Lightweight EDF Scheduling with Deadline Inheritance

Pierre G. Jansen, Sape J. Mullender, Paul J.M. Havinga, Hans Scholten
jansen,havinga,scholten@cs.utwente.nl, sape@plan9.bell-labs.com

May 9, 2003

Abstract

EDFI is a lightweight real-time scheduling protocol that combines EDF with deadline *inheritance* over shared resources. We will show that EDFI is *flexible* during a task's admission control, *efficient* with scheduling and dispatching, and *straightforward* in feasibility analysis. The application programmer only needs to specify a task's timing constraints (deadline, period, runtime) and resource needs, after which EDFI can execute admission control, scheduling, dispatching and resource synchronisation automatically. EDFI avoids gratuitous task switching and its programming overhead as well as runtime overhead is very low, which makes it ideal for lightweight and featherweight kernels. We will illustrate the elegance of the underlying theory and we will shortly discuss the implementation of EDFI in three different operating systems[1].

Keywords: preemption, deadline inheritance, feasibility analysis, run-to-completion semantics

1 Introduction

In many embedded systems, applications have hard real-time (RT) requirements while others are soft RT or can be best effort. Soft RT or best effort systems are not good at guaranteeing deadlines while hard RT systems are. Various attempts have been made to introduce RT schedulers to general-purpose operating systems. Most of these systems cannot give hard RT guarantees when shared resource are used, because the required mechanisms are too complex in terms of needed code or required system state. In general, a rather complex feasibility analysis is needed to determine whether

[1]This research is sponsored by the IBM Equinox Programme and by the EU project Eyes, IST 2001-34734.

a set of task can meet deadlines beforehand. This analysis can be done in a rigid way by pre-executing a *static* schedule that is executed by a dispatcher, or it can be done in a flexible way by letting the dispatcher decide *dynamically* to derive scheduling decisions. Dynamic scheduling is more flexible but in general more expensive in terms of computing time.

We propose a dynamic scheduler to be used with *on-line* feasibility analysis. Our analysis is simple enough to be executed on-line – outside the real-time time budget – to handle admission control of new tasks in a precise and flexible way. We allow shared resources, with a minimum of needed code and memory usage. We keep the feasibility analysis outside the RT timing budget, because the analysis itself does not take advantage from a fixed deadline. If system resources, in particular memory, are very scarce it is better to do the analysis off-line.

In this paper we present a new scheduling and dispatching technique for real-time support in operating systems, which range from normal to featherweight. Our scheduling method is based on preemptive Earliest Deadline First (EDF), as first introduced by Liu & Layland [1], in a context where shared resources can be used under mutual exclusion. This, in general, complicates scheduling, resource synchronisation and switching and confronts the application programmer with a rather complicated environment. We will tame this complexity with the EDFI approach as described in the following.

Under the EDFI protocol we propose to combine preemptive EDF with deadline inheritance over shared resources. EDFI can manage scheduling and dispatching very efficiently. It uses very little system code and processing overhead and it hardly needs additional memory (RAM). Therefore it is suitable for feather-light micro kernels. EDFI allows for a straightforward feasibility analysis, which is derived from an elegant underlying mathematical model, which results from the introduction of inherited deadlines under EDF. A consequence of this straightforward analysis is that it is practical to do admission control on-line. This offers a service in which admission control, based on feasibility can flexibly allow or reject new or changed tasks. Moreover, process switching is limited to a minimum, while mutual exclusion of shared resources is granted at system level so that the programmer does not need to take care of synchronisation: processes are simply not scheduled by the system when there is a potential resource conflict.

We have integrated this technique in a few operating systems, namely RT-Linux, in Plan 9 and in a tiny operating system called "Real-Time eYes" (RTY), which we use for radio connected sensor networks. Although other operating systems may also have RT support, we believe there is no other operating system with a comparable native support for RT applications that is so light weight as our scheduler and dispatcher.

In the subsequent sections, we shall describe our system and the theory behind it, omitting, for lack of space detailed proofs. In section 4 we shortly

Table 1: Specification of Ω_1

Ω_1	D_i	T_i	C_i
τ_1	3	4	1
τ_2	5	8	1
τ_3	6	10	2
τ_4	9	15	4

describe our experiments with RT-Linux, Plan 9 and RTY. For a more formal introduction, see Jansen & Laan [2].

2 Theory

A *task set* Ω consists of a set of preemptable tasks τ_i ($i = 1...n$). Each task τ_i is specified by a *period* T_i, a *deadline* D_i, a *cost* C_i, and a *resources specification* ρ_i. It is *released* every T_i time units and must be able to consume at most C_i seconds of CPU time before reaching its deadline D_i seconds after release ($C_i <= D_i <= T_i$). We use capital letters for time intervals (e.g., T, D, C) and lower case for absolute 'points in time': r for the next release time, d for the next deadline.

The *utilisation* U of Ω is defined as $U = \sum_{i=0}^{n} C_i/T_i$. For Ω to be schedulable, $U <= 1$ must hold. We define two functions, *processor demand* $H(t)$, introduced by Baruah *et al˙* [3], and *workload* $W(t)$, introduced by Audsley *et al˙* [4]. $H(t)$ represents the total amount of CPU time that must be available between 0 and t for Ω to be schedulable. $W(t)$ represents the cumulative amount of CPU time that is consumable by all task releases between time 0 and t.

$$H(t) = \sum_{i=0}^{n} \left\lfloor \frac{t - D_i + T_i}{T_i} \right\rfloor C_i, \qquad W(t) = \sum_{i=0}^{n} \left\lceil \frac{t}{T_i} \right\rceil C_i$$

Figure 1 illustrates the functions for an example task set. Our feasibility analyses is based on the behaviour of $H(t)$ and $W(t)$ and on the observations earlier proved by Baruah *et al˙* [3]:

> "If for any interval with length L, all work load offered during $[0, L]$ can be resolved before or at L, then this can be concluded for any arbitrary time interval $[t, t + L]$."

Therefore all tasks in Ω are released simultaneously at $t = 0$, in which case they will produce the largest response time. If the tasks in Ω can make their deadlines from $t = 0$, they can make their deadlines from any point in time.

Figure 1 shows the functions $H(t)$ and $W(t)$. Both are used for schedulability analysis of the task set Ω. Note that the vertical distance between

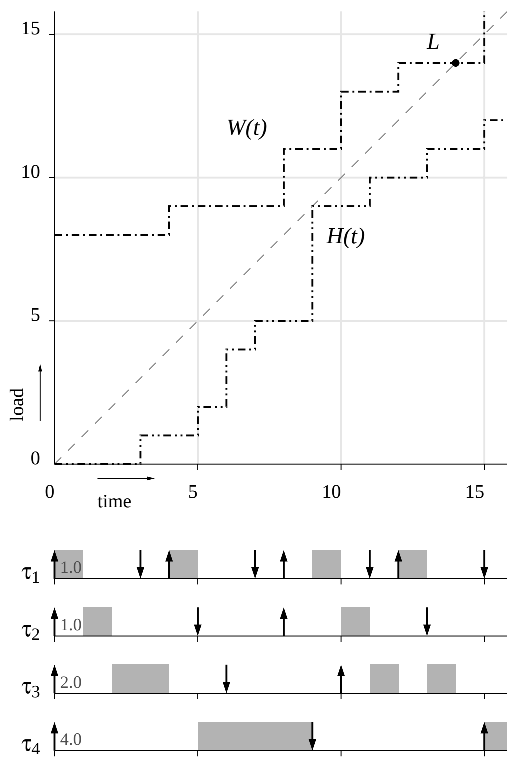

Figure 1: Example task set and its EDF schedule with processor demand $H(t)$ and workload functions $W(t)$.

$W(t)$ and the diagonal in the graph represents the amount of work still to do in released tasks. At point L, there is no more work to do and the system becomes idle. $H(t)$ represents the amount of work that must be finished. If $H(t)$ crosses the diagonal, then more work would have to be finished than there is time available. The schedulability analysis tracks $W(t)$ and $H(t)$ until either $W(t)$ *touches* the diagonal or $H(t)$ *crosses* it. If $H(t)$ crosses the diagonal, the task set is not schedulable. If $W(t)$ touches before $H(t)$ could cross, the task set is schedulable. The example task set Ω is thus schedulable. Task sets can be constructed in which neither $W(t)$ nor $H(t)$ reaches the diagonal. The schedulability analysis, therefore, traces these functions for only a predetermined maximum number of steps and rejects a task set if this maximum is reached.

The scheduler manages the set of admitted tasks using two queues and a stack. The *Wait Queue*, holds tasks awaiting their release. When a task gives up the processor or reaches its deadline, it is put on this queue, from which it will be transferred to the next queue when it is released. The *Released Queue* holds processes that have been released but have not yet

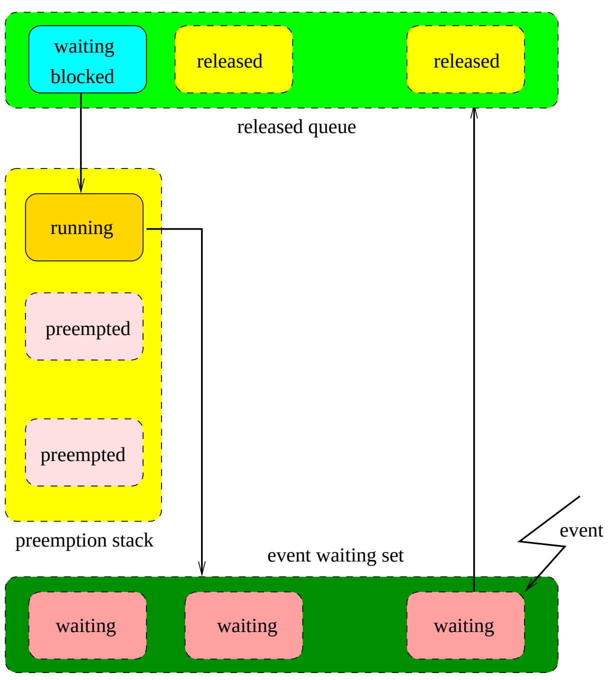

Figure 2: Transaction system: released queue, preemption stack and event waiting queue

run. This queue is maintained in deadline order, earliest deadline first. The *Run Stack* holds the tasks that have already run; the currently running task is at the top of the stack and the tasks below it were preempted by the tasks immediately above them.

The *Release Timer* goes off when a task in the Wait Queue needs to be released. Released tasks are then transferred to the Released Queue. When a task gets to the front of the Released Queue or when a task is popped from the Run Stack, the deadlines of the task τ_h at the head of the Released Queue and the running task at the top of the Stack τ_r are compared. If $d_h < d_r$, τ_h is removed from its queue and pushed onto the Run Stack. If both Run Stack and Released Queue are empty, best effort processes are scheduled.

Nested Critical Sections (NCSs) [5] can be used in tasks for the use of shared resources. What we need to do is to specify NCSs and their durations. NCSs in combination with inheritance have been used in other protocols such as the well-known Priority Ceiling (PC) protocol [6] and the Stack Resource protocol [7]. An in-depth overview is given by Rajkumar [8]. Sha *et al˙* [9] give an overview of how to generalise PC for DM under blocking and they present how to use this protocol for a practical system implementation. The protocol we present in this chapter has similarities with PC and SR. PC is from the class of fixed priority protocols while SR belongs to the class of

Table 2: Specification of Ω_2

Ω_2	D_i	T_i	C_i	ρ_i
τ_1	4	5	1	0.9{ a B }
τ_2	5	8	1	0.8{ a 0.2{ B 0.1{ C }}}
τ_3	6	10	2	0.2{ b } 1.7{ c 1.3{ b }}
τ_4	9	9	3	1.8{ a c }

dynamic priority protocols.

A *resource specification* ρ of a task τ is specified according to the following syntax:

ρ_{list} : *float* '{' R_{list} ρ_{list} '}' | ε
R_{list} : R_{list} R | ε
R : 'a' . . . 'z' | 'A' . . . 'Z'
$float$: floating-point number

in which the non-capital resource R indicates a read access to a shared resource, while a capital resource R indicates an exclusive-access to it. An example of a task set with a resource specification is given in table 2.

Task 1 has a period of 5 seconds, a deadline of 4 seconds (if it is released at t, its deadline is at $t + 4$ and its next release is at $t + 5$; it needs at most 1 second of CPU time between release and deadline. Resource a is shared by tasks 1, 2 and 4. All tasks only require read access to the resource, so no restrictions on the schedulability of these tasks exist. Resource b/B is shared by tasks 1, 2 and 3. Task 1 needs exclusive access to B for 0.9 time units, it also holds read resource a. Task 3 needs shared-read access to resource b for 0.2 time units and again for 0.13 time units while holding resource c for 1.7 time units.

The principle behind scheduling a task set with shared resources is that a released tasks stays on the Released Queue if it needs resources that are already in use by one of the task in the Run Stack, even if such a released task has a shorter deadline. Therefore, once a task τ_r is on the Run Stack, it will never claim a resource already held by another, preempted, task. Such a task τ_r would simply not have been scheduled.

We enforce this by *deadline inheritance*, which is similar to Priority Inheritance, introduced by Sha *et al˙* [6]. Every resource, R is assigned an *inherited deadline* $D_R = min_{\rho_i \in \Omega}\{D_i \mid R \in \rho_i\}$, the minimum of the deadline of all tasks using R, where ρ_i denotes the set of tasks in use by task τ_i. If $\rho'_i \subseteq \rho_i$ denotes the subset of resources in use by τ_i, then the inherited deadline of τ_i is $\Delta'_i = min_{\{R \in \rho'_i\}}\{\Delta_R\}$. The minimal inherited deadline of τ_i is reached if all resources are used: $\Delta_i = min_{\{R \in \rho_i\}}\{\Delta_R\}$. A task's Δ' thus changes as the task acquires and releases resources.

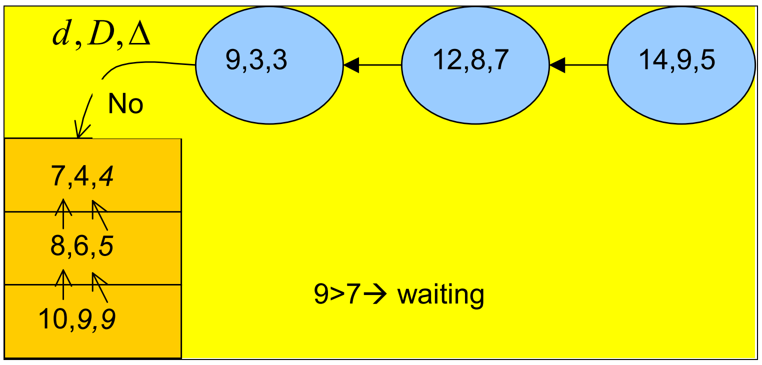

Figure 3: Example Run Stack (rectangles) and Released Queue (ellipses); the arrows indicate the partial order between the parameters. No preemption is allowed because $D_h < \Delta'_r$ $(9 < 7)$ is not true.

Each released task is now characterised by the triple d, D, Δ', where d is the current *absolute* deadline.

Earlier, we presented the EDF scheduling rule that the task τ_h at the head of the Released Queue would move to the top of the Run Stack if its d_h was less than d_r of the task τ_r on top of the Run Stack. A released task with an earliest deadline will preempt the currently running task. Now we modify that rule to:

$$\tau_h \text{ preempts } \tau_r \text{ iff } \ d_h < d_r \wedge D_h < \Delta'_r$$

Figure 3 shows an example Run Stack (rectangles) and Released Queue (ellipses). At this time, the task at the head of the Released Queue may not preempt the one on top of the Run Stack because ($9 < 7 \wedge 3 < 4$ is false). For every task τ_i, $\Delta'_i <= D_i$ and, because of the scheduling rule, for a task τ_h higher on the Run Stack than another task τ_l, $D_h < \Delta'_l$. There is, therefore, a total ordering from D to Δ' to D, etc. up and down the Run Stack. This is indicated by the arrows in figure 3. This ordering, plus the definition of Δ, establishes the property that the currently running task, which is at the top of the Run Stack, will not attempt to acquire any resources held by preempted tasks, which are further down in the Run Stack. This is because, if they held such resources, their Δ would be less than or equal to the D of the running task and this the scheduler does not allow.

A second property is that there is no transitive blocking, because a process that is blocked due to shared resource usage only has to wait for this only blocker to release the resource. Stated more formally, task τ_h at the head of the Released Queue is blocked by τ_s on the Run Stack despite having a higher priority ($d_h < d_s$) because $D_h \geq D_s$ prevents the preemption of τ_s. It can be proved that under these circumstances $\Delta'_s \leq D_h < D_s$. Due to the full ordering property of D's and $\Delta's$ on the stack the number of blockers has a maximum of 1. This is also a property of the Priority Ceiling protocol [6], the first protocol that to introduce static priority inheritance, similar to our static deadline inheritance.

The schedulability analysis is only moderately more complex with resource sharing. The processor demand and workload functions do not change, because the work that needs to be done and when it needs to be done is the same. But we do have to take into account now that *one* task, not more, may *block* another's access to the CPU. To illustrate the process, we use the same specification as the one given before, this time in a more convenient mathematical notation, in table 3. Blocking can be represented graphically by adding spikes at time t to the processor demand function, as illustrated in figure 4.

The height of a spike is the result of calculating the *blocking times* of a maximum blocker from the resource specification: at $t = 4$, τ_1 reaches its deadline. Before reaching the deadline, it may have been prevented from being scheduled by a task with a longer deadline, but holding a resource that τ_1 might need. For τ_1, the amount of slack in the schedule needed is 1.3 time units, because that is how long τ_3 might hold resource b. Similarly, at $t = 5$, τ_2 needs 1.8 time units of slack to compensate for τ_4, which might hold resource c, preventing τ_2 from being scheduled. The maximum potential blocking is given by $C_B(t) = \max_\Omega\{C'_{\tau'} \mid \Delta'_{\tau'} \leq t < D_\tau\}$ where τ' is a nested critical section, C' its cost, $\Delta'_{\tau'}$ its inherited level and t is the length of the interval over which blocking has to be computed. The new admission rule calculates these potential blockings as spikes on the processor demand function at the expiration times of deadlines and declares a task set inadmissible if one of the spikes crosses the diagonal. If there are no shared resources, there is no blocking (there are no spikes), and the schedulability test reduces to the normal preemptive-EDF schedulability test. If there is one resource, shared full-time by all tasks, the schedulability test reduces to the non-preemptive schedulability test. This schedulability test spans the range between the extremes of completely preemptive and completely non-preemptive scheduling.

A more formal consideration for feasibility analyses under EDF with deadline inheritance is given in Jansen & Laan [2].

Table 3: The Δs are converted to tuples consisting of *inherited deadline* and *usage time*.

Ω	D_i	T_i	C_i	*resources* $\rightarrow \Delta$s
τ_1	4	5	1	0.9{ a B } $\rightarrow$ (4,0.9)
τ_2	5	8	1	0.8{ a 0.2{ B 0.1{ C }}} $\rightarrow$ (∞,0.8)(4,0.2)(5,0.1)
τ_3	6	10	2	0.2{ b } 1.7{ c 1.3{ b }} $\rightarrow$ (4,0.2)(5,1.7)(4,1.3)
τ_4	9	9	3	1.8{ a c }$\rightarrow$ (5,1.8)

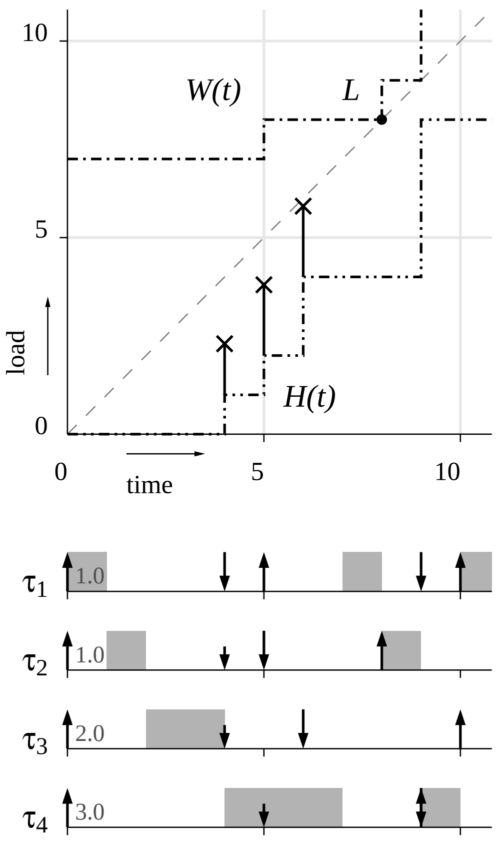

Figure 4: Ω_2 is feasible under EDF with nested critical sections

3 Application interface

Our RT scheduler only needs a small amount of dynamic memory space. Feasibility analyses can be carried out off-line and outside the kernel so that they do not burden the kernel's limited resources too much. After an analysis has been successfully carried out, the task set can be loaded on the target processor and run. Its original task specification with shared resources has automatically been converted to one in which the Δ tuples replace the shared resource specification, as exemplified in table 3.

After the specification of a task set only very little additional work is needed at the application level. A task can only *block* before it starts running while a *running* task may always enter a critical section undisturbed and start using its resources. On entering a critical section the application must notice the system, which section is entered so that the system can adapt the Δ'_r level. On leaving the critical section the system is again notified and it will restore the previous value of Δ'_r. Of course the duration of the critical section may not exceed it specified length. This can be checked automatically by a timer.

Note that at application level no semaphores are needed to force synchronisation because no blocking is possible while running. Also blocking before running is limited since the number of running blockers for the task at top of the Released Queue is at most one. Deadlock cannot occur.

It is possible to specify tasks as so called *RT transactions*, which have only one critical section with runtime length. Then, all synchronisation can be done by the scheduler itself. An example is given in table 4 and we refer to these types of tasks as *RT transactions*. In such a case the inherited deadline Δ will not change during runtime and therefore synchronisation can be fully handled by the scheduler. No synchronisation is needed from the user, preemption, scheduling mutual exclusion of shared resources and resource synchronisation is completely done at system level, while the application programmer does not need to be aware of this and may think in terms of a 'run-to-completion' model.

Table 4: Conversion of resources to *inherited deadlines* for RT transactions

Ω	D_i	T_i	C_i	*resources* $\rightarrow \Delta$s
τ_1	4	5	1	1{ a B } $\rightarrow$ (4,1)
τ_2	5	8	1	1{ a B C }$\rightarrow$ (4,1)
τ_3	6	10	2	2{ b c }$\rightarrow$ (4,2)
τ_4	9	9	3	3{a c }$\rightarrow$ (5,3)

4 Tools and implementation

We have developed a testing tool for feasibility analysis and offer a graphical web interface [10] for off-line feasibility analysis. This analysis is suitable for our EDF model with nested critical sections. It asks as input a task specification according to table 2 and it produces a graphical output according to figure 4. The tool can also handle analysis of resource using task sets scheduled with other protocols like Rate Monotonic [1], Deadline Monotonic [4] and the Stack Resource protocol [7]. These protocols are beyond the scope of this paper.

Currently we have three EDFI implementations: in Linux-RT, in Plan 9 and in RTY. They have in common that one timer controls the RT portion of the scheduler: the Release Timer goes off when a task in the Wait Queue must be released. If that task gets to the front of the Release Queue, a scheduling decision is made, otherwise, the current task continues running: τ_h preempts τ_r iff $d_h < d_r \wedge D_h < \Delta'_r$, according to scheduling condition given earlier. When the Deadline Timer goes off, the running task has used up its quantum and the processor is taken away from it until the next release. We also raise an exception in the process.

RT-Linux

Our first target for scheduling according to the principles as described in this paper was RT-Linux [11], and Linux RTAI [12]. The existing scheduler could easily be replaced by ours. Scheduling overhead was mainly due to the ordering of tasks in the Ready Queue and it turned out to be in the order of some percent of the time needed for switching a tasks; the main overhead was due to task switching.

Plan 9

Our second target was Plan 9, which needed to be adapted from a general-purpose distributed operating system to a kernel for RT applications. We implemented the scheduler in Plan 9. This was a fairly straightforward process, although we had to change the behaviour of spin locks in the kernel slightly. A process is now allowed to finish its critical section before being subject to scheduling. None of the spin locks hold the CPU longer than about 50μs and they are small enough to be disregarded by the scheduler.

The interesting part about the implementation is the use of a file system to control the system. In the default mount point of `/dev/realtime` we find three files, `clone`, `resources`, `time`, and a directory: `task`. Existing tasks are represented by files (whose names are numbers) in the `task` directory. A new task is created by opening the file `clone`, which then behaves like the corresponding (new) file in the `task` directory. The main loop for a typical real-time process looks as follows:

```
char *clonedev = "/dev/realtime/clone";

void
processvideo(void){
   int fd;

   fd = open(clonedev, ORDWR);
   if (fprint(fd, "T=33ms D=20ms C=8ms procs=self admit") < 0)
      sysfatal("%s: admission: %r", clonedev);
   while (processframe())
      fprint(fd, "yield");
   fprint(fd, "remove");
   close(fd);
}
```

This sequence creates a new task by opening `/dev/realtime/clone`, sets period, deadline and cost and puts the running process into the process group of the task. It then asks the scheduler to admit the new task by running the schedulability test. If the write succeeds, the task was admitted.

The main loop processes a video frame and then gives up the processor (`yield`) while waiting for the next frame. When the application has finished, it removes the task from the system and exits.

In order to avoid additional introduction of naming and runtime lookup of NCSs, we also transformed the specification of NCSs to one that is more convenient to handle and implement, but less intuitive as the one presented in table 2.

Real Time eYes OS

Currently we are implementing our scheduler in RTY, the OS for a sensor node in our Eyes project [13]. RTY runs on limited hardware with a TI MSP430 RISC processor. The small board is hosting 2 UARTs, AD-converters, analog comparators, timers and a radio unit. There is 60Kbytes of flash memory and 2Kbytes of RAM. The total RAM usage of RTY is less than 512 bytes, which is mainly used for communication over the radio and the serial interface. The kernel itself needs about 80 bytes (without the stack). This does not beat TinyOS [14] that needs about 50 bytes of RAM, however we offer hard RT services, which TinyOS does not.

Because of given limitations RTY uses RT *transactions* tasks, which is the simplest form for a resource using RT task. Such a resource is for instance a communication interface. When using transactions, all needed resources are claimed during runtime. This may limit preemption possibilities of the system, however it simplifies the programmers interface even further: a user does not need to synchronise the use of resources; this is entirely done by the scheduler. Feasibility analysis and admission control is done off-line. Whether we can afford a timer to indicate the expiration of a deadline is still under discussion.

Another important issue to be considered in more detail is the relation between Quality of Service, and energy usage of a sensor node. Our approach is to lower the clock-rate as far as possible in order to save energy. However, a consequence of a slower clock is the increase of the run time costs C of our tasks, which in turn, can cause the excess of deadlines of these tasks. Our feasibility analyses can easily detect this excess and it can help to find the lowest clock rate with which the feasibility is still guaranteed, thus offering the best QoS for the lowest price.

We have had some lively debates over whether it is worthwhile to have a RT scheduler that can manage shared resources. Most of the RT applications we considered do not have any resources that are shared. However, some are non-preemptive and we can use the presented feasibility analysis technique by modelling non-preemption with transactions sharing a single resource. We use this model for admission control in RT communication in our real-time network RTnet [15] and currently also for Quality of Service control experiments with Bluetooth [16]. Another application where we could use the presented technique is Clockwise [17], a mixed-media file system, originally based on Jeffay's theory [18], in which the presented scheduling algorithms were used for non-preemptable RT disk scheduling.

The battle about whether or not to include support for resource sharing in our RT scheduler was won by the resource-sharing camp: the schedulability test is straightforward and the run-time complexity is practically O(1): only the queue insertions are not constant-time operations, but the queues are invariably very short. In addition, the scheduler prevents resource contention from causing gratuitous context switches and it is completely deadlock free. Finally, the same scheduler can trivially be used for preemptive or non-preemptive EDF scheduling.

5 Conclusion

We have presented a real time scheduling/dispatcher that can be can handled by very minimal means, even in the case when shared resources are used. By extending EDF with deadline inheritance, an elegant feasibility algorithm has been derived that is used for on-line admission control of new tasks. An additional advantage of the proposed method is that synchronisation of tasks due to resource usage is almost completely shifted to the system level, which results in a simple user interface: the application programmer may have the illusion of a run-to-completion semantic for each task and the system is deadlock free. Nevertheless the system preempts running and resource-using tasks in favour of tasks with shorter deadlines but avoids gratuitous task switching. We have implemented our scheduler/dispatcher in several, quite different operating systems. From this we can conclude that our method is easy to implement, needs few code and a negligible amount additional RAM, while run-time overhead is low. This makes our scheduling technique ideal for use in a large variety of RT systems, for processing as well as for communications. In particular it is appropriate for those systems that have to work with limited resources.

Acknowledgements

Acknowledgements are due to Ferdy Hanssen for carefully reading this document, for discussions and for implementing the RT-Linux implementation and to Job Mulder for a first implementation of EDFI in the RTY kernel.

References

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[2] P. G. Jansen and R. Laan, "The stack resource protocol based on real-time transactions," *IEE Proceedings Software*, vol. 146, no. 2, pp. 112–119, Apr 1999.

[3] S. K. Baruah, A. K. Mok, and L. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proceedings of the Real-Time Systems Symposium*, Dec 1990, pp. 182–190.

[4] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Hard Real-Time Scheduling: The Deadline Monotonic Approach," in *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[5] E. W. Dijkstra, *Cooperating sequential processes.* Academic Press, 1968, pp. 43–112.

[6] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep 1990.

[7] T. P. Baker, "Stack-based scheduling of real-time processes," *The journal of real-time systems*, vol. 3, no. 1, pp. 67–99, 1991.

[8] R. Rajkumar, *Synchronization in Real-Time Systems, A priority Inheritance Approach.* Kluwer Academic Press, 1991.

[9] L. Sha, R. Rajkumar, and S. Sathaye, "Generalised rate-monotonic scheduling theory: A framework for developing real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 68–82, Jan 1994.

[10] "Real-Time feasibility analysis tool web site," http://wwwes.cs.utwente.nl/feas/.

[11] "Real-Time Linux web site," http://www.rtlinux.org/.

[12] "Real-Time Application Interface for Linux web site," http://opensource.lineo.com/rtai.html.

[13] "The eyes project," http://eyes.eu.org/publications/d1.2.pdf.

[14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System architecture directions for networked sensors," in *Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 93–104.

[15] J. Scholten, P. G. Jansen, F. T. Y. Hanssen, and T. Hattink, "An In-Home network architecture for Real-Time and Non-Real-Time communication," in *IEEE Region 10 International Conference on Computers, Communications, Control and Power Engineering (TENCON).* Beijing, China: IEEE Computer Society Press, Los Alamitos, California, Oct 2002, pp. 728–731.

[16] J. Haartsen, M. Naghshineh, J. Inouye, O. Joeressen, and W. Allen, "Bluetooth: Vision, goals, and architecture," *Mobile Computing and Communications Review*, vol. 2, no. 4, pp. 38–45, Oct 1998.

[17] P. Bosch, S. J. Mullender, and P. G. Jansen, "Clockwise: A Mixed-Media file system," in *IEEE Int. Conf. on Multimedia Computing and Systems (ICMCS)*, vol. II. Firenze, Italy: IEEE Computer Society Press, Los Alamitos, California, Jun 1999, pp. 277–281.

[18] D. K. Jeffay, Stanat, and C. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," in *Proc. of the* 12^{th} *IEEE Real-Time Sys. Symp.*, 1991, pp. 129–139.