MIT/LCS/TM-77

TASK SCHEDULING IN THE CONTROL ROBOTICS ENVIRONMENT

ALOYSIUS KA-LAU MOK

SEPTEMBER 1976

MIT/LCS/TM-77

Task Scheduling in the Control Robotics Environment

by

Aloysius Ka-Lau Mok

September 1976

Massachusetts Institute of Technology
Laboratory for Computer Science

(formerly Project MAC)

Cambridge Massachusetts 02139

TASK SCHEDULING IN THE CONTROL ROBOTICS ENVIRONEMNT

by

Aloysius Ka-Lau Mok

Submitted to the Department of Electrical Engineering and Computer Science on August 6, 1976 in partial fulfillment of the requirements for the Degree of

MASTER OF SCIENCE

ABSTRACT

Scheduling problems involved in Control Robotics, a software approach to control engineering are studied. The capability of a multiprocessor system to handle tasks with hard, real-time deadlines is investigated according to whether complete or partial _a priori_ knowledge of the deadlines, computation times and frequencies of occurence of individual tasks is available. A model of preemptive scheduling, the "scheduling game" is introduced to explore mathematical relationships for different scheduling situations. A necessary and sufficient condition for scheduling tasks with simultaneous requests or deadlines is derived. Partial solutions and the difficulties involved in scheduling tasks with distributed requests are discussed. It is shown that in the most general case, there is no globally optimal algorithm in the absence of _a priori_ knowledge about the distribution of requests of future tasks in time.

THESIS SUPERVISOR: Michael L. Dertouzos

TITLE: Professor of Electrical Engineering and
Computer Science

ACKNOWLEDGEMENTS

The author wants to thank Professor Michael L. Dertouzos who initiated the research problems in this thesis. For his guidance and patience, the author is deeply grateful.

The author also wants to thank his colleaque Knut Nordbye of Norsk Data-Elektronikk, Norway for his valuable discussion.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 Control Robotics

The availability of inexpensive microprocessors has opened up new prospects in the field of control engineering. Dertouzos [1] proposed a programming approach : Control Robotics for the control of physical processes. In this approach, a number of continuously evaluating processes called daemons are set up to monitor sensors and deliver corrective actions to the physical processes under control. Execution of each corrective procedure is guaranteed within a real-time deadline from the time a request for service is recognized. In a typical application, daemons are created for each of the key state variables of a physical process. Each daemon has five attributes :

<daemon_name>

<condition>

<recognize_within_time>

<corrective_procedure>

<service_within_time>

Whenever the boolean variable <condition> is satisfied (recognition is by measurements on the physical process and is guaranteed within the <recognize_within_time>), the <corrective_procedure> associated with the daemon is executed within the time specified by the <service_within _time>. By a careful coordination of the daemons, various control objectives may be achieved. This software approach to control is specially aimed at combating the high costs of programming computers for specific applications, a situation which is becoming increasingly intolerable. Control Robotics attempts to provide a conceptually clean and readily applicable software/hardware interface system to users interested in automation. The fact that high level languages may be used to write the <corrective_procedures> of the daemons provides ease and extra flexibility as both function oriented (linear and non-linear) control systems which are common in classical control theory and goal oriented systems as may be required in discrete manufacturing can be readily implemented. As it is often the case that new tools precede theory, challenging problems in optimization, stability, goal(trajectory) achievement of physical processes controlled via coordination of daemons exist for the theorists. For the more practical minded users, Control Robotics provides a testbed for experimentation on control strategies. An example may be found in assembly of discrete parts by simultaneous use of several mechanical arms whereas a human operator can only guide two of the arms at the same time.

1.2 Scheduling Problems

To meet the real-time deadlines imposed by the daemons, a scheduler capable of handling multiprocessing in a hard-real-time environment is required. Geiger [2] implemented a scheduler for a single processor system, the scheduling algorithm used being one that executes at any time the task whose deadline is earliest. This Earliest Deadline has been shown to be optimal by Dertouzos [1] in the sense that if scheduling can be achieved by any algorithm, it can be achieved by the Earliest Deadline algorithm. In Geiger's implementation, the daemon <condition>s are periodically checked so that the <recognize_within_time>s are observed whenever a need for corrective action arises. This takes relatively little time and poses no serious constraint in scheduling. The major problem then is to schedule the execution of <corrective_procedure>s so that all <service_within_time>s are met. Unfortunately, the optimality of the Earliest Deadline algorithm cannot be established for the case where there are more than one processor in the system. To write a scheduler for a multiprocessor system, some theoretical problems in scheduling have first to be resolved.

Each task to be scheduled can be characterized by the parameters C_j (corresponding to the number of units of processor time required to execute the <corrective_procedure> concerned), D_j (corresponding to the <service_within_time> which is the deadline from the time the <condition> is recognized). In cases where the engineering environment requires that a request for corrective action be followed by a subsequent one for the same task only after some guaranteed minimum time, an additional parameter P_j (the minimum period between requests for the same task) can be introduced. However, this last parameter does not exist in the present formulation of Control Robotics and will not be discussed in this thesis. Throughout this thesis, time is considered to be discrete (takes on only integer values) to conform to the nature of digital computers. Scheduling is said to be achieved if all deadlines are met in the course of time. For a multiprocessor system, the following questions are of interest :

(1) Under what conditions does there exist an optimal algorithm in the sense that if scheduling is achievable, then it can be achieved by the optimal algorithm ? It has already been mentioned that for a single processor

system, the Earliest Deadline algorithm is optimal under no restriction on the parameters C and D. This result does not hold for the case of more than one processor.

(2) Given *a priori* knowledge of the parameters C and D of every task, what are the necessary and sufficient conditions for achieving scheduling if a) the requests occur simultaneously at time zero; b) the requests are distributed (but known *a priori*) over time ?

1.3 Previous Work

A review of previous work done in this area reveals relatively few results. Apart from Dertouzos's work, Liu and Layland [3] considered the case where the tasks are periodic (D = P) and have fixed computation times. For a single processor, a necessary and sufficient condition for achieving scheduling was derived, the scheduling algorithm employed being the Earliest Deadline algorithm. Again, this result cannot be generalized to the multiprocessor case. Some researchers (for example, Garey and Johnson [4], Brucker, Lenstra and Kan [5]) have tackled deadline scheduling problems under the serious restriction that no preemption is allowed i.e. a task, once started had to be continued until completion. In this case, even some relatively simple scheduling problems have been shown to be mathematically intractible (NP-complete). Since preemption is permissible in the Control Robotics environment, the unpleasant situation of having to deal with problems which possibly require exponential time to solve can be avoided and we shall concern ourselves with preemptive scheduling.

In the following chapters, we shall derive some positive and negative results on multiprocessor scheduling. A device to model the scheduling problem will first be introduced. For the multiprocessor case, it will be shown that without *a priori* knowledge of the distribution of requests in time (i.e. no information of when a task will occur), there can be no optimal algorithm such as the one in the single processor case. A necessary and sufficient condition for achieving scheduling will be derived for the case where all requests occur at time zero and where all deadlines and computation time are known *a priori*. The difficulties in scheduling in the case where requests are distributed (and known) in time will be discussed. A sufficient condition for scheduling periodic tasks on a multiprocessor system by "time slicing" will also be presented.

CHAPTER 2

PROBLEM REPRESENTATION : SCHEDULING GAME

2.1 Past Representations

In the literature, proofs of results on scheduling often make use of timing diagrams such as FIG. 2.1 and FIG. 2.2 (known as the Gantt Chart). Timing diagrams such as FIG. 2.1 are especially useful in the case of a single processor. They yield little insight as to how preemptive scheduling can be achieved when there are more than one processor involved as it is difficult to insure that no one task is being executed simultaneously on two or more processors (this is clearly not permissible). Gantt Charts are useful for non-preemptive scheduling problems but become very cubersome when preemption is allowed since individual tasks can no longer be represented by contiguous blocks. A more serious deficiency of the Gantt Chart is that it does not show the relative urgency of the tasks. To focus on this important aspect, a device to dynamically model a scheduling situation is now introduced.

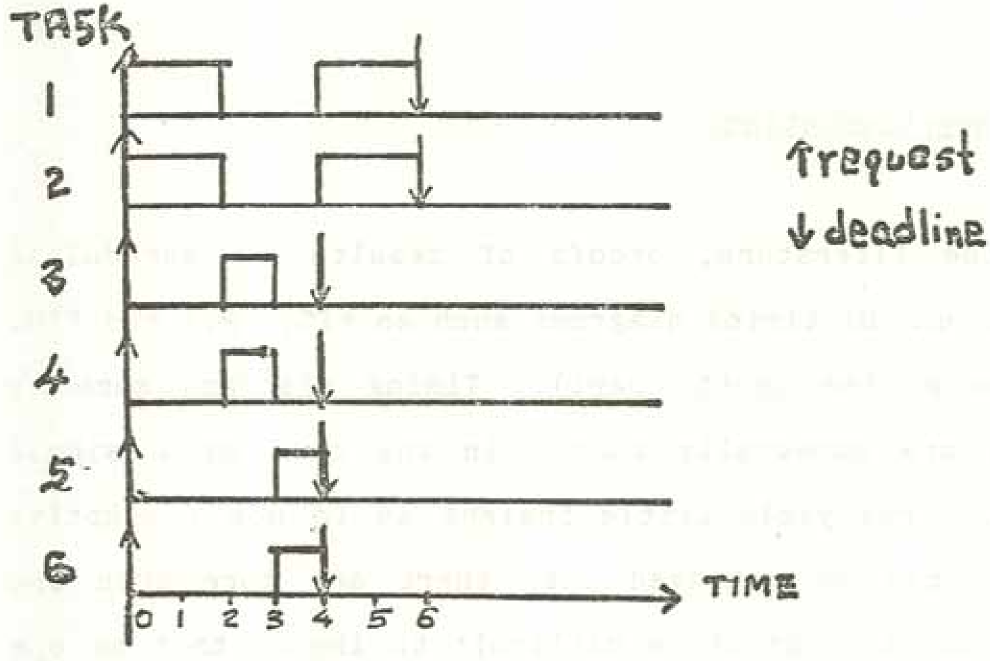


FIG. 2.1

A TIMING DIAGRAM

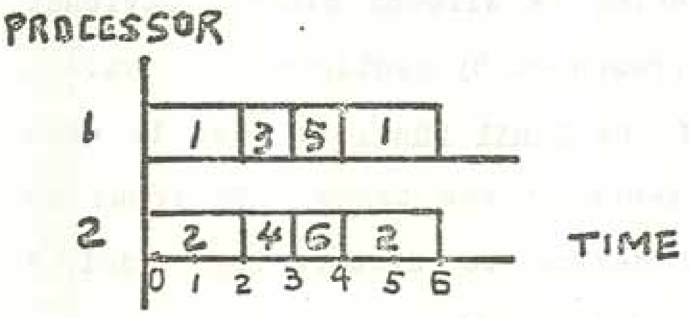


FIG. 2.2

GANTT CHART

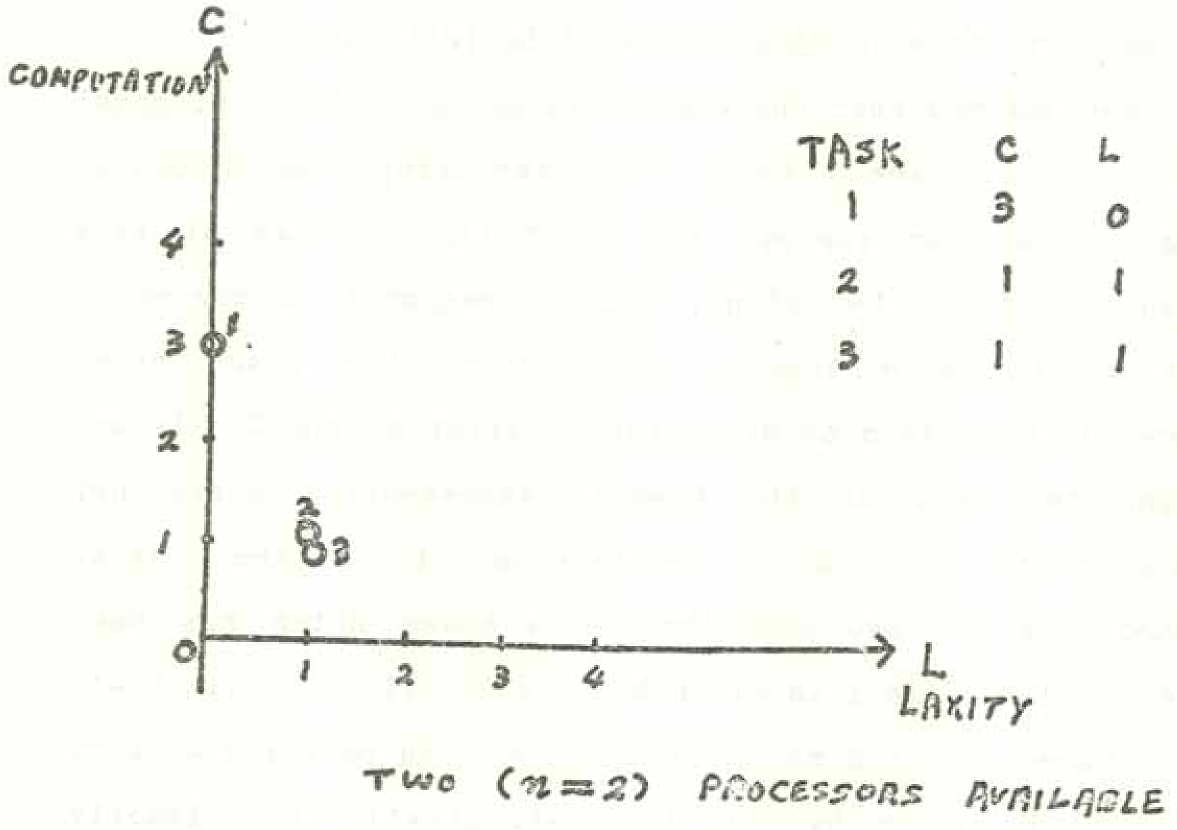
Instead of describing the activities of the processors versus time, it is more beneficial to consider the status of all the tasks at a given point in time. The status of each task which has been requested and not finished can be characterized by two parameters : the amount of computation $C(i)$ which remains to be done at time=i and the deadline $D(i)$ by which to complete it. By definition, a task requested at time i has $C(i) = C$ and $D(i) = D$ where C and D are the *a priori* (if available) computation time and deadline of the particular task. For convenience, another useful parameter $L(i)$, the *laxity* of a task is defined:

$$L(i) = D(i) - C(i) \tag{2.1}$$

In other words, the *laxity* of a task is the time left after the task is completed until its deadline assuming that the task could be executed immediately without preemption. Thus a task with zero laxity must be executed right away until it is finished. Note that a negative laxity indicates the fact that it is already too late to meet a particular deadline although it is not due yet. In this case, the magnitude of the laxity is a measure of the seriousness of failure.

2.2 The Scheduling Game Model

The scheduling situation at time i can be modelled by a configuration of tokens in the first quadrant of a Cartesian plane with vertical axis C and horizontal azis L. Each task is represented by a token. Specifically, the token representing task j with parameters $C_j(i)$ and $L_j(i)$ at time i is located at the position $L=L_j(i)$, $C=C_j(i)$ on the L-C plane. FIG. 2.3 shows an example with three tokens (tasks).



TASK	C	L	D
1	3	0	3
2	1	1	2
3	1	1	2

FIG. 2.3

EXAMPLE CONFIGURATION OF TOKENS

Note that more than one token can occupy the same position (tasks 2 and 3), that the number of processors in the system is not explicitly shown in the graphical representation (it will be incorporated into the rules for manipulating the tokens) and that whenever time dependency is not important, the index i (time is <u>discrete</u>) will be left out.

Suppose now that there are n processors at our disposal and that there are m tasks (m>n) requiring computation at time i. We can execute any n of the m tasks at this instant. On the L-C plane, this corresponds to moving at most n of the m tokens (representing the tasks being executed) one division downwards parallel to the C axis and moving the rest of the tokens (representing tasks not executed) one division to the left parallel to the L axis. Accordingly the new position for a token which has been "executed" at time i is given by L(i+1)=L(i), C(i+1)=C(i)-1. For a token that has not been executed, its position at time i+1 will be given by L(i+1)=L(i)-1, C(i+1)=C(i). Exactly which tokens are moved downwards is decided by the scheduling algorithm. Since a task with a negative laxity denotes a failure, a token being allowed to move into the second quadrant of the L-C plane means that scheduling cannot be achieved by the scheduling algorithm used. Tokens which reach the L (horizontal) axis without moving across the C (vertical) axis represent tasks whose deadlines are successfully met and can be removed from the L-C plane as soon as they reach the L axis. Thus a schedule (following some scheduling algorithm) can be simulated by a sequence of configurations of tokens on the L-C plane, each configuration representing a scheduling <u>situation</u> at a point in time. The rules for this scheduling simulation game can now be stated :

(1) The scheduler is provided with an initial configuration of m tokens in the first quadrant of a Cartesian game board (the L-C plane). This configuration models the tasks to be carried out.

(2) At each step of the game, the scheduler is allowed to move at most n (corresponding to the number of available processors) of the tokens one division downwards towards the horizontal axis. The rest of the tokens must be moved one division to the left towards the vertical axis.

3) Any token reaching the horizontal axis can be ignored or removed from the game board for the rest of the game (its deadline has been met).

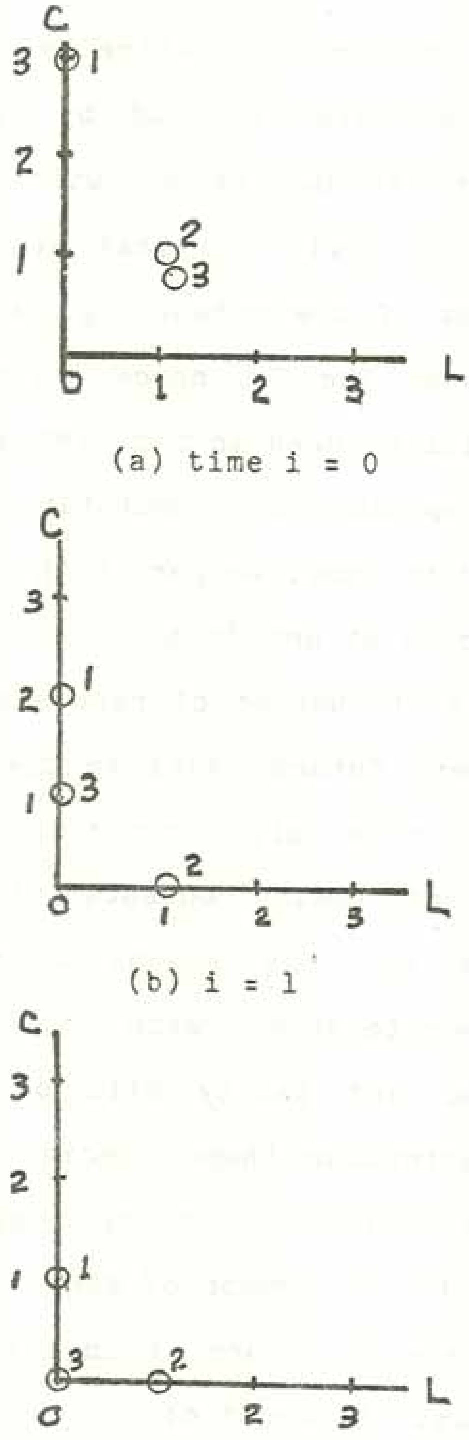
(4) The scheduler fails if any token crosses the vertical axis into the second quadrant of the Cartesian game board before reaching the horizontal axis.

(5) The scheduler wins (achieves scheduling) if in a

succession of steps all tokens are evacuated without incurring failure.

As an example, the scheduling game initialized by FIG. 2.3 is played by a scheduler who wins it. The number of processors available is two ($n = 2$). The sequence of situations are shown in FIG. 2.4.

TWO ($n=2$) PROCESSORS



(a) time i = 0

(b) i = 1

(c) i = 2

FIG. 2.4

EXAMPLE OF SCHEDULING GAME

It should be pointed out that an alternative model could use the D-C space (with coordinates defined by deadline and remaining computation) or the D-L space (with coordinates defined by deadline and laxity). In that case, a slight modification of the rules of the scheduling game would be required. For our purposes the C-L space seems to be the most convenient and so will be used in this thesis.

In the event that requests for computation (occurences of tasks) are distributed in time, we permit the addition of new tokens to the game board at any step of the game. If _a priori_ knowledge of the distribution of requests in time is available, we can represent future tasks by special tokens which are constrainted to move only horizontally in the L-C plane until the time that their requests will actually occur. From that time on, the tokens which will be positioned at their proper locations with respect to their initial computation time and laxity will be treated as ordinary tokens. To distinguish these special tokens from the rest, we shall write inside each token a _count_ which is a positive integer equal to the number of steps the token is constrainted to move laterally before it is allowed to move downwards. In other words, the count of a token is the time from the present moment when the request will actually occur in the future and it will be decremented by one at each step until it is zero. For simplicity, tokens with zero count (i.e. the ordinary tokens) will not have zeros written inside them. (This representation of future requests is convenient for graphic illustrations and for simulation.)

The above model provides us with a twofold benefit. First, it keeps track of the status of each task at a specific point in time and provides us with a clearer view of how the execution of one task may affect the completion of another whereas timing diagrams show complete schedules but not the way they can be derived. Secondly, the two dimensional representation of a scheduling situation provides us with a natural device to do accounting on the parameters of the tasks. To this effect, define

$$M(k) = |\{J_j : D_j = k\}| \qquad (2.2)$$

$$N(k) = |\{J_j : L_j = k\}| \qquad (2.3)$$

The graphical meaning of these two variables are apparent in FIG. 2.5. They will be used in an algebraic identity later.

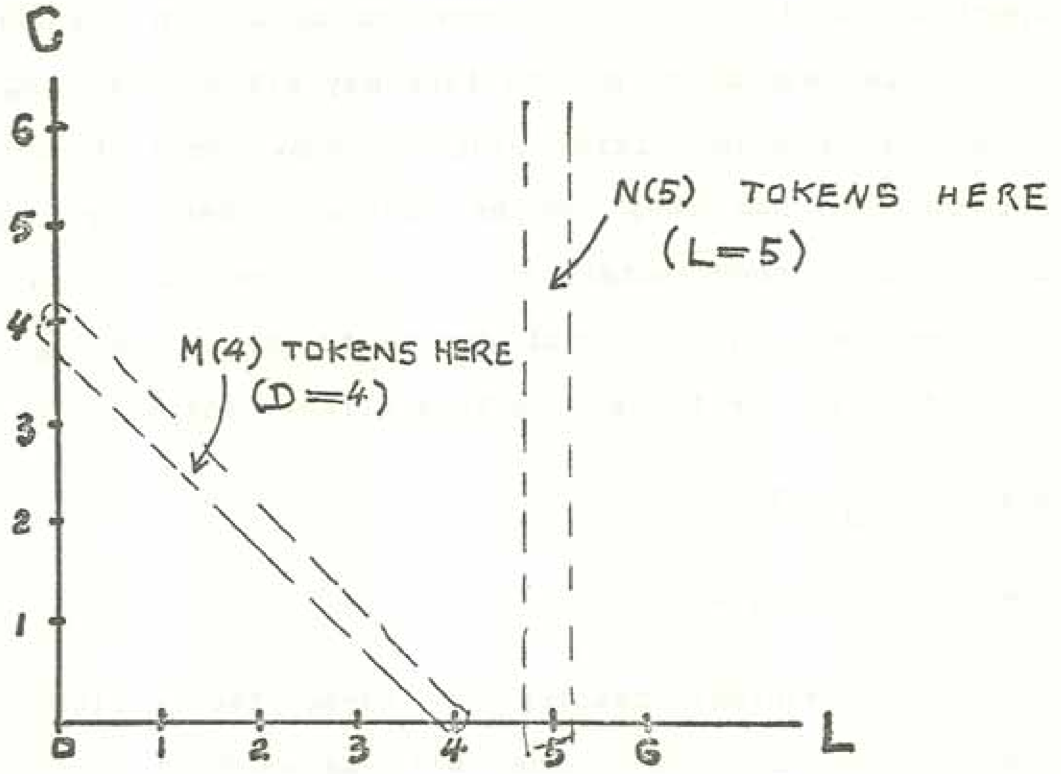


FIG. 2.5

GRAPHICAL MEANING OF M(k) AND N(k)

CHAPTER 3

SEARCH FOR AN OPTIMAL ALGORITHM

3.1 The Earliest Deadline Algorithm

A little thought shows that the capability of a n-processor system (in terms of meeting deadlines) cannot exceed that of a single processor with a speed n times as fast. This is the case because we can always simulate the performance of the n-processor system by having the fast processor execute in each time unit what each of the slow processors accomplishes in that time interval. The converse, however, is not necessarily true since a task cannot be executed simultaneously on more than one processor at any time. This is a serious limitation as a consequence of which few of the consequences in single processor scheduling can be generalized to the multiprocessor case. As an example, the Earliest Deadline algorithm which is optimal in the single processor case (in the sense that this algorithm achieves scheduling whenever scheduling is possible) is no longer optimal when there are two processors.

The optimality of the Earliest Deadline algorithm depends on the fact that for a single processor system, it is always possible to transform a feasible schedule to one which follows the <u>Earliest</u> <u>Deadline</u> algorithm. This is so because if at any time the processor executes some task other than the one which has the closest deadline, then it is possible to interchange the order of execution of these two tasks i.e. execute the task with the closest deadline first and make up for the loss of one unit of processor time of the sacrificed task by executing it at a later time when the task with the closest deadline would have been executed. Since the sacrificed task has a more distant deadline, making up for its one unit of lost processor time before the closest deadline certainly does not violate its own deadline. This is illustrated by the timing diagrams of FIG. 3.1. In the schedule of FIG. 3.1(a), task 1 which has a deadline at t = 3 is executed at t = 0 instead of task 2 which has a deadline at t = 2. The alternative schedule which follows the <u>Earliest</u> <u>Deadline</u> algorithm is shown in FIG. 3.1(b). The swapping of the shaded blocks does not cause task 1 to fail.

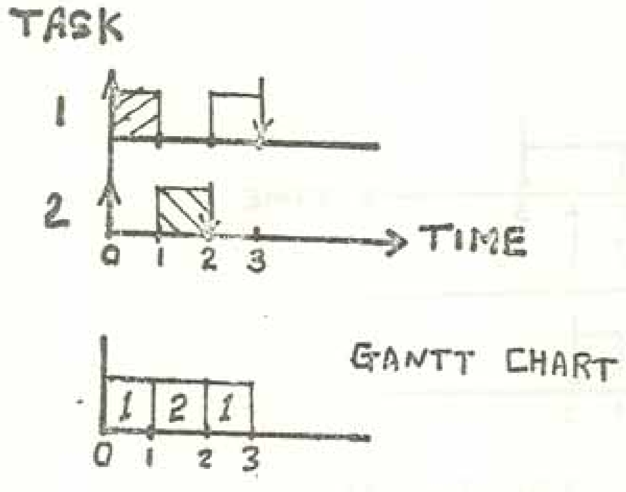


FIG. 3.1 (a)

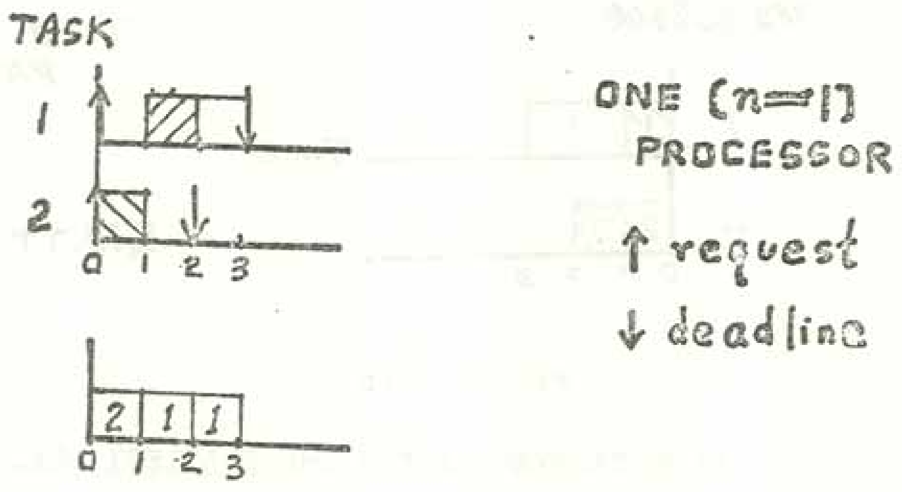


FIG. 3.1 (b)

ILLUSTRATION OF EARLIEST DEADLINE ALGORITHM

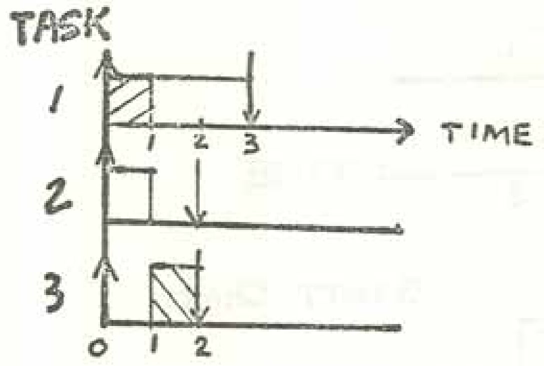


FIG. 3.2 (a)

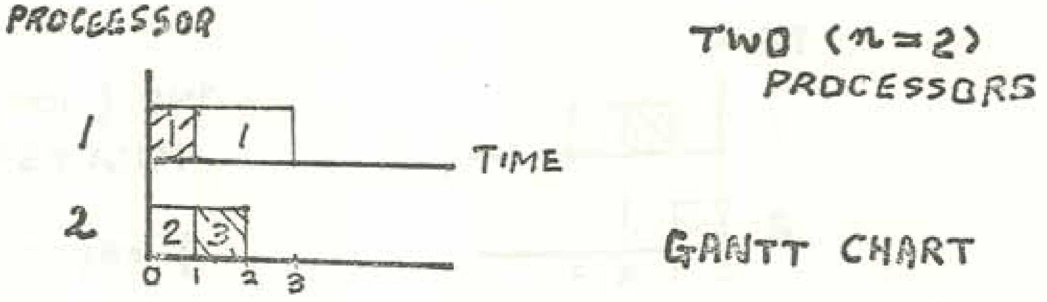


FIG. 3.2 (b)

COUNTER-EXAMPLE FOR THE <u>EARLIEST</u> <u>DEADLINE</u> ALGORITHM

Now consider the scheduling situation shown in FIG. 3.2(a) (which is the timing diagram representation of FIG. 2.3). There are two processors available. Tasks 2 and 3 have deadlines two time units away whereas task 1 has a deadline of three time units. According to the _Earliest Deadline_ algorithm tasks 2 and 3 should be executed first. Token 3 immediately crosses into the second quadrant of the L-C plane where L is negative, indicating a failure to meet the deadline of task 3. However, scheduling is possible if task 1 is executed first, as FIG. 2.4 illustrates. The timing diagram of FIG. 3.2 (b) shows why the _Earliest Deadline_ algorithm fails : the shaded blocks can no longer be swapped because task 1 would then be executed simultaneously on both processors. Thus the _Earliest Deadline_ algorithm is not optimal in the multiprocessor case.

3.2 _The Non-existence of Optimal Algorithms_

The optimality of the _Earliest Deadline_ algorithm in the case of a single processor is a very desirable property since it is a "best effort" algorithm and allows the system to degrade gracefully. Furthermore, the _Earliest Deadline_ algorithm is driven by deadlines alone and _does not require_ a priori _knowledge of the computation time of the tasks or even the distribution of the requests in time_. So it would be especially pleasing if we could find an algorithm which has this property. Unfortunately, such an algorithm does not exist. In particular, if we do not have _a priori_ knowledge of any one set of the following parameters : (i)deadlines, (ii)computation time or (iii)the distribution of requests in time, then for _any_ algorithm one might propose, one can always find a set of tasks which cannot be scheduled by the proposed algorithm but which can be scheduled by an appropriate algorithm.

This can be established by "adversary arguments" as follows.

Lemma 3.1 There can exist no optimal scheduling algorithm if the computation time of the tasks are not known _a priori_.

Proof Consider the scheduling situation in FIG. 3.3. There are three tokens on the game board and the scheduler is allowed to move two tokens downwards at each step (i.e. there are two processors). All three tokens represent tasks with a deadline of two time units. Furthermore, two of the tasks need one unit of computation time and their corresponding tokens are marked by triangles. The other task needs two units of computation time and the corresponding token is marked by a circle. In order to achieve scheduling, the task with two units of computation time must be immediately executed i.e. the circular token must be moved downwards at once. Since no information about computation time is available, all three tasks appear to be the same to the scheduler. That is to say, the scheduler cannot distinguish between the shape of the tokens. Thus if a scheduling algorithm executes task j first, we can always arrange our example so that task j is represented by a triangular token, in which case the proposed algorithm will fail while some other algorithm will succeed (namely, the one that moves the circular token down first).

Lemma 3.2 There can be no optimal scheduling algorithm if the deadlines of the tasks are not known *a priori*.

Proof By an entirely similar argument as the one above, using the situation shown in FIG. 3.4.

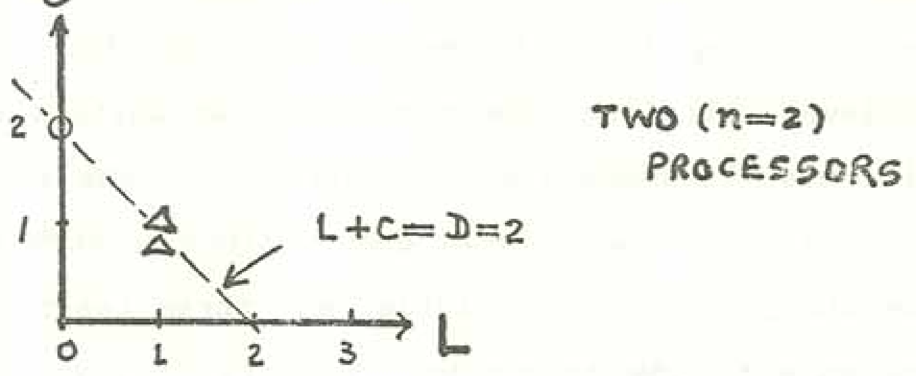


FIG. 3.3

SCHEDULING TASKS WITH UNKNOWN COMPUTATION TIMES

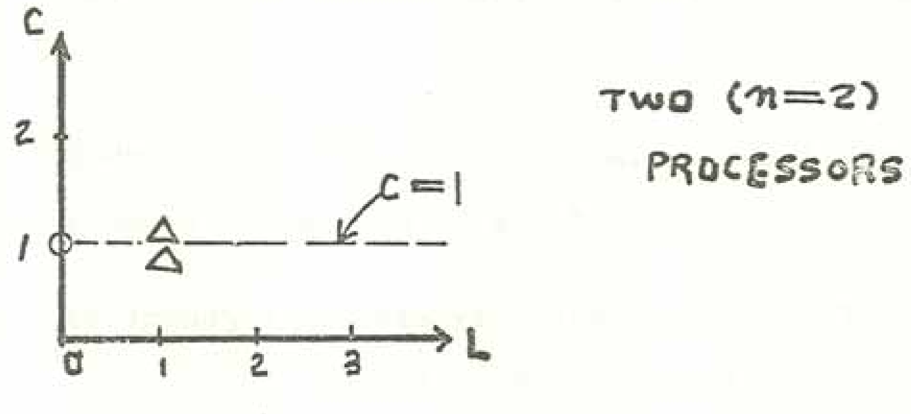


FIG. 3.4

SCHEDULING TASKS WITH UNKNOWN DEADLINES

Lemma 3.3 There exists no optimal scheduling algorithm if the distribution of the requests in time is not known _a priori_.

Proof Consider the situation shown in FIG. 3.5(a). There are three tokens (marked A, B and C) and the scheduler is allowed to move there are two of them downward at each step (i.e. there are two processors). Since B has zero laxity, it must be moved down at once. Thus depending on the scheduler's decision, there are three cases :

Case 1 A and B are moved downwards at i = 0. In this case, C will be on the vertical axis at i = 1. Consider the situation shown in FIG. 3.5(b). Two extra tokens (marked D and E) are introduced at i = 1. This is permissible since there is no _a priori_ knowledge of the distribution of the requests in time, and D and E represent tasks whose requests occur at i = 1. Consequently, there are three tokens (C,D and E) on the vertical axis and since the scheduler can evacuate at most two of them, scheduling cannot be achieved. However, if the scheduler had moved B and C down instead of A and B at i = 0, the situation at i = 1 would have looked like the one shown in FIG. 3.5(c). Scheduling could then be achieved by evacuating D and E at i = 1 and A subsequently.

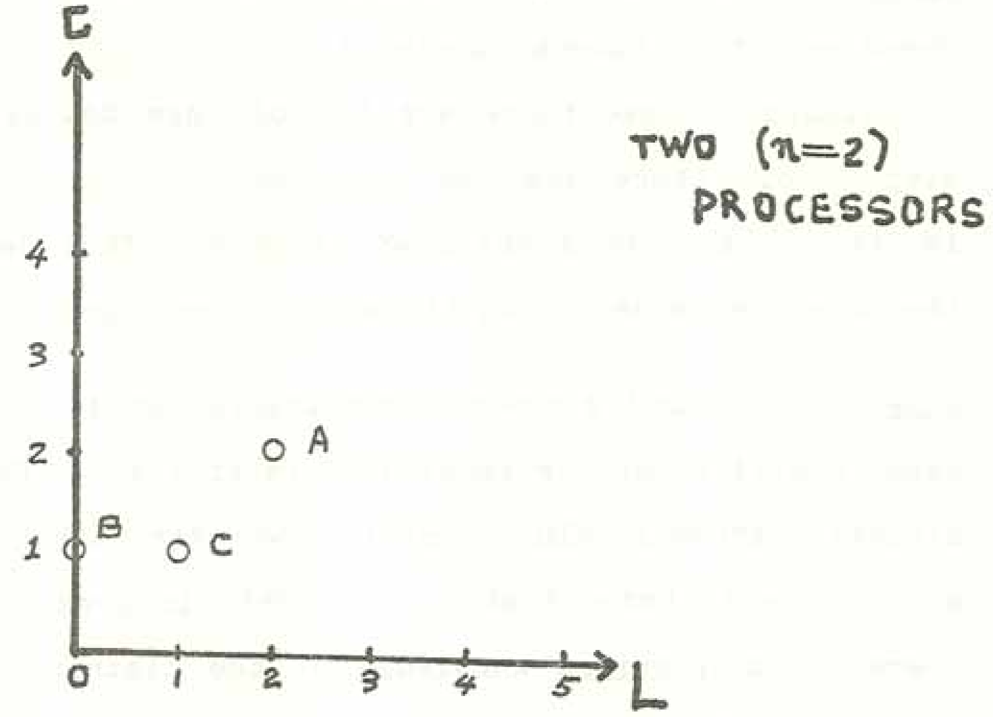


FIG. 3.5 (a)

SCHEDULING FOR UNKNOWN FUTURE REQUESTS

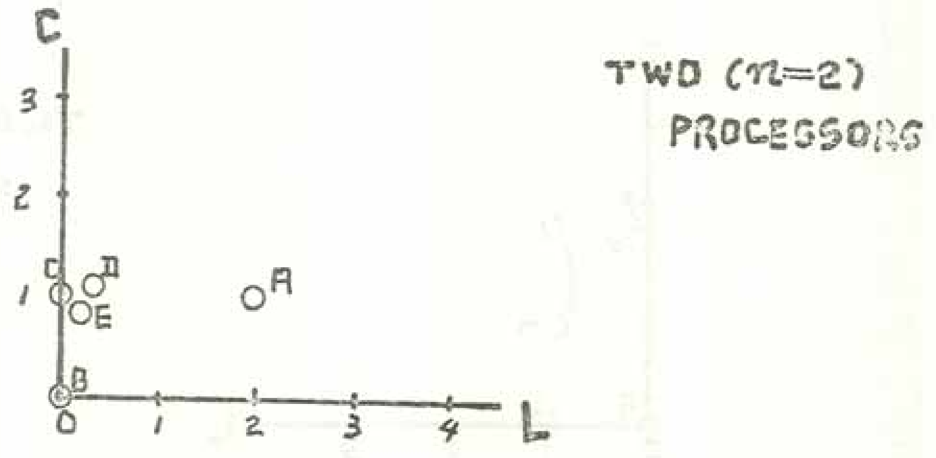


FIG. 3.5 (b)

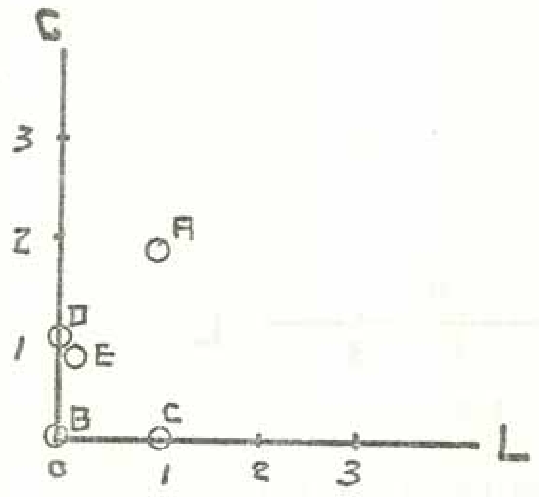
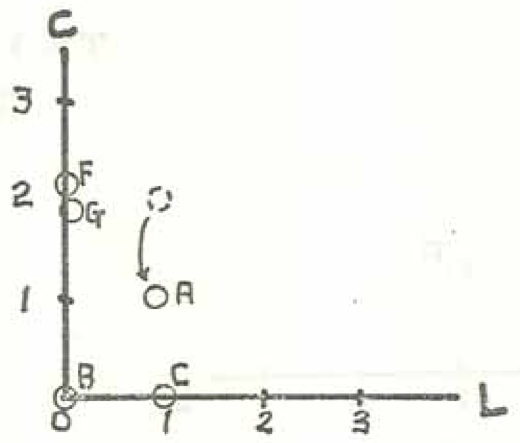


FIG. 3.5 (c)

CASE ONE OF LEMMA 3.3



TWO ($n=2$)

PROCESSORS

FIG. 3.5 (d)

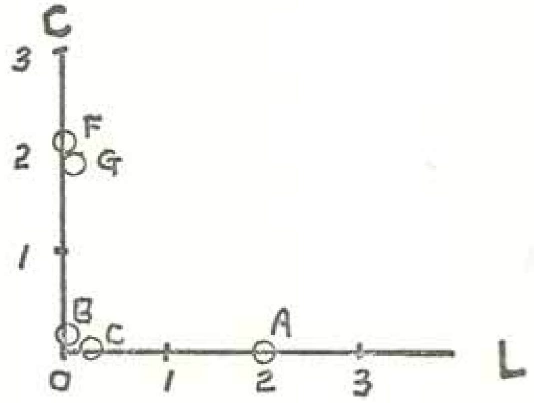


FIG. 3.5 (e)

CASE TWO OF LEMMA 3.3

<u>Case 2</u> B and C are moved downwards at i = 0. In this case, A will be at position (L=1,C=2) at i = 1. Consider the situation shown in FIG. 3.5(d). Two extra tokens (marked F and G) are introduced at i = 2. Since F and G are on the vertical axis, they have to be moved downwards in the next two time units. Hence it is impossible to evacuate A before it crosses over the vertical axis and the scheduler fails. However, if A and B had been moved downwards at i = 0 instead of B and C, the situation would have been like the one shown in FIG. 3.5(e) at i = 2. This is so because both A and C could have been evacuated at i = 1. Thus the scheduler fails because of a wrong decision made at i = 0.
<u>Case 3</u> Only B is moved downwards at i = 0. In this case, extra tokens introduced at i = 1 such as D and E in FIG. 3.5(b) will cause the scheduler to fail unnecessarily.

Thus for any decision that a proposed optimal scheduler makes at i = 0, there is a set of future requests (D and E or F and G) which will cause that scheduler to fail, whereas if a different scheduler had been used at i = 0, then scheduling would have been achieved. In other words, no optimal scheduling algorithm is possible.

Some observations can be made about the above Lemma. Focusing on FIG. 3.5(a), we can see that there are basically two choices that a scheduler can make : Either execute the shorter tasks (B and C) and run the risk of letting one processor idle at i = 1 which is what happens in FIG. 3.5(d); or execute the longer task (A) first and run the risk that short but urgent tasks may occur in the near future (at i = 1 in FIG. 3.5(b)). The former case corresponds to a strategy of minimizing the number of unfinished tasks as soon as possible and the latter a strategy of minimizing the overall completion time of the schedule. These are conflicting objectives and will occur again in some later discussion.

Combining Lemmas 3.1, 3.2 and 3.3, we have the following :

Theorem 3.1

For two or more processors, there exists no optimal algorithm if any one or more of the following parameter sets is *a priori* unknown, (i)deadlines, (ii)computation times and (iii)the distribution of requests of the tasks.

It should be noted that although the theorem has been proved for the case of two processors, the extension to any number of (more than one) processors is straight forward.

Also, no restriction has been put on the parameters of the tasks (their deadlines, computation time and the time distribution of the requests) in arriving at the above theorem. If restrictions are applied, optimal algorithm(s) may exist such as is in the single processor case. For example, if all tasks have unit computation time, then the _Earliest_ _Deadline_ algorithm is again optimal. Under this restriction, the proof that the _Earliest_ _Deadline_ algorithm is optimal in the case of a single processor applies even if there are more than one processor. This is so because the _swapping_ of tasks that is necessary in the proof will not result in the same task being executed simultaneously on two or more processors at a later time. Admittedly this is an unrealistic restriction. On the other hand, there is no reason why less severe restrictions should not exist. In fact, if we define the _utilization_ _factor_

$$U = \sum_j (C_j / D_j) \tag{3.1}$$

where the j^th^ task has parameters C_j, D_j and if we assume that any task may repeat itself in the course of time, then a possible restriction is that U be less than or equal to the number of processors n. The author has not been able to find a time distribution of requests from a set of tasks (which may be repeated any time after their immediate deadlines) under the above restriction which can be used to prove Lemma 3.3. This appears reasonable because in the worst case where all tasks repeat themselves continuously, the amount of computation these periodic tasks require does not exceed the computation power the multiprocessor system can at most put out.

To see this, compute the total computation time required in a period of $P = D_1*D_2*D_3*...*D_m$.

$C_{total} = C_1*P/D_1 + C_2*P/D_2 + ... + C_m*P/D_m = P * (C_1/D_1 + C_2/D_2 + ... + C_m/D_m) = P * U$

In P units of time a total of n*P units of processor time is available. Therefore,

$P*U \leq n*P$ and so $\sum_j (C_j/D_j) \leq n$

CHAPTER 4

THE LEAST LAXITY ALGORITHM

4.1 Simultaneous Requests

Since it is impossible to find an optimal algorithm if there is no a priori knowledge of the parameters of the tasks which may be requested, the next question is naturally what scheduling algorithms exist for the case where all information about the tasks is available (we can imagine the existence of an oracle for the scheduler). If the distribution of requests in time spans only a finite interval, then it is obviously possible to determine whether it is possible to achieve scheduling over this period of time. (Exhaustive search, if for no other method will yield a solution whenever one exists because the number of schedules the multiprocessor system can have over a finite interval is finite, although possibly large). The important question then becomes whether there exists any efficient algorithm for achieving scheduling. From a practical point of view, such an algorithm is desirable since we usually have a good idea of the approximate size of these parameters, either statistically or from direct measurement. If the distribution of requests extends indefinitely into the future, then an interesting question to ask is how far along into the future does the scheduler have to ask the oracle about the requests that may occur so that an appropriate scheduling decision can be made at the present moment. It seems that the answer to the second question hinges on that to the first question. So the first question will be tackled. As a first attempt, consider the special case where all requests occur at $i = 0$.

Some notations are in order. For the ease of counting, the L-C plane is divided into three regions as illustrated in FIG. 4.1. For every natural number k, all the tokens on a game board can be partitioned into three distinct sets :

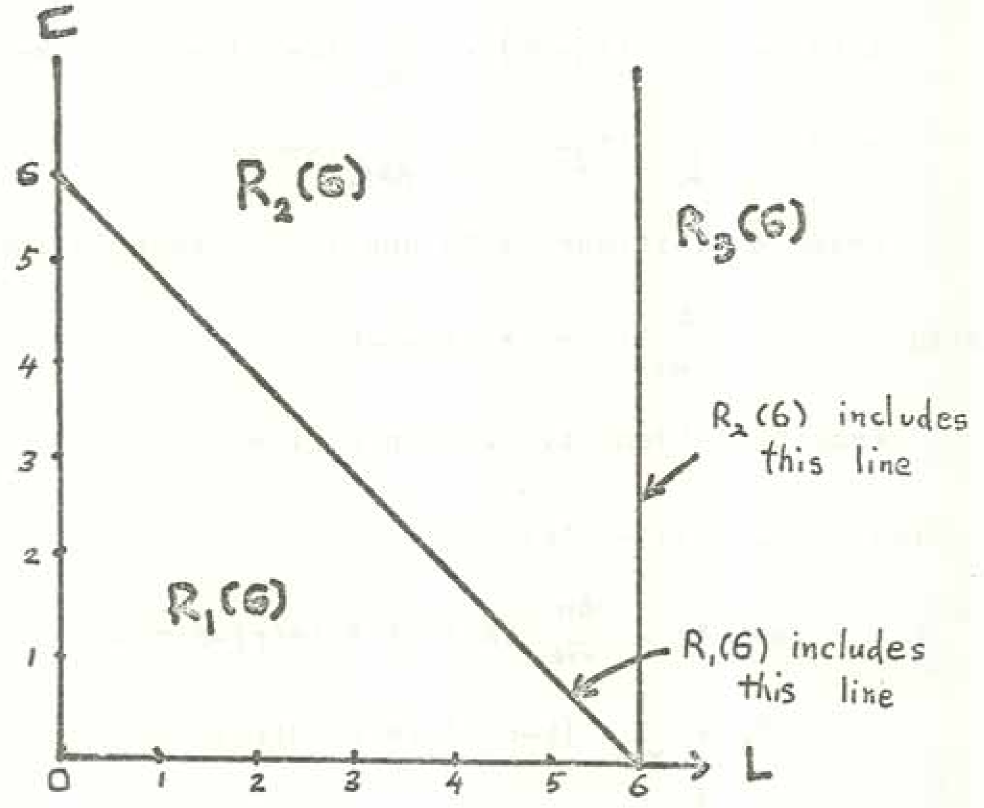
$$\begin{aligned} R_1(k) &= \{J_j : D_j \leq k\} \\ R_2(k) &= \{J_j : L_j \leq k \text{ and } D_j > k\} \qquad (4.1) \\ R_3(k) &= \{J_j : L_j > k\} \end{aligned}$$

Next define for every positive integer k, the function

$$F(k) = k*n - \sum_{R_1(k)} C_j - \sum_{R_2(k)} (k-L_j) \qquad k>0 \qquad (4.2)$$

The function F(k)* defined above is a measure of the "surplus" computing power of the system in the next k time units as will be shown later. For computational purposes, it is convenient to express F(k) in terms of tasks with the same deadline or the same laxity.

FIG. 4.1



$$F(k) \triangleq n \cdot k - \sum_{R_1(k)} C_j - \sum_{R_2(k)} (k - L_j)$$

DEFINITION OF THE FUNCTION F(k)

---

* The significance of F(k) was discovered by Knut Nordbye. Theorem 4.1 is a more elegant result arrived at after discussion between Nordbye and the author.

$$\begin{aligned}
F(k) &= k*n - \sum_{R_1(k)} C_j - \sum_{R_2(k)} (k-L_j) \\
&= k*n - \sum_{R_1} (D_j-L_j) - \sum_{R_2} (k-L_j) \\
&= k*n - \sum_{R_1} (D_j- k) - \sum_{R_2} (k-L_j) - \sum_{R_1} (k-L_j) \\
&= k*n - \sum_{R_1} (D_j- k) - \sum_{R_1 \cup R_2} (k-L_j)
\end{aligned}$$

Using definitions (2.2) and (2.3), this is the same as

$$F(k) = k*n - \sum_{r=0}^{k} (k-r) * (N(r)-M(r)) \tag{4.3}$$

From this identity, we can derive

$$\begin{aligned}
F(k) &= F(k + 1) - F(k) \\
&= (k+1)*n - \sum_{r=0}^{k+1} (k+1-r) * (N(r)-M(r)) \\
&\quad - k*n + \sum_{r=0}^{k} (k-r) * (N(r)-M(r)) \\
&= n - \sum_{r=0}^{k+1} (N(r)-M(r))
\end{aligned} \tag{4.4}$$

For a given configuration of a finite number of tokens on a game board, $N(r)$ is zero for all $r > L_{max}$ where L_{max} is the maximum of the laxity of all the tokens. Thus $F(k) > 0$ for $k > L_{max}$. Thus it can be concluded that

<u>Lemma 4.1</u> For a given configuration of a finite number of tokens, $F(k) \geq 0$ for all k if $F(k) \geq 0$ for $0 < k \leq L_{max}$.

In fact F(k) ~ k*n for large enough k. This is to be expected since if F(k) is a measure of the "surplus" computation power of a n-processor system, then the computation power left over from processing a finite number of tasks should be proportional to k*n.

Using the function F(k), we can now prove a necessary and sufficient condition for achieving scheduling when all requests occur at time i = 0 (i.e. all tokens have zero count). To emphasize that F(k) is really a function of time, its time dependence will be explicitly written out, whenever necessary as F(k,i) which will be understood as the value of F(k) computed on the scheduling situation at time i. We now prove

<u>Lemma 4.2</u> A necessary condition for achieving scheduling when the requests of all the tasks occur at time i = 0 is that for all k > 0, $F(k,0) \geq 0$

<u>Proof</u> Referring to FIG. 4.2, it can be seen that any token in R_1 must be evacuated in k steps since all of them have a deadline of k time units. This requires $\Sigma\ C_j$ units of processor time for all the tokens in R_1. For a token (with a laxity of L_j) in R_2, it can be allowed to move

laterally towards the vertical axis for at most L_j divisions in the next k steps before it crosses over into the second quadrant. For the rest of the time, it must be moved downwards until it is evacuated. Since it is in R_2, it needs $C_j = D_j - L_j > k - L_j$ units of processor time. Thus in the next k steps, the token must be moved at least $k-L_j$ divisions downwards. The minimum amount of processor time all the tokens in R_2 needs is thus $\sum_{R_2} (k-L_j)$. In k steps, the maximum amount of processor time we can get from a n-processor system is k*n. Hence, to satisfy the minimum computation requirements of the tasks in R_1 and R_2, we must have

$$F(k) = k*n - \sum_{R_1} C_j - \sum_{R_2} (k-L_j) \geq 0 \qquad (4.5)$$

Since this must hold for all $k > 0$, the Lemma is proved.

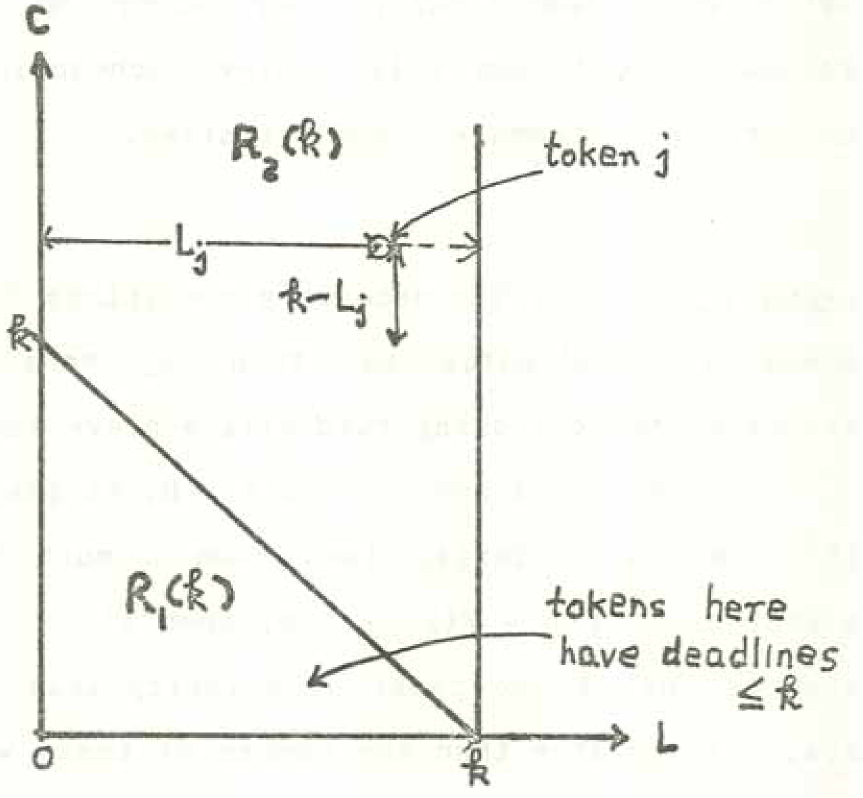


FIG. 4.2

NECESSITY OF THE CONDITION $F(k) \geq 0$

The interpretation of $F(k)$ as "surplus computing power in the k steps" should be clear from the proof of Lemma 4.1.

The condition of (4.5) is also sufficient for achieving scheduling. A family of algorithms exist whenever (4.5) is satisfied. Specifically, any algorithm satisfying the following condition will achieve scheduling whenever the conditions of Lemma 4.2 are satisfied.

Lemma 4.3 If the necessary conditions for scheduling of Lemma 4.2 are satisfied, then any scheduling algorithm observing the following rule will achieve scheduling :

At any step i and for all $k > 0$, at least $n - F(k,i)$ of the tasks with laxity less than k must be selected for execution. If $n - F(k,i) \leq 0$, then it is not necessary to execute any of the tasks with laxity less than k. If $n - F(k,i)$ is greater than the number of tasks with laxity less than k, then all of these tasks must be selected for execution.

In terms of the scheduling game, this requires that for all k, all or at least $n - F(k)$ (zero if $n-F(k)<0$) of the tokens left of the line $L = k$ must be moved downwards.

Proof The strategy is to first prove that any algorithm which follows the conditions of the Lemma will not fail at the immediate step. Then it will be shown that the conditions of the Lemma will again hold after the first step, thus guaranteeing that no failure will occur at any subsequent step. In other words, we have to show

(1) All the tokens on the C (vertical) axis are moved downwards at the first step.

(2) $F(k,0) \geq 0$ for all $k>0$ implies that for all $r>0$, $F(r,1) \geq 0$.

We observe that $F(1,0) \geq 0$ implies that $n \geq \sum_{D_j \neq 1} C_j + \sum_{L_j \neq 1, D_j > 1} (1-L_j)$. However, the right hand side of this last inequality is exactly the number of tokens on the C (vertical) axis and furthermore is identical to $n-F(1,0)$. Thus we are guaranteed that there are enough processors to allow the scheduler to move all the tokens on the C axis downwards and that any algorithm which satisfies the conditions of the Lemma will indeed do so. Thus proposition (1) is proved. To verify proposition (2), the strategy is to prove that for all $k>0$, $F(k,1) \geq F(k',0)$ for some k'. Since by hypothesis $F(k',0) \geq 0$ for any $k'>0$, it can then be concluded that $F(k,1) \geq 0$. Referring to FIG. 4.3, there are three cases to consider.

<u>Case</u> <u>1</u> (FIG. 4.3(a))

In this case, all the tokens in R_1 and R_2 are moved downwards at the first step. We can pick $k' = k$ i.e. we want to prove that $F(k,1) \geq F(k,0)$. It is noticed that after the first step, some tokens may have moved from R_3 to the line $L = k$ which is part of R_2 and that some tokens from R_2 may have moved to the line $L + C = k$ which is in R_1. The former contribute nothing to $F(k,1)$ since $k - L_j$ is zero in their case. The latter contribute to both $F(k,1)$ and $F(k,0)$ in equal amount since $k - L_j = C_j - 1$ in their case. All other tokens in R_1 subtract from the term C_j by moving one division downwards while all other tokens in R_2 contribute the same amount to both $F(k,1)$ and $F(k,0)$. Hence, the final tally is as follows :

$$F(k,1) = k*n - \sum_{R_1} C_j - \sum_{R_2} (k-L_j)$$

$$F(k,0) = k*n - \sum_{R_1} C_j - \sum_{R_2} (k-L_j)$$

$$\overline{\qquad 0 \qquad |R_1(k)| \qquad 0 \qquad}$$

Thus for Case 1, $F(k,1) - F(k,0) = |R_1(k)|$

<u>Case</u> <u>2</u> (FIG. 4.3(b))

In this case, some of the tokens in R_1 or R_2 are moved laterally towards the C axis at the first step. It is now necessary to pick $k' = k+1$ i.e. we want to prove $F(k,1) \geq F(k+1,0)$. Since not all the tokens in R_1 or R_2 are moved downwards at the first step, at least $n - F(k+1,0)$ tokens to the left of the line $L = k+1$ must be, by the condition of the Lemma moved downwards at the first step (all tokens have zero count and are free to move downwards). Observing that all the tokens in $R_1(k)$ and $R_2(k)$ after the first step must be from $R_1(k+1)$ and $R_2(k+1)$ and that the tokens on the line $L = k + 1$ do not contribute to $F(k+1,0)$, we can compute the difference between $F(k,1)$ and $F(k+1,0)$ by considering only the tokens in $R_1(k)$ and $R_2(k)$ after the first step. If a token is moved downwards in the first step, then it contributes positively to $F(k,1)-F(k+1,0)$ as may be easily verified. If it is moved laterally at the first step, then it does not contribute to $F(k,1)-F(k+1,0)$ at all. Since there are at least $n-F(k+1,0)$ tokens which are moved downwards in the first step, the total contribution of the tokens to $F(k,1)-F(k+1,0)$ is $n-F(k+1,0)$ at least. The final tally is as follows :

$$\begin{array}{llll}
F(k,1) & = k*n & - \sum_{R_1(k)} C_j(1) & - \sum_{R_2(k)} (k-L_j(1)) \\
F(k+1,0) & = (k+1)*n & - \sum_{R_1(k+1)} C_j(0) & - \sum_{R_2(k+1)} ((k+1)-L_j(0)) \\
\hline
 & \quad -n & \multicolumn{2}{l}{\text{at least } n - F(k+1,0)}
\end{array}$$

For Case 2, $F(k,1)-F(k+1,0) \geq -F(k+1,0)$ which implies that

$$F(k,1) \geq 0.$$

<u>Case 3</u> (FIG. 4.3(c))

In this case, all of the tokens in R_1 and R_2 are moved laterally towards the C axis at the first step. As in Case 2 above, we can pick k' = k+1 i.e. we want to prove that $F(k,1) \geq F(k+1,0)$. Furthermore, since all the tokens are moved laterally, they contribute nothing to the difference $F(k,1)-F(k+1,0)$ as may be seen from FIG. 4.3(c). Hence $F(k,1)-F(k+1,0)$ can be tallied as follows :

$$F(k,1) = k*n - \sum_{R_1(k)} C_j(1) - \sum_{R_2(k)} (k-L_j(1))$$

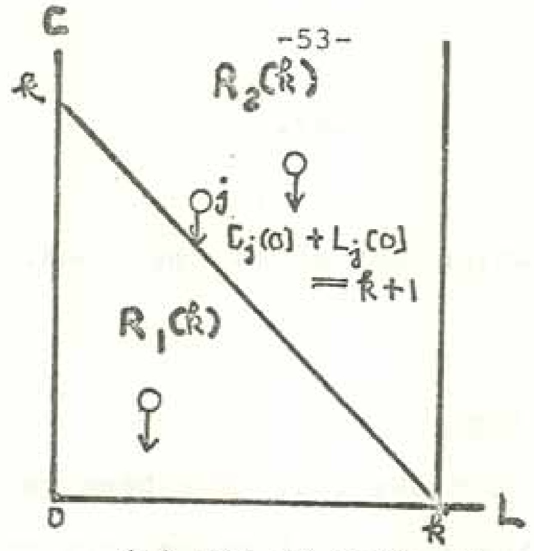
$$F(k+1,0) = (k+1)*n - \sum_{R_1(k+1)} C_j(0) - \sum_{R_2(k+1)} ((k+1)-L_j(0))$$

$$-n \qquad\qquad 0$$

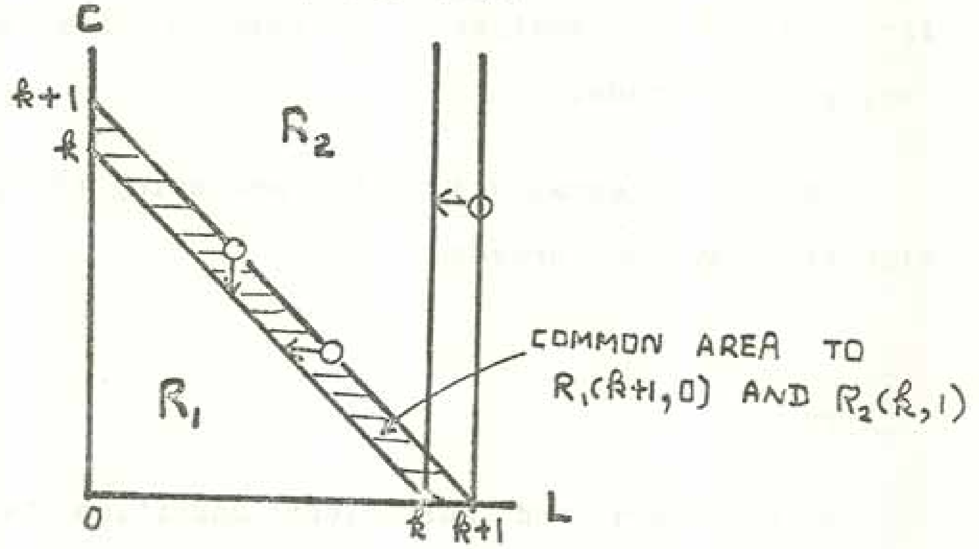
For Case 3, $F(k,1)-F(k+1,0) \geq -n$, that is

$F(k,1) \geq -(n-F(k+1,0)) \geq 0$ since $n-F(k+1,0) \leq 0$.

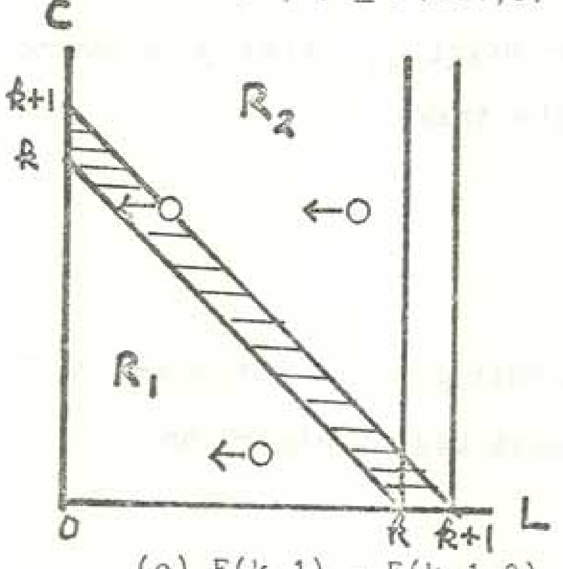
Since for every $k>0$, one of the above three cases must occur at the first step, it can be concluded that $F(k,1) \geq 0$ for all $k>0$ and the Lemma is proved.



(a) F(k,1) $\leq$ F(k,0)



(b) F(k,1) $\leq$ F(k+1,0)



(c) F(k,1) = F(k+1,0)

FIG. 4.3

SUFFICIENCY OF THE CONDITION F(k) $\geq$ 0

An algorithm which satisfies the conditions of Lemma 4.3 is the

Least Laxity Algorithm

Select among the tasks that have been requested and not yet completed, n (the number of processors available) whose laxities are the smallest. In case of ties, an arbitrary choice can be made.

Combining Lemmas 4.1 to 4.3 and using the Least Laxity algorithm, we have proved

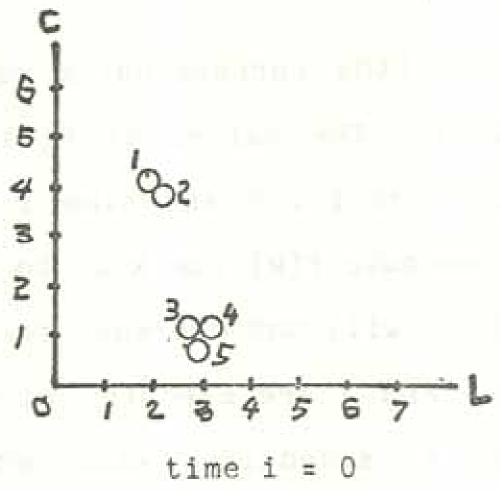
Theorem 4.1

A necessary and sufficient condition for achieving scheduling when the requests of all the tasks occur at time $i = 0$ is that for $0<k\leq L_{max}$, $F(k) \geq 0$ where L_{max} is the largest laxity of the tasks.

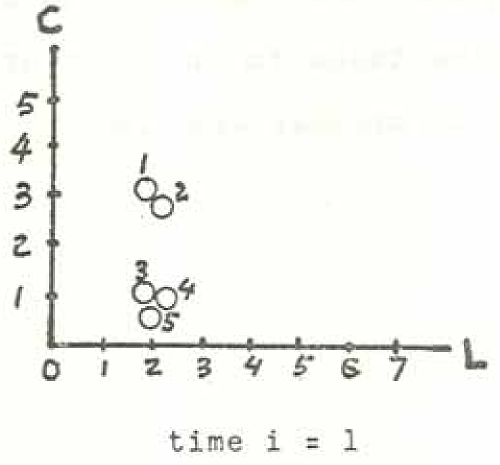
Theorem 4.2

Under the conditions of Theorem 4.1, an optimal algorithm is the Least Laxity algorithm.

An example of scheduling using the Least Laxity algorithm is shown in FIG. 4.4 (the corresponding timing diagrams are those in FIG. 2.1). The values of F(k) are computed for the situations at time i = 0 and time i = 2. Notice that it is necessary to compute F(k) for k up to L_{max} as Lemma 4.1 guarantees that F(k) will not decrease for k > L_{max}. Also note that if tasks 1 and 2 are executed at time i = 2 instead of tasks 3 and 4, scheduling will not be achieved. This is reflected by the fact that n-F(2,2) = 2-1 = 1 so that at least one of the tasks to the left of the line L = 2 must be executed for an optimal algorithm.

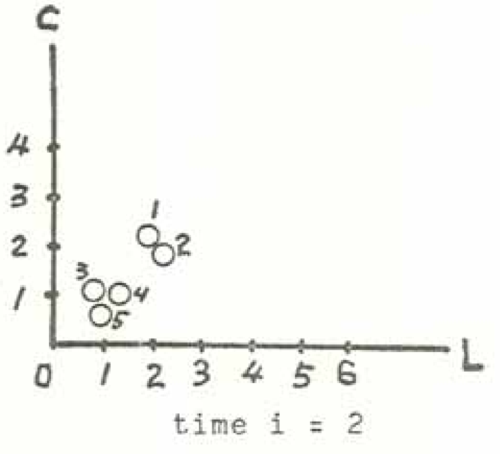


k	F(k)
1	2 − 0 = 2
2	4 − 0 = 4
3	6 − 2 = 4



TWO ($n=2$)

PROCESSORS



k	F(k)
1	2 − 0 = 2
2	4 − 3 = 1

FIG. 4.4

EXAMPLE USING THE <u>LEAST</u> <u>LAXITY</u> ALGORITHM

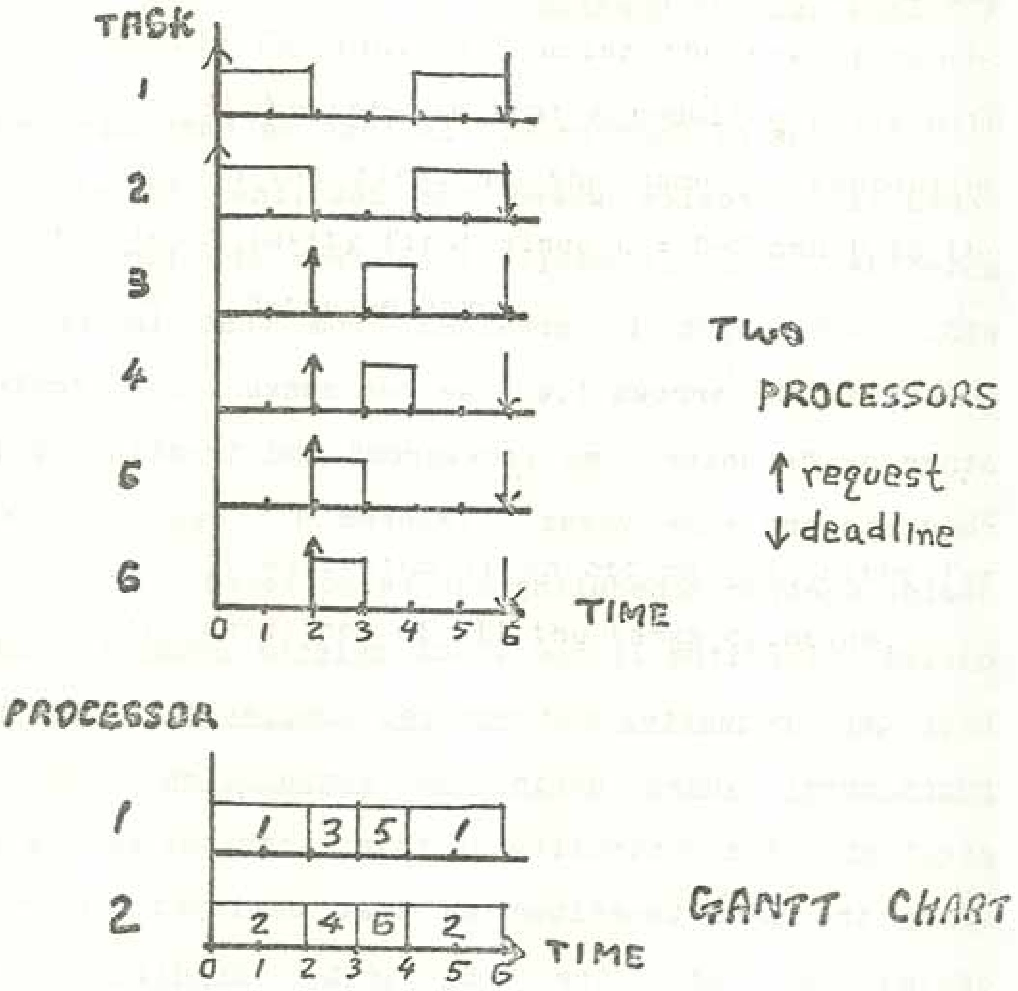


FIG. 4.5

CASE OF SIMULTANEOUS DEADLINES

4.2 Simultaneous Deadlines

A bonus of Theorem 4.1 is that we have also solved the scheduling problem where the deadlines of all the tasks coincide. This is easily seen from the timing diagram of FIG. 4.5 which is obtained from that in FIG. 2.1 by inverting the arrows i.e. we can convert one problem to the other by "running time backwards" and treating deadlines as requests and vice versa. Theorem 4.1 can then be used to decide whether scheduling can be achieved. In this case, an optimal algorithm is one which selects among the tasks that have been requested and not yet completed, n (the number of processors) whose unfinished computation times are the greatest. The optimality of this algorithm can be proved by using the same technique as that employed in proving the optimality of the Earliest Deadline algorithm. Specifically, if at any time a task is preempted by another with less computation, then it is always possible to rearrange the schedule so that the task with greater computation time is executed first. Since all tasks have the same deadline, we are guaranteed that there is a later time before the common deadline when the task with the greater computation is executed and the shorter task is not. Therefore this swapping of tasks will not result in the same task being executed simultaneously on more than one processor as is illustrated in FIG. 4.6 which is an alternative schedule for FIG. 4.5 using the new approach. An interesting point to note is that scheduling tasks with the greatest computation first is the same as scheduling tasks with the least laxity first since L = D-C and D is the same for all tasks. Hence we have

Theorem 4.3

The Least Laxity algorithm is an optimal algorithm for the case where the deadlines of all the tasks coincide.

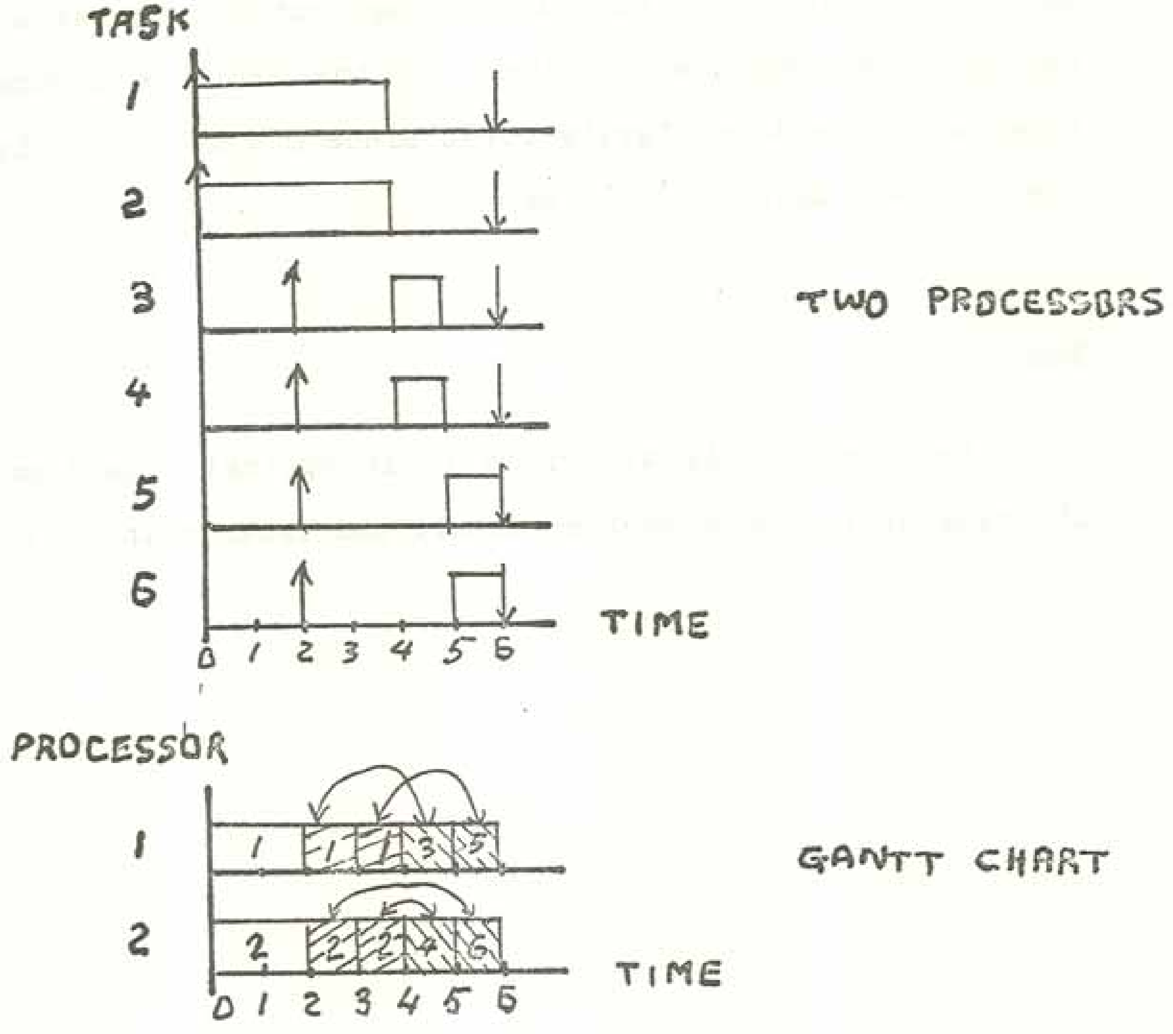


FIG. 4.6

ALTERNATIVE SCHEDULE FOR THE CASE OF SIMULTANEOUS DEADLINE

We could have proved the optimality of the Least Laxity algorithm in a more straight forward manner by the above approach. The merit of theorem 4.1 is however, in completely characterizing the class of optimal algorithms by using the function F(k). This allows us to minimize the number of preemptions which are really needed. The Least Laxity algorithm is also a very efficient algorithm. For m tasks, it takes m subtractions to compute the laxities from the deadlines and computation times and O(m*logm) comparisons to order them. So the computation requirement is approximately linear (O(m**2) at most) at each step and so for a complete schedule, it takes $O(m*D_{max})$ where D_{max} is the maximum of the deadlines (it takes fewer comparisons to order the laxities after the first step). This compares well with an exhaustive search which takes exponential time with respect to the task parameters. We now move on to the general case where requests may be distributed (but known a priori) in time.

CHAPTER 5
DISTRIBUTED REQUESTS

5.1 Limitations of the Least Laxity Algorithm

The function F(k) defined in the last chapter is an efficient measure of the surplus computing power of a multiprocessor system when the requests of all tasks occur at time = 0. It was proved that any algorithm which executes all or at least n - F(k) (zero if n-F(k)<0) of the tasks with laxity less than k is optimal. The Least Laxity algorithm is one which meets this criterion and is the "most optimum" in the following sense :

Lemma 5.1 If the Least Laxity algorithm is applied at time i=0, then the resulting situation at i=1 is such that for all k>0, F(k,1) is maximal, i.e. no other algorithm can yield a larger F(k,1) for any k>0.

Proof Since no other algorithm can move more tokens downwards left of any vertical line in the L-C plane than the Least Laxity algorithm, it is clear from the proof of Lemma 4.2 that the resulting F(k,1) have maximal values.

With this observation, we can prove

Theorem 5.1

The Least Laxity algorithm is optimal for scheduling tasks

If the requests of all tasks occur at time=0 or time=1, then the Least Laxity algorithm is optimal (in the usual sense).

Proof

We have proved that the Least Laxity algorithm is optimal if the requests of all tasks occur at time=0. Suppose there are some requests which occur at time=1 such that the Least Laxity algorithm fails subsequently, then by Lemma 4.2, there must be some k' such that F(k',1)<0. But by Lemma 5.1, F(k',1) is maximal. Therefore, for any other algorithm the resulting F(k'1) will also be negative and by Lemma 4.2, a failure will occur.

Thus the Least Laxity algorithm anticipates one step into the future. However, this nice property does not hold for all future steps as the scheduling situation in FIG. 5.1 domonstrates. Here the square tokens are used to represent future requests and are confined to move horizontally for the number of steps written inside the squares themselves.

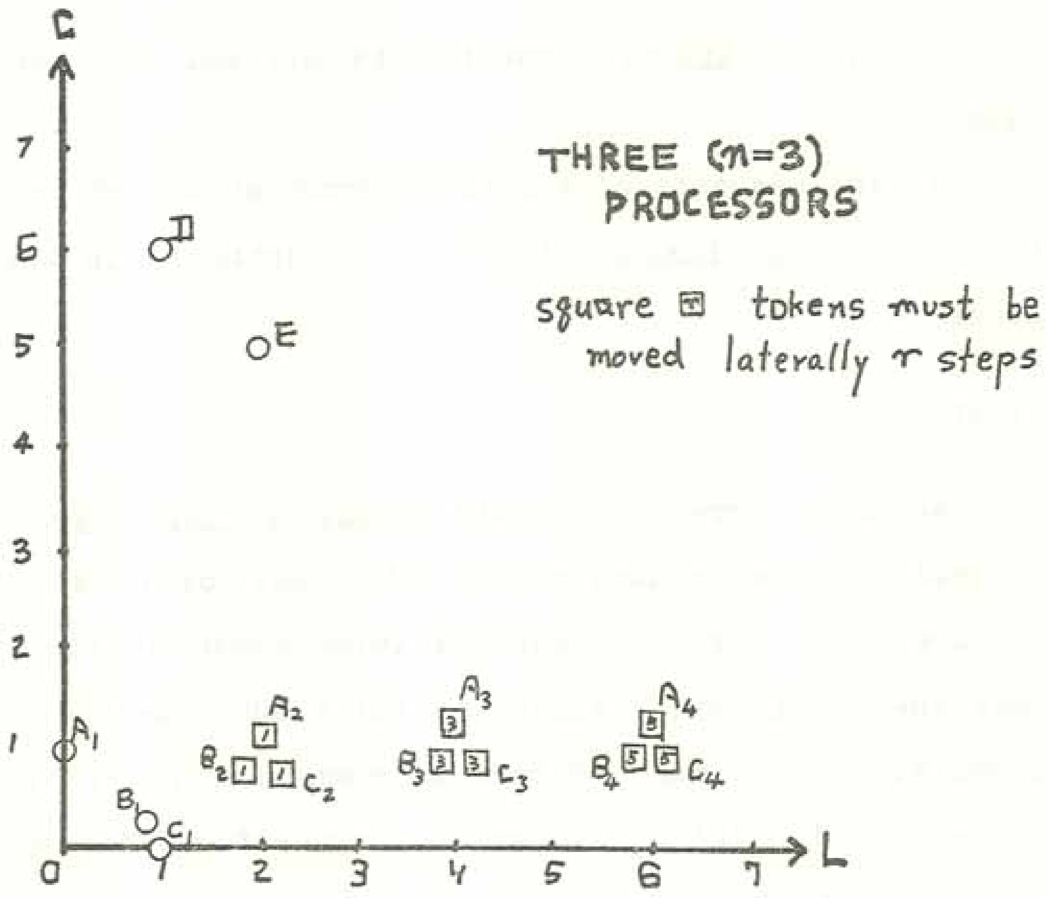
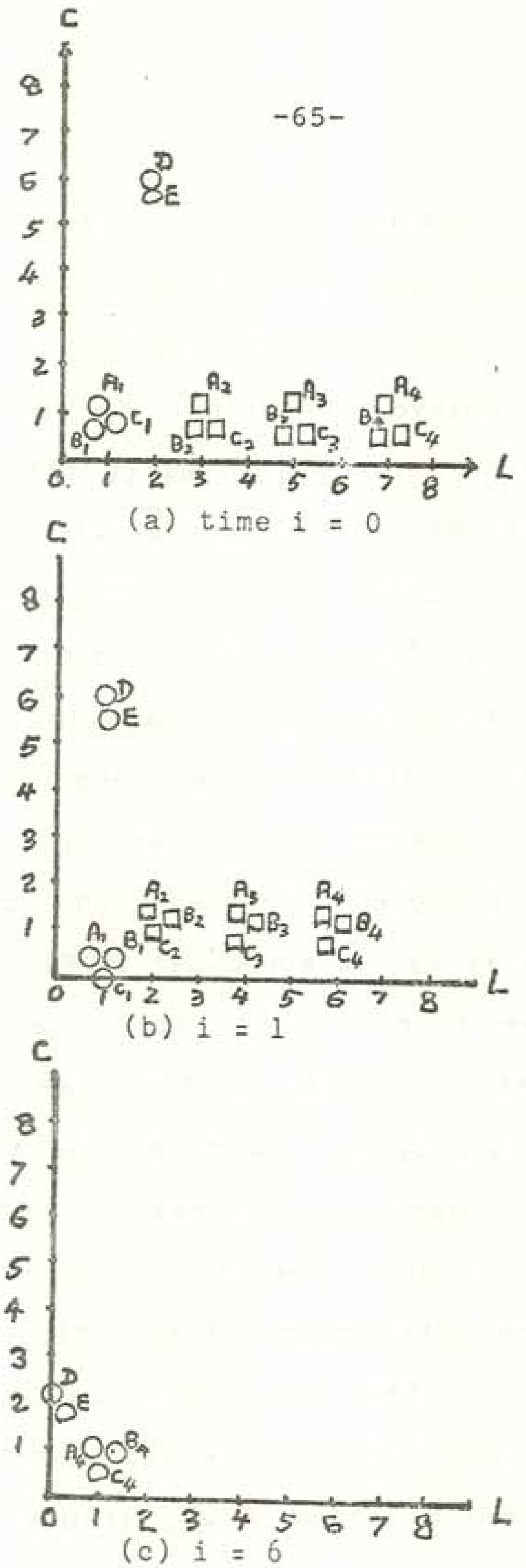


FIG. 5.1 (a)

A FEASIBLE SCHEDULE



(a) time i = 0

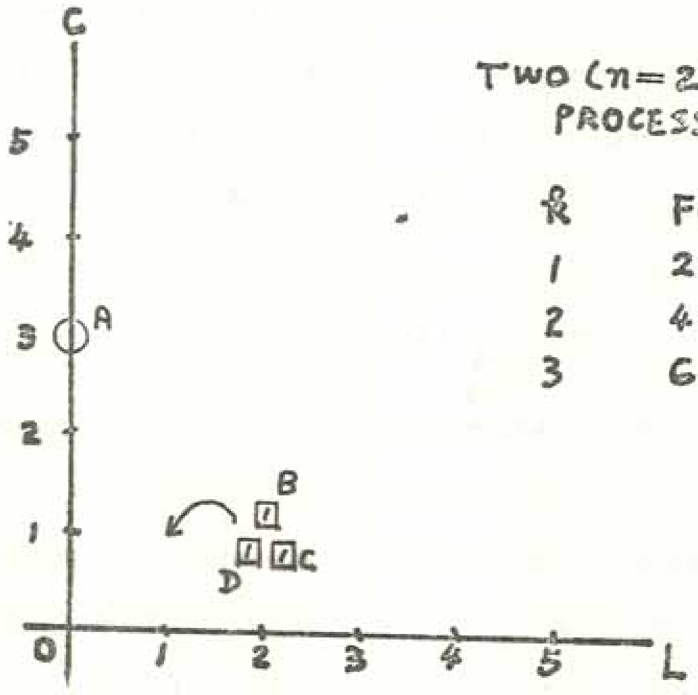
(b) i = 1

(c) i = 6

FIG. 5.1 (b)

COUNTER-EXAMPLE FOR THE <u>LEAST</u> <u>LAXITY</u> ALGORITHM

FIG. 5.1(b) shows the sequence of steps that leads to failure for the <u>Least Laxity</u> algorithm (this counter example is credited to my colleaque Knut Nordbye). FIG. 5.1(a) shows one way to achieve scheduling. On comparison between the two, one sees that the <u>Least Laxity</u> algorithm executes the short tasks A1, B1 and C1 too soon with the result that a processor is left idle at the second step. Since the tasks need a total of 24 units of processor time in 8 units of time (the longest deadline is at time i=8) and there are only three processors available, we cannot afford to lose even one unit of processor time. One may suspect that if the pitfall can be detected beforehand, then one may be able to schedule around it as the scheduler in FIG. 5.1(a) does. Unfortunately, the function F(k) no longer holds enough information to let us decide whether scheduling can be achieved if there are square tokens on the game board. If some of the square tokens are constrainted to move laterally until they have crossed the C (vertical) axis, then obviously a failure will occur. Presumably such occasions will not occur since they correspond to tasks which are impossible to complete before their deadlines. However, we still run into trouble even if we exclude such "unfair" tasks.



TWO ($n=2$) PROCESSORS

k	$F(k)$
1	$2-1=1$
2	$4-2=2$
3	$6-6=0$

FIG. 5.2

THE INSUFFICIENCY OF THE CONDITION $F(k) \geq 0$

In FIG. 5.2, the square tokens B, C, D are confined to move horizontally for one step. Hence at time i=0, the scheduler is forced to idle one of the two available processors. Since the total processor time required by the four tasks is 6 units of processor time and that they have to be finished by time i=3, a failure is bound to occur if a processor is allowed to idle. However, the values of F(k) tabulated in FIG. 5.2 are all non-negative, indicating that scheduling could be achieved. On examination of

$$F(k) = k*n - \sum_{D_j \leq k} C_j - \sum_{\substack{D_j > k \\ L_j \leq k}} (k-L_j)$$

one can see that F(k) assumes an available processor power of k*n units whereas the "effective" processor power is less because of processors being forced to idle. In other words,

<u>The condition F(k) ≥ 0 for all k>0 is not sufficient for achieving scheduling when the requests are distributed in time.</u>

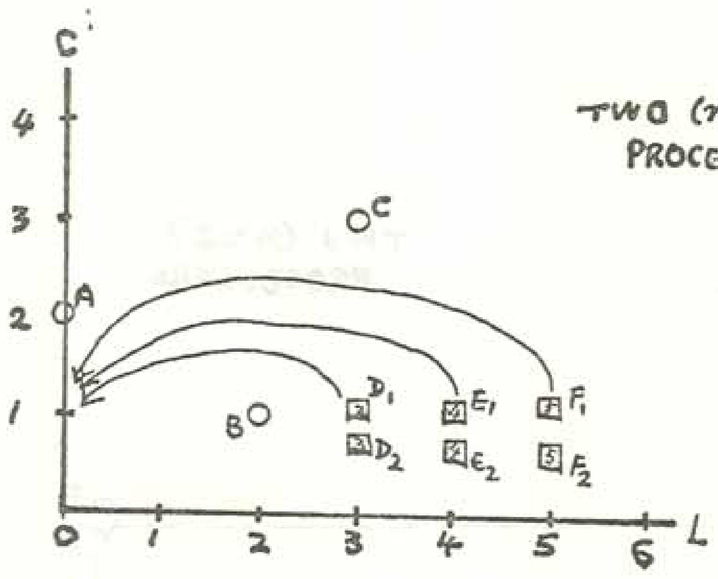
5.2 Difficulty of the General Case

The root of the problem here can be traced back to the proof of Lemma 3.3. There are situations where the scheduler has to decide whether to minimize the overall completion time of the tasks which have been requested (by executing as many of the longer tasks as possible) or to minimize the number of urgent tasks (by executing the tasks with the least laxity). The first decision is better made if the processors are expected to be heavily utilized before the longer deadlines of the present requests run out whereas the second decision is preferable if there are urgent requests in the near future. This suggests two possible approaches :

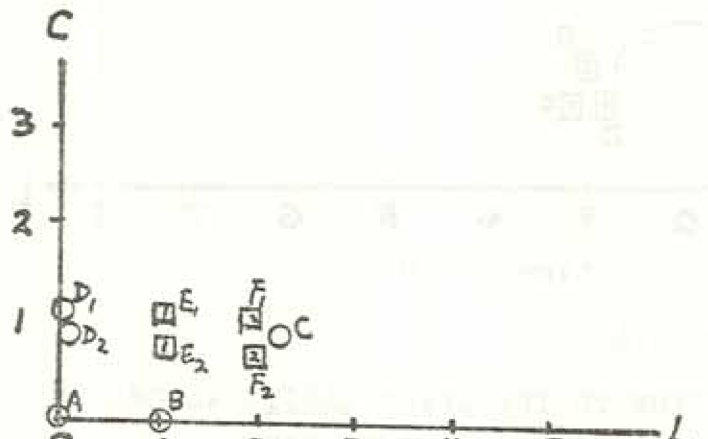
(a) Execute as many as possible tasks with small laxities while making sure that no processor will be idle nunecessarily in the next step.

(b) Execute as many as possible tasks with bigger computation time while keeping the necessary condition $F(k) \geq 0$, for $k>0$.

The first approach prevents unnecessary waste of processor time in the next step. However, it does not guarantee that no processor will be idled unnecessarily in all future steps. A counter example is shown in FIG. 5.3(a). If tokens A and B are moved down in the first step, the scheduling situation shown in FIG. 5.3(b) will result at time i=3 and a failure will occur. However, if tokens A and C are moved down in the first two steps, all three tokens A, B and C will have reached the horizontal axis at time i=3 and this shows that scheduling can indeed be achieved. The fault of the second approach is more subtle as is illustrated in FIG. 5.4. If tokens E and F instead of token A are moved downwards in the first step, then we shall have a situation almost the same as FIG. 5.2 (FIG. 5.4 has the additional tokens E and F) for which scheduling cannot be achieved. This is again due to the fact that $F(k) \geq 0$ no longer guarantees success.



(a) time i = 0



(b) i = 3

FIG. 5.3

MODIFICATION TO THE LEAST LAXITY ALGORITHM TO AVOID IDLE PROCESSORS

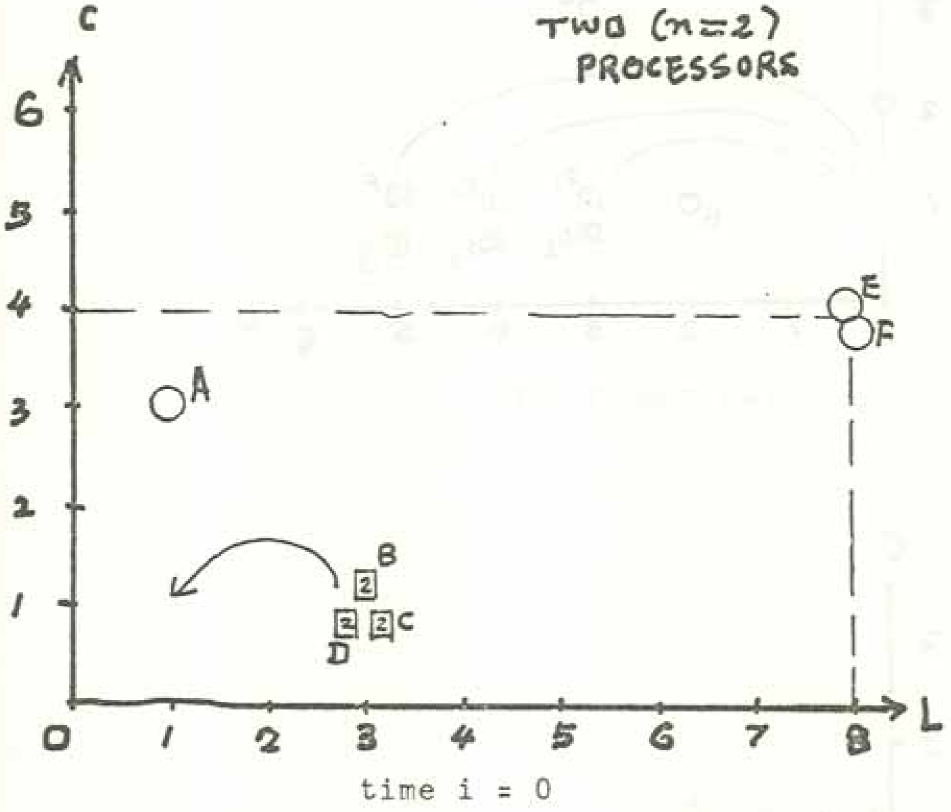


FIG. 5.4

MODIFICATION TO THE LEAST LAXITY ALGORITHM
TO MINIMIZE COMPLETION TIME

The optimal algorithm must then lie between these two extreme approaches. A heuristic which has been suggested is to compute the ratio C_j/L_j for each task and execute those with the largest C_j/L_j. Notice that when $L_j=0$, the above ratio is infinitely big and the corresponding task will have first priority while a bigger C_j also increases its chance of being executed, thus helping to prevent unnecessary loss of processor time. Unfortunately, this algorithm is not optimal as the counter example of FIG. 5.5 shows.

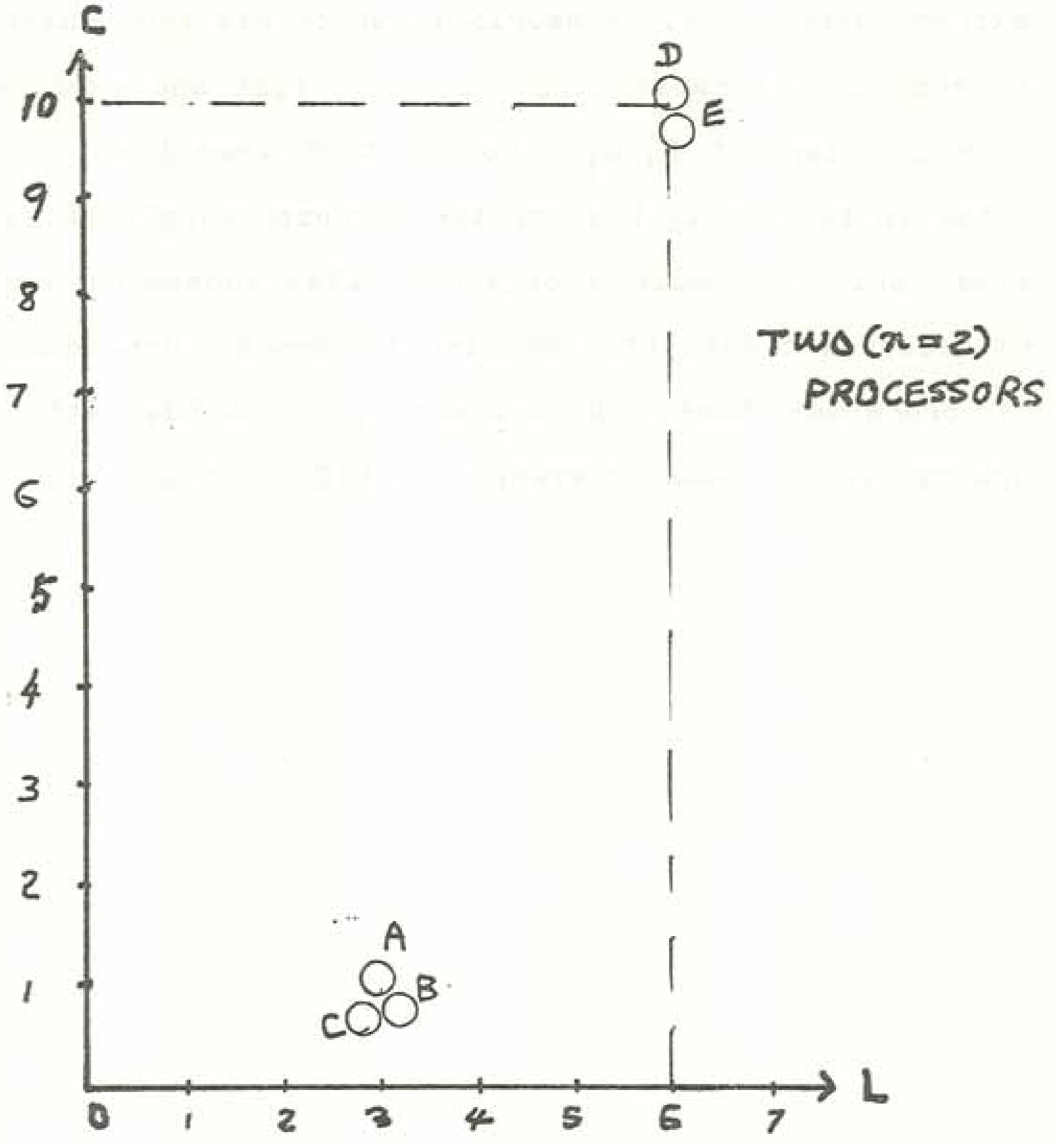


FIG. 5.5

COUNTER-EXAMPLE TO THE MAX { C/L} ALGORITHM

Our problem would be solved if we could find a (in A.I. terms) static evaluator such as F(k) for the general case. On the other hand, a schedule which is good for the interval time=0 to, say time=k may have to be modified to accomodate requests which occur after time=k. Thus we cannot solve our problem one piece at a time as we are able to do in the case where all requests occur at time=0 (the _Least Laxity_ algorithm is an example) or in the case where the deadlines are coincident (and the _Least Lazity_ algorithm is the same as one which executes tasks with the longest unfinished computation so that the conflicting objectives mentioned above are resolved). The fact that we cannoot readily apply divide-and-conquer strategy (or dynamic programming for that matter) is an indication of how difficult the problem seems to be.

5.3 _A Sufficient Condition for Scheduling_

To conclude this chapter, we shall prove a sufficient (but not necessary) condition for achieving scheduling when the requests for the tasks are distributed in time. Specifically, we prove the following

Theorem 5.2

If scheduling can be achieved for a set of tasks when all of their requests occur at time 0, then scheduling can be achieved for these tasks no matter how their requests are distributed in time.

Proof

For ease of proof, we shall hold all tokens with non-zero counts i.e. those tasks whose requests occur after time=0 at the positions where they are first free to move downwards (the j th token is placed at $L=L_j$, $C=C_j$) until the time their requests actually occur i.e. when their counts turn zero and from then on they are free to move. The scheduling algorithm to be used in the proof is the Least Laxity algorihtm and the proof is very similar to that of Lemma 4.3. Specifically, we shall show that $F(k,0)\geq 0$ for all k implies that for all $r>0$, $F(r,1)\geq 0$.

First we observe that all tokens on the C axis must have zero count for otherwise their computation times must be greater than their deadlines and that is not allowed. Hence $F(1,0)\geq 0$ implies that all tokens on the C axis must be moved downwards and so there is no immediate failure.

To prove that $F(r,1)\geq 0$ for all r, there are two cases to consider: (since these two cases correspond closely to the first two cases of Lemma 4.3, we shall discuss only the difference owing to the presence of stationary tokens with non-zero count.)

Case 1 (FIG. 4.3 (a))

In this case, all the tokens in R_1 and R_2 are moved downwards at the first step. The tokens with non-zero counts do not contribute to the difference $F(k,1)-F(k,0)$ since for these tokens C_j and $k-L_j$ remain unchanged.

Case 2 (FIG. 4.3(b))

In this case, some of the tokens in R_1 or R_2 are moved laterally towards the C axis at the first step. Observe that by the Least Laxity algorithm, n tokens in R_1 or R_2 must have been moved downwards at the first step. If a token with non-zero count is in R_1, then it contributes equally to $F(k,1)$ and $F(k+1,0)$. If it is in R_2, then it contributes $-(k-L_j)$ to $F(k,1)$ and $-(k+1-L_j)$ to $F(k+1,0)$. Hence it contributes positively (by one) to the difference $F(k,1)-F(k+1,0)$. For Case 2, $F(k,1)-F(k+1,0) \geq$ number of tokens with non-zero counts in R_2.

The third case in Lemma 4.3 does not exist if the Least Laxity algorithm is used since some tokens with zero count must be moved downwards in R_1 or R_2 for all k. If all the tokens in R_1 and R_2 have non-zero count or if there is no token in R_1 and R_2, then $F(k,1) = F(k,0) \geq 0$.

This exhausts all possible cases and the theorem is proved.

It is easy to see that any algorithm which moves at least (n - F(k) - number of stationary tokens in R_2) left of the line L=k for all k will also achieve scheduling. Theorem 5.2 is a confirmation of the intuition that for a given set of tasks, the "most restrictive" case is when all of their requests are synchronized.

CHAPTER 6
PERIODIC REQUESTS

6.1 Extension of Single Processor Scheduling

Having seen the difficulty in dealing with the general case, we shall retreat a step by putting restrictions on the way requests are distributed. Specifically, we shall focus on periodic tasks i.e. requests for the same task occur as soon as current deadlines are due. A necessary restriction for achieving scheduling is an upper bound on the utilization factor U :

$$U = \sum_{j} (C_j / D_j) \leq n \qquad (6.1)$$

Liu and Layland [3] have proved that in the case of a single processor $U \leq 1$ is a necessary and sufficient condition for scheduling periodic tasks and the algorithm used is not surprisingly the Earliest Deadline algorithm. This result immediately provides us with a lower bound on the utilization factor such that if U is smaller than the claimed lower bound, scheduling can always be achieved even

---

see the footnote for equation (3.1)

on a multiprocessor system. Specifically,

<u>A sufficient condition for scheduling m periodic tasks on n processors is U $\leq$ n/2</u>.

The proof to this claim follows easily if we make an analogy of this scheduling problem with the <u>bin packing</u> problem. The key observation is that if we can partition the m tasks into n sets so that for each set of tasks $\Sigma\ C_j/D_j \leq 1$, then we can assign each set of tasks to a specific processor and reduce the multiprocessor scheduling problem to the single processor problem. Now suppose that the proposed partition is impossible. Then increase the number of partitions to say n+r until the restriction $\Sigma\ C_j/D_j \leq 1$ holds for each set. For each of these n+r partitions, maybe except for one, the inequality $\Sigma\ C_j/D_j > 1/2$ must hold for otherwise we can merge two of the offending sets into one and still obey the restriction $\Sigma\ C_j/D_j \leq 1$. Thus there are at least n sets (for the case r=0) with $\Sigma\ C_j/D_j > 1/2$, implying that $\Sigma\ C_j/D_j > 1/2 * n$, a contradiction.

This <u>bin packing</u> approach is especially unsatisfactory when each periodic task has a C_j/D_j factor of slightly above 1/2. However, this lower bound of 1/2 serves as a base line of the efficiency of any suboptimal algorithm.

Naturally, the next question is whether there are any algorithms for which the necessary condition (6.1) is also a sufficient one. While there is no evidence to the contrary, no such optimal algorithm exists to the knowledge of this author. Even with the additional condition (6.1), both the Earliest Deadline and the Least Laxity algorithms fail unnecessarily. The situation shown in FIG. 5.1 is an example of such failures if we consider tokens A1, A2 etc. as representing requests (spaced two time units apart) from the same periodic task. Notice that both the Least Laxity and Earliest Deadline algorithms select tasks A1, B1, C1 for execution instead of D and E, and a processor is allowed to idle.

6.2 Scheduling by "Time Slicing"

While it is not clear whether the condition $U \leq n$ is sufficient for scheduling periodic tasks on n processors (for example, modifications to the Least Laxity algorithm described in section 5.2 have not been found to fail when $U \leq n$), it is obvious that if we can execute task j a fraction C_j/D_j of a time unit in each time unit, then (6.1) does guarantee that scheduling can be achieved, e.g. Coffman et al. [6]. This is possible if the basic time unit is small enough. Exactly how fine the resolution must be is given by theorem 6.1.

<u>Theorem 6.1</u>

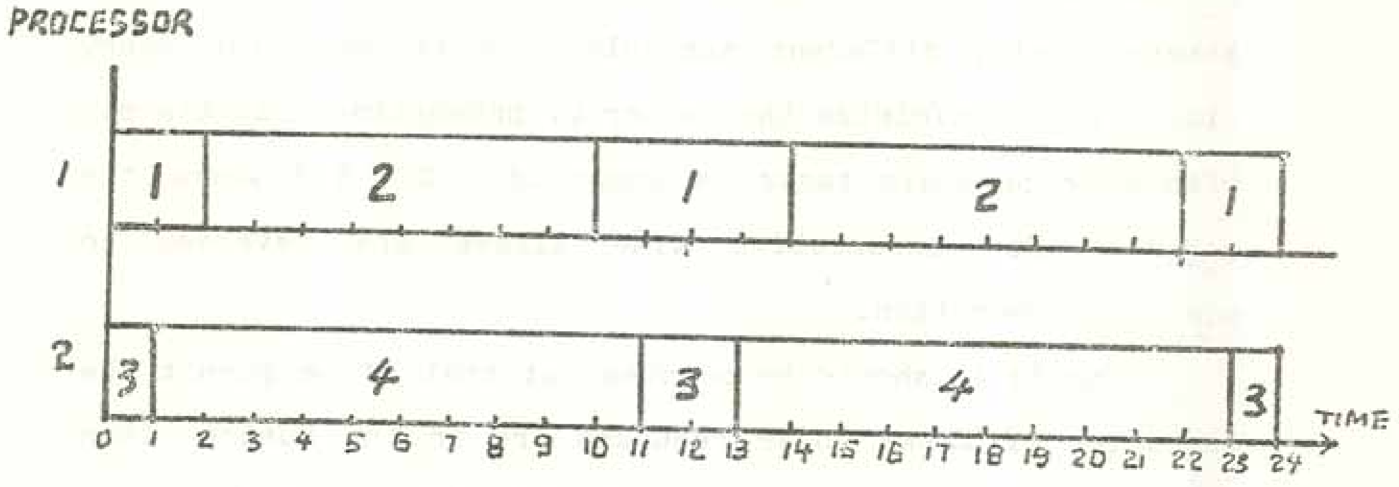
A sufficient condition for scheduling m periodic tasks with a utilization factor $U \leq n$ on n processors is that

$t = GCD(T, T*C_1/D_1, T*C_2/D_2, \ldots, T*C_m/D_m)$

be an integer where $T = GCD(D_1, D_2, \ldots D_m)$.

<u>Proof</u>

If t is integral, then all of T, $T*C_1/D_1$, $T*C_2/D_2$, and $T*C_m/D_m$ must be integers. Hence we can execute task 1 for $T*C_1/D_1$ time units, task 2 for $T*C_2/D_2$ time units,..., task m for $T*C_m/D_m$ time units for <u>every T time units</u>. For task j, every request requires C_j units of computation time and this has to be done in D_j time units. However, in every D_j time units, there are D_j/T (guaranteed to be an integer by the definition of T) "time slices" in each of which $T*C_j/D_j$ units of execution time is alloted to task j. Hence task j gets $D_j/T * T*C_j/D_j = C_j$ units of computation time every period. Within every "time slice", task j requires $T * C_j/D_j$ units of computation time. It is obvious that the computation required every "time slice" can be completed in $(\Sigma T*C_j/D_j) / n$ or $\max\{T*C_j/D_j\}$, whichever is greater. Hence the theorem is proved.



TWO PROCESSORS

FOUR	TASKS	C_j	D_j
	J_1	2	6
	J_2	4	6
	J_3	2	12
	J_4	20	24

$$T = GCD\,(6, 6, 12, 24) = 6$$
$$t = GCD\,(6\times\tfrac{2}{6}, 6\times\tfrac{4}{6}, 6\times\tfrac{2}{12}, 6\times\tfrac{20}{24}, 6)$$
$$= GCD\,(2, 4, 1, 5, 6)$$
$$= 1$$

FIG. 6.1

EXAMPLE OF "TIME SLICING"

Note that the same schedule may be used for all time slices and so it needs to be computed only once. Alternatively, different schedules may be used for every time slice to minimize the number of preemption. An example with four periodic tasks is shown in FIG. 6.1 where the schedule for consecutive time slices are reversed to minimize preemption.

Finally it should be pointed out that if we permit the execution of tasks whose requests are in the future, then the condition $U \leq n$ is also a sufficient condition for achieving scheduling. Specifically, Knut Nordbye has proved that $U \leq n$ implies that for all $k>0$, $F(k) \geq 0$ (tokens representing future requests of task j are placed every D_j divisions apart). In this particular case, the <u>Least</u> <u>Laxity</u> algorithm will achieve scheduling provided that tokens representing future requests are allowed to move downwards.

CHAPTER 7

OBSERVATIONS AND FURTHER WORK

We set out to find optimal algorithms for scheduling real-time tasks with deadlines on a multiprocessor system. Unfortunately, the results of chapter three indicate that such algorithms do not exist without _a priori_ knowledge of the deadlines, computation times of the tasks or the distribution of their requests in time. Even if such _a priori_ knowledge is available, feasible schedules are still difficult to find as chapter five bears witness. It may well be the case that efficient algorithms are impossible to find, but our search must go on until it can be proved that the problem is NP-complete (which is tentamount to saying that the problem is mathematically intractible and that heuristic solutions are called for).

From the point of view of _Control Robotics_, algorithms which require _a priori_ knowledge of the parameters of the tasks are less than desirable. While it is true that the computation times of the tasks can be statistically determined, it is impractical to require _a priori_ knowledge of the distribution of requests in time. Furthermore, any practical algorithm must be computationally efficient and this excludes algorithms approaching exhaustive search (as may be the case if the problem is NP-complete). At the same time, it may be conducive to our goal to look for alternative solutions to our scheduling problem. Heuristics are satisfactory as far as they can provide bounds on the statistical rate of success, but such suboptimal algorithms are not very useful when the criterion of success is meeting deadlines. Another alternative is to place enough restrictions on the tasks so that optimal algorithms may once again exist.

A simple example in the use of heuristics is to use the Earliest Deadline algorithm while making sure that the utilization factor of each processor (defined as the sum of the fractions C_j/D_j of the tasks currently assigned to the processor) does not exceed unity. As long as it is possible to do so, every deadline will be met. This approach is in fact a reformulation of the scheduling problem into the bin-packing problem which has been extensively studied e.g. Johnson [7]. In our case, the assignment of tasks to processors can be done dynamically i.e. assignments are made as requests occur and it is not necessary to assign a task to the same processor all the time. The performance of various bin-packing algorithms is rather well known.

If restrictions are put on the set of tasks which can occur, then perhaps we can avoid the unpleasant situation as discussed in chapter three. From the engineering point of view, this may not be unreasonable since the tasks we may be asked to handle in real life may fall into certain patterns which the scheduler can make use of. Furthermore, we can always increase the number of processors in the system until some criteria for the existence of optimal algorithms are met. One such restriction was mentioned in chapter three, namely, that all tasks have unit computation time. Obviously, this restriction is far too severe. As the results in chapter six indicate, the <u>Earliest</u> <u>Deadline</u> algorithm (using bin-packing algorithms for processor assignment) is optimal if the utilization factor U is not greater than 0.5 * n for a n-processor system. It remains a speculation whether $U \leq n$ is sufficient to guarantee the existence of optimal algorithms.

In the beginning of this thesis, it was noted that an additional parameter P may be associated with a task if there is a minimum period between two successive requests of the same task. This is another kind of restriction that we may realistically put on a task. Theorem 5.2 guarantees that if a set of tasks can be scheduled when all of their requests occur simultaneously (and theorem 4.1 provides a efficient way for testing feasibility), then scheduling can be achieved no matter how the requests (one for each task) are synchronized with respect to one another. Now if we make each P_j to be the longest of the deadlines i.e.

$\max\{D_j\}$, then it should not be surprising that theorem 5.2 can be generalized to the case where requests from the same task must be spaced at least $\max\{D_j\}$ time units apart. The scheduling game formulation of the problem should be a valuable tool for investigating the kinds of restrictions which are necessary to make certain algorithms optimal, e.g. for a given configuration of tokens on the L-C plane, there are regions where the introduction of additional tokens will cause the scheduler to fail; the geometry of these regions as they evolve in time depends on the scheduling algorithm.

Finally it must be pointed out that nothing has been said about the scheduling overhead which must be incurred in any real life systems. "Context switching" costs may be substantial if the scheduling algorithm used requires many preemptions. In this respect the <u>Earliest Deadline</u> algorithm is seen to be very efficient if we make the following observation. Suppose we add to each C_j (the computation time of the j th task) the processor time to do one "context switching" i.e. the time needed to save the status of the task being preempted and to load the preempting task and also the time to reload a task after it has been preempted, then we can charge the cost of each preemption to the preempting task and the cost of reloading a task after it has been preempted to the task which has been completed just before it. Notice that this is possible only in the case of the Earliest Deadline algorithm since each task will preempt another task (because of a more imminent deadline) only once for each of its requests. Since a processor stops executing a task and executes another one only if the former is being preempted by the latter or if the former is being completed, all the overhead is accounted for in the computation time. This cost is close to the best we can achieve since each task must be loaded at least once. In view of the efficiency of the Earliest Deadline algorithm, it is especially worthwhile to look for restrictions on a set of tasks so that the Earliest Deadline algorithm is optimal.

In this thesis, more questions have been asked than there are answers. The author hopes that he has provided some insight into the complexity of the problem of deadline scheduling and some limitation of what can be done. Future efforts may well be directed towards finding efficient algorithms and the associated restrictions on the tasks for optimality. Presumably, more efficient algorithms (less system overhead) require more stringent restrictions on the set of tasks they can successfully schedule. It is a challenge to the theorists to find a range of algorithms for the system users to choose from so that scheduling can be achieved with the least overhead.

REFERENCES

[1] M. Dertouzos, "Control Robotics: the procedural control of physical processes", Proceedings of the IFIP Congress 1974, pp. 807-813.

[2] S. Geiger, A New Language Approach to Computerized Process Control, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1974, also MAC TR-36.

[3] C.L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment", JACM, vol. 20, Number 1, January 1973.

[4] M. Garey and D. Johnson, "Complexity results for multiprocessor scheduling under resource constraints", Proceedings of the Eighth Annual Princeton Conference on Information Sciences and Systems 1974. pp. 168-172.

[5] P. Brucker, J. Lenstra and A.H.G. Kan, "Complexity of machine scheduling problems", Report BW 43/75, Mathematisch Centrum, Amsterdam, 1975.

[6] E.G. Coffman Jr. et al., Computer and Job-Shop Scheduling Theory, Wiley, New York, 1976. 1976.

[7] D. Johnson, Near-Optimal Bin Packing Algorithms, Ph. D Thesis, Department of Mathematics, M.I.T., June 1973.