Technical Report

Sporadic Multiprocessor Scheduling with Few Preemptions

Björn Andersson

TR-070501

Version: 1.0

Date: May 2007

Sporadic Multiprocessor Scheduling with Few Preemptions

Björn ANDERSSON

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: bandersson@dei.isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

Consider the problem of scheduling a set of n sporadically arriving tasks with the goal of meeting deadlines on a computer system comprising m processors. Processors are identical. A task τi is characterized by its minimum inter-arrival time Ti and its execution time Ci. Tasks can be preempted and they can migrate between processors. We propose an algorithm with utilization bound no lower than 88% and it generates few preemptions.

Sporadic Multiprocessor Scheduling with Few Preemptions

Björn Andersson
IPP-HURRAY! Research Group,
Polytechnic Institute of Porto (ISEP-IPP),
Rua Dr. António Bernardino de Almeida 431,
4200-072 Porto, Portugal
bandersson@dei.isep.ipp.pt

Abstract

Consider the problem of scheduling a set of n sporadically arriving tasks with the goal of meeting deadlines on a computer system comprising m processors. Processors are identical. A task τ_i is characterized by its minimum inter-arrival time T_i and its execution time C_i. Tasks can be preempted and they can migrate between processors. We propose an algorithm with utilization bound no lower than 88% and it generates few preemptions.

1. Introduction

Consider the problem of preemptive scheduling of n sporadically arriving tasks on m identical processors. A task τ_i is given a unique index in the range $1..n$ and a processor is given a unique index in the range $1..m$. A task τ_i generates a (potentially infinite) sequence of jobs. The time when these jobs arrive cannot be controlled by the scheduling algorithm and the time of a job arrival is unknown to the scheduling algorithm before the job arrives. It is assumed that the time between two consecutive jobs from the same task τ_i is at least T_i. Every job released from task τ_i requests to finish the execution of C_i time units at most T_i time units after its arrival. It is assumed that $0 \leq C_i \leq T_i$ and T_i and C_i are real numbers. A processor can execute at most one job at a time, and a job cannot execute on two or more processors simultaneously. The utilization is defined as $U_s = \frac{1}{m} \cdot \sum_{i=1}^{n} \frac{C_i}{T_i}$. The utilization bound UB_A of an algorithm A is the maximum number such that if $U_s \leq UB_A$ then all tasks meet their deadlines when scheduled by A.

Multiprocessor scheduling algorithms are often categorized as *partitioned* or *global scheduling* [11, 9, 13]. Global scheduling algorithms store tasks that have arrived but not finished their execution in one queue which is shared among all processors. At every moment the m highest-priority tasks among the tasks that have arrived but not finished their execution are selected for execution on the m processors. In contrast, partitioned scheduling algorithms partition the set of tasks such that all tasks in a partition are assigned to the same processor. Tasks are not allowed to migrate from one processor to another processor, and hence the multiprocessor scheduling problem is transformed to many uniprocessor scheduling problems. This simplifies scheduling and schedulability analysis because the wealth of results in uniprocessor scheduling can be reused. Unfortunately, all partitioned multiprocessor scheduling algorithms have a utilization bound of 50% or less. Global scheduling can achieve a utilization bound of 100% using a family of algorithms called *pfair scheduling* [5, 1]. But this great utilization bound comes at a price; all task parameters must be multiples of a time quantum and in every time quantum, a new task is selected for execution. As a result, the number of preemptions can be high [8]. We believe (as do Baker, see [4, page 12]) it is desirable to achieve a high utilization bound without suffering from a large number of preemptions.

In this paper we propose an algorithm for scheduling sporadic tasks. Its utilization bound is no lower than 88%. The algorithm assigns tasks to processors. A few tasks are assigned to two processors and they are carefully dispatched to ensure that they never execute on the two processors simultaneously; the other tasks are assigned to only one processor. This design circumvents the limitation of partitioned scheduling (the utilization bound of 50%) yet, it retains the advantages of partitioned scheduling in that (i) the time complexity of dispatching is independent of the number of processors and (ii) most tasks do not need to migrate at all and if a task is of the type that needs to migrate then it only needs to execute on two processors.

The algorithm generates few preemptions. Let $TMIN$ denote the minimum of all T_i. Let $njobs_p(t)$ denote the maximum number of jobs that can arrive during a time interval of length t and are assigned to only execute on processor

p. It holds that during a time interval of length t, the algorithm generates at most $12 \cdot \lceil t/TMIN \rceil + 2 + njobs_p(t)$ preemptions on processor p.

The remainder of this paper is organized as follows. Section 2 discusses the role of preemptions in real-time scheduling. It also discusses how to design a scheduling algorithm with high utilization bound and few preemptions. This discussions leads to our design, which is presented in Section 3 and its performance is proven. Section 4 gives conclusions.

2. Background

We say that a task τ_i is preempted at time t if (i) τ_i executed just before time t (let p denote this processor) and (ii) τ_i did not execute on processor p just after time t and (iii) τ_i has remaining execution time at time t. With this definition, a job that starts executing is not preempted and a job that finishes executing is not preempted either. Also, observe that a job may execute just before time t and also just after time t but these executions are on different processors so with our definition there is a preemption at time t. We believe this captures the notion of preemption used in the research community.

Preemptions are important to meet deadlines in real-time scheduling on both a uniprocessor and on a multiprocessor. In fact, every non-preemptive scheduling algorithm has a utilization bound of 0% (see Example 2 in [2]).

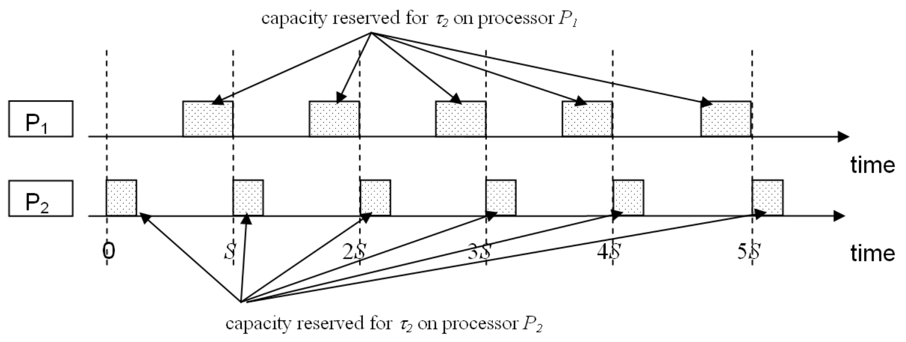
Although preemption is useful, it is important to not overuse it because a preemption has an associated operating system overhead [15, 14, 12]. The exact cost of a preemption is application and architectural dependent and we will not discuss that (see [6] for an excellent coverage). We will count the number of preemptions and in order to do so we need a metric. One metric could be the maximum number of preemptions that a job can suffer from. This is problematic though because there are task sets for which all scheduling algorithms that meet deadlines cause an infinite number of preemptions per job (one such task set is m=1, n=2, T_1=10, C_1=5, T_2=10k, C_2=5k and let k be an integer which approaches infinity). Another metric could be the number of preemptions in a time interval divided by the number of jobs that arrive in the time interval. Unfortunately, this metric is problematic as well because there are task sets sets for which all scheduling algorithms that meets deadlines generate a preemption and no job arrives in this interval; and hence the number of preemptions per job is infinite (see Example 3 in [2]).

We will express an upper bound on the number of preemptions in a time interval as a function of the duration of this time interval and the number of jobs that arrive in this time interval. For this metric, one can show (see Example 4 in [2]) that an inherent property of pfair scheduling is that the number of preemptions can be large. (The fact that pfair scheduling can cause many preemption has also been pointed out by Devi and Anderson [8].) And that enforcing that the pfair constraint only needs to be satisfied when a job arrives was a promising technique to reduce the number of preemptions. The algorithm BF [16] and variants of BF [7, 10] do that. An algorithm known as EKG [3] can be configured to achieve a utilization bound 100% and with even fewer preemptions. This is possible because it requires that only a subset of tasks must satisfy the pfair constraint when jobs arrive. Unfortunately, these algorithms were only designed to schedule periodically arriving tasks; that is, tasks where the time between two consecutive jobs of task τ_i is exactly T_i. This model allows algorithms to know, at every time, when the next job arrives and it permits these algorithms to calculate the amount of time that a task should be assigned in the time interval until the next job arrival. But for sporadic tasks, this technique cannot be used: the time of next arrival is unknown. For this reason, we will reason about how to design an algorithm for sporadic tasks; this reasoning takes as starting point partitioned scheduling, and then (in Section 3) we will present the new algorithm based on that reasoning.

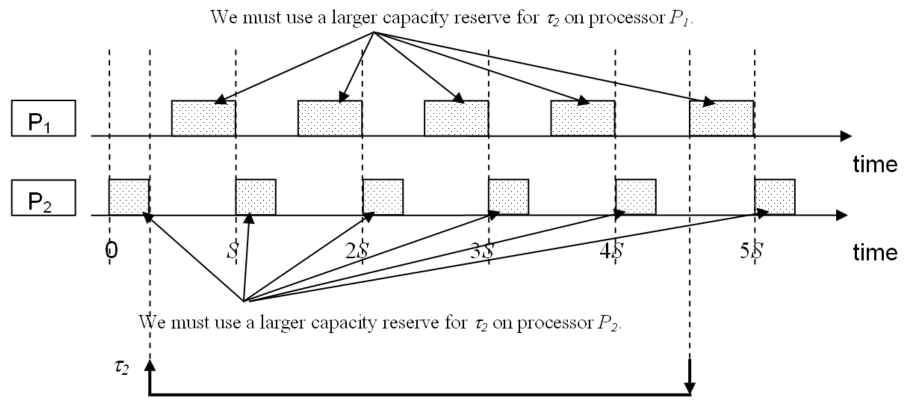
It is well-known that partitioned scheduling has a utilization bound of at most 50%. For illustrative purpose, the argument is repeated in Example 1.

Example 1. *Consider $n = m + 1$ tasks τ_i with $T_i = 1$ and $C_i = 0.5 + \epsilon$. In partitioned scheduling, tasks cannot migrate; they are assigned to a processor and always execute there. Since $n > m$, there is one processor which is assigned two or more tasks. Therefore, the utilization of this processor will be at least $1 + 2\epsilon$, and this is more than 100%. By choosing $m \to \infty$ and $\epsilon \to 0$ the task set will have $U_s \to 0.5$ and a deadline is missed. Hence, the utilization bound of every partitioned scheduling algorithm is 50% or less.*

This example stresses the fact that deadlines can be missed simply because a task could not be assigned to a processor, although there was plenty of idle time in the overall system. The idle time was spread out on different processors and could not be used. However, if in the same previous example a task is given some utilization on one processor and some utilization on another processor then it is possible to assign tasks such that the utilization on every processor reaches 100%. This approach was used in EKG. For many tasks, it gives utilization to only a single processors but for a few tasks, it gives utilization to two processors. For those tasks, it is imperative however that a task does not execute on two processors simultaneously. This property was ensured for those tasks by giving such tasks execution such that the amount of execution between two consecutive arrivals divided by the time between these two


capacity reserved for τ_2 on processor P_1
P1
time
P2
0
S
2S
3S
4S
5S
time
capacity reserved for τ_2 on processor P_2

(a) The reserves on processors for the task that is assigned to two processors.


We must use a larger capacity reserve for τ_2 on processor P_1.
P1
time
P2
0
S
2S
3S
4S
5S
time
We must use a larger capacity reserve for τ_2 on processor P_2.
τ_2

(b) In order to ensure that the split tasks meet deadlines for all possible arrivals, it is necessary to increase the size of the reserves.

Figure 1. How to perform run-time dispatching of tasks that are assigned to two processors.

consecutive arrivals is exactly equal to the utilization of the task on that processor. For sporadically arriving tasks such a scheme is not possible because the next arrival time is unknown. One can however, subdivide the timeline into time slots of duration S and ensure that each task that executes on two processors is given execution proportional to its utilization on these time slots. This approach does not require knowledge of future arrival times.

Figure 1 presents an example that illustrates how to perform such run-time dispatching. The task τ_2 has C_2/T_2=0.55. We can see (in Figure 1(a)) that τ_2 is assigned a reserve of 40% of the processing capacity of processor P_1 and τ_2 is assigned a reserve of 15% of the processing capacity of processor P_2. This processing capacity is distributed in each time interval of duration S. If τ_2 arrives exactly in the beginning of a time slot and if the deadline of τ_2 is exactly at the end of a time slot then τ_2 is given enough execution and hence it will meets its deadlines. But for other arrival times this may not be the case. Consider Figure 1(b). Here we can see the arrival time of a job of task τ_2. For this arrival time, it is necessary that the reserves of τ_2 are larger in order to meets its deadline. We will let the reserves of a processor assigned to a task be the same for all time slots. Consequently, the reserve on processor 1 given to task τ_2 will be more than 40% regardless of when and if τ_2 arrives.

It is necessary to assign a value to S and several choices are possible. Choosing S to be small implies that the reserves only need to be increased by a small amount to meet deadlines and this offers a high utilization bound. On the other hand, choosing S to be large implies that the number of preemptions is small. As a compromise, we choose S=$TMIN/4$.

3. The new algorithm

The new algorithm brings the ideas that were successful for the design of EKG and exploits them for sporadically arriving tasks. The new algorithm consists of two steps. First, tasks are assigned to processors. Some tasks may be assigned to two processors, meaning that this task may execute on any of these two processors but it is not permitted

```
1.  for p in 1..$m$ do
2.     U[p] := 0
3.     lo_split[p] := 0
4.     hi_split[p] := 0
5.  end for
6.  Let $\tau^{heavy}$ denote the set of tasks such that $C_i/T_i >$ SEP
7.  Let $\tau^{light}$ denote the set of tasks such that $C_i/T_i \leq$ SEP
8.  L := | heavy |
9.  Order tasks such that $\tau_i$ with $i$ in 1..L are all in $\tau^{heavy}$
10.      and $\tau_i$ with $i$ in $L$+1..$n$ are all in $\tau^{light}$
11. Sort tasks in L+1..$m$ such that $T_{L+1} \leq T_{L+2} \leq \ldots \leq T_m$
11. for $i$ in 1..L do
12.    p := i
13.    U[p] := U[p] + $C_i/T_i$
14.    $\tau_i$.processor_id1 := p
15.    $\tau_i$.processor_id2 := p
16. end for
17. p := L + 1
18. for i := L+1 to n do
19.    if U[p]+$C_i/T_i \leq$ SEP then
20.       U[p] := U[p] + $C_i/T_i$
21.       $\tau_i$.processor_id1 := p
22.       $\tau_i$.processor_id2 := p
23.    else
24.       if p+1 $\leq$ m then
25.          hi_split[p] := SEP-$C_i/T_i$
26.          lo_split[p+1] := $C_i/T_i$-hi_split[p]
27.          $\tau_i$.processor_id1 := p
28.          $\tau_i$.processor_id2 := p+1
29.          $U[p]$ := $U[p]$ + hi_split[p]
30.          $U[p$+1] := $U[p$+1] + lo_split[p+1]
31.          $p$ := $p$+1
32.       else
33.          declare FAILURE
34.       endif
35.    end if
36. end for
37. declare SUCCESS
```

Figure 2. An algorithm for assigning tasks to processors.

to execute on both processors simultaneously. We refer to such a task as *split tasks*. Naturally tasks that are not split tasks are refered to as *non-split tasks*. The algorithm for task assignment is presented in Figure 2 and it is performed before run-time. Then, at run-time, the tasks are dispatched; Figure 3 presents this algorithm. Let us now understand the behavior, the rationale of the design of the new algorithm and then prove its performance.

Figure 2 presents the algorithm for assigning tasks to processors. The algorithm assigns tasks to processors such that on all processors the utilization does not exceed 100%. The algorithm treats *heavy* and *light* tasks differently. A task τ_i is heavy if $C_i/T_i >$ SEP, otherwise it is light. SEP means separator. The exact value of SEP will be specified later. First, the algorithm assigns heavy tasks to their own dedicated processors (at lines 11-16); no other task will be assigned to these processors. The main idea of the algorithm is that there is a current processor, with index p and tasks are considered one by one; index i denotes the current task. The task currently under consideration is attempted to be assigned to the current processor p. This is performed at line 19. If the condition stated in that line is true then the task can be assigned to processor p. If the condition is false, the utilizaton of the task is assigned to two processors (at lines 25-30) and the task assigned to the current processor p and processor p + 1. Then the processor with a higher index is considered (line 31). As already pointed out, the algorithm assigns heavy tasks to some processors and light tasks to some other processors. L separates these tasks: heavy tasks are assigned to processors with indices ranging from 1 up to L, while light tasks are assigned to processors with indices ranging from L + 1 up to m. The performance of the task assignment scheme (presented in Figure 2) is given by Lemma 1.

Lemma 1. *If* $0 \leq SEP \leq 1$ *and a task set satisfies* $U_s \leq SEP$ *and the algorithm in Figure 2 is used then the algorithm in Figure 2 declares SUCCESS.*

Proof. The proof is by contradiction. Suppose that the lemma was false then there is a task set that satisfies $U_s \leq SEP$ and the algorithm in Figure 2 is used and the algorithm in Figure 2 declares FAILURE.

Let us consider the value of i when the algorithm declared failure. Let us delete all tasks with a higher index than i. We obtain that the algorithm still declares failure. If there was any task with $C_j/T_j >$ SEP then this task is deleted and this processor is deleted. If we rerun the algorithm with the new task set then it still declares failure. We can repeat this argument until it holds for all tasks that $\forall j : C_j/T_j \leq$ SEP. Consequently we now have a task set $\{\tau_1, \tau_2, \ldots, \tau_n\}$ that satisfies

$$U_s \leq SEP \tag{1}$$

and $\forall j$: $C_j/T_j \leq$ SEP and the algorithm in Figure 2 is used and the algorithm in Figure 2 declares FAILURE. From line 19 and lines 25-30 we obtain:

$$\forall p \in 1..m-1 : U[p] = SEP \tag{2}$$

$$U[m] + C_n/T_n > SEP \tag{3}$$

Combining Equality 2 and Inequality 3 and using the knowledge that τ_1, τ_2, ..., τ_{n-1} are assigned on processors P_1,P_2,...,P_m yields:

$$\sum_{i=1}^{n} C_i/T_i > SEP \cdot m \tag{4}$$

Dividing by m and using the definition of U_s yields

$$U_s > SEP \tag{5}$$

This contradicts Inequality 1 and hence the lemma is correct. □

We are now in position to see a precise statement of the dispatcher used at run-time. Figure 3 presents this algorithm. There is a constant, α which will be specified later in this paper. It is assumed that if the array `split_tasks` is indexed out of bounds then it returns NULL. It is also assumed that if `switch_to` is called with a NULL parameter then the processor is kept idle.

The variable rq (on line 11), means run-queue and it is a variable which store tasks that are ready for execution or are executing. This ready queue consists of two types of tasks (i) tasks that may execute on two different processors (called `split_tasks`) and (ii) tasks that may only execute on a single processor (called `non_split_tasks`). It can be seen (from the algorithm that assigns tasks to processors, in Figure 2) that at most m-1 tasks are split_tasks and there can be at most one task split between processor p and p+1. For this reason, we only need an array of m-1 elements for storing the split tasks. For the non-split tasks, however, it is necessary to store a priority queue for each processor (see line 8). This priority queue (line 6) contains data structures of the type `task_struct` and they are ordered in EDF order. On line 24 in Figure 3, a task struct is inserted in a queue; it is assumed that if this task struct is already in the queue then this operation takes no effect. Analogously, on line 34 in Figure 3, a task struct is removed from a queue; it is assumed that if this task struct is not in the queue then this operation takes no effect.

Each task is represented in the operating system by a variable of the type `task_struct` (shown in lines 1-5). The notation τ_i refers to the task struct of task τ_i. The variables `processor_id1` and `processor_id2` state which processor a task is assigned to. If a task is only assigned to one processor then `processor_id1` and `processor_id2` are set to the id of this processor. The values of `processor_id1` and `processor_id2` are outputs from the algorithm in Figure 2. The variable `T` state T_i of a task. The variable `d` state the absolute deadline of the current job of a task.

The variable t_0 and t_1 (on line 15) stores the current time slot of length S. This is the time slot discussed in Figure 1. t_0 is the start time of the time slot, whereas t_1 is the finishing time of the time slot. For a processor p, the time interval [t_0,t_1) can be divided into three subinterval [t_0,timea[p]), [timea[p],timeb[p]) and [timeb[p],t_1). A processor p is said to be in state `a` if the current time is in [t_0,timea[p]); the processor is in state `x` if the current time is in [timea[p],timeb[p]) and the processor is in state `b` if current time is in [timeb[p],t_1). There is an array: `state`. The variable `state[`p`]` stores the state of processor p. timea[p] and timeb[p] are computed based on the `lo_split` and `hi_split` (which were computed in the algorithm in Figure 2) and α (which control how much larger the reserve needs to be in order to ensure that split tasks meet their deadlines for all arrival times).

The event handler "'**when** the current time becomes = t_1 **do**"' calculates the times that govern when processors should perform state transition. In particular, when the time becomes t_1 then new values of t_0,$timea$, $timeb$ and t_1 are calculated (as shown on lines 50-53).

The scheduling is performed by the lines 16-77. It is a state machine executing on each processor and it reacts on events. The main idea is that if a processor p is in state `a` then it executes the reserve that it shares the processor $p-1$, whereas if a processor p is in state `b` then it executes the reserve that it shares the processor $p+1$ and if a processor p is in state `x` then it uses EDF to dispatch a non-split tasks assigned to processor p. Observe however that if the task that was intended to be executed on the reserve does not need to execute then the non-split task with the earliest deadline is selected.

We will now assign values to the parameters SEP and α. From the task assignment presented in Figure 2, we know that if a task τ_i is assigned to two processor p and p+1 then it holds that:

$$lo\_split[p+1] + hi\_split[p] = \frac{C_i}{T_i} \tag{6}$$

For this purpose of understanding how to choose SEP and α, Figure 4 is instrumental. Consider task τ_i in Figure 4(a) and this particular arrival time. From this figure and considering p=1 and considering lines 50-53 in Figure 3 we obtain that:

$$x = S \cdot (\alpha + lo\_split[p+1]) \tag{7}$$

and

$$y = S \cdot (\alpha + hi\_split[p]) \tag{8}$$

and

$$4 \cdot S + (S - (x+y)) = T_i \tag{9}$$

If τ_i missed a deadline it must have been that it used all the time available in the reserves. That is, it must have been that:

$$4 \cdot (x+y) < C_i \tag{10}$$

```
type task_struct = record
      processor_id1, processor_id2 : integer
      $T$ : time
      $d$ : time_span
    end record
type EDF_ready_queue = priority queue of task_struct
type rq_split_and_non_split = record
    non_split_tasks : array[1..$m$] of EDF_ready_queue
    split_tasks : array[1..$m$-1] of ready_task
  end record
rq : rq_split_and_non_split
hi_split, lo_split : array[1..$m$] of floating point number
state : array[1..$m$] of enumerated variable {a,x,b}
timea, timeb : array[1..$m$] of time
$t_0$, $t_1$ : time
when the system boots do
  $t_1$ := current time; $S$=$TMIN/4$
  call "when the current time becomes = $t_1$ do"
end when
when a job of task $\tau_i$ arrives do
  if $\tau_i$.processor_id1=p or $\tau_i$.processor_id2=p then
    $\tau_i.d$ := current time + $\tau_i.T$
    if $\tau_i$.processor_id2=$\tau_i$.processor_id1 then
      insert( rq.non_split_tasks[p], $\tau_i$)
    else
      rq.split_tasks[p] := $\tau_i$
    end if
    call dispatch
  end if
end when
when a job of task $\tau_i$ finishes execution do
  if $\tau_i$.processor_id1=p or $\tau_i$.processor_id2=p then
    if $\tau_i$.processor_id2=$\tau_i$.processor_id1 then
      remove( rq.non_split_tasks[p], $\tau_i$ )
    else
      rq.split_tasks[p] := NULL
    end if
    call dispatch
  end
end when
when the current time becomes = timea[p] do
  state[p] := x
  call dispatch
end when
when the current time becomes = timeb[p] do
  state[p] := b
  call dispatch
end when
when the current time becomes = $t_1$ do
  $t_0$ := $t_1$
  $t_1$ := $t_0$ + S
  timea[p] := $t_0$ + $S \cdot$ (lo_split[p] + $\alpha$)
  timeb[p] := $t_1$ - $S \cdot$ (hi_split[p] + $\alpha$)
  state[p] := a
  call dispatch
end when
procedure dispatch is
  if state[p]=a then
    if rq.split_tasks[p-1] $\neq$ NULL then
      switch_to( rq.split_tasks[p-1] )
    else
      switch_to( rq.non_split_tasks[p].peek )
    end if
  elseif state[p]=b then
    if rq.split_tasks[p] $\neq$ NULL then
      switch_to( rq.split_tasks[p] )
    else
      switch_to( rq.non_split_tasks[p].peek )
    end if
  else
    if rq.non_split_tasks[p].peek $\neq$ NULL then
      switch_to( rq.non_split_tasks[p].peek )
    else
      switch_to( NULL )
    end if
  end if
end when
```

Figure 3. An algorithm for run-time dispatching on processors L+1..m.

Combining Inequality 10 and Equality 9 yields:

$$\frac{4 \cdot (x+y)}{4 \cdot S + (S-(x+y))} < \frac{C_i}{T_i} \tag{11}$$

Let us apply Equality 7 and Equality 8 on Inequality 11. Then apply Inequality 6 yields:

$$\frac{4 \cdot (2\alpha + \frac{C_i}{T_i})}{5 - (2\alpha + \frac{C_i}{T_i})} - \frac{C_i}{T_i} < 0 \tag{12}$$

One can show that if we choose α as $\alpha = \frac{9}{2} - 2\sqrt{5}$ then it holds for all C_i/T_i such that $0 \leq C_i/T_i \leq 1$ then the left-hand side of Inequality 12 is non-negative. Hence we have that if $\alpha = \frac{9}{2} - 2\sqrt{5}$ and if the task τ_i arrives as show by Figure 4(a) then τ_i meets its deadline.

Let us now assume that we have selected $\alpha = \frac{9}{2} - 2\sqrt{5} \approx 0.0278$ and let us consider how it impacts a task τ_j that is only assigned to a single processor. Figure 4(b) shows a specific scenario. Let us explore how much processor p in Figure 4(b) can be utilized. From this figure we obtain that:

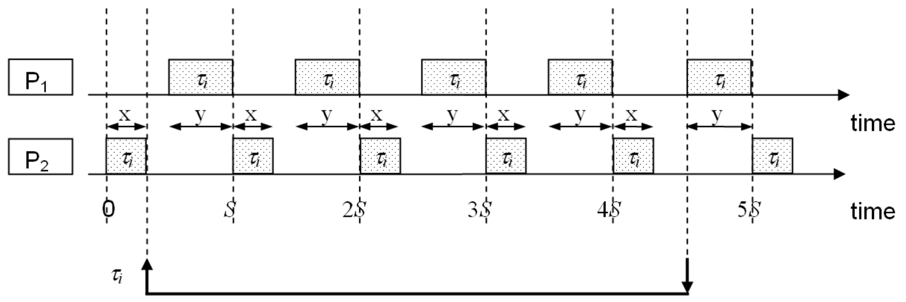
$$x = S \cdot (\alpha + lo\_split[p]) \tag{13}$$

and

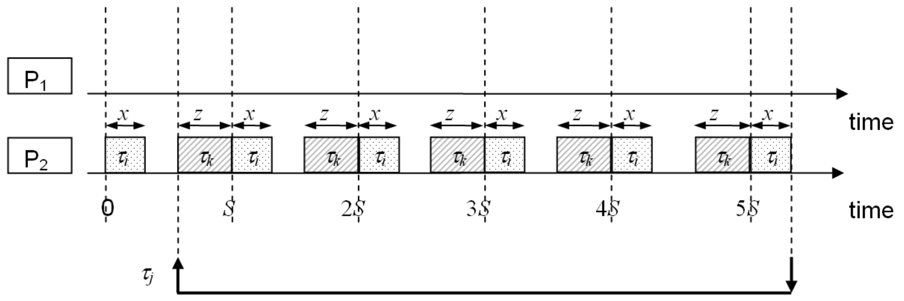
$$z = S \cdot (\alpha + hi\_split[p]) \tag{14}$$

and

$$4 \cdot S + (x+z) = T_j \tag{15}$$



(a) The reserves must be large enough to ensure that a task τ_i that is assigned to two processors receives enough execution even when it arrives at an unfavorable time.



(b) The reserves must not be too large because then they can cause interference on another task τ_j which is only assigned to a single processor.

Figure 4. How to inflate the reserves properly.

If task τ_j missed its deadline for the scenario shown in Figure 4(b) then the processor p was busy during the time from when the job τ_j arrived until its deadline expired. (This is true because τ_j is a non-split task and non-split tasks are allowed to be executed in state a and b with a lower priority than the tasks that were assigned these reserves. See line 62 and line 68 in the algorithm in Figure 3.) Because of this deadline miss it holds that:

$$5 \cdot (x+z) + C_j > T_j \tag{16}$$

Dividing Inequality 16 by T_j and applying Equality 15 yields:

$$\frac{5 \cdot (x+z)}{4 \cdot S + (x+z)} + C_j/T_j > 1 \tag{17}$$

Applying Equality 13 and Equality 14 and rewriting yields:

$$\frac{5 \cdot (2\alpha + lo\_split[p] + hi\_split[p])}{4 + (2\alpha + lo\_split[p] + hi\_split[p])} + C_j/T_j > 1$$

One can show that for the α selected and for lo_split[p] + hi_split[p] ≥ 0 it holds that:

$$\begin{aligned} &18 - 8\sqrt{5} + lo\_split[p] + hi\_split[p] \\ &\geq \frac{5 \cdot (2\alpha + lo\_split[p] + hi\_split[p])}{4 + (2\alpha + lo\_split[p] + hi\_split[p])} \end{aligned} \tag{18}$$

Combining these yields:

$$\begin{aligned} &18 - 8\sqrt{5} + \\ &lo\_split[p] + hi\_split[p] + C_j/T_j > 1 \end{aligned} \tag{19}$$

Rewriting yields:

$$lo\_split[p] + hi\_split[p] + \frac{C_j}{T_j} > 8\sqrt{5} - 17 \tag{20}$$

As a consequence of this reasoning, it follows that if

$$lo\_split[p] + hi\_split[p] + \frac{C_j}{T_j} \leq 8\sqrt{5} - 17 \tag{21}$$

then τ_j meets its deadline for the scenario depicted in Figure 4(b).

We have now shown that the choice $\alpha = \frac{9}{2} - 2\sqrt{5} \approx$ 0.0278 and SEP=$8\sqrt{5}$-17 $\approx$ 0.88854382 causes deadlines to be met in the examples shown in Figure 4(a) and Figure 4(b). We use these parameters in the remainder of the paper and prove the utilization bound of the algorithm. We will do so by first stating lemmas that ensure that (i) heavy tasks meet their deadlines, (ii) split tasks do not execute on two or more processors simultaneously and (iii) split tasks meet their deadlines. Using these lemmas and reasoning about non-split tasks will give us the utilization bound.

Lemma 2. *Assume that tasks assigned to processors 1..L are dispatched by any work-conserving algorithm. Then all tasks assigned to processors 1..L meet their deadlines.*

Proof. Follows from the fact that we assume $0 \leq C_i \leq T_i$ (which was stated in the introduction). □

Lemma 3. *Assume that tasks are assigned using the algorithm presented in Figure 2 (with SEP=8$\sqrt{5}$-17) and tasks on processors 1..L are dispatched by any work-conserving uniprocessor scheduling algorithm and tasks on processors L+1..m are dispatched using the algorithm presented in Figure 3 (with $\alpha = \frac{9}{2} - 2\sqrt{5}$ and S=$TMIN/4$). Then it holds for every task τ_i that τ_i never executes on two or more processors simultaneously.*

Proof. If τ_i is a non_split task then the truth of the lemma is obvious. We will now prove that the lemma is true also for the case when τ_i is a split task. Suppose that the lemma was incorrect then there must be a time t when a split task τ_i executed on two processor simultaneously. Observe that the algorithm presented in Figure 2 assigns a task to at most two processors. Hence τ_i executes on two processors. Let these processors be denoted p and p+1. Due to the assumption on the falsity of the lemma it holds that processor p and processor p+1 execute τ_i at time t. From the algorithm Figure 3 it follows that $t \leq$ timea[p+1] and timeb[p] $\leq t$. As a consequence, it follows that:

$$timeb[p] \leq timea[p+1] \tag{22}$$

This can obviously be rewritten as:

$$0 \leq timea[p+1] - timeb[p] \tag{23}$$

Using lines 50-53 in the algorithm presented in Figure 3 yields:

$$1 \leq lo\_split[p+1] + hi\_split[p] + 2 \cdot (\frac{9}{2} - 2\sqrt{5}) \tag{24}$$

From our choice of SEP we have that:

$$lo\_split[p+1] + hi\_split[p] \leq 8\sqrt{5} - 17 \tag{25}$$

Combining Inequality 25 with Inequality 24

$$1 \leq -8 + 4\sqrt{5} \tag{26}$$

But this is impossible. Hence the lemma is true. □

Lemma 4. *Assume that tasks are assigned using the algorithm presented in Figure 2 (with SEP=8$\sqrt{5}$-17) and tasks on processors 1..L are dispatched by any work-conserving uniprocessor scheduling algorithm and tasks on processors L+1..m are dispatched using the algorithm presented in Figure 3 (with $\alpha = \frac{9}{2} - 2\sqrt{5}$ and S=$TMIN/4$). Then it holds that every split task meet its deadline.*

Proof. Suppose that the lemma was false then there would be a split task that misses a deadline. We can delete all non-split tasks and then we obtain the same schedule because split tasks are executed with higher priority (see the algorithm presented in Figure 3). And hence there is still a split task that misses a deadline. Let us consider the first time that a deadline was missed. Let τ_i denote the task that released this job. Since $T_i \geq TMIN$ and S=$TMIN/4$ we obtain that: $T_i \geq 4S$. Let A_i denote the time time when this job arrived. Consequently the deadline (that was missed) is at time A_i+T_i. We know (from the algorithm in Figure 2) that a split task is only permitted to execute on two processors and they have consecutive index. Let p and p+1 be denote those processor where τ_i is permitted to execute. From the algorithm in Figure 2, we also have that:

$$C_i/T_i = hi\_split[p] + lo\_split[p+1] \tag{27}$$

Let x and y be denoted as:

$$x = S \cdot (\alpha + lo\_split[p+1]) \tag{28}$$

and

$$y = S \cdot (\alpha + hi\_split[p]) \tag{29}$$

Since a deadline was missed for this task τ_i it implies that (i) $\forall\ t \in [A_i$,A_i+T_i) such that at time t processor p+1 was in state a, it holds that τ_i executed and (ii) $\forall\ t \in [A_i$,A_i+T_i) such that at time t processor p was in state b, it holds that τ_i executed. Consequently, the amount of time that τ_i is given execution during [A_i,A_i+T_i) is at least:

$$\begin{aligned} &\lceil T_i/S \rceil \cdot (x+y) \\ &-min(\lceil T_i/S \rceil \cdot S - T_i, (x+y)) \end{aligned} \tag{30}$$

Since less than C_i time units was performed by task τ_i during [A_i,A_i+T_i) it follows that:

$$\begin{aligned} &\lceil T_i/S \rceil \cdot (x+y) \\ &-min(\lceil T_i/S \rceil \cdot S - T_i, (x+y)) < C_i \end{aligned} \tag{31}$$

Applying Inequalities 27, 28, 29 and the value of α yields:

$$\begin{aligned} &\lceil T_i/S \rceil \cdot S \cdot (C_i/T_i + 9 - 4\sqrt{5}) \\ -&min(\lceil T_i/S \rceil \cdot S - T_i, S \cdot (C_i/T_i + 9 - 4\sqrt{5})) < C_i \end{aligned} \tag{32}$$

We also know from the assumptions of the lemma that:

$$C_i/T_i \leq 8\sqrt{5} - 17 \tag{33}$$

and due to the choice of S:

$$4 \cdot S \leq T_i \tag{34}$$

Dividing Inequality 32 by T_i and then subtracting both sides by C_i/T_i yields:

$$\lceil T_i/S \rceil \cdot (S/T_i) \cdot (C_i/T_i + 9 - 4\sqrt{5}) \\ -min(\lceil T_i/S \rceil \cdot S/T_i - 1, \\ (S/T_i) \cdot (C_i/T_i + 9 - 4\sqrt{5})) - C_i/T_i < 0 \quad (35)$$

One can show that under the assumption of Inequality 34 it holds that for every C_i/T_i such that $0 \le C_i/T_i \le 1$ then

$$0 \le \lceil T_i/S \rceil \cdot (S/T_i) \cdot (C_i/T_i + 9 - 4\sqrt{5}) \\ -min(\lceil T_i/S \rceil \cdot S/T_i - 1, \\ (S/T_i) \cdot (C_i/T_i + 9 - 4\sqrt{5})) - C_i/T_i \quad (36)$$

Inequality 36 contradicts Inequality 35 and hence the lemma is correct.

□

We saw in Lemma 4 that all split tasks meet their deadlines. In the following, we will prove that also the non-split tasks meet their deadlines.

Theorem 1. *Assume that tasks are assigned using the algorithm presented in Figure 2 (with SEP=8$\sqrt{5}$-17) and tasks on processors 1..L are dispatched by any work-conserving uniprocessor scheduling algorithm and tasks on processors L+1..m are dispatched using the algorithm presented in Figure 3 (with $\alpha = \frac{9}{2} - 2\sqrt{5}$ and S=$TMIN/4$). If $U_s \le 8\sqrt{5}$-17 then all deadlines are met and no task executes on two or more processors simultaneously.*

Proof. From Lemma 2 it follows that all heavy tasks meet their deadlines and clearly they do not execute on two or more processors simultaneously. For the processors L+1..m that schedules light tasks, we know (from the algorithm presented in Figure 2) that $\forall p$: U[p] $\le 8\sqrt{5}$-17. From Lemma 4 it follows that also all light tasks that are split meet their deadlines and clearly (from Lemma 3) it holds that they do not execute on two or more processors simultaneously.

It remains to prove that on processors L+1..m, all non-split tasks meet their deadlines. We will do so now.

Suppose that a non-split task missed its deadline. If there are many jobs that missed a deadline then let us consider the job with the earliest deadline. Let τ_i denote the task that released that job and let p denote the processor to which τ was assigned. Let A_i denote the time when this job arrived and consequently the deadline of this job is at time A_i+T_i. We know that the processor p is busy during [A_i,A_i+T_i) (This is true because τ_i is a non-split task and non-split tasks are allowed to be executed in state a and b with a lower priority than the tasks that were assigned these reserves. See line 62 and line 68 in the algorithm in Figure 3.) Let Q denote the latest time before A_i+T_i such that processor p is busy during the time interval [Q,A_i+T_i). We know that this interval [Q,A_i+T_i) is non-empty because $Q \le A_i$. We can delete all jobs with arrival times before Q; the job released by τ_i is still missed. Let us consider another job released from a non-split task τ_j. If this job arrived later than A_i+T_i-T_j then its deadline is later than A_i+T_i. Such a job can be deleted and because of EDF scheduling, the job released by τ_i still misses its deadline. Let L denote the length of the busy period; that is, L=A_i+T_i-Q. Observe that $L \ge T_i$ and $T_i \ge 4 \cdot S$ and this gives us:

$$L \ge 4 \cdot S \quad (37)$$

Let x denote (lo_split[p]+α) · S and let z denote (hi_split[p]+α) · S. We know that split tasks meet their deadlines and hence an upper bound on the amount of work that is performed by split tasks on processor p during the time interval of length L is at most:

$$\lfloor \frac{L}{S} \rfloor \cdot (x+z) + \\ min(L - \lfloor \frac{L}{S} \rfloor \cdot S, x+z) \quad (38)$$

Let NS[p] denote the set of tasks that are only assigned to processor p. Since a deadline was missed and the processor was busy during the time interval of length L, it follows that:

$$(\sum_{\tau_j \in NS[p]} \lfloor \frac{L}{T_j} \rfloor \cdot C_j) + \\ \lfloor \frac{L}{S} \rfloor \cdot (x+z) + \\ min(L - \lfloor \frac{L}{S} \rfloor \cdot S, x+z) > L \quad (39)$$

Dividing by L and using the definition of x and z and using the knowledge that $\frac{C_j}{T_j} \ge \lfloor \frac{L}{T_j} \rfloor \cdot \frac{C_j}{L}$. yields:

$$(\sum_{\tau_j \in NS[p]} \frac{C_j}{T_j}) + \\ \lfloor \frac{L}{S} \rfloor \cdot \frac{S \cdot (lo\_split[p] + hi\_split[p] + 9 - 4\sqrt{5})}{L} + \\ min(1 - \lfloor \frac{L}{S} \rfloor \cdot \frac{S}{L}, \\ \frac{S \cdot (lo\_split[p] + hi\_split[p] + 9 - 4\sqrt{5})}{L}) > 1$$

One can show that for the α selected and for lo_split[p] + hi_split[p] ≥ 0 and $L \ge 4 \cdot S$ it holds that:

$$18 - 8\sqrt{5} + lo\_split[p] + hi\_split[p] \ge \\ \lfloor \frac{L}{S} \rfloor \cdot \frac{S \cdot (lo\_split[p] + hi\_split[p] + 9 - 4\sqrt{5})}{L} + \\ min(1 - \lfloor \frac{L}{S} \rfloor \cdot \frac{S}{L}, \\ \frac{S \cdot (lo\_split[p] + hi\_split[p] + 9 - 4\sqrt{5})}{L})$$

And combining them yields:

$$(\sum_{\tau_j \in NS[p]} \frac{C_j}{T_j}) +$$
$$18 - 8\sqrt{5} + lo\_split[p] + hi\_split[p] > 1$$

Rewriting yields:

$$(\sum_{\tau_j \in NS[p]} \frac{C_j}{T_j}) +$$
$$lo\_split[p] + hi\_split[p] > 8\sqrt{5} - 17 \qquad (40)$$

From the algorithm in Figure 2 we have $U[p] \leq 8\sqrt{5}$-17. Hence we obtain that:

$$(\sum_{\tau_j \in NS[p]} \frac{C_j}{T_j}) +$$
$$lo\_split[p] + hi\_split[p] \leq 8\sqrt{5} - 17 \qquad (41)$$

Inequality 41 contradicts Inequality 40 and hence τ_i does not miss its deadline. It follows that the statement of the theorem is true. □

The number of preemptions is stated in Theorem 2.

Theorem 2. *Assume that tasks are assigned using the algorithm presented in Figure 2 (with SEP=8$\sqrt{5}$-17) and tasks on processors 1..L are dispatched by any work-conserving uniprocessor scheduling algorithm and tasks on processors L+1..m are dispatched using the algorithm presented in Figure 3 (with $\alpha = \frac{9}{2} - 2\sqrt{5}$ and S=$TMIN/4$). Let $njobs_p(t)$ denote the maximum number of jobs that can arrive during a time interval of length t and are assigned to only execute on processor p. It holds that during a time interval of length t, the algorithm generates at most $12 \cdot \lceil t/TMIN \rceil$ + 2 + $njobs_p(t)$ preemptions on processor p.*

Proof. In a time interval, there are $12 \cdot \lceil t/TMIN \rceil$ + 2 preemptions due to the execution of reserves. There are $njobs_p(t)$ preemptions due to EDF scheduling in state X. Adding these preemptions gives us the theorem. □

4. Conclusions

We have proposed an algorithm for scheduling sporadic tasks. Its utilization bound is 88% and it generates few preemptions. We left open the important question on how to extend this algorithm for sporadic tasks where the deadline of a task is not equal to its minimum inter-arrival time.

References

[1] J. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *Journal of Computer and System Sciences*, 68(1):157–204, 2004.

[2] B. Andersson. Sporadic multiprocessor scheduling with few preemptions. Technical report, IPP-HURRAY Research Group. Institute Polytechnic Porto, HURRAY-TR-070501, Available at http://www.hurray.isep.ipp.pt, May 2007.

[3] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006.

[4] T. P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority EDF scheduling for hard real time. Technical report, Department of Computer Science, Florida State University, Tallahassee, July 2005.

[5] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996.

[6] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. Litmusrt: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, 2006.

[7] H. Cho, B. Ravindran, and E. D. Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 101–110, Rio de Janeiro, Brazil, Dec. 5–8, 2006.

[8] U. Devi and J. Anderson. Tardiness bounds for global edf scheduling on a multiprocessor. In *26th IEEE Real-Time Systems Symposium*, 2005.

[9] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[10] A. Khemka and R. K. Shyamasundar. Multiprocessor scheduling of periodic tasks in a hard real-time environment. In *Proc. of the International Parallel Processing Symposium*, 1992.

[11] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[12] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, 1994.

[13] D.-I. Oh and T. P. Baker. Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real Time Systems Journal*, 15(2):183–192, 1998.

[14] V. Padmanabhan and D. Roselli. The real cost of context switching. Technical report, UC Berkeley, 1994.

[15] F. Sebek. The real cost of task pre-emption - measuring real-time-related cache performance with a hw/sw hybrid technique. Technical report, Mälardalens Högskola, Västerås, August 2002.

[16] D. Zhu, D. Mossé, and R. Melhem. Multiple-resource periodic scheduling problem: how much fairness is necessary? In *Proc. of the IEEE Real-Time Systems Symposium*, pages 142–151, Cancun, Mexico, Dec. 3–5, 2003.

A. Background: Details

Example 2. *The most interesting part of the schedule in this example is shown in Figure A. Consider m+1 tasks to be scheduled on a multiprocessor with m identical processors where the scheduling algorithm is non-preemptive. Tasks are permitted to change priorities at any time. Because non-preemptive scheduling is used, the only migration permitted is that jobs of the same task may execute on different processors. We assume that the scheduling algorithm is permitted to insert idle time. Let the tasks be characterized as follows. T_1=1 and C_1=2. For all i=2..m+1: T_i= ϵ^{i-1}, C_i=2ϵ^i. We choose ϵ <1/2. We let O_i denote the time when τ_i arrives for the first time. The scheduling algorithm is not permitted to choose O_i. Let us consider an arbitrary job release by τ_1. Let A_1 denote the arrival time of that job and let s_1 denote the start time of the execution of that job. Because the execution time of τ_1 is twice as long as T_2 we know that every time τ_1 executes, τ_2 can arrive at least once such that $s_1 \leq A_2 < A_2$+$T_2 \leq s_1$+C_1, and hence there is a time when τ_1 and τ_2 must execute simultaneously. Repeating this argument, we obtain that there is a time when all tasks have to execute. All the m+1 tasks must execute on different processors and we only have m processors. Hence a deadline is missed. Here, U_s=2 $\cdot$ (m+1)/m. Choosing $\epsilon \rightarrow$ 0 yields that the utilization bound is zero. Naturally this example can be used to show that the utilization bound of every non-preemptive uniprocessor scheduling algorithm is zero as well. This reasoning can be performed for both periodically and sporadically arriving tasks. Consequently, the utilization bound of non-preemptive scheduling is zero for any number of processors and for both periodically arriving tasks and sporadically arriving tasks.*

Example 3. *Consider m processors and n=m + 1 tasks. The tasks are characterized by $\forall i \in$ 1..m+1: C_i=m/(m+1), T_i=1 and $m \geq 4$ and they all arrive simultaneously at time 0. Consider $[1/(2(m+1)), 1-1/(2(m+1))]$. In this time interval, a task must execute at least m/(m+1)-1/(m+1) $\geq$ 0.6 time units in order to meet deadlines. Since each task must execute at least 0.6 time units in this interval, at least one task is preempted in the time interval. Also, no tasks arrive in the time interval. Consequently, the number of preemptions divided by the number of arriving jobs in this interval is infinite.*

Example 4. *Consider n=6 tasks to be scheduled on m=5 processors. We consider the special case when all tasks arrive at time 0 and all tasks arrive periodically. $primes$(k) denotes the k^{th} prime number. Let T_1=2 $\cdot$ $primes$(5)=22, T_2=2 $\cdot$ $primes$(6)=26, T_3=2 $\cdot$ $primes$(7)=34, T_4=2 $\cdot$ $primes$(8)=38, T_5=2 $\cdot$ $prime$(9)=46 and T_6=2 $\cdot$ $primes$(10)=54. Let C_1=13, C_2=15, C_3=19, C_4=21, C_5=24 and C_6=28. This task set has U_s=0.6639. Let lcm denote lcm(T_1,T_2,...,T_n). lcm is 57366738 and the number of jobs during [0, lcm) is 10320350. We will now compute the number of preemptions. We will use pfair algorithms to schedule this task set and we will assume that the tasks arrive as frequently as permitted by the sporadic model.*

The schedule generated by the pfair algorithms, PD^2 [5], early-release ER pfair [1] and bounded fairness (BF) [16] are illustrated in Figure A. PD^2 causes 15.47 preemptions/job during [0, lcm). The result is 3.75 for early-release pfair, and 3.82 for bounded fairness. We have selected this example for illustrative purpose but there are task sets where the number of preemptions is even larger. If all T_i and C_i of tasks are multiplied by $k \cdot lcm$ (where k is a positive integer) then the number of preemptions per job becomes at least $k \cdot lcm$ times greater for PD^2. With this reasoning, it is easy to see that the number of preemptions divided by the number of jobs can approach infinity for PD^2. By multiplying C_i of all tasks by a constant such that U_s becomes 100% we obtain that ER obtains an infinite number of preemptions per job as well.

All these three algorithms use the concept of lag of a task τ_i at time t. The lag is defined as lag(τ_i,t)=t $\cdot$ C_i/T_i allocated(τ_i,[0,t)). (We let allocated(τ_i,[0,t)) denote the amount of time that τ_i was allocated during the time interval [0,t)). The reason for the difference in the number of preemptions is that the constraints on lag(i,t) are different. PD^2 requires at every t:-1<lag(i,t)<1. ER pfair requires at every t: lag(i,t)<1 and bounded fairness requires at those t where a task arrives:-1<lag(i,t)<1.

We can see that relaxing the pfair constraint, as BF does, is useful in order to reduce the number of preemptions. In this example we assumed that T_i and C_i are integers; this is different from what we assume in the other parts of the paper and the reason for doing this is that pfair algorithms are only defined for the case where T_i and C_i are integer.

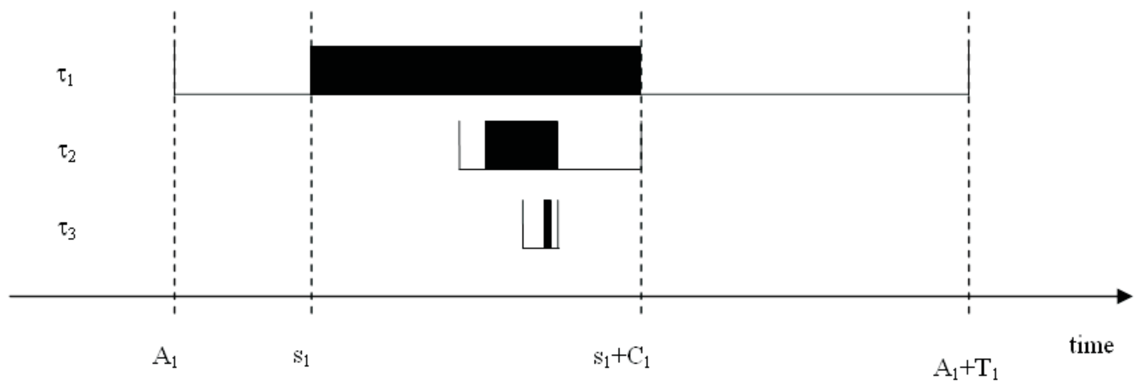


Figure 5. All non-preemptive scheduling algorithms have the utilization bound 0.

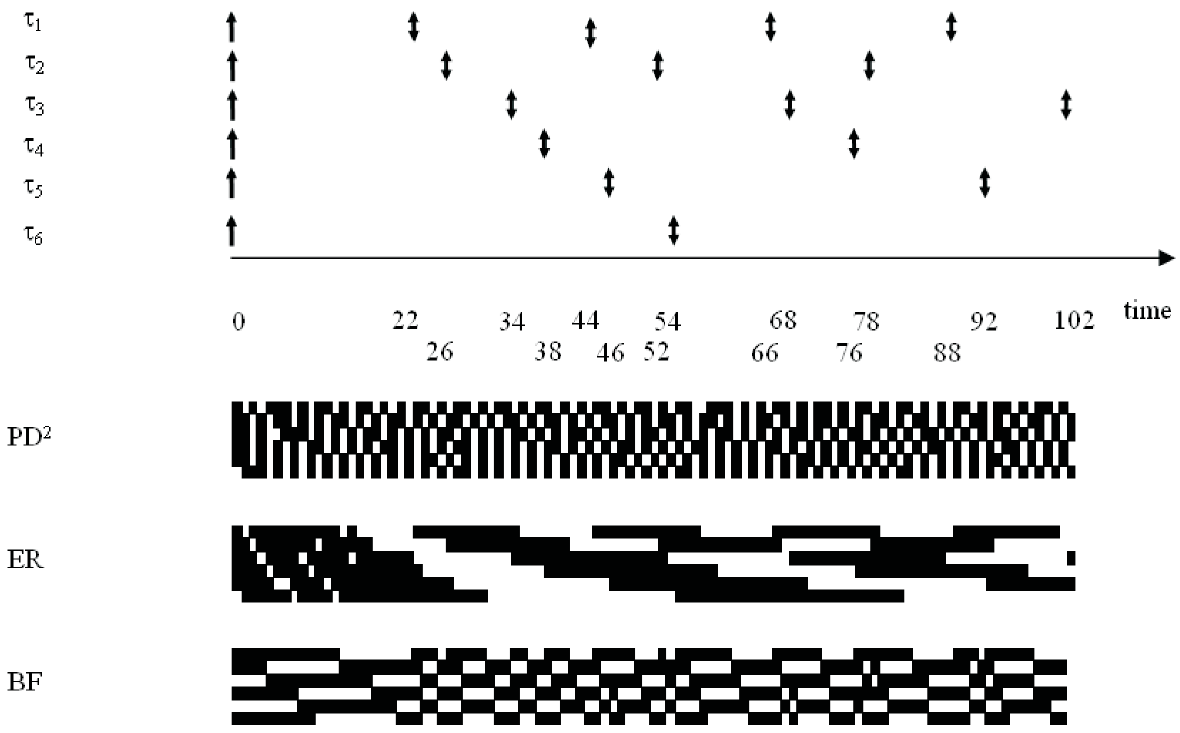


Figure 6. Pfair scheduling algorithms cause a large number of preemptions. The marks above the time line show the arrival times and deadlines of tasks. An arrow pointing upwards indicate an arrival time; an arrow pointing downwards indicate a deadlines. The Gantt charts below the time line show the schedule of PD^2, Early-Release Pfair and Bounded fairness. For each scheduling algorithm, the Gantt chart shows the execution of each of the 6 tasks. A black filled box indicates that a task executes; a white box indicates that a task does not execute. Consequently, a black box followed by a white box indicates a context switch. If a job makes a context switch but if it has remaining execution, then it is a preemption.