A Survey of Reservation-Based Scheduling

Lindberg, Mikael

2007

Document Version:
Publisher's PDF, also known as Version of record

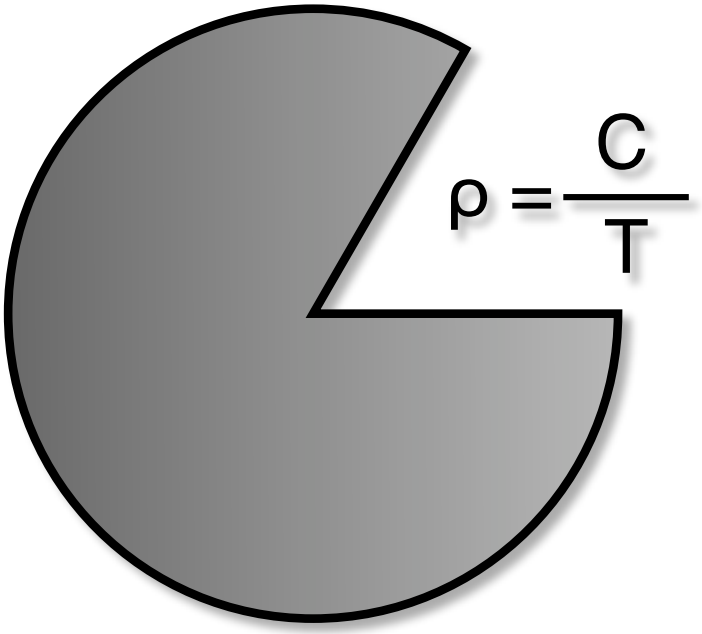
ISSN 0280–5316
ISRN LUTFD2/TFRT--7618--SE

A Survey of Reservation-Based Scheduling

Mikael Lindberg

Department of Automatic Control
Lund Institute of Technology
October 2007

1. Introduction



This survey covers the current state of research regarding Reservation Based Scheduling (RBS). This form of scheduling is used together with a class of real-time applications whose quality of output depend on sufficient access to a resource over time. These are difficult to handle in terms of traditional hard real-time theory. The typical application will be concerned with some type of continuous media task (playback or encoding), and it was in fact the need for support for media applications that perhaps ignited interest in the field. This was in the early 90:ies when computers started to make their way into mainstream media production and consumption. While this remains the favoured use case even today, other forms of computing can also benefit from RBS. We will see that this includes classes which traditionally have been considered hard real-time. Before discussing the different algorithms for RBS we need to examine the specifics of the problem in a bit more detail.

1.1 Time sensitive applications

Consider a video playback application at work with decoding a movie with a frame rate of 25 fps (frames-per-second). In order for the viewing to be pleasant, the frames needs to be computed in time and this could be solved by treating the application as a traditional periodic hard real time task with a deadline T of 1/25 s, but we also need to figure out how long it will take to decode the frame. As can be seen in figure 1.1 and 1.1, the CPU requirements playing the same movie but at different resolutions and encodings can differ a lot and vary a bit during playback as well. In the second movie, the opening sequence has a lot of moving graphics which accounts for the high CPU load during the first 50 seconds. Obviously, using this period for calculating the WCET[1] for decoding a frame will yield a unreasonably conservative result. The price for guaranteeing timely playback

[1] **W**orst **C**ase **E**xecution **T**ime

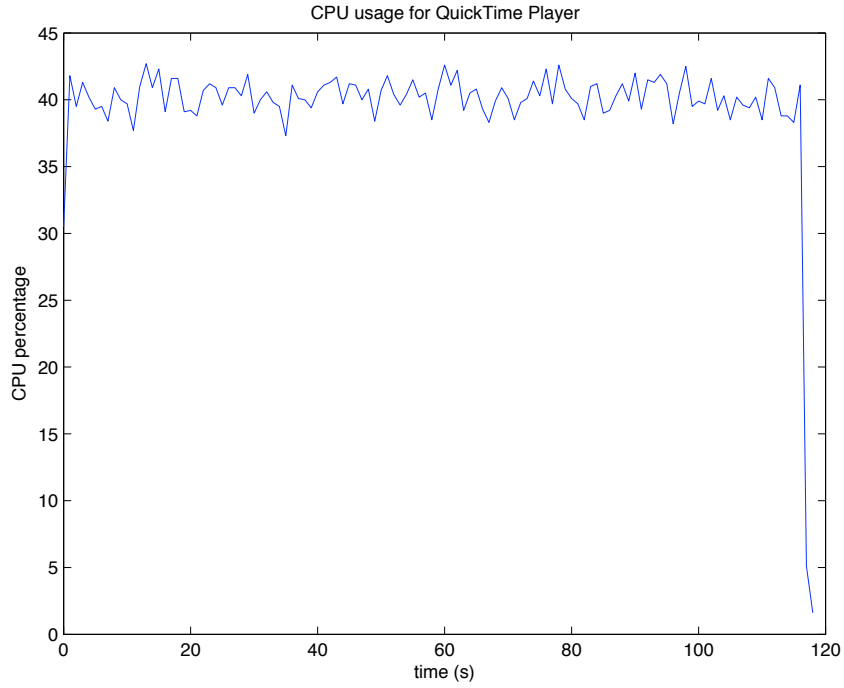


Figure 1 The CPU usage of the QuickTime Player run on a MacBook Pro, playing a short movie sequence in medium resolution.

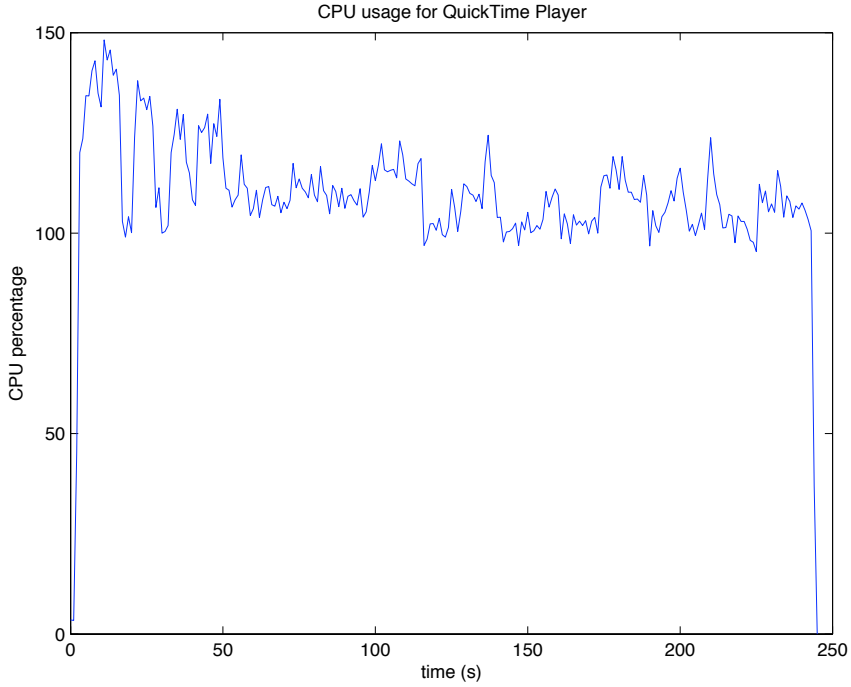


Figure 2 The CPU usage of the QuickTime Player run on a MacBook Pro, playing a short movie sequence in high resolution using VBR encoding. Note that the computer has two CPU cores, hence the load over 100%.

is very high and will utilise the resources in a typical PC very inefficiently. It is perhaps also unnecessary as occasional and slight deadline violations will typically go unnoticed by a human viewer. How well we can adhere to the deadlines will determine the quality of playback, in general called the Quality of Service (QoS). We will call applications which display this connection between timeliness and QoS for *time sensitive applications*.

1.2 Temporal Isolation

A highly desirable property of the RBS approach is that a task who has reserved a specific amount of a resource should have access to this regardless of what other tasks are running on the system. This is called temporal isolation and makes very good sense for the continuous media type of applications we have used as example so far. Video playback should continue unaffected if other applications are started (or if the OS spawns tasks in the background).

1.3 Graceful QoS degradation

While it is entirely possible to create an admission policy which would deny tasks which would make the scheduler unable to sustain reservations this might not be desirable from a user perspective. It might be preferable to have some slight (and predictable) degradation in QoS as opposed to not being able to start more applications. This becomes even more evident in embedded systems where resources are scarce. Consider a mobile phone user engaged in a video conference call when a SMS message comes in. Most would be content to have some slight degradation in video quality and still being able to accept the SMS message. If the playback application in question is designed to be aware of its resource allocation, it can be designed lower QoS in an as graceful way as possible.

1.4 Consumer grade systems and non-real-time applications

Today regular consumer grade computers are used for media purposes which means that a strategy for handling their resource requirements should ideally be implementable on a consumer grade OS and able to handle regular applications as well as time sensitive ones without any special knowledge requirements on the user. Ordinary applications are rarely definable as either periodic or sporadic. Even if they were, knowing the WCET is likely not possible since PC configurations vary greatly (and may even degrade over time). For any solution to the above mentioned problems to be feasible for a consumer product, it should use a minimal set of assumptions on software and hardware environment.

2. Reservation Based Scheduling

2.1 Origins

The case for Reservation Based Scheduling (RBS) was perhaps most famously made by Mercer et al in the 1993 technical report Processor Capacity Reserves for Multimedia Operating Systems [16]. The article discusses processor reserves as a way to describe computational requirements to handle continuous media type applications and some challenges when this is implemented on a microkernel architecture.

The model proposed by the articles says that the needs of a periodic task can be specified using three entities: the processor percentage (or rate) ρ, the computation time C and the period T, related as

$$\rho = C/T \tag{1}$$

In the general case, C is very difficult to compute and the article suggests that the programmer supplies an initial estimate and that the scheduler then measure and adjust the estimate (in fact a feedback scheduling technique). For non-periodic tasks, it is instead proposed to formulate the delay D of a task executing at the rate ρ with total computation time S as

$$D = S/\rho \tag{2}$$

It is easy to derive an admission policy that would only accept new reserves as long as

$$\sum \rho_i \leq \rho_{tot} \tag{3}$$

where ρ_{tot} represents some upper bound of reservable resources which would be less or equal to 1. This simple and intuitive analysis is an argument in favour of RBS.

If the RBS scheme is in turn implemented on top of a regular real-time scheduler such as a RM[2] or EDF[3] scheduler, the admission policy must then also satify

$$\sum_{i=1}^{n} Ci/Ti \leq n(2^{1/n} - 1) \approx 0.69 \tag{4}$$

and

$$\sum_{\forall i} Ci/Ti \leq \rho_{tot} \tag{5}$$

respectively. The choice of underlying scheduling algorithm matters mostly when the system is being overloaded. For the RM case, high priority tasks will likely still meet their deadlines while for the EDF case, the results are unpredictable. This might suggest that RM would be preferable, but considering the significantly more pessimistic bound on utilisation posed by RM, EDF could be made feasible by reserving some $5\% - 10\%$ for handling transient overloads and still schedule more load than RM.

Although [16] is one of the more well cited works, many of the aspects of resource reservations and continuous media had already been discussed in earlier work.

[2] **R**ate-**M**onotonic
[3] **E**arliest-**D**eadline-**F**irst

Ralf Guido Herrtwich presents a number of insights around the problem in his 1991 article The Role of Performance, Scheduling and Resource Reservation in Multimedia Systems [11]. Like in [16], the use of conventional scheduler schemes is deemed as inefficient and perhaps not serving the user needs. Herrtwich also brings up the importance of preventing ill behaved applications from disturbing others (temporal isolation) and that the user might prefer occasional but graceful degradation of QoS to being unable to start new applications when the system comes close to overload. Unlike Mercer however, Herrtwich states that the actual computing needs of the specific applications (in Mercer's terms C) is unlikely to be possible to calculate.

Herrtwitch article quotes heavily from the even earlier article Support for Continuous Media in the DASH System [3], by Anderson et al in 1989, which detailed how media type applications would be served by a resource reservation scheme (in which resources were generalised to include not only CPU but also disk, networking and more) based on preemptive deadline scheduling (it is unclear if they mean EDF). This article is more heavy on theory but lacks some of the softer insights in Herrtwich's work. This article is also referenced by [16].

2.2 Taking a Que from Telecommunication

In what seems like unrelated work, the telecommunications society was around this time researching queuing algorithms which it would turn out share some problem formulations with our process scheduling problem. Demers et al published an article in 1989 on the topic of Analysis and Simulation of a Fair Queuing Algorithm [7] which deals with the problem of splitting a shared resource between users in a fair way so that an ill behaved user wouldn't be able to steal from the others. Although shares are not specified explicitly (in the telecommunications world it is common to assume that the involved parties would want to split all available bandwidth between them), the scheduling algorithm is responsible for making sure all users gets their allocated share.

The article studies a Fair Queuing (FQ) gateway with clients using either FTP type applications (always ready to transmit and who are interested in overall throughput) and Telnet type applications where the loads come irregularly and where average packet delay is the most interesting quantity (packets are assumed to be so small that transmission speed is irrelevant). These two types of transmission cases can be likened to the CPU-bound and I/O-bound scheduling cases respectively. The scheduling algorithm is described as follows:

Consider a hypothetical Round Robin (RR) scheme where $R(t)$ denotes the number of rounds made at time t (partial laps are represented by fractions). A job of size P units arriving at time t_0 will be completed P rounds later, or at time t_1 where $R(t_1) = R(t_0) + P$. The unit of P will be bits for the queuing case and some computational time quanta for the scheduling case. For any task α consisting of the job sequence $j_0^\alpha, j_1^\alpha, ...$ we let t_i^α represent the arrival time of job j_i^α (part of α) and P_i^α, S_i^α and F_i^α the job size and the values of $R(t)$ when the job started and finished respectively. The following relations will then be true.

$$F_i^\alpha = S_i^\alpha + P_i^\alpha \tag{6}$$

$$S_i^\alpha = max(F_{i-1}^\alpha, R(t_i^\alpha)) \tag{7}$$

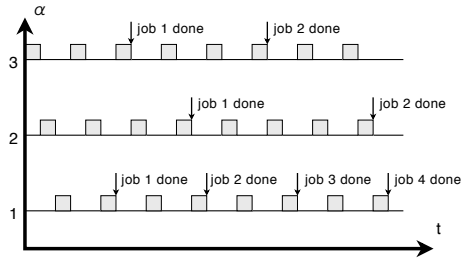


Figure 3 Scheduling diagram over three tasks, where $P_i^1 = 3$, $P_i^2 = 4$ and $P_i^3 = 2$. It's assumed all three jobs are released simultaneously, i.e. $t_0^1 = t_0^2 = t_0^3$

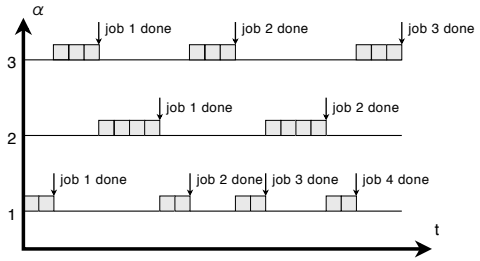


Figure 4 Same tasks as in Figure 2.2 but in "earliest finish first" order

It follows from that $R(t)$ is a strictly monotonically increasing function that the sequence $F_0^\alpha, F_1^\alpha, ..., F_i^\alpha$ share the same order as the finishing times of the jobs $0, 1, ..., i$ of task α. At this point, all entities are explicitly determinable when a job arrive (which is albeit more realistic in the case of a network packet than a task execution), which means that we can use them all when we formulate our scheduling principle. Figure 2.2 is a scheduling diagram displying how three tasks 1, 2 and 3 are interleaved. This scheme is obviously fair in the sense that no task gets more processing than any other and that a task can't misbehave. However, our current scheme where we interleave all tasks between every unit of processing is likely to be too wasteful if actual task switching costs are taken into account. One way to achieve the same order of completion of the tasks is to process them in order of F_i^α. In case of a tie, we pick at random. The execution would then look as shown in figure 2.2.

In both cases, the CPU shares for all three tasks with asymptotically go towards 1/3. A slight variation on the algorithm, called Weighted Fair Queuing (WFQ), introduces weights so that the shares can be scaled if one task is more important than the other. By recalculating this weight in case the set of active tasks changes, a task can be given a constant share of CPU. FQ scheduling was originally proposed by Nagle in his 1988 article "On Packet Switches with Infinite Storage" [17] and has since been a very popular algorithm to study.

The scheme shows a lot of similarities with EDF scheduling. If the above problem instead was formulated as three periodic tasks with (c, T, d) equal to $(3, 9, 9)$, $(4, 12, 12)$ and $(2, 6, 6)$ we would get the same behaviour. We will also later see how the FQ scheme can be altered to account for more cases.

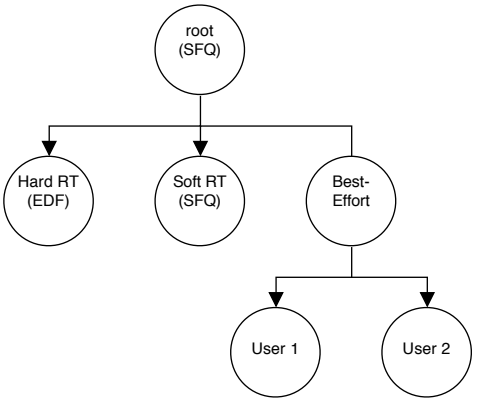


Figure 5 Hierarchical structure with schedulers. Note that SFQ is used on more than one level.

2.3 Hierarchical Scheduling Structures

One important aspect of this problem is that we want to be able to mix real-time applications with regular applications. Often this leads to a construction with a hierarchy of schedulers, typically with some hard-real time scheduler on top and soft real-time and regular best-effort schedulers underneath. Pawan Goyal et al suggests one such approach in their paper "A Hierarchical CPU Scheduler for Multimedia Operating Systems" [10]. They propose a tree structure where each node is either a scheduler node or a leaf node. Parents scheduler their children down to leaf level where the regular tasks live. An example is provided in figure 2.3. They also introduce a variant of WFQ called Start-time Fair Queuing (SFQ). SFQ schedules on increasing start tag instead of finish tag and provides better guarantee of fairness if the amount of available processing power fluctuates over time (see 2.8 for a more details). It provides two ways to model the fluctuations and tools for computing throughput and delay guarantees.

The hierarchical approach to scheduling is also proposed by other groups. The RTAI/Xenomai extensions to Linux runs a RT-scheduler as root and then the Linux operating system as a thread. The structure is similar to the one proposed by [10]. The Bandwidth Server class of RBS algorithms also use a hierarchy of schedulers, typically using an EDF scheduler on top. Hard real-time tasks are scheduled directly by the EDF algorithm, while soft real-time tasks have dedicated "servers" who dynamically set their deadlines to achieve CPU reservations. Ordinary applications can be scheduled by a separate server. Lipari et al presents a hierarchical Constant Bandwidth Server construct called the H-CBS in [15], but this wasn't until in 2001.

Supervisory Schedulers The abilities of the top level schedulers become the most important in the case of insufficient resources, e.g. when a new application is started when the system is already on its limits. It might not be acceptable to just deny the new application (especially if it cannot be considered less important than any other) so the system needs some way gracefully handling the situation. The same is true if the computational needs of an application suddenly changes or if they are unknown.

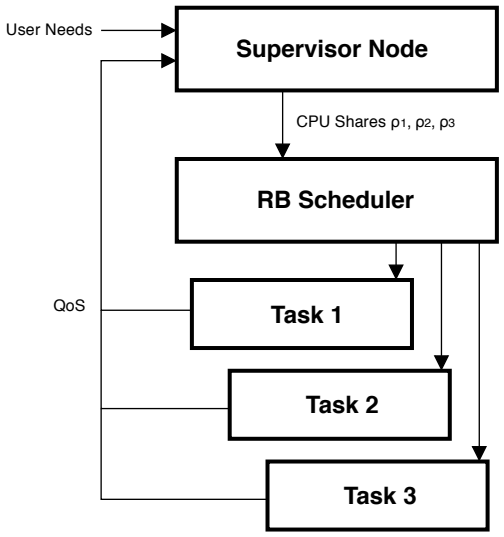


Figure 6 Structure for a supervisory scheduling scheme.

A feedback approach seems pertinent here as the situation is not unlike the typical cascading controller set-up. The high level scheduler determines the shares for each lower level scheduler so that over all system performance is maximized. Of course, this becomes much simpler if the leaf level processes can adapt to changing resource availability (such as adjusting bit rates, frame rates or calculation precision). See figure 2.3 for a diagram of the principle.

2.4 Bandwidth Servers

The concept of bandwidth servers was derived from Dynamic Priority Servers by Buttazzo. Dynamic Priority Servers was a way to handle aperiodic or sporadic tasks in fixed priority systems, basically by introducing a hierarchy of schedulers where the highest level scheduler is a RM scheduler or similar. The Priority Server is a periodic task with a specified execution time. Arriving aperiodic tasks are placed in a queue and executed by the Priority Server when it is scheduled to run. In the original formulation, unused capacity is just lost.

Buttazzo brought the concept of a server presiding over a predetermined amount of CPU capacity to the dynamic scheduling algorithms. The Dynamic Priority Exchange Server (DPE) and the Total Bandwidth Server (TBS) were the first formulations using EDF as a root level scheduler with some predetermined CPU share. The object was still to handle aperiodic tasks, so a lot of theory concerned handling of unused bandwidth. In 1998, Buttazzo and Abeni published the article "Integrating Multimedia Applications in Hard Real-Time Systems" [15] which introduces the Constant Bandwidth Server (CBS). By then, the CM problem had already been addressed using the Bandwidth Server metaphor by Kaneko et al in the article "Integrated Scheduling of Multimedia and Hard Real-Time Tasks" from 1996 [13].

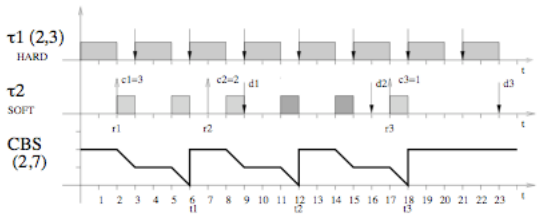


Figure 7 An example of CBS scheduling. The figure is taken from Abeni and Buttazzo's article Integrating Multimedia Applications in Hard Real-Time Systems

2.5 Constant Bandwidth Server

We will now introduce the terminology from the article and it will become evident that it shares structure with the Fair Queuing techniques.

Consider a set of tasks τ_i where a task consists of a sequence of jobs $J_{i,j}$ with arrival time $r_{i,j}$, C_i the WCET of any job (AET[4] for soft real-time tasks) in the sequence and T_i the minimum arrival interval between jobs (desired activation period for soft real-time). We will be focusing on the soft real-time tasks since we consider CM type applications. For any job, we assign a deadline $d_{i,j} = r_{i,j+T_i}$.

A CBS for the task τ_i can then be defined as:

- A budget c_s and by a pair (Q_s, T_s) where Q_s is the maximum budget and T_s is the period. The ratio $U_s = Q_s/T_s$ is called the server bandwidth. At each instant, a fixed deadline $d_{s,k}$ is assigned with the server with $d_{s,0} = 0$.
- The deadline $d_{i,j}$ of $J_{i,j}$ is set to the current server deadline $d_{s,k}$. If the server deadline is recalculated, then so is the job deadline.
- When a job associated with the server executes, c_s is decreased by the same amount.
- When $c_s = 0$ the budget is replenished to the value of Q_s and the deadline is recalculated as $d_{s,k+1} = d_{s,k} + T_s$. This happens immediately when the budget is depleated, the budget cannot be said to be 0 for any finite duration.
- Should $J_{i,j+1}$ arrive before $J_{i,j}$ is finished, it will be put in a FIFO queue.

In figure 2.5 we can see two tasks τ_1 and τ_2, where τ_2 is being scheduled by a CBS.

CBS^{hd} One possible drawback with the CBS algorithm when dealing with things sensitive to deadline overrun is that although the server is completely replenished when budget c_s is exhausted, the new deadline might be too far into the future. The CBS^{hd} algorithm changes the replenishment rule to better handle this. If $c^r_{i,j}$ is the remaining computational need for

[4] **A**verage **E**xecution **T**ime

$J_{i,j}$ when the budget is exhausted, we apply the following replenishment rule:

$$if(c^r_{i,j} \geq Q_s) \tag{8}$$
$$c_s = Q_s; \tag{9}$$
$$d_{s,k+1} = d_{s,k} + T_s; \tag{10}$$
$$else \tag{11}$$
$$c_s = c^r_{i,j}; \tag{12}$$
$$d_{s,k+1} = d_{s,k} + c^r_{i,j}/U_s \tag{13}$$

This means that is the overrun is less than the budget, the new deadline will be calculated less pessimistically. This is investigated in the article Elastic Feedback Control by Buttazzo et al.

The Control Server (CS) Cervin and Eker presented in their 2003 article "The Control Server, A Computational Model for Real-Time Control Tasks" [5] a modification to the CBS scheme that would make it easier to handle the timing needs of a control application. Though control tasks are typically implemented using hard real-time scheduling, the inherent robustness in feedback control schemes can make hard real-time guarantees unnecessarily expensive. The CS makes the following change to the CBS setup:

- Each task τ_i is associated with a set of $n^i \geq 1$ segments $S_{i,1}, S_{i,2}, ..., S_{i,n_i}$ of lengths $l_{i,1}, l_{i,2}, ..., l_{i,n_i}$ such that $\sum_{k=0}^{n_i} l_{i,k} = T_i$
- τ_i has a set of inputs I_i
- τ_i has a set of outputs O_i
- Each set $S_{i,k}$ is associated with a code function $f_{i,k}$, a subset of the inputs $I_{i,k} \in I_i$ and a subset of the outputs $O_{i,k} \in O_i$
- The server has a segment counter m_s

The algorithm is also changed in the following way:

- Server is initiated $c_s = m_s = 0$
- When $c_s = 0$ then
  - $m_s := mod(m_s, n_i) + 1$
  - $d_s := d_s + l_{i,m_s}$ and
  - $c_i = U_s l_{i,m_s}$

The result of this change is that the server budget c_s is spread out over a number of smaller segments, reducing the uncertainty as to when a certain input will be read, a certain output be set or a code function executed. A trade of will have to be done between jitter and latency. If the segment lengths $l_{i,k}$ are set to the WCETs of the code functions $f_{i,k}$, the jitter will be 0 but control performance will be affected by greater latency. By forming a cost function including both jitter and latency, an optima may be calculated.

In figures 2.5 and 2.5 we compare the performance of regular EDF scheduling and using the CS. The set up consists of two PID controllers in a cascade configuration and a disturbance task. The execution trace of the controller and the disturbance task is visible in the lower half of the two figures. The figures are borrowed from the [5]

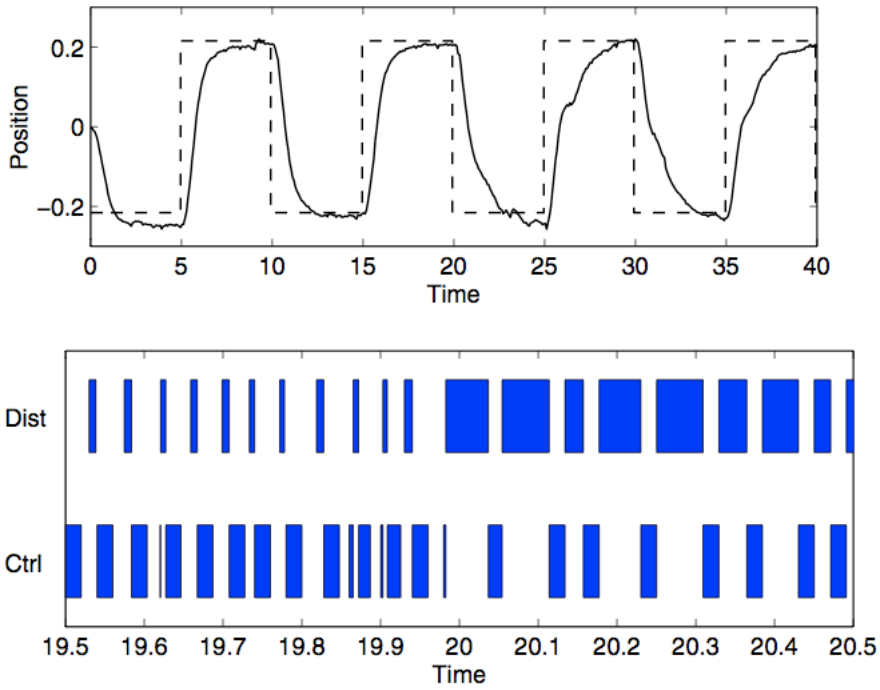


Figure 8 PID control implemented on EDF scheduler

The Atropos Scheduler The Xen virtualisation software can use a scheduler called the Atropos Scheduler to devide the CPU of the host computer between the virtualisation instances. This scheduler is a CBS like construct with the ability to reclaim slack CPU time, running on top of an EDF scheduler. See 3.3 for more on the Xen software.

Adaptive Reservations aka Feedback Scheduling One problem when doing RB scheduling is that the execution time for a periodic task may vary over time. As we don't want to base our calculations on the worst case, we are likely to miss a few deadlines. While the CBS scheme can handle transient overruns, non transient changes will lead to eventually infinite deadlines (instability). One way to remedy this would be to dynamically set the budget for a server based on prior overrun statistics in a feedback control manner.

In the article Adaptive Bandwidth Reservation for Multimedia Computing, Abeni and Buttazzo [1] introduces a metric call the scheduling error. If we have a periodic task τ_i whose with period is T_i, then the scheduling error ϵ_s is defined as

$$\epsilon_s = d_s - (r_{i,j} + T_i) \tag{14}$$

that is the difference between the server deadline and the task's soft deadline. We can now apply feedback control on the scheduling error, using e.g. Q_s as our control signal and try to drive ϵ_s towards 0.

In the article Adaptive Reservations in a Linux Environment [6], Lipari et al investigates a few design techniques for such a controller. For the purpose of making the analysis simpler, we impose the restriction that even if there is extra unused bandwidth available, a task τ_i scheduled by a CBS will only receive the bandwidth Q_s (a so called *hard reservation*).

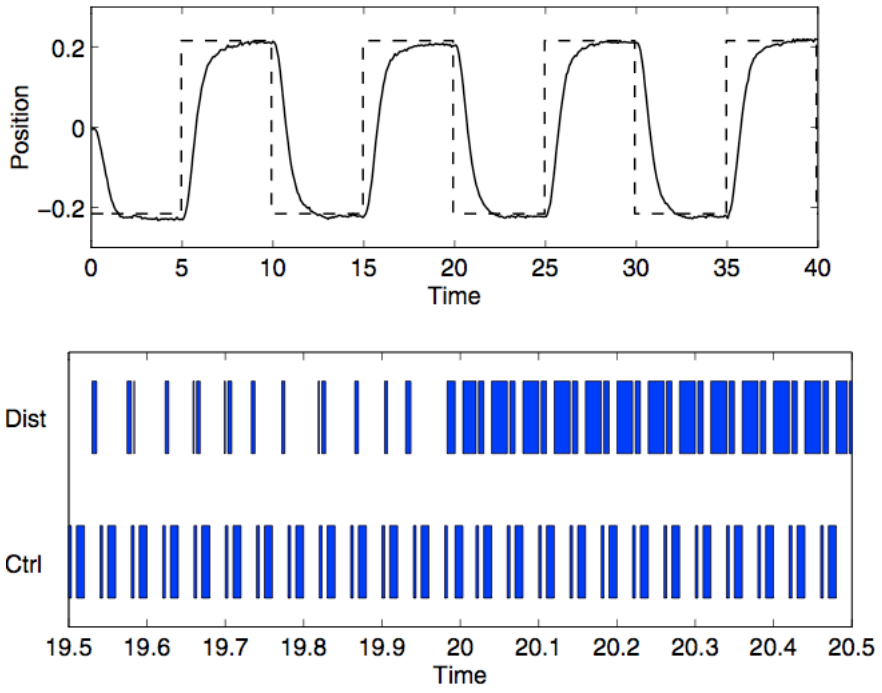


Figure 9 PID control implemented on CS

Assume that τ_i is a periodic task being served with a CBS. This gives $r_{i,j+1} = r_{i,j} + T_i$, where T_i is the task period. Each job is associated with a soft deadline $d_{i,j} = r_{i,j} + T_i$, that is $d_{i,j} = r_{i,j+1}$. It makes sense to choose the server period T_s to be some multiple of T_i. Let $f_{i,j}$ be the actual finish time for $J_{i,j}$ and $v_{i,j}$ be the finish time had τ_i been running alone on a CPU with the fraction $b_i = Q_s/T_s$ of the actual CPU speed. $v_{i,j}$ is called the *virtual finish time* of $J_{i,j}$. The article uses a modified definition of the scheduling error compared to 14

$$\epsilon_{i,j} = (f_{i,j-1} - d_{i,j-1})/T_i \tag{15}$$

which is the scheduling error experienced for $J_{i,j-1}$. Note that since we are using hard reservations, having both $\epsilon_j > 0$ and $\epsilon_j < 0$ is undesirable situations since we'd be either missing our deadlines or wasting bandwidth. The relation

$$v_{i,j} - \delta \leq f_{i,j} \leq v_{i,j} + \delta \tag{16}$$

where $\delta = (1 - b_i)T_s$. This tells us that we can make the CBS approximate the General Processor Sharing (GPS) algorithm by letting T_s go towards 0, but this is impractical since the context switching overhead starts to become relevant if T_s is sufficiently small. Even with normal choices of T_s it is reasonable to use 15 and 16 to approximate the scheduling error with

$$e_{i,j} = (v_{i,j-1} - d_{i,j-1})/T_i \tag{17}$$

In the article "Analysis of a Reservation-Based Feedback Scheduler" [2] Abeni et al derrives a difference equation for the evolution of the scheduler error:

$$'e_{i,j+1} = max(e_{i,j}, 0) + c_{i,j}/Q_s - T_i; \tag{18}$$

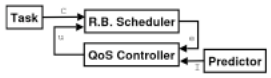


Figure 10 Structure for stochastic control of server bandwidth. The figure is taken from [6]

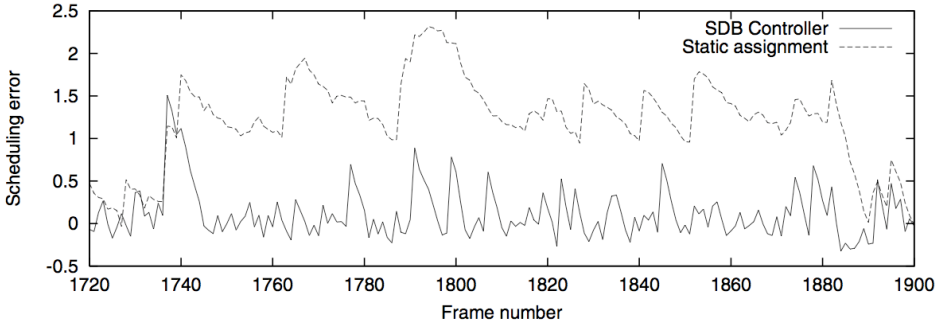


Figure 11 Results from using the adaptive CBS scheme on the Xine player. The feedback controller is in this case a stochastic deadbeat controller. The figure is borrowed from [6]

where we model the sequence $c_{i,j}$ as a discrete-time continuous valued stochastic process. The article then proposes a predictor based control structure (see figure 2.5) and three examples of control design using invariant based design, stochastic dead bead design and optimal cost design respectively.

Some experiments run on the algorithm implemented in Linux gives the results shown in figure 2.5.

Slack Reclaiming (aka Slack Stealing) When scheduling soft real-time or non real-time tasks together with hard real-time tasks or other soft real-time tasks using RBS, inevitably there will be some unused bandwidth. Hard real-time typically uses WCET values which will rarely be met and soft real-time tasks can also underuse resources. Ideally, this "extra" bandwidth should be used to improve performance for other soft real-time tasks or non real-time tasks. A complication is that dynamic slack can only be detected in run-time and can therefore also only be allocated in run-time. Similarly, overrun situations can in general not be predicted beforehand. The traditional way of handling slack was to schedule all real-time tasks first and then allow the slack consuming tasks to run on whatever remains. The CBS algorithm itself has some manner of slack reclaiming in that if the current job $J_{i,j}$ terminates before c_s is spent, $J_{i,j+1}$ can begin immediately to execute, making use of that budget. However, if the task hasn't finished before the remaining c_s has been spent, it will be given a new deadline based on the server period T_s, meaning that it can be forced to execute over an even longer period of time. Additionally, should the task overrun a lot, the deadline might be postponed so far that it will not execute anything for several server periods, possibly leading to starvation.

Numerous methods have been proposed to solve this and other slack reclaiming problems. The inventors of the CBS algorithm has published several themselves, including CASH, GRUB and IRIS. Lin and Brandt

proposed another handful in the article "Improving Soft Real-Time Performance Through Better Slack Reclaiming" [14]. Most of these algorithms explore different methods to determine whom to give unused slack to and how. Their most advanced proposal, the BACKSLASH algorithm uses the following four principles to determine who will get what from whom:

- Allocate slack as early as possible and with the priority of the donating task. This means that the scheduler shouldn't wait until the completion of all real-time tasks before it allocates slack. By executing the slack at the same priority as the donator, there is no risk that it would disturb the execution of tasks that wouldn't have been disturbed by that same donator.
- Allocate slack to the task with the highest original priority (earliest original deadline). Basically this means give the slack to the task in most dire need. See the principle below for a rationale for using the original priority/deadline.
- Allow tasks to borrow against their own future resource reservations (with the priority of the job from which resources are borrowed) to complete their current job. This is the standard deadline postponing from the CBS algorithm.
- Retroactively allocate slack to tasks that have borrowed from their current budget to complete a previous job.

Figure 2.5 shows the performance of SRAND, SLAD, SLASH and BACKSLASH (which cumulatively uses the four principles above) compared to CBS, CASH and a primitive algorithm "EDF" which allocates slack to the task with earliest deadline when all other tasks are idle.

2.6 Rate Based Execution (RBE)

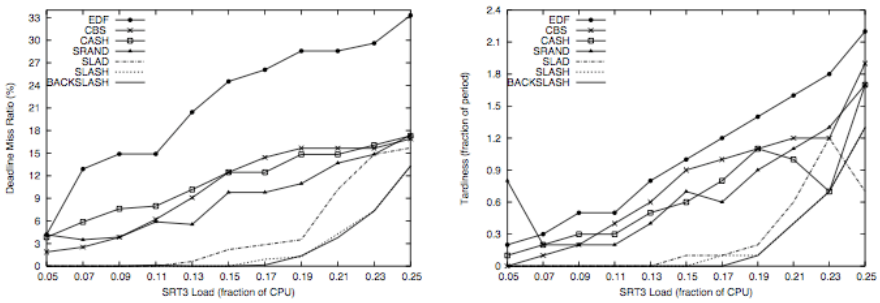
Rate Based Execution was originally proposed by Jeffay and Goddard in their 1999 article "A Theory of Rate-Based Execution" [12]. It is presented as a generalisation of the sporadic task model by Mok [4] and is essentially another scheme for setting the deadlines for the jobs $J_{i,j}$ released by a task τ_i, aiming in this case to limit the number of jobs the task can schedule during some interval. If the WCET of the jobs is known, it is also possible to compute a worst case bound for how much CPU the task will use within a specified time interval. This is slightly different from the CBS algorithm which limits the bandwidth directly.

Formally, a RBE task τ_i is defined by a four-tuple $(y_i, x_i, \Delta d_i, c_i)$.

- y_i is a time interval
- x_i is the maximum number of jobs expected to be released during an interval y_i
- Δd_i is the relative deadline for any job $J_{i,j}$ belonging to τ_i
- c_i is the WCET for any job $J_{i,j}$ belonging to τ_i
- $r_{i,j}$ is the release time for the $J_{i,j}$

The pair (x_i, y_i) is called the "rate specification" of the RBE task. When $J_{i,j}$ is released, it is given an absolute deadline $d_{i,j}$ given by

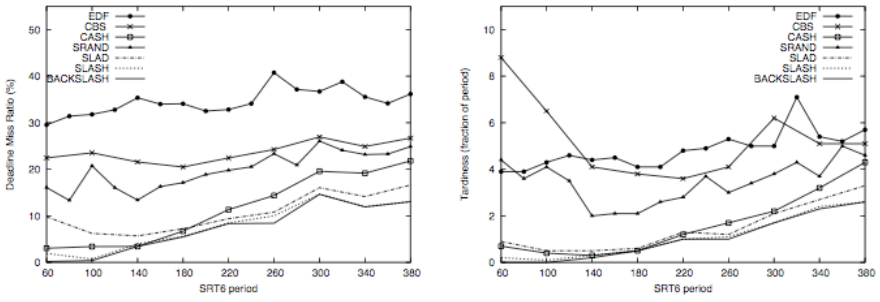
$$d_{i,j} = \begin{cases} r_{i,j} + \Delta d_i & 1 \leq j \leq x_i \\ max(r_{i,j} + \Delta d_i, d_{i,j-x_i} + y_i) & j > x_i \end{cases} \tag{19}$$



(a) Deadline Miss Ratio as a function of load

(b) Tardiness as a function of load

Figure 5. Load effect on performance (one soft real-time task, $p = 300$ms)



(a) Deadline Miss Ratio as a function of period

(b) Tardiness as a function of period

Figure 12 Performance for a number of slack reclaiming options. The figure is taken from [14].

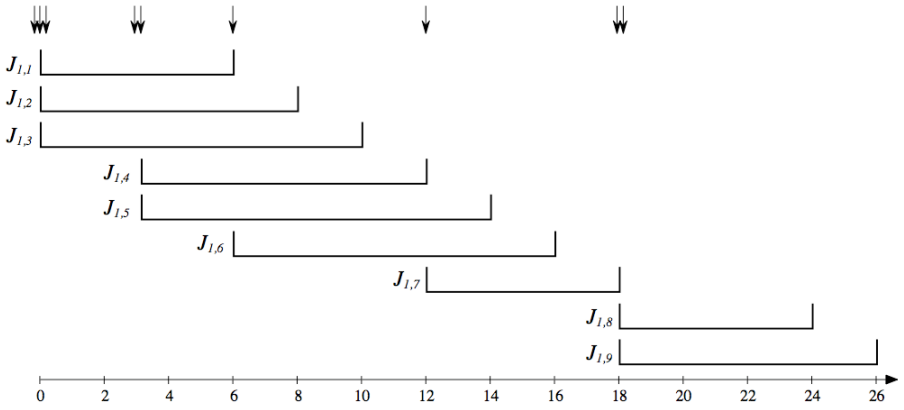


Figure 13 A bursty release scenario allowed by RBE. Borrowed from [12]. We can see how the deadlines are spaced out with a minimum distance, even when a burst of jobs are released simultaneously for a RBE task ($x = 1, y = 2, d = 6, c$)

Setting the deadlines this way guarantees that between two jobs $J_{i,j}$ and $J_{i,j+x_i}$ are separated by at least y_i. Note that this doesn't prevent small clusters of jobs from being released simultaneously and while the RBE is active, instead of queuing them, like CBS does. This can be observed in 2.6.

The article then proves that for any interval $[0, L]$ where $L > 0$, the least upper bound for processor demand from a RBE task τ_i defined by

$(y_i, x_i, \Delta d_i, c_i)$ is

$$f(\frac{L - \Delta d_i + y_i}{y_i})x_i c_i \tag{20}$$

where

$$f(a) = \begin{cases} \lfloor a \rfloor & a > 0 \\ 0 & a \leq 0 \end{cases} \tag{21}$$

Under pre-emptive scheduling, a set of RBE tasks $1..n$ will then be schedulable if and only if

$$L \geq \sum_{i=0}^{n} f(\frac{L - \Delta d_i + y_i}{y_i})x_i c_i \tag{22}$$

The article also provides feasability conditions for non pre-emptive scheduling and for pre-emptive scheduling but with shared resources.

2.7 Proportional Share Scheduling (PS)

Perhaps one of the oldest time-sharing algorithms for any type of resource is the Round-Robin scheme. By adding a weight to the participants in the ring so that they each round get to spend a time proportional to that weight with the resource, you get Weighted-Round-Robin (WRR). Such a division of a resource between participants is called Proportional Share resource management and is a large class of scheduling algorithms. As mentioned before, FQ is another way to achieve PS, but the class of algorithms derived from FQ is so big it has been given its own section.

There are several reasons as to why WRR isn't as useful for RBS purposes as it might initially seem. While it might work for a general purpose CPU bound application (which is always ready to run and which never blocks), periodic tasks where computation time fluctuate will not be served well by the WRR scheme. There are a few reasons to this:

- WRR doesn't handle overrun, it will preempt a task when its allocated time slice is spent. This means that you need to use WCET to figure the size of the reservation, which in turn will lead to a lot of waste.
- WRR doesn't have any scheme for slack reclaiming. Slack can be consumed by non-RT applications, but other participants in the RR scheme will not benefit.
- For tasks that block, the WRR scheme will be almost impossible. The typical I/O-bound task will be blocking most of the time. When it wakes up, it will have to wait for its turn in the RR scheme and perhaps not have time to finish its work. It then has to wait another round before it may complete. These types of tasks are generally very latency sensitive, something which is difficult to handle in plain WRR.
- WRR can't handle sporadic real-time tasks without dynamically recalculating the weights (in which case it's a different algorithm).

There are many alterations to WRR to remedy these problems. The FQ family of algorithms explore one path but there are other well known examples.

Lottery Scheduling and Stride Scheduling A somewhat unorthodox proposition made by Waldspurger et al in 1994 [21] involved using a random lottery scheme to select which process would run next. The scheme goes quite far in the lottery metaphor, including not only actual tickets which can be passed around between processes but also ticket currencies and exchange rates. Simply put, each process is awarded a number of tickets which corresponds to its intended proportional share of the scheduled resource. The scheduler then draws one ticket each time quanta to decide who gets to run. Given sufficient many lotteries, the algorithm is probabilistically fair. The probability p that a task τ_i holding m_i tickets in a lottery with in total M tickets will win the lottery is simply $p = m_i/M$. After n identical lotteries, the expectation of the number of wins w_i is $E[w_i] = np$, with variance $\sigma_w^2 = np(1-p)$. The coefficient of variation for the observed proportion of wins is $\sigma_w/E[w_i] = \sqrt{(1-p)/np}$. Thus, the task share is proportional to its ticket allocation, with accuracy that improves with $\sqrt{n}$. From this follows that the expecation of the number of lotteries a task needs to participate in before winning $E[n]$ is $1/p$, making the average response time for a process inversely proportional to its ticket share.

Apart from the above mathematical results, Lottery Scheduling has the following properties:

- Since there is no notion of round or turn, changing the ticket allocation (the weights) has immediate effect.
- Transferring tickets between tasks (usually temporarily) enables a task blocking pending some other task completion to help the other process and indirectly itself in a priority inheritance like way.
- A task that only spends the fraction f of its allocated resource can be granted a compensation ticket that inflates its value by $1/f$ until the task wins the next time. This way, the task's resource consumption, equal to fp is adjusted to $f\dot{1}/fp = p$ which matches its allocated share. Since we're trusting the law of big numbers, there is no need for more complicated accounting.
- By dynamically issuing tickets, a task can get escalated rights without the need for ticket exchanges. This technique should be used only in trusted systems as allowing this freely would be like letting people print their own money.

Implementation wise, Lottery Scheduling is dependant on a good random number generator and a fast way to find the owner of each ticket (including considering ticket transfers). For small systems, using a simple linear search with $O(n)$ complexity might be good enough while for larger systems, such as regular desktop OSes with hundreds of tasks, more efficient data structures must be employed. The article also discusses the used of controlled ticket inflation to govern the share allocated to a task based on some measurable output from the task.

Figure 2.7 shows an example from [21] where three MPEG-decoders are initially given a A:B:C = 3:2:1 ratio of tickets. At the time indicated with the arrow, the ticket ratios are changed to 3:1:2.

In the follow up article from 1995, the Stride Scheduling algorithm is introduced [20]. This is in essence a deterministic sibling to the Lottery Scheduling algorithm, motivated by improved accuracy over relative

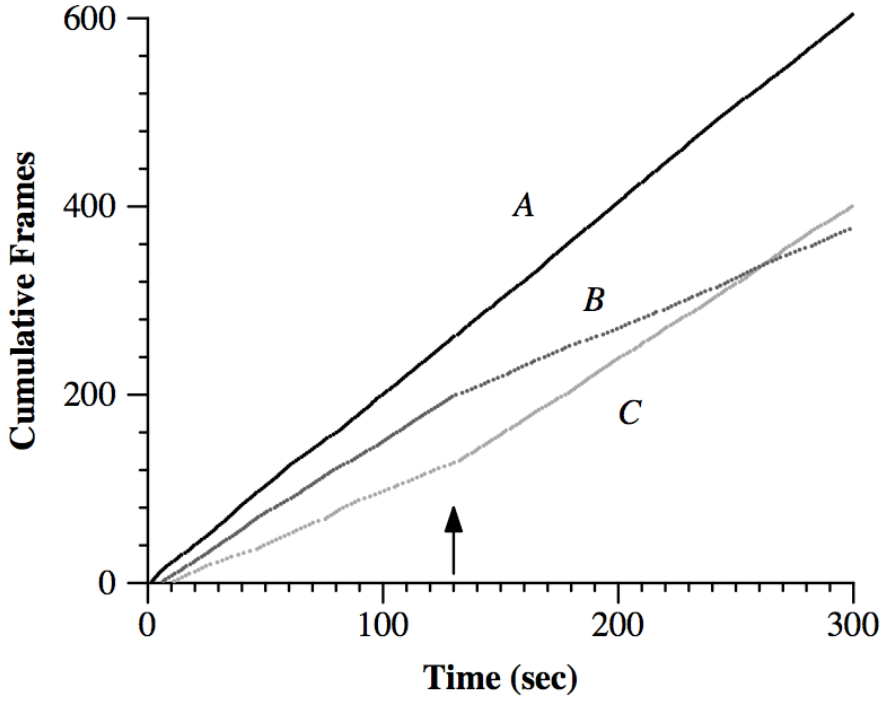


Figure 14 An experiment from [21] with the three tasks A:B:C.

throughput rates and with lower response time variability. In Stride Scheduling, each task is given a share of tickets equal to the desired proportional share of the scheduled resource. However, instead of using the tickets to perform a lottery, a stride value s_i, calculated as the inverse of the number of tickets, is used. Stride is meant to represent the interval between times the task executes, measured in passes. We will call the current pass value for a task p_i. Passes is a virtual time unit, very similar to the $R(t)$ function used in FQ, which would make stride comparable to a task's P_i-value. The algorithm uses a constant $stride_1$ which represents the stride for a task with one ticket. The algorithm then goes as follows:

- Tasks are initited with $s_i = stride_1/m_i$, $p_i = s_i$
- The scheduler selects the task with lowest p_i (ties are broken arbitrarily). After executing it, p_i is updated to $p_i + fs_i$, where f is the fraction of the allocated time quanta that the task actually used (allowed to be > 0).

As before, ticket transfers and ticket inflation to dynamically change resource allocations, but the cases become more complex now when the stride value of a task can approach infinity (if for example a task transfers all its tickets to another process).

Group-Ratio Round Robin A problem when dealing with proportional share resource management is that algorithms can easily get computationally intensive. We often have data structures which must be sorted or searched, leading to work at least $O(log(n))$ where n is the number of scheduleable tasks. In recent years, a lot of focus has come on the so

called $O(1)$-algorithms. Unfortunately, in order to make the scheduling decision possible to make in constant time, the algorithms itself tend to get a bit messy. A showcase for this is the Group Ratio Round-Robin algorithm (GR3) which is $O(1)$. It was presented by Caprita et al in 2004 and has seen experimental implementations in Linux where it has been compared to the standard Linux $O(1)$-scheduler.

GR3 changes the standard WRR scheme as follows:

- Tasks τ_i are put into groups depending on their weight ϕ_i. A group G is of order $k \geq 0$ is assigned tasks with $\phi \in [2^k, 2^{k+1} - 1]$.
- The group weights $\Phi = \sum_{\{i;\tau_i \in G\}} \phi_i$ are calculated for all groups, which are then put into a list sorted after decreasing Φ. Ties are broken by group order. We let G_j denote the j^{th} group in the list and Φ_j the corresponding group weight.
- Group work W_j is defined as the sum of execution time received by all tasks $\tau_i \in G_j$.

The scheduling algorithm then works as follows (initiate with $j = 1$):

- Run a task from G_j selected in a Deficit WRR fashion. The task weights are normalised with respect to the minimal possible weight for a task in the group (2^k).
- Increment W_j correspondingly.
- if $\frac{W_j+1}{W_{j+1}+1} > \frac{\Phi_j}{\Phi_{j+1}}$ then increment j, else $j = 1$.

The reasons for this somewhat complicated set-up is that the normalised weights in the groups reduce the effects of skewed weight ratios which affect fairness in RR-type algorithms, while still allowing us the $O(1)$-complexity scheduling decision (more or less increment j or set $j = 0$). The analysis can be found in the article but is not trivial.

2.8 Fair Queuing (FQ)

We introduced the FQ family of algorithms in the beginning of this chapter and now we wrap things up by taking a closer look at where things went after Demers et al published their results. Before doing that, we need to formally define the concept of fairness, which the name of the algorithms would suggest is central to the theme.

Fairness Fairness was originally introduced by Nagle in [17]. The semantics were initially informal, saying simply that a fair algorithm divides the resources between peers equally. He also prototyped the FQ algorithm we explored but with little formalism. The first analysis of the FQ algorithm was done by Demers in [7], where an algorithm for dividing a resource is defined as fair if

- no used receives more than its request,
- no other allocation scheme satisfying the first condition has a higher minimum allocation and
- the second condition remains recursively true as we remove the minimal user and reduce the total resource accordingly

For our type of applications, the conditions can be expressed in another way. Assume that we have a finite resource D and n users of that resource. Each user "deserves" a fair share equal to D/n of this resource, but is allowed to ask for less, in which case the difference can be allocated to a user who would like more. Let d_i denote the share a user requests and a_i the share he is given. We now define the maximally fair allocation such that the fair share d_f is computed subject to the following two constraints:

$$\sum_{i=1}^{n} a_i = D \tag{23}$$

$$a_i = min(d_i, d_f) \tag{24}$$

From this we can determine the maximally fair allocation $\{A_1^*, A_2^*, ...\}$. To classify an actual allocation $\{a_1, a_2, ...\}$ we use a fairness function

$$Fairness = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2} \tag{25}$$

where $x_i = a_i/A_i^*$. The fairness will be between 0 and 1, where 1 represents a maximally fair allocation.

Using this metric, we can discuss how fair an algorithm is, how quickly it achieves it and how sensitive it is to fluctuating conditions. More notions of fairness does however exist. The formulation above is limited to calculating the over all fairness, but is difficult to apply to specified time intervals. For that, we need a more advanced formulation. In the article "A Self-Clocked Fair Queueing Scheme for Broadband Applications" Golestani introduces a notion of fairness which is based on the concept of normalised service. Let r_i be the service share allocated to a task τ_i and $W_i(t)$ the aggregate amount of service this task has received in the interval $[0, t)$. The normalised service is then $w_i(t) = \frac{1}{r_i} W_i(t)$. An algorithm is then considered fair in an interval $[t_1, t_2]$ if

$$w_i(t_2) - w_i(t_1) = w_j(t_2) - w_j(t_1) \tag{26}$$

for any two tasks τ_i and τ_j who has enough work to execute during the entire interval and fair if this is true for any interval. Eqation 26 can be written more compactly as

$$\Delta w_i(t_1, t_2) - \Delta w_j(t_1, t_2) = 0 \tag{27}$$

As the semantics allow us to choose $t_2 - t_1$ arbitrarily small, this is only possible to obtain if work is infinitely divisible and all tasks can be serviced simultaneously. This theoretical form of algorithm is modelled in the article from 1993 by Parekh and called the Generalized Processor Sharing algorithm. Note that GPS would be completely fair given both definitions. Golestani also presents a theorem for the upper bound of (un)fairness for an algorithm based of the FQ family. Given any time period $[t_1, t_2]$ and any two tasks w_i and w_j who wants to execute in the entire interval, if the following bound holds

$$|\Delta w_i(t_1, t_2) - w_j(t_1, t_2)| \leq D(t_1, t_2) \tag{28}$$

where $D(i,j)$ does not depend on $[t_1, t_2]$, then $D(i,j)$ always satisfies the following

$$D(i,j) \geq \frac{1}{2}(\frac{WCET_i}{r_i} + \frac{WCET_j}{r_j}) \tag{29}$$

Self-Clocked Fair Queuing (SCFQ) One of the drawbacks with the basic (W)FQ presented earlier is that you need to do simulations of each task's future stop tags (a function of the virtual time $v(t)$ experienced by a task) in order to decide which job goes next. Given a large number of tasks, this can get computationally intensive and infeasible for situations with a lot of context switches. In the example we used in the beginning of this chapter, all tasks were CPU bound which mean in FQ terms that they were continuously backlogged, which simplified the calculations. As mentioned before, Self-Clocked FQ was introduced by Golestani in 1993 [9], where he proposes to approximate $v(t)$ with the stop tag F_i^α of the job currently in service. He proves that this leads to a fairness of

$$|\Delta w_i(t_1, t_2) - \Delta w_j(t_1, t_2)| \leq \frac{1}{r_i} WCET_i + \frac{1}{r_j} WCET_j \tag{30}$$

which is a factor 2 worse than the optimal case.

Start-time Fair Queuing (SFQ) [10] discusses another drawback of the WFQ scheme in their article on Start-time Fair Queuing. WFQ is sensitive to fluctuations in available bandwidth and they show that WFQ can be unfair regardless if the bandwidth increases or decreases. SFQ is proven to have the same fairness bound as SCFQ and is also proven to handle fluctuating server capacities by making assumptions on the disturbances. The article introduces two disturbance models, the Fluctuation Constrained (FC) server and the Exponentially Bounded Fluctuation Server (EBF).

The FC server has two parameters, C and $\delta(C)$, which correspond to the computational capacity of the server (per second) and the worst case work loss the server can experience during one busy period. More formally, if $W(t_1, t_2)$ denotes the amount of work done in such an interval, then

$$W(t_1, t_2) \leq C(t_2 - t_1) - \delta(C) \tag{31}$$

The EBF formulation makes a stochastic relaxation of FC and is defined by C and δ as before, but also the parameters B, α and γ, such that

$$P(W(t_1, t_2) < C(t_2 - t_1) - \delta(C) - \gamma) \leq Be^{-\alpha\gamma} \tag{32}$$

$$\gamma \geq 0 \tag{33}$$

The results are also proven by experimental comparisons between WFQ and SFQ. SFQ is also proven to have much lower maximum delay than SCFQ and since it shares the fairness bound, SFQ is said to be strictly better than SCFQ.

Borrowed Virtual Time scheduling (BVT) Duda et al makes another change to the WFQ scheme in their article from 1999 on the Borrowed-Virtual Time algorithm [8]. When using FQ for scheduling computer processes, interactivity becomes an issue and the BVT aims to handle just that.

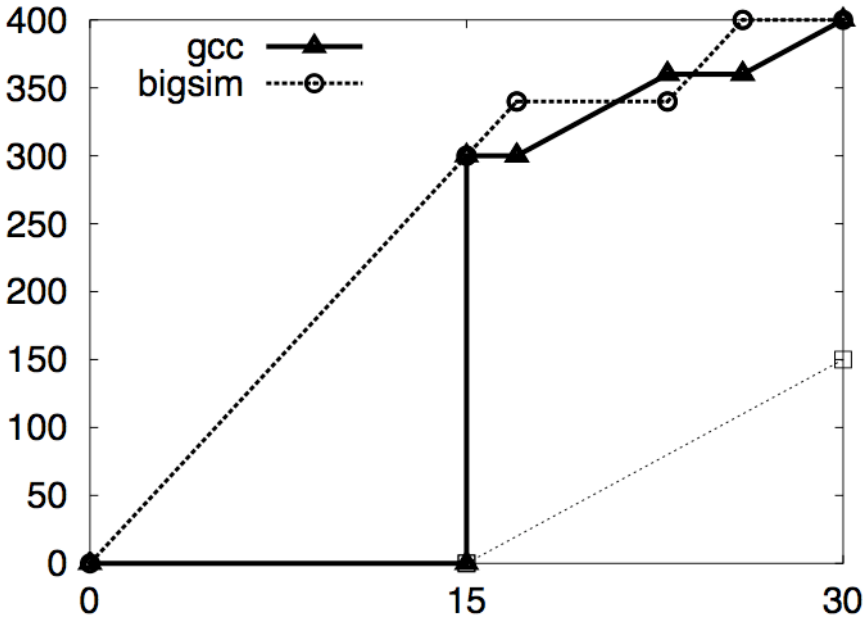


Figure 15 Graph borrowed from [8]. The lower dotted line shows gcc's AVT if it hadn't been adjusted. It would then have starved bigsim for along time before catching up. Y-axis shows AVT.

BVT distinguishes between virtual time, effective virtual time (EVT) and actual virtual time (AVT). The AVT A_i of a task is increased when it by its running time divided by the weight r_i. It also recognises context switch costs by introducing a switching allowance C and switches to task j when

$$A_j \leq A_i - \frac{C}{r_i} \tag{34}$$

From this it follows that the maximum deviation from fair CPU allowance equals $C + m$, where m is the minimal time unit used in the system.

$$A_j \leq A_i - \frac{C}{r_i} \tag{35}$$

The system handles sleepers by setting

$$A_i := max(A_i, SVT) \tag{36}$$

when the thread wakes up, where SVT (the Server Virtual Time) is the AVT for any non-blocking task. This way, a daemon type task which spends most of its time sleeping can't monopolise the CPU forever once it wakes up. Newly created tasks can be initiated in the same way. See figure 2.8 for an example of this.

Tasks which are interactive can be allowed to "warp" in time. They have the additional parameters $warp$, W_i, L_i and U_i, which here denotes if it is allowed to warp, the amount of time it can warp, the warp time limit

and un-warp time requirement respectively. Warping affects a task's EVT as follows

$$E_i := \begin{cases} A_i & warp = 0 \\ A_i - W_i & warp \neq 0 \end{cases} \tag{37}$$

L_i controls how long a task τ_i can stay warped (i.e. how long the *warp* flag can be set) and U_i the minimal amount of time *warp* must stay of after having been set. By using warp, an interactive task can appear to have been waiting to execute longer than it actually has and improve its response time.

A multi CPU extension is presented where all CPUs have their own run-queues where all tasks are included. EVT for a task in a specific CPU run queue is here calculated as

$$E_i := \begin{cases} A_i + M & warp = 0 \\ A_i - W_i + M & warp \neq 0 \end{cases} \tag{38}$$

where M is a migration penalty added if τ_i ran on another CPU last.

The article goes into great detail in investigating how the parameters of BVT can be used to tune the algorithm for different scenarios. Reservation based results can be achieved by scaling m per task so that the desired share is achieved, which is more or less equal to the dynamic weight approach for regular FQ algorithms.

The Completely Fair Scheduler (CFS) Recently, a new scheduler for the Linux kernel was introduced by Ingo Molnar, the creator of the original $O(1)$-scheduler used for the Linux 2.6 kernel series. The scheduler is named "The Completely Fair Scheduler", but the design document recognises that this is impossible on real hardware. The scheduling principle is simple, each task is given a `wait_runtime` value which represents how much time the task needs to run in order to catch up with its fair share of the CPU. The scheduler then picks the task with the largest `wait_runtime` value. On an imaginary completely fair system, `wait_runtime` would always be 0.

The implementation of this is slightly less simple. Each CPU maintains a `fair_clock` which tracks how much time a task would have fairly got had it been running that time. This is used to timestamp the tasks and then to sort them, using a red-black binary tree, by the key `fair_clock - wait_runtime`. As with BVT, penalties are given for migrating to other CPUs. Weights are also exists, but as is common in POSIX systems they are called nice levels and have the reverse meaning (a nice process would have a low weight). `wait_runtime` is also capped so that heavy sleepers won't get too far behind. As the scheduler isn't released at the time or writing, going into more detail is difficult as they are constantly changing.

One thing which is obvious from the general approach is that although not explicitly stated, this looks a lot like the FQ algorithms, perhaps BVT most obviously.

Comparison CBS and FQ Having introduced both the bandwidth server and fair queuing approach to RBS, we can now compare the two methods and see how they differ. The two schemes are compared in the 1999 article by Abeni et al which we will use as basis for the comparison. First we take a look at the interface they provide for reserving bandwidth. The CBS

dedicates an absolute share while FQ uses relative shares. FQ can emulate CBS but with the need to dynamically recalculate the weights when a new task is admitted. FQ algorithms also typically provide bounds on delay, which can be seen as a bound on what deadline requirements a new task can pose. Both schemes have been extended with feedback to adjust weights or bandwidth to achieve some QoS set-point. On the other hand, FQ can more easily be used with mixed RT and non-RT tasks.

The run-time properties of the algorithms are also different. CBS doesn't use quantified time which makes it's performance more consistent over varying hardware platforms. FQ is on the other hand simpler to implement and most consumer OSes. FQ enforces fairness at all times while CBS only guarantees bandwidth allocations between deadlines, making it less conservative. The article makes the case that FQ isn't suitable for CM applications since it doesn't have the notion of task period or deadline, but one can argue that the maximum lag property of a FQ algorithm is a global deadline guarantee, shared by all tasks currently in the system. It is however true that the maximum lag often depend on the number of tasks in the system and the distribution of weights, making the temporal isolation property of FQ less strong. The article also says that FQ would generate lots of context switches in order to enforce fairness, something which isn't the case in general. Instead this depends on the typical job size. CBS will be able to handle arbitrarily large job sizes while FQ uses some time quanta for scheduling. However, as seen with e.g. the BVT algorithm, scheduling allowances can be worked in to reduce the number of context switches, at the cost of worse moment-by-moment fairness.

Latency-Rate Servers In the article "Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms" by Stiliadis et al from 1996, a generalisation of different FQ algorithms are proposed. The class of schedulers called Latency Rate servers (LR-servers) are defined as any scheduling algorithm who guarantees an average rate of service offered to a busy task, over every interval starting at time Θ from the beginning of the busy period, is at least equal to its reserved rate. Θ is called the latency of the server. A large set of the FQ algorithms fit into this class, including WRR, WFQ and SCFQ. Even non-fair algorithms can qualify (one such example is the Virtual Clock algorithm).

The article goes on to derive a number of results for this rather general class of schedulers, including delay guarantees and fairness bounds. One interesting result is that a net of LR-servers can be analysed using one equivalent single LR-server. This can be useful when considering a hierarchy of schedulers.

2.9 Scheduler Control Schemes

The semantics of real-time theory isn't always easy to apply. As we have discussed before, many CM type applications would get very conservative values for WCET or deadlines compared to the average case, making the case for using some sort of feedback scheme. Both CBS and FQ algorithms have been modified to dynamically set server quota and weights respectively to achieve robust and efficient RBS, other feedback schemes have been examined.

Real-Rate Scheduling One of the first results in this direction was published by Steere et al in 1999. The novel approach they take is that they use some task output to measure the rate of progress and thereby eliminate the need for the software designer to assign deadlines or CPU share directly. They use a slightly modified Linux 2.0 series kernel augmented with a RMS based RBS scheme. In the article, a task with no known period or CPU share requirements but a measurable progress is called a real-rate task. The example used in the article is a video pipeline with a producer and a consumer. They exchange data via a queue, which fill level is the metric used for progress. The scheduler samples the queue and decides if either of the two are falling behind getting too far ahead. They use half filled queue as the set-point and then design a PID controller to decide the CPU share needed. Period is decided using some heuristics and is based on the size of the share, lower share meaning longer period.

Controlling Linux in a Nice Way In 2005, Ohlin published his thesis on using a PI-controller to dynamically adjust the nice value of a process in the Linux OS in order to achieve some predetermined bandwidth. At the time, Linux 2.6 was still using Ingo Molnar's $O(1)$-scheduler, which is examined in detail in thesis. The scheduler has some features that makes analysis tricky.

- It uses interactivity heuristics to determine which tasks are interactive (I/O-bound) or not. Based on this, a task can receive a priority bonus or penalty in the interval $[-5, 5]$.
- Nice values are inverted compared to priority levels
- Nice values are non-linearly mapped onto time slice sizes (see listing 1)
- Tasks tagged as interactive are handled differently when they have used up their time slice.

A model for how to calculate the CPU share F_i for task τ_i from the nice value n_i is proposed as

$$F_i = \frac{t_i(n_i)}{\sum_{\forall j} t_j(n_j)} \tag{39}$$

where $t_i(n_i)$ is the time slice for τ_i given its nice value. If we have tasks $\tau_1, \tau_2, \tau_3, \tau_4$ with corresponding nice values $(n_1, n_2, n_3, n_4) = (0, 0, 0, -1)$, they would get the time slices $(t_1, t_2, t_3, t_4) = (100, 100, 100, 420)$ ms. From that we get that

$$F_4 = \frac{420}{100 + 100 + 100 + 420} \approx 58\% \tag{40}$$

Experiments using a standard Linux desktop shows that this gives a correct value within $\pm 1\%$. A controller is then implemented as a kernel module which samples the statistics of a tasks and then sets a new nice value by means of PI control. The reference value for a task is given through a `/proc` interface. The approach works well and is also extended to handle sleeping tasks. The transient behaviour when controlling several tasks simultaneously is however not investigated much. Studying the results when

Listing 1 Code for calculation of a task's time slice, taken from <kernel/sched.c> and <include/linux/sched.h>.

```
#define MAX_USER_RT_PRIO        100
#define MAX_RT_PRIO             MAX_USER_RT_PRIO
#define MAX_PRIO                (MAX_RT_PRIO + 40)

#define USER_PRIO(p)            ((p)-MAX_RT_PRIO)
#define MAX_USER_PRIO           (USER_PRIO(MAX_PRIO))

#define MIN_TIMESLICE           max(5 * HZ / 1000, 1)

#define NICE_TO_PRIO(nice)      (MAX_RT_PRIO + (nice) + 20)

#define DEF_TIMESLICE           (100 * HZ / 1000)

/*
 * task_timeslice() scales user-nice values [ -20 ... 0 ... 19 ]
 * to time slice values: [800ms ... 100ms ... 5ms]
 *
 * The higher a thread's priority, the bigger timeslices
 * it gets during one round of execution. But even the lowest
 * priority thread gets MIN_TIMESLICE worth of execution time.
 */

#define SCALE_PRIO(x, prio) \
        max(x * (MAX_PRIO - prio) / (MAX_USER_PRIO/2), MIN_TIMESLICE)

static inline unsigned int task_timeslice(task_t *p)
{
        if (p->static_prio < NICE_TO_PRIO(0))
                return SCALE_PRIO(DEF_TIMESLICE*4, p->static_prio);
        else
                return SCALE_PRIO(DEF_TIMESLICE, p->static_prio);
}
```

controlling two tasks reveals some inference when one of the tasks changes its share. A possibly way to improve performance here would be to handle this as a multivariable control problem.

3. Implementations of RBS

Presently, a number of free or open source implementations of RBS schemes exists. Here follows descriptions of a few important initiatives.

3.1 OCERA

OCERA stands for Open Components for Embedded Real-time Applications, and is an European project, based on Open Source, which provides an integrated execution environment for embedded real-time applications. From a RBS point of view, OCERA offers a number of interesting components. The OCERA code is based on the RTLinux extension.

The Generic Scheduler patch for Linux kernel It is a small patch for the Linux kernel that provides "hooks" for modules implementing generic scheduling policies. Our particular interest would be to use it to implement a resource reservation scheduling module.

Integration Patch This patch makes the Preemptive Kernel Patch work for RTLinux. The Preemptive Kernel work was done by Robert Love with the aim of improving latency by making system calls possible to preempt.

Resource Reservation Scheduling module This is a dynamically loadable module for the Linux kernel, that provides a resource reservation scheduler for soft real-time tasks in user space. It uses a CBS based algorithm, modified to handle some practical issues. It includes optional slack reclamation functionality using the GRUB algorithm. The module provides a new scheduling policy, `SCHED_CBS`.

Quality of Service Manager for Linux A QoS management services module for the Linux kernel. This is more or less a controller who changes the bandwidths according to scheduling error. The approach is more or less that presented in 2.5.

Other work OCERA includes a large body of work which is not directly related to RBS. Such work include Application-defined Scheduling (ADS), enhanced memory allocation algorithms and a wide range of improvements to the RTLinux kernel.

3.2 AQuoSA

AQuoSA stands for Adaptive Quality of Service Architecture and is another initiative to bring QoS to the Linux kernel. It can be seen as based on the work provided by OCERA and is partially sponsored by the FRESCO European project (Framework for Real-time Embedded Systems based on COntRacts). Structure wise it retains the components used by OCERA.

3.3 Xen

Virtualisation technologies makes it possible to partition a physical computer into several logical instances, each one running a separate operating system that thinks it is running alone on the hardware. The layer beneath the OS layer is sometimes called the hypervisor layer as it uses a special mode enabled in modern CPUs called the hypervisor mode. Xen is an open source hypervisor created in 2003 at the University of Cambridge in a project headed by Ian Pratt.

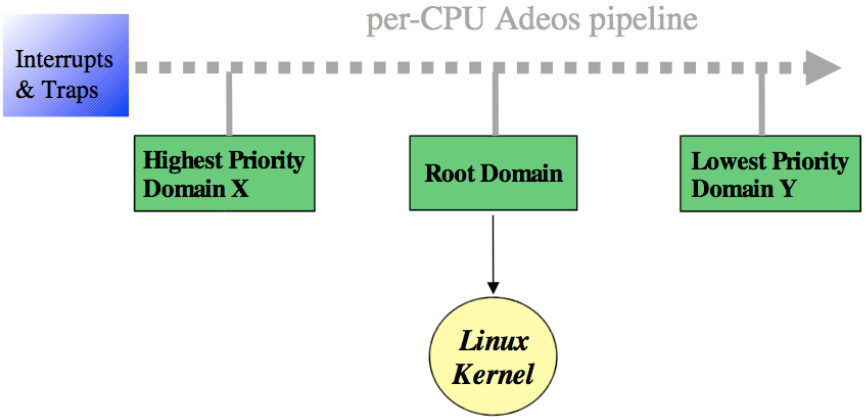


Figure 16 A schematic over the Adeos interrupt pipe. Taken from the paper "Life With Adeos"

Architecture A Xen system has multiple layers, the lowest and most privileged of which is Xen itself. Xen can host multiple guest operating systems, each of which is executed within a virtual instance of the physical machine, a domain. Domains are scheduled by Xen and each guest OS manages its own applications. This makes up a hierachy of schedulers with the Xen scheduler on top.

Xen supports several top level schedulers, including EDF and BVT. Xen 2 also supported the Atropos scheduler, but this has been removed in Xen 3.

3.4 Xenomai

Xenomai is a real-time development framework cooperating with the Linux kernel, in order to provide a pervasive, interface-agnostic, hard real-time support to user-space applications, seamlessly integrated into the GNU/Linux environment. It is an alternative to RTLinux and RTAI and is a possible platform for developing RBS schemes. It was launched in 2001 and in 2003 merged with the RTAI project. The projects split again in 2005, going after separate goals.

It achieves superior RT performance compared to regular Linux while still allowing regular applications. The RT kernel is a small, efficient run-time which executes the Linux kernel as a low priority task. Real-time tasks will then be run by the RT kernel, next to the Linux kernel. The Xenomai kernel also provides an API with extensive support for real-time primitives that the Linux kernel lack, including support for periodic task models. Xenomai uses a hypervisor called Adeos, which is a lot like the Xen hypervisor but focuses on real-time performance. Specifically it is used to handle interrupts so that even if a Linux process has requested interrupts to be turned off, the RT-tasks will be able to pick them up. See figure 3.4.

3.5 Class-based Kernel Resource Management (CKRM)

The CKRM project aims to created a framework for providing differentiated services to resources such as CPU, memory and I/O. This means that not only can a process reserve a certain CPU bandwidth, they can also

reserve I/O bandwith, memory access etc. The class concept is used to group together tasks with similar goals and similar importance and in that way govern their right to resources. Each class is associated with a set of controllers responsible for managing access to the resources. The project changed named to Resource Groups lately, but have been somewhat coldly received on the LKML, mainly motivated with it being a very large patchset. This is something which may be fixed with the Generic Process Container patchset.

3.6 Generic Process Containers

This is a patchset which has seen a positive reaction on the LKML, aiming to do similar things as CKRM, but by extending cpusets, a construct already in the Linux kernel. The patchset extracts the process grouping code from cpusets into a generic container system, and makes the cpusets code a client of the container system. The intention is that the various resource management and virtualization efforts can also become container clients, with the result that

- the userspace APIs are (somewhat) normalised
- the additional kernel footprint of any of the competing resource management systems is substantially reduced, since it doesn't need to provide process grouping/containment, hence improving their chances of getting into the kernel

3.7 Resource Kernels and Linux/RK

A more formal approach to what CKRM tries to accomplish has been suggested by Rajkumar et al in their work on Resource Kernels. This was introduced in their article "Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems" [19]. The article tries to solve a number of problems associated with multi-resource reservations, including the computational complexity of such a system, which has been shown to be an NP-complete problem.

Design Requirements The ambition of the project becomes evident from studying the goals set down for the design of the kernel. In short, they are as follows

- **G1. Timeliness of resource usage.** An application which has made a reservation must be given access to it promptly when needed. An application must also be able to up and downgrade its resource usage (for adaptation and graceful degeneration purposes).
- **G2. Efficient resource utilization.** By this, we mean that the OS must be able to satisfy G1 while making as few restrictions as possible. E.g. it is possible to satisfy G1 by only allowing one process at a time, regardless of how small reservations it wants to make.
- **G3. Enforcement and protection.** The enforcement of resources should be such that abuse by one application does not hur other applications.
- **G4. Access to multiple resources.** Access to multiple resources by the same process must be possible.

- **G5. Portability and automation.** Applications should ideally be able to specify their resource requirements regardless of hardware (e.g. the CPU clock frequency). In addition, resource demands should be automatically tunable.
- **G6. Upward compatibility with fielded operating systems.** The resource kernel should provide support for regular OS services such as regular scheduling algorithms, real-time structures with priority inheritance etc.

Reservation Model The Resource Kernel uses the parameters C, T, D, S and L with the meanings of a computation time C every T time-units within a deadline D from the start time S over an allocation life-time L. The semantics allow for several types of reservations:

- *Hard reservations.* A hard reservation will not be replenished on depletion, even if possible.
- *Firm reservations.* Such a reservation will be replenished if all other reservations are depleted.
- *Sort reservation.* Will be replenished if possible, even if other non-depleted reservations exist.

Portable RK In the article "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior" [18], a portable implementaion of the Resource Kernel concepts is presented and shown how to work on the Linux kernel. It is considered portable since it doesn't require any changes to the OS code itself if given access to

- a fixed-priority scheduler
- an interface to change the priority of running tasks
- an interface for suspending and resuming jobs
- an interface for acquiring events within execution objects, needed for accounting reserves

The first three are available from most modern OSes, while the last required some hooks to be inserted in the Linux kernel. The article states that investigations for how to use the native debugging interface could be exploited for these purposes and remove the need for modifying the OS. Arguably, the solution isn't entirely portable yet.

4. References

[1] Luca Abeni and Giorgio Buttazzo. Adaptive bandwidth reservation for multimedia computing. *rtcsa*, 00:70, 1999.

[2] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. Analysis of a reservation-based feedback scheduler. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 71, Washington, DC, USA, 2002. IEEE Computer Society.

[3] David P. Anderson, Shin Tzou, Robert Wahbe, Ramesh Govindan, and Martin Andrews. Support for continuous media in the dash system. Technical report, Berkeley, CA, USA, 1989.

[4] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosie. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *rtss*, volume 00, pages 182–190, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[5] Anton Cervin and Johan Eker. The control server: A computational model for real-time control tasks. *ecrts*, 00:113, 2003.

[6] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L. Abeni. Adaptive reservations in a linux environment. *rtas*, 00:238, 2004.

[7] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 1–12, New York, NY, USA, 1989. ACM Press.

[8] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. *SIGOPS Oper. Syst. Rev.*, 33(5):261–276, 1999.

[9] S. Jamaloddin Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOM '94. Networking for Global Communications. 13th Proceedings IEEE*, pages 636–646 vol.2, 1994.

[10] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchial cpu scheduler for multimedia operating systems. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 107–121, New York, NY, USA, 1996. ACM Press.

[11] Ralf Guido Herrtwich. The role of performance, scheduling and resource reservation in multimedia systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, pages 279–284, London, UK, 1991. Springer-Verlag.

[12] Kevin Jeffay and Steve Goddard. A theory of rate-based execution. *rtss*, 00:304, 1999.

[13] H. Kaneko, J.A. Stankovic, S. Sen, and K. Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. *rtss*, 00:206, 1996.

[14] Caixue Lin and Scott A. Brandt. Improving soft real-time performance through better slack reclaiming. *rtss*, 0:410–421, 2005.

[15] Giuseppe Lipari and Sanjoy Baruah. A hierarchical extension to the constant bandwidth server framework. In *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, page 26, Washington, DC, USA, 2001. IEEE Computer Society.

[16] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves for multimedia operating systems. Technical report, Pittsburgh, PA, USA, 1993.

[17] J. B. Nagle. On packet switches with infinite storage. pages 136–139, 1988.

[18] S. Oikawa and R. Rajkumar. Portable rk: A portable resource kernel for guaranteed and enforced timing behavior. *rtas*, 00:111, 1999.

[19] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: a resource-centric approach to real-time and multimedia systems. pages 476–490, 2001.

[20] C. A. Waldspurger and E. Weihl. W. Stride scheduling: Deterministic proportional- share resource management. Technical report, Cambridge, MA, USA, 1995.

[21] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. 1994.