Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment

C. L. LIU

Project MAC, Massachusetts Institute of Technology

AND

JAMES W. LAYLAND

Jet Propulsion Laboratory, California Institute of Technology

ABSTRACT. The problem of multiprogram scheduling on a single processor is studied from the viewpoint of the characteristics peculiar to the program functions that need guaranteed service. It is shown that an optimum fixed priority scheduler possesses an upper bound to processor utilization which may be as low as 70 percent for large task sets. It is also shown that full processor utilization can be achieved by dynamically assigning priorities on the basis of their current deadlines. A combination of these two scheduling techniques is also discussed.

KEY WORDS AND PHRASES: real-time multiprogramming, scheduling, multiprogram scheduling, dynamic scheduling, priority assignment, processor utilization, deadline driven scheduling

CR CATEGORIES: 3.80, 3.82, 3.83, 4.32

1. *Introduction*

The use of computers for control and monitoring of industrial processes has expanded greatly in recent years, and will probably expand even more dramatically in the near future. Often, the computer used in such an application is shared between a certain number of time-critical control and monitor functions and a non-time-critical batch processing job stream. In other installations, however, no non-time-critical jobs exist, and efficient use of the computer can only be achieved by a careful scheduling of the time-critical control and monitor functions themselves. This latter group might be termed "pure process control" and provides the background for the combinatoric scheduling analyses presented in this paper. Two

This paper presents the results of one phase of research carried out at the Jet Propulsion Laboratory, California Institute of Technology, under Contract No. NAS-7-100, sponsored by the National Aeronautics and Space Administration.

Authors' present addresses: C. L. Liu, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801; James W. Layland, Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91103.

scheduling algorithms for this type of programming are studied; both are priority driven and preemptive; meaning that the processing of any task is interrupted by a request for any higher priority task. The first algorithm studied uses a fixed priority assignment and can achieve processor utilization on the order of 70 percent or more. The second scheduling algorithm can achieve full processor utilization by means of a dynamic assignment of priorities. A combination of these two algorithms is also discussed.

2. *Background*

A process control computer performs one or more control and monitoring functions. The pointing of an antenna to track a spacecraft in its orbit is one example of such functions. Each function to be performed has associated with it a set of one or more *tasks*. Some of these tasks are executed in response to events in the equipment controlled or monitored by the computer. The remainder are executed in response to events in other tasks. None of the tasks may be executed before the event which requests it occurs. Each of the tasks must be completed before some fixed time has elapsed following the request for it. Service within this span of time must be guaranteed, categorizing the environment as "hard-real-time" [1] in contrast to "soft-real-time" where a statistical distribution of response times is acceptable.

Much of the available literature on multiprogramming deals with the statistical analysis of commercial time-sharing systems ([2] contains an extensive bibliography). Another subset deals with the more interesting aspects of scheduling a batch-processing facility or a mixed batch-time-sharing facility, usually in a multiple processor configuration [3–8]. A few papers directly attack the problems of "hard-real-time" programming. Manacher [1] derives an algorithm for the generation of task schedules in a hard-real-time environment, but his results are restricted to the somewhat unrealistic situation of only one request time for all tasks, even though multiple deadlines are considered. Lampson [9] discusses the software scheduling problem in general terms and presents a set of ALGOL multiprogramming procedures which could be software-implemented or designed into a special purpose scheduler. For the allocation of resources and for the assignment of priorities and time slots, he proposes a program which computes estimated response time distributions based on the timing information supplied for programs needing guaranteed service. He does not, however, describe the algorithms which such a program must use.

The text by Martin [10] depicts the range of systems which are considered to be "real-time" and discusses in an orderly fashion the problems which are encountered in programming them. Martin's description of the tight engineering management control that must be maintained over real-time software development is emphatically echoed in a paper by Jirauch [11] on automatic checkout system software. These discussions serve to emphasize the need for a more systematic approach to software design than is currently in use.

3. *The Environment*

To obtain any analytical results about program behavior in a hard-real-time environment, certain assumptions must be made about that environment. Not all of

these assumptions are absolutely necessary, and the effects of relaxing them will be discussed in a later section.

(A1) The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.

(A2) Deadlines consist of run-ability constraints only—i.e. each task must be completed before the next request for it occurs.

(A3) The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks.

(A4) Run-time for each task is constant for that task and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption.

(A5) Any nonperiodic tasks in the system are special; they are initialization or failure-recovery routines; they displace periodic tasks while they themselves are being run, and do not themselves have hard, critical deadlines.

Assumption (A1) contrasts with the opinion of Martin [2], but appears to be valid for pure process control. Assumption (A2) eliminates queuing problems for the individual tasks. For assumption (A2) to hold, a small but possibly significant amount of buffering hardware must exist for each peripheral function. Any control loops closed within the computer must be designed to allow at least an extra unit sample delay. Note that assumption (A3) does not exclude the situation in which the occurrence of a task τ_j can only follow a certain (fixed) number, say N, of occurrences of a task τ_i. Such a situation can be modeled by choosing the periods of tasks τ_i and τ_j so that the period of τ_j is N times the period of τ_i and the Nth request for τ_i will coincide with the 1st request for τ_j and so on. The run-time in assumption (A4) can be interpreted as the maximum processing time for a task. In this way the bookkeeping time necessary to request a successor and the costs of preemptions can be taken into account. Because of the existence of large main memories out of which programs are executed and the overlapping of transfers between main and auxiliary storage and program execution in modern computer systems, assumption (A4) should be a good approximation even if it is not exact. These assumptions allow the complete characterization of a task by two numbers: its request period and its run-time. Unless stated otherwise, throughout this paper we shall use $\tau_1, \tau_2, \cdots, \tau_m$ to denote m periodic tasks, with their request periods being $T_1, T_2, \cdots, T_m$ and their run-times being $C_1, C_2, \cdots, C_m$, respectively. The *request rate* of a task is defined to be the reciprocal of its request period.

A scheduling algorithm is a set of rules that determine the task to be executed at a particular moment. The scheduling algorithms to be studied in this paper are preemptive and *priority driven* ones. This means that whenever there is a request for a task that is of higher priority than the one currently being executed, the running task is immediately interrupted and the newly requested task is started. Thus the specification of such algorithms amounts to the specification of the method of assigning priorities to tasks. A scheduling algorithm is said to be *static* if priorities are assigned to tasks once and for all. A static scheduling algorithm is also called a *fixed* priority scheduling algorithm. A scheduling algorithm is said to be *dynamic* if priorities of tasks might change from request to request. A scheduling algorithm is said to be a *mixed scheduling algorithm* if the priorities of some of the tasks are fixed yet the priorities of the remaining tasks vary from request to request.

4. *A Fixed Priority Scheduling Algorithm*

In this section we derive a rule for priority assignment that yields an optimum static scheduling algorithm. An important concept in determining this rule is that of the *critical instant* for a task. The *deadline* of a request for a task is defined to be the time of the next request for the same task. For a set of tasks scheduled according to some scheduling algorithm, we say that an *overflow* occurs at time t if t is the deadline of an unfulfilled request. For a given set of tasks, a scheduling algorithm is *feasible* if the tasks are scheduled so that no overflow ever occurs. We define the *response time* of a request for a certain task to be the time span between the request and the end of the response to that request. A *critical instant* for a task is defined to be an instant at which a request for that task will have the largest response time. A *critical time zone* for a task is the time interval between a critical instant and the end of the response to the corresponding request of the task. We have the following theorem.

THEOREM 1. *A critical instant for any task occurs whenever the task is requested simultaneously with requests for all higher priority tasks.*

PROOF. Let $\tau_1, \tau_2, \cdots, \tau_m$ denote a set of priority-ordered tasks with τ_m being the task with the lowest priority. Consider a particular request for τ_m that occurs at t_1. Suppose that between t_1 and $t_1 + T_m$, the time at which the subsequent request of τ_m occurs, requests for task τ_i, $i < m$, occur at t_2, $t_2 + T_i$, $t_2 + 2T_i$, $\cdots$, $t_2 + kT_i$ as illustrated in Figure 1. Clearly, the preemption of τ_m by τ_i will cause a certain amount of delay in the completion of the request for τ_m that occurred at t_1, unless the request for τ_m is completed before t_2. Moreover, from Figure 1 we see immediately that advancing the request time t_2 will not speed up the completion of τ_m. The completion time of τ_m is either unchanged or delayed by such advancement. Consequently, the delay in the completion of τ_m is largest when t_2 coincides with t_1.

Repeating the argument for all τ_i, $i = 2, \cdots, m-1$, we prove the theorem.

One of the values of this result is that a simple direct calculation can determine whether or not a given priority assignment will yield a feasible scheduling algorithm. Specifically, if the requests for all tasks at their critical instants are fulfilled before their respective deadlines, then the scheduling algorithm is feasible. As an example, consider a set of two tasks τ_1 and τ_2 with $T_1 = 2$, $T_2 = 5$, and $C_1 = 1$, $C_2 = 1$. If we let τ_1 be the higher priority task, then from Figure 2(a) we see that such a priority assignment is feasible. Moreover, the value of C_2 can be increased at most to 2 but not further as illustrated in Figure 2(b). On the other hand, if we let τ_2 be the higher priority task, then neither of the values of C_1 and C_2 can be increased beyond 1 as illustrated in Figure 2(c).

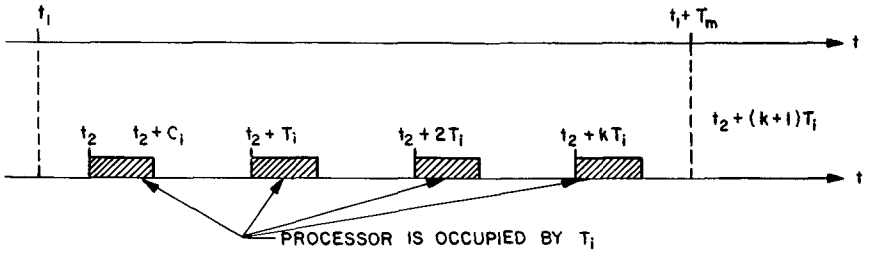

FIG. 1. Execution of τ_i between requests for τ_m

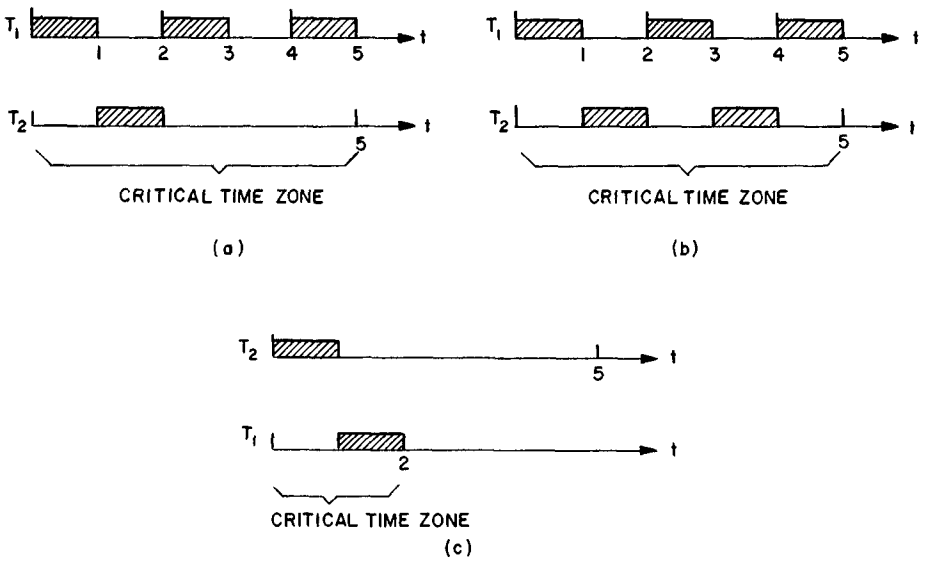

FIG. 2. Schedules for two tasks

The result in Theorem 1 also suggests a priority assignment that is optimum in the sense that will be stated in Theorem 2. Let us motivate the general result by considering the case of scheduling two tasks τ_1 and τ_2. Let T_1 and T_2 be the request periods of the tasks, with $T_1 < T_2$. If we let τ_1 be the higher priority task, then, according to Theorem 1, the following inequality must be satisfied:[1, 2]

$$\lfloor T_2/T_1 \rfloor \; C_1 + C_2 \leq T_2 . \tag{1}$$

If we let τ_2 be the higher priority task, then, the following inequality must be satisfied:

$$C_1 + C_2 \leq T_1 . \tag{2}$$

Since

$$\lfloor T_2/T_1 \rfloor \; C_1 + \lfloor T_2/T_1 \rfloor \; C_2 \leq \lfloor T_2/T_1 \rfloor \; T_1 \leq T_2 ,$$

(2) implies (1). In other words, whenever the $T_1 < T_2$ and C_1, C_2 are such that the task schedule is feasible with τ_2 at higher priority than τ_1, it is also feasible with τ_1 at higher priority than τ_2, but the opposite is not true. Thus we should assign higher priority to τ_1 and lower priority to τ_2. Hence, more generally, it seems that a "reasonable" rule of priority assignment is to assign priorities to tasks according to their request rates, independent of their run-times. Specifically, tasks with higher request rates will have higher priorities. Such an assignment of priorities will be known as the *rate-monotonic priority assignment*. As it turns out, such a priority assignment is optimum in the sense that no other fixed priority assignment rule can schedule a task set which cannot be scheduled by the rate-monotonic priority assignment.

[1] It should be pointed out that (1) is necessary but not sufficient to guarantee the feasibility of the priority assignment.

[2] $\lfloor x \rfloor$ denotes the largest integer smaller than or equal to x. $\lceil x \rceil$ denotes the smallest integer larger than or equal to x.

THEOREM 2. *If a feasible priority assignment exists for some task set, the rate-monotonic priority assignment is feasible for that task set.*

PROOF. Let $\tau_1, \tau_2, \cdots, \tau_m$ be a set of m tasks with a certain feasible priority assignment. Let τ_i and τ_j be two tasks of adjacent priorities in such an assignment with τ_i being the higher priority one. Suppose that $T_i > T_j$. Let us interchange the priorities of τ_i and τ_j. It is not difficult to see that the resultant priority assignment is still feasible. Since the rate-monotonic priority assignment can be obtained from any priority ordering by a sequence of pairwise priority reorderings as above, we prove the theorem.

5. *Achievable Processor Utilization*

At this point, the tools are available to determine a least upper bound to processor utilization in fixed priority systems. We define the *(processor) utilization factor* to be the fraction of processor time spent in the execution of the task set. In other words, the utilization factor is equal to one minus the fraction of idle processor time. Since C_i/T_i is the fraction of processor time spent in executing task τ_i, for m tasks, the utilization factor is:

$$U = \sum_{i=1}^{m} (C_i/T_i).$$

Although the processor utilization factor can be improved by increasing the values of the C_i's or by decreasing the values of the T_i's it is upper bounded by the requirement that all tasks satisfy their deadlines at their critical instants. It is clearly uninteresting to ask how small the processor utilization factor can be. However, it is meaningful to ask how large the processor utilization factor can be. Let us be precise about what we mean. Corresponding to a priority assignment, a set of tasks is said to *fully utilize* the processor if the priority assignment is feasible for the set and if an increase in the run-time of any of the tasks in the set will make the priority assignment infeasible. For a given fixed priority scheduling algorithm, the *least upper bound* of the utilization factor is the minimum of the utilization factors over all sets of tasks that fully utilize the processor. For all task sets whose processor utilization factor is below this bound, there exists a fixed priority assignment which is feasible. On the other hand, utilization above this bound can only be achieved if the T_i of the tasks are suitably related.

Since the rate-monotonic priority assignment is optimum, the utilization factor achieved by the rate-monontonic priority assignment for a given task set is greater than or equal to the utilization factor for any other priority assignment for that task set. Thus, the least upper bound to be determined is the infimum of the utilization factors corresponding to the rate-monotonic priority assignment over all possible request periods and run-times for the tasks. The bound is first determined for two tasks, then extended for an arbitrary number of tasks.

THEOREM 3. *For a set of two tasks with fixed priority assignment, the least upper bound to the processor utilization factor is* $U = 2(2^{\frac{1}{2}} - 1)$.

PROOF. Let τ_1 and τ_2 be two tasks with their periods being T_1 and T_2 and their run-times being C_1 and C_2, respectively. Assume that $T_2 > T_1$. According to the rate-monotonic priority assignment, τ_1 has higher priority than τ_2. In a critical time

zone of τ_2, there are $\lceil T_2/T_1 \rceil$ requests for τ_1. Let us now adjust C_2 to fully utilize the available processor time within the critical time zone. Two cases occur:

Case 1. The run-time C_1 is short enough that all requests for τ_1 within the critical time zone of T_2 are completed before the second τ_2 request. That is,

$$C_1 \leq T_2 - T_1 \lfloor T_2/T_1 \rfloor .$$

Thus, the largest possible value of C_2 is

$$C_2 = T_2 - C_1 \lceil T_2/T_1 \rceil .$$

The corresponding processor utilization factor is

$$U = 1 + C_1[(1/T_1) - (1/T_2) \lceil T_2/T_1 \rceil].$$

In this case, the processor utilization factor U is monotonically decreasing in C_1.

Case 2. The execution of the $\lceil T_2/T_1 \rceil$th request for t_1 overlaps the second request for τ_2. In this case

$$C_1 \geq T_2 - T_1 \lfloor T_2/T_1 \rfloor .$$

It follows that the largest possible value of C_2 is

$$C_2 = -C_1 \lfloor T_2/T_1 \rfloor + T_1 \lfloor T_2/T_1 \rfloor$$

and the corresponding utilization factor is

$$U = (T_1/T_2) \lfloor T_2/T_1 \rfloor + C_1[(1/T_1) - (1/T_2) \lfloor T_2/T_1 \rfloor].$$

In this case, U is monotonically increasing in C_1.

The minimum of U clearly occurs at the boundary between these two cases. That is, for

$$C_1 = T_2 - T_1 \lfloor T_2/T_1 \rfloor$$

we have

$$U = 1 - (T_1/T_2)[\lceil T_2/T_1 \rceil - (T_2/T_1)][(T_2/T_1) - \lfloor T_2/T_1 \rfloor]. \qquad (3)$$

For notational convenience,[3] let $I = \lfloor T_2/T_1 \rfloor$ and $f = \{T_2/T_1\}$. Equation (3) can be written as

$$U = 1 - f(1 - f)/(I + f).$$

Since U is monotonic increasing with I, minimum U occurs at the smallest possible value of I, namely, $I = 1$. Minimizing U over f, we determine that at $f = 2^{\frac{1}{2}} - 1$, U attains its minimum value which is

$$U = 2(2^{\frac{1}{2}} - 1) \simeq 0.83.$$

This is the relation we desired to prove.

It should be noted that the utilization factor becomes 1 if $f = 0$, i.e. if the request period for the lower priority task is a multiple of the other task's request period.

We now derive the corresponding bound for an arbitrary number of tasks. At this moment, let us restrict our discussion to the case in which the ratio between any two request periods is less than 2.

[3] $\{T_1/T_2\}$ denotes $(T_2/T_1) - \lfloor T_2/T_1 \rfloor$, i.e. the fractional part of T_2/T_1.

THEOREM 4. *For a set of m tasks with fixed priority order, and the restriction that the ratio between any two request periods is less than 2, the least upper bound to the processor utilization factor is* $U = m(2^{1/m} - 1)$.

PROOF. Let $\tau_1, \tau_2, \cdots, \tau_m$ denote the m tasks. Let $C_1, C_2, \cdots, C_m$ be the run-times of the tasks that fully utilize the processor and minimize the processor utilization factor. Assume that $T_m > T_{m-1} > \cdots > T_2 > T_1$. Let U denote the processor utilization factor. We wish to show that

$$C_1 = T_2 - T_1 .$$

Suppose that

$$C_1 = T_2 - T_1 + \Delta, \qquad \Delta > 0.$$

Let

$$\begin{aligned} C_1' &= T_2 - T_1 \\ C_2' &= C_2 + \Delta \\ C_3' &= C_3 \\ &\vdots \\ C_{m-1}' &= C_{m-1} \\ C_m' &= C_m . \end{aligned}$$

Clearly, $C_1', C_2', \cdots, C_{m-1}', C_m'$ also fully utilize the processor. Let U' denote the corresponding utilization factor. We have

$$U - U' = (\Delta/T_1) - (\Delta/T_2) > 0.$$

Alternatively, suppose that

$$C_1 = T_2 - T_1 - \Delta, \qquad \Delta > 0.$$

Let

$$\begin{aligned} C_1'' &= T_2 - T_1 \\ C_2'' &= C_2 - 2\Delta \\ C_3'' &= C_3 \\ &\vdots \\ C_{m-1}'' &= C_{m-1} \\ C_m'' &= C_m . \end{aligned}$$

Again, $C_1'', C_2'', \cdots, C_{m-1}'', C_m''$ fully utilize the processor. Let U'' denote the corresponding utilization factor. We have

$$U - U'' = -(\Delta/T_1) + (2\Delta/T_2) > 0.$$

Therefore, if indeed U is the minimum utilization factor, then

$$C_1 = T_2 - T_1$$

In a similar way, we can show that

$$\begin{aligned} C_2 &= T_3 - T_2 \\ C_3 &= T_4 - T_3 \\ &\vdots \\ C_{m-1} &= T_m - T_{m-1} . \end{aligned}$$

Consequently,

$$C_m = T_m - 2(C_1 + C_2 + \cdots + C_{m-1}).$$

To simplify the notation, let

$$g_i = (T_m - T_i)/T_i, \qquad i = 1, 2, \cdots, m.$$

Thus

$$C_i = T_{i+1} - T_i = g_iT_i - g_{i+1}T_{i+1}, \qquad i = 1, 2, \cdots, m-1$$

and

$$C_m = T_m - 2g_1T_1$$

and finally,

$$\begin{aligned} U &= \sum_{i=1}^{m} (C_i/T_i) = \sum_{i=1}^{m-1} [g_i - g_{i+1}(T_{i+1}/T_i)] + 1 - 2g_1(T_1/T_m) \\ &= \sum_{i=1}^{m-1} [g_i - g_{i+1}(g_i + 1)/(g_{i+1} + 1)] + 1 - 2[g_1/(g_1 + 1)] \\ &= 1 + g_1[(g_1 - 1)/(g_1 + 1)] + \sum_{i=2}^{m-1} g_i[(g_i - g_{i-1})/(g_i + 1)]. \end{aligned} \tag{4}$$

Just as in the two-task case, the utilization bound becomes 1 if $g_i = 0$, for all i.

To find the least upper bound to the utilization factor, eq. (4) must be minimized over the g_j's. This can be done by setting the first derivative of U with respect to each of the g_j's equal to zero, and solving the resultant difference equations:

$$\partial U/\partial g_j = (g_j^2 + 2g_j - g_{j-1})/(g_j + 1)^2 - (g_{j+1})/(g_{j+1} + 1) = 0, \qquad j = 1, 2, \cdots, m-1. \tag{5}$$

The definition $g_0 = 1$ has been adopted for convenience.

The general solution to eqs. (5) can be shown to be

$$g_j = 2^{(m-j)/m} - 1, \qquad j = 0, 1, \cdots, m-1. \tag{6}$$

It follows that

$$U = m(2^{1/m} - 1),$$

which is the relation we desired to prove.

For $m = 2$, eq. (6) is the same bound as was found directly for the set of two tasks with no restrictions on the request periods. For $m = 3$, eq. (6) becomes

$$U = 3(2^{1/3}) - 1) \simeq 0.78$$

and for large m, $U \simeq \ln 2$.

The restriction that the largest ratio between request period less than 2 in Theorem 4 can actually be removed, which we state as:

THEOREM 5. *For a set of m tasks with fixed priority order, the least upper bound to processor utilization is* $U = m(2^{1/m} - 1)$.

PROOF. Let $\tau_1, \tau_2, \cdots, \tau_i, \cdots, \tau_m$ be a set of m tasks that fully utilize the processor. Let U denote the corresponding utilization factor. Suppose that for

some i, $\lfloor T_m/T_i \rfloor > 1$. To be specific, let $T_m = qT_i + r$, $q > 1$ and $r \geq 0$. Let us replace the task τ_i by a task τ_i' such that $T_i' = qT_i$ and $C_i' = C_i$, and increase C_m by the amount needed to again fully utilize the processor. This increase is at most $C_i(q - 1)$, the time within the critical time zone of τ_m occupied by τ_i but not by τ_i'. Let U' denote the utilization factor of such a set of tasks. We have

$$U' < U + [(q - 1)C_i/T_m] + (C_i/T_i') - (C_i/T_i)$$

or

$$U' \leq U + C_i(q - 1)[1/(qT_i + r) - (1/qT_i)].$$

Since $q - 1 > 0$ and $[1/(qT_i + r)] - (1/qT_i) \leq 0$, $U' \leq U$. Therefore we conclude that in determining the least upper bound of the processor utilization factor, we need only consider task sets in which the ratio between any two request periods is less than 2. The theorem thus follows directly from Theorem 4.

6. *Relaxing the Utilization Bound*

The preceding section showed that the least upper bound imposed upon processor utilization by the requirement for real-time guaranteed service can approach $\ln(2)$ for large task sets. It is desirable to find ways to improve this situation, since the practical costs of switching between tasks must still be counted. One of the simplest ways of making the utilization bound equal to 1 is to make $\{T_m/T_i\} = 0$ for $i = 1, 2, \cdots, m - 1$. Since this cannot always be done, an alternative solution is to buffer task τ_m and perhaps several of the lower priority tasks and relax their hard deadlines. Supposing that the entire task set has a finite period and that the buffered tasks are executed in some reasonable fashion—e.g. in a first come first served fashion—then the maximum delay times and amount of buffering required can be computed under the assumptions of this paper.

A better solution is to assign task priorities in some dynamic fashion. The remaining sections of this paper are devoted to one particular method of dynamic priority assignment. This method is optimum in the sense that if a set of tasks can be scheduled by some priority assignment, it can also be scheduled by this method. In other words, the least upper bound on the processor utilization factor is uniformly 100 percent.

7. *The Deadline Driven Scheduling Algorithm*

We turn now to study a dynamic scheduling algorithm which we call the *deadline driven scheduling algorithm*. Using this algorithm, priorities are assigned to tasks according to the deadlines of their current requests. A task will be assigned the highest priority if the deadline of its current request is the nearest, and will be assigned the lowest priority if the deadline of its current request is the furthest. At any instant, the task with the highest priority and yet unfulfilled request will be executed. Such a method of assigning priorities to the tasks is a dynamic one, in contrast to a static assignment in which priorities of tasks do not change with time. We want now to establish a necessary and sufficient condition for the feasibility of the deadline driven scheduling algorithm.

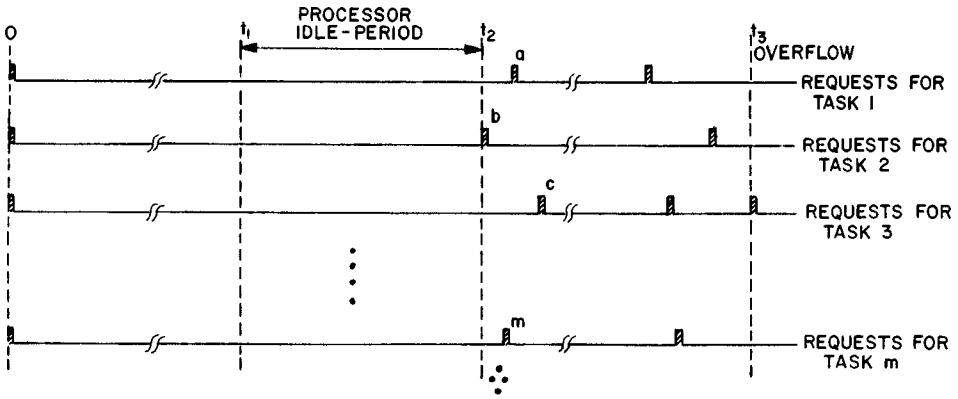

FIG. 3. Processing overflow following a processor idle period

THEOREM 6. *When the deadline driven scheduling algorithm is used to schedule a set of tasks on a processor, there is no processor idle time prior to an overflow.*

PROOF. Suppose that there is processor idle time prior to an overflow. To be specific, starting at time 0, let t_3 denote the time at which an overflow occurs, and let t_1 , t_2 denote the beginning and the end, respectively, of the processor idle period closest to t_3 (i.e. there is no processor idle time between t_2 and t_3 .) The situation is illustrated in Figure 3, where the times of the first request for each of the m tasks after the processor idle period $[t_1 , t_2]$ are denoted $a,b, \cdots , m$.

Suppose that from t_2 on we move all requests of task 1 up so that a will coincide with t_2 . Since there was no processor idle time between t_2 and t_3 , there will be no processor idle time after a is moved up. Moreover, an overflow will occur either at or before t_3 . Repeating the same argument for all other tasks, we conclude that if all tasks are initiated at t_2 , there will be an overflow with no processor idle period prior to it. However, this is a contradiction to the assumption that starting at time 0 there is a processor idle period to an overflow. This proves Theorem 6.

Theorem 6 will now be used to establish the following theorem:

THEOREM 7. *For a given set of m tasks, the deadline driven scheduling algorithm is feasible if and only if*

$$(C_1/T_1) + (C_2/T_2) + \cdots + (C_m/T_m) \leq 1.$$

PROOF. To show the necessity, the total demand of computation time by all tasks between $t = 0$ and $t = T_1T_2 \cdots T_m$, may be calculated to be

$$(T_2T_3 \cdots T_m)C_1 + (T_1T_3 \cdots T_m)C_2 + \cdots + (T_1T_2 \cdots T_{m-1})C_m .$$

If the total demand exceeds the available processor time, i.e.

$$(T_2T_3 \cdots T_m)C_1 + (T_1T_3 \cdots T_m)C_2 + \cdots + (T_1T_2 \cdots T_{m-1})C_m > T_1T_2 \cdots T_m \qquad (7)$$

or

$$(C_1/T_1) + (C_2/T_2) + \cdots + (C_m/T_m) > 1,$$

there is clearly no feasible scheduling algorithm.

To show the sufficiency, assume that the condition

$$(C_1/T_1) + (C_2/T_2) + \cdots + (C_m/T_m) \leq 1$$

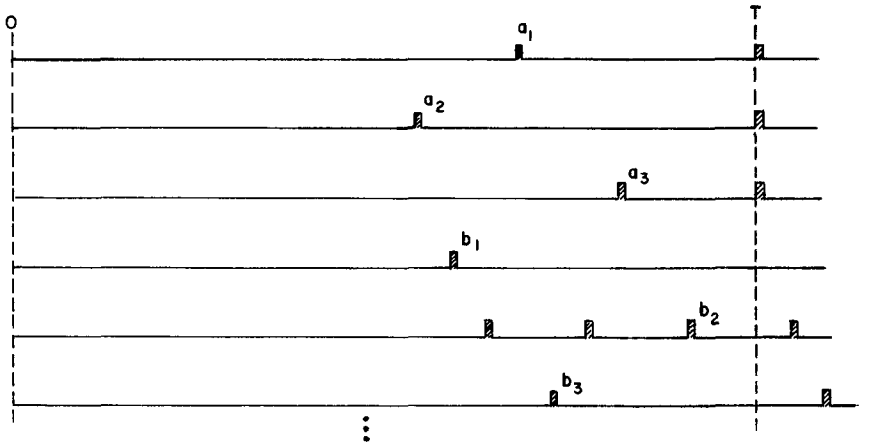

FIG. 4. Processing overflow at time T

is satisfied and yet the scheduling algorithm is not feasible. That is, there is an overflow between $t = 0$ and $t = T_1T_2 \cdots T_m$. Moreover, according to Theorem 6 there is a $t = T$ $(0 \leq T \leq T_1T_2 \cdots T_m)$ at which there is an overflow with no processor idle time between $t = 0$ and $t = T$. To be specific, let $a_1, a_2, \cdots, b_1, b_2, \cdots$, denote the request times of the m tasks immediately prior to T, where $a_1, a_2, \cdots$ are the request times of tasks with deadlines at T, and $b_1, b_2, \cdots$ are the request times of tasks with deadlines beyond T. This is illustrated in Figure 4.

Two cases must be examined.

Case 1. None of the computations requested at $b_1, b_2, \cdots$ was carried out before T. In this case, the total demand of computation time between 0 and T is

$$\lfloor T/T_1 \rfloor C_1 + \lfloor T/T_2 \rfloor C_2 + \cdots + \lfloor T/T_m \rfloor C_m .$$

Since there is no processor idle period,

$$\lfloor T/T_1 \rfloor C_1 + \lfloor T/T_2 \rfloor C_2 + \cdots \lfloor T/T_m \rfloor C_m > T.$$

Also, since $x \geq \lfloor x \rfloor$ for all x,

$$(T/T_1)C_1 + (T/T_2)C_2 + \cdots + (T/T_m)C_m > T$$

and

$$(C_1/T_1) + (C_2/T_2) + \cdots + (C_m/T_m) > 1,$$

which is a contradiction to (7).

Case 2. Some of the computations requested at $b_1, b_2, \cdots$ were carried out before T. Since an overflow occurs at T, there must exist a point T' such that none of the requests at $b_1, b_2, \cdots$ was carried out within the interval $T' \leq t \leq T$. In other words, within $T' \leq t \leq T$, only those requests with deadlines at or before T will be executed, as illustrated in Figure 5. Moreover, the fact that one or more of the tasks having requests at the b_i's is executed until $t = T'$ means that all those requests initiated before T' with deadlines at or before T have been fulfilled before T'. Therefore, the total demand of processor time within $T' \leq t \leq T$ is less than or equal to

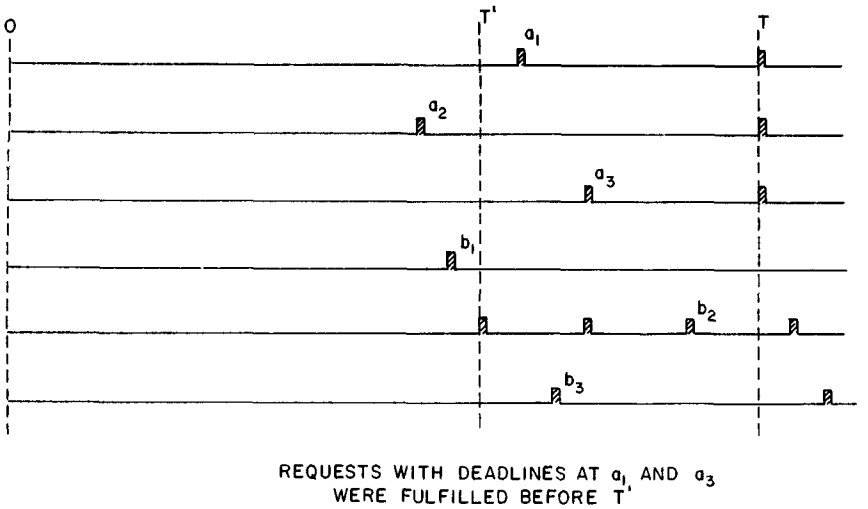

FIG. 5. Processing overflow at time T without execution of $\{b_i\}$ following T'

$$\lfloor (T - T')/T_1 \rfloor C_1 + \lfloor (T - T')/T_2 \rfloor C_2 + \cdots + \lfloor (T - T')/T_m \rfloor C_m .$$

That an overflow occurs at T means that

$$\lfloor (T - T')/T_1 \rfloor C_1 + \lfloor (T - T')T_2 \rfloor C_2 + \cdots + \lfloor (T - T')/T_m \rfloor C_m > T - T',$$

which implies again

$$(C_1/T_1) + (C_2/T_2) + \cdots + (C_m/T_m) > 1,$$

and which is a contradiction to (7). This proves the theorem.

As was pointed out above, the deadline driven scheduling algorithm is optimum in the sense that if a set of tasks can be scheduled by any algorithm, it can be scheduled by the deadline driven scheduling algorithm.

8. *A Mixed Scheduling Algorithm*

In this section we investigate a class of scheduling algorithms which are combinations of the rate-monotonic scheduling algorithm and the deadline driven scheduling algorithm. We call an algorithm in this class a mixed scheduling algorithm. The study of the mixed scheduling algorithms is motivated by the observation that the interrupt hardware of present day computers acts as a fixed priority scheduler and does not appear to be compatible with a hardware dynamic scheduler. On the other hand, the cost of implementing a software scheduler for the slower paced tasks is not significantly increased if these tasks are deadline driven instead of having a fixed priority assignment. To be specific, let tasks $1, 2, \cdots, k$, the k tasks of shortest periods, be scheduled according to the fixed priority rate-monotonic scheduling algorithm, and let the remaining tasks, tasks $k + 1$, $k + 2, \cdots, m$, be scheduled according to the deadline driven scheduling algorithm when the processor is not occupied by tasks $1, 2, \cdots, k$.

Let $a(t)$ be a nondecreasing function of t. We say that $a(t)$ is sublinear if for all

t and all T

$$a(T) \leq a(t + T) - a(t).$$

The availability function of a processor for a set of tasks is defined as the accumulated processor time from 0 to t available to this set of tasks. Suppose that k tasks have been scheduled on a processor by a fixed priority scheduling algorithm. Let $a_k(t)$ denote the availability function of the processor for tasks $k + 1, k + 2, \cdots, m$. Clearly, $a_k(t)$ is a nondecreasing function of t. Moreover, $a_k(t)$ can be shown to be sublinear by means of the critical time zone argument. We have:

THEOREM 8. *If a set of tasks are scheduled by the deadline driven scheduling algorithm on a processor whose availability function is sublinear, then there is no processor idle period to an overflow.*

PROOF. Similar to that of Theorem 6.

THEOREM 9. *A necessary and sufficient condition for the feasibility of the deadline driven scheduling algorithm with respect to a processor with availability function* $a_k(t)$ *is*

$$\lfloor t/T_{k+1} \rfloor C_{k+1} + \lfloor t/T_{k+2} \rfloor C_{k+2} + \cdots + \lfloor t/T_m \rfloor C_m \leq a_k(t)$$

for all t's which are multiples of T_{k+1}, *or* $T_{k+2}, \cdots,$ *or* T_m.

PROOF. The proof is quite similar to that of Theorem 7. To show the necessity, observe that at any moment the total demand of processor time cannot exceed the total available processor time. Thus we must have

$$\lfloor t/T_{k+1} \rfloor C_{k+1} + \lfloor t/T_{k+2} \rfloor C_{k+2} + \cdots + \lfloor t/T_m \rfloor C_m \leq a_k(t)$$

for any t.

To show the sufficiency, assume that the condition stated in the theorem is satisfied and yet there is an overflow at T. Examine the two cases considered in the proof of Theorem 7. For Case 1,

$$\lfloor T/T_{m+1} \rfloor C_{k+1} + \lfloor T/T_{k+2} \rfloor C_{k+2} + \cdots + \lfloor T/T_m \rfloor C_m > a_k(T),$$

which is a contradiction to the assumption. Note that T is multiple of T_{k+1}, or $T_{k+2}, \cdots,$ or T_m. For Case 2,

$$\lfloor (T - T')/T_{k+1} \rfloor C_{k+1} + \lfloor (T - T')/T_{k+2} \rfloor C_{k+2} + \cdots + \lfloor (T - T')/T_m \rfloor C_m > a_k(T - T').$$

Let ϵ be the smallest nonnegative number such that $T - T' - \epsilon$ is a multiple of T_{k+1}, or $T_{k+2}, \cdots,$ or T_m. Clearly,

$$\lfloor (T - T' - \epsilon)/T_{k+j} \rfloor = \lfloor (T - T')/T_{k+j} \rfloor \qquad \text{for each } j = 1, 2, \cdots, m - k$$

and thus

$$\lfloor (T - T' - \epsilon)/T_{k+1} \rfloor C_{k+1} + \lfloor (T - T' - \epsilon)/T_{k+2} \rfloor C_{k+2} + \cdots + \lfloor (T - T' - \epsilon)/T_m \rfloor C_m > a_k(T - T') \geq a_k(T - T' - \epsilon),$$

which is a contradiction to the assumption. This proves the theorem.

Although the result in Theorem 9 is a useful general result, its application involves the solution of a large set of inequalities. In any specific case, it may be advantageous to derive sufficient conditions on schedule feasibility rather than work directly from Theorem 9. As an example, consider the special case in which three

tasks are scheduled by the mixed scheduling algorithm such that the task with the shortest period is assigned a fixed and highest priority, and the other two tasks are scheduled by the deadline driven scheduling algorithm. It may be readily verified that if

$$1 - (C_1/T_1) - \min [(T_1 - C_1)/T_2, (C_1/T_1)] \geq (C_2/T_2) + (C_3/T_3),$$

then the mixed scheduling algorithm is feasible. It may be also verified that if

$$C_2 \leq a_1(T_2), \quad \lfloor T_3/T_2 \rfloor \; C_2 + C_3 \leq a_1(\lfloor T_3/T_2 \rfloor \; T_2), \text{ and}$$
$$(\lfloor T_3/T_2 \rfloor + 1) \; C_2 + C_3 \leq a_1(T_3),$$

then the mixed scheduling algorithm is feasible.

The proof of these statements consists of some relatively straightforward but extensive inequality manipulation, and may be found in Liu [13]. Unfortunately, both of these sufficient conditions correspond to considerably lower processor utilization than does the necessary and sufficient condition of Theorem 9.

9. *Comparison and Comment*

The constraints in Theorem 9 strongly suggest that 100 percent utilization is not achievable universally by the mixed scheduling algorithm. The following simple example will illustrate this. Let $T_1 = 3$, $T_2 = 4$, $T_3 = 5$, and $C_1 = C_2 = 1$. Since $a_1(20) = 13$, it can be easily seen that the maximum allowable C_3 is 2. The corresponding utilization factor is

$$U = \tfrac{1}{3} + \tfrac{1}{4} + \tfrac{2}{5} = 98.3\%.$$

If these three tasks are scheduled by the deadline driven scheduling algorithm, C_2 can increase to $2.0833\cdots$ and achieve a 100 percent utilization. If they are all scheduled by the fixed priority rate-monotnic scheduling algorithm, C_3 is restricted to 1 or less, and the utilization factor is restricted to at most

$$U = \tfrac{1}{3} + \tfrac{1}{4} + \tfrac{1}{5} = 78.3\%,$$

which is only slightly greater than the worst case three task utilization bound.

Although a closed form expression for the least upper bound to processor utilization has not been found for the mixed scheduling algorithm, this example strongly suggests that the bound is considerably less restrictive for the mixed scheduling algorithm than for the fixed priority rate-monotonic scheduling algorithm. The mixed scheduling algorithm may thus be appropriate for many applications.

10. *Conclusion*

In the initial parts of this paper, five assumptions were made to define the environment which supported the remaining analytical work. Perhaps the most important and least defensible of these are (A1), that all tasks have periodic requests, and (A4), that run-times are constant. If these do not hold, the critical time zone for each task should be defined as the time zone between its request and deadline during which the maximum possible amount of computation is performed by the tasks having higher priority. Unless detailed knowledge of the run-time and request periods are available, run-ability constraints on task run-times would have to be

computed on the basis of assumed periodicity and constant run-time, using a period equal to the shortest request interval and a run-time equal to the longest run-time. None of our analytic work would remain valid under this circumstance, and a severe bound on processor utilization could well be imposed by the task aperiodicity. The fixed priority ordering now is monotonic with the shortest span between request and deadline for each task instead of with the undefined request period. The same will be true if some of the deadlines are tighter than assumed in (A2), although the impact on utilization will be slight if only the highest priority tasks are involved. It would appear that the value of the implications of (A1) and (A4) are great enough to make them a design goal for any real-time tasks which must receive guaranteed service.

In conclusion, this paper has discussed some of the problems associated with multiprogramming in a hard-real-time environment typified by process control and monitoring, using some assumptions which seem to characterize that application. A scheduling algorithm which assigns priorities to tasks in a monotonic relation to their request rates was shown to be optimum among the class of all fixed priority scheduling algorithms. The least upper bound to processor utilization factor for this algorithm is on the order of 70 percent for large task sets. The dynamic deadline driven scheduling algorithm was then shown to be globally optimum and capable of achieving full processor utilization. A combination of the two scheduling algorithms was then discussed; this appears to provide most of the benefits of the deadline driven scheduling algorithm, and yet may be readily implemented in existing computers.

REFERENCES

1. MANACHER, G. K. Production and stabilization of real-time task schedules. *J. ACM 14*, 3 (July 1967), 439–465.
2. MCKINNEY, J M. A survey of analytical time-sharing models. *Computing Surveys 1*, 2 (June 1969), 105–116.
3. CODD, E. F. Multiprogram scheduling. *Comm ACM 3*, 6, 7 (June, July 1960), 347–350; 413–418.
4. HELLER, J. Sequencing aspects of multiprogramming. *J. ACM 8*, 3 (July 1961), 426–439.
5. GRAHAM, R. L. Bounds for certain multiprocessing anomalies. *Bell System Tech. J. 45*, 9 (Nov. 1966), 1563–1581.
6. OSCHNER, B. P. Controlling a multiprocessor system. *Bell Labs Record 44*, 2 (Feb. 1966), 59–62.
7. MUNTZ, R. R., AND COFFMAN, E. G., JR. Preemptive scheduling of real-time tasks on multiprocessor systems. *J. ACM 17*, 2 (Apr 1970), 324–338.
8. BERNSTEIN, A. J., AND SHARP, J. C. A policy-driven scheduler for a time-sharing system. *Comm. ACM 14*, 2 (Feb. 1971), 74–78.
9. LAMPSON, B. W. A scheduling philosophy for multiprocessing systems. *Comm. ACM 11*, 5 (May, 1968), 347–360.
10. MARTIN, J. *Programming Real-Time Computer Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1965.
11. JIRAUCH, D. H. Software design techniques for automatic checkout. *IEEE Trans. AES-3*, 6 (Nov. 1967), 934–940.
12. MARTIN, J. Op. cit., p. 35 ff
13. LIU, C. L. Scheduling algorithms for hard-real-time multiprogramming of a single processor. JPL Space Programs Summary 37–60, Vol. II, Jet Propulsion Lab., Calif. Inst. of Tech., Pasadena, Calif., Nov. 1969.

RECEIVED MARCH 1970; REVISED JANUARY 1972