Technical Report

Multiprocessor Scheduling with Few Preemptions

Böjrn Andersson

Eduardo Tovar

TR-060811

Version: 1.0

Date: August 2006

Multiprocessor Scheduling with Few Preemptions

Böjrn ANDERSSON, Eduardo TOVAR

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, emt}@dei.isep.ipp.pt

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

Abstract

Consider the problem of scheduling a set of periodically arriving tasks on a multiprocessor with the goal of meeting deadlines. Processors are identical and have the same speed. Tasks can be preempted and they can migrate between processors. We propose an algorithm with a utilization bound of 66% and with few preemptions. It can trade a higher utilization bound for more preemption and in doing so it has a utilization bound of 100%.

Multiprocessor Scheduling with Few Preemptions

Björn Andersson and Eduardo Tovar
IPP Hurray Research Group
Polytechnic Institute of Porto, Portugal
{bandersson,emt}@dei.isep.ipp.pt

Abstract

Consider the problem of scheduling a set of periodically arriving tasks on a multiprocessor with the goal of meeting deadlines. Processors are identical and have the same speed. Tasks can be preempted and they can migrate between processors. We propose an algorithm with a utilization bound of 66% and with few preemptions. It can trade a higher utilization bound for more preemption and in doing so it has a utilization bound of 100%.

1. Introduction

Consider the problem of preemptive scheduling of n periodically arriving tasks on m identical processors. A task τ_i can arrive at any time when it arrives for the first time, but then it arrives periodically with a period T_i. Every time task τ_i arrives, it needs to execute C_i time units before it arrives again. A processor can execute at most one task at a time, and a task cannot execute on two or more processors simultaneously. The utilization is defined as $U_s = (1/m) \times \sum C_i / T_i$. 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 algorithm A.

The design space of preemptive multiprocessor scheduling algorithms can be categorized as *partitioned* or *global scheduling* [1, 2]. Global scheduling algorithms store in one queue (shared among all processors) all tasks that have arrived but have not finished their execution. At any time instant the m highest-priority tasks in that queue are selected for execution on the m processors using preemption and migration if necessary. 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 into m uniprocessor scheduling problems. This simplifies scheduling and schedulability, since a large number of results available for uniprocessor scheduling can then be reused. Unfortunately, all partitioned multiprocessor scheduling algorithms have a utilization bound of 50% or less [3]. Conversely, global scheduling can achieve a utilization bound of 100% by using a family of algorithms called *pfair scheduling* [4, 5]. Regrettably, this 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 significantly high. We believe (as does Baker [6]) that it is desirable to achieve a higher (>50%) utilization bound without incurring the cost of an undesirable number of preemptions.

Therefore, in this paper we propose a multiprocessor scheduling algorithm with a utilization bound of 66%. It causes provably few preemptions: the number of preemptions divided by the number of jobs is at most 4. Of those algorithms in previous works that achieve the utilization bound (66%) of our algorithm, none of them has a finite number that bounds the number of preemptions divided by the number of jobs.

The remainder of this paper is organized as follows. Section 2 presents our new algorithm. Section 3 proves its utilization bound and Section 4 proves an upper bound on the number of preemptions per job. Section 5 offers discussion on previous work and conclusions.

2. The new algorithm

In order to understand the design of the new algorithm, we will first (in Section 2.1) consider partitioned scheduling of a specific task set example, with the purpose of stressing how partitioned scheduling may perform poorly. The reasoning on the example will provide the guiding principles on the design of the proposed new scheduling algorithm, which will then be formally presented in Sections 2.2 and 2.3.

2.1. Understanding the problem

Consider the following partitioned scheduling example: m processors and $n = m + 1$ tasks τ_i with $T_i = 1$ and $C_i = 0.5 + \varepsilon$.

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 $1 + 2\varepsilon$, and this is more than 100%. Essentially, by choosing $m \to \infty$ and $\varepsilon \to 0$ the task set will have $U_s \to 0.5$ and, above all, 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 split into two sub-tasks and these sub-tasks are assigned to two different processors, then it is possible to assign tasks such that the utilization on every processor reaches 100%.

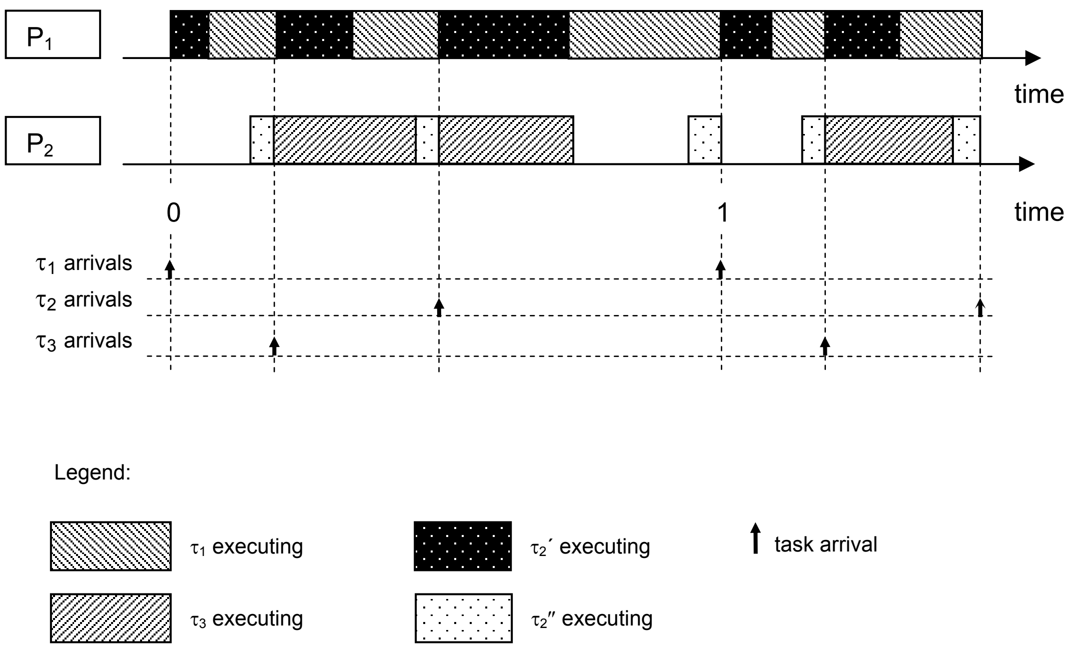

Figure 1: Schedule for the example task set using task splitting

Note, however, that sub-tasks of the same task cannot execute simultaneously. A solution to this is a uniprocessor scheduling algorithm that is aware of tasks on other processors, so that a sub-task on one processor is not executed (even partially) at the same time as its corresponding sub-task on another processor.

Let us denote the two sub-tasks of a task τ_i as τ_i' and τ_i''. Note that τ_i' and τ_i'' will have the same periods and will arrive at the same time. When a task (or sub-task) arrives on any processor, let t_0 denote the arrival time, and let t_1 denote the time of the next arrival of a job on any processor. Task τ_i' executes $(C_i' / T_i') \times (t_1 - t_0)$ time units without preemption and starts executing at time t_0. Task τ_i'' executes $(C_i''/ T_i'') \times (t_1 - t_0)$ time units without preemption and finishes execution at time t_1. Note that τ_i' and τ_i'' execute on different processors and their executions will not overlap in time since the original task τ_i had $C_i/T_i \leq 1$ and hence $C_i' / T_i' + C_i''/ T_i'' \leq 1$. We schedule these sub-tasks with the highest priority. The other tasks are scheduled according to EDF and have lower priority than the sub-tasks. Applying this approach to the same task set example, would correspond to the schedule illustrated in Figure 1. There are three tasks $\{\tau_1, \tau_2, \tau_3\}$, with $T_i = 1$ and $C_i = 0.5 + \varepsilon$. These tasks are scheduled on two processors ($m = 2$): P_1 and P_2. The task τ_2 has been split into two sub-tasks, τ_2' and τ_2'', with $T_2' = T_2'' = 1$ and $C_2' = 0.5 - \varepsilon$, $C_2'' = 2\varepsilon$. Using the task splitting approach the task set is schedulable while it would not be using partitioned scheduling.

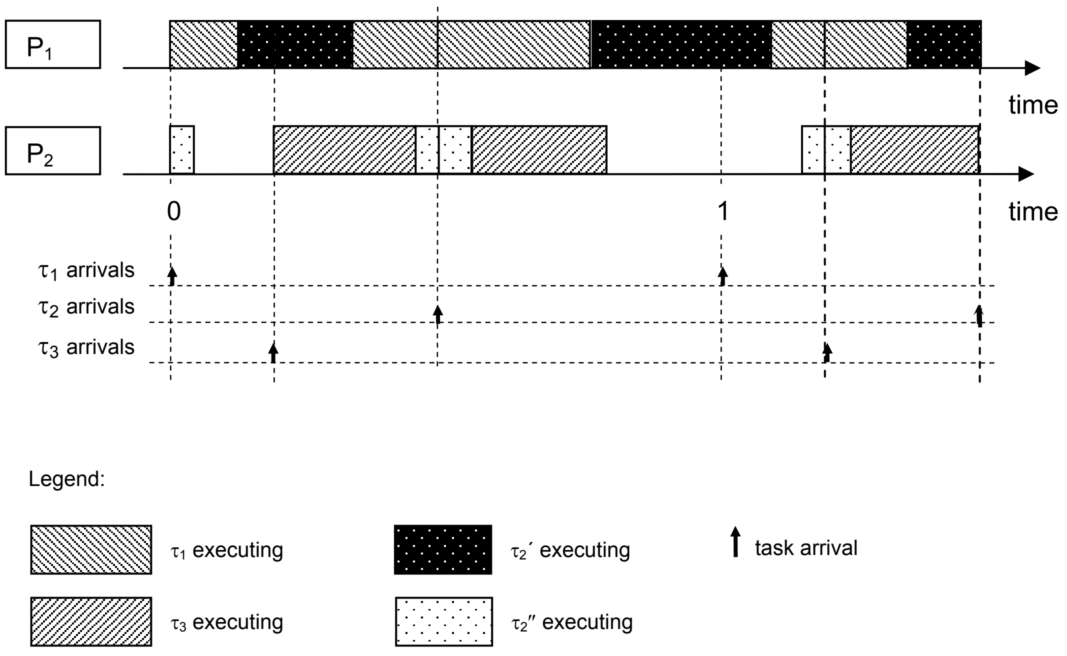

Figure 2: Using mirroring on the example in Figure 1 decreases the number of preemptions

In the approach illustrated in Figure 1, task τ_i' is always executed at the beginning of the time interval between any two job arrivals while task τ_i'' is executed at the end of that time interval. However, by mirroring the schedule of the split tasks in half of the time intervals fewer preemptions would result. Figure 2 illustrates this for the same task set example.

2.2. Task assignment

We will now describe a general algorithm for assigning tasks to processors and scheduling tasks on a uniprocessor. The algorithm for assigning tasks to processors is given as pseudo-code in Algorithm 1 (Figure 3). The algorithm assigns tasks to processors such that on all processors the utilization does not exceed 100%. It has a parameter k which should be selected by the designer such that $1 \leq k \leq m$. 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 value of *SEP* is computed at line 4 in Algorithm 1, by calling a function described in Algorithm 3 (Figure 5). *SEP* will be used later on in Lemma 1 (Section 3), and the rationale for how to select it will be better perceived there. For the moment it is sufficient to realize that we select *SEP* as:

$$SEP = \begin{cases} \dfrac{k}{k+1} & k < m \\ 1 & k = m \end{cases} \tag{1}$$

First, the algorithm assigns heavy tasks to their own dedicated processors (at lines 10-13 in Algorithm 1). Doing so at this early stage in the algorithm improves performance, since the rest of it is devoted to dealing with the light tasks. The main

```
1.  for p in 1..m do
2.    U[p] := 0   $\tau$[p] := {}
3.  end for
4.  $SEP$ := calc_SEP
5.  Let $\tau^{heavy}$ denote the set of tasks with $C_i/T_i>SEP$
6.  Let $\tau^{light}$ denote the set of tasks with $C_i/T_i<=SEP$
7.  L := |$\tau^{heavy}$|
8.  Order tasks such that $\tau_i$ with i in 1..L are all in $\tau^{heavy}$ and $\tau_i$ with i
    in L+1..n are all in $\tau^{light}$
9.  if |$\tau^{heavy}$|<=m then
10.   for i in 1..L do
11.     p := i
12.     $\tau$[p] := $\tau$[p] $\cup${$\tau_i$}         U[p] := U[p] + $C_i/T_i$
13.   end for
14.   if |$\tau^{light}$|>0 then
15.     if L+1<=m then
16.       p := L+1
17.       for i in L+1..n do
18.         if U[p]+$C_i/T_i$<=1 then
19.           $\tau$[p] := $\tau$[p] $\cup${$\tau_i$} U[p] := U[p] + $C_i/T_i$
20.         else
21.           if p+1<=m then
22.             if (p-L) mod k=0 then
23.               p := p+1
24.               $\tau$[p] := $\tau$[p] $\cup${$\tau_i$}         U[p] := U[p] + $C_i/T_i$
25.             else
26.               split task $\tau_i$ into $\tau_i'$ and $\tau_i''$ such that
                          $T_i' = T_i$
                          $T_i'' = T_i$
                          $C_i' = (1- U[p]) * T_i'$
                          $C_i'' = (C_i/T_i - C_i'/T_i') * T_i''$
27.               $\tau$[p]   := $\tau$[p]$\cup${$\tau_i'$}         U[p]   := U[p]   + $C_i'/T_i'$
28.               $\tau$[p+1] := $\tau$[p+1]$\cup${$\tau_i''$}       U[p+1] := U[p+1] + $C_i''/T_i''$
29.               p := p+1
30.             end if
31.           else
32.             declare FAILURE
33.           end if
34.         end if
35.       end for
36.     else
37.       declare FAILURE
38.     end if
39.   else
40.     declare SUCCESS
41.   end if
42. else
43.   declare FAILURE
44. end if
```

Figure 3: Algorithm 1 (for task assignment)

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 18 in Algorithm 1. If the condition stated in that line (a schedulability test for EDF [7]) is true then the task can be assigned to processor *p*. If the condition is false, the task is split into two portions (at line 26) and assigned to the current processor *p* and processor *p* + 1 (lines 27-28). Then the processor with a higher index is considered (line 29).

```
for processors with index p in 1..L:
  when the system starts do
    do nothing
  end do
  when a task arrives on processor p do
    execute that task
  end do

for processors with index p in L+1..m:
  when the system starts do
    mirrorflag[p]:= false
    minindex[p]:=L+⌊(p-L)/k⌋*k+1
    maxindex[p]:=L+⌊(p-L)/k⌋*k+k
    if maxindex[p]>m then
      maxindex[p]:= m
    end if
    wait until all processors with index within range
      minindex[p]..maxindex[p] have reached this line
  end do
  when any task arrives on a processor with index in [minindex[p]..maxindex[p]] do
    t0:=current time
    t1:=next time a task arrives on a processor with index within range
      minindex[p]..maxindex[p]
    call calc_splitted_tasks // here we calculate timea_and_timeb
                             // we also calculate firsttask and lasttask
    if firsttask[p]=NULL then
      idle processor p during [t0,timea[p])
    else
      execute firsttask[p] on processor p during [t0,timea[p])
    end if
    call schedule_tasks_with_EDF
    if lasttask[p]=NULL then
      idle processor p during [timeb[p],t1)
    else
      execute lasttask[p] on processor p during [timeb[p],t1)
    end if
    mirrorflag[p]:=not (mirrorflag[p])
  end do
```

Figure 4: Algorithm 2 (run-time dispatching)

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 indexes ranging from 1 up to L, while light tasks are assigned to processors with indexes ranging from $L + 1$ up to m. The light tasks are assigned to different groups of processors, and there are at most k processors in a group. Processors with indexes in the range $\{L + 1..L + k\}$ correspond to one group. Processors with indexes in the range $\{L + k + 1..L + 2k\}$ correspond to another group, and so forth. If a task is attempted to be assigned to the last processor in a group and it fails, then it is not split; instead it is simply assigned to the next processor in a new group. Lines 23-24 take care of this situation in Algorithm 1. This ensures that tasks in one group do not interact with tasks in another group. This algorithm is actually similar to the *next-fit bin-packing* algorithm [2]. It differs however since task splitting is permitted in our approach.

```
1.   procedure calc_splitted_tasks is
2.   begin
3.     if mirrorflag[p] then
4.       if there is a task $\tau_i'$ assigned
           to processor p then
5.         lasttask[p]:= $\tau_i'$
6.         lastdur[p]:= $(C_i'/T_i')$*(t1-t0)
7.       else
8.         lasttask[p]:= NULL
9.         lastdur[p] := 0
10.      end
11.      if there is a task $\tau_i''$ assigned
           to processor p then
12.        firsttask[p] := $\tau_i''$
13.        firstdur[p]:=$(C_i''/T_i'')$*(t1-t0)
14.      else
15.        firsttask[p]:= NULL
16.        firstdur[p]:= 0
17.      end
18.    else
19.      if there is a task $\tau_i'$ assigned
           to processor p then
20.        firsttask[p]:= $\tau_i'$
21.        firstdur[p]:=$(C_i'/T_i')$*(t1-t0)
22.      else
23.        firsttask[p] := NULL
24.        firstdur[p]:= 0
25.      end
26.      if there is a task $\tau_i''$ assigned
           to processor p then
27.          lasttask[p]:= $\tau_i''$
28.          lastdur[p]:=$(C_i''/T_i'')$*(t1-t0)
29.      else
30.        lasttask[p]:= NULL
31.        lastdur[p] := 0
32.      end
33.    end if
34.    timea[p]:= t0+firstdur[p]
35.    timeb[p]:= t1-lastdur[p]
36.  end
37.

38.  procedure schedule_tasks_with_EDF is
39.  begin
40.    let ready[p] denote the set of
         tasks $\tau$[p]\{firsttask[p],lasttask[p]}
         such that they have a job which
         has arrived at t0 or earlier
         but the job has remaining
         execution at time t0
41.    t[p]:= timea[p]
42.    execute_entire_job[p]:=true
43.    while execute_entire_job[p] do
44.      current_task[p]:= dequeue the
           task with the least
           absolute deadline from
           ready[p]
45.      if current_task[p]=NULL then
46.        execute_entire_job[p]:= false
47.      else
48.        remain[p] := the remaining execution
          time of the job of current_task
49.        if t[p]+remain[p]<=timeb[p] then
50.          execute_entire_job[p]:=true
51.          execute the job of current_task
               during [t[p],t[p]+remain[p])
53.          t[p]:=t[p]+remain[p]
54.        else
55.          execute_entire_job[p]:=false
56.        end if
57.      end if
58.    end while
59.    if current_task[p]=NULL then
60.      idle processor p until timeb[p]
61.    else
62.      execute the job of current_task
         during [t[p], timeb[p])
63.    end if
64.  end
65.
66.  function calc_SEP return real is
67.  begin
68.    if k<m then
69.      return k/(k+1)
70.    else
71.      return 1
72.    end if
73.  end
```

Figure 5: Algorithm 3 (auxiliary procedures and functions)

2.3. Dispatching

We will now turn our attention to the problem of run-time dispatching. Our approach was already informally described in Section 2.1, and will now be more formally described through Algorithm 2 (Figure 4). Algorithm 2 calls two subroutines; these are described in Algorithm 3 (Figure 5). All variables in Algorithm 2 and Algorithm 3 are global variables. The dispatching of heavy tasks is described on lines 1-7 in Algorithm 2. It entails a simple approach: whenever a task arrives, it executes on its assigned processor.

A light task is assigned to a group of processors, and whenever a task arrives in that group of processors, dispatchers are executed on all processors in that group. t_0 denotes the time when a task arrives, and t_1 denotes the time when any task in that group arrives next. This is computed on lines 20-21 within Algorithm 2. After that, Algorithm 2 calculates two time instants, `timea` and `timeb`, when tasks should be preempted. One of the split tasks is executed before `timea` (line 26) and the other split task is executed after `timeb` (line 32). During the time span [`timea`,`timeb`) the non-split tasks are scheduled according to EDF. This is performed by the subroutine `schedule_tasks_with_EDF` described within Algorithm 3. If a non-split task finishes its execution during the time span [`timea`,`timeb`) then the next non-split task with the earliest deadline is selected. We denote the approach consisting of the use of Algorithms 1, 2 and 3 as *EKG* approach, as a short-hand notation for ***E****DF with task splitting and* ***k*** *processors in a* ***g****roup.*

3. Proving the utilization bound

The aim of this section is to prove that the utilization bound of the EKG approach is *SEP* (as given by Equation (1), in Section 2.2). In order to do this, in Section 3.1, we prove that the task assignment scheme (Algorithm 1) declares success for all task sets with utilization lower than or equal to the utilization bound. Lemma 2 states it and Lemma 1 is used to prove it. We also prove certain properties of the distribution of the execution of the task assignment in Lemmas 3-4.

In Section 3.2 we show that if the properties assured by the task assignment scheme are true, then using the dispatcher (Algorithm 2) will enable all tasks to meet their deadlines. This proves the utilization bound and Theorem 1 states this.

3.1. Task assignment

Lemma 1. If Algorithm 1 declares failure then the task set satisfies $U_s > SEP$.

Proof: One possibility is that the algorithm declared failure on line 43. If so, then all tasks that were assigned to processors had $C_i / T_i > SEP$. Since there are $m + 1$ or more tasks, it follows that:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > \frac{1}{m}\times SEP\times(m+1) > SEP \tag{2}$$

Another possibility is that the algorithm declared failure on line 37. If this was the case then all m processors were assigned heavy tasks and then there was a light task in the task set that could not be assigned. This would imply the following:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > \frac{1}{m}\times SEP\times m = SEP \tag{3}$$

Yet another possibility would be that the algorithm declared failure on line 32. Before that it must have been a task (denoted with index f) that was considered ($i = f$) on line 18, and the condition on that line was evaluated false. We observe that the utilization of the task set is no less than the sum of the utilization of all the processors when Algorithm 1 declared failure added to the utilization of the task *f*. Thus, it follows that:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > \frac{1}{m}\times\left(\left(\sum_{p=1}^{m}U[p]\right)+\frac{C_f}{T_f}\right) \tag{4}$$

Inequality (4) can be re-written as follows to better express the fact that the overall utilization is equal to the sum of the utilizations of each individual sub-set of processors:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > \frac{1}{m}\times\left(\sum_{1\le p\le L}U[p]+\sum_{(L+1\le p\le m-1)\wedge((p-L)\bmod k=0)}U[p]+\sum_{(L+1\le p\le m-1)\wedge((p-L)\bmod k\neq 0)}U[p]+\sum_{p=m}U[p]+\frac{C_f}{T_f}\right) \tag{5}$$

Since all processors p with indexes $1 \le p \le L$ have $U[p] > SEP$ and $U[m] + C_f / T_f > 1$ (because the algorithm declared failure), inequality (5) can be re-written as follows:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > \frac{1}{m}\times\left(L\times SEP+\sum_{(L+1\le p\le m-1)\wedge((p-L)\bmod k=0)}U[p]+\sum_{(L+1\le p\le m-1)\wedge((p-L)\bmod k\neq 0)}U[p]+1\right) \tag{6}$$

Let us now consider the set of processors p with index $L + 1 \le p \le m - 1$. There are $(m - L - 1)$ of these processors. For some of these processors, the condition at line 22 in Algorithm 1 resulted true. This happens when $(p - L) \bmod k = 0$. For the task i, considered to be assigned to processor p, it must have been that $U[p] + C_i / T_i > 1$. Actually there are $\lfloor (m - L - 1) / k \rfloor$ of such processors. Since task i is light then $U[p] > (1 - SEP)$. The other processors have $U[p] = 1$ when Algorithm 1 declared failure (the lines 26 and 27 guarantee that). Taking this reasoning into account, we can now state the following:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > \frac{1}{m}\times\left(L\times SEP+\left\lfloor\frac{m-L-1}{k}\right\rfloor\times(1-SEP)+m-L-1-\left\lfloor\frac{m-L-1}{k}\right\rfloor+1\right) \tag{7}$$

We can obtain a lower bound on the right-hand side of (7). For details, see Appendix A in [8] . This gives us:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > SEP \tag{8}$$

All cases where Algorithm 1 could have declared failure have been explored. If Algorithm 1 declares failure then inequalities (2), (3) and (8) enforce that $U_s > SEP$. This proves Lemma 1.

□

Lemma 2. If a task set satisfies $U_s \le SEP$ and Algorithm 1 is used then Algorithm 1 declares success.

Proof: Follows from Lemma 1 and the fact that Algorithm 1 terminates on every input.

□

Lemma 3. If Algorithm 1 declares success then $\forall p$: $U[p] \le 1$.

Proof: For those processors p with $p \leq L$, the claim that U[p] ≤ 1 follows from the fact that $C_i / T_i \leq 1$ and the observation that Algorithm 1 lines 10-13 only assigns one task to each processor. For those processors p with $p \geq L + 1$, the claim U[p] ≤ 1 follows from the actions taken by Algorithm 1 in line 18, line 24 and lines 26-28.

□

Lemma 4. If Algorithm 1 declares success then $\forall \tau_i$ that is split, $C_i' / T_i' + C_i'' / T_i'' \leq 1$.

Proof: Follows from line 26 in Algorithm 1 and the fact that before a task is split it satisfied the condition $C_i / T_i \leq 1$.

□

3.2. Dispatching

Lemma 5. Consider a time interval $[t_0, t_1)$ such that a task arrives at time t_0 and a task arrives at time t_1 but no tasks arrive during (t_0, t_1). If ($\forall p$: U[p] ≤ 1) and (for all split tasks it holds that $C_i' / T_i' + C_i'' / T_i'' \leq 1$) and Algorithm 2 is used to schedule tasks on each processor in the group, then a task τ_i that was split by Algorithm 1 never executes on two or more processors simultaneously during the time span $[t_0, t_1)$.

Proof: Let p denote the processor to which τ_i' is assigned, and let $p + 1$ denote the processor to which τ_i'' is assigned. From Algorithm 2 it holds, for every processor p and every processor q, that `mirrorflag[p]= mirrorflag[q],` since `mirrorflag[p]` changes only when a task arrives. We will consider two cases.

Case 1. `mirrorflag[p]=false` during the time span $[t_0, t_1)$.

From Algorithm 2, τ_i' will execute during $[t_0,$ `timea`$)$ on processor p and τ_i'' will execute during [`timeb`, $t_1)$ on processor $p + 1$. An assumption associated to the lemma is that $C_i' / T_i' + C_i'' / T_i'' \leq 1$. Hence:

$$0 \leq (t_1 - t_0) \times \left(1 - \left(C_i'/T_i' + C_i''/T_i'\right)\right) \tag{9}$$

which can be re-written as follows:

$$0 \leq (t_1 - (t_1 - t_0) \times C_i''/T_i'') - (t_0 + (t_1 - t_0) \times C_i'/T_i') \tag{10}$$

From Algorithm 3 (lines 34-35), the two terms in (10) correspond to `timea` and `timeb,` and therefore it results that (10) can be re-written as follows:

$$0 \leq timeb - timea \tag{11}$$

Clearly, from (11), `timea` $\leq$ `timeb` and this assures that τ_i' and τ_i'' do not execute simultaneously during the time span $[t_0, t_1)$.

Case 2. `mirrorflag[p]=true` the time span $[t_0, t_1)$

The reasoning is similar to Case 1.

Thus, regardless of which one of the two cases is true, it holds that a split task does not execute simultaneously on two or more processors.

□

Lemma 6. If ($\forall p$: U[p] ≤ 1) and (for all split tasks it holds that $C_i' / T_i' + C_i'' / T_i'' \leq 1$) and (Algorithm 2 is used to schedule tasks on each processor in the group) then a task never executes on two or more processors simultaneously.

Proof: We will prove this lemma by proving that an arbitrary task τ_i never executes on two or more processors simultaneously.

Case 1. The task τ_i was not split by Algorithm 1.

Case 1a). τ_i is assigned to a processor p with $p \leq L$. This task is assigned to a processor and does not execute on any other processor. Hence τ_i never executes on two or more processors simultaneously.

Case 1b). τ_i is assigned to a processor p with $L + 1 \leq p$. This case is similar to Case 1a); but on a processor p that satisfies $L + 1 \leq p$ there may be other tasks executing. Nevertheless, the reasoning in Case 1a) can be used in this case too, and thus supporting that a task τ_i never executes on two or more processors simultaneously.

Case 2. The task τ_i was split by Algorithm 1.

Consider a time interval $[t_0, t_1)$ such that a task arrives at time t_0 and a task arrives at time t_1 but no tasks arrive during (t_0, t_1). From Lemma 5 it results that τ_i' and τ_i''never execute simultaneously during $[t_0, t_1)$. This argument can be repeated for every time interval between any two consecutive task arrivals, and hence it results that τ_i never executes simultaneously on two or more processors.

Therefore, regardless of which one of the cases is true, the lemma holds.

□

Lemma 7. Consider a time interval $[t_0, t_1)$ such that a task arrives at time t_0 and a task arrives at time t_1 but no tasks arrive during (t_0, t_1). If ($\forall p$: U[p] ≤ 1) and (for all split tasks it holds that $C_i' / T_i' + C_i'' / T_i'' \leq 1$) and (Algorithm 2 is used to schedule tasks on each processor in the group) then a task τ_i that was split by Algorithm 1 executes $(C_i / T_i) \times (t_1 - t_0)$ time units during $[t_0, t_1)$.

Proof: Let p denote the processor to which τ_i' is assigned, and let $p + 1$ denote the processor to which τ_i'' is assigned. From Algorithm 2, and for every processor p and every processor q, it holds that `mirrorflag[p]=mirrorflag[q]` since `mirroflag[p]` changes only when a task arrives.

We will consider two cases.

Case 1. `mirrorflag[p]=false` during the time span $[t_0, t_1)$.

From Algorithm 2, it results that τ_i' executes during $[t_0,$ `timea`$)$ on processor p and τ_i'' executes during [`timeb`, $t_1)$ on processor $p + 1$. Hence, the amount that τ_i executes during $[t_0, t_1)$ is given by:

$$(t_1 - timeb) + (timea - t_0) \tag{12}$$

As reasoned previously in the proof of Lemma 5, Algorithm 2 states that:

$$\begin{cases} timea = t_0 + (t_1 - t_0) \times C_i' / T_i' \\ timeb = t_1 - (t_1 - t_0) \times C_i'' / T_i'' \end{cases} \tag{13}$$

and hence, (12) can be re-written as follows:

$$(t_1 - t_0) \times C_i' / T_i' + (t_1 - t_0) \times C_i'' / T_i'' \tag{14}$$

Since from Algorithm 1 (line 26) a split task τ_i was split such that $C_i' / T_i' + C_i''/ T_i'' = C_i / T_i$ and $T_i' = T_i'' = T_i$, then the amount of execution performed by τ_i during the time interval $[t_0, t_1)$ is given by:

$$(t_1 - t_0) \times C_i / T_i \tag{15}$$

Case 2. `mirrorflag[p]=true` during the time span $[t_0, t_1)$

The reasoning is similar to Case 1.

Thus, and regardless of which case occurs, the statement of the lemma is obtained.

□

Lemma 8. If ($\forall p$: `U`$[p] \le 1$) and (for all split tasks it holds that $C_i' / T_i' + C_i'' / T_i'' \le 1$) and (Algorithm 2 is used to schedule tasks on each processor in the group) then deadlines are met for all tasks τ_i such that (τ_i is assigned to a processor p with $L + 1 \le p$) $\wedge$ (τ_i is not split).

Proof: We will make definitions and algebraic manipulations in the first part of the proof and use them to prove the lemma by contradiction in a second part of the proof. 2.

1. Let $\tau^{not\_split}[p]$ denote the set of tasks such that each task in $\tau^{not\_split}[p]$ is (τ_i is assigned a processor p with $L + 1 \le p$) $\wedge$ (τ_i is not split). Let τ_a' denote the prime-task assigned to processor p and let τ_b'' denote the bis-task assigned to processor p. Note that τ_a' and τ_b'' do not belong to the same original task. From Algorithm 1 it results that:

$$\left(\sum_{i \in \tau^{not\_split}[p]} \frac{C_i}{T_i} \right) + \frac{C_a'}{T_a'} + \frac{C_b''}{T_b''} \le 1 \tag{16}$$

which can be re-written as follows:

$$\left(\sum_{i\in\tau^{not\_split}[p]}\frac{C_i}{T_i}\right)\leq 1-\left(\frac{C_a^{'}}{T_a^{'}}+\frac{C_b^{"}}{T_b^{"}}\right) \tag{17}$$

Both sides of inequality (17) can be multiplied by *S*, where *S* is any positive real number. In doing it and applying an algebraic re-writing, the following inequality holds:

$$\forall S>0:\sum_{i\in\tau^{not\_split}[p]}\left(\left\lfloor\frac{S-T_i}{T_i}\right\rfloor+1\right)\times C_i\leq S\times\left(1-\left(\frac{C_a}{T_a}+\frac{C_b}{T_b}\right)\right) \tag{18}$$

2. Let us now consider the task set $\tau^{not\_split}[p]$, and prove that all tasks in $\tau^{not\_split}[p]$ meet their deadlines. We prove it using contradiction. Suppose that a task in $\tau^{not\_split}[p]$ missed a deadline. Let us study the busy period [9] before the first deadline miss. Our busy period is defined slightly different from the definition in [9]; our busy period starts when a task arrives and ends when the first deadline miss occurs. During this time period, it is permitted that processor *p* is idle. However it must be that `ready[p]` (see line 40 in Algorithm 3) is non-empty. Just before the beginning of the busy period, `ready[p]` (line 40 in Algorithm 3) is empty. Since the tasks in $\tau^{not\_split}[p]$ are scheduled with preemptive EDF (pseudo-code in Algorithm 3 – Figure 5) it results that for a deadline miss to occur, it must have been that there is a time interval *Q* such that (*Q* is a subset of our busy period) $\wedge$ (*Q* starts when a task arrives and ends when the deadline is missed) $\wedge$ (the supply of processing time [9] for the tasks in τ^{not_split}[p] during *Q* was less than the demand of processing time from tasks in τ^{not_split}[p] during *Q*.). Let *LQ* denote the length of *Q*. It is known that the demand is calculated as follows:

$$\sum_{i\in\tau^{not\_split}[p]}\left(\left\lfloor\frac{LQ-T_i}{T_i}\right\rfloor+1\right)\times C_i \tag{19}$$

Let us consider a time interval $[t_0, t_1)$ such that at time t_0, a task arrives and at time t_1 a task arrives but during (t_0, t_1) no tasks arrive. Let us consider a task $\tau_a{'}$ and a task $\tau_b{'}$ which are split tasks and are assigned to the same processor *p*. Note that they do not belong to the same original task. It holds that the task $\tau_a{'}$ executes for $(C_a{'} / T_a{'})\times(t_1 - t_0)$ time units and the task $\tau_b{'}$ executes for $(C_b{''}/ T_b{''})\times(t_1 - t_0)$ time units. This holds for every such time $[t_0, t_1)$. Hence, the supply of processing time for the tasks in τ^{not_split}[p] in the time interval $[t_0, t_1)$ is given by:

$$(t_1-t_0)\times\left(1-\left(\frac{C_a^{'}}{T_a^{'}}+\frac{C_b^{''}}{T_b^{''}}\right)\right) \tag{20}$$

This is true regardless of the value of `mirror[p]`. Observe that the time interval Q can be subdivided into intervals where inequality (20) can be applied. The repeated application of (20) to all those intervals yields that the supply of processing time for the task in $\tau^{not\_split}[p]$ during Q is:

$$LQ \times \left(1 - \left(\frac{C_a^{'}}{T_a^{'}} + \frac{C_b^{''}}{T_b^{''}}\right)\right) \tag{21}$$

It is known from previous research [9] that since a deadline is missed the demand exceeded the supply. Hence the following inequality will hold:

$$\sum_{i \in \tau^{not\_split}[p]} \left(\left\lfloor \frac{LQ - T_i}{T_i} \right\rfloor + 1 \right) \times C_i > LQ \times \left(1 - \left(\frac{C_a^{'}}{T_a^{'}} + \frac{C_b^{''}}{T_b^{''}}\right)\right) \tag{22}$$

But this contradicts (18). Hence, all deadlines of tasks in $\tau^{not\_splitted}[p]$ are met.

□

Lemma 9. If ($\forall p$: U[p] ≤ 1) and (for all split tasks it holds that $C_i' / T_i' + C_i'' / T_i'' \le 1$) and (Algorithm 2 is used to schedule tasks on each processor in the group) then all deadlines are met.

Proof: We will prove this lemma by proving that an arbitrary task τ_i meets its deadline.

Case 1. The task τ_i was not split in Algorithm 1.

Case 1a). τ_i is assigned to a processor p with $p \le L$.

There are no other tasks on this processor. Since $C_i \le T_i$, the task meets its deadlines.

Case 1b). τ_i is assigned to a processor p with $L + 1 \le p$.

It results from Lemma 8 that τ_i meets its deadlines.

Case 2. The task τ_i was split in Algorithm 1.

Consider a time interval $[t_0, t_1)$ such that a task arrives at time t_0 and a task arrives at time t_1 but no tasks arrive during (t_0, t_1). From Lemma 7 it results that τ_i executes $(C_i / T_i) \times (t_1 - t_0)$ time units during $[t_0, t_1)$. This argument can be repeated for every time interval between two task arrivals, and hence it results that τ_i meets all its deadlines.

Therefore, regardless of which one of the cases is true, the lemma holds.

□

We will finalise Section 3 by stating the following theorem.

Theorem 1. If a task set satisfies the condition $U_s \le SEP$ and `EKG` is used, then all deadlines are met and a task never executes on two or more processors simultaneously.

Proof: It follows from Lemma 6 and Lemma 9.

□

4. Proving a bound on the number of preemptions

In this section we will state (Theorem 2) a bound on the number of preemptions divided by the number of jobs for the entire task set. Its proof depends on Lemma 10, which studies the special case of tasks in one group of processors. We let *lcm* denote the least common multiple of periods of the tasks. Since Lemma 10 and Theorem 2 makes claims about the number of preemptions per job, it is necessary to define whether a split task is one task or two tasks; analogously for jobs. In this section, in Lemma 10 and Theorem 2, we say that if a task is split then it is counted as one task. The jobs from these tasks are split. We count such a split job as one job. We think this is the right way of counting because the splitting of tasks and jobs is only used internally in the scheduling algorithm; the application does not know that a task or a job is split.

Lemma 10. Consider the case when all tasks arrive at time 0 and they are scheduled using `EKG`. Consider the processors in one of the groups consisting of *k* processors. We claim that the number of preemptions during [0, *lcm*) divided by the number of jobs that executed during [0, *lcm*) is at most 2*k*.

Proof: See Appendix B in [8]. □

Theorem 2. Consider the case when all tasks arrive at time 0 and they are scheduled using `EKG`. We claim that the number of preemptions during [0, *lcm*) divided by the number of jobs that executed during [0, *lcm*) is at most 2*k*.

Proof: From Lemma 10 we know that for the groups with *k* processors, the number of preemptions divided by the number of jobs is at most $2 \times k$. The last group may consist of less than *k* processors; nevertheless, it holds that the number of preemptions divided by the number of jobs is at most $2 \times k$. The processors with index $p \leq L$ have only one task per processor so no preemptions can occur there. Taking these three facts together imply that the number of preemptions on all processors during [0, *lcm*) divided by the number of jobs during [0, *lcm*) in the entire task set is no greater than $2 \times k$. □

5. Conclusions

We have presented an algorithm to schedule tasks to meet deadlines on a multiprocessor. By selecting $k = 2$, Theorem 1 tells us that the utilization bound of the algorithm is 66% and Theorem 2 tells us that it causes at most 4 preemptions per job on average per hyperperiod. A higher value of *k*, gives a higher utilization bound at the expense of more preemptions. By selecting $k = m$ we obtain that the utilization bound is 100%.

Pfair scheduling algorithms [4, 5] and the algorithm BF [10] have a utilization bound of 100%. Unfortunately neither of them have proven bounds on the number of preemptions per job. Several algorithms were presented by Khemka and

Shyamasundar for multiprocessor scheduling with a high utilization bound and few preemptions [11]. They offered a bound on the number of preemptions as a function of the greatest common divisor of the periods of all tasks in the task set. Unfortunately, for some task sets the bound on the number of preemptions divided by the number of jobs approaches infinity. This happens for task sets with periods selected as $T_i = (q + i)$:th prime number and where q approaches infinity. As already pointed out, with our algorithm this cannot happen.

References

[1] J. Leung and J. Whitehead, "On the Complexity of Fixed-priority Scheduling of Periodic Real-Time Tasks," *Performance Evaluation, Elsevier Science*, vol. 22, pp. 237-250, 1982.

[2] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, pp. 127-140, 1978.

[3] D.-I. Oh and T. P. Baker, "Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment," *Real Time Systems Journal*, vol. 15, pp. 183-192, 1998.

[4] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate Progress: A Notion of Fairness in Resource Allocation," *Algorithmica*, vol. 15, pp. 600-625, 1996.

[5] J. Anderson and A. Srinivasan, "Mixed Pfair/ERfair Scheduling of Asynchronous Periodic Tasks," *Journal of Computer and System Sciences*, vol. 68, pp. 157-204, 2004.

[6] T. P. Baker, "Comparison of Empirical Success Rates of Global vs. Partitioned Fixed-Priority EDF Scheduling for Hard Real Time," Department of Computer Science, Florida State University, Tallahassee, FL 32306 July 2005.

[7] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM (JACM)*, vol. 20, pp. 46-61, 1973.

[8] B. Andersson and E. Tovar, "Multiprocessor Scheduling with Few Preemptions," IPP Hurray Research Group, Polytechnic Institute of Porto, Portugal HURRAY-TR-060811, Available at http://www.hurray.isep.ipp.pt/privfiles/tr-hurray-060811.pdf, August 2006.

[9] S. K. Baruah, R. Howell, and L. Rosier, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Systems*, pp. 301-324, 1990.

[10] D. Zhu, D. Mossé, and R. Melhem, "Multiple-Resource Periodic Scheduling Problem: how much fairness is necessary?," presented at 24th IEEE International Real-Time Systems Symposium, 2003.

[11] A. Khemka and R. K. Shyamasundar, "Multiprocessor Scheduling of Periodic Tasks in a Hard Real-Time Environment," presented at International Parallel Processing Symposium, 1992.

Appendix A: Algebraic rewriting

Consider the following inequality.

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > \frac{1}{m}\times\left(L\times SEP+\left\lfloor\frac{m-L-1}{k}\right\rfloor\times(1-SEP)+ m-L-1-\left\lfloor\frac{m-L-1}{k}\right\rfloor+1\right)$$

Simplifying yields:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > \frac{1}{m}\times\left(L\times SEP-\left\lfloor\frac{m-L-1}{k}\right\rfloor\times SEP+m-L\right)$$

Consider two cases:

<u>Case 1.</u> *k*<*m*

Observing that $\lfloor x\rfloor \le x$ yields:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > \frac{1}{m}\times\left(L\times SEP-\left(\frac{m-L-1}{k}\right)\times SEP+m-L\right)$$

Using (1) gives us that the RHS is unchanged if *L* changes. Hence, selecting *L*=0 does not change the RHS. This gives us:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > 1-\left(\frac{m-1}{k}\right)\times\frac{1}{m}\times SEP$$

Using (1) again yields:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > 1-\frac{m-1}{m\times(k+1)}$$

Simplifying yields:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > \frac{m\times k+1}{m\times(k+1)} > \frac{m\times k}{m\times(k+1)} = \frac{k}{(k+1)} = SEP$$

<u>Case 2.</u> *k*=*m*.

Observing that $\lfloor(m-L-1)/k\rfloor=0$ yields:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > \frac{1}{m}\times(L\times SEP+m-L)$$

and observing that according to (1) with *k*=*m*, we have *SEP*=1. This gives us:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > 1$$

Hence regardless of whether Case 1 or Case 2 is true, we obtain that:

$$\frac{1}{m}\sum_{i=1}^{n}\frac{C_i}{T_i} > SEP$$

Appendix B: Proof of a lemma

Lemma 10. If a task is split let us consider that as one task in this lemma and if the job from such a task is split then this job is only considered as one job.

Consider the case when all tasks arrive at time 0 and they are scheduled using `EKG`. Consider the processors in one of the groups consisting of *k* processors. We claim that the number of preemptions during [0, *lcm*) divided by the number of jobs that executed during [0, *lcm*) is at most 2*k*.

Proof: The t_j denote the *j*:th time that a task arrives on the group of processors during the time interval [0,lcm). As an illustration of t_j we have: t_1=0 and t_2=min{T_i}.

We will first (in 1. below) prove an upper bound on the number of preemptions during [t_1,t_j) and then (in 2. below) prove a lower bound on the number of jobs that arrive during [t_1,t_j).

1. We claim (for 2≤*j*) it holds that:

$$\begin{aligned}&\text{The number of preemptions during } [t_1, t_j) \\ &\quad \le (j-1) * 2 * k. \end{aligned} \tag{23}$$

We will prove it using induction.

Base case. We claim that (23) is true for *j*=2.

Proof of the case case. Consider [t_1,t_2) All tasks arrive at time t_1 and it may cause context switches. But since the tasks have not executed before t_1, it is not a preemption. Algorithm 2 computes two times `timea` and `timeb` where preemptions may occur. It may cause preemptions on all *k* processors in the group. If a non-splitted task finishes execution during (`timea`,`timeb`) then there will be a context switch but this is not a preemption. Hence we obtain that during [t_1,t_2) there is at most 2**k* preemptions in the group during [t_1,t_2).

The induction step. We claim that if (23) is true for *j*=*q* where 2≤*q* then (23) is true for *j*=*q*+1. **Proof of the induction step.** We know that the number of preemptions during [t_1,t_q) is at most (*q*-1)*2**k*. Let us now consider the time interval [t_q,t_{q+1}). Consider [t_q,t_{q+1}). According to Algorithm 2, there are two instants when preemptions can occur `timea` and `timeb`. One might think that a preemption can also occur at t_q if t_q≠`timea`. This is not correct though. To see this, consider the case where t_q≠`timea`. Then it follows that t_q<`timea`. Let us divide this into two further cases.

<u>Case 1.</u> `Firsttask[p]` arrived at t_q.

There may be a context switch at t_q but since the job of `firsttask[p]` has not executed before this context switch, it is not a preemption.

Case 2. `Firsttask[p]` did not arrive at t_q.

Then it holds that `firsttask[p]` executed before just before t_q because of mirroring and hence it is not preempted at time t_q.

We conclude that if $t_q \neq$ `timea[p]` then there is no preemption at t_q.on processor p.

Since there are k processors in a group, there can be at most $2k$ preemptions in a group during $[t_q,t_{q+1})$. Adding those preemptions to the preemptions during $[t_1,t_q)$ we obtain that the number of preemptions during $[t_1,t_{q+1})$ is at most $(q-1)*2*k+2k=((q+1)-1)*2*k$. This is the statement of the induction step, so the induction step is proven.

Using the base case, the induction step and performing induction on j we obtain (23) is true.

2. We claim (for $2 \leq j$) it holds that:

$$\text{The number of jobs that arrive during } [t_1, t_j) \geq j-1 \qquad (24)$$

The claim (24) is true because every time interval $[t_j,t_j+1)$ starts and ends with the arrival of at least one job.

Combining (23) and (24) yields:

$$\frac{\text{The number of preemptions during } [t_1, t_j)}{\text{The number of jobs that arrive during } [t_1, t_j)} \leq 2 \times k \qquad (25)$$

We know that a task arrives at time lcm. Hence there is a j such that $t_j=lcm$. By applying $t_1=0$ and $t_j=lcm$ on (25) we obtain the statement of the lemma. □