Performance comparison between the SRP and DFP synchronization protocols in MaRTE OS

Marina Gutiérrez
Universidad de Cantabria
gutierrezlm@unican.es

Alan Burns
University of York
alan.burns@york.ac.uk

Mario Aldea
Universidad de Cantabria
aldeam@unican.es

Michael González
Universidad de Cantabria
mgh@unican.es

January 17, 2013

Abstract

The Deadline Floor Protocol (DFP) is a mutual exclusion synchronization protocol designed as an alternative to the Baker's Stack Resource Protocol (SRP) to access shared resources in a system scheduled under the Earliest Deadline First (EDF) policy. We have implemented both protocols in the real-time operating system MaRTE OS and we have compared their performance. Overall, DFP is easier to implement and performs better than SRP for the same data structure, a doubly-linked list (DLL). More significantly, there is a more efficient data structure (the binary heap) that cannot be used with SRP and outperforms the DLL with both protocols.

1 Introduction

The Earliest Deadline First (EDF) is the most widely studied optimal dynamic scheduling algorithm for uniprocessor real-time systems [1]. For realistic programs, tasks must be allowed to exchange data by sharing resources that must be accessed under mutual exclusion.

With EDF scheduled systems, access to such resources is usually controlled by the use of Baker's Stack Resource Protocol (SRP) [2] [3], but recently Burns [4] introduced a new protocol called Deadline Floor Protocol (DFP). The DFP has all the key properties of SRP and leads to the same worst-case blocking. Both protocols are briefly reviewed in Section 2. The purpose of this work is to compare their performance.

MaRTE OS [5] is a real-time operating system developed by the Computers and Real-time group at the University of Cantabria. Most of its code is written in Ada [6] with some C and assembler parts. MaRTE OS provides support for Ada and C/C++ concurrent applications. In the case of C/C++ applications, they can make use of the POSIX/C interface provided by MaRTE OS [7].

MaRTE OS supports many advanced real-time features to be used by Ada or C applications, not present in the POSIX standard. In particular, it implements the EDF scheduling policy along with the Baker's Stack Resource Protocol.

In the work presented in this report, we have added to MaRTE OS support for the Deadline Floor Protocol in order to perform a fair comparison in the same system between both protocols (EDF and SRP) when used by POSIX/C applications.

Details of the implementation of both protocols can be found in Sections 3 and 4. In Section 5 we have made a comparative analysis between the SRP and DFP implementations and conclusions are contained in Section 6.

2 Resource Sharing Policies

2.1 System Model

Our model of a hard real-time system comprises a set of n real-time tasks $\{\tau_1, \tau_2, ..., \tau_n\}$ executing on a uniprocessor, each task consists of a potentially unbounded stream of jobs which must be completed before their deadlines. Let τ_i indicate any given task of the system, and let j_i indicate any given job of τ_i.

When tasks are periodic all their jobs have a regular inter-arrival time T_i , we call T_i the period of τ_i . If a job for a periodic task arrives at time t, then the next job of τ_i must arrive at $t + T_i$. For sporadic tasks each task has a minimum interarrival time that is also called T_i, so the next job of τ_i must arrive not earlier than $t + T_i$.

Each job of task τ_i has an execution time that is bounded by the task's worst-case execution time, C_i . Each job of τ_i has the same relative deadline which equals the task's relative deadline D_i.

If a job of τ_i arrives at time t, the required worst-case execution time C_i must be completed within D_i time units, and the absolute deadline of this job is $d_i = t + D_i$. The term deadline refers to an absolute deadline of some job in the system.

Contained within the system are m shared resources $(r_1, ..., r_m)$. Tasks may access (under mutual exclusion) these resources, but we make no assumption as to when each job accesses these shared resources during its execution. We do assume however that tasks do not self-suspend whilst accessing a resource. Resources can make use of other resources and hence nested relationships are possible, however, resources must be used in a strictly nested way.

The inclusion of shared resources in the system model implies that tasks may suffer *blocking* – which must be taken into account in the scheduling analysis.

According to the EDF scheduling algorithm, in the absence of blocking, the job with the earliest absolute deadline has the highest priority and will be executed on the processor. If more than one job has the same deadline then they are scheduled in FIFO order; the one that has been in the system the longest time will execute first. At any time, a released job with an earlier absolute deadline will preempt the execution of a job with a later absolute deadline. When a job completes its execution the system chooses, for execution, the oldest pending (released) job with the earliest deadline.

2.2 Stack Resource Policy (SRP)

Given an application defined by a set of tasks $(\tau_1, \tau_2, ..., \tau_n)$, a set of resources $(r^1, r^2, ..., r^m)$ and a task-resource access relation, $\mathcal{A}$ the SRP [2] [3] is defined as follows:

1. Each task τ_i is assigned a preemption level π_i. Under EDF scheduling, the preemption level of a task correlates inversely to its relative deadline, ie. $\pi_i < \pi_j \Leftrightarrow D_i > D_j$.

2. Each resource, r^j, is assigned a ceiling preemption level denoted as Π^j which is set equal to the maximum preemption level of any task that may access it.

$$\Pi^j = max\{\pi_j : \tau_j \in \mathcal{A}(r^i)\}.$$

3. When a task τ_i accesses resource r^j the task's preemption level is compared with resource's preemption level. If the resource's preemption level is higher than task's preemption level, then the task inherits the resource's preemption level; that is $\pi_i \leftarrow max\{\pi_i, \Pi^j\}$.

4. When this task frees the resource its preemption level immediately returns to the value that it had before entering the resource.

5. A task can only be chosen to execute if its preemption level is strictly higher than the maximum of the preemption level of the running task and the preemption levels of the resources currently held in the system.

2.3 Deadline Floor Inheritance Protocol (DFP)

Given an application defined by a set of tasks ($\tau_1, \tau_2, ..., \tau_n$), a set of resources ($r^1, r^2, ..., r^m$) and the task-resource access relation, $\mathcal{A}$ the DFP [4] is defined as follows:

1. Each resource, r^i, has a relative deadline D^i given by:

$$D^i = min\{D_j : \tau_j \in \mathcal{A}(r^i)\}$$

2. When a task τ_j released at time s accesses resource r^i at time t (so $s < t$) its relative deadline is immediately reduced to D^i, and as a result its active absolute deadline is also (potentially) reduced; that is $d_j \leftarrow min\{t + D^i, d_i\}$.

3. When this task frees the resource its deadline immediately returns to the value that it had before entering the resource.

The scheduling rules remain the same as in any other EDF scheduled system: a task can only be chosen to execute if its absolute deadline is strictly earlier than the current absolute deadline of the running task.

3 SRP Implementation in MaRTE OS

3.1 Basic EDF Implementation

MaRTE OS uses a hierarchical scheduler with two levels. The base scheduler uses fixed priorities as defined in the POSIX standard. Each task (called thread in POSIX) is assigned an integer called the priority that is used as the primary criterion for scheduling. A higher priority task will always preempt a lower priority task. Within each priority level POSIX allows different behaviours (such as FIFO or round-robin ordering). MaRTE OS adds a new secondary scheduler that uses the EDF policy for tasks of the same priority. This hierarchical scheduling model is also supported in Ada 2005 for specific priority bands [8].

In this work we focus on a comparison of the SRP and DPF protocols for EDF scheduling, so all our experiments and task model assume that all the tasks share the same base priority and are scheduled just according to plain EDF rules.

The main data structure managed by the MaRTE scheduler is the ready queue. The ready queue is an array of queues implemented with doubly-linked lists, a queue for each system priority. Each queue holds the set of tasks of the same priority and for tasks with an EDF scheduler this set is where the EDF rules apply.

The doubly-linked list (DLL) is a linked data structure that consists of a set of sequentially linked nodes. Each node contains two links, that are references to the previous and to the next

node in the sequence of nodes. General insert and remove operations are $O(n)$. Removing the head or the tail is $O(1)$.

When a task becomes ready, it must be added to the queue that corresponds to its priority. The particular position occupied by the task in the queue depends on its scheduling policy and parameters. For example, a just activated fixed priority FIFO task will occupy the tail of the queue that corresponds to its priority.

Besides the priority, EDF tasks in MaRTE OS have two new scheduling parameters to implement the EDF policy: the relative and absolute task deadlines.

At each activation, an EDF task is assigned a new absolute deadline. EDF tasks are sorted in the queue that corresponds to its priority according to their absolute deadlines, more urgent deadlines first.

As it will be described in Section 3.2, when SRP is in use the ordering of EDF tasks gets more complex, being based not only on their absolute deadlines but also on their preemption levels.

3.2 EDF scheduling in presence of SRP

As explained in 2.2 a new scheduling restriction is added by SRP to the basic EDF scheduling: "a task can only be chosen to execute if its preemption level is strictly higher than the maximum of the preemption level of the running task and the preemption levels of the resources currently locked in the system".

In fact this restriction forces a deep change in the ordering criterion of the ready queue. While in plain EDF, tasks with the same priority are ordered in their queue according to their absolute deadlines, when using SRP two tasks (τ_a and τ_b) in the ready queue must be order according to this order relation:

$$\tau_a \leq \tau_b \Leftrightarrow \begin{cases} d_a \leq d_b & if\ \ \tau_b\ \ has\ \ no\ \ resources\ \ held \\ d_a \leq d_b\ \ and\ \ \pi_a \geq \pi_b & if\ \ \tau_b\ \ has\ \ some\ \ resources\ \ held \end{cases} \quad (1)$$

Consequently, looking for the right place of a task in the ready queue is more complex with SRP than when using a plain EDF scheduling. With SRP, to find the correct place of a newly activated task in the queue that corresponds to its priority we must start from tail to head comparing the new task's absolute deadline and preemption level with the parameters of the other tasks in the queue according to expression (1). The new task will be placed just before the first task τ that does not verify $\tau_{new} < \tau$. The search of the new task's position is an $O(n)$ operation. A example of task ordering in the ready queue can be found in Appendix A.

At this point it is necessary to introduce the concept of "Total order". An order relation $\leq$ is a total order if it verifies the properties [9]:

$$\begin{array}{ll} if\ \ a \leq b\ \ and\ \ b \leq a \Rightarrow a = b & (antisymmetry) \\ if\ \ a \leq b\ \ and\ \ b \leq c \Rightarrow a \leq c & (transitivity) \\ a \leq b\ \ or\ \ b \leq a & (totality) \end{array} \quad (2)$$

One interesting behaviour of a total order relation is that it leads to a unique linear ordering among the elements added to an ordered set. This order is always the same independently of the order in which the elements had been added to the set.

The order criterion in (1) produces a "non-total order" as can be seen in the following counter example. Consider tasks τ_a and τ_b with active jobs with deadlines $d_a = 11$ and $d_b = 6$, preemption levels $\pi(\tau_a) = 4$ and $\pi(\tau_b) = 3$ and where the only tasks with resources locked is τ_a. Using the order criterion in (1) we have $\tau_a \leq \tau_b \rightarrow false$ and $\tau_b \leq \tau_a \rightarrow false$ what does not verify the totality property showed in (2).

The fact that SRP leads to a non-total order has a big relevance as it will be shown in Section 4.1.

3.3 Accessing a resource with SRP

Like in any other POSIX operating system, access to shared resources in MaRTE OS is implemented with a mutex interface. So from now on, we will use the mutex terminology to refer to shared resources.

Procedures `Running_Task_Locks_Mutex` and `Task_Unlocks_Mutex` provide the basic kernel support for the implementation of POSIX mutexes. Mutual exclusion among Ada tasks accessing protected objects also relays on the Ada Run-Time System invoking these two procedures.

Figure 1 shows the details of the SRP implementation on MaRTE OS. We now describe the full process of accessing and freeing a resource with this protocol.

The ready queue must be ordered according to the rules of an EDF scheduled system with SRP exposed in (1). This comparison is performed using the function < operating on two tasks.

When a task accesses a mutex the procedure `Running_Task_Locks_Mutex` is executed. First we increase the `Num_Mutex_Owned` by the task, and we store the task's current preemption level in the `Owner_Preemption_Level` field of the mutex. Then, if the mutex's preemption level is higher than task's preemption level, the task's preemption level is set equal to the mutex's preemption level. Although the scheduling parameters of the task may have changed, possibly making the task even more urgent, the task is currently the most urgent one, i.e., the first one in the ready queue, so there is no need to reorder the ready queue.

When this task frees the mutex the procedure `Task_Unlock_Mutex` is executed. First we decrease the `Num_Mutes_Owned` by the task and restore the task's previous preemption level which was stored in `Owner_Preemption_Level`. At this time the task's scheduling parameters could change, possibly making it less urgent, and therefore the ready queue must be reordered. The procedure `Reorder_And_Dispatch` reorders the task in the ready queue and, in the case the task is no longer at the head of the queue, this procedure performs a context switch preempting the task that has just unlocked the mutex.

In order to find the right place in the ready queue, procedure `Reorder_And_Dispatch` compares the task's scheduling parameters (absolute deadline and preemption level) with the parameters of the other tasks in the queue corresponding to its priority. As explained in Section 3.2, this comparison is performed from the tail to the head of the queue to find the correct place for the task obeying the scheduling rules of SRP. The comparison of the task parameters is done by function < shown in Figure 1.

3.4 SRP Interface

Although EDF nor SRP are in the POSIX standard, both are included in MaRTE OS as extensions to the POSIX standard. A POSIX-like interface is provided to applications to manage the SRP specific scheduling parameters.

The details of the interface can be seen in Figure 2. All functions return an integer as an error code, but it has been omitted for simplicity. Functions `Pthread_Mutexattr_Setpreemptionlevel` and `Pthread_Mutexattr_Getpreemptionlevel` and `Pthread_Attr_Setpreemptionlevel` and `Pthread_Attr_Getpreemptionlevel` set and get the preemption level statically through the attributes object used to create mutexes or threads/tasks respectively. Functions `Pthread_Mutex_Setpreemptionlevel` and `Pthread_Mutex_Getpreemptionlevel` and `Pthread_Setpreemptionlevel` and `Pthread_Getpreemptionlevel` dynamically set and get the preemption level of mutexes and threads/task, respectively.

SRP Implementation

```
type Task_Preemption_Level;  -- Integer Type
HIGHEST_CEILING_PRIORITY := constant Locking_Policy;

type Mutex is new Mutex.Element with record
  ...
  Preemption_Level :  Task_Preemption_Level : = First;
  Owner_Preemption_Level :  Task_Preemption_Level : = First;
end record;

type TCB is new Task_Control_Block.Element with record
  ...
  Deadline :  HWTime : = 0; -- Not SRP specific
  Preemption_Level :  Task_Preemption_Level : = First;
end record;
```

```
procedure Running_Task_Locks_Mutex (Mutex) is
  ...
  Task.Num_Mutex_Owned ++;
  Mutex.Owner_Preemption_Level : = Task.Preemption_Level;
  if Task.Preemption_Level < Mutex.Preemption_Level then
    Task.Preemption_Level := Mutex.Preemption_Level;
  end if;
  ...
end Running_Task_Locks_Mutex;
```

```
procedure Task_Unlocks_Mutex (Task, Mutex) is
  ...
  Task.Num_Mutex_Owned --;
  Task.Preemption_Level := Mutex.Owner_Preemption_Level;
  Reorder_and_Dispatch (Task);
  ...
end Task_Unlocks_Mutex;
```

```
function < (Left_Task, Right_Task) return Boolean is
  ...
  return Left_Task.Deadline < Right_Task.Deadline
    and then (not Mutexes_Owned or else
      Left_Task.Preemption_Level > Right_Task.Preemption_Level);
end <;
```

Figure 1: SRP Implementation in MaRTE OS.

SRP Interface

```
function Pthread_Mutexattr_Setpreemptionlevel
  (Attr:  access Mutex_Attributes; Level :  in Task_Preemption_Level)
function Pthread_Mutexattr_Getpreemptionlevel
  (Attr:  access Mutex_Attributes; Level :  access Task_Preemption_Level)

function Pthread_Mutex_Setpreemptionlevel
  (M: access Mutex; Level :  in Task_Preemption_Level)
function Pthread_Mutex_Getpreemptionlevel
  (M: access Mutex; Level :  access Task_Preemption_Level)

function Pthread_Attr_Setpreemptionlevel
  (Attr:  access Pthread_Attributes; Level :  in Task_Preemption_Level)
function Pthread_Attr_Getpreemptionlevel
  (Attr:  access Pthread_Attributes; Level :  access Task_Preemption_Level)

function Pthread_Setpreemptionlevel
  (Id:  in Task_Id; New_Level :  in Task_Preemption_Level)
function Pthread_Getpreemptionlevel
  (Id:  in Task_Id; Level :  access Task_Preemption_Level)
```

Figure 2: SRP interface in MaRTE OS.

4 DFP Implementation in MaRTE OS

4.1 EDF Scheduling in Presence of DFP

As it is said in 2.2, DFP doesn't introduce new scheduling rules to an EDF scheduled system, so the ordering criterion of the ready queue remains the same as in any other EDF system, that is: tasks with the same priority are ordered in their queue according just to their absolute deadline. So when using DFP, two tasks (τ_a and τ_b) in the ready queue must be ordered according to:

$$\tau_a \leq \tau_b \Leftrightarrow d_a \leq d_b \tag{3}$$

Unlike with SRP (1), this order criterion verify the rules of a total order relation (2). One of the advantages of this is that we can use the binary heap to implement the queues in the ready queue.

The binary heap (Figure 3) is a heap data structure created using a binary tree, with two additional constraints: all levels of the tree, except possibly the last one (deepest) are fully filled, and each node is less than or equal to each of its children according to a comparison predicate defined for the data structure.

The binary heap is an efficient implementation of an ordered queue that requires a total order relation. Insert and remove operations take $O(log(n))$ time, while peeking the head takes constant time $O(1)$. Consequently, insert and remove operations are more much efficient on the binary heap than on the doubly-linked list used with SRP (general insert and remove operations on a doubly-linked list are $O(n)$).

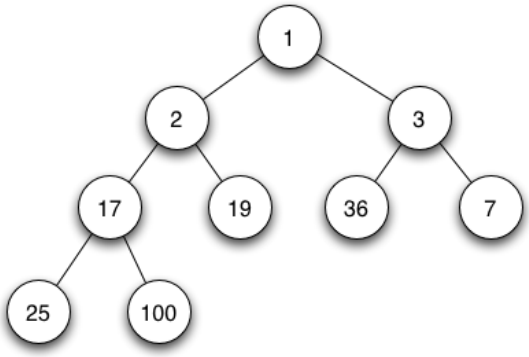

Figure 3: Binary Heap

4.2 Accessing a resource with DFP

In order to implement the DFP in MaRTE OS, we need to add some parameters and modify the functions that are used every time a task locks and unlocks a mutex so that the ready queue is ordered according to the task's absolute deadline. Those changes can be seen in Figure 4.

We have defined a new locking policy for the DFP and the `Deadlinefloor` parameter is added to the attributes of the mutex. We have also added a parameter to the mutex where the task's original deadline can be stored.

We now describe the full process of accessing and freeing a resource with this protocol.

When a task accesses a mutex the procedure `Running_Task_Locks_Mutex` is executed. First we store the task's current absolute deadline in a field of the mutex. Then we calculate the `Heir_Deadline`, which is the mutex access time (`Now`) plus the mutex deadline floor. This is the absolute deadline that the task may inherit from the mutex. Notice that this operation requires reading the clock. Then if the `Heir_Deadline` is shorter/sooner than the task's current absolute deadline, the task reduces its absolute deadline to the `Heir_Deadline`. Although the scheduling parameters of the task may have changed, making the task even more urgent, the task is currently the most urgent one, i.e., the first in the ready queue, so there is no need to reorder the ready queue.

When this task frees the mutex the procedure `Task_Unlock_Mutex` is executed. First we restore the task's previous absolute deadline. The task's scheduling parameters could have changed again, this time making it less urgent, and therefore the ready queue must be reordered. The procedure `Reorder_And_Dispatch` reorders the task in the ready queue and, in the case the task is no longer at the head of the queue, this procedure performs a context switch to preempt the task that has just unlocked the mutex.

The procedure `Reorder_And_Dispatch` removes the task from the head of the queue corresponding to its priority (a binary heap) and inserts it again into the same queue. The binary heap uses the function `<` to find the new place of the task in the data structure.

Since DFP does not impose any new rule to the basic EDF scheduling, there is no need to account for the number of mutexes locked by a task.

4.3 DFP Interface

There is a new interface for the DFP in MaRTE OS. The details of the interface can be seen in Figure 5. All functions return an integer as an error code, but it has been omitted for simplicity. Functions `Pthread_Mutexattr_Setdeadlinefloor` and `Pthread_Mutexattr_Getdeadlinefloor`

DFP Implementation

```
DEADLINE_FLOOR := constant Locking_Policy

type Mutex is new Mutex.Element with record
   ...
   Deadlinefloor :  HWTime := 0;
   Task_Deadline :  HWTime := 0;
end record;
```

```
procedure Running_Task_Locks_Mutex (Mutex) is
   ...
   Mutex.Owner_Deadline := Task.Deadline;
   Heir_Deadline := Now + Mutex.Deadlinefloor;
   if Task.Deadline > Heir_Deadline then
      Task.Deadline := Heir_Deadline;
   end if;
   ...
end Running_Task_Locks_Mutex;
```

```
procedure Task_Unlocks_Mutex (Task, Mutex) is
   ...
   Task.Deadline := Mutex.Owner_Deadline;
   Reorder_and_Dispatch (Task);
   ...
end Task_Unlocks_Mutex;
```

```
function < (Left_Task, Right_Task) return Boolean is
   ...
   return Left_Task.Deadline < Right_Task.Deadline;

end <;
```

Figure 4: DFP implementation in MaRTE OS.

and `Pthread_Mutex_Setdeadlinefloor` and `Pthread_Mutex_Getdeadlinefloor` set and get the deadline floor of the mutex statically and dynamically, respectively. There is no need to add functions to change the deadline of a task since that is a general EDF feature that it is already available in MaRTE OS.

DFP Interface

```
function Pthread_Mutexattr_Setdeadlinefloor
  (Attr:  access Mutex_Attributes, New_Deadlinefloor:  access Timespec)
function Pthread_Mutexattr_Getdeadlinefloor
  (Attr:  access Mutex_Attributes, Deadlinefloor:  access Timespec)

function Pthread_Mutex_Setdeadlinefloor
  (M: access Mutex, New_Deadlinefloor:  access Timespec)
function Pthread_Mutex_Getdeadlinefloor
  (M: access Mutex, Deadlinefloor:  access Timespec)
```

Figure 5: DFP interface in MaRTE OS.

5 Comparative Analysis

5.1 Implementation Complexity/Size

To measure the complexity of implementing SRP and DFP we count the number of attributes and operations needed for their implementation. Common attributes and operations for both protocols (such as data structures) are not taken into account.

In Table 1 we can see those numbers as well as the total code lines. The implementation of DFP is simpler than the one for SRP.

	SRP	*DFP*
Mutex Fields	2	2
Mutex Operations	4	4
Task Fields	1	0
Task Operations	4	0
Code Lines	54	34

Table 1: SRP and DFP implementation summary.

5.2 Tests Description

The following tests will be used to compare the performance of SRP and DFP. All tests are executed in a 800 MHz Pentium III.

- Test A. One resource, several tasks. One task with short relative deadline executes once accessing the resource a million times in a loop. The other tasks have longer relative deadline and they don't access the resource. We measure the short relative deadline task execution time which allows us to calculate the average time required to lock and unlock the resource.
- Test B. One resource, several tasks. One task with short relative deadline executes once accessing the resource a million times in a loop. The other tasks have longer relative deadline and they don't access the resource. This is like Test A except that we measure only the short relative deadline task unlock time.

- Test C. No resources, several tasks. One task with short relative deadline, several task with long relative deadline. The most urgent task is released periodically and we measure the difference between the theoretical and the actual activation time of this task, i.e., if the task's first activation is at t the next theoretical activation will be at $a_{theor} = t + T$, however the task won't be able to execute its first instruction until its actual activation, a, that will be later than a_{theor}. So, we measure $a - a_{theor}$ which gives an indication of the time needed to perform the context switch.

5.3 Tests Analysis

Tests A and B use the same scenario: a short deadline task accesses a resource while a number of long deadline tasks remain active in the ready queue. The only difference between these two tests is that the test A measures the lock+unlock time while the test B only measures the unlock time.

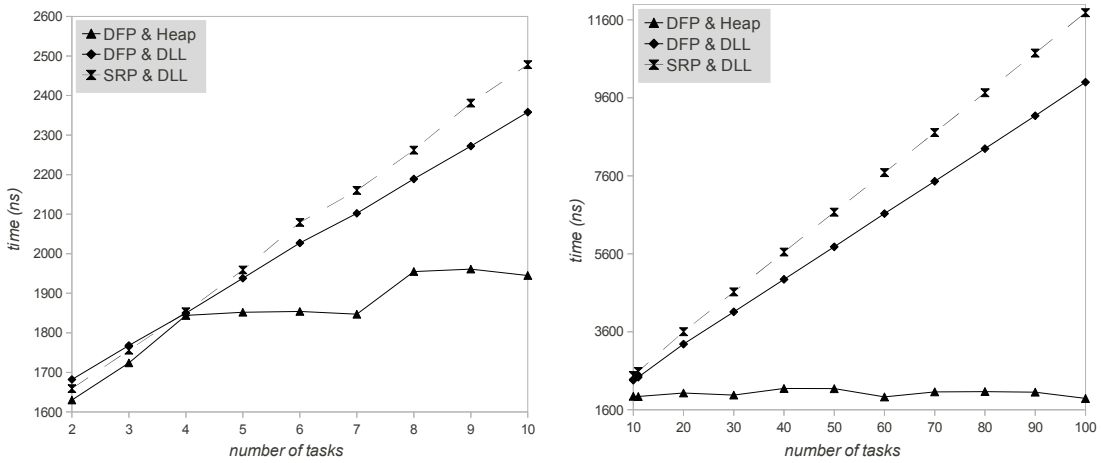

Figure 6: Test A experimental results. Left: Up to 10 tasks. Right: From 10 to 100 tasks.

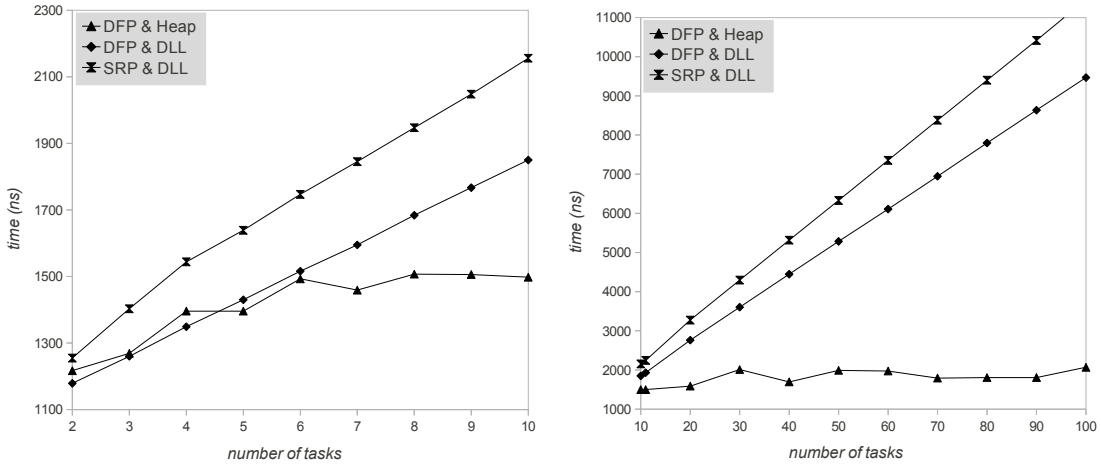

Figure 7: Test B experimental results. Left: Up to 10 tasks. Right: From 10 to 100 tasks.

This difference between tests A and B can be seen in Figures 6 and 7, for the case with less than 10 tasks in the ready queue (left charts on the figures) or with more than 10 (right charts). In Figure 7 we can see how the SRP & DLL has the worst performance, while the DFP & DLL and the DFP & Heap are more or less the same until we have more than five tasks, where the logarithmic behaviour of the heap starts to show its benefits.

In Figure 6 the measured times are higher than in Figure 7 because we are measuring both lock and unlock time, but it is clear that the lock operation is slower in the DFP as we can see how both DFP & Heap and DFP & DLL have worsened more than SRP & DLL. This is an expected outcome, since the locking operation in the DFP requires reading the clock (as it can be seen in Figure 4). But this disadvantage of DFP is quickly overcome as we increase the number of tasks, and for more than three tasks we can see how DFP performs better than SRP.

Now let us focus on the right charts of Figures 6 and 7 where the behaviour of tests A and B for medium/large number of tasks is represented. The logarithmic time of DFP using a heap (DFP & Heap) clearly outperforms the SRP using a doubly linked list (SRP & DLL). DFP using a doubly linked list (DFP & DLL) also behaves better than SRP & DLL. The difference between them is proportional to the number of threads. This is due to the simpler comparison function used by DFP. This result is an indication that DFP will outperform SRP with any data structure we could try to use. That means that it is very probable that, even if somebody finds a data structure more efficient than the heap that could be used by both DFP and SRP, it looks very probable that DFP will beat SRP.

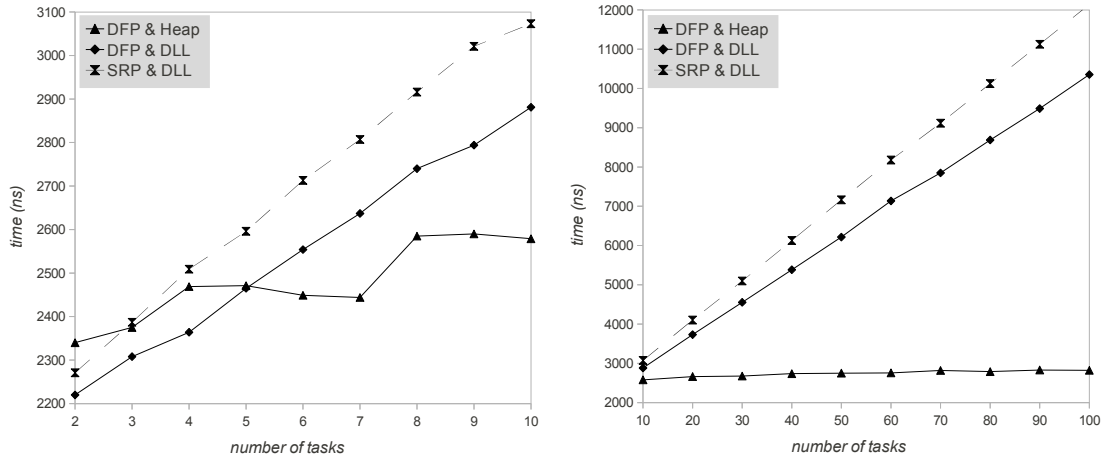

Figure 8: Test C experimental results. Left: Up to 10 tasks. Right: From 10 to 100 tasks.

Test C is similar to tests A and B but without any resource in the system: a short deadline task is released periodically while a number of long deadline tasks remain active in the ready queue. Therefore we can measure the time required to place the task in the ready queue and make the context switch.

In Figure 8 we can see the results of test C. The behaviour is basically the same as what we have seen in tests A and B with a resource being used. That means that the major factor that causes the difference between DFP & DLL and SRP & DLL is the simpler way of ordering the ready queue in the DFP, where tasks are just ordered by their absolute deadline. Even without resources locked in the system, an SRP implementation must check the value of the boolean `Mutexes_Owned` every step in the enqueing operation, as it can be seen in Figure 3.

It is interesting to notice an effect caused by the heap in Figures 6, 7 and 8 when there are less than ten tasks in the ready queue (left chart). We can see how the time increases abruptly from 3 to 4 tasks and the same increase happens from 7 to 8 tasks in the system. These steps are caused by the levels of the heap structure, as it can be seen in Figure 3. When there are up to 3 tasks in the system, we only have two levels of the heap occupied. From 4 to 7 tasks there are three levels occupied. The following changes of level are not so noticeable because the increase in time is proportionally less significant.

6 Conclusion

In this paper we have explained the details of the implementation of SRP and DFP in MaRTE OS. We have shown that, unlike SRP, DFP doesn't need to add more parameters to the tasks besides the ones necessary for any EDF scheduled system. In addition, DFP doesn't modify the scheduling rules. This also guarantees a total order relation between the tasks in the ready queue, which allows us to use the binary heap data structure to implement the MaRTE OS ready queue.

The possibility of using a heap with the DFP makes this protocol much more efficient than SRP and also, thanks to the simpler comparison function used by DFP this protocol also performs better than SRP even in the case the same data structure is used for both.

A Example: Ordering Tasks in SRP

This example shows how the order criterion imposed by SRP leads to a non-total order relation between the tasks in the ready queue. Figure 9 shows the expected execution of a system of three tasks (TA, TB and TC) and one resource (R1), where just one of the tasks accesses the resource.

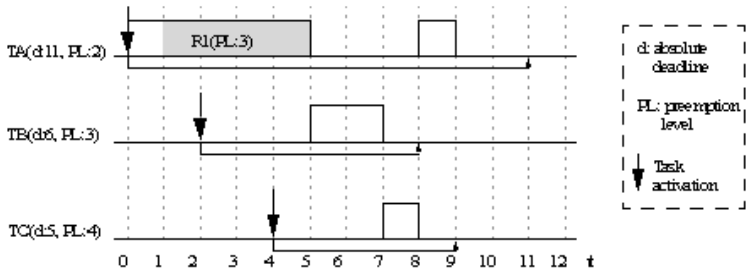

Figure 9: Expected timeline of three tasks and one resource using EDF+SRP.

In Figure 10 we can see the status of the ready queue as the time goes by. The head of the queue, with the highest priority element, is shown at the left of the figure, while the tail of the queue is at the right. The key point here is to realize that to find the correct place for TC in the queue at t=4 (when TC is activated) we must start looking from the tail of the queue towards the head. Going from tail to head, we start comparing TC with TB and we find TB < TC and, consequently, TC can not overtake TB in the queue.

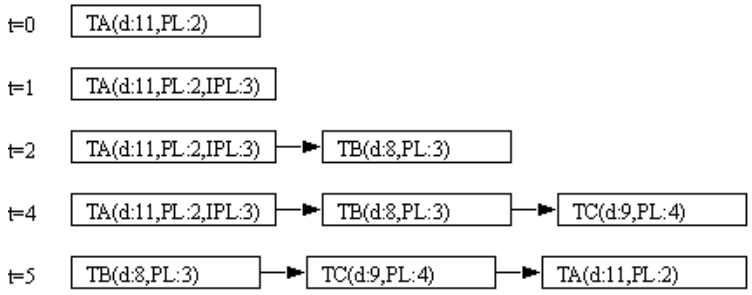

Figure 10: Ready queue status.

If we had started from the head of the queue, we would have found TC < TA and then TC would have been placed at the head of the queue, producing a priority inversion with respect to TB,

which is waiting for the resource and has a sorter deadline than TC. Here we can notice that the transitive property is NOT verified with the EDF+SRP rules: TC<TA and TA<TB but TB<TC.

References

[1] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *JACM*, 20(1):46–61, 1973.

[2] T.P. Baker. A Stack-Based Resource Allocation Policy for Realtime Processes. In *Proceedings IEEE Real-Time Systems Symposium (RTSS)*, pages 191–200, 1990.

[3] T.P. Baker. Stack-Based Scheduling of Realtime Processes. *Journal of Real-Time Systems*, 3(1), March 1991.

[4] Alan Burns. A Deadline-Floor Inheritance Protocol for EDF Scheduled Real-Time Systems with Resource Sharing. Technical Report YCS-2012-476, Department of Computer Science, University of York, UK, 2012.

[5] M. Aldea Rivas and M. González Harbour. MaRTE OS: An Ada kernel for real-time embedded applications. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference, Leuven*. Springer Verlag, LNCS 2043, 2001.

[6] ISO/IEC 8652:2012(E). Ada 2012 Reference Manual.

[7] POSIX: IEEE 1003.1-2008. *Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifications, Issue 7.*

[8] ISO/IEC 8652:2007(E). Ada 2005 Reference Manual.

[9] Roberto Tamassia Michael T. Goodrich. *Data Structures & Algorithms in JAVA*, chapter 8, pages 320–321. John Wiley & Sons, Inc, 2006.