able to specify that a running task should only be preempted by a *much* higher priority task, and not just *any* higher priority task.

Unfortunately, this mechanism can introduce a much worse problem whereby a high priority task within the window may be blocked indefinitely by a lower priority task, causing starvation of the high priority task.

Resource Partitioning

Instead of having all tasks in the system compete for unfettered access to the processor, resource partitioning segments the tasks into smaller groups, and the groups are allocated a portion of the processor time. The tasks within each group only contend with one another, and can never exceed the processor allocation of the group.

Two type of resource partitioning exist: *fixed* and *adaptive*. With fixed partitioning, if a group of tasks does not require all the CPU cycles assigned to it, those cycles are not reallocated and end up wasted. This can result in a very high performance processor being greatly underutilized. Adaptive partitioning remedies this by reassigning unused cycles from one partition to another.

Regardless of the partitioning method employed, this approach does not eliminate resource starvation, and may in fact exacerbate the problem. For example, on a fully utilized system containing three partitions, each with 3 tasks of different priorities, you may end up having 6 starved tasks; only the highest priority task within each partition would execute. Resource partitioning offers protection between task groups, but not within the group itself.

Priority Aging

Priority aging is a mechanism usually reserved for high performance multiuser systems. With it, tasks are monitored to ensure they are not being starved of access to the processor. If they are, priority aging gradually increases the task's priority until it gets scheduled, after which the task will revert to its initial priority upon blocking or getting suspended.

This is most valuable for short duration tasks that must execute at least occasionally, even on heavily loaded systems, and for very long duration tasks which do not have hard deadlines but need to run predictably.

However, a task which runs due to priority aging must not cause a critical task to become starved for resources either, and a robust priority aging implementation will prevent this, such that hard real-time requirements are met.

Abassi Advantages

Recognizing the inherent flaws with priority windows and resource partitioning, the Abassi kernel implements a highly modified priority aging mechanism, optimized for hard real-time environments. By adding an age ceiling, enforcing limits on an aged task's access to the processor, and confining priority aging to a single task at a time, overall system reliability and predictability is greatly improved. And the solution seamlessly coexists with all other kernel features, including priority inheritance, round robin scheduling, etc.

Priority Cap

Traditional priority aging steadily increases a task's priority until it is scheduled. In systems containing critical tasks which must not be preempted, or must execute in a timely manner, this behavior is not acceptable. To correct this, Abassi allows the user to selectively limit the maximum priority a task can be aged to.

A good example of this is a vehicle's collision avoidance system. If such a mechanism has engaged and is in the process of computing the necessary countermeasures, even a short interruption could be catastrophic, versus the temperature control system which can be delayed without adverse effect.

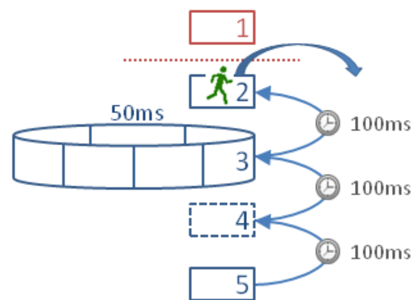
By adding a priority cap, Abassi combines standard priority aging with the solitary benefit of priority windowing, creating a robust yet automated protection mechanism, perfectly tailored for hard real-time demands.

Priority Snap Back

Conventional priority aging does not limit the time an aged task can keep control of the processor once it has been scheduled. This can result in unbounded priority inversion, as a lower priority task can preempt a high priority task for an unlimited duration.

The Abassi real-time kernel only permits the aged task to execute for a user defined duration, after which it reverts to its original priority, and may begin the priority aging process again. If it was not for priority snap back, very low priority tasks that were aged to the running state could monopolize the system. This mechanism preserves overall system responsiveness, and enables background tasks to execute, even on a fully utilized processor.

Consider an example with the starvation mechanism configured for a wait time of 100ms, a run time of 100ms, and a priority cap of 2. Round robin scheduling is present at priority 3, with a time slice of 50ms.



A low priority task is under starvation protection and was not scheduled within the 100ms wait time, so its priority is increased from 5 to 4. This happens again the next 100ms; its priority increases to 3, and it enters the round robin queue. Depending on the round robin scheduling, the task may execute for a single 50ms time slice, but will not reach the 100ms run time requirement, and will be promoted a final time. Since the task has reached the priority cap, it remains at that priority until it has fulfilled the 100ms run time requirement, at which time it snaps back to its original priority of 5.

Aging Queue

The Abassi real-time kernel allows multiple tasks to be queued for starvation protection, but only one will be actively in the aging process. Once the aging task has run for its configured duration, it reenters the queue, and the next protected task will begin the aging process.

This is done to guarantee fairness for all tasks in the queue, and to ensure that high priority tasks do not get stalled by multiple aged tasks sequentially. If multiple tasks could age simultaneously, the aging could overlap such that high priority tasks were blocked for a long duration, adversely affecting the overall system.

Adaptive Resource Partitioning

Even though resource partitioning is not effective protection against starvation, it can be approximated through judicious use of the Abassi round robin scheduling mechanism, which permits per task time slice configuration.

For example, consider three partitions, each containing two tasks, which should receive 50%, 30% and 20% of the CPU, respectively. All tasks must have the same priority level, with tasks in the first partition configured for 5 time slices each, in the second for 3 time slices each, and in the final one for 2 time slices each. Now, the tasks will all execute for up to their configured duration, or if they do not need the entire allocation, they will block and the remaining time will be automatically reallocated to the other queued tasks.

Conclusion

By employing a greenfield design approach, Code Time Technologies has been able to create a next generation real-time kernel that vastly improves upon any available today. The Abassi real-time kernel outperforms all existing platforms by combining code size and CPU efficiency with an unrivalled set of features.