Scheduling Independent Tasks with Due Times on a Uniform Processor System

SARTAJ SAHNI AND YOOKUN CHO

University of Minnesota, Minneapolis, Minnesota

ABSTRACT. An algorithm to preemptively schedule n tasks on m uniform processors is presented. It is assumed that each task is available at time 0. Associated with each task is a due time by which it is to be completed. The algorithm schedules all tasks to complete by their due times whenever possible. The asymptotic time complexity of the algorithm is $O(n \log n + mn)$. It generates $O(mn)$ preemptions in the worst case. An example of n tasks requiring $O(mn)$ preemptions is also presented. The algorithm can also be used when all tasks have the same due times but different release times.

KEY WORDS AND PHRASES: independent tasks, preemptive schedule, due time, uniform processors, complexity

CR CATEGORIES: 5.25, 5.3, 5.4

1. *Introduction*

Let $P = \{P_1, P_2, \ldots, P_m\}$ be a set of m processors. Let t_i, r_i, and d_i, $1 \leq i \leq n$, be the task times, release times, and due times, respectively, of n independent tasks. Associated with each processor P_i is a speed s_i, $s_i > 0$. Processor P_i has an effective processing capability of s_i units of processing per time unit. Task j can be processed on P_i in t_j/s_i units. The processors are said to be *uniform*, as they operate at a constant speed independent of time. When $s_i = 1$, $1 \leq i \leq m$, the processors are said to be *identical*. In this paper we are concerned only with preemptive schedules. A schedule S for the n tasks is a *DD-schedule* iff the processing of each task commences no earlier than its release time and completes no later than its due time. A DD-schedule will also be referred to as a *feasible* schedule.

For the case of identical processors, Horn [3] presents an $O(n^3)$ algorithm to obtain a DD-schedule for any set of n independent tasks for which such a schedule exists. Sahni [5] presents a fast algorithm that obtains DD-schedules (whenever they exist) when all tasks have either the same release time or the same due time. For the case of two uniform processors, Bruno and Gonzalez [1] present an $O(n^3)$ algorithm that obtains a DD-schedule (whenever one exists). Gonzalez and Sahni [2] have developed an $O(n + m \log m)$ algorithm that can be used when all tasks have the same release time and also the same due time.

In this paper we study the case when all tasks have the same release time. Different tasks may, however, have different due times. Our algorithm to construct a DD-schedule (if one exists) for this case takes $O(mn)$ time in the worst case. DD-schedules containing at most mn preemptions are generated. While the algorithm is discussed in terms of a set of

This research was supported in part by the National Science Foundation under Grants MCS 76-21024 and MCS 78-15455.

Authors' present addresses: S. Sahni, Department of Computer Science, University of Minnesota, 136 Lind Hall, 207 Church St. SE, Minneapolis, MN 55455; Y. Cho, Computer Science Department, Seoul National University, Seoul, Korea.

tasks with the same release time, it should be noted that the algorithm may also be used when all tasks have the same due time but different release times. As pointed out in [3] and [5], the two situations are isomorphic. An instance of one may be transformed into an instance of the other by simply interchanging the roles of due times and release times.

2. *Preliminaries*

In this section we develop an algorithm that constructs a DD-schedule (if one exists) for n independent tasks all of which have the same release time and the same due time. The processor model assumed is, however, more general than the uniform processor model. In the next section this algorithm will be used to construct a DD-schedule (if one exists) for the one-release-time–many-due-time problem for a uniform processor system.

A *generalized processor* is a processor whose speed is a nondecreasing function of time. While the ideas presented in this section can be applied when the processor speed is given by any nondecreasing function of time, we limit ourselves here to the case where the speed changes only a finite number of times. Thus the characteristics of any generalized processor G in the time interval $[0, d]$ may be described by a finite list of pairs (σ_i, γ_i), $1 \le i \le p$. σ_i and γ_i, respectively, represent time and speed. G operates at speed γ_i in the interval $[\sigma_i, \sigma_{i+1}]$. We may assume that $\sigma_1 = 0$, $\sigma_{p+1} = d$, $\sigma_i < \sigma_{i+1}$, $\gamma_i \ge 0$, and $\gamma_i < \gamma_{i+1}$. The *effective processing capability*, L, of G in the interval $[0, d]$ is equal to $\sum_{i=1}^{p} (\sigma_{i+1} - \sigma_i)\gamma_i$.

A *generalized processor system* (GPS) is an ordered set of m generalized processors $\{G_1, G_2, \ldots, G_m\}$, $m \ge 1$. This set of generalized processors has the property that at each instance $t \in [0, d]$, the speed of G_i is no less than that of G_{i+1}, $1 \le i < m$.

It should be pointed out that a uniform processor system in which the processors have been ordered by speed is a special case of a GPS. The algorithm we shall develop here for a GPS is different from that developed in [2] for uniform processor systems. Theorem 1 gives a necessary and sufficient condition that every set of n tasks must satisfy if they are to be completed in the interval $[0, d]$ on a GPS. The remainder of this section is devoted to the proof of this theorem. Since the proof is constructive, it immediately leads to an algorithm for obtaining a DD-schedule for the case when all tasks have the same release time and the same due time. From now on we shall refer to a generalized processor simply as a processor. This should lead to no confusion.

THEOREM 1. *Let* $\{G_1, G_2, \ldots, G_m\}$ *be an m processor GPS. Let* t_i, $1 \le i \le n$, *be the task times of any set of n independent tasks. Assume* $t_i \ge t_{i+1}$, $1 \le i < n$, *and that* $n \ge m$. *Let* $T_i = \sum_1^i t_j$, $1 \le i < m$, *and* $T_m = \sum_1^n t_j$. *Let* L_i *be the effective processing capability of* G_i *in the interval* $[0, d]$, $1 \le i \le m$. *The n tasks can be scheduled to complete by time d iff* $\max_i\{T_i/\sum_1^i L_j\} \le 1$.

The statement of the theorem assumes that $n \ge m$. In case this is not true (i.e., $n < m$) we may discard $G_{n+1}, \ldots, G_m$ from the GPS as there is no advantage to scheduling a task on any of these processors. This follows from the following observations: (i) a task may be scheduled on at most one processor at any time, and (ii) at any instance $t \in [0, d]$, each of processors $G_1, \ldots, G_n$ is no slower than any of $G_{n+1}, \ldots, G_m$.

The "only if" part of the theorem is easily established. We need to show that if $\max\{T_i/\sum_1^i L_j\} > 1$, then the set of tasks cannot be completed in the interval $[0, d]$. If we consider only the largest i tasks, $i < m$, then for the same reasons as above we need consider only $G_1, \ldots, G_i$. If $T_i > \sum_1^i L_j$, then these i processors do not have enough effective processing capability in $[0, d]$ to complete these tasks. It should be easy to see that if this is the case, then these i tasks cannot be completed in $[0, d]$ when considered together with the remaining $n - i$ tasks and $m - i$ processors. Finally, if $T_m > \sum_1^m L_j$, then there is not enough effective processing capability in $[0, d]$ to perform all n tasks.

The proof for the "if" part is by construction. We show how to obtain a feasible schedule when $\max_i\{T_i/\sum_1^i L_j\} \le 1$. This construction requires the notion of a disjoint processor system (DPS). A processor G is said to be *idle in the interval* $[t_1, t_2]$ if it operates at nonzero speed in this interval and no task has been assigned it in this interval. Two processors G_1 and G_2 are said to be *disjoint in the interval* $[t_1, t_2]$ if they have disjoint idle times (i.e., no overlap in idle times) in this interval. A set D of processors is a *disjoint processor system in the interval* $[t_1, t_2]$ (abbreviated DPS$[t_1, t_2]$) iff the processors of D are pairwise disjoint in $[t_1, t_2]$. When t_1 and t_2 are clear from the context, we denote DPS$[t_1, t_2]$ simply by DPS. Note that our definition permits several processors with no idle time in the interval $[t_1, t_2]$ to be in a DPS$[t_1, t_2]$. Further, the union of the idle times on all processors in a DPS$[t_1, t_2]$ need not stretch from t_1 to t_2. Hence this definition is more general than the one used in [2]. For any DPS$[t_1, t_2]$, let R denote its remaining effective processing capability. R is the sum of the effective processing that can be carried out in the idle time of each processor in the DPS.

If $\max_i\{T_i/\sum_{j=1}^{i} L_j\} \le 1$, then the n independent tasks may be scheduled to complete in $[0, d]$ as follows. Initially, each generalized processor G_i is regarded as a separate DPS$[0, d]$, D_i. So we start with an ordered set $\{D_1, D_2, \ldots, D_m\}$ of DPSs where $D_i = \{G_i\}$. We use R_i to denote the remaining effective processing capability of D_i. Since no tasks have yet been scheduled, we have $R_i = L_i$, $1 \le i \le m$. Starting from this configuration, the n tasks will be scheduled one by one using one of the three scheduling rules R1–R3 (to be described soon). Each of these rules has associated with it a condition, and a rule may be used to schedule a task only if the task satisfies the condition associated with the rule. The condition for R1 is tested first, then the condition for R2, and finally that for R3. The first condition satisfied results in a rule application. Preceding and following the use of a rule, the following conditions will be true:

C1. We shall have a set $\{D_1, D_2, \ldots, D_k\}$, $k \le m$, of k DPSs. The idle time in a DPS is continuous and stretches from t' to d, where $t' \ge 0$. Let $\{G_{i_1}, G_{i_2}, \ldots, G_{i_j}\}$ be the ordered set of processors in D_i that have some idle time. The ordering is given by the requirement that $i_1 > i_2 > \cdots > i_j$. There exist $u_0^i, u_1^i, \ldots, u_j^i$ such that $0 \le u_0^i$, $u_j^i = d$, $u_r^i < u_{r+1}^i$, $1 \le r < j$, and G_{i_r} is idle exactly from u_{r-1}^i to u_r^i. Note that this implies that speed in the idle time of a DPS is nondecreasing (left to right). It will also be the case that $u_0^i \le u_0^{i+1}$, $1 \le i < k$.

C2. $R_i \ge R_{i+1}$, $1 \le i < k$.

It should be easy to see that the set of DPSs $\{D_1, D_2, \ldots, D_m\}$ defined above satisfies both C1 and C2 initially (i.e., when no tasks have been scheduled).

In the following statement of the three scheduling rules, t is the processing requirement of the task to be scheduled.

Rule R1

Condition: $t \le R_k$.

Action: Let $\{G_{k_1}, G_{k_2}, \ldots, G_{k_j}\}$, $k_1 > k_2 > \cdots > k_j$, be the processors with idle time in D_k. Let $u_0^k, \ldots, u_j^k$ be as defined in C1. Schedule the task from u_0^k up to time t' such that the effective processing assigned equals t. In case $t = R_k$, append all the processors of D_k to D_{k-1} (if $k \ne 1$). Replace k by $k - 1$. Note that this assignment leaves behind a set (possibly empty) of DPSs having the properties C1 and C2.

Rule R2

Condition: $R_i = t$ for some i, $1 \le i < k$.

Action: Schedule this task on D_i so as to use up all the available idle time in D_i. Combine the processors of D_i with those of D_{i+1} to form a new DPS with index i. Reindex the DPSs $D_{i+2}, \ldots, D_k$ to $D_{i+1}, \ldots, D_{k-1}$. Replace k by $k - 1$. Once again, C1 and C2 hold.

Rule R3

Condition: $R_{i+1} < t < R_i$ for some i, $1 \le i < k$.

Action: Let $\{G_{i_1}, G_{i_2}, \ldots, G_{i_j}\}$, $i_1 > i_2 > \cdots > i_j$, be the processors in D_i that have some idle time. Let $\{G_{r_1}, G_{r_2}, \ldots, G_{r_a}\}$, $r_1 > r_2 > \cdots > r_a$, be the processors in D_{i+1} that have some idle time. Let $u_0, u_1, \ldots, u_j$ be such that G_{i_b} is idle exactly from u_{b-1} to u_b. Let

$v_0, \ldots, v_a$ be the corresponding times for D_{i+1}. Note that $v_a = u_j = d$. From C1 it follows that $u_0 \leq v_0$. Let Q be the effective processing available in D_i from u_0 to v_0. If $Q + R_{i+1} \geq t$, then schedule the task to use up all the idle time in D_{i+1}. The remainder of the task is scheduled on D_i from u_0 up to t' where t' is such that the effective processing assigned on D_i from u_0 to t' plus that assigned on D_{i+1} from v_0 to d equals t. Note that $t' \leq v_0$. In case $Q + R_{i+1} < t$, determine t'' such that the effective processing time available in D_i from u_0 to t'' plus that available in D_{i+1} from t'' to d equals t. Note that because of the relationships among t, R_i, R_{i+1}, and Q, such a t'' must exist. Schedule the task for processing in the idle time of D_i from u_0 to t'' and in the idle time of D_{i+1} from t'' to d. Combine the processors of D_i and D_{i+1} to form a new DPS with index i. Reindex $D_{i+2}, \ldots, D_k$ to $D_{i+1}, \ldots, D_{k-1}$ and replace k by $k - 1$. It should be easy to verify that this scheduling rule preserves C1 and C2.

The fact that scheduling by these three rules leads to a DD-schedule (whenever one exists) is established in Lemma 2.

LEMMA 1. *Let $\{D_1, D_2, \ldots, D_m\}$ be a set of DPSs each consisting of exactly one generalized processor. Assume that no tasks are initially scheduled on any of these DPSs. Let $\{D'_1, D'_2, \ldots, D'_r\}$ be the set of DPSs remaining following the scheduling of some number of tasks using rules R1–R3. Let $|D'_j|$ be the number of processors in D'_j.*

(*i*) *If $j < r$, then exactly $|D'_j| - 1$ tasks are scheduled in D'_j.*
(*ii*) *If $j = r$ and rule R1 has never been used, then $|D'_r| - 1$ tasks are scheduled in D'_r.*

PROOF (BY INDUCTION ON $|D'_j|$). The proof for both parts is the same. If $j < r$, then rule R1 could not have been used to schedule any task in D'_j, as this rule can be used only on the last DPS. So for the remainder of the proof we may assume that D'_j is such that no task was scheduled in it using rule R1. Whenever rule R2 or R3 is used to schedule a task, two DPSs D''_p and D''_{p+1} get combined. Hence if $|D'_j| = 1$, then no task has been assigned to D'_j and the lemma holds. Assume that the lemma is true whenever $1 \leq |D'_j| < u$ and that D'_j has no task scheduled by an application of rule R1. We now show the lemma to be true when $|D'_j| = u$. Consider the last task scheduled in D'_j. Either rule R2 or R3 was used to schedule this task. Let D''_p and D''_{p+1} be the two DPSs combined during this rule application. Since $|D''_p| < u$, $|D''_{p+1}| < u$, and neither DPS contains a task scheduled by R1, it follows from the induction hypothesis that the number of tasks scheduled in each DPS is $|D''_p| - 1$ and $|D''_{p+1}| - 1$, respectively. $|D'_j| = |D''_p| + |D''_{p+1}|$, and the number of tasks scheduled in D'_j is $|D''_p| - 1 + |D''_{p+1}| - 1 + 1 = |D'_j| - 1$. $\square$

LEMMA 2. *Let G_i, $1 \leq i \leq m$; t_i, $1 \leq i \leq n$; T_i, $1 \leq i \leq m$; L_i, $1 \leq i \leq m$; and d be as defined in Theorem 1. Assume that $\max_i\{T_i/\sum_1^i L_j\} \leq 1$ and that initially the set of DPSs is $\{D_1, D_2, \ldots, D_m\} = \{\{G_1\}, \{G_2\}, \ldots, \{G_m\}\}$. If the n tasks are scheduled one by one, in any order, using rules R1–R3, then a DD-schedule is always obtained.*

PROOF. Suppose the tasks are being scheduled in the order $\sigma(1), \sigma(2), \ldots, \sigma(n)$ and that we are unable to schedule the ith task in this sequence. The task time t of this task is $t_{\sigma(i)}$. Since task $\sigma(i)$ cannot be scheduled, none of the conditions for R1–R3 holds. Let $\{D'_1, D'_2, \ldots, D'_k\}$ be the DPSs existing at this time. If $k \leq 1$, then it must be that $T_m/\sum_1^m L_j > 1$. This contradicts the assumption on d, and so we may assume that $k > 1$. From Lemma 1 it follows that exactly $|D'_1| - 1$ tasks are scheduled in D'_1. Let T' be the sum of the task times of these tasks. Clearly, $T' + t \leq T_{|D'_1|}$. From the way in which DPSs get combined by the scheduling rules, we see that the processors in D'_1 are $G_1, G_2, \ldots, G_j$, where $j = |D'_1|$. The total processing initially available on these j processors is $\sum_1^j L_i$. Since task $\sigma(i)$ cannot be scheduled, $t > R_1$ where R_1 is the remaining effective processing capability of D'_1. Hence $T' + t > \sum_1^j L_i$, and so $T_j/\sum_1^j L_i > 1$. This contradicts the assumption on d. Hence R1–R3 cannot fail to construct a DD-schedule. $\square$

In passing, we note that the algorithm resulting from rules R1–R3 can be used to generate minimum finish time schedules if we first find the smallest d for which the conditions of Theorem 1 hold.

3. *The Algorithm*

We are now ready to describe our algorithm for obtaining a DD-schedule (if one exists) for a uniform processor system. It should be noted that the discussion holds just as well for a GPS. Let d_i, $1 \le i \le k$, be the distinct due times of the n tasks. We may assume that $d_i < d_{i+1}$, $1 \le i < k$. Let $n_i > 0$ be the number of tasks with due time d_i, $1 \le i \le k$. Clearly, $\sum n_i = n$. Let $Q_{i,j}$ be the jth task with due time d_i, and let $s_1, s_2, \ldots, s_m$ be the speeds of the m uniform processors. We assume that $s_i \ge s_{i+1}$, $1 \le i < m$. The DD-schedule is constructed in k phases.

Phase 1 begins with the GPS $\{G_1, \ldots, G_m\}$ where each G_i is idle from 0 to d_1 and has constant speed s_i throughout this interval. The tasks $Q_{1,j}$, $1 \le j \le n_1$, are scheduled on this GPS as described in Section 2. In case all tasks cannot be scheduled, our algorithm terminates with the conclusion that no DD-schedule exists for the given task set. In case all tasks are scheduled, the algorithm described in Section 2 will leave behind a set $\{D_1, \ldots, D_p\}$ of p DPSs. The GPS $\{G_1', G_2', \ldots, G_m'\}$ for the next phase is constructed from $\{D_1, \ldots, D_p\}$ as follows. For $i \le p$, let $u_0^i, u_1^i, \ldots, u_j^i$ be as in C1 of Section 2. G_i' operates at speed 0 from 0 to u_0^i. From u_0^i to u_j^i its speed is given by the speed in the idle time of D_i. From $u_j^i = d_1$ to d_2 the speed of G_i' is s_i, $1 \le i \le p$. For $i > p$, G_i' operates at speed 0 from 0 to d_1 and at speed s_i from d_1 to d_2. Using the fact that $u_0^i \le u_0^{i+1}$, $1 \le i < j$ (see C1), it should be easy to verify that $\{G_1', \ldots, G_m'\}$ as constructed is a GPS. In phase 2, the tasks $Q_{2,j}$, $1 \le j \le n_2$, are scheduled on $\{G_1', \ldots, G_m'\}$ in the interval $[0, d_2]$. Again, this is done as in Section 2. From the DPS left behind, the GPS for the next phase may be constructed as above. This process is repeated until k phases have completed or one of them is unsuccessful in scheduling the tasks for that phase. If the former happens, a DD-schedule has been obtained. Otherwise, as we shall see, no DD-schedule exists at all!

Example 1. This example illustrates the working of the algorithm described above. We have $m = 5$ uniform processors $\{P1, P2, \ldots, P5\}$ with speeds 4, 3, 2, 2, and 1, respectively. Ten tasks are to be scheduled. The first four tasks have a due time $d_1 = 5$. The task times are 12, 3, 13, and 12. The remaining six tasks have a due time $d_2 = 10$. The task times are $\{13, 29, 10, 12, 5, 12\}$. One may readily verify that this task set satisfies the conditions of Theorem 1 for both phases. So a DD-schedule exists.

In phase 1 the GPS consists of five uniform processors available from 0 to d_1. Tasks with due time d_1 are scheduled in this phase. Steps 1–5 of Table I together with the figures of Figure 1 corresponding to these steps describe how the algorithm proceeds in phase 1. Initially each processor forms a DPS and we have five DPSs. The effective processing capability available on each of the DPSs is 20, 15, 10, 10 and 5, respectively. Task 1 has a task time of 12. Rule R3 is used and the DPSs $D2$ and $D3$ are combined. Figure 1 shows the schedule. The task is processed up to time 2 on $P2$ and from 2 to 5 on $P3$. Four DPSs remain. The available processing capabilities are 20, 13, 10, and 5. Task 2 is scheduled using rule R1. Task 3 is scheduled on $D2$ using rule R2. We are now left with three DPSs (see Figure 1 for step 4). Task 4 is scheduled using rule R3. It is to be processed on $P1$ from 0 to 1 and on $P4$ from 1 to 5. We are now left with two DPSs. $D1$ includes processors $P1$–$P4$ and $D2$ includes the single processor $P5$. This completes phase 1.

Step 6 reflects the initial configuration for phase 2. We have five generalized processors $G1$–$G5$. $G1$ operates at speed 2 (i.e., as $P4$) from 0 to 1, at speed 4 from 1 to 5 (as $P1$ from $D1$), and at speed 4 from 5 to 10. $G2$ operates at speed 0 from 0 to 3 (as $D2$ has no idle time in this interval), at speed 1 from 3 to 5 (i.e., as $P5$), and at speed 3 from 5 to 10. The solid area in Figure 1 (step 6) indicates processor operation at zero speed. Steps 7–12 indicate how the remaining six tasks are scheduled on $G1$–$G5$. For instance, task 5 is scheduled using rule R3. It is processed on $G2$ from 3 to 6 and on $G3$ from 6 to 10. Note that in terms

TABLE I. SCHEDULING RULE APPLIED AND RESULTING DPSS

Step	Task	Task time	Condition	Rule applied	Number of DPSs left	Processing capability for each DPS (R_i)
1	Initial	0	0	0	5	20, 15, 10, 10, 5
2	1	12	$R_3 < 12 < R_1$	R3	4	20, 13, 10, 5
3	2	3	$3 < R_4$	R1	4	20, 13, 10, 2
4	3	13	$13 = R_2$	R2	3	20, 10, 2
5	4	12	$R_2 < 12 < R_1$	R3	2	18, 2
6	Initial	0	0	0	5	38, 17, 10, 10, 5
7	5	13	$R_3 < 13 < R_2$	R3	4	38, 14, 10, 5
8	6	28	$R_2 < 28 < R_1$	R3	3	24, 10, 5
9	7	10	$R_2 = 10$	R2	2	24, 5
10	8	12	$R_2 < 12 < R_1$	R3	1	17
11	9	5	$5 < R_1$	R1	1	12
12	10	12	$12 = R_1$	R2	0	0

of the original processors $P1$–$P5$, this means that this task is to be processed on $P5$ from 3 to 5, on $P2$ from 5 to 6, and on $P3$ from 6–10. Figure 2 shows the final schedule. □

We now prove that the algorithm described above actually generates a DD-schedule for every set of n tasks $Q_{i,j}$, $1 \le j \le n_i$, $1 \le i \le k$, $\sum n_i = n$, for which such a schedule exists.

THEOREM 2. *The algorithm described above generates a DD-schedule for every task set $Q_{i,j}$, $1 \le j \le n_i$, $1 \le i \le k$, for which there exists a DD-schedule.*

PROOF. The proof is by induction on the number of distinct due times k. When $k = 1$, the proof follows from Lemma 2. Assume the algorithm works for all tasks sets that have DD-schedules and at most $K - 1$ due times. We proceed to show that the algorithm works for all task sets that have DD-schedules and have K distinct due times. Let S be a DD-schedule for the task set $Q_{i,j}$, $1 \le j \le n_i$, $1 \le i \le K$. Let r, $r \ge 0$, be the number of tasks with due time at least d_2 which have been assigned for a nonzero amount of processing in S in the interval $[0, d_1]$. Let the amount of processing carried out in this interval for these r tasks be $c_1, c_2, \ldots, c_r$. Let $\{D_1, D_2, \ldots, D_a\}$ be the set of DPSs following the scheduling in this interval (i.e., following phase 1 of the algorithm but before construction of the new GPS for phase 2). Then, from Theorem 1 it follows that $c_1, c_2, \ldots, c_r$ can also be scheduled on this set of DPSs. Hence there exists a DD-schedule for the task set $Q_{i,j}$, $1 \le j \le n_i$, $2 \le i \le K$, on the GPS constructed prior to the start of the phase 2 scheduling. From the induction hypothesis it follows that the algorithm will construct a DD-schedule for this set of tasks and this GPS (as the number of distinct due times is now $K - 1$). □

Number of Preemptions

LEMMA 3. *Let $\{D_1, D_2, \ldots, D_r\}$ be the set of DPSs resulting from the scheduling of some tasks on the GPS $\{G_1, G_2, \ldots, G_m\}$. Let p_i and q_i, respectively, be the number of preemptions and number of processors with idle time in D_i. Let $|D_i|$ be the number of processors in D_i. Then $p_i + q_i - 1 \le 2(|D_i| - 1)$.*

PROOF. If there are no tasks scheduled on the processors in D_i, then $|D_i| = 1$, $p_i = 0$, and $q_i = 1$. Clearly the theorem is true for this case. Assume the theorem is true for all DPSs with at most u tasks scheduled on them. Let D_i be such that $u + 1$ tasks have been scheduled on the processors of D_i.

If the last task is scheduled using rule R1, then $p_i' + q_i' - 1 \le 2(|D_i| - 1)$ preceding the scheduling of the $(u + 1)$st task. p_i' and q_i' are the number of preemptions and number of processors in D_i with idle time. The scheduling of the $(u + 1)$st task uses up all the idle time on $x - 1$ processors and a fraction (or all) on another processor. The number of

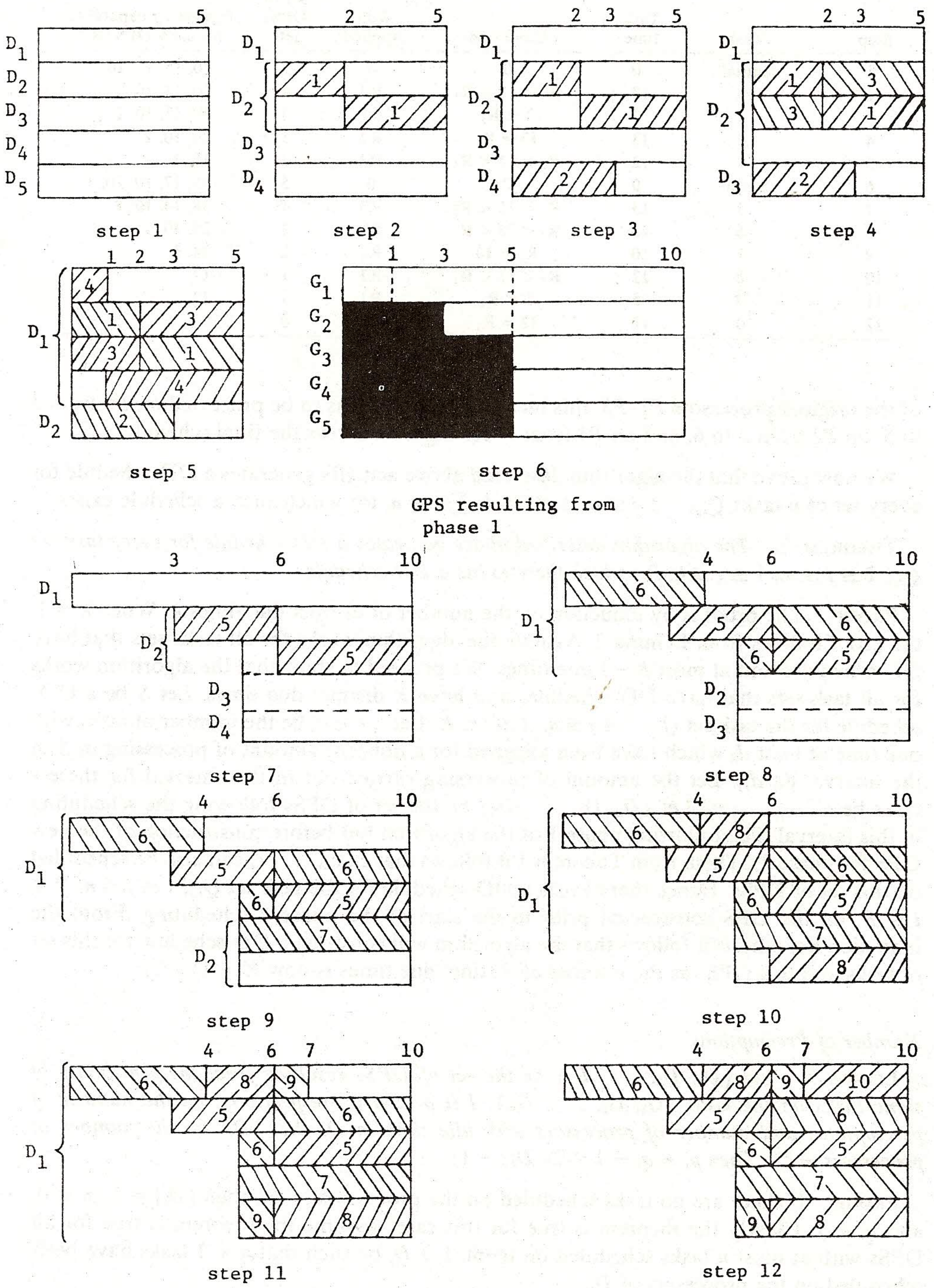

FIG. 1. Scheduling in Example 1.

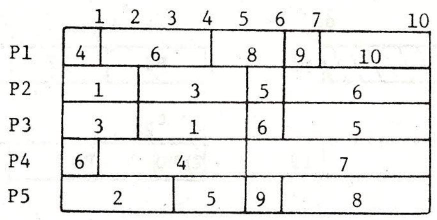

FIG. 2. Final schedule for Example 1.

preemptions introduced is at most $x - 1$. The number of processors with some remaining idle time is at most $q'_i - x + 1$. Hence $p_i + q_i - 1 \le p'_i + x - 1 + q'_i - x + 1 - 1 = p'_i + q'_i - 1 \le 2(|D_i| - 1)$.

If the $(u + 1)$st task is scheduled using rule R2, then D_i is created from two DPSs D'_i and D'_{i+1}. For these we have $p'_i + q'_i - 1 \le 2(|D'_i| - 1)$ and $p'_{i+1} + q'_{i+1} - 1 \le 2(|D'_{i+1}| - 1)$. The scheduling of the $(u + 1)$st task uses up all the idle time in D'_i. The number of preemptions introduced is $q'_i - 1$ and $q_i = q'_{i+1}$. Hence $p_i + q_i - 1 = p'_i + p'_{i+1} + q'_i - 1 + q'_{i+1} - 1 \le 2(|D'_i| - 1) + 2(|D'_{i+1}| - 1) = 2(|D_i| - 2) < 2(|D_i| - 1)$.

If rule R3 is used to schedule the $(u + 1)$st task, then let D'_i and D'_{i+1} be the two DPSs combined. Assume that the task is scheduled on v of the processors in D'_i and w of the processors in D'_{i+1}. Then $p_i = p'_i + p'_{i+1} + v + w - 1$; $q_i \le q'_i + q'_{i+1} - v - w + 2$. Hence $p_i + q_i - 1 \le p'_i + q'_i + p'_{i+1} + q'_{i+1} \le 2\,|D'_i| - 1 + 2\,|D_{i+1}| - 1 = 2(|D_i| - 1)$. □

If the n tasks to be scheduled have k distinct due times, then the algorithm described earlier obtains the schedule in k phases. At the beginning of phase i we have a GPS available from time 0 to time d_i where d_i is the ith smallest due time. Assume that the DPSs remaining after the ith phase are $D^i_1, D^i_2, \ldots$. Let p^i_j be the number of preemptions in D^i_j. This number includes only the preemptions introduced in phase i and not those due to the mapping of the DPSs of the previous phase into the GPS for this phase. Let q^i_j be the number of processors in D^i_j with some idle time. Then from Lemma 3 it follows that $p^i_j + q^i_j - 1 \le 2(|D^i_j| - 1)$. Hence $\sum_{j=1}^{w_i} p^i_j + \sum_{j=1}^{w_i} q^i_j \le 2m - w_i$, where w_i is the number of DPSs. The total number of preemptions introduced in all phases is $\sum_i \sum_j p^i_j$. This sum is at most $2km - \sum(w_i + \sum_{j=1}^{w_i} q^i_j)$.

To get a bound on the total number of preemptions we need to add the preemptions that result from the mapping of uniform processors to generalized processors (see Figure 3). The phase 1 scheduling introduces no additional preemptions of this kind, as each generalized processor is a uniform processor. For the remaining phases, however, each generalized processor will in general represent different uniform processors in different time intervals. If a generalized processor represents b different uniform processors, then at most $b - 1$ additional preemptions can be introduced when the schedule is mapped back onto the uniform processors. Thus the additional phase 2 preemptions are at most $\sum_1^{w_1} q^1_j$. (Note that a processor in phase 2 is a DPS of phase 1 followed by a uniform processor.) If f preemptions of this type are actually introduced in mapping the phase 2 schedule onto the uniform processors, then no more than $\sum_1^{w_2} q^2_j + \sum_1^{w_1} q^1_j - f$ such preemptions can be introduced by mapping the phase 3 schedule from the generalized processors back onto the uniform processors. In general, the total number of additional preemptions that can be created by mapping the GPS schedules onto the uniform processors is at most $\sum_{i=2}^{k} \sum_{j=1}^{w_{i-1}} q^{i-1}_j$. Thus the total number of preemptions in the DD-schedule constructed by our algorithm is at most $\sum_{i=1}^{k} \sum_{j=1}^{w_i} p^i_j + \sum_{i=2}^{k} \sum_{j=1}^{w_{i-1}} q^{i-1}_j$. Using our earlier result we obtain $2km - \sum_{i=1}^{k} w_i - \sum_{j=1}^{w_k} q^k_j$ as a bound on the number of preemptions.

Let n_i be the number of tasks with due time d_i. Then $\sum_{i=1}^{k} n_i = n$. If $n_i < m$, then $w_i \ge m - n_i$. To see this, note that each time a task is scheduled, the number of DPSs either remains unchanged (rule R1 is used) or decreases by one (rules R2 or R3). At the beginning of each phase the number of DPSs is m. If $n_i \ge m$, then $w_i \ge 1$. This observation allows us to further simplify the bound on the number of preemptions. The bound now becomes $2km - \sum_{i=1}^{k} (m - n_i) - \sum_j q^k_j \le km + n$.

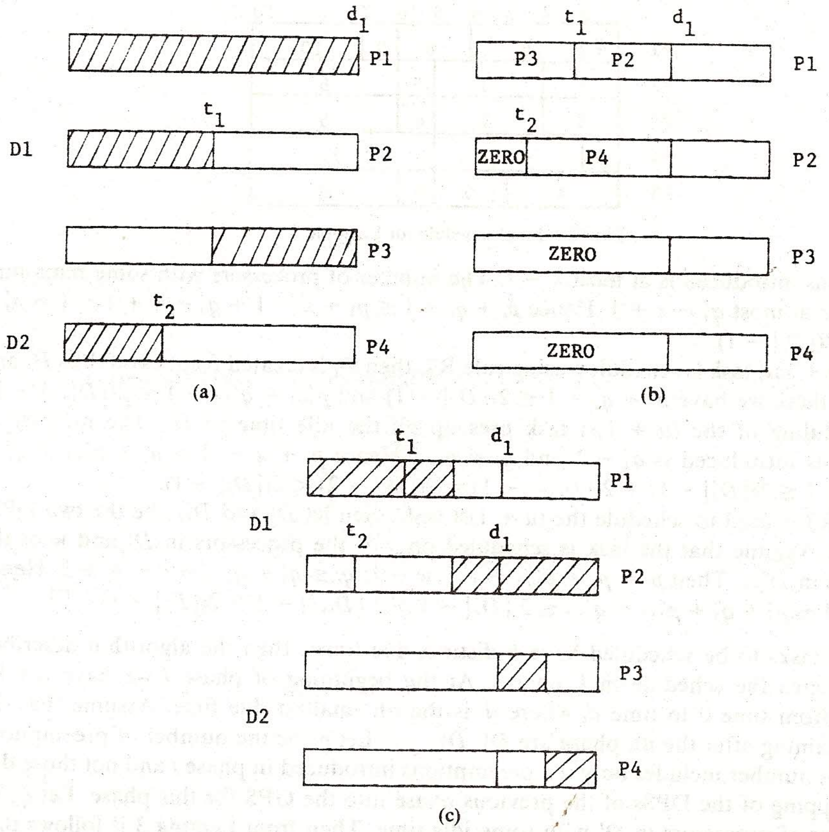

FIG. 3. Additional preemptions caused by a GPS. (a) DPSs *D1* and *D2* following phase 1. (b) GPS for phase 2. (c) DPSs *D1* and *D2* following phase 2. Note: In Figure 3c there is an uncounted preemption at t_1 on *P1* and at d_1 on *P2*. Another may occur at d_1 on P_1 later as uniform processor *P3* is to the left of d_1, while uniform processor *P1* is to the right.

The bound $km + n$ can actually be reduced to $k(m - 1) + n$. To see this, consider the first DPS D_1^i remaining at the end of phase *i*. If this DPs has idle time, then this time must be from v to d_i for some v, $0 \le v < d_i$. In case the uniform processor P_1 does not represent any of this idle time, then rule R2 must have been used at least once in the *i*th phase. From the analysis carried out in Lemma 3, it follows that $p_1^i + q_1^i \le 2(|D_1^i| - 1)$; i.e., there is at least one fewer preemption contributed in this phase. In case the uniform processor P_1 is idle in D_1^i, then it must be idle from v' to d_k for some $v' < d_i$. The generalized processor G_1 constructed for phase $i + 1$ is therefore made up of the same uniform processor on both sides of the boundary time d_i. Hence, when the resulting schedule is viewed in terms of the original uniform processors, the boundary d_i on G_1 does not contribute to a preemption (this preemption was added into the bound of $km + n$). In case D_1 has no idle time at all it is again clear that the scheduling of the last task produces one fewer preemption than included in the bound. Hence we conclude that the bound $km + n$ includes at least k preemptions more than is possible. The new bound is therefore $k(m - 1) + n$.

THEOREM 3. *The scheduling algorithm that results from the use of rules R1–R3 generates DD-schedules with at most $k(m - 1) + n$ preemptions. k is the number of distinct due times. Since $k \le n$, no more than mn preemptions exist in any DD-schedule constructed by the algorithm.*

In the next example we show that there exist task sets for which the minimum number of preemptions in any DD-schedule is almost $(m - 1)n$.

Example 2. Let s_i, $1 \le i \le m$, be the speeds of m uniform processors $\{P_1, P_2, \ldots, P_m\}$. Let $s_i > s_{i+1}$, $1 \le i < m$. Let d_i, $1 \le i \le n$, $n > m$, be the due times of the n tasks. Assume that $d_i < d_{i+1}$, $1 \le i < n$. Define $d_0 = 0$ and $t_i = \sum_{j=1}^{i} (d_j - d_{j-1}) * s_{i+1-j}$, $1 \le i \le m$. Define $t_i = \sum_{j=1}^{m} (d_{i+1-j} - d_{i-j}) * s_j$, $m < i \le n$. It should be obvious that there exists only one DD-schedule for this set of tasks. In this schedule task i is scheduled on processor $i + 1 - j$ in the interval d_{j-1} to d_j, $1 \le j \le i$, $1 \le i \le m$. For $i > m$, task i is scheduled on processor j in the interval d_{i-j} to d_{i+1-j}, $1 \le j \le m$. The total number of preemptions is $\sum_{i=1}^{m-1} i + (m - 1)(n - m)$. This is equal to $(m - 1)(n - m/2)$. □

By using appropriate data structures it is possible to implement the scheduling rule described in $O(mn + n \log n)$ time. $n \log n$ time is needed to sort the tasks by due times. Since each GPS has at most m processors in it, each task can be scheduled in $O(m)$ time. Hence a total of $O(nm)$ time is needed to schedule all the tasks. The implementation details appear in the appendix. It is shown that all updates of GPSs and other overheads do not increase the asymptotic complexity of the algorithm.

Finally, we wish to point out that the scheduling rule described in this paper can be used on a more generalized processor model.[1] In this model the speed of machine i is given by a function γ_i where

(1) $\gamma_i(\tau)$ is the speed of machine i at time $\tau \ge 0$ ($\gamma_i(\tau) \ge 0$),
(2) γ_i is a nondecreasing function of τ,
(3) the γ_i's satisfy the ordering property that if $i < j$, then $\gamma_i(\tau) \ge \gamma_j(\tau)$, $\tau \ge 0$.

Appendix

In order to determine the asymptotic computing time requirements of the proposed algorithm, it is necessary to specify the algorithm in greater detail. In particular, we need to specify the data structures to be used to represent the DPSs.

Each DPS will be represented by a chain of nodes. The chain will have a head node with five fields: UP, DOWN, LINK, LAST, and R. R is the remaining effective processing capability of the DPS. The field LINK is used to link to the remainder of the chain for the DPS. LAST is a pointer to the last node on this chain. The head nodes of the DPS chains are linked together as a doubly linked list by using the fields UP and DOWN. Each node in a DPS chain (other than the head node) has three fields: M, ST, and LINK. LINK is used to link to the next node in the chain. For each distinct processor in the DPS that has idle time, there is one node on the DPS chain. The M field is the index of the processor and the ST field is the time at which this processor becomes available. The nodes on the chain are in increasing order of ST. If X and Y are two adjacent nodes on such a chain, then processor $P_{M(X)}$ is idle from ST(X) to ST(Y). If Y is the last node on the chain and the algorithm is in phase i, then $P_{M(Y)}$ is idle from ST(Y) to d_i. d_i is the ith distinct due time. Figure 4 shows the representation of the DPSs following the initialization for phase 2 of Example 1. The positions of various fields is shown in Figure 5.

Our algorithm for scheduling tasks is more formally stated as procedure ONERT (see Figure 6). The algorithm assumes the existence of standard list processing algorithms GETNODE and RET. The former provides a chain node currently not in use. The latter frees a chain node that was in use and is no longer needed. There are m head nodes indexed $D(i)$, $1 \le i \le m$. It is assumed that the tasks have been sorted into nondecreasing order of their due times. The number of distinct due times is k. The due times themselves are d_i, $1 \le i \le k$, and $d_i < d_{i+1}$, $1 \le i < k$. We assume $d_0 = 0$ has also been defined. Tasks with the same due time are in order of nonincreasing task times. The n tasks have task times $t_1, t_2, \ldots, t_n$. The speeds of the m processors are s_i, $1 \le i \le m$. It is assumed that $s_i \ge s_{i+1}$, $1 \le i < m$. For convenience, the algorithm uses q_i, $0 \le i \le k$. The q_i's are defined such that all tasks with index j, $q_{i-1} < j \le q_i$, have due time d_i, $1 \le i \le k$. Note that $q_0 = 0$ and $q_k = n$.

[1] We are grateful to an anonymous referee for pointing out this generalization.

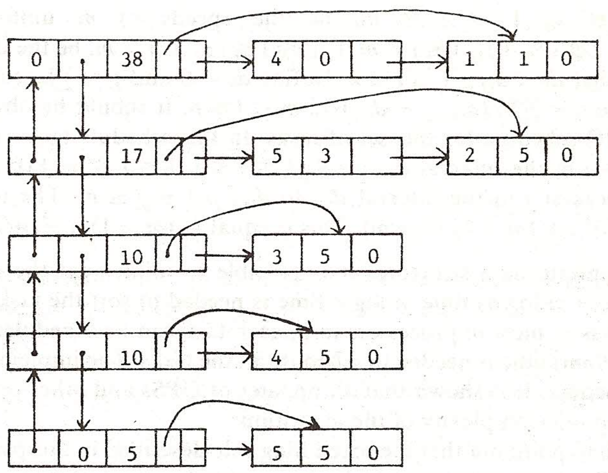

FIG. 4. Representation for Example 1.

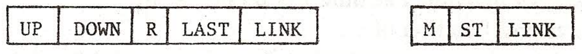

FIG. 5. Node structures. (a) Head node. (b) Chain node.

Much of the code that should be in ONERT has been replaced by calls to four different algorithms: RULE_R1, RULE_R2, RULE_R3, and INIT. In an actual program these would be written in line rather than as separate subalgorithms. For this reason our statement of these subalgorithms contains no parameters.

The main loop (lines 2–20) cycles through the k scheduling phases. Procedure INIT sets up the GPS for this phase. It uses H to determine whether any DPSs remain from the previous phase. $H = 0$ iff there are no DPSs with idle time remaining at the end of the previous phase. Since the DPSs are always linked in order of nonincreasing $R(D(i))$, the test in line 5 correctly determines whether task j can be scheduled. The loop of lines 8–19 schedules the tasks with due time d_i one by one. At the start of each iteration of this loop H either points to a DPS with $R(D(H)) \geq t_j$, or if this is not the case, then there is no DPS with this property. One may readily verify that lines 11–18 correctly determine the scheduling rule to be used.

The four unspecified subalgorithms used in ONERT work as follows:

I. INIT (Figure 7). INIT sets up the initial m DPSs for each phase. When $H = 0$, no DPSs with idle time remain following the previous phase. When $H \neq 0$, some DPSs with nonzero idle time remain. These are represented by the list of head nodes $D(1)$; DOWN($D(1)$); DOWN(DOWN($D(1)$)); etc. Each of these is associated with a processor that is available from d_{i-1} to d_i. This association is done by simply linking the list for a DPS from the previous phase to a node representing the uniform processor available from d_{i-1} to d_i (lines 4–14). In case the fastest processor on the DPS is the same as the one to which it is being associated, no new node is needed. The availability of the uniform processor is implicit in the start time of the fastest processor in the DPS. Only the effective processing capability is to be updated in this case. Lines 15–21 set up the DPSs that are available only from d_{i-1} to d_i. Note that we do not explicitly represent the portion of a generalized processor that operates at speed zero.

II. RULE_R1 (Figure 8). At entry to this subalgorithm H points to the DPS onto which task j is to be scheduled. In the loop of lines 4–12 we move down the list for $D(H)$ until t_j units of processing time have been accumulated. The DPS is appropriately updated

```
line  procedure ONERT(t, s, d, m, k, q)
          //see text for an explanation of variables in use//
  1       H ← 0 //initialization parameter//
  2       for i ← 1 to k do //phase i//
  3           call INIT //set up GPS//
  4           j ← q_{i-1} + 1 //next task//
  5           if R(D(1)) < t_j then [print("no DD-schedule")
  6                                  stop]
  7           H ← largest index v such that R(D(v)) ≥ t_j
  8           for j ← q_{i-1} + 1 to q_i do //all tasks with due time d_i//
  9               if H = 0 or R(D(H)) < t_j then [print("no DD-schedule")
 10                                               stop]
                  //find last DPS for j//
 11               while DOWN(D(H)) ≠ 0 and R(D(H)) > t_j do
 12                   H ← DOWN(D(H))
 13               end
 14               case
 15                   :DOWN(D(H)) = 0 and R(D(H)) ≥ t_j: call RULE_R1
 16                   :R(D(H)) = t_j: call RULE_R2
 17                   :else: call RULE_R3
 18               end
 19           end
 20       end
 21   end ONERT
```

FIG. 6. Procedure ONERT.

```
line  procedure INIT
  1       v ← 1 //next DPS to set up//
  2       if H ≠ 0 then [//concatenate last phase DPSs//
  3                   H ← 1
  4                   while H ≠ 0 do
  5                       D(v) ← D(H); R(D(v)) ← R(D(v)) + s_i*(d_i − d_{i-1})
  6                       if M(LAST(D(v))) ≠ v then
  7                           [call GETNODE(X)
  8                           M(X) ← v; ST(X) ← d_{i-1}
  9                           LINK(X) ← 0
 10                           LINK(LAST(D(v))) ← X
 11                           LAST(D(v)) ← X]
 12                       H ← DOWN(H); DOWN(v) ← v + 1
 13                       UP(v) ← v − 1; v ← v + 1
 14                   end]
 15       for u ← v to m do
 16           call GETNODE(X)
 17           LINK(D(u)) ← X; LINK(X) ← 0
 18           R(D(u)) ← s_i*(d_i − d_{i-1}); M(X) ← u
 19           UP(D(u)) ← u − 1; ST(X) ← d_{i-1}
 20           DOWN(D(u)) ← u + 1
 21       end
 22       UP(D(1)) ← DOWN(D(m)) ← 0
 23   end INIT
```

FIG. 7. Procedure INIT.

to reflect the assignment of task j (lines 15–17). Finally, in lines 18–23, H is updated to point to a DPS with enough available capacity to accommodate the next task (if any). If $R(D(H)) = 0$, then lines 20–21 delete the DPS. If UP$(D(H)) \neq 0$, then the next task must fit on UP$(D(H))$ as tasks are in nonincreasing order of task times. If no idle time remains, then H is set to zero (line 22).

III. RULE_R2 (Figure 9). Task j is scheduled to use up all the idle time on $D(H)$ (lines 1–8). This DPS is removed from the list of DPSs (line 9) and H updated to point to a DPS onto which the next task must fit. In case no DPS with idle time remains, DOWN$(H) = 0$ and H is updated to zero in line 11.

```
procedure RULE_R1
    sum ← 0 //amount already assigned//
    $Q$ ← LINK($D(H)$) //first node in chain//
    $st$ ← ST($Q$)
    loop
        next ← LINK($Q$)
        if next = 0 then $ft \leftarrow d_i$
                    else $ft$ ← ST(next)
        if sum + $(ft - st)*s_{M(Q)} \geq t_j$ then exit
        print ("schedule," $j$, "on," M($Q$), "from," $st$, "to," $ft$)
        sum ← sum + $(ft - st)*s_{M(Q)}$
        call RET($Q$); $Q$ ← next; $st \leftarrow ft$
    forever
    $qt \leftarrow (t_j - \text{sum})/s_{M(Q)} + st$
    print ("schedule," $j$, "on," M($Q$), "from," $st$, "to," $qt$)
    if $ft \neq qt$ then [ST($Q$) ← $qt$; LINK($D(H)$) ← $Q$]
        else LINK($D(H)$) ← next
    R($D(H)$) ← R($D(H)$) − $t_j$
    case
        :UP($D(H)$) ≠ 0 and R($D(H)$) ≠ 0: $H$ ← UP($D(H)$)
        :UP($D(H)$) ≠ 0 and R($D(H)$) = 0: $H$ ← UP($D(H)$)
                                           DOWN($D(H)$) ← 0
        :UP($D(H)$) = 0 and R($D(H)$) = 0: $H$ ← 0
    end
end RULE_R1
```

FIG. 8. Procedure RULE_R1.

```
procedure RULE_R2
    $Q$ ← LINK($D(H)$); $st$ ← ST($Q$)
    while $Q \neq 0$ do
        next ← LINK($Q$)
        if next = 0 then $ft \leftarrow d_i$
                    else $ft$ ← ST(next)
        print("schedule," $j$, "on," M($Q$), "from," $st$, "to," $ft$)
        call RET($Q$); $Q$ ← next; $st \leftarrow ft$
    end
    delete $D(H)$ from the doubly linked list of head nodes
    if UP($D(H)$) ≠ 0 then $H$ ← UP($D(H)$)
                    else $H$ ← DOWN($D(H)$)
end RULE_R2
```

FIG. 9. Procedure RULE_R2.

IV. RULE_R3 (Figure 10). Initially, task j is assigned to run on the DPS H from time 0 to time d_i. SUM represents the total amount of processing that can be done by the current assignment. In the loop of lines 5–20, this initial assignment is modified so that task j is assigned to the DPS U from 0 to TU and on H from TU to d_i. Note that the DPSs H and U may operate at speed zero for some initial portion of either or both of these intervals. TH represents the time up to which the DPS H operates at speed SX. In the while loop of 6–14, portions of task j are moved from H to U. $(LT-TU)*(s_{M(Y)} - SX)$ represents the effective increase in processing that can be achieved by transferring j from H to U for $LT-TU$ time units. Note that $s_{M(Y)} - SX \geq 0$ by definition of a GPS. In case the conditional of line 11 is true, the equalizing time is between TU and LT. Otherwise, it must be larger than LT. In the latter case task j is reassigned to U from TU to LT. On a normal exit from the loop of lines 6–14 it is necessary to reassign more of j from H to U. Lines 15–19 reset the variables so that reassignment can be carried out up to time TH. On exit from line 11, line 21 correctly determines the additional amount to be moved from H to U so that exactly t_j units of processing are obtained and the processors in H and U together form a DPS. Note that $s_{M(Y)} - SX > 0$ as otherwise the movement from H to U cannot increase SUM to exceed t_j (line 11). Having found the equalizing time TU (line 22), the remainder of the algorithm schedules j on U from 0 to TU (line 23) and on H from TU to d_i (line 25). Note that since U operates at speed 0 from 0 to ST(LINK($D(U)$)) and H

```
line  procedure RULE_R3
 1        SUM ← R(D(H)); U ← UP(D(H))
 2        Y ← LINK(D(U)); TU ← ST(Y)
 3        X ← LINK(D(H)); TH ← ST(X)
 4        N ← Y; SX ← 0
 5        loop
 6            while TU < TH do
 7                Y ← N; N ← LINK(N)
 8                if N = 0 then FT ← $d_i$
 9                         else FT ← ST(N)
10                LT ← min{TH, FT}
11                if SUM + (LT − TU)*($s_{M(Y)}$ − SX) ≥ $t_j$ then go to L1
12                SUM ← SUM + (LT − TU)*($s_{M(Y)}$ − SX)
13                TU ← LT
14            end
15            X1 ← LINK(X)
16            if X1 = 0 then FT ← $d_i$
17                     else FT ← ST(X1)
18            SX ← $s_{M(X)}$; X ← X1; TH ← FT
19            N ← Y
20        forever
21    L1: DEL ← ($t_j$ − SUM)/($s_{M(Y)}$ − SX)
22        TU ← TU + DEL
23        print schedule for j on U from ST(LINK(D(U)) to TU
24        TH ← max{TU, ST(LINK(D(H)))}
25        print schedule for j on H from TH to $d_i$
26        combine the idle time on DPS H from ST(LINK(D(U))) to TH with the
          idle time on DPS U from TU to $d_i$ to form a new DPS U
27        return all nodes freed in line 26
28        delete D(H) from the doubly linked list of head nodes
29        if UP(D(U)) ≠ 0 then H ← U
                         else H ← UP(D(U))
30    end RULE_R3
```

FIG. 10. Procedure RULE_R3.

operates at speed 0 from 0 to ST(LINK($D(H)$)), lines 23 and 25 omit the schedule portion at zero speed. Line 26 combines the list for H with that for U. Only those nodes representing available idle time are retained. Others are discarded (line 27). Line 29 resets H so that the next task to be scheduled must fit on this DPS.

ANALYSIS OF ONERT. INIT clearly takes $O(m)$ time. It should be clear that the list for any DPS has at most m nodes at any time. Hence each application of any of the three rules takes at most $O(m)$ time. Since n applications are made, the total time for rule applications as well as for GPS initializations is $O(mn)$. If we include the time needed to initially sort the tasks in order of due times, then the total time needed to obtain a DD-schedule is $O(mn + n \log n)$. From the discussion in the previous section it is clear that at most $k(m - 1) + n$ preemptions are introduced. k is the number of distinct due times. When $k = 1$, it may be shown that at most $2(m - 1)$ preemptions are introduced.

REFERENCES

(Note. Reference [4] is not cited in the text.)

1. BRUNO, J., AND GONZALEZ, T. Scheduling independent tasks with release dates and due dates on parallel machines Tech. Rep. No. 213, Pennsylvania State U., University Park, Pa., Dec. 1976.
2. GONZALEZ, T., AND SAHNI, S. Preemptive scheduling of uniform processor systems. *J. ACM 25*, 1 (Jan. 1978), 92–101.
3. HORN, W. Some simple scheduling algorithms. *Nav. Res. Log. Qtr. 21* (1974), 177–185.
4. LIU, J., AND LIU, C. Bounds on scheduling algorithms for heterogeneous computing systems. In *Information Processing 1974*, North Holland, Amsterdam, pp. 349–353.
5. SAHNI, S. Preemptive scheduling with due dates. *Oper. Research 27*, 5 (Sept.–Oct. 1979), 925–934.

RECEIVED MAY 1977; REVISED MARCH 1979; ACCEPTED JUNE 1979