Preemptive Scheduling of Uniform Processor Systems

TEOFILO GONZALEZ

The Pennsylvania State University, University Park, Pennsylvania

AND

SARTAJ SAHNI

University of Minnesota, Minneapolis, Minnesota

ABSTRACT. An $O(n)$ time algorithm is presented to obtain an optimal finish time preemptive schedule for n independent tasks on m uniform processors. This algorithm assumes that the tasks are initially ordered by task length and that the uniform processors are ordered by processor speed.

KEY WORDS AND PHRASES: uniform processors, preemptive schedules, optimal finish time schedules, independent tasks

CR CATEGORIES: 4.32, 5.39

1. *Introduction*

Let $P_1, P_2, \ldots, P_m$ be a system of m uniform processors and let the speed of P_i be s_i. Without loss of generality we may assume $s_1 \geq s_2 \geq \cdots \geq s_m$. n independent tasks are to be scheduled on these m processors. Let t_i be the length of (or processing time required by) task i. We assume that $t_1 \geq t_2 \geq \cdots \geq t_n$. The execution of task i on processor P_j requires t_i/s_j time, $1 \leq i \leq n$ and $1 \leq j \leq m$. The *finish time* of a schedule is the time at which all tasks have been executed. A preemptive schedule is a schedule in which one may suspend the execution of a task before its completion and resume its execution at a later time (possibly on a different processor). In a *nonpreemptive* schedule the processing of a task on a given processor cannot be suspended until its completion. An *optimal finish time* (OFT) preemptive (nonpreemptive) schedule is a preemptive (nonpreemptive) schedule with minimum finish time among all feasible preemptive (nonpreemptive) schedules for the given independent tasks and uniform processors. The reader is referred to [2] for a more precise definition of these terms.

The problem of obtaining OFT nonpreemptive schedules for a uniform processor system with $m \geq 2$ processors is known to be NP-complete (see Karp [7]). Liu and Liu [8] present worst-case bounds on a simple heuristic that obtains approximately optimal OFT schedules. Gonzalez, Ibarra, and Sahni [3] analyze the performance of an LPT (largest processing time first) type heuristic for obtaining near optimal OFT nonpreemptive schedules. Further approximation methods to obtain near optimal schedules for uniform processors are given by Horowitz and Sahni [5].

In this paper we are concerned solely with obtaining OFT preemptive schedules. For this problem Liu and Yang [9] have obtained the following bound on the length of an

This research was supported in part by the National Science Foundation under Grants DCR74-10081 and MCS76-21024.

Authors' addresses: T. Gonzalez, Department of Computer Science, The Pennsylvania State University, University Park, PA 16802; S. Sahni, Department of Computer Science, University of Minnesota, Minneapolis, MN 55455.

OFT schedule. Let w be the length of an OFT preemptive schedule, let $T_j = \sum_{1\le i\le j} t_i$ let $S_j = \sum_{1\le i\le j} s_i$, and let $n \ge m$; then

$$w \ge \max\left\{\max_{1\le j<m}\{T_j/S_j\},\ T_n/S_m\right\}\dots. \tag{1}$$

[9] also presents an algorithm that obtains an OFT schedule with this lower bound length for the case $s_i = 1,\ 1 < i \le m$ and $s_1 \ge 1$. Horvath, Lam, and Sethi [6] adapt the "critical path" algorithm of Muntz and Coffman [11] to the uniform processor case and obtain an $O(mn^2)$ time algorithm to obtain OFT preemptive schedules. Their algorithm shows that (1) is in fact always an equality, i.e. the length w of an OFT schedule always satisfies the equality:

$$w = \max\left\{\max_{1\le j<m}\{T_j/S_j\},\ T_n/S_m\right\}\dots. \tag{2}$$

The algorithm of [6] generates schedules with an excessive number of preemptions. For n tasks and m processors, the schedules generated by this algorithm may have as many as $n^2(m-1)$ preemptions.

In this paper we first show that for any set of n independent tasks and any uniform processor system with $m \ge 1$ processors, there exists an OFT schedule with at most $2(m-1)$ preemptions. Hence, there exists a large gap between the worst-case number of preemptions using the algorithm of Horvath et al. and the maximum number of preemptions needed. Our proof that $2(m-1)$ is the maximum number of preemptions needed is constructive and it leads to an algorithm with this property. The time complexity of the resulting algorithm is $O(n)$. The complexity analysis does not include the time to sort the tasks and processors into order by task length and processor speed, respectively. If this is to be done then the complexity becomes $O(n + m \log n)$ since our algorithm requires only the largest m tasks to be sorted (note that heap sort can be used to obtain the m largest of a set of n numbers in $O(m \log n)$ time). This time can be further reduced to $O(n + m \log m)$ since the mth largest task can be found in time $O(n)$ [1]. Having found this task, the m largest tasks can be found in time $O(n)$ and then sorted in time $O(m \log m)$. For the case of identical processors, our algorithm reduces to that of McNaughton [10] and so generates OFT schedules with no more than $(m-1)$ preemptions.

2. *A Bound for the Maximum Number of Preemptions*

In this section we show that every uniform processor system can be optimally scheduled using at most $2(m-1)$ preemptions. The proof of this is constructive. We exhibit an algorithm that generates OFT schedules and show that the resulting schedules have at most $2(m-1)$ preemptions. Our scheduling algorithm uses four scheduling rules, R1–R4. It begins by scheduling tasks one at a time in order of nonincreasing task lengths. Rules R1–R3 are used at this stage. The first task considered is one with largest length. This continues until the next task to be scheduled satisfies one of two conditions (A1) or (A2) (to be specified later). At this time, all unscheduled tasks are scheduled using rule R4. The four rules are specified in a semiformal manner. The mathematical details of the rules are relegated to Section 3, where the algorithm is presented formally. This allows us to concentrate on the important features of the scheduling rules that result in OFT schedules with at most $2(m-1)$ preemptions.

Throughout this section and Section 3, we assume $n \ge m$. In case $n < m$ then only the fastest n processors need be considered and the remaining discarded. Before presenting the rules, we develop some notation and state the conditions (A1) and (A2) referred to above. $I = \{P_{i_1}, P_{i_2}, \dots, P_{i_k}\}$ will represent a subset of the m processors. We shall denote by S_I the quantity $\sum_{P_{i}\in I} s_i$. It will be assumed that the n tasks are ordered such that $t_1 \ge t_2 \ge \cdots \ge t_n$. T_k will represent the sum $\sum_{i=1}^{k} t_i$. w will be the length of an OFT schedule for the given n tasks and m processors. w is given by (2). By *idle time* on processor P_j we shall mean time in the interval $[0, w]$ when processor P_j is not processing any task. Thus, idle time is relative to a given partial schedule. We shall denote by $np(i_1, i_2, \ldots, i_j)$ the number of preemptions in the processors $P_{i_1}, P_{i_2}, \ldots, P_{i_j}$. A *disjoint processor system* (DPS) is a set of $r \geq 1$ processors $P_{i_1}, P_{i_2}, \ldots, P_{i_r}$ such that $i_1 \leq i_2 \leq \cdots \leq i_r$ and processor P_{i_j} is idle exactly from u_{j-1} to u_j, $1 \leq j \leq r$, for some $u_0, u_1, \ldots, u_r$ with the property $u_0 = 0$, $u_r = w$, and $u_{j-1} < u_j$, $1 \leq j \leq r$. Note that the $np(i_1, \ldots, i_j)$ and a DPS are defined relative to a given (partial) schedule.

For a given subset I of processors and a given partial schedule, J will represent the subset of I consisting of those processors in I that have some idle time. k will be the index of the next task to be scheduled by rules R1–R3. As we shall see, when task k is to be scheduled, the cardinality of I will also be k. q_k will denote the cardinality of J at this time. The scheduling algorithm will begin with $I = \{P_1\}$ and $k = 1$. At this time no tasks have been scheduled and so $J = \{P_1\}$ and J is a DPS. We shall see that the application of rules R1–R3 will always leave J a DPS. Rules R1–R3 are applied until the next task k to be scheduled is such that either

(A1) $k = m$

or

(A2) $T_k/S_l < w$ and $t_k \leq sw$ where s is the speed of the slowest processor that is not in I.

During the presentation of the rules R1–R3 we shall show that following the scheduling of task k, the following hold:

(B1) the subset J of I is a DPS,

(B2) $np(i_1, i_2, \ldots, i_{k+1}) + q_{k+1} - 1 \leq 2k$ where $I = \{P_{i_1}, P_{i_2}, \ldots, P_{i_{k+1}}\}$.

The proof of this is by induction on k. Clearkly, when $k = 0$, no tasks have been scheduled and $I = \{P_1\}$. At this time J is a DPS, $q_1 = 1$ and both (B1) and (B2) hold.

In describing the rules we shall assume that I is partitioned into two disjoint sets $I1 = \{P_1, P_2, \ldots, P_j\}$ and $I2 = \{P_{l_1}, P_{l_2}, \ldots, P_{l_{k-j}}\}$ such that $j + 1 \neq l_1$ and $l_1 < l_2 < \cdots < l_{k-j}$. It is easy to see that this can always be done for any arbitrary set I of processors. Depending on the relation between T_k/S_l and w, one of rules R1–R3 will be used to schedule the next task k.

*Rule R*1. Condition: $T_k/S_l = w$.

Since the processors in I with idle time form a DPS, task k can be scheduled to use up all the idle time in I. Let P_j be the fastest processor such that $P_j \notin I$. Replace I by $I \cup \{P_j\}$. Note that now $J = \{P_j\}$ is a DPS.

Clearly, the number of preemptions increases by $q_k - 1$ and $q_{k+1} = 1$. Hence we have:

$$np(i_1, \ldots, i_{k+1}) + q_{k+1} - 1 = np(i_1, \ldots, i_k) + q_k - 1 \leq 2(k - 1) < 2k. \qquad \square$$

*Rule R*2. Condition: $T_k/S_l > w$.

Let $I = I1 \cup I2$, $I1 = \{P_1, \ldots, P_j\}$ and $I2 = \{P_{l_1}, \ldots, P_{l_{k-j}}\}$. By definition of $I1$ and $I2$, $P_{j+1} \notin I$. Also, $I2 \neq \emptyset$ as otherwise $j = k$ and $T_k/S_k \leq w$. Rule R3 will be the only rule that adds processors to $I2$. Since this addition is done by selecting the smallest indexed processor r such that $t' > s_r w$ where $t' \geq t_k$ is the length of the task being scheduled and since $P_{j+1} \notin I$, it follows that $t_k \leq s_{j+1} w$. Furthermore, this implies that the DPS in I must include at least one of the processors in $I2$ as otherwise $T_k/S_l \leq w$.

Having made these observations we readily see that task k may be scheduled as shown in Figure 1. This figure shows only the DPS $J = \{P_{a_1}, \ldots, P_{a_b}\}$ of I and the processor P_{j+1}. The unshaded regions represent idle time in the interval $[0, w]$ before task k is scheduled. In scheduling this task, all the idle time in $I1$ is used up. All this idle time can be used since $T_k/S_l > w$. Enough idle time from I_2 is used so that the remaining processors with idle time form a DPS. The observation $t_k \leq s_{j+1}w$ guarantees that task k may be so scheduled. If the processing of task k is assigned to P_{j+1} and to k' of the processors in J then the number of preemptions increases by k' and the DPS left

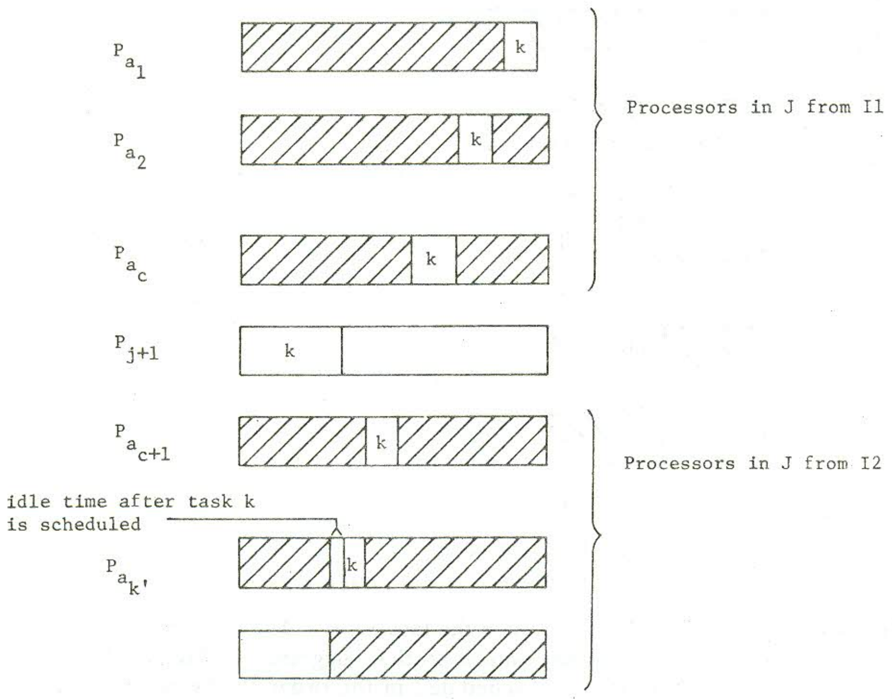

FIG. 1. Scheduling task k under rule R2

behind consists of at most $q_k - k' + 2$ processors. Hence, replacing I by $I \cup \{P_{j+1}\}$ we have:

$$np(i_1, \dots, i_{k+1}) + q_{k+1} - 1 \le np(i_1, \dots, i_k) + k' + q_k - k' + 2 - 1 \le 2(k-1) + 2 = 2k.$$

One may readily verify that the new J satisfies B1. The exact distribution of task k over the processors will be specified in Section 3. At this time it is sufficient to realize that the distribution can be done in the manner described above. □

*Rule R*3. Condition: $T_k/S_l < w$.

Let P_j be the highest indexed processor such that $P_j \notin I$ and $t_k > s_j w$. One may show that j is larger than the largest index of a processor in I. To see this, let P_r be the largest indexed processor in I. If P_r was included in an application of either R1 or R2 or if $r = 1$ then clearly $P_i \in I$ for all $i \le r$. Hence j must be greater than r. If P_r is included during an application of R3 then let k' be the value of k when P_r is included. From R3 it follows that $t_{k'} \le s_i w$ for all $i < r$ and $P_i \notin I$. Hence, if $t_k > s_j w$, $j > r$. Also, note that since (A2) does not hold for task k, P_j must exist. Task k is assigned to P_j and to some of the processors $J = \{P_{a_1}, \dots\}$ in I that form a DPS. Figure 2 shows the processor P_j and the set J. The assignment is carried out in such a way as to leave behind a DPS. Clearly, the conditions under which this rule is applied guarantee that this may be done. The details are left to the algorithm. Unshaded areas in the figure represent idle time before task k is scheduled.

I is replaced by $I \cup \{P_j\}$. The analysis showing that $np\{i_1, \dots, i_{k+1}\} + q_{k+1} - 1 \le 2k$ is identical to that for rule R2. One may readily verify that the new I has all the desired properties. □

*Rule R*4. Condition $k = m$ or ($t_k \le s_j w$ for all P_j not in I and $T_k/S_l \le w$).

First, let us assume $k \ne m$. Let $P_{r_1}, \dots, P_{r_l}$ be the processors not in I, ordered so that $r_1 \ge r_2 \ge \cdots \ge r_l$. Let i be the smallest index such that $\sum_{k \le j \le i} t_j \ge w \sum_{1 \le j \le l} s_{r_j}$. If there is no such i, let $i = n$.

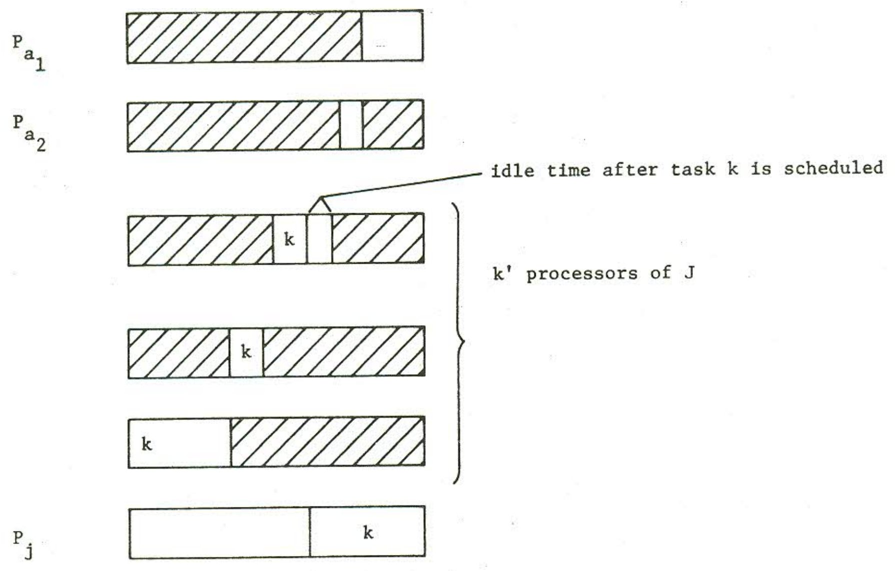

FIG. 2. Scheduling task k under rule R3

(i) When $\sum_{k\le j\le i} t_j \le w \sum_{1\le j\le l} s_{r_j}$ then the tasks $k, k+1, \ldots, i$ may be scheduled on $P_{r_1}, P_{r_2}, \ldots, P_{r_l}$ using a procedure similar to that suggested by McNaughton [10] for identical processors. Processors are scheduled in the order P_{r_1} before P_{r_2} before P_{r_3}, etc. Each processor is scheduled right to left (Figure 3). Since for any task k' with $k \le k'$ i and processor P_j with $r_1 \le r_j \le r_l$ we have $t_{k'} \le ws_{r_j}$, it follows that if k' is preempted, it can be completed on the next processor without any overlap in processing. The scheduling of tasks $k, k+1, \ldots, i$ in this way introduces at most $l-1$ preemptions. If $i \ne n$ then $\sum_{k\le j\le i} t_j = w \sum_{1\le j\le l} s_{r_j}$ and by definition of w, the remaining tasks $i+1, \ldots, n$ may be trivially scheduled on the q_k processors forming the DPS of I. This is similar to Figure 4 and introduces at most another $q_k - 1$ preemptions. Hence, in this case, at most $q_k + l - 2$ additional preemptions are needed. Since the cardinality of I is k, $l = m - k$. For the partial schedule created by rules R1–R3 we have from (B2):

$$np(i_1, i_2, \ldots, i_k) + q_k - 1 \le 2(k-1).$$

Hence, for the complete schedule resulting from the application of rule 4 we have:

$$np(1, 2, \ldots, m) \le 2(k-1) - q_k + 1 + q_k + m - k - 2 = m + k - 1 \le 2(m-1).$$

(ii) If $\sum_{k\le j\le i} t_j > w \sum s_{r_j}$ then all but the ith task may be scheduled on $P_{r_1}, P_{r_2} \ldots, P_{r_l}$ as described in Figure 3. A fraction of task i to fit the remaining space on P_{r_l} is scheduled on P_{r_l}. The rest of task i is scheduled on the DPS of I beginning with the fastest processor in I (Figure 4). In this figure, the processors in I are drawn in nondecreasing order of speed. To see that task i can be so scheduled without overlapping its processing on P_{r_l}, let $P_{k'}$ be the last (slowest) processor in the DPS of I on which task i also gets scheduled. If $s_{k'} \le s_{r_l}$ then this follows from the fact that $T_k/S_l \le w$ and $t_i \le t_k$. When $s_{k'} > s_{r_l}$, the nonoverlap is ensured by the condition $t_i \le t_k \le s_{r_l} w$. The remaining tasks $i+1, \ldots, n$ may now be trivially scheduled in the idle time remaining on the processors that formed the DPS of I. This is similar to scheduling tasks $i+1, \ldots, n$ in case (i). Once again, before the application of rule R4, we have:

$$np(i_1, i_2, \ldots, i_k) + q_k - 1 \le 2(k-1).$$

The scheduling above introduces at most $l + k' - 1$ preemptions for the tasks $k, \ldots, i$ and another $q_k - k'$ for the remaining tasks. Hence, for the complete schedule we have:

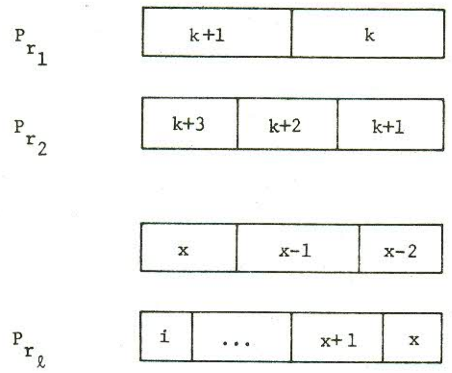

FIG. 3. Scheduling on $P_{r_1}, \dots, P_{r_l}$

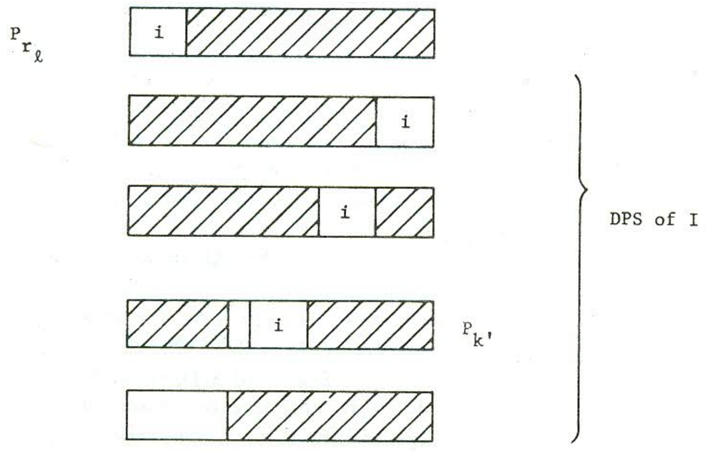

FIG. 4. Completing task i in the DPS of I

$$np(1, 2, \dots, m) \le 2(k-1) - q_k + 1 + q_k + m - k - 1 = m + k - 2 < 2(m-1).$$

(iii) When $k = m$, the tasks $k, \dots, n$ have to be scheduled on the DPS of I. From the definition of w, there is enough space here for this. The scheduling is similar to the DPS scheduling of (i). At most $q_k - 1$ preemptions are introduced. Hence, we have

$$np(1, 2, \dots, m) \le 2(m-1) - q_k + 1 + q_k - 1 = 2(m-1). \qquad \square$$

The analysis of the above rules leads to the following theorem:

THEOREM. *For every system of m uniform processors and n tasks there exists an OFT schedule with at most $2(m-1)$ preemptions.*

It is interesting to note that if all processors are identical then condition (A2) holds for $k = 1$ and the theorem calls only for the application of R4. R4 in this case behaves exactly as McNaughton's rule [10] and so at most $m - 1$ preemptions are introduced. The following example shows that for every m, there exists a system of m uniform processors and m tasks such that every OFT schedule requires at least $2(m-1)$ preemptions. Hence, the bound of the theorem is tight.

Example. Let $k > 1$, $s_i = k$, $1 \le i < m$, and $s_m = 1$. Let $t_i = k - (k-1)/m$, $1 \le i \le m$.

For this system we have $T_i/S_i < 1$, $1 \le i < m$, and $T_m/S_m = 1$. Hence, $w = 1$ and in every OFT schedule, none of the m processors is idle in the interval $[0, 1]$. As a result of this, all m tasks must finish at the same time. If this is to happen then each of the m tasks must spend some time on P_m. Let the sequence of task executions of P_m in an OFT schedule with minimum preemptions be $\alpha_1, \alpha_2, \dots, \alpha_p$ where $p \ge m$. Clearly, for tasks α_1 and α_p there is at least one preemption and for the remaining tasks at least two preemptions. Hence, the total number of preemptions is at least $2(m-1)$. □

3. *The Algorithm*

Our algorithm to obtain an OFT schedule with at most $2(m-1)$ preemptions is simply a restatement of rules R1–R4. It is presented as a series of five procedures: SCH, RULE_R1, RULE_R2, RULE_R3, and RULE_R4. SCH schedules n tasks on m uniform processors. We may assume $n \geq m$. SCH makes use of RULE_R1, RULE_R2, RULE_R3, and RULE_R4. Since these four procedures are invoked only from one point in SCH, any implementation of these five procedures would probably combine all five procedures into one.

Before presenting the procedures, we describe the significance of various variable names and also the data structures used. Procedure SCH obtains an OFT schedule for n tasks on m processors. The length of this schedule is at most w. The task times are $T(1) \geq T(2) \geq \cdots \geq T(n)$ and the processor speeds are $S(1) \geq S(2) \geq \cdots \geq S(m)$. Using the notation of Section 2, we represent the set $I = I1 \cup I2$ using different representations for $I1$ and $I2$. $I1$ is simply represented by the integer j which is the highest index of a processor in $I1$. $I2$ is represented as a singly linked list where each node has exactly one field: $LINK$. f points to the first node in this list and e to the last node. s is the index of the slowest processor not in I. When the processor with index m has been included in I then either the next task to be scheduled has index m or for the next task k we have $T(k) < wS(s)$. In either case application of rules R1–R3 terminates. Hence it is sufficient to just set $S(m+1) = M$ for some number M for which $T(k) < wM$. This is done in line 4.

The DPS of I is represented as a sequential list in the one-dimensional array D with p and q being pointers to the first and last elements in the DPS, respectively. The DPS is made up of processors $D(p)$, $D(p+1)$, ... , $D(q)$ and $S(D(p)) \geq S(D(p+1)) \geq \cdots \geq S(D(q))$. Since the DPS will always have at least one processor, it isn't necessary to explicitly check for an empty DPS. Another one-dimensional array F is used to indicate the end of the idle interval on a processor in the DPS. Thus processor $D(q)$ is idle from 0 to $F(q)$ and if $p \neq q$ then processor $D(i)$ is idle from $F(i+1)$ to $F(i)$, $p \leq i < q$.

To aid in the implementation of rule R3 the variable v is used. At any time v is the largest index such that $P_v \in I$.

For each task being scheduled, the output includes a list of processors together with the time for which the task is to be processed on each processor.

The procedures for the four rules are written with no parameters as they will actually be written into SCH. Hence, all variable names have the same meaning as in SCH. w is as given by (2). The procedures are written in SPARKS. The reader unfamiliar with some of the constructs used is referred to [4] for clarification.

```
     procedure SCH(n, m, T, S, w)
       //obtain an OFT schedule with length less than or equal to w//
 1     declare T(n), S(m + 1) LINK(m), D(m), F(m)
 2     j ← 1; f ← 0 //initialize I1 and I2//
 3     D(p) ← p ← q ← 1; F(p) ← w //initialize DPS//
 4     S(m + 1) ← T(1)/w + 1; s ← m
 5     T_I ← T(1); S_I ← S(1); k ← 1
       //use rules R1–R3//
 6     while k < m and (T_I/S_I ≥ w or T(k) > w * S(s)) do
 7       print("Schedule for task", k)
 8       case
 9         :T_I/S_I = w:call RULE_R1
10         :T_I/S_I > w:call RULE_R2
11         :T_I/S_I < w:call RULE_R3
12       end
13       k ← k + 1; T_I ← T_I + T(k)
```

```
14      end
15      call RULE_R4
16  end SCH

    procedure RULE_R1
      //schedule task $k$ when $T_l/S_l = w$//
1     $T1 \leftarrow 0$
2     for $i \leftarrow q$ to $p$ by $-1$ do //use up DPS//
3       print("Processor", $D(i)$, "From", $T1$, "To", $F(i)$)
4       $T1 \leftarrow F(i)$
5     end
6     $j \leftarrow j + 1$; $p \leftarrow q \leftarrow 1$; $S_l \leftarrow S_l + S(j)$ //$I \leftarrow I \cup \{P_{j+1}\}$//
7     $D(p) \leftarrow j$; $F(p) \leftarrow w$ //update DPS//
8     while $j + 1 = f$ do //update $I1$ and $I2$//
9       $j \leftarrow j + 1$
10      $f \leftarrow LINK(f)$
11    end
12  end RULE_R1

    procedure RULE_R2
      //schedule task $k$ when $T_l/S_l > w$//
1     $T1 \leftarrow 0$
2     while $D(p) \le j$ do //use up processors from $I_1$//
3       print("Processor", $D(p)$, "From", $F(p + 1)$, "To", $F(p)$)
4       $T1 \leftarrow T1 + (F(p) - F(p + 1)) * S(D(p))$
5       $p \leftarrow p + 1$
6     end
7     $F(q + 1) \leftarrow 0$
      //use some processors from $I2$//
8     while $T(k) - T1 - (F(p) - F(p + 1)) * S(D(p)) > S(j + 1) * F(p + 1)$ do
9       print("Processor", $D(p)$, "From", $F(p + 1)$, "To", $F(p)$)
10      $T1 \leftarrow T1 + (F(p) - F(p + 1)) * S(D(p))$
11      $p \leftarrow p + 1$
12    end
13    $T2 \leftarrow (T(k) - T1 - F(p) * S(D(p)))/(S(j + 1) - S(D(p)))$
14    print("Processor", $j + 1$, "From", "0", "To", $T2$)
15    print("Processor", $D(p)$, "From", $T2$, "To", $F(p)$)
16    if $T2 \ne F(p + 1)$ then [$F(p) \leftarrow T2$; $p \leftarrow p - 1$]
17    $j \leftarrow j + 1$; $D(p) \leftarrow j$; $F(p) \leftarrow w$; $S_l \leftarrow S_l + S(j)$ //update DPS and $I$//
18    while $j + 1 = f$ do //update $I1$ and $I2$//
19      $j \leftarrow j + 1$
20      $f \leftarrow LINK(f)$
21    end
22  end RULE_R2
```

The algorithm for rule R2 begins by scheduling the task k preemptively on the processors of $I1$. This is done in lines 2–6. From the discussion of this rule in Section 2, we know that $I2$ is not empty and that not all of task k can be scheduled in $I1$. $T1$ denotes the amount of processing that has been thus far assigned to processors. Lines 8–12 use up all idle time on processors in $I2$ until we reach a processor $P_{D(p)}$ that permits the construction of the new DPS. The condition of line 8 correctly determines this processor. Line 13 allocates the remaining processing to P_{j+1} and $P_{D(p)}$ in such a way that the processors of I with idle time form a DPS. Thus task k is scheduled on P_{j+1} from time 0 to time $T2$ and on $P_{D(p)}$ from $T2$ to $F(p)$. $T2$ must therefore satisfy the following equality:

$$T1 + T2 * S(j + 1) + (F(p) - T2) * S(D(p)) = T(k),$$

from which we get

$$T2 = (T(k) - T1 - F(p) * S(D(p)))/(S(j + 1) - S(D(p))).$$

Lines 16–21 simply update the DPS, $I1$, and $I2$. Note that the values of v and s remain unchanged.

In the algorithm for rule R3 lines 1–3 determine the next processor to include in I. Lines 4–8 update $I1$ or $I2$. From the definition of v it follows that if $I1$ is being updated then $I2$ must be empty. In line 9 s is updated to $m + 1$ in case the slowest processor P_m has been included. It is now the case that for the remaining tasks the task time is less than the processor time available on any processor not in I. $T1$ represents the real time up to which the schedule for task k has been built and $T2$ represents the amount of $T(k)$ that has been scheduled up to time $T1$. $T3$ is the processing capability left on the next processor $P_{D(q)}$ to be considered. If the entire capability $T3$ is used then to schedule the remainder of task k on P_v we will have to schedule task k on P_v from $T4$ to w. The three cases of lines 14–25 appropriately schedule task k. In case $T4 \le F(q)$ then it is necessary to use the entire interval from $T1$ to $F(q)$ on $P_{D(q)}$. If $T4 = F(q)$ then the processing of task k can be completed on P from $T4$ to w. This will leave behind a DPS (Figure 2). If $T4 < F(q)$ then more of task k has to be scheduled on the DPS of I. When $T4 > F(q)$ then not all of the idle time on $P_{D(q)}$ is to be used. In this case we determine a time $T5$ such that if k is processed on $P_{D(q)}$ from $T1$ to $T5$ then it can be completed on P_v from $T5$ to w. This would leave behind a DPS in I. $T5$ must be such that the processing on $P_{D(q)}$ (i.e. $(T5 - T1) * S(D(q))$) plus the processing on P_v (i.e. $(w - T5) * S(v)$) equals the remaining processing requirement of task k (i.e. $T(k) - T2$). Solving for $T5$ we get

$$T5 = (T(k) - T2 + T1 * S(D(q)) - wS(v))/(S(D(q)) - S(v)).$$

```
    procedure RULE_R3
      //schedule task $k$ when $T_l/S_l < w$ and $T(k) > sw$//
 1    repeat //find processor to include in $I$//
 2      $v \leftarrow v + 1$
 3    until $T(k) > S(v) * w$
 4    if $v = j + 1$ then [$j \leftarrow v$] //update $I1$//
 5              else [//update $I2$//
 6                   $LINK(v) \leftarrow 0$
 7                   if $f = 0$ then [$f \leftarrow e \leftarrow v$]
 8                              else [$LINK(e) \leftarrow v; e \leftarrow v$]]
 9    if $v = m$ then [$s \leftarrow m + 1$]; $S_l \leftarrow S_l + S(v)$
      //schedule task $k$//
10    $T1 \leftarrow T2 \leftarrow 0$
11    repeat
12      $T3 \leftarrow (F(q) - T1) * S(D(q))$
13      $T4 \leftarrow w - (T(k) - T2 - T3)/S(v)$
14      case
15        :$T4 < F(q)$:print("Processor", $D(q)$, "From", $T1$, "To", $F(q)$)
16                  $q \leftarrow q - 1$; $T2 \leftarrow T2 + T3$; $T1 \leftarrow F(q)$
17        :$T4 = F(q)$:print("Processor", $D(q)$, "From", $T1$, "To", $F(q)$)
18                  print("Processor", $v$, "From", $F(q)$, "To", $w$)
19                  $D(q) \leftarrow v$; return
20        :$T4 > F(q)$:$T5 \leftarrow (T(k) - T2 + T1 * S(D(q)) - w * S(v))/(S(D(q)) - S(v))$
21                  print("Processor", $D(q)$, "From", $T1$, "To", $T5$)
22                  print("Processor", $v$, "From", $T5$, "To", $w$)
23                  $q \leftarrow q + 1$; $D(q) \leftarrow v$; $F(q) \leftarrow T5$
24                  return
25      end
26    forever
27    end RULE_R3
```

The last procedure, RULE_R4, is straightforward and so is not presented.

ANALYSIS OF THE SCHEDULING ALGORITHM. First, let us analyze RULE_R1 through RULE_R3. Since at most m processors can enter the DPS, the total time for DPS additions and deletions is $O(m)$. The same reasoning leads us to conclude that the total update time for $I1$ and $I2$ (lines 8–11 of RULE_R1 and lines 18–21 of RULE_R2) is also $O(m)$. The remaining portions of these three procedures takes $O(1)$ for each call of the procedure. Since the total number of calls is at most m, the total time spent in these three procedures is $O(m)$.

The procedure RULE_R4 may be easily implemented to work in $O(n)$ time. Hence, the time complexity of SCH is $O(n + m)$. Since we may assume $n \geq m$, we have $O(n)$ as the complexity of SCH. □

REFERENCES

1. Blum, M., Floyd, R.W., Pratt, V.R., Rivest, R.L., and Tarjan, R.E. Time bounds for selection. *J Comptr. Syst. Sci. 7*, 4 (1972), 448–461.
2. Coffman, E.G. Jr. *Computer and Job Shop Scheduling Theory*. Wiley, New York, 1976.
3. Gonzalez, T., Ibarra, O.H., and Sahni, S. Bounds for LPT schedules on uniform processors. *SIAM J. Comptng. 5*, 1 (1977), 155–166.
4. Horowitz, E., and Sahni, S. *Fundamentals of Data Structures*. Computer Science Press, Woodland Hills, Calif., 1976.
5. Horowitz, E., and Sahni, S. Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM 23*, 2 (April 1976), 317–327.
6. Horvath, E.C., Lam, S., and Sethi, R. A level algorithm for preemptive scheduling. *J. ACM 24*, 1 (Jan. 1977), 32–43.
7. Karp, R.M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher, Eds., Plenum Press, New York, 1972, pp. 85–103.
8. Liu, J.W.S., and Liu, C.L. Bounds on scheduling algorithms for heterogeneous computing systems. Information Processing 74, North-Holland Pub. Co., Amsterdam, 1974, pp. 349–353.
9. Liu, J.W.S., and Yang, A. Optimal scheduling of independent tasks on heterogeneous computing systems. Proc. ACM Annual Conf., San Diego, Calif., Nov. 1974, pp. 38–45.
10. McNaughton, R. Scheduling with deadlines and loss functions. *Manage. Sci. 6* (1959), 1–12.
11. Muntz, R.R., and Coffman, E.G. Preemptive scheduling of real time tasks on multiprocessor systems. *J. ACM 17*, 2 (April 1970), 324–338.

RECEIVED MAY 1976; REVISED MARCH 1977