# FLEXIBLE SCHEDULING IN MULTIMEDIA KERNELS: AN OVERVIEW

**Pierre G. Jansen, University of Twente, Netherlands**
**Hans Scholten, University of Twente, Netherlands**
**Rene Laan, Océ Technologies B.V., Netherlands**
**Keywords: R & D, Multimedia, Interactive Systems**

**Abstract:**

Current Hard Real-Time (HRT) kernels have their timely behaviour *guaranteed* on the cost of a rather restrictive use of the available resources. This makes current HRT scheduling techniques inadequate for use in a multimedia environment where we can make a considerable profit by a better and more *flexible* use of the resources. We will show that we can improve the *flexibility* and *efficiency* of multimedia kernels. Therefore we introduce *Real Time Transactions* (RTT) with *Deadline Inheritance* policies for a small class of scheduling algorithms and we will evaluate these algorithms for use in a multimedia environment

## 1 INTRODUCTION

Recent developments in the field of multimedia and communication architectures, open an exciting range of new applications. This is particularly true when combining them with new powerful systems that deal with vision, sound and control. In the field of multimedia we have the possibility of manipulating sound and vision, robotics deals with control, while recent developments on computer and communication architectures open the possibility to distribute these functions. This opens a range of new architectures among which we find TV computers, Set Top Boxes and network computers. Typical applications for these architectures are shopping, games, (tele-) education, travel services, dating, (video-) conferencing, etc. Moreover, multimedia can and will also be used in other than these typical applications. For instance, in the process control environment we see the deployment of the combination of remote viewing, remote hearing and remote control, mainly composed of “off the shelf” components. Furthermore the deployment of Asynchronous Transfer Mode (ATM) networks allows for real-time transmission of data.

A multimedia application tries to establish a contract between several parties, such as a producer, a network and a consumer for the delivery of multimedia services. These services, such as recording, transport, processing, displaying or broadcasting of sound or video, are agreed to be delivered with a certain Quality of Service (QoS). QoS is measured with different parameters for each service. For video recording and displaying these are for instance frame rates, window surface, resolution and number of colours. For sound the sample frequency, the jitter and the number of bits have to be considered.

Processing power is supplied under the responsibility of a scheduler. The QoS parameters here are the period, run length, deadline, release time and resource

usage. A difficult issue in QoS is that it is not always clear how to relate QoS parameters of the different services to each other. A possibility that we would like to support is to vary the requested QoS in order to adapt to the latest requirements of an end-user who would like to add, delete or change a multimedia application. In these cases a QoS manager has to arrange the new or changed setup by acting as a dealer for services.

Among others, the task of a QoS manager is to derive a QoS specification for the scheduler. These specifications include process period, arrival time, release time, computation time and deadline, thereby taking into account the use of resources. With this information the scheduler could start a schedulability analysis. It might be that the QoS manager is satisfied with an average QoS schedulability analysis of the scheduler. However, under circumstances a hard guarantee may be needed. In both cases a positive scheduling analysis from the scheduler to the QoS manager implies that a contract can be concluded. This paper presents scheduling algorithms that allow for QoS analysis. The analyses itself is not presented here. It is however ready for publication but not published yet.

Multimedia applications require some degree of timeliness ranging from Soft Real-Time (SRT) for most multimedia applications to Hard Real-Time (HRT) in embedded systems. For these environments we will consider scheduling techniques delivered by (RT) kernels that offer RT precision when needed and flexibility when possible.

There are already numerous RT kernels on the market. They can be found in a general-purpose “time sharing” environment or in a dedicated “real-time” environment. General-purpose operating systems (GPOSs) sometimes have some kind of RT provisions such as (fixed) priorities associated with tasks. For embedded systems these kernels offer superfluous non-RT provisions such as paging and windowing systems. On the other hand they mostly lack provisions such as priority inheritance which is needed to avoid phenomena like priority inversion, late reactions and unnecessary reservation of resources. Examples of such systems are OS/2, Solaris and NT.

It is also possible to adapt GPOSs to RT needs by incorporating full preemption of the kernel -- needed to serve RT request timely -- and by introducing suitable process scheduling techniques. Typical examples are RT Mach (Nakajima, 1993) and also Real/IX (Furth, 1991). RT-Linux (Yodaiken, 95) uses a virtual machine technique which runs both Linux as well as RT-Linux on a hardware adaption layer on the same computer.

Dedicated RT kernels have provisions to serve HRT tasks such as reservation of shared resources and schedulability analysis. These can be done off-line or at run-time or a combination of both. Off-line handling offers speed but is inflexible. Handling at run-time might be time consuming in the real-time domain. Also scheduling of RT tasks can be needlessly complex. This paper investigates

*simple*, *flexible*, *fair* and *responsive* scheduling strategies with low administration overhead in such a way that a straightforward analysis of RT behaviour is attainable. We have good reasons to claim that these strategies can be used for targets ranging from dedicated RT kernels to GPOSs with RT properties. We have working prototypes of (1) dedicated HRT kernels for embedded systems, (2) kernels for multimedia, and (3) we have a proposal for embedding these techniques in a RT GPOS. In particular our techniques allow for dynamic admission of new tasks and for QoS variation of running tasks.

RT algorithms are also meaningful classified as static or dynamic:

- A *static* algorithm uses *a priori* information of tasks about periods, arrival times and resource usage. Scheduling decisions are typically made off-line or in background and are put in a time-ordered event list.
- A *dynamic* algorithm uses little or no a priori information about the arrival of tasks. Scheduling decisions are made on-line at distinguished events such as arrival or completion of tasks, or while claiming or releasing resources.

Static algorithms can be tuned optimally for static tasks. They are however not flexible. Dynamic algorithms offer more flexibility. In the context of this paper we refer to dynamic scheduling algorithms. They are all based on variations of the Earliest Deadline First (EDF) rule (Liu, 1973). First we give an overview of the existing techniques in the dynamic kernels in section 2. Our task model is described in section 3. Interesting scheduling protocols are presented and evaluated in section 4. Finally an overview of our implementations and tests is given in section 5 as well as a proposal for a GPOS kernel.

## 2 EXISTING PREEMPTIVE SCHEDULING METHODS

In preemptive scheduling algorithms tasks are scheduled according to a *priority*. A task may preempt another task if it has a higher priority. The priorities of the following well-known scheduling methods are determined as follows:

- Earliest Deadline First (EDF). Priority increases dynamically when the deadline comes closer.
- Rate Monotonic (RM). Priority is static and is inversely proportional to the period time: short periods are mapped on high priorities. The deadline is equal to the end of the period.
- Deadline Monotonic (DM). Priority is static and is inversely proportional to the deadline interval. The deadline is before the end of the period.

Note that RM is a special case of DM. Note also that EDF task do not necessarily need to have a fixed repetition period. However, when it comes to “QoS schedulability analysis” we need to take periodic repetition into account in order to compute the task's work load.

Without any precautions scheduling methods may lead to the phenomenon of *blocking*, *priority inversion* or *transitive waiting*. Blocking may happen when *shared resources* are used. In this context we mean by shared resources those resources for which a task has to obtain mutual exclusive (mutex) use. A waiting task cannot preempt a running one to use a mutex resource if the latter is using this resource. Note that in this context the processor itself is a shared but not a mutex resource, since it is made preemptable at any time. If we mention resources in the sequel of this paper we mean mutex resources except stated otherwise explicitly. Blocking happens when a high priority task must wait for the release of a resource by a low priority task. Priority inversion is a special form of blocking. It occurs when a high priority task is blocked, waiting for a resource that is held by a low priority task that is preempted by a medium priority task. Transitive waiting occurs in a chain of tasks, which are all waiting for the release of resources of their predecessors. This may cause large (indirect) blocking values.

The priority of a task can be *static* or *dynamic*. A static priority does not vary in time while a dynamic priority does. Note that in EDF a deadline can be expressed as a static or a dynamic priority. A deadline interval -- from release to deadline -- is associated with a static priority while an absolute deadline will be associated with a dynamic priority. A scheduler orders tasks to priority and a dispatcher assigns these tasks to the processor(s) in the resulting order. In dynamic RT systems the dispatcher and scheduler are mostly combined to one entity and referred to as “the scheduler”. This scheduler executes protocols such as

- basic protocols, like the Fixed Priority (FP) protocol and the Basic Inheritance (BI) protocol,
- ceiling protocols, like the original Priority Ceiling Protocol (Sha, 1990) and the Stack Resource protocol (Baker, 1991),
- transaction protocols, in several variants of the Real-Time Transaction (RTT) protocol (Jansen, 1996).

All but the FP-protocol provides methods to bound the duration of blocking. BI realises this by inheriting either static or dynamic priority. A low priority task $t_l$, owning shared resources that are also requested by high priority tasks $t_h$, inherits the high priority from $t_k$. BI bounds blocking, however it cannot avoid transitive waiting. The ceiling protocols -- PC and SR -- limit blocking. Both use static priorities. They avoid both priority inversion and transitive waiting. The basic idea is to make way for a high priority task -- say $t_k$ -- by *not* allowing preemption of a low priority task -- say $t_l$ -- by any medium priority task -- say $t_m$ -- if $t_l$ uses resources also claimed by $t_k$. This strategy limits blocking to *one single* task only -- or more precise -- to one critical section only. This implies that transitive waiting is not possible and consequently *deadlock* is impossible. The Real-Time Transaction protocols (RTTs) also avoid priority inversion and transitive waiting. They are based on EDF, either with absolute or relative deadlines.

When a transaction starts, it simultaneously acquires all resources it needs to complete the transaction. During the transaction, resources can only be released. A transaction completes when it has released all resources. Priority inheritance is applied dynamically when a high priority transaction must wait for resources in use by a low priority transaction. This avoids preemption of low priority transactions and advances the release of resources. We will now introduce a task model suited for flexible scheduling of transactions. Based on this model we will introduce our scheduling protocols and analyse their pros and cons in section 4.4.

# 3 TASKS AND TRANSACTIONS

We now introduce our task model. Tasks are based on transactions, which make the use of critical sections for mutual exclusive resources superfluous. This makes our task model quite straightforward and has positive consequences for administration overhead and for schedulability analysis. We will consider them in the relevant sections.

A non-periodic task can be considered as a sequence of “free-running” or “resource-using'” *transactions*. A “free-running transaction” is not subject to *mutex scheduling constraints* since it does not use shared resources. An invocation of a `”resource using” transaction, can only be run if it can acquire all its resources *simultaneously*. This condition guarantees that a transaction always runs to completion. Unbounded priority inversion, transitive waiting and deadlock are impossible. A transaction may release its shared resources at any time. However, for the sake of simplicity, we assume that a transaction releases its resources when it runs to completion. Dealing with early release times is possible, however, a little more complicated and beyond the scope of this paper.

When we refer to *periodic* tasks we model them as a single periodic transaction. Whether transactions are periodic is not relevant for the proposed scheduling algorithm, but, for QoS schedulability analysis, it is relevant.

A transaction may be in one of the following states: *sleeping, ready*. The ready state is split up in *released, running* or *preempted*. A transaction is put into the administration after it is admitted to the system. It is then put in the sleeping state where it waits for its release time, after which and it enters the ready queue.

In the *ready* state a transaction can be *released* when it is waiting for the processor, *running* when it has the processor or *preempted* when it had to leave the processor to a transaction with a higher priority. When a transaction is done, it is put into the *sleeping* state waiting for the following release event. When a transaction is completely finished it is withdrawn from the administration.

A transaction $t_i$ is a member of the set of all transactions $\boldsymbol{t} = \{ t_1, \ldots, t_n \}$

**Definition Transaction**: Transaction $t_i$ is defined as the tuple of static parameters $(D_i,T_i,C_i,R_i)$

where $D_i$ is the deadline interval, $T_i$ is the time interval between two successive invocations -- the period -- , $C_i$ is the maximum run-time interval $T_i$ takes to complete and $R_i$ is the set of resources which are used by $t_i$. The first invocation of $T_i$ is denoted by $T_i^0$. Invocation $j$ of $T_i$ is denoted by $T_i^j$

**Definition Invocation**: Invocation $t_i^j$ is associated with static parameters $(t_i^j,d_i^j)$ of the $j^{th}$ invocation of $t_i$

where $r_i^j$ is the absolute release time from which an invocation $j$ may run and $d_i^j$ the absolute deadline at which an invocation $j$ has to be completed. Note that $D_i = d_i^j - r_i^j$ for all $j$.

A transaction $t_i$ is also associated with a static priority $P_i$. An invocation $t_i^j$ is associated with a dynamic priority $d(t_i^j)$. A priority determines the processor rights. If there is a competition for the processor, the transaction or the invocation with the highest priority wins. In the protocols as described in section 4 both types of priority will be used. In general, for EDF-oriented scheduling protocols, the following relations between deadline and priority hold:

$$\begin{aligned} &D(t_a) > D(t_b) \Leftrightarrow P(t_a) < P(t_b) \\ &d(t_a^j) > d(t_b^k) \Leftrightarrow p(t_a^j) < p(t_b^k) \end{aligned} \qquad (1)$$

# 4 THE PROPOSED INHERITANCE PROTOCOLS

This section discusses two protocols for scheduling of tasks that are variants of the Priority Ceiling protocol (PC) and the Stack Resource (SR) protocol. Both protocols are based on real-time transactions and run under an inherited *preemption level*. The preemption level determines which transactions may preempt a running one. Preemption levels can be based on absolute deadlines or on deadline intervals. PC as well as SR has a preemption level that is statically derived from deadline intervals; SR has a dynamic refinement.

## 4.1 CEILING PROTOCOLS

Ceilings are used in the Priority Ceiling protocol (Sha, 1990) and in the Stack Resource protocol (Baker, 1991). We will now introduce variants of these protocols and evaluate their advantages and disadvantages. For clarity we have chosen *not* to introduce these protocols in their full glory but only in their essentials. Consequently we use transactions instead of nested critical sections and single-unit resources instead of multiple-unit resources. First we will introduce the notion of *ceiling* and *preemption level*. Then we introduce a simple variant of the Priority Ceiling (PC) protocol and successively we will extend this protocol to an interesting variant of it, the Stack Resource (SR) protocol. PC and SR were

originally defined in terms of priority. In order to prevent confusion with the already introduced notions, we prefer to present them with deadlines instead of priority. This can hardly lead to confusion with the relations as given in (1) in mind.

## 4.2 PRIORITY CEILING PROTOCOL

In the Priority Ceiling protocol inheritance of deadlines is effectuated over the use of shared resources and the smallest deadline -- the highest priority , traditionally the ceiling -- of any transaction that uses this resource is of interest. The ceiling $D_R$ of a resource R is defined as the smallest deadline of any transaction that uses this resource:

$$D_R = min\{D_x, R \in R_x\} \quad (2)$$

The inherited preemption deadline $D_a$ of a transaction $t_a$ defined as follows:

$$D_a = min\{D_a, D_R \mid R \in R_a\} \quad (3)$$

$D_a$ is a static property of $t_a$ and can be computed off-line. The smallest preemption deadline of all running or preempted transactions is the running one and denoted as $d$.

**Definition PC**: PC is defined by the following rules:

1. Released but not yet running or preempted invocations are ordered to their static deadlines $D_i$.
2. The invocation $t_a^k$ with the shortest deadline -- say $D_a$ - is selected for processor competition.
3. $t_a^k$ will preempt the running invocation iff $D_a < d$.

Note that this protocol is very easy to implement. All static information can be computed off-line or in background. Our PC variant, based on transactions, is stricter than the original PC. Note that our variant shows a last-in first-out behaviour of running transactions. The running invocation is the last one which is introduced and if it is not preempted it will be the first one to complete. This opens the possibility for using a single shared stack for all transactions. In the case that many small transactions are running, this would considerably limit the amount of memory needed. We now introduce a refined variant of the PC protocol, the SR protocol.

## 4.3 STACK RESOURCE PROTOCOL

SR is a refinement of PC. Under SR an invocation $t_a$ does not have only a static deadline $D_a$ but also a dynamic one $d_a^j$. Its preemption level is determined by a pair $(D_a , d_a^j)$ where $D_a$ is defined as in (3) under PC.

**Definition SR**: SR is defined by the following rules:

1. Released but not yet running or preempted invocations are ordered to their dynamic deadlines $d_l^j$.
2. The invocation $t_a^k$ , with the shortest dynamic deadline -- say $d_a^k$ -- is selected for processor competition.
3. $t_a^k$ will preempt the running invocation $t_r^l$ iff $(D_a < D_r) \wedge (d_a^k < d_r^l)$

Due to the last-in first-out character of SR we may conclude that the running invocation is on top of a stack of preempted invocations. The maximum priorities *P* and *p* are associated with the running invocation. SR was earlier published in (Baker, 91). It used several refinements such as multiple unit resources and nested critical sections. For more details we refer to the original article.

## 4.4 EVALUATION OF PC AND SR

Both protocols do *not* need explicit use of synchronisation primitives such as semaphores. Due to inheritance and order -- EDF in our case -- synchronisation is implicitly accomplished. This obliterates the explicit request for mutual exclusion; o additional synchronisation primitives are needed. This makes these algorithms straightforward, easy to reason about, and easy to implement.

In our further evaluation we will consider several aspects of the protocols such as:

- Efficiency: what is the complexity of the protocol; is it easy to implement and does it run efficiently; can it run on a single shared stack?
- Flexibility: does the protocol adapt to changes in its immediate environment, can it be extended with multiple unit resources and can transactions be refined to nested use of resources?
- Blocking: Blocking occurs if a high priority invocation must wait for the release of resources by one or more low priority invocations. Blocking is an important issue in QoS schedulability analysis, since it has as a consequence that some blocking load has to be executed also before the blocked invocation may start.

Both protocols are easy to implement and run efficiently. They can have resource usage of transactions replaced by nested resource usage. The use of resources is nested if an earlier acquired resource is released later then the release of any later acquired resource. This has as an advantage that blocking can decrease. This is because the preemption level is increased at the actual acquisition of a resource instead of the start of a transaction and decreased when the resource is released. The disadvantage is some more administration overhead. However the main characteristics of PC and SR do not really change.

PC is very straightforward. Our variant can run on a stack. It has a small overhead. Ceilings and preemption levels can be computed off line or in background. Blocking is limited to only one invocation. PC is a good and powerful candidate

for use in any RT environment. It has the small disadvantage that all used scheduling information is static. This might make its dynamic behaviour somewhat inflexible.

SR brings dynamic behaviour into play again by adding dynamic priority to static priority when scheduling decisions have to be made. SR inherits all the good static properties from PC: small overhead, possibility of off-line computation of ceilings, a maximum of one blocking invocation and the possibility of a shared stack. It offers some freedom in the choice of the dynamic part: earliest deadline first (EDF), rate monotonic (RM) or deadline monotonic (DM) may be chosen on top of the static part. Choosing RM would lead to a resource-using variant of the original RM: with the possibility of using a stack and with limited blocking. Choosing DM does to our opinion not make much sense. Our favourite is EDF: the dynamic priority is derived from absolute deadlines. When, in the following, SR is mentioned without further specification, we mean SR/EDF. If we compare SR/EDF to PC, we see that SR/EDF gives a higher priority to transactions that are waiting for execution for already some time and for which the deadline comes closer.

## 5 IMPLEMENTATION AND TESTS

A framework for scheduling experiments with PC and SR/EDF has been added to Inferno and is described in (Bos, 97). Test results are not yet available. A considerable amount of work has been done for the determination of the QoS feasibility analysis for PC (Jeffay, 1991) and for SRP/EDF (Ripoll, 1996). We could improve the results of the latter by presenting an algorithm with a refined estimation of the blocking component. The details are beyond the scope of this overview, but are ready for publication elsewhere.

## 6 CONCLUSIONS

We have evaluated two dynamic, real-time scheduling policies adapted to *multimedia* requirements but also suited to hard real-time. These policies are based on the principle of (1) *Earliest Deadline First*, (2) on *Real-Time Transactions* and (3) on *selected inheritance strategies*. We experimented with variants of the *Ceiling Protocol* and the *Stack Resource protocol*. Our experiments show that all policies are flexible and fair and allow for an efficient multimedia kernel-level scheduling. Although not presented in detail, in this paper, we like to emphasise that the Ceiling Protocol and the Stack Resource protocol allow for precise *Quality of Service* analysis. Consequently these two are not only suited to “multimedia quality of service management” but also to ``task feasibility determination” of hard real-time systems.

The scheduling overhead of our run-time policy is low. This is due to the orthogonality of the ingredients, which enable a systematic implementation.

Among others, mutual exclusion is guaranteed by the aforementioned ingredients (1) to (3). No additional synchronisation primitives are needed. Experiments have shown a scheduling overhead can be less than 1 percent. This is really a low price for the offered services. The dynamic behaviour of the Stack Resource protocol, combined with its possibility for precise "quality of service analysis", makes the presented SR variant our favourite for further use and analysis in real-time and multimedia systems.

## ACKNOWLEDGMENTS

Acknowledgements are due to Ties Bos and Ferdy Hanssen for their contributions during the Wednesday afternoon sessions and for comments on an early version of this manuscript.

## REFERENCES

Baker T.P. (1991), "Stackbased scheduling of real-time processes," *The journal of real-time systems,* Vol. 2, pp.67-99.

Bos M. (1997), "Real-time Scheduling in Inferno," *Master's thesis,* Univ. of Twente.

Furth B. et al. (1991), "Real-Time UNIX System: Design and Application Guides," *Kluwer Academic Press*.

Jansen P. G., Wijgerink E., (1996), "Flexible Real-Time Scheduling of Continuous Media Streams: A Multimedia Experiment," *Multimedia 96 Conference,* Yokohama, Japan, pp. 323-330.

Jeffay K., Stanat D.F., Martel C.U., (1991) "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks," *Proc. of the 12$^{th}$ IEEE Real-Time Sys. Symp,* pp. 129-139.

Liu C.L. and Layland J.W., (1973), "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM,* Vol. 20, pp. 40-61.

Nakajima T., Kitayama T., Tokuda H., (1993), "Experiments with Real-time Servers in Real-time Mach," *Proc. of USENIX Mach III Symposium*, pp. 1-19.

Ripoll I., Crespo A., Mok A.K., (1996), "Improvement in Feasibility testing for Real-time Tasks," *The Journal of Real-time Systems,* Vol. 1, pp. 19-39.

Sha L., Rajkumar R., Lehoczky J.P., (1990), "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. on Comp.,* Vol. 39, No. 9, pp. 1175-1185.

Yodaiken V., Barabanov M., (1995), "A real-time Linux," *Technical report,* New Mexico Institute of Technology, Aug.