VOLUME 6
NUMBER 1

October 1959

Management Science

SCHEDULING WITH DEADLINES AND LOSS FUNCTIONS*[1]

ROBERT McNAUGHTON

The Moore School of Electrical Engineering, University of Pennsylvania

The problem of this paper is that of scheduling several one-stage tasks on several processors, which are capable of handling the tasks with varying degrees of efficiency, to minimize the total loss, which is a sum of losses associated with the individual tasks.

Each task has a deadline; the individual loss associated with it is a function of amount of time between the deadline and the time of completion if the former precedes the latter, and zero otherwise.

1. General Comments on Scheduling

There are at least two different kinds of requirements for scheduling a number of tasks. One is the requirement that all tasks be finished as soon as possible. This kind of requirement has received attention in the literature.[2] Another kind of requirement which has received little, if any, attention is that the tasks be completed according to their relative urgency; each task has associated with it a deadline, and loss for any failure to meet the deadline. It is the latter type of scheduling problem which is treated in this paper.

As far as scheduling is concerned it is to be noted that there are two kinds of deadline. The first kind is what might be called an *absolute deadline*, one in which a task has no value at all if it is not completed by the deadline: in other words one which might just as well be forgotten once the deadline has passed. An example of a task with an absolute deadline is a computation for the construction of a telescope lens needed for an observation of a particular celestial phenomenon on a certain hour of a certain date. More often a deadline is an expectation date set more or less arbitrarily; the task is by no means without any value if it passes

* Received November 1958.

[1] This paper is a revision of material in Section 2 of the Sixth Quarterly Progress Report, and Section 2 of the Eighth Quarterly Progress Report, of Contract No. DA-36-039-SC-72344 (15 December 1957) at the Moore School of Electrical Engineering, University of Pennsylvania. The sponsoring agent was the Signal Engineering Laboratories; the problem investigated was that of the scheduling of the multiple computer. It has seemed desirable to present the material of this paper from a more general standpoint.

[2] An expository account of the material in this literature and a copious bibliography can be found in Richard Bellman's "Mathematical Aspects of Scheduling Theory", *Journal for the Society of Industrial and Applied Mathematics*, Vol. 4 (1956) pp. 168–205.

its deadline, although in the usual case it may lose some of its value. The type of deadline which will be of concern in this paper is one in which there is a loss which is a function at the time of completion. The loss is zero up to a certain point; from that point on, the loss is a monotonic increasing function of the time of completion. The crucial point of time is called a *relative deadline.*

The kind of loss function to assume is something which is not yet settled, in the search for a scheduling method. In this paper I shall assume that when there is a loss function it is linear in the time after the deadline. Thus for any task there exists a deadline d and a penalty coefficient p such that, for every t, if the task finishes at t there is a loss of $\max(0, p(t - d))$; in other words, there is no loss if the task is completed on or before time d, but a loss of p units for every unit of time after d during which the task is incomplete. There are frequent examples of such tasks in business contracts where it is written into the agreement that there is to be a reduction in payment of a certain fixed amount for every day which the task goes beyond the deadline.

However, the objective in all of this work is to arrive at a generally satisfactory scheduling method. The objective, admittedly, is rather vague, but the linear loss function is not suitable for every occasion. It would seem better to assume a quadratic loss function for the general case of a relative deadline, since in many cases the further past the deadline a task gets, the more urgent it becomes to get it done, and hence the greater the loss per unit of time delay in getting it done.

One kind of task, which is quite frequent, is a task which has no deadline at all but is simply placed on a waiting list and is done when its turn comes up. Although there is no explicit loss function, the situation always constitutes an implicit loss function. For if the task were to be misplaced or left incomplete for some other reason, there would certainly be someone to complain; in one form or another, there would be a loss which would be a monotonic (at least weakly monotonic) increasing function of the time of completion. On a single processor (i.e. device used to execute the task) these constitute no problem for scheduling; they are simply done in order of their appearance in the hopper. However, if there are several processors, the matter is not that easy. The following scheduling problem, for example, might arise: task A has been placed in the hopper before task B; however, the one available processor can do B much more efficiently than A; should A be put on the processor which does it inefficiently simply because it was placed first, or should B be put on this processor and A held off so that it can be put on a more efficient processor later? In other words, should priority or efficient use of equipment be the decisive consideration?

My suggestion for a scheduling method in this type of situation is to have some person assign a quadratic loss function to each task. He can do this subjectively on the basis of what he knows about the situation. The loss will be $p_2(t - h)^2 + p_1(t - h)$ where h is the time when the task is put into the hopper and t is the time of completion; p_2 and p_1 are constants which must be assigned by the human being in charge. Note that in this formulation the loss at time h is zero. This is by fiat; what matters for scheduling is the loss of t relative to

other times; an addition of a constant to the loss function is inconsequential. Note also that the first derivative of this function is given by the expression $2p_2(t - h) + p_1$. This quantity is a measure of the urgency at time t of getting the task done. The larger p_2, the faster this urgency increases as the task goes farther and farther past its introduction into the hopper. (Note that, in this situation, time h, the time the task is put into the hopper, behaves mathematically like a deadline. This fact shows that the theorems below, which assume that the deadlines are all at time 0, such as theorem 2.3, do have application.)

In my opinion, quadratic loss functions will play an important role in scheduling. The present study is just the beginning; the only loss functions studied in the remaining mathematical sections are linear. If there is any further research on this problem, it should ultimately attempt to extend the present work to quadratic loss functions.

In this general discussion of scheduling I should like to make a vague distinction between two methods of computing a schedule. The first method is an *instantaneous decision method*, made on the basis of data about the tasks at hand. (An example is the method given in theorem 2.3 of the next section (section 2) for the special conditions of that theorem.) The other is that of a more complicated computation with the data, which is in effect a placing of the tasks on a sort of time table and reshuffling them until the loss is minimized. I call this the *simulation method*, for before beginning on any task it is necessary first to simulate the carrying through of all the tasks. (An example is the method suggested for scheduling the single processor described at the end of section 2.)

Perhaps it is well to elaborate the difference in these two methods of scheduling by means of an example from every-day life. When someone sits down at a desk at the beginning of a busy day and decides that a certain letter must first be written, without going through in his mind the other tasks which must be completed in the course of the day, we have an example of the working of the instantaneous method. On the other hand if the individual at the desk has to go through all the tasks, seeing if he has time to finish them all, etc., before deciding which task to perform first, then we have an example of the simulation method; in this case the simulation is the doing of all the tasks in the imagination before beginning them actually.

The distinction between the instantaneous and the simulation method is perhaps no more than the difference between a simply computed schedule and a schedule whose computation is involved. But it is rather significant that the mathematical results of the following sections seem to show that, in general, optimal scheduling will require the simulation method. It is rather disappointing to realize that we may have to forego the simplicity of the instantaneous method.

In the following sections four problems are discussed in a technical and abstract manner. The results are isolated. There is a need for further research, but the most fruitful immediate direction for this research is not clear. One important complication which has not been considered in this paper is that often the tasks have several stages and that various stages have various rates of efficiency on

the various processors. Thus, for example, the first stage of a task may take 10 minutes on one processor and 20 minutes on another while the second stage takes 20 minutes on the first and 10 minutes on the second. This state of affairs seems very likely; for, if there is any reason to build two processors which differ in performance, it seems natural to think that they should be built so that one excels in one type of operation and the other in another. It also seems likely that most tasks will contain stretches of both types of operation. Here the problem will bear some relation to the book-binding problem, except that it will be more difficult.

2. Scheduling a Single Processor with Tasks Having Deadlines and Linear Loss Functions

A number of tasks, m, for a single computer are given, (1), (2), $\cdots$, (m). For each task (i), we know the time a_i it takes to complete (i); there is a deadline at time d_i; if the task is finished d_i or less units of time after the present (time 0) there is no loss; otherwise there is a loss of $p_i x$ ('p' for penalty) units of value on (i), x being the number of units of time from time d_i to the completion of (i). The problem is to schedule these m tasks on the processor so as to minimize the total loss. Let $r_i = p_i/a_i$.

THEOREM 2.1. Assume that both (i) and (j) must be scheduled in $a_i + a_j$ consecutive units of time starting at time t after both deadlines have passed. If $r_i > r_j$, then the loss on (i) and (j) together is less if (i) is done before (j).

PROOF: Let c be the loss on (i) and (j) together computed as far as time t. Then the loss on (i) and (j), if (i) is done before (j), is

$$c + a_i p_i + (a_i + a_j)p_j .$$

The loss, if (j) is done before (i) is

$$c + a_j p_j + (a_i + a_j)p_i .$$

Taking $c' = c + a_i p_i + a_j p_j$, the two losses are, respectively, $c' + a_i p_j$ and $c' + a_j p_i$. If $r_i = (p_i/a_i) > r_j = p_j/a_j$ then $a_j p_i > a_i p_j$ and the first loss is less than the second; the loss is then less if (i) is done before (j).

In scheduling it is possible to split a task in two, for example, to do part of it between time 4 and time 6 and the remaining part between time 8 and 11. Similarly it may be split in any number of parts. The following theorem shows that splitting is not necessary. (This result holds only for a single processor; in other cases, the optimal solution generally requires splitting tasks.)

THEOREM 2.2. For every scheduling problem there exists an optimal solution in which no task is split.

PROOF: Suppose there is a schedule in which some tasks are split. I shall prove theorem 2.2 by showing that there exists a schedule without splitting whose total loss is less than or equal to that of the given schedule. For each i suppose (i) is split into s_i parts; $s_i = 1$ if (i) is not split. We must prove that for a given schedule there exists another schedule with no greater loss in which $\sum_{i=1}^{m} s_i = m$.

I prove this fact by proving that for a given schedule in which $\sum_{i=1}^{m} s_i > m$, there exists a schedule with no greater loss in which $\sum_{i=1}^{m} s_i$ is reduced.

Suppose then that $\sum_{i=1}^{m} s_i > m$. Then at least one task is split into at least two parts. Suppose (i) is so split and that the time between t_1 and t_2 and the time between t_3 and t_4 are two consecutive parts of (i)'s schedule; this means that $t_1 < t_2 < t_3 < t_4$, (i) is not scheduled immediately before t_1 or immediately after t_4 or at anytime between t_2 and t_3. Rescheduled so that part of (i) which occurs between t_1 and t_2 is scheduled between $t_3 - (t_2 - t_1)$ and t_3 (thus reducing the number of parts of (i) by one), and so that anything scheduled between t_2 and t_3 is moved back $t_2 - t_1$ units of time. (i) is completed at the same time; tasks scheduled between t_2 and t_3 are finished sooner or at the same time; other tasks are finished at the same time. Therefore the loss is either decreased or remains the same. The number of parts of (i) is decreased by 1. The number of parts of any other task is either decreased (as when two parts are brought together at time t_1) or remains the same. Hence $\sum_{i=1}^{m} s_i$ is decreased by at least 1.

THEOREM 2.3. If $d_1 = d_2 = \cdots = d_m = 0$ and if tasks scheduled without splits in order of decreasing r_i then the total cost is minimized (which is the solution of the scheduling problem for this case).

The proof follows directly from theorems 2.1 and 2.2.

The following theorem, a generalization of theorem 2.3, gives a sufficient (but not necessary) condition that a schedule be minimal.

THEOREM 2.4. If the problem is such that the tasks can be scheduled without splitting in order of decreasing r_i, with no unused time before all tasks are finished, and with no task finishing before its deadline, then this schedule is minimal.

It must be proved that any other schedule S' for the given problem has a total loss $c(S')$ greater than or equal to $c(S)$, the total loss of the given schedule S. Let $c^*(S')$ be the total loss for S' computed on the assumption that $d_1 = d_2 = \cdots = d_m = 0$, $c^*(S)$ be the total loss for S computed on this assumption. By theorem 2.3, $c^*(S') > c^*(S)$. By theorem 2.2 we can assume that S' has no splits. S' differs from S in that certain tasks (i_1), (i_2), $\cdots$ (i_q) occur later, and certain other tasks (j_1), (j_2), $\cdots$, (j_r) occur earlier; let us suppose that all tasks other than these are scheduled at the same time in S' as in S. Let $c(S, i)$ be the loss on (i) in S as figured by the data of the problem and $c^*(S, i)$ be the loss on (i) in S computed on the assumption that $d_1 = d_2 = \cdots = d_m = 0$. Obviously,

$$c(S') - c(S) = \sum_{k=1}^{q} [c(S', i_k) - c(S, i_k)] - \sum_{k=1}^{r} [c(S, j_k) - c(S', j_k)]$$

and similarly for c^*. Now, for each k, $c(S', i_k) - c(S, i_k) = c^*(S', i_k) - c^*(S, i_k)$; this assertion is justified by the fact that the deadline of (i_k) for the given problem is before the finish of (i_k) in S, by hypotheses of the theorem, and also before the finish of (i_k) in S', since (i_k) occurs later in S' than in S. For each k, $c(S, j_k) - c(S', j_k) \leqq c^*(S, j_k) - c^*(S', j_k)$; this assertion is justified by the

fact that in scheduling (j_k) earlier in S' than in S it may be scheduled to finish before its deadline of the original problem; but, since the deadline used in computing c^* is assumed to be 0, it cannot be scheduled to finish before that deadline. Hence.

$$c(S') - c(S) \geqq c^*(S') - c^*(S).$$

Since $c^*(S') - c^*(S) \geqq 0$ by theorem 2.2, $c(S') - c(S) \geqq 0$. Q.E.D.

The procedure of scheduling for the general problem of this section, suggested by the last theorem, involves the following two steps. First, arrange the tasks consecutively without splits and without unused time in order of decreasing r_i, where if $r_i = r_j$ schedule (i) before (j) if (i) has an earlier deadline. Second, if there are tasks which finish before their deadline, investigate the possibilities of decreasing the cost by rearranging so that these tasks finish later; select the arrangement with the minimal total cost. The second part of this procedure has not been investigated in detail, and as it stands is not in satisfactory form.

It is interesting that theorem 2.2 does not depend on the linear loss function; all the other theorems, unfortunately, do.

3. Scheduling Many Processors Which Are Exactly Alike to Finish All Tasks as Soon as Possible

In this section I assume that there are n processors, all of which are exactly alike. m tasks are given, each of which takes as long on one processor as on any other. Let $a_1, \cdots, a_m$ be the times for the m tasks. The tasks can be split in any number of ways on the processors, but of course two processors cannot work on the same task at the same time. Thus, for example, (1) could be scheduled on the first processor for $a_1/2$, then on the fourth processor for $a_1/4$ and then on the sixth processor for $a_1/4$, whereupon it would be completed.

THEOREM 3.1. If $s \geqq (1/n) \sum_{i=1}^{m} a_i$, then a necessary and sufficient condition that there exists a schedule in which all tasks are completed by time s is that, for all i, $a_i \leqq s$.

PROOF: The necessity is obvious. I shall give a constructive proof of sufficiency. Assume each $a_i \leqq s$. I shall show how the tasks can be scheduled so that after each task has been scheduled, the following *condition* is fulfilled: the schedule of every processor, with the possible exception of one, is either entirely filled up or is entirely empty. Scheduling (1) entirely on the first processor, there will be $s - a_1$ units of time left on the first processor to be filled. Obviously the condition is fulfilled. Now assume the $i - 1$ tasks, (1), (2), $\cdots$, $(i - 1)$, have been scheduled and that the condition is fulfilled. I must prove that it is possible to schedule the i^{th} task so that the condition is still fulfilled. Note that if $i \leqq m$, after the first $i - 1$ tasks have been scheduled the total unscheduled time for all the processors is less than or equal to a_i, since $ns \geqq \sum_{i=1}^{m} a_i$. But since (i) cannot be put on two processors at the same time, we cannot be sure merely from this fact that (i) can be scheduled. Case I. There is no processor whose schedule is partially filled. Then schedule (i) on an empty processor. (It is not difficult to see that there must be one.) The condition is still fulfilled. For cases

II and III, let the processor P be the processor whose schedule is not yet filled; suppose it has x units of time left unscheduled and, hence, $s - x$ units of time scheduled. Case II. $x \geqq a_i$. Schedule (i) wholly on the processor with the partially filled schedule. The condition is then still fulfilled. Case III. $x < a_i$. Suppose the schedule of the processor in question is filled from time t to time $t - (s - x)$, and empty elsewhen. Schedule (i) from 0 to t on P, from t to $t + a_i - x$ on a new processor, and then from $t + (0 - x)$ to s on P. That $t + a_i - x \leqq t + s - x$, i.e. that (i) can be taken off the new processor before it is put on P, follows from the fact that $a_i \leqq s$. The condition is still fulfilled, since the new processor is now the only one whose schedule is partially filled. Since these considerations are valid for $i \leqq m$, the proof is complete.

If $s < (1/n) \sum_{i=1}^{m} a_i$ then it is easy to see that it is not possible to finish all the tasks before time s. Hence theorem 3.1 is the complete solution to the problem as formulated in this section.

4. Scheduling Many Processors Which Are Alike, with Deadlines and Linear Loss Functions

The assumptions are the same here as in the last section except that each task (i) has a deadline d_i and a penalty p_i for each unit of time the completion of (i) is delayed beyond time d_i. In general, the optimal solution will involve splitting tasks. "Splitting" was defined in section 2.2, but here a task which begins on one processor may continue on another processor; such a task is *split* whether or not it is interrupted. A task is not split only when it runs on one processor and runs without interruption.

All the work of this section is a proof of theorem 4.1, which says that splitting is never necessary for the case where $d_1 = d_2 = \cdots = d_m = 0$. This case is interesting, because it is the extreme linear case. I say that it is the extreme linear case because the presence of non-zero deadlines does something to destroy the linearity of the loss function by constituting what is essentially a discontinuity in it: the loss of a task (i) finishing at t is 0 for $t \leqq d_i$ and $p_i(t - d_i)$ for $t > d_i$, and is therefore not truly linear unless $d_i = 0$. It has already been mentioned that, if the deadlines are not all zero, a minimal solution may necessarily have splits.

There is more to be said about the problem of this section beyond theorem 4.1. Indeed there is more to be said even for the case where the deadlines are all zero, but no work has been done on this part of the problem.

The *data* of a schedule are the number of processors, the number of tasks, and the length of each task. (Note that the deadlines and the loss functions are not included.) Given a schedule, the data are uniquely determined. In transforming one schedule to another, in a fixed problem, the data must be kept the same.

THEOREM 4.1. If $d_1 = d_2 = \cdots = d_m = 0$ and S is a schedule with at least one split, but only finitely many, then there is a schedule for the same data without splits whose loss is no greater than that of S.

Several lemmas will be necessary to prove this theorem. By its hypothesis there are finitely many splits, and hence finitely many times $t_1, t_2, \cdots, t_b$ (where

$t_1 < t_2 < \cdots < t_b$) such that for each $i \leqq b$ at least one task which has not been completed is taken off some processor, and these are the only such times. The method of proof for theorem 4.1 will be to show that there is an effective method of determining a schedule $f(S)$ for the same data having a cost no greater than that of S, where if $b > 1$ then $f(S)$ is exactly like S up to time t_{b-1} but has no incomplete tasks taken off any processors after time t_{b-1}, and where if $b = 1$ then $f(S)$ has no incomplete tasks taken off any processors after time 0 (or, in other words, $f(S)$ has no splits). In short $f(S)$ reduces b by 1 without increasing the cost of S. Thus $f^b(S)$ will be a schedule for the same data, will have no splits and will have a cost which is less than or equal to that of S. The remainder of the proof will be concerned with showing that f has the desired properties.

Let tasks (i_1), (i_2), $\cdots$, (i_r) be all the tasks which are, respectively, taken off processors P_1, P_2, $\cdots$, P_r (these being numbered for convenience) incomplete at time t_b. Each of these tasks must at some time after time t_b be placed on some processor and left there until completed.

LEMMA 1. Let S' result from S by interchanging the assignments for P_i and P_j $(i, j \leqq r)$ after time t_b. (That is, let all tasks assigned to P_i in S after time t_b be assigned to P_j in S' in the same order after t_b, and let all such tasks assigned to P_j in S be similarly assigned to P_i in S'). Then S' is a schedule for the same data, no tasks are taken off any processor incomplete in S' at any time unless it is taken off the same processor incomplete at the same time in S; and the cost of S' is the same as the cost of S.

The proof of lemma 1 is straightforward, and is omitted here.

Each P_i, $1 \leqq i \leqq r$, may have some of the tasks (i_1), $\cdots$, (i_r) on it after time t_b. It is convenient to divide the argument into two cases, according to whether there are or there are not such processors.

CASE I. Some of the processors P_1, $\cdots$, P_r have some of the tasks (i_1), $\cdots$, (i_r) on them after time t_b. Without loss of generality, suppose that (i_1), $\cdots$, (i_s) $(s \leqq r)$ each occurs on one of the processors P_1, $\cdots$, P_r after time t_b and is the first such, and that each of (i_{s+1}), $\cdots$, (i_r) (if $s < r$) occurs after t_b either not on any of the P_1, $\cdots$, P_r, or on one of them after one of the (i_1), $\cdots$, (i_s) has been completed. Using lemma 1 a number of times S can be rearranged so that, for each $j \leqq s$, (i_j) is on P_j after time t_b. (Quite possibly after doing this some tasks (i_j) may not be taken off P_j at time t_b. If this is true for all j from 1 to s then S comes under Case II. In what follows we assume that for each $j \leqq s$, (i_j) is taken off P_j incomplete at time t_b). Let us assume that this is true of S. The following will then be true, for each j $(1 \leqq j \leqq s)$.

(A) For some time q_j, $t_{b-1} \leqq q_j < t_b$, (i_j) is on P_j between q_j and t_b (but not between t_{b-1} and q_j if $q_j \neq t_{b-1}$). (B) There exist times u_j and u'_j such that $u'_j > u_j > t_b$ and such that (i_j) is on P_j between u_j and u'_j. (C) (i_j) is not on any other processor after time t_{b-1}, and not on P_j except during the two intervals mentioned. (D) None of (i_j), $\cdots$, (i_r) is on P_j between time t_b and time u_j; what is more, (E) no task on P_j between time t_b and time u_j is on any other processor after time t_{b-1} (for (i_1), $\cdots$, (i_r) are the only tasks taken off processors incomplete after time t_{b-1}).

Now let S' be obtained from S by changing P_j, for all j, $1 \leqq j \leqq r$, between time q_j and time u_j as follows. Reschedule (i_j) to occur between $u_j - (t_b - q_j)$ and u_j instead of between q_j and t_b, and move ahead $t_b - q_j$ units of time all that has occurred between time t_b and time u_j.

LEMMA 2. That the conditions described in the penultimate paragraph above hold for all j, $1 \leqq j \leqq r$, imply that S', obtained from S as described in the last paragraph above, (W) is a schedule for the same data, (X) has no tasks taken off any processor incomplete except at times when tasks are taken off processors incomplete in S, (Y) has a cost no greater than that of S and (Z) falls under Case II.

PROOF: To prove that S' is a schedule it is only necessary to prove that in constructing S' from S no task is scheduled on two distinct processors at the same time. But this follows from the construction, and from (E) and the fact that S itself is a schedule. That S' has the same data as S follows from the construction. Thus (W) is proved.

(X) follows immediately by construction and conditions described. (Note here that, quite possibly, (i_j) might be scheduled on P_j for an interval beginning before t_{b-1} and ending at t_b. In that case $q_j = t_{b-1}$, and as a result of the construction (i_j) will be taken off processor P_j incomplete at time t_{b-1} in S' although it was not so taken off in S. (X) in this case is still true, because t_{b-1} is a time when incomplete tasks are taken off processors in S.)

(Y) follows from the fact that no task finishes later in S' than in S and some tasks finish earlier (namely the tasks scheduled between t_b and u_j on P_j, for each j between 1 and s).

(Z) follows from the fact that in S' the only tasks taken off processors incomplete at time t_b are the tasks $(i_{s+1}), \cdots, (i_r)$, which are taken respectively off $P_{s+1}, \cdots, P_r$; and from the fact that the processors $P_{s+1}, \cdots, P_r$ have none of the tasks $(i_j), \cdots, (i_r)$ scheduled on them (in S or S') after time t_b.

This completes the proof of lemma 2, which shows, in effect, that Case I can be reduced to Case II. Note that so far no use has been made of the fact that $d_1 = \cdots = d_m = 0$ or of the fact that the loss functions are linear. However, use has been made of the uniformity of the processors in the proof of lemma 1.

CASE II. None of the processors $P_1, \cdots, P_r$ have any of the tasks $(i_1), \cdots, (i_r)$ on them after time t_b. The existence of $f(S)$ will be proved by proving that, for whatever value of $r \geqq 1$, there exists a schedule S' for the same data with no greater loss in which only $r - 1$ tasks are taken off processors incomplete at time t_b, which falls under Case II if $r > 1$, and in which no task is taken off any processor incomplete at any time except at times when tasks are taken off processors incomplete in S. Thus $f(S)$ can be taken as $S^{(r)}$, and the proof of theorem 4.1 will be complete.

Let P_k, $k > r$, be one of the processors having at least one of the tasks $(i_1), \cdots, (i_r)$ scheduled on it after time t_b. Suppose (i_j) is the earliest such task to be scheduled on P_k after time t_b. Then (i_j) is scheduled on P_j between time q_j and t_b where $q_j \geqq t_{b-1}$, and on P_k between times u_j and u'_j where $u'_j > u_j \geqq t_b$; otherwise it does not appear at all after t_{b-1} in S. Other than (i_j) no task appears

both on P_j after time t_{b-1} and on P_k after time t_{b-1} ; and no task appearing on either P_j or P_k after time t_{b-1} appears on any other processor after time t_b .

Let $c = t_b - q_j$, $d = u'_j - u_j$, and $e = u_j - t_b$. Let S_1 be like S except that (i_j) is not scheduled on P_j at all after time t_{b-1} but is scheduled from u_j to $u'_j + c$ on P_k , all the tasks on P_j in S after t_b are moved ahead c units of time and all the tasks on P_k in S after u'_j are delayed c units of time. Let S_2 be like S except that all the tasks scheduled on P_k after u_j in S are scheduled in the same amount and in the same order on P_j after t_b in S_2 , and vice versa. (In other words, we obtain S_2 from S by interchanging the schedule of P_j after time t_b with the schedule of P_k after time u_j.) By lemmas 3 and 4 below, S' can be taken as either S_1 or S_2 . Hence, with the proofs of those lemmas, the proof of theorem 4.1 is complete.

LEMMA 3. (A) S_1 and S_2 are both schedules for the same data, (B) no task is taken off any processor incomplete at any time in S_1 or in S_2 except at times when tasks are taken off processors incomplete in S, (C) S_1 and S_2 both fall under case II and (D) there are $r - 1$ tasks taken off processors incomplete at time t_b in S_1 and in S_2 .

PROOF: (A) follows from the construction which alters S only after time t_{b-1} , because in S (i_j) is the only task on both P_k and P_j after t_{b-1} , (i_j) is not on any processor other than P_k and P_j after t_{b-1} , no other task on P_j after t_b is on any other processor after t_{b-1} , and no other task on P_k after u'_j is on any other processor after t_b . (B), (C), and (D) follow directly from the construction.

LEMMA 4. Either the loss of S_1 is less than or equal to the loss of S, or the loss of S_2 is less than or equal to the loss of S.

PROOF: Let $X = \sum p_i$, where the sum is taken over all those i such that (i) is a task on P_j in S after time t_b . Let $Y = \sum p_i$, where the sum is taken over all those i such that (i) is a task on P_k in S after u_j . (It will include (i_j).) Let L_1 (L_2) be the loss of S minus the loss of S_1 (S_2). All tasks on P_j in S after t_b are completed on P_j after t_b ; and all tasks on P_k in S after u_j are completed on P_k after u_j . From these facts and from the construction of S_1 and S_2 from S, it follows that

$$L_1 = cX - cY = c(X - Y), \qquad \text{and}$$

$$L_2 = eY - eX = -e(X - Y).$$

Now since c is positive and e is non-negative, either L_1 or L_2 is non-negative. Q.E.D.

5. Scheduling Three Tasks on Two Processors Which Are Not Alike to Finish as Soon as Possible

When the processors are not alike the problem becomes quite difficult. It is for this reason that only the case where $m = 3$ and $n = 2$ (the simplest case of any interest) is discussed. The solution will be presented mostly without proof because of the elementary nature of the problem considered. The fact that the solution divides into so many cases for the simplest case is an indication that the general case will be quite involved.

The two processors I shall call the *main processor* and the *alternate processor*. We have three programs which I shall refer to by number as (1), (2) and (3). Let a_1 be the time it takes the main processor by itself to complete (1), and $\rho_1 a_1$ the time for the alternate processor. The programs are assumed to be uniform; thus the amount of time it would take to do a third of (1) on the main and the two-thirds on the alternate would be $\frac{1}{3} a_1 + \frac{2}{3} \rho_1 a_1$. Similarly a_2, $\rho_2 a_2$, a_3, $\rho_3 a_3$, are the other times. Note that ρ_1 is the ratio of the efficiencies of the main to the alternate in performing (i), $i = 1, 2, 3$. Assume without loss of generality that $\rho_1 \geqq \rho_2 \geqq \rho_3$. The problem is to schedule (1) (2) and (3) on, the main and alternate in such a way that (1), (2) and (3) are all completed, as soon as possible.

The complete solution is as follows. The cases are to be thought of as considered in order. I shall assume, in each case, that all preceding cases do not apply.

ONE. If $\rho_1 \geqq 1$ and $a_1 \geqq \rho_2 a_2 + \rho_3 a_3$ then schedule (1) on the main and (2) and (3) on the alternate. TWO. If $\rho_1 < 1$ and $\rho_1 a_1 \geqq a_2 + a_3$ then schedule (1) on the alternate and (2) and (3) on the main. THREE. If $\rho_2 \geqq 1$ and $a_2 \geqq \rho_1 a_1 + \rho_3 a_3$ then schedule (2) on the main and (1) and (3) on the alternate. FOUR, FIVE, and SIX, similarly.

SEVEN. If the conditions

$$\text{(I)} \quad a_2 + a_1 > \rho_3 a_3$$

$$\text{(II)} \quad \rho_2 a_2 + \rho_3 a_3 > a_1$$

and

$$\text{(III)} \quad \rho_2 a_2 < \rho_2 \rho_3 a_3 + a_1$$

hold then put (1) and a fraction of (2) equal to

$$\frac{\rho_2 a_2 + \rho_3 a_3 - a_1}{(1 + \rho_2) a_2}$$

on the main; put (3) and a fraction of (2) equal to

$$\frac{a_1 + a_2 - \rho_3 a_3}{(1 + \rho_2) a_2}$$

on the alternate.

Sketch of proof for SEVEN. Note first that by virtue of (I) and (II) both fractions are positive; since their sum is 1, each is less than 1. By virtue of (III), the sum of the times when (2) is on the two processors is less than or equal to the time each processor works in all. This fact insures the possibility of scheduling, since otherwise there would have to be a time when (2) is on both processors at the same time, which is certainly impossible. The first step is to prove that the processors work equal times, and hence neither is idle until all the work is done. This fact is easily verified by computation. Another important step is to show that any schedule which involves putting (1) on the alternate part of the time and either (2) or (3) on the main part of the time can be improved (or, at least, not made worse) by taking that part of (1)'s schedule off the al-

ternate and putting it on the main, and taking enough of (2) or (3) off the main and putting it on the alternate. The improvement is due to the assumed condition that $\rho_1 \geqq \rho_2 \geqq \rho_3$. Similarly, if (3) is on the main any part of the time, the scheduling is improved (or not made worse) by rescheduling it on the alternate.

SEVEN. If conditions (I) (II), and (III) are satisfied then the solution is as indicated above.

EIGHT. If (I) is false, then it must be that $\rho_3 > 1$ (by virtue of the failure of SIX). Try (1) on the main and splitting up both (2) and (3) between the main and alternate so that (A) the processors are used for the same amount of time t, and (B) the total time for (3) on the main and alternate is t.

NINE. If (I) is false and EIGHT fails, it must be than any (pseudo) solution satisfying (A) fails to satisfy (B), from which it can be proved that the total time assigned to (3) is greater than t. (This would mean that (3) would have to be on the main and the alternate at the same time, which is impossible.) Then the solution is to put (2) on the alternate and split (1) and (3) between the main and alternate in such a way that (A) the processors are used for the same amount of time t, and (B) the total time for (3) is t.

If SEVEN has failed because (II) is false, TEN and ELEVEN are similar to EIGHT and NINE, respectively, except that a_1 takes the role of a_3, and $\rho_1 < 1$.

TWELVE. If (III) is false and $\rho_2 \geqq 1$, schedule (3) on the alternate and split (1) and (2) between the main and alternate so that (A) the time for the processors are the same t, and (B) the total time for (2) is t.

THIRTEEN. If (III) is false and $\rho_2 < 1$, schedule (1) on the main and split up (2) and (3) on the main and alternate so that (A) the times for the processors are the same, t, and (B) the total time for (2) is t.