NASA TECHNICAL NOTE

NASA TN D-5786

C.1

NASA TN D-5786

TASK SCHEDULING FOR A REAL TIME MULTIPROCESSOR

by John W. Jordan

Electronics Research Center

Cambridge, Mass. 02139

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION • WASHINGTON, D. C. • MAY 1970

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TN D-5786		

4. Title and Subtitle
Task Scheduling for a Real Time Multiprocessor

5. Report Date
May 1970

6. Performing Organization Code

7. Author(s)
John W. Jordan

8. Performing Organization Report No.
C-111

9. Performing Organization Name and Address
Electronics Research Center
Cambridge, Mass.

10. Work Unit No.
125-23-07-06

11. Contract or Grant No.

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration

13. Type of Report and Period Covered
Technical Note

14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

This report describes an algorithm for scheduling real-time tasks in a multiprocessor system. The algorithm guarantees that the deadlines of all scheduled tasks will be met. If the number of active tasks exceeds the capability of the multiprocessor, then only the highest priority tasks will be scheduled. The algorithm is suboptimal in that it may fail to schedule one or more low-priority tasks which could be accommodated. A hardware implementation of the algorithm is discussed.

17. Key Words
- Real-Time Tasks Scheduling
- Multiprocessor System
- Algorithm
- Hardware Implementation

18. Distribution Statement
Unclassified — Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	24	$3.00

*For sale by the Clearinghouse for Federal Scientific and Technical Information
Springfield, Virginia 22151

TASK SCHEDULING FOR A REAL TIME MULTIPROCESSOR

By John W. Jordan
Electronics Research Center
Cambridge, Massachusetts

SUMMARY

This note presents an algorithm for scheduling real time tasks in a multiprocessor. The algorithm guarantees that the deadlines of all scheduled tasks will be met. If the number of active tasks exceeds the capability of the multiprocessor, then only the highest priority tasks will be scheduled.

First, a simple scheduling algorithm is developed which guarantees real time deadlines for periodic tasks. A second algorithm is then derived from the first which also guarantees task deadlines but requires fewer processor interruptions. A system load parameter is defined and used to divide the tasks into two categories -- high priority tasks which can be scheduled with guaranteed deadlines and low priority tasks for which current resources are insufficient to allow execution. At this point, the division of the total system load among the individual processors is considered. This is an optimization problem, but at the present time a sub-optimal solution seems sufficient. The allocation of the total system load among the individual processors is in terms of a "task load" parameter and does not necessitate a consideration of real time task deadlines, thus considerably simplifying the scheduling problem.

Non-period tasks, interrupts and a hardware implementation of the scheduling algorithm are discussed. An appendix considers memory access conflicts.

INTRODUCTION

A multi-processor is a computer system consisting of a number of processors which can communicate through a common memory bank. Since the organization of these units may influence the choice of scheduling algorithms the structure of Figure 1 will be assumed. The most important characteristic of this system is that it is "fully distributed" in that each processor can access any memory module.

The number of memory units and processors is not important, and the system may be modular with a variable number of units. The procedures (programs) which are executed by the multi-processor are divided into tasks. When a processor completes a task it

executes a special procedure called the executive which determines which task will be processed next. This note is concerned with the algorithm used by the executive to select the next task.

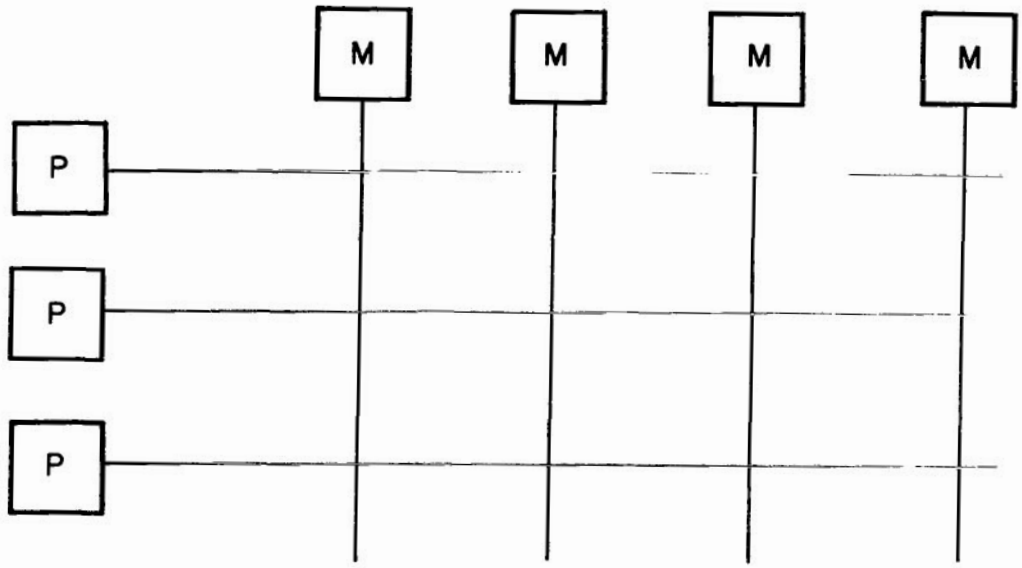

Figure 1.- A Multiprocessor

The most interesting aspect of the multiprocessor configuration is the possibility of "fail soft" operation -- that is, the failure of a memory module or processor will result in loss or degradation of some functions while still maintaining other, more important, functions. Achieving this "fail soft" operation is complicated by the possibility that the failure of a processor, for example, will not only reduce the computational resources of the system but may also impose an additional load on the system since special diagnostic routines must be executed to isolate the faulty unit and one or more tasks executed by the faulty processor may have to be "rolled back" or corrected for possible errors. For this reason it is desirable to design an algorithm which requires a minimum of re-configuration of the task structure in the event of a hardware failure. Ideally, the algorithm would insure the continued execution of the most important tasks and forego execution of less important tasks. It is then possible to introduce a degraded mode of operation with a modified task structure.

PROBLEM DEFINITION

Each real time task must be completed within a specified time frame as shown in Figure 2.

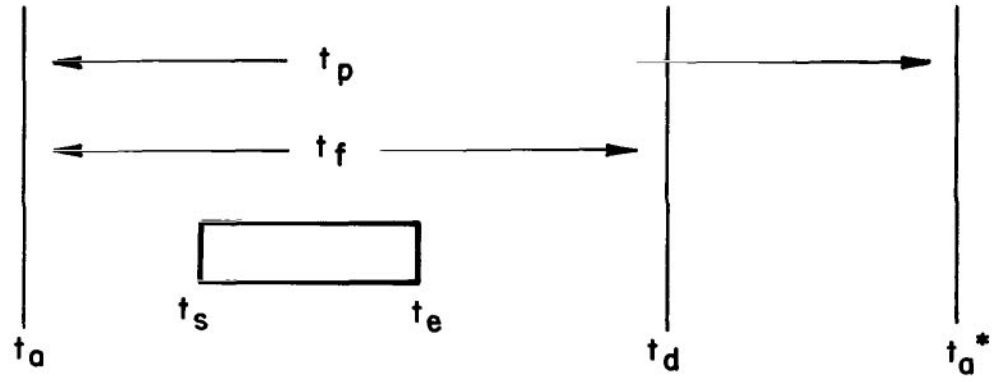

Figure 2.- Real Time Task

The times involved are – t_a the activation time after which the task can be executed. It is often dependent upon some system event such as an external interrupt, I/O completion or a processor timer. t_d is the task deadline. t_p is the task period if the task is periodic. t_s is the time when the task execution starts and t_e is the time when it ends. For the time being, only independent periodic tasks will be considered. These are characteristic of a real time sampled data system. For simplicity the time frame t_f during which the task must be processed will be taken as equal to the task period as shown in Figure 3. The activation time t_a is controlled by a processor timer and is precisely known.

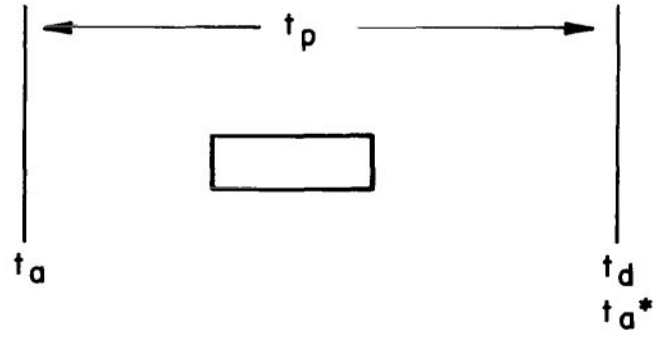

Figure 3.- Simplified Periodic Task

A solution of the scheduling problem requires

(1) the development of an algorithm such that those tasks chosen for execution are completed within the specified time frame and

(2) since the current number of operative processors may be incapable of processing all the active tasks, the algorithm must also select which of the active tasks are to be executed.

Since the "importance" of a task is only meaningful in the context of the task's function, it is up to the application programmer to assign to each task an integer variable called its priority index. Since mission conditions may change, the priority index may also change. Normally, however, it can be expected to be a relatively constant value.

It is important to note that this assignment of priority is not a determination of which task is to be executed next; it simply represents the relative value (at the present time) of eventually executing task A rather than task B provided that the multiprocessor is unable to execute both within their specified time frames.

It is easy to show an example (Appendix A) where executing tasks in order of their priority will result in missed deadlines when the multiprocessor is actually capable of executing all tasks satisfactorily. Conversely, an algorithm which considers only task deadlines will inevitably result in the execution of tasks of low priority while tasks of higher priority miss their deadline when the multiprocessor is operating under an overload condition. Actually, the constraints imposed by real time deadlines may require that tasks of low priority be scheduled before tasks of higher priority; but the algorithm must also consider priority so that if all the deadlines cannot be met the tasks of highest priority will be successful. The resolution of this potential conflict between task deadlines and task priorities is the fundamental design problem.

It should be noted that priority is considered to be a variable independent of other task parameters such as length, period, etc. Correlations between task parameters may permit the use of simpler scheduling algorithms.

IDEAL MULTIPROCESSOR

In order to simplify the discussion of scheduling algorithms, it is helpful to define an ideal (but unrealizable) multiprocessor. An ideal multiprocessor behaves like a single processor and memory unit. The addition of more processors is equivalent to increasing the speed of the ideal processor by a unit amount. The scheduling of an ideal multiprocessor is then similar to multiprogramming with a single processor. In a later section the restriction of an ideal multiprocessor will be removed.

EXECUTIVE IMPLEMENTATION

The executive has at least two lists -- a timer list and the Active Task List (ATL). The timer list is controlled by a real time clock in each processor as described in Ref. 1. When the processor clock reaches a specified time the processor moves any tasks on the timer list which are due to be activated at that time to the ATL. The processor clock is then reset. As the processors become idle, they take the topmost task from the ATL. The scheduling algorithm is responsible for ordering the ATL.

Not all the tasks in the system will have real time requirements. These "background" tasks may be kept in a separate list ordered by priority. They are executed after the real time tasks have been processed.

SCHEDULING ALGORITHM ONE

Associate with each task a number t_m which is the maximum time required to execute that task. Consider the following group of tasks:

TABLE I

Task	Period	t_m
A	T	T/3
B	2T	2T/3
C	3T	T

With one processor the following schedule will meet the periodic deadlines. The reader will note that it is not a unique solution.

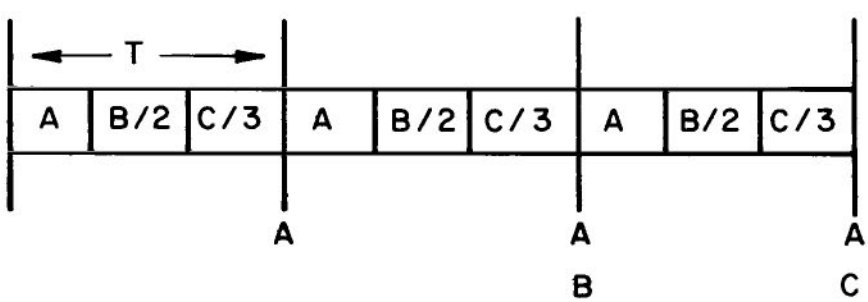

Figure 4.- Task Schedule

The letters below the lines indicate the task deadlines. The minimum task period T is a fundamental parameter and will be called a "cycle." The scheduling algorithm is

(1) Since A must be executed every cycle schedule A

(2) Since B must be completed in two cycles, do one-half of B each cycle

(3) Since C must be completed in three cycles, do one-third of C each cycle.

In general, if t_m is the maximum run time for task k and its period is t_p then

$$\alpha(k) = T t_m(k)/t_p(k) \tag{1}$$

is the amount of time which must be devoted to that task during eacy cycle in order to meet its deadline. For N_p processors and N_T tasks

$$T\ N_p \leq \sum_{k1}^{N_T} \alpha(k) \tag{2}$$

A multiprocessor loading parameter may be defined by

$$L = \frac{1}{T\ N_p} \sum_{k1}^{N_t} \alpha(k) \tag{3}$$

Computer loading is a basic parameter since it represents the fractional part of the computer's processing capability necessary to process the N_T tasks. A value of greater than one means that all the tasks cannot be processed. The value 100 x L is the percentage of the multiprocessor time required to process the tasks.

Each cycle $\alpha(k)$ microseconds are spent executing the kth task. Each cycle is exactly like all other cycles. This means that the processors must switch from task to task during a cycle without completing the tasks. For example, if there were 20 active tasks and two processors, each processor would make about nine task changes per cycle. Although $\alpha(k)$ would be different

for each task, if the basic cycle time T were 20 ms an average time of about 2 ms would be spent on each task per cycle. The task switching can be accomplished by a hardware interrupt or by subdividing the tasks into segments requiring α_k seconds or less to execute. A special switching instruction would bound each segment.

A suitable organization of the ATL is shown in Figure 5.

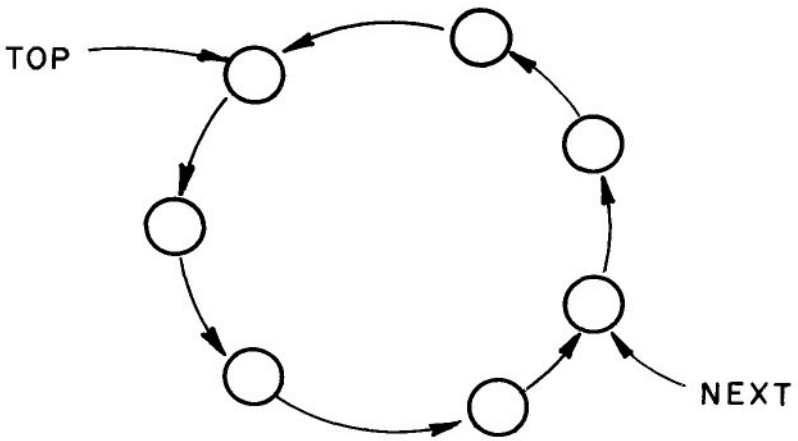

Figure 5.- Active Task List (ATL)

The tasks form a circular chain. There is a fixed pointer (TOP) which designates one task as the top of the chain. Another pointer (NEXT) points to the next task to be executed. An idle processor will pick up the NEXT task and advance the NEXT pointer. The processor timer will be set to cause an interrupt after α_k microseconds at which time the NEXT task will be executed. Every T microseconds the NEXT pointer will be reset to the TOP pointer and a new cycle initiated. Since each α_k is based upon maximum task run time and the actual run times will be somewhat less, the NEXT pointer will, in all likelihood complete more than one full circle per cycle. However, this additional time can also be used to process background (nonreal time) tasks.

Since the order in which the tasks are executed does not effect the deadline requirements they can be arranged in order of priority with the TOP pointer pointing to the task of highest priority. If one or more processors fail, the NEXT pointer will complete less than one full circle per cycle. However, those tasks with the greatest priority will be executed. It is interesting to note that this algorithm does not need any explicit information about the number of operating processors -- that is, Eq. (3) does not have to be evaluated -- since the algorithm automatically adjusts to the number of operating processors. On

the other hand by observing the position of the NEXT pointer at the cycle interrupt every T microsecond some information about the number of operating processor can be obtained.

A disadvantage of this algorithm is that if the number of active tasks is large there will be a corresponding increase in the number of interruptions per cycle. The overhead involved in changing the processor state may become excessive if there are a large number of active registers to be stored. In the next section, another scheduling algorithm which circumvents this difficulty will be presented.

SCHEDULING ALGORITHM TWO

For scheduling algorithm one a typical schedule might appear as in Figure 6. In this example L = 1 and task priority has been ignored.

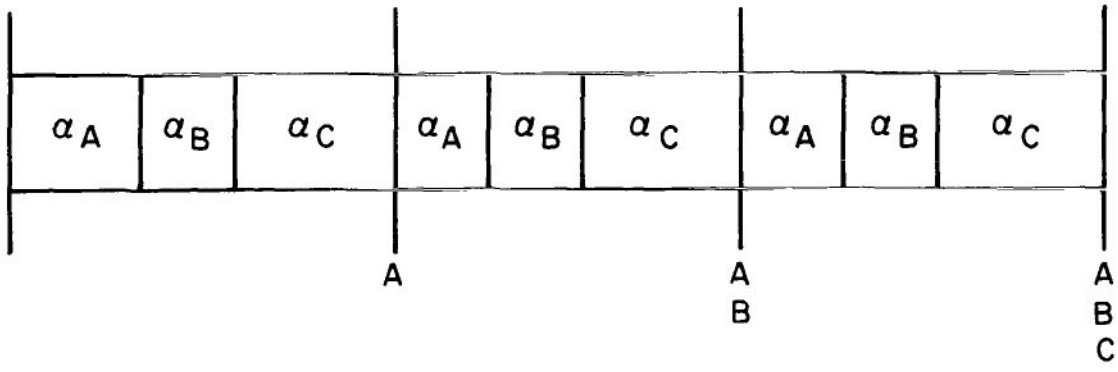

Figure 6.- Real Time Schedule

Once again, the letters under the graph indicate task deadlines. Since task A is due every cycle it must be executed each cycle. However, task B is due every two cycles and it makes little difference whether it is executed during cycle one or two. Since α_C is greater than α_B it is possible to move α_B from cycle two to cycle one and replace it with an equivalent amount of α_C from cycle one. Even if α_B were greater than α_C it would still be possible to move part of α_B into cycle one. Clearly, it is possible to rearrange the time of execution of a task as long as the execution is not delayed past the task deadline. One such rearrangement (not the only one) is to consolidate the tasks

such that those tasks with the earliest deadlines are executed first. This is scheduling algorithm two. Since insofar as deadlines are concerned, there is no essential difference between algorithm one and algorithm two it follows that algorithm two (earliest deadline) also guarantees that all the real time deadlines will be met. However, the number of task interruptions necessary will be reduced to one every T microseconds and only the cycle interrupt will be required. During the cycle the processors process the tasks to completion in order of their deadlines.

However, it is clear that as it presently stands algorithm two is sensitive to resource variations. For example, if a processor were to fail the tasks would continue to be executed according to their deadlines and without regard to their priority. To correct this condition, the implementation of Figure 7 can be used.

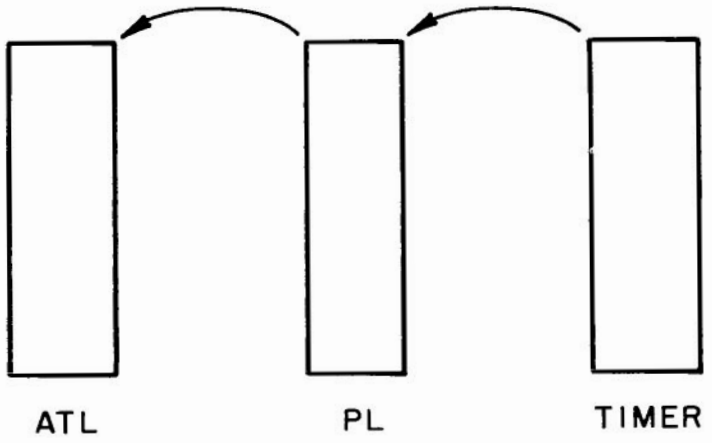

Figure 7.- Implementation of Algorithm Two

A third list called the priority list (PL) is added. All active tasks appear on the PL, ordered by priority. Only a certain number of the tasks on PL also appear on the ATL where they are ordered by deadline. In order to determine how many of the PL tasks may also appear on the ATL, the computer load parameter of Eq. (3) is calculated for the tasks on the PL starting with the highest priority task. When the load parameter equals or exceeds one, no further tasks may appear on the ATL. Thus modified, algorithm two guarantees that the real time deadlines of scheduled tasks will be met, and if the multiprocessor cannot do all tasks only those of the highest priority will be done. It has an advantage over algorithm one in that the processors must be interrupted only at the end of every cycle no matter how many tasks are active.

MULTIPROCESSOR ANOMALIES

The restriction of an "ideal" multiprocessor will now be removed. The occurance of so called "anomalies" in a realistic multiprocessor presents a serious difficulty in scheduling real time tasks. The literature (Ref. 2, 3, 4) contains numerous examples where shortening the execution time of one or more tasks results in an increase in the overall execution time of a string of tasks. This counterintuitive response can result when the shortened run time of a task alters the sequence in which subsequent tasks are executed, thus producing a complex re-ordering of the execution time history of the entire task string. Ref. 2 gives examples where decreasing the actual task run time, relaxing precedence relations between tasks and adding more processors can actually increase the overall time required to process a task string. In terms of the algorithms of this note the problem is illustrated by Figure 8.

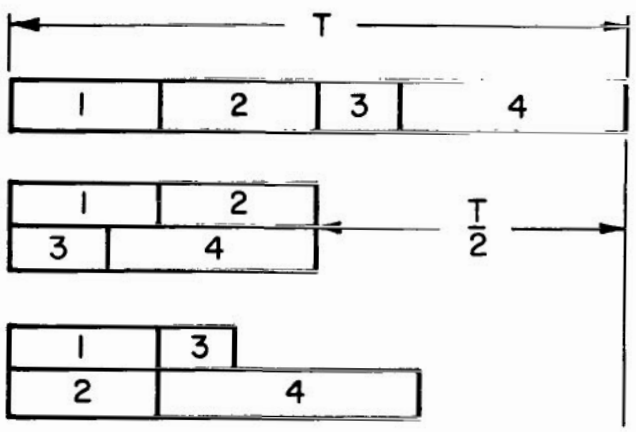

Figure 8.- Multiprocessor Anomality

Figure 8a represents a multiprocessor with one processing unit. If a second processor were added to an "ideal" multiprocessor, the task list would be processed in half the time. However, for an actual multiprocessor, Figures 8b and 8c clearly show that the time required is a function of the order in which the tasks are processed. What is required is to take the single task string of Figure 8a of length T and sub-divide it into N separate task strings of length T/N. In general, there is no procedure for doing this -- optimal or otherwise. For some task strings it may not be possible; for example, if Figure 8a consisted of one single task which could not be executed in parallel. However, Figure 8b and 8c represent two attempts at a subdivision

of Figure 8a -- one of which is successful; the other which is not. Quite possible there is an algorithm which is optimal in the sense that it subdivides a string of length T into N strings of length N/T in such a way that a minimum number of the original tasks must be discarded. Graham (Ref. 2) points out that the optimal solution (for an equivalent problem) can be found by trying all possible combinations but that this is practical for only a limited number of tasks. He suggests ordering the single task list by task length. This procedure cannot be applied to the algorithms of this note since algorithm one orders the tasks by priority and algorithm two by their deadline. Graham also provides a number of bounds for multiprocessor anomalities. For the case where T is the time required to process the single task list and T* is the time required to process the multiple lists

$$T^* = T$$

which simply says that the addition of more processors may provide no improvement -- which is indeed the case if the system contains one long task. Although this bound is of little help in this situation it indicates that processor anomalities must be considered in any multiprocessor scheduling philosophy.

The problem of converting an ideal multiprocessor schedule into an actual multiprocessor schedule can be considered from another viewpoint. Define a task load parameter by

$$\beta(k) = t_m/t_F \tag{4}$$

so that $\beta(k)$ is the fractional processor capacity required to process task k. From Eq. (3)

$$N_p \geq \sum_{1}^{N_T} \beta(k) \tag{5}$$

where N_p is the number of processors and N_T is the number of tasks which can be processed by an ideal multiprocessor. The scheduling problem is now transformed from the time domain to a "computer loading" domain. The ideal multiprocessor can accommodate a load of N_p but each real processor can accommodate a maximum load of only 1. Figure 9 illustrates this for $N_p = 2$.

The scheduling problem becomes one of allocating the total computational load of N_p into N_p separate lists, none of which can exceed unity. Of course, this is exactly the problem considered by Graham, only now in terms of computer load rather than task run times. Since the $\beta(k)$'s are discrete, it may not

be possible to decompose the list exactly and some $\beta(k)$'s may be left over. However, an additional degree of freedom has been introduced since the actual division into the separate lists can be done in any fashion including Graham's method of assigning the longest $\beta(k)$'s first. There is a constraint, however, in that if some $\beta(k)$'s (and their corresponding tasks) are discarded they should not have a priority higher than any task left. The "best" solution can be found by trying all possibilities although this soon becomes impractical if the number of active tasks is large. The simplest sub-optimal solution is to assign the $\beta(k)$'s to the processors in a round-robin fashion starting at the top of the priority list. Appendix B discusses another allocation strategy.

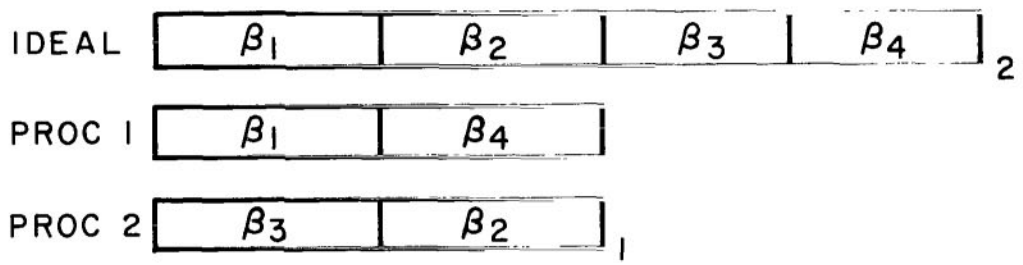

Figure 9.- Processor Loading

The assignment problem actually consists of decomposing the overall task list into separate lists for each processor. Eventually, of course, this assignment must be made. The inefficiency results from the discrete nature of the tasks and hence the $\beta(k)$'s. It is easy to find an upper bound to the inefficiency which results. From Eq. (3) the loading of the "ideal" multiprocessor is

$$L = \sum_{1}^{N_T} \beta(k)$$

For the individual processors the loading is

$$L_K = \sum_{1}^{R} \beta(k)$$

where R is the last task on the kth processor's list. The percent inefficiency can be defined as

$$7 = 100 \frac{L^*}{L} \qquad (6)$$

where

$$L^* = \sum_{K=1}^{N_P} L_K$$

It is reasonable to establish a limit on the ratio of a task's maximum run time to its execution time frame

$$\Delta \geq t_m/t_F$$

or

$$\Delta = \max \{\beta(k)\} \tag{7}$$

From Eq. (6) and Eq. (7) it follows that

$$\eta \leq 100\Delta \tag{8}$$

If Δ is .5, the maximum inefficiency is 50 percent; and if Δ is .2, the maximum inefficiency is 20 percent. This shows the value of dividing the total system load into many smaller tasks. However, the division criteria is not the length of the task but rather the ratio of the task length to the time frame in which the task must be executed.

In summary, scheduling a realistic multiprocessor requires the allocation of the total multiprocessor load among the individual processors. This allocation can be regarded as an optimization problem; however, a simple sub-optimal solution would seem to be adequate. When all the real time tasks can be scheduled, any solution can be regarded as "optimal." By allocating the task loading it is no longer necessary to consider the real time constraints on the individual tasks, thus considerably simplifying the scheduling problem.

Up to now, it has been assumed that all the tasks were periodic and independent. When the problem of scheduling these tasks is transformed from the time domain to the computer loading domain, it becomes a static problem of decomposing one long list into N_p shorter lists. However, the tasks are being periodically activated, run, and terminated so the computer loading is actually a dynamic rather than static parameter. But it is much simpler to consider it as a static quantity. One method of accomplishing this is as follows: when each periodic task is assigned to a processor it is tagged with that processors number. When the executive timer activates the task it puts it directly onto an Active Task List (ATL) for the proper

processor. The task load is always assigned to that processor, thus assuring the capacity to meet the task deadlines. If the number of available processors changes, the task lists must be rescheduled.

NONPERIODIC TASKS

As mentioned before, the total multiprocessor load is divided into real-time tasks which have associated deadlines and background tasks without deadlines. The real-time tasks are processed first. However, it is to be expected that in an actual system there would be additional real time tasks which do not meet the restrictions which have been assumed up to this point. Some examples are real-time tasks which are activated after the completion of other tasks and tasks controlled by external interrupts. If the tasks are activated after the prerequisite conditions have been met, then all active tasks can still be considered independent tasks.

If new, nonperiodic tasks are activated, then the computer loading and task scheduling must become dynamic since no prior allowance for the task's additional loading has been made. If the multiprocessor is processing a mixture of real-time and background tasks, this may be quite simple, since the multiprocessor has additional capacity beyond that necessary to handle the real time tasks. The new task can simply be added to any processor list which will accommodate it with less time remaining for background work. It is implicity assumed that the background tasks have lower priorities than the real-time tasks. However, it is also possible for the executive to maintain a balance between real-time and background tasks, based upon their priorities. A problem arises when a real-time task cannot fit on any processor list but its priority is greater than that of a currently scheduled task. In this case, the tasks must be rescheduled so that lower priority tasks are removed.

TASK ACTIVATION TIMES

The computer loading of any task, periodic or not, is given by

$$\beta(k) = t_m/(t_D - t)$$

where

t_m = Maximum task run time

t_D = Task deadline

t = Time the task is assigned to a processor

However, if a task can be assigned to a processor when it is activated, then

$$t_D = t + t_F$$

$$\beta(k) = t_m/t_F$$

and t_F can be considered to be the "response time" required from the task.

A HARDWARE EXECUTIVE

The task schedule must be recalculated when: (1) a real-time task is added but cannot be accommodated by any processor and tasks of lower priority are scheduled. A partial reconfiguration involving the tasks of lower priority is necessary, and (2) when a processor fails. Since it is desirable to accomplish the reconfiguration in a minimum amount of time, especially in the event of a processor failure, it is reasonable to seek a hardware solution even if this may present an additional reliability problem.

The hardware executive functions like an associative memory and a 16-bit adder, although it may be built using a small, conventional, high-speed memory. A typical memory word appears as in Figure 10.

Figure 10.- Task word

It would contain its own microprogram control (perhaps in the same memory). When a task is activated, its first Flag bit is set at 1. The hardware executive searches all active tasks which have the highest priority index. Assume that the microprogram control is such that the executive will divide the total system load among the processors by a simply round-robin sequence. In this case, all tasks having the highest priority index will be assigned to the processors in sequence and the load parameters for each processor will be incremented by the β field of the task word. The priority index is then incremented and the search repeated. The PRO field of the task word is set equal to the number of the processor to which the task is assigned. A second flag bit indicates that the task is assigned. When a processor requests a task, it is given the task with the nearest due time (TD).

INTERRUPTS

Interrupts may be handled by the hardware executive in a manner similar to that suggested in references 1 and 6. The executive can compare the next due time of the task currently being processed to the due time of the task at the head of the corresponding processors ATL and generate a processor interrupt, if necessary. Note that this involves a comparison between a processor and its ATL and, unlike reference 6, it is not necessary to make a comparison between processors, or to select a processor to be interrupted. In this way, it would not be necessary to interrupt each processor every T microsecond and the cycle interrupt is no longer required.

Reference 1 points out that external interrupts can be handled by simply activating their corresponding task. All that is necessary is to provide a mechanism in the hardware executive for setting the active bit of the correct task word. The operation of this interrupt system would be quite different from conventional systems, since an external interrupt which activates a task of the highest priority may not cause a processor interrupt even if all processors are working on tasks of lesser priority. Instead, the executive treats the interrupt task exactly like any other task. If it has the resources to guarantee the tasks execution with the required time frame, it may defer execution of the task. If its resources are insufficient, then the task is added to a system-wide queue with the winners selected by priority.

Another flag bit is used to indicate that a task is a member of the "timer" list. In this case, the TD field contains the time at which the task should be activated. The nearest activation time can be placed in a hardware register (labeled timer in Figure 11). When the nearest time arrives, the associative processor is interrupted and all tasks with their timer bits on and a TD field corresponding to the interrupt time are activated.

CONCLUSION

Algorithm two of this note, modified for a realistic multiprocessor is proposed as a reasonable solution to the real-time multiprocessor scheduling problem.

On a system basis, the algorithm does not insist that the active task of greatest priority or even the nearest deadline be processed first. It simply allocates the total task load among the individual processors in such a way that the deadlines of all allocated tasks are guaranteed. If the system resources are insufficient to allow all of the real time tasks to be executed, then those of highest priority will be scheduled.

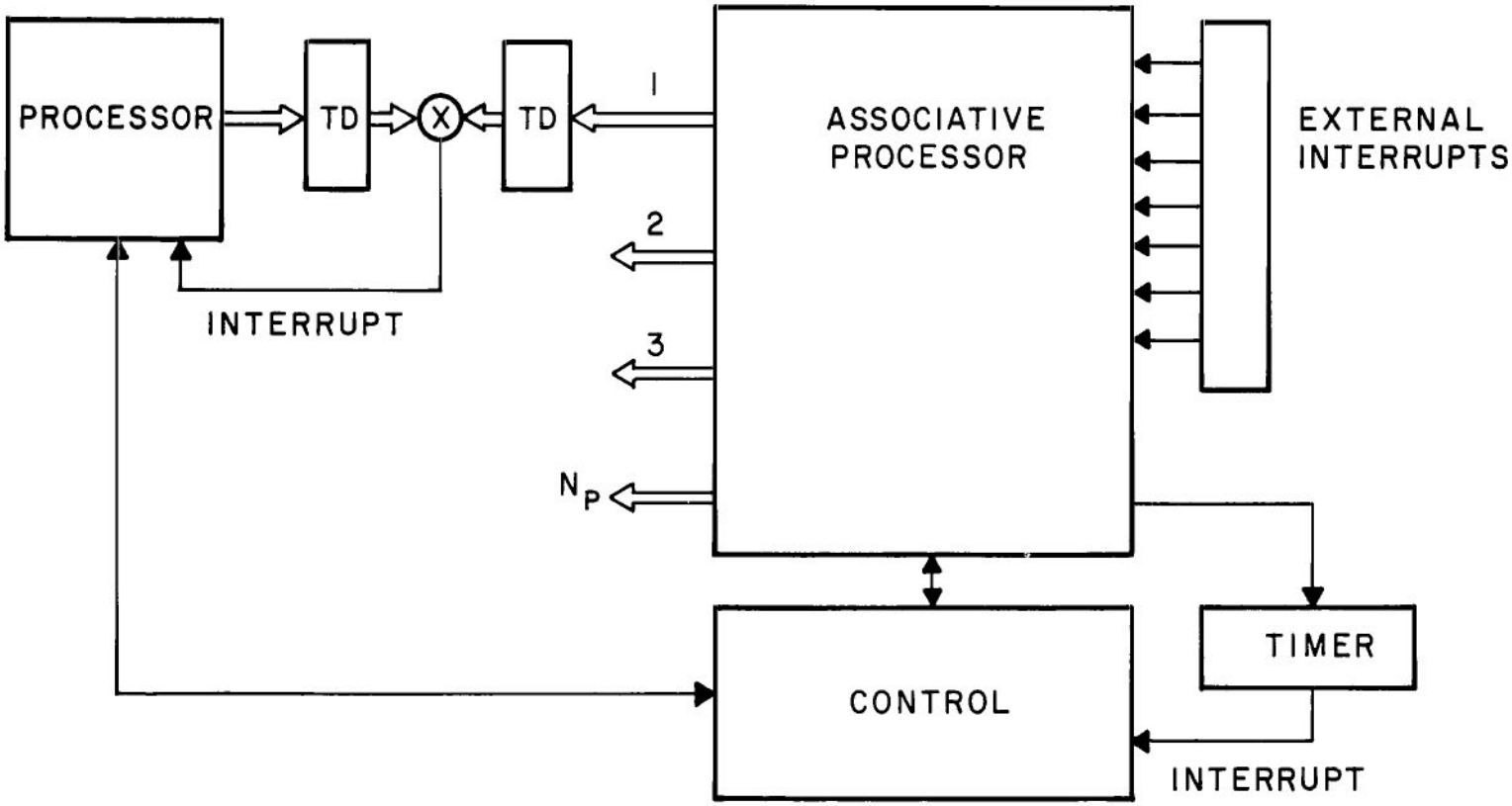

Figure 11.- Hardware executive

The algorithm is sub-optimal in that the allocation of the total workload to the individual processors may not be the "best" possible. However, the allocation strategy is independent of the overall scheduling algorithm and, therefore, subject to further improvement.

The algorithm is also sub-optimal in that it considers only the currently active tasks and does not attempt a global optimization using knowledge of future task loads. However, it is assumed that there is a class of tasks activated by interrupts for which the information required for the global optimization is incomplete and not available. If the executive does have information on the future activity of a task (for example, periodic tasks), it can use this to do static rather than dynamic scheduling, thus decreasing executive overhead and improving response time.

APPENDIX A

SCHEDULING BY PRIORITIES

Consider the following four periodic tasks:

TASK	t_p	t_m
A	T	T/4
B	2T	T/2
C	3T	3T/4
D	4T	T

Assume task priorities are assigned as follows:

$$P_A > P_B > P_C > P_D$$

A successful task schedule is shown in Figure A-1. The tasks are scheduled by deadlines. Note that this requires scheduling C before D during cycle 2 and D before C during cycle 4. Scheduling by priority would result in task D missing its deadline, although the multiprocessor is capable of executing all tasks successfully.

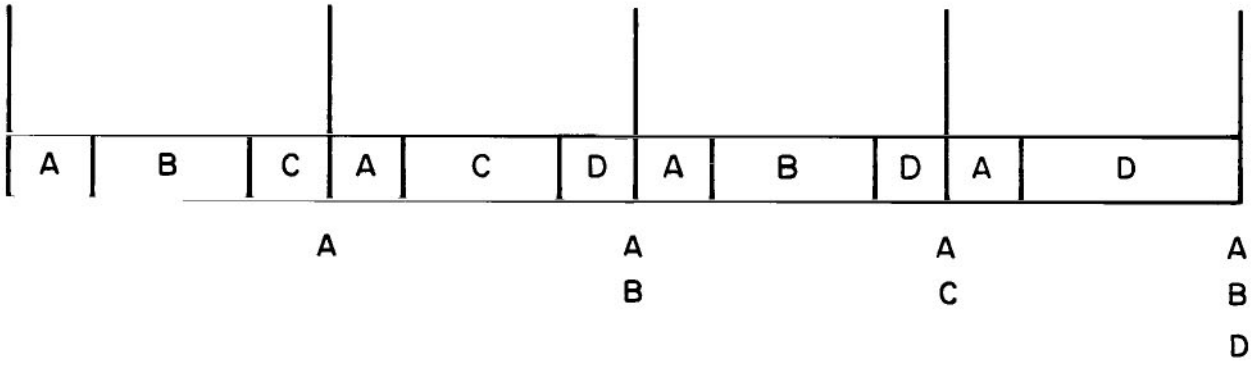

Figure A-1.- Task schedule

APPENDIX B

MEMORY CONFLICTS

Another reason for the departure of a realistic multiprocessor from an "ideal" multiprocessor is the presence of hardware and software conflicts. Software conflicts are due to exclusive data areas which can be modified by only one processor at a time. Hardware conflicts can result during I/O operations or, more probably, from multiple accesses of a single memory module. Conflicts complicate scheduling because they may extend the execution time of a task beyond its "maximum run time" parameter.

In order to estimate the effects of memory conflicts, a simulation was conducted using the multiprocessor configuration of Figure 1. The number of memory modules was assumed to be equal to the number of processors. The memory conflict depends on the cycle time of the memory and the processor operating speed which determines the rate of memory accesses. Processor timing was assumed to be as in Figure B-1.

t_{fI} time to fetch on instruction from the memory

t_D time to decode instruction address

t_{fO} time to fetch an operand

t_E time to execute an instruction

Figure B-1.- Processor cycle

The processors competed for each individual memory on a round-robin basis. A memory was attached to a particular processor for a memory cycle time (t_{mc}). If the required memory was unavailable, the processor cycle of Figure B-1 would be lengthened accordingly. Values used in the simulation were:

$t_{Of} = t_{fI} = t_{mc}/2$

t_D = 1 microsec (μs)

t_E = 1 μs (60%), 9 μs (40%)

Both the instructions and data were considered to be randomly distributed in pages among the memory modules. A total of 256 accesses was made from each page and then a new page was chosen at random. The resultant system through-put and percentage of a processor's time lost to a memory conflict appear in Figures B-2a and B-2b. For the processors assumed, the memory conflicts do not appear critical for 1 and 2 micro-second memories and the task maximum run-time parameter can be modified to take memory conflicts into account.

Since the scheduling algorithm of this note allows some measure of freedom in allocating the total system workload among the individual processors, one possible algorithm is to assign individual tasks to the processors so as to minimize potential memory conflicts. In this case, there would be a complex interaction between the task scheduling and memory allocation functions of the Executive.

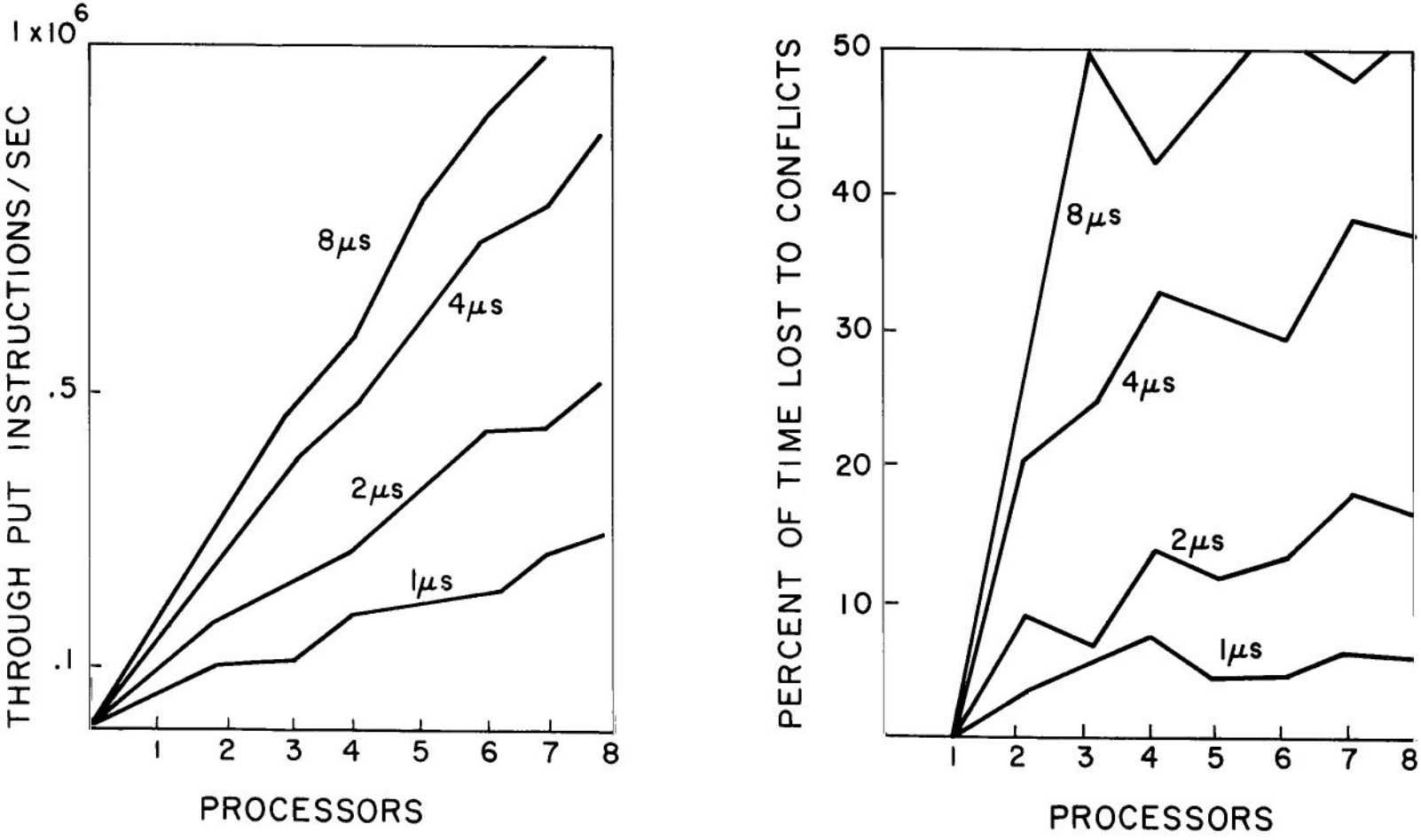

Figure B-2.- Memory conflicts

REFERENCES

1. Lampson, Butler W.: A scheduling Philosophy for Multiprocessing Systems. Communications of the ACM, vol. 11, no. 5, May 1968.

2. Graham, R. L.: Bounds on Multiprocessor Timing Anomalies. SIAM J. Appl. Math., vol. 17, no. 2, March 1969.

3. Manacher, G. K.: Production and Stabilization of Real-Time Task Schedules. Journal of the Association for Computing Machinery, vol. 14, no. 3, July 1967.

4. Ochsner, B. P.: Controlling a Multiprocessor System. Bell Laboratories Record, February 1966.

5. Richards, Paul I.: Parallel Programming. Report under contract No. AF33(600)-35190 at Tech. Operations, Inc., Burlington, Mass.

6. Gountanis, R. J., and Viss, N. L.: A Method of Processor Selection for Interrupt Handling in a Multiprocessor System. Proceedings of the IEEE, vol. 54, no. 12, December 1966.

"The aeronautical and space activities of the United States shall be conducted so as to contribute . . . to the expansion of human knowledge of phenomena in the atmosphere and space. The Administration shall provide for the widest practicable and appropriate dissemination of information concerning its activities and the results thereof."

— NATIONAL AERONAUTICS AND SPACE ACT OF 1958

NASA SCIENTIFIC AND TECHNICAL PUBLICATIONS

TECHNICAL REPORTS: Scientific and technical information considered important, complete, and a lasting contribution to existing knowledge.

TECHNICAL NOTES: Information less broad in scope but nevertheless of importance as a contribution to existing knowledge.

TECHNICAL MEMORANDUMS: Information receiving limited distribution because of preliminary data, security classification, or other reasons.

CONTRACTOR REPORTS: Scientific and technical information generated under a NASA contract or grant and considered an important contribution to existing knowledge.

TECHNICAL TRANSLATIONS: Information published in a foreign language considered to merit NASA distribution in English.

SPECIAL PUBLICATIONS: Information derived from or of value to NASA activities. Publications include conference proceedings, monographs, data compilations, handbooks, sourcebooks, and special bibliographies.

TECHNOLOGY UTILIZATION PUBLICATIONS: Information on technology used by NASA that may be of particular interest in commercial and other non-aerospace applications. Publications include Tech Briefs, Technology Utilization Reports and Notes, and Technology Surveys.

Details on the availability of these publications may be obtained from:

SCIENTIFIC AND TECHNICAL INFORMATION DIVISION

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

Washington, D.C. 20546