# Deos™ Slack Scheduling

Time partitioned real-time operating systems guarantee that a specific computation will have access to the CPU for a specific amount of time (i.e., budget) at a bounded, deterministic location within the scheduling timeline (i.e., the hyper-period). This guarantee is a key enabler towards the development of highly integrated systems that allow software of varying degrees of criticality to coexist on the same platform. High criticality computations are guaranteed access to the CPU even in the face of misbehaving lower criticality software. In other words, one can always ensure their high criticality applications have the CPU time they demand by budgeting for their *worst case* execution. This approach works well for *periodic* computations that have a small deviation between their *worst case* execution and their *nominal case* execution.



Lightly Loaded ARINC-653 Timeline
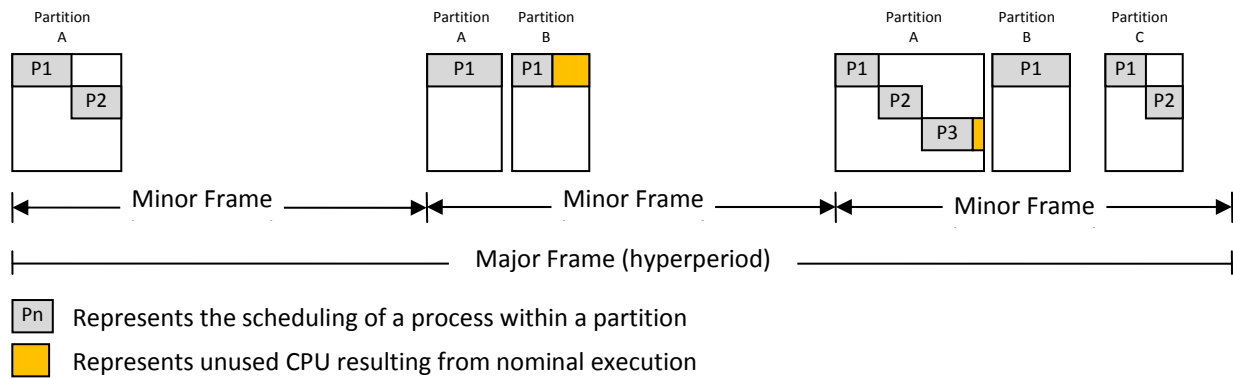
However, this guarantee comes at a price. By setting aside *worst case* execution times, the scheduler ensures *worst case* performance, *every time*. In other words, from a CPU bandwidth perspective, it is as if every time the hyper-period is executed, every computation experiences it *worst case* execution, *every time*. This leads to the all too common occurrence of being out of CPU *budget* time (i.e., no more time to allocate in the hyper-period), while profiling shows actually average CPU utilization of 50% (or less in some scenarios). The cause of this situation is the fact that, in the vast majority of cases, computations experiencing their *worst case* executions are rare. Even more improbable is the occurrence of multiple computations experiencing their *worst case* executions during the same period. Thus, for the vast majority of executions, unused CPU budget shows up as useless CPU *idle time*.
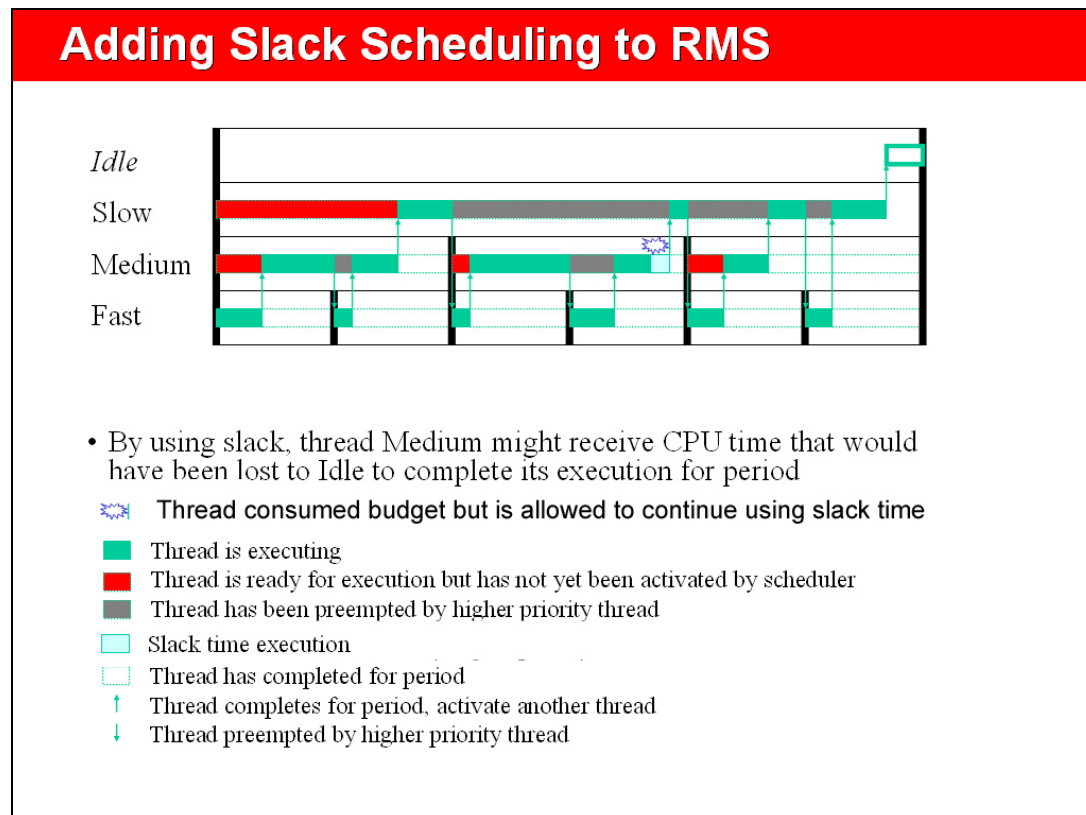
Further exacerbating the problem are the following common scenarios:

1) Aperiodic activities must be performed (e.g., interrupts to service, aperiodic client-server exchanges, etc.).

2) The CPU budget required in order to guarantee *safe* execution is insufficient to meet *customer delight* performance needs.

3) The deviation between *nominal case* and *worst case* execution times is significant.

Deos'™ patented slack scheduling technology addresses these all too common problems and enables designs to utilize 100% of the CPU.

What is slack? When considering slack, it is often helpful to think of a bank account. Deposits of time are made into the slack account from which threads (i.e., computations) can withdrawal until the account balance reaches zero. Where do these deposits come from (i.e., what are the sources of slack time)? There are two sources of slack time: 1) Budgeted CPU time that goes unused during a thread's execution; and 2) Unallocated CPU time (i.e., when adding up the total budgeted CPU time, the sum – for the hyper-period – is less than 100%). Thus, at the beginning of each hyper-period, the slack account has a balance equal to the total unallocated CPU time. As threads execute and complete early (with respect to their *worst case* budget) they donate their remaining unused budgeted time to the slack account (i.e., they make implicit deposits). Conversely, as threads execute and wish to use slack time, they make explicit withdrawals. The Deos™ scheduler manages the deposits and withdrawals to ensure time partitioning and system schedule-ability.



## Adding Slack Scheduling to RMS

- By using slack, thread Medium might receive CPU time that would have been lost to Idle to complete its execution for period

  ⚘ Thread consumed budget but is allowed to continue using slack time

  ▮ Thread is executing
  ▮ Thread is ready for execution but has not yet been activated by scheduler
  ▮ Thread has been preempted by higher priority thread
  ▯ Slack time execution
  ⬚ Thread has completed for period
  ↑ Thread completes for period, activate another thread
  ↓ Thread preempted by higher priority thread

How does one use slack? Threads are explicitly identified, at design time[1], as slack requesters. All threads participate in generating slack (i.e., making deposits of time into the account), but only slack requesters are allowed to consume slack (i.e., make withdrawals from the slack account). A slack requester must first use all of its budgeted CPU time[2]. Once a slack requester has depleted all of its

---

[1] This is defined in the Deos Registry (see the "Deos™Integration Tool" Whitepaper).

[2] In Deos, each thread must have *at least* enough fixed budget for one context switch.

budgeted CPU time, it will be given *immediate* access to *all* available slack time (i.e., all of the time in the slack account). It can use all, or a portion, of the available slack time (at its discretion). If the slack requester uses all of the available slack time, and a subsequent thread generates slack (i.e. deposits into the slack account), the slack requester could be scheduled again and given access to the recently generate slack time. Since the Deos™ scheduler always schedules the highest priority thread that is ready-to-run; slack will be consumed by the highest priority slack requester first. In this way, slack is a form of load shedding.

Therefore, with Deos'™ patented slack scheduling technology all available CPU time can be used as threads with *nominal* execution times less than their *worst case* execution times will be *generating* time to be used by slack requesting threads.

One of the most common uses of slack is to remove the lowest criticality applications from the high criticality, fixed budget time line. In other words, run your low criticality applications purely[2] on slack. The Deos™ development environment provides a classic example of this. All of the Ethernet based applications (e.g., the network stack, FTP server, Telnet server, etc.) execute purely[2] on slack; even the network's interrupt service routine. Before slack scheduling, these applications demanded over 50% of budgeted CPU time in order to achieve the customer's expected performance. With the advent of slack, budgeted CPU time for these applications dropped 80% ,with a 300% increase in performance; definitely a win/win.

Deos'™ patented slack scheduling technology provides another key advantage particularly useful in the client-server arena, namely: the ability for threads to execute multiple times within the *same period*[3]. This nuance of slack scheduling allows a client thread and its server thread to exchange data, perhaps multiple times, back-to-back, within the *same period*, in order to complete a transaction. By contrast, in time partitioning schemes described at the top of this paper, clients must wait for their server thread to be scheduled. When a transaction takes multiple interactions, the delay can be significant. To reduce the delay, non-Deos™ users are forced to 'play scheduler', in order to craft a hyper-period timeline that balances the needs of time critical applications with (perhaps multiple) less time critical client-server application's performance needs. However, Deos™ users can use slack and let the Deos™ scheduler take care of this balance for them.

Another common use of slack is the ability to budget in order to meet your safety requirement, but enable slack in order to get the most out of your processor. For example, let's say you have a safety requirement that indicates your display should update at least ten times a second. While this update rate is deemed safe, it falls short of meeting customer expectations. In this case, you could chose to budget for your 10hz update rate (i.e., ensure you have enough fixed budgeted CPU time to meet your 10hz rate) while enabling slack in order to meet your customer delight requirement. Thus, instead of your display performing in worst case mode all the time, it is guaranteed to meet its minimum safety requirements, but will perform in its best case mode as often as possible (i.e., you'll get all the performance your hardware has to offer).

---

[3] This capability is also possible when using thread budget transfer (see "Deos'™ Thread Budget Transfer Feature" Whitepaper).

Additionally, slack can allow you to address your software requirements in a way commonly used before time partitioned operating systems.  Namely, by assigning a requirement, which must be accomplished, to a low priority thread and guaranteeing that it is accomplished by monitoring its activity from a high priority thread.   For example, one could meet their continuous built-in test (CBIT) requirement by assigning that activity to a low priority, pure[2] slack thread and then monitoring the CBIT thread's adequate completion rate from the high priority, fixed CPU budget thread.  Once again, the Deos™ scheduler helps spread the CPU load across the timeline (vs. the user/designer having to 'play scheduler') and the high priority, fixed budget thread guarantees that the activity is occurring per the specification.

By giving software designers the capability to factor computations into slack scheduling and/or the high criticality timeline,  Deos'™ patented slack scheduling technology enables software designers to leverage all the power of today's modern processors, without sacrificing the safety of space & time partitioning.

**For Additional Information**