ON-LINE SCHEDULING OF REAL-TIME TASKS*

Kwang S. Hong and Joseph Y-T. Leung
Computer Science Program
University of Texas at Dallas
Richardson, TX 75083

Abstract : We consider the problem of on-line scheduling a set of n independent, real-time tasks on $m \geq 1$ identical processors. An on-line scheduler is said to be optimal if it performs as well as the best off-line scheduler. We show that no optimal on-line scheduler can exist if the tasks have more than one distinct deadline. We then give an optimal on-line scheduler for tasks with one common deadline. Finally, we consider environments where processors can go down unexpectedly, and an optimal on-line scheduler is also given for this case.

1. Introduction.

One of the most difficult problem in the design of real-time systems is the scheduling of sporadic tasks. Sporadic tasks are tasks that have hard deadlines and random arrival times. Because of the unpredictable nature of their arrivals, it is extremely difficult to design a real-time system with the guarantee that the deadlines of all sporadic tasks can be met. In practice, we try to meet the deadlines of the most important tasks when the system is heavily loaded. Thus, we need an on-line scheduler that works as follows. When one or more sporadic tasks arrive at time t, the execution times and deadlines of the tasks are made known to the system. The on-line scheduler is called upon to decide if the newly arrived tasks, along with the unfinished tasks at time t, could be completed so that all deadlines are met. If it were possible to meet all deadlines, the system would execute the tasks according to the schedule constructed by the on-line scheduler. Otherwise, the system would try to meet the deadlines of the most important tasks and allows the deadlines of the less important tasks to be missed. The above process is repeated whenever new tasks arrive. The intention of this paper is to study the design of such an on-line scheduler. Specifically, we study the question of whether we can have an on-line scheduler that performs as well as the best off-line scheduler.

Formally, a task system $\tau = (\{T_i\}, \{r(T_i)\}, \{d(T_i)\}, \{e(T_i)\})$ consisting of n independent tasks is to be scheduled on a processor system $P = \{P_i\}$ consisting of $m \geq 1$ identical processors. For each task T_i, $r(T_i)$, $d(T_i)$ and $e(T_i)$ denote the *release time, deadline* and *execution time* of T_i, respectively. A task system τ is said to be *feasible* on $m \geq 1$ processors if there is a preemptive schedule for the tasks in τ on m processors such that each task T_i is executed within its *executable interval* $EI_i = [r(T_i), d(T_i)]$. Such a schedule is called a *feasible schedule*. An *off-line* scheduler is one that schedules tasks with a complete knowledge of the release times of the tasks, while an *on-line* scheduler is one that schedules tasks without this knowledge. A scheduler (off-line or on-line) is said to be *optimal* for m processors if it constructs a feasible schedule for every task system that is feasible on m processors.

*Research supported in part by the ONR grant N00014-87-K0833.

Horn [3] has given an optimal off-line scheduler for any $m \geq 1$. He reduces the problem of finding a feasible schedule to a network flow problem. Using standard network flow technique, his scheduler can be implemented to run in $O(n^3)$ time. There are faster off-line schedulers for special task systems; see [2,8]. Since an on-line scheduler schedules tasks with less information than an off-line scheduler, it is more difficult to obtain an optimal on-line scheduler. For a single processor, it is not difficult to show that the Deadline Algorithm [1,4,5,7] is an optimal on-line scheduler. At each instant of time t, the Deadline Algorithm schedules that active task (a task which has been released but has not yet finished execution) whose deadline is closest to t; ties can be broken arbitrarily. In this paper we give an optimal on-line scheduler for task systems with one common deadline and any $m \geq 1$. We also show that for every $m > 1$, no optimal on-line scheduler can exist for task systems with two or more distinct deadlines. Sahni and Cho [9] have studied the so-called *nearly on-line* schedulers. A nearly on-line scheduler schedules tasks like an on-line scheduler, except that it has the additional information of when the next release time is. They give an optimal nearly on-line scheduler for task systems with one common deadline and any $m \geq 1$. Sahni [8] has shown that for every $m > 1$, no optimal nearly on-line scheduler (and hence on-line scheduler) can exist for task systems with arbitrary deadlines.

We also consider environments where processors can go down unexpectedly. We assume, however, that the duration of the down time is known at the moment a processor goes down. We model this situation by adding ***urgent tasks*** to the task system. A task T is said to be an urgent task if $d(T) = r(T) + e(T)$. Whenever a processor goes down, we consider the processor as if it is executing an urgent task that has just arrived. Assuming that urgent tasks have arbitrary deadlines and non-urgent tasks have one common deadline, we give an optimal on-line scheduler for this class of task systems and any $m \geq 1$. We note that our scheduler is still optimal even if the durations of processor down times are unknown. However, our scheduler might not be able to detect infeasible task systems until the common deadline is reached.

In the next section we show that for every $m > 1$, no optimal on-line scheduler can exist for task systems with two or more distinct deadlines. We also give an optimal on-line scheduler for task systems with one deadline and any $m \geq 1$. In section 3, we give an optimal on-line scheduler for task systems that can have urgent tasks. Finally, we draw some concluding remarks in the last section.

2. On-Line Scheduler.

In this section we first show that for every $m > 1$, no optimal on-line scheduler can exist for task systems with two or more distinct deadlines. We then give an optimal on-line scheduler for task systems with one deadline.

Theorem 1 : For every $m > 1$, no optimal on-line scheduler can exist for task systems with two or more distinct deadlines.

Proof : We prove the theorem for $m = 2$. It is easy to see that the proof can be generalized to $m > 2$. Suppose there is an optimal on-line scheduler for two processors and consider the following scenario. At time 0, three tasks T_1, T_2 and T_3 are released with $d(T_1) = d(T_2) = 4$, $d(T_3) = 8$, $e(T_1) = e(T_2) = 2$ and $e(T_3) = 4$. The optimal on-line scheduler would schedule these three tasks on two processors, starting at time 0. We have the following two cases to consider, depending on whether T_3 is executed in the interval (0, 2) or not.

Case I: T_3 is executed in the interval (0, 2).

In this case, at least one of T_1 and T_2 cannot be finished by time 2. Since T_1 and T_2 are identical tasks, we may assume that T_2 is not finished at time 2. Now, consider the scenario where T_4 and T_5 are released at time 2 with $d(T_4) = d(T_5) = 4$ and $e(T_4) = e(T_5) = 2$. Clearly, the schedule constructed by the optimal on-line scheduler is not feasible. However, the task system is readily seen to be feasible on two processors.

Case II: T_3 is not executed in the interval (0, 2).

In this case, at most two units of T_3 can be executed by time 4. Thus, at time 4, the remaining execution time of T_3 is at least two units. Now, consider the scenario where T_4 and T_5 are released at time 4 with $d(T_4) = d(T_5) = 8$ and $e(T_4) = e(T_5) = 4$. Again, the optimal on-line scheduler fails to construct a feasible schedule while the task system is feasible on two processors. □

As a result of Theorem 1, we are motivated to study task systems with one common deadline. In the following we give an optimal on-line scheduler for this class of task systems. Our algorithm, to be called Algorithm A, iteratively reschedules tasks whenever new tasks arrive. The algorithm to reschedule tasks is called Algorithm Reschedule, and it is based on McNaughton's algorithm [6] to find the shortest preemptive schedule for a set of independent tasks on m processors. The idea of McNaughton's algorithm is as follows. Let x_1 denote the length of the longest task and x_2 denote the total execution time of the tasks divided by m. Let x denote the larger of x_1 and x_2. McNaughton's algorithm schedules the tasks in the interval $(0, x)$. Tasks are sequentially scheduled on P_1, starting at time 0. When a task is encountered that finishes later than x, we reschedule that portion of the task that exceeds x on P_2, starting at time 0. The above process is iterated until all tasks have been scheduled.

Algorithm Reschedule

```
Comments :   m — the number of processors
             t — the current time
             d — the common deadline
             U — the set of active tasks
interval ← d − t;
capacity ← m * interval;
E ← the total remaining execution time of the tasks in U;
longest ← the largest remaining execution time of the tasks in U;
if E > capacity or longest > interval, then print "infeasible" and stop;
p ← 1;
avail ← m;
while U ≠ ∅
{
    width ← E/avail;
     if longest ≤ width
     then
     { Schedule the tasks in U on P_p through P_m in the interval
          (t, t+width) by McNaughton's algorithm;
       U ← ∅;
     }
     else
     { U' ← the set of tasks with remaining execution time longest;
       k ← the number of tasks in U';
       Schedule the tasks in U' on P_p through P_{p+k−1}, one task per
          processor;
       avail ← avail − k;
       U ← U − U';
       E ← the total remaining execution time of the tasks in U;
       longest ← the largest remaining execution time of the tasks
          in U;
       p ← p + k;
     }
}
```

Figure 1(a) shows the schedule constructed by Algorithm Reschedule at time 0 for the task system given in Figure 1. Observe that P_1 has only one task assigned to it and that its finishing time is larger than the minimum finishing time of all processors. From the nature of the algorithm, it is easy to see that if the finishing time of a processor is larger than the minimum finishing time of all processors, then the processor has only one task assigned to it, starting at time t. We state this fact without proof in the following lemma. First, let us define some notations. If S is a schedule, then $f_i(S)$ denotes the finishing time of P_i in S. When S is understood, we simply denote $f_i(S)$ by f_i.

Lemma 1 : Let S be the schedule constructed by Algorithm Reschedule at time t and f^* be the minimum finishing time of all processors in S. For any $1 \leq i \leq m$, if $f_i > f^*$, then P_i has only one task assigned to it. Furthermore, the task is assigned to P_i from t until f_i.

Algorithm A is given below. It simply calls Algorithm Reschedule whenever new tasks arrive.

Algorithm A

```
Whenever new tasks arrive, do
{     t ← the current time;
      U ← the set of active tasks at time t;
      Call Algorithm Reschedule to schedule the tasks in U;
}
```

Figure 1(b) shows the final schedule constructed by Algorithm A for the task system given in Figure 1. The next theorem shows that Algorithm A is an optimal on-line scheduler for task systems with one common deadline.

Theorem 2 : For any task system τ with one common deadline to be scheduled on $m \geq 1$ processors, Algorithm A constructs a feasible schedule for τ if and only if τ is feasible on m processors.

Proof : It is clear that if Algorithm A successfully constructs a schedule for τ, then the schedule is a feasible one and hence τ is feasible on m processors. To complete the proof, we need to show that if Algorithm A fails to construct a schedule for τ on m processors, then there is no feasible schedule for τ on m processors. In the remainder of the proof we use the same variables with the same meanings as in Algorithm Reschedule. Let $t_1 < t_2 < \ldots < t_k$ be the sequence of times at which new tasks arrive and t_k be the first instant at which Algorithm Reschedule fails to construct a schedule. Then, by the algorithm, we have either $E >$ *capacity* or *longest* $>$ *interval*. We claim that the case *longest* $>$ *interval* cannot hold. Let T be the task with the largest remaining execution time at t_k. Clearly, T cannot be one of the tasks that arrive at t_k. For otherwise, τ cannot have any feasible schedules. On the other hand, T cannot be one of the tasks that arrived prior to t_k either, since this would imply that Algorithm Reschedule failed prior to t_k. Therefore, the case *longest* $>$ *interval* cannot hold and hence Algorithm Reschedule fails to construct a schedule because $E >$ *capacity*. Thus, the total remaining execution time of all active tasks at t_k is larger than $m(d - t_k)$.

Let S be the schedule constructed by Algorithm A in the interval (t_1, t_k). If S has no idle processor time, then it is clear that τ cannot have any feasible schedules on m processors. Thus, we assume that S has some idle processor times and let a be the largest index such that S has some idle processor times in the interval (t_a, t_{a+1}). For each $1 \leq i < k$, let S_i denote the schedule constructed by Algorithm Reschedule at t_i, and $S[t_i, t_{i+1}]$ and $S_i[t_i, t_{i+1}]$ denote the segments of S and S_i in the interval $[t_i, t_{i+1}]$, respectively. Clearly, for each $1 \leq i < k$, we have $S[t_i, t_{i+1}] = S_i[t_i, t_{i+1}]$. Since $S[t_a, t_{a+1}]$ has some idle processor times, $S_a[t_a, t_{a+1}]$ must also have some idle processor times. Suppose that every processor in S_a finishes by t_{a+1}. Then, every task that arrives prior to t_{a+1} has finished execution by t_{a+1} in S, and hence any tasks executed at or after t_{a+1} in S must have arrived at or after t_{a+1}. Since $S[t_{a+1}, t_k]$ has no idle processor time and since $E >$ *capacity* at t_k, the total amount of tasks that arrive at or after t_{a+1} is larger than $m(d - t_{a+1})$. Thus, τ is not feasible on m processors.

From the above discussions, we may assume that some processors in S_a finish later than t_{a+1} and some processors finish earlier than t_{a+1}. By Lemma 1, the processors that finish later than t_{a+1} must have only one task assigned to each of them, starting at t_a. Let X_a be the set of tasks assigned to the processors that finish later than t_{a+1} in S_a. If we can show that each task in X_a is executed continuously in S since its arrival, then it is clear that we cannot execute more of the tasks in X_a by t_{a+1} than that in S. Since $S[t_{a+1}, t_k]$ has no idle processor time and since $E >$ *capacity* at t_k, τ cannot be feasible on m processors. Thus, all we need to show is that each task in X_a is executed continuously in S since its arrival. If each task in X_a arrives at t_a, then we are done. Otherwise, let $Y_a \subseteq X_a$ be the set of tasks that arrive prior to t_a and n_a be the number of tasks in Y_a. Clearly, we have $n_a < m$. Let Z_a be the set of tasks that arrive prior to t_a, have not finished execution by t_a in S, but finished by t_{a+1}. Clearly, the tasks in Z_a can be executed on at most $m - n_a$ processors in $S[t_a, t_{a+1}]$. Since $S[t_a, t_{a+1}]$ contains some idle processor times, the total remaining execution time of the tasks in Z_a at t_a must be less than $(m - n_a)(t_{a+1} - t_a)$. On the other hand, the remaining execution time of each task in Y_a at t_a is larger than $t_{a+1} - t_a$. A moment of reflection shows that each task in Y_a must be executed continuously in $S_{a-1}[t_{a-1}, t_a]$, starting at t_{a-1}. If all tasks in Y_a arrive at t_{a-1}, then we are done. Otherwise, we repeat the above argument to show that the tasks which arrive prior to t_{a-1} must be executed continuously in S since its arrival. □

3. Urgent Tasks.

In this section we consider environments where processors can go down unexpectedly. We model this situation by adding urgent tasks to the task system. An urgent task T has the characteristic that $d(T) = r(T) + e(T)$. It is assumed that non-urgent tasks have the same deadline while urgent tasks can have arbitrary deadlines. It is easy to see that Algorithm A given in the previous section is not optimal for this type of task systems. Consider the task system $\tau = (\{T_1, T_2, T_3, T_4\}, \{0, 0, 0, 2\}, \{5, 5, 4, 4\}, \{2, 2, 4, 2\})$ to be scheduled on $P = \{P_1, P_2\}$. Observe that T_3 and T_4 are urgent tasks while T_1 and T_2 are non-urgent tasks. At time 0, Algorithm A will schedule T_1, T_2 and T_3 in the intervals (0, 2), (2, 4) and (0, 4), respectively. At time 2, Algorithm A will not be able to schedule the remaining tasks so that all deadlines are met. However, τ is feasible on two processors.

In the following we give an optimal on-line scheduler, to be called Algorithm B, for this type of task systems and any $m \geq 1$. Algorithm B employs a subroutine, Algorithm Slack-Time, to reschedule tasks whenever new tasks arrive. Before we introduce Algorithm Slack-Time, we need to define the following notations. If e denotes the remaining execution time of an active task T at time t, then the slack-time of T at t is defined to be $d(T) - t - e$. Thus, the slack-time of an urgent task is 0 at its release time. Algorithm Slack-Time schedules tasks with the smallest slack-time first, one task per processor. If several tasks are tied for the same slack-time and there are not enough processors to be assigned to the tasks, then they share the remaining processors equally.

Algorithm Slack-Time

1. Let U be the set of active tasks at the current time t. For each task T in U, compute the slack-time of T at t. If there is a task with negative slack-time or the number of tasks with zero slack-time is larger than m, then print "infeasible" and stop. Otherwise, assign the tasks with zero slack-time to the highest indexed processors, with the longest task assigned first. Then, the tasks with positive slack-time are assigned to the lowest indexed processors, with the smallest slack-time task assigned first. If several tasks are tied for the same slack-time and there are not enough processors to be assigned to the tasks, then they share the remaining processors equally. Repeat this assignment until all processors or all active tasks have been assigned, whichever comes first.
2. Reassign all processors as above to the unexecuted portion of the active tasks whenever one of the following two events occurs : (1) A task is finished. (2) We reach a point where, if we were to continue with the present assignment, we would be executing a task with a larger slack-time at a faster rate than other tasks with a smaller slack-time.
3. If the constructed schedule has a missed deadline, then print "infeasible" and stop.

Figure 2(a) shows the schedule constructed by Algorithm Slack-Time at time 0 for the task system given in Figure 2. Observe that T_4 and T_7 are the only urgent tasks in the task system. At time 0, the urgent task T_4 is assigned to P_3, and T_1 and T_2 are assigned to P_1 and P_2, respectively. At time 1, T_1 has the smallest slack-time while T_2 and T_3 have the same slack-time. Thus, T_1 is assigned to one processor while T_2 and T_3 share one processor equally. At time 3, T_1, T_2 and T_3 have the same slack-time. Thus, they share two processors equally. At time 4, T_4 finishes execution, and hence T_1, T_2 and T_3 are reassigned to P_1, P_2 and P_3, respectively. We will prove in the next theorem some important properties of the schedules constructed by Algorithm Slack-Time. These properties are needed in showing that Algorithm B is an optimal on-line scheduler. First, we need the following lemma.

Lemma 2 : Let $c_1 \geq c_2 \geq \dots \geq c_n$ be n nonnegative real numbers. If $a \leq (\sum_{i=1}^{n} c_i)/n$, then we have $ak \leq \sum_{i=1}^{k} c_i$ for each $1 \leq k \leq n$.

Proof : Assume the lemma is false and let k be the smallest integer such that $ak > \sum_{i=1}^{k} c_i$. Since $a(k-1) \leq \sum_{i=1}^{k-1} c_i$, we have $a > c_k$. Since $c_k \geq c_{k+1} \geq \dots \geq c_n$, we have $a(n-k) > \sum_{i=k+1}^{n} c_i$. Thus, $\sum_{i=1}^{n} c_i = \sum_{i=1}^{k} c_i + \sum_{i=k+1}^{n} c_i < ak + a(n-k) = an$. This contradicts our assumption that $an \leq \sum_{i=1}^{n} c_i$. □

Theorem 3 : Let TS be a set of tasks released at time 0 and $TS' = \{T_1, T_2, \dots, T_n\}$ be the set of non-urgent tasks in TS. Let S^* be the schedule constructed by Algorithm Slack-Time for TS on the processor system P and S^+ be any feasible schedule for TS on P. For any given time t, let $a_1(t) \geq a_2(t) \geq \dots \geq a_n(t)$ and $b_1(t) \geq b_2(t) \geq \dots \geq b_n(t)$ be the remaining execution times of the tasks in TS' at t in S^* and S^+, respectively. Then, for each $t \geq 0$, we have $\sum_{i=1}^{k} a_i(t) \leq \sum_{i=1}^{k} b_i(t)$ for each $1 \leq k \leq n$.

Proof : Without loss of generality, we may assume that the urgent tasks are scheduled on the same processors in both S^* and S^+. For each time t, let $\alpha(T_i, t)$ and $\beta(T_i, t)$ denote the remaining execution times of T_i at t in S^* and S^+, respectively. Let the non-urgent tasks be reindexed such that for each $1 \leq i < n$, the slack-time of T_i at t in S^* is no larger than that of T_{i+1}. Since the non-urgent tasks have one common deadline, we have $a_i(t) = \alpha(T_i, t)$ for each $1 \leq i \leq n$. Let $R \subseteq TS'$ be the set of non-urgent tasks executed at time t in S^* and r be the number of tasks in R. By the nature of Algorithm Slack-Time, $R = \{T_1, T_2, \dots, T_r\}$. Let R be partitioned into $R_1, R_2, \dots, R_x$ such that R_1 is the set of tasks with the smallest slack-time, R_2 is the set of tasks with the second smallest slack-time, and so on. Note that each task in R, except possibly the tasks in R_x, is executed on one processor at t in S^*. For each $1 \leq i \leq x$, let $C_i = R_1 \cup R_2 \cup \dots \cup R_i$ and r_i be the number of tasks in C_i. Hence, we have $C_x = R$ and $r_x = r$. Letting $r_0 = 0$, we have $R_i = \{T_{r_{i-1}+1}, T_{r_{i-1}+2}, \dots, T_{r_i}\}$ for each $1 \leq i \leq x$. Observe that for each $1 \leq i \leq x$, we have $\alpha(T_{r_{i-1}+1}, t) = \alpha(T_{r_{i-1}+2}, t) = \dots = \alpha(T_{r_i}, t) = a_{r_{i-1}+1}(t) = a_{r_{i-1}+2}(t) = \dots = a_{r_i}(t)$. Thus, we may reindex the tasks in each R_i such that $\beta(T_{r_{i-1}+1}, t) \geq \beta(T_{r_{i-1}+2}, t) \geq \dots \geq \beta(T_{r_i}, t)$.

For each $1 \leq i \leq m$, let $p_i(t)$ denote the total amount of non-urgent tasks executed on P_i in the interval $(0, t)$ in S^*. By the nature of Algorithm Slack-Time, we have $p_1(t) \geq p_2(t) \geq \dots \geq p_m(t)$ for each $t \geq 0$. By induction on j, we show that for each $1 \leq k \leq r_j$, $\sum_{i=1}^{k} \alpha(T_i, t) \leq \sum_{i=1}^{k} \beta(T_i, t)$. As the basis case, we consider $j = 1$. Since the tasks in R_1 have the smallest slack-time, the tasks in R_1 and the urgent tasks are the only tasks that can be scheduled on the first r_1 processors in the interval $(0, t)$ in S^*. Furthermore, there are no idle processor time on the first r_1 processors in the interval $(0, t)$ in S^*. Thus, we have $\sum_{i=1}^{r_1} \alpha(T_i, t) \leq \sum_{i=1}^{r_1} \beta(T_i, t)$. Since $\alpha(T_1, t) = \alpha(T_2, t) = \dots = \alpha(T_{r_1}, t)$, we have $\alpha(T_i, t) \leq \sum_{j=1}^{r_1} \beta(T_j, t)/r_1$ for each $1 \leq i \leq r_1$. Thus, by Lemma 2, we have for each $1 \leq k \leq r_1$, $\sum_{i=1}^{k} \alpha(T_i, t) \leq \sum_{i=1}^{k} \beta(T_i, t)$.

For the inductive step, we assume that the claim is true for all $j < j'$, we want to show that it is true for $j = j'$. Since the tasks in $C_{j'}$ have smaller slack-time than the tasks in $TS' - C_{j'}$, it is clear that the tasks in $C_{j'}$ and the urgent tasks are the only tasks that can be scheduled on the first $min\{m, r_{j'}\}$ processors in the interval $(0, t)$ in S^*. Thus, we have $\sum_{i=1}^{r_{j'}} \alpha(T_i, t) \leq \sum_{i=1}^{r_{j'}} \beta(T_i, t)$. If the claim is not true for $j = j'$, then let l be the smallest index such that $\sum_{i=1}^{l} \alpha(T_i, t) > \sum_{i=1}^{l} \beta(T_i, t)$. By the induction hypothesis, we must have $r_{j'-1} < l \leq r_{j'}$, and hence $\{T_l, T_{l+1}, \dots, T_{r_{j'}}\}$ is a subset of $R_{j'}$. Therefore, we have $\alpha(T_l, t) = \alpha(T_{l+1}, t) = \dots = \alpha(T_{r_{j'}}, t) > \beta(T_l, t) \geq \beta(T_{l+1}, t) \geq \dots \geq \beta(T_{r_{j'}}, t)$, and hence $\sum_{i=l+1}^{r_{j'}} \alpha(T_i, t) > \sum_{i=l+1}^{r_{j'}} \beta(T_i, t)$. Thus, we have $\sum_{i=1}^{r_{j'}} \alpha(T_i, t) = \sum_{i=1}^{l} \alpha(T_i, t) + \sum_{i=l+1}^{r_{j'}} \alpha(T_i, t) > \sum_{i=1}^{l} \beta(T_i, t) + \sum_{i=l+1}^{r_{j'}} \beta(T_i, t) = \sum_{i=1}^{r_{j'}} \beta(T_i, t)$. This is the contradiction we sought. Thus, for any $1 \leq k \leq r_{j'}$, we have $\sum_{i=1}^{k} \alpha(T_i, t) \leq \sum_{i=1}^{k} \beta(T_i, t)$. By induction, we have shown that for each $1 \leq k \leq r$, $\sum_{i=1}^{k} \alpha(T_i, t) \leq \sum_{i=1}^{k} \beta(T_i, t)$.

Let $Q = TS' - R$; i.e., $Q = \{T_{r+1}, T_{r+2}, \dots, T_n\}$. Then, either all tasks in Q have finished by t or all tasks in Q have not started by t in S^*. In the former case, we have $\alpha(T_i, t) = 0 \leq \beta(T_i, t)$ for each $T_i \in Q$. Thus, for each $r < k \leq n$, we have $\sum_{i=1}^{k} \alpha(T_i, t) = \sum_{i=1}^{r} \alpha(T_i, t) \leq \sum_{i=1}^{r} \beta(T_i, t) \leq \sum_{i=1}^{k} \beta(T_i, t)$. In the latter case, we have $\sum_{i=1}^{r} \alpha(T_i, t) = \sum_{i=1}^{r} e(T_i, t) - \sum_{i=1}^{m} p_i(t)$. Thus, we have $\sum_{i=1}^{k} \alpha(T_i, t) = \sum_{i=1}^{k} e(T_i, t) - \sum_{i=1}^{m} p_i(t)$ for each $r < k \leq n$. Since $\sum_{i=1}^{k} \beta(T_i, t) \geq \sum_{i=1}^{k} e(T_i, t) - \sum_{i=1}^{m} p_i(t)$, we have $\sum_{i=1}^{k} \alpha(T_i, t) \leq \sum_{i=1}^{k} \beta(T_i, t)$ for each $r < k \leq n$. In both cases, we have $\sum_{i=1}^{k} \alpha(T_i, t) \leq \sum_{i=1}^{k} \beta(T_i, t)$ for each $r < k \leq n$. Combining this with the previous result, we have $\sum_{i=1}^{k} \alpha(T_i, t) \leq \sum_{i=1}^{k} \beta(T_i, t)$ for each $1 \leq k \leq n$. Thus, we have $\sum_{i=1}^{k} a_i(t) = \sum_{i=1}^{k} \alpha(T_i, t) \leq \sum_{i=1}^{k} \beta(T_i, t) \leq \sum_{i=1}^{k} b_i(t)$ for each $1 \leq k \leq n$. □

Theorem 4 : Let τ be a task system such that the urgent tasks have arbitrary deadlines and the non-urgent tasks have one common deadline. Suppose τ has one distinct release time. Then, τ is feasible on m processors if and only if Algorithm B constructs a feasible schedule for τ on m processors.

Proof : If Algorithm B constructs a feasible schedule for τ on m processors, then τ is clearly feasible on m processors. To complete the proof, we show that if τ is feasible on m processors, then Algorithm B constructs a feasible schedule for τ on m processors. Let S^+ be any feasible schedule for τ on m processors. Since τ has only one distinct release time, Algorithm Slack-Time will be called exactly once by Algorithm B. Let S^* be the schedule constructed by Algorithm Slack-Time for τ on m processors. Without loss of generality, we may assume that the urgent tasks are scheduled on the same processors in both S^* and S^+. Let there be n non-urgent tasks in τ and d be their common deadline. Let $a_1(d) \geq a_2(d) \geq \dots \geq a_n(d)$ and $b_1(d) \geq b_2(d) \geq \dots \geq b_n(d)$ be the remaining execution times of the non-urgent tasks at time d in

S^* and S^+, respectively. Then, by Theorem 3, we have $\sum_{i=1}^{k} a_i(d) \leq \sum_{i=1}^{k} b_i(d)$ for each $1 \leq k \leq n$. Since S^+ is a feasible schedule, we have $b_i(d) = 0$ for each $1 \leq i \leq n$. Thus, $a_i(d) = 0$ for each $1 \leq i \leq n$. Hence, S^* is a feasible schedule for τ. □

Algorithm B is given below. It simply calls Algorithm Slack-Time whenever new tasks arrive.

Algorithm B

```
Whenever new tasks arrive, do
{    $t \leftarrow$ the current time;
     $U \leftarrow$ the set of active tasks at $t$;
     Call Algorithm Slack-Time to schedule the tasks in $U$;
}
```

Figure 2(b) shows the final schedule constructed by Algorithm B for the task system given in Figure 2. To prove that Algorithm B is an optimal on-line scheduler, we need to show that certain tasks can be scheduled in the idle intervals in a partial schedule. It will be more convenient to have some kind of canonical (partial) schedules. Let S be a partial schedule. S is said to be ***monotone*** if P_i is idle during the interval (a, b) implies that P_j is idle in the same interval for each $1 \leq j < i$. Figure 3(b) shows a monotone schedule. Note that if S is a monotone schedule and $p_i(S)$ denotes the total idle time of P_i in S, then $p_i(S) \geq p_{i+1}(S)$ for each $1 \leq i < m$. The next lemma shows that every schedule S can be converted into an equivalent monotone schedule S'.

Lemma 3 : For every schedule S, there is an equivalent monotone schedule S'.

Proof : Let $t_1 < t_2 < \ldots < t_k$ be the sequence of times at which a processor becomes idle or busy in S. For each interval (t_i, t_{i+1}), let m_i be the number of processors that are idle in S. We obtain S' from S as follows. For each interval (t_i, t_{i+1}), we reschedule the tasks in the interval so that the first m_k processors are idle and the remaining processors are busy. It is easy to see that S' is a monotone schedule. □

Figure 3(b) shows the monotone schedule obtained from the schedule given in Figure 3(a). Our next goal is to derive a neccessary and sufficient condition for a set of tasks to be schedulable in the idle intervals in a monotone schedule.

Theorem 5 : Let $TS = \{T_i\}$ be a set of n independent tasks such that $e(T_i) \geq e(T_{i+1})$ for each $1 \leq i < n$. Let S be a monotone schedule. For each $1 \leq i \leq m$, let $p_i(S)$ denote the total idle time of P_i in S. Then, TS can be scheduled in the idle intervals in S if and only if the following two conditions hold.

1. For each $1 \leq k < m$, $\sum_{i=1}^{k} e(T_i) \leq \sum_{i=1}^{k} p_i(S)$.
2. $\sum_{i=1}^{n} e(T_i) \leq \sum_{i=1}^{m} p_i(S)$.

Proof : Since S is a monotone schedule, we have $p_i(S) \geq p_{i+1}(S)$ for each $1 \leq i < m$. If TS can be scheduled in the idle intervals in S, then condition (2) clearly holds. Condition (1) also holds since we can always transform the schedule for TS into one such that the longest k tasks are scheduled in the idle intervals of the first k processors in S. We now show that if both conditions (1) and (2) hold, then TS can be scheduled in the idle intervals in S. The proof is by induction on the number of processors, m. The basis case, $m = 1$, is clearly true. Assume the claim is true for all $m < m'$, we now show that the claim is true for $m = m'$.

We schedule on $P_{m'}$ the tasks $T_n, T_{n-1}, \ldots, T_1$ in that order until we first encounter a task $T_{n'}$ such that $e(T_{n'})$ is larger than or equal to the remaining idle time on $P_{m'}$. Then, we schedule $T_{n'}$ as follows. If there is a P_a such that $e(T_{n'}) = p_a(S)$, then we schedule $T_{n'}$ completely on P_a and remove P_a from consideration. Otherwise, we can find an index a such that $p_a(S) > e(T_{n'}) > p_{a+1}(S)$. We schedule $p_{a+1}(S)$ amount of $T_{n'}$ on P_{a+1} and the remaining amount of $T_{n'}$ on P_a. This is done in such a way that the resulting schedule is a monotone schedule; see [2] on how this can be done. We then remove P_{a+1} from consideration. We need to show that the remaining $n' - 1$ tasks can be scheduled on the remaining $m' - 1$ processors. Let the processors be relabelled as $Q_1, Q_2, \ldots, Q_{m'-1}$ such that $q_1(S) \geq q_2(S) \geq \ldots \geq q_{m'-1}(S)$, where $q_i(S)$ denotes the total remaining idle time of Q_i. If $T_{n'}$ is completely scheduled on P_a, then $\{Q_1, Q_2, \ldots, Q_{m'-1}\} = \{P_1, \ldots, P_{a-1}, P_{a+1}, \ldots, P_{m'}\}$; otherwise, $\{Q_1, Q_2, \ldots, Q_{m'-1}\} = \{P_1, \ldots, P_a, P_{a+2}, \ldots, P_{m'}\}$. Clearly, we have $\sum_{i=1}^{n'-1} e(T_i) \leq \sum_{i=1}^{m'-1} q_i(S)$. If we can show that

$$\sum_{i=1}^{k} e(T_i) \leq \sum_{i=1}^{k} q_i(S) \qquad (*)$$

for each $1 \leq k < m' - 1$, then the $n' - 1$ tasks can be scheduled on the $m' - 1$ processors by the induction hypothesis. For each $1 \leq k \leq a - 1$, (*) clearly holds. It remains to be shown that (*) holds for each $a \leq k < m' - 1$.

If $T_{n'}$ is completely scheduled on P_a, then we have $q_i(S) = p_i(S)$ for each $1 \leq i < a$ and $q_i(S) = p_{i+1}(S)$ for each $a \leq i < m'$. Thus, for each $a \leq k < m' - 1$, we have $\sum_{i=1}^{k} q_i(S) = \sum_{i=1}^{a-1} q_i(S) + \sum_{i=a}^{k} q_i(S) = \sum_{i=1}^{a-1} p_i(S) + \sum_{i=a+1}^{k+1} p_i(S) = \sum_{i=1}^{k+1} p_i(S) - p_a(S) = \sum_{i=1}^{k+1} p_i(S) - e(T_{n'})$. On the other hand, if $T_{n'}$ is partially scheduled on P_a, then we have $q_i(S) = p_i(S)$ for each $1 \leq i < a$, $q_a(S) = p_a(S) - (e(T_{n'}) - p_{a+1}(S))$ and $q_i(S) = p_{i+1}(S)$ for each $a < i < m'$. Thus, for each $a \leq k < m' - 1$, we have $\sum_{i=1}^{k} q_i(S) = \sum_{i=1}^{a-1} q_i(S) + q_a(S) + \sum_{i=a+1}^{k} q_i(S) = \sum_{i=1}^{a-1} p_i(S) + p_a(S) - (e(T_{n'}) - p_{a+1}(S)) + \sum_{i=a+2}^{k+1} p_i(S) = \sum_{i=1}^{k+1} p_i(S) - e(T_{n'})$. Thus, we have in both cases $\sum_{i=1}^{k} q_i(S) = \sum_{i=1}^{k+1} p_i(S) - e(T_{n'})$. Since $e(T_{n'}) \leq e(T_{k+1})$, we have $\sum_{i=1}^{k} e(T_i) + e(T_{n'}) \leq \sum_{i=1}^{k+1} e(T_i) \leq \sum_{i=1}^{k+1} p_i(S) \leq \sum_{i=1}^{k} q_i(S) + e(T_{n'})$. Hence, $\sum_{i=1}^{k} e(T_i) \leq \sum_{i=1}^{k} q_i(S)$. □

Using Theorem 5, we can show that Algorithm B is an optimal on-line scheduler for any $m \geq 1$. This result is given in the next theorem.

Theorem 6 : Let τ be a task system such that the non-urgent tasks have one common deadline and the urgent tasks have arbitrary deadlines. Then, τ is feasible on $m \geq 1$ processors if and only if Algorithm B constructs a feasible schedule for τ on m processors.

Proof : If Algorithm B constructs a feasible schedule for τ on m processors, then τ is clearly feasible on m processors. By induction on the number of distinct release times in τ, we show that Algorithm B constructs a feasible schedule for τ on m processors if τ is feasible on m processors. If τ has only one distinct release time, then the claim follows immediately from Theorem 4. Assuming that the claim holds for all task systems with less than r distinct release times, we want to show that it holds for all task systems with r distinct release times. Let τ be a task system with r distinct release times. Let S^+ and $\hat{S}$ be a feasible schedule and the schedule constructed by Algorithm B for τ on m processors, respectively. Without loss of generality, we may assume that the urgent tasks are scheduled on the same processors in both S^+ and $\hat{S}$. Thus, the non-urgent tasks are the only tasks that can be scheduled differently in the two schedules. If S is a schedule, we let $S[t_1, t_2]$ denote the subschedule of S in the interval $[t_1, t_2]$.

Let the first and second release times in τ be 0 and t', respectively, and let Q and R be the sets of urgent and non-urgent tasks released at time 0, respectively. Let S^* be the schedule constructed by Algorithm Slack-Time for the tasks in $Q \cup R$ at time 0. Clearly, we have $\hat{S}[0, t'] = S^*[0, t']$. Let $a_1(t') \geq a_2(t') \geq \ldots \geq a_n(t')$ and $b_1(t') \geq b_2(t') \geq \ldots \geq b_n(t')$ be the remaining execution times of the n tasks in R at t' in S^* and S^+, respectively. Then, by Theorem 3, we have for each $1 \leq k \leq n$

$$\sum_{i=1}^{k} a_i(t') \leq \sum_{i=1}^{k} b_i(t') \qquad (*)$$

Let S' be the schedule obtained by removing the tasks in R from the subschedule $S^+[t', d]$, where d is the common deadline of the non-urgent tasks. Let S_1 be the monotone schedule equivalent to S' and $p_i(S_1)$ be the total idle time of P_i in S_1. By Theorem 5, we have

$$\sum_{i=1}^{k} b_i(t') \leq \sum_{i=1}^{k} p_i(S_1) \quad \text{for each } 1 \leq k < m \qquad (1)$$

and

$$\sum_{i=1}^{n} b_i(t') \leq \sum_{i=1}^{m} p_i(S_1). \qquad (2)$$

By (*), (1), (2) and Theorem 5, we can schedule a set of n independent tasks with execution times $a_1(t') \geq a_2(t') \geq \ldots \geq a_n(t')$ in the idle intervals in S_1. Thus, there is a feasible schedule for these n tasks and the remaining tasks in τ. By the induction hypothesis, Algorithm B constructs a feasible schedule for these n tasks and the tasks released after time 0. Thus, $\hat{S}$ is a feasible schedule. □

4. Conclusions.

In this paper we have shown that there can be no optimal on-line scheduler for any $m > 1$ when the task systems have two or more distinct deadlines. We then give an optimal on-line scheduler for any $m > 1$ when the task systems have only one common deadline. Our results give a sharp delineation of the theoretical limit of having an optimal on-line scheduler. We also consider environments where the processors can go down unexpectedly, and an optimal on-line scheduler is also given in this case. We note that Algorithm B can be modified to yield an optimal on-line scheduler for the situation where the durations of processor down times are unknown. However, the modified algorithm will not be able to detect infeasible task systems until it reaches the common deadline of the non-urgent tasks. Algorithm B is also optimal for task systems without urgent tasks (and hence no processor down time). However, Algorithm A is more desirable in that case since it is faster and does not involve processor-sharing.

For future research, it is desirable to study whether optimal on-line scheduler exists for uniform processor systems. The study must necessarily be restricted to task systems with one common deadline, since our result implies that no such algorithm exists for task systems with two or more distinct deadlines. We note that Sahni and Cho [9] have given an optimal nearly on-line scheduler for this case. Another direction that is worthwhile to pursue is to derive other classes of task systems for which an optimal on-line scheduler exists.

References

1. Dertouzos, M., "Control Robotics: The Procedural Control of Physical Processes," *Proc. of the IFIP Congress,* 1974, pp 807-813.
2. Hong, K. S. and J. Y-T. Leung, "Preemptive Scheduling with Release Times and Deadlines," *Proc. of the 1988 Conference on Information Sciences and Systems,* Princeton, N.J., March 1988.
3. Horn, W., "Some Simple Scheduling Algorithms," *Naval Res. Logist. Quart. 21,* 1974, pp 177-185.
4. Labètoulle, J., "Some Theorems on Real Time Scheduling," in E. Gelenbe and R. Mahl, Eds., *Computer Architecture and Networks,* North Holland, Amsterdam, 1974, pp 285-293.
5. Liu, C. L. and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM. 20,* 1973, pp 46-61.
6. McNaughton, R., "Scheduling with Deadlines and Loss Functions," *Management Science 12,* 1959, pp 1-12.
7. Mok, A. K., *Fundamental Design Problems of Distributed Systems for the Hard Real Time Environment,* Ph. D. Dissertation, M.I.T., 1983.
8. Sahni, S., "Preemptive Scheduling with Due Dates," *Oper. Res. 27,* 1979, pp 925-934.
9. Sahni S. and Y. Cho, "Nearly On Line Scheduling of a Uniform Processor System with Release Times," *SIAM J. on Computing 8,* 1979, pp 275-285.

T_i	T_1	T_2	T_3	T_4	T_5	T_6
$r(T_i)$	0	0	0	0	3	3
$d(T_i)$	10	10	10	10	10	10
$e(T_i)$	6	3	3	2	5	3

$m = 3$

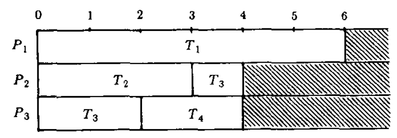

Figure 1(a). The Schedule at Time 0.

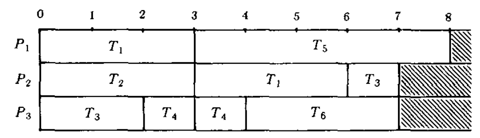

Figure 1(b). The Schedule at Time 3.

Figure 1. An Example Illustrating Algorithm A.

T_i	T_1	T_2	T_3	T_4	T_5	T_6	T_7
$r(T_i)$	0	0	0	0	3	3	3
$d(T_i)$	10	10	10	4	10	10	6
$e(T_i)$	5	4	3	4	4	5	3

$m = 3$

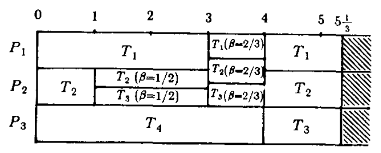

Figure 2(a). The Schedule at Time 0.

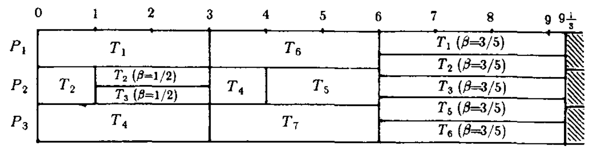

Figure 2(b). The Schedule at Time 3.

Figure 2. An Example Illustrating Algorithm B.

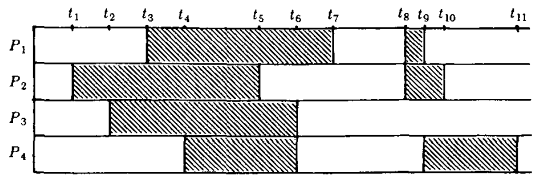
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11}
P_1
P_2
P_3
P_4

Figure 3(a). An Arbitrary Schedule S.

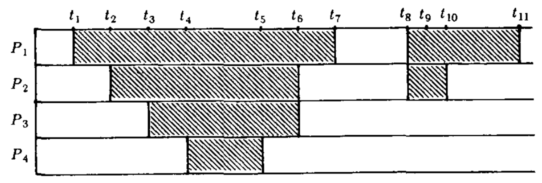
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11}
P_1
P_2
P_3
P_4

Figure 3(b). The Monotone Schedule S' Obtained from S.

Figure 3. An Example Monotone Schedule.