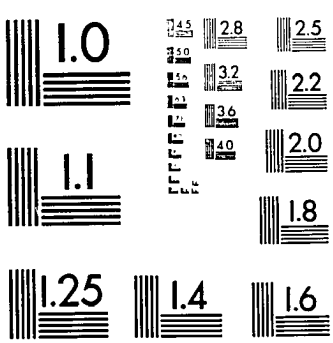
UMI University Microfilms International



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

University Microfilms Inc.
300 N. Zeeb Road, Ann Arbor, MI 48106

Downloaded from SHAREOK. Persistent Handle URL: https://hdl.handle.net/11244/5385

INFORMATION TO USERS

This reproduction was made from a copy of a manuscript sent to us for publication and microfilming. While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. Pages in any manuscript may have indistinct print. In all cases the best available copy has been filmed.

The following explanation of techniques is provided to help clarify notations which may appear on this reproduction.

1. Manuscripts may not always be complete. When it is not possible to obtain missing pages, a note appears to indicate this.

2. When copyrighted materials are removed from the manuscript, a note appears to indicate this.

3. Oversize materials (maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or in black and white paper format.*

4. Most photographs reproduce acceptably on positive microfilm or microfiche but lack clarity on xerographic copies made from the microfilm. For an additional charge, all photographs are available in black and white standard 35mm slide format.*

***For more information about black and white slides or enlarged paper reproductions, please contact the Dissertations Customer Services Department.**

UMI University Microfilms International

8603509

Davari, Sadegh

SCHEDULING PERIODIC-TIME-CRITICAL TASKS ON MULTIPROCESSOR COMPUTING SYSTEMS

The University of Oklahoma PH.D. 1985

University
Microfilms
International 300 N. Zeeb Road, Ann Arbor, MI 48106

THE UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

SCHEDULING PERIODIC-TIME-CRITICAL TASKS ON MULTIPROCESSOR COMPUTING SYSTEMS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

By

SADEGH DAVARI

Norman, Oklahoma

1985

SCHEDULING PERIODIC-TIME-CRITICAL TASKS

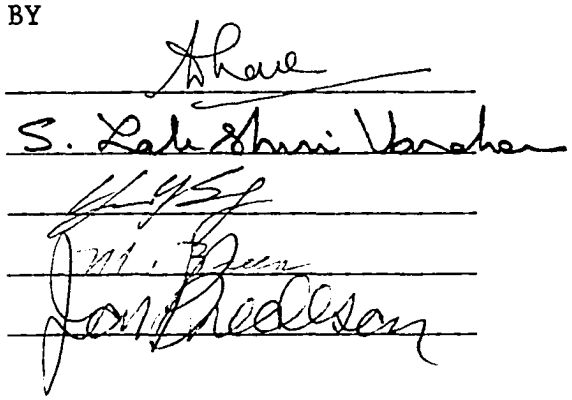
ON MULTIPROCESSOR COMPUTING SYSTEMS

A DISSERTATION

APPROVED FOR THE SCHOOL OF ELECTRICAL ENGINEERING

AND COMPUTER SCIENCE

BY



ACKNOWLEDGMENTS

I wish to express my sincere thanks and appreciation to my dissertation advisor, Dr. S. K. Dhall, who originally suggested that I pursue this problem, and provided intuitions that led to new results. His constant encouragement, receptive ear, and invaluable guidance made this work possible.

I would like to express my appreciation to Dr. S. K. Kahng, Director of the School of Electrical Engineering and Computer Science, for his support and thoughtful advice throughout my doctoral program.

I would also like to thank my dissertation committee members, Dr. J. Bredeson, Dr. M. Breen, Dr. S. Lakshmivarahan, and Dr. Y. Sung for their contribution and advise.

My special thanks go to my wife, Patricia, for her patience, understanding, and devoted love.

And finally, I would like to dedicate this work to my parents who brought me up and made me what I am with their endless love.

TABLE OF CONTENTS

	Page
LIST OF TABLES	vi
LIST OF FIGURES	vii
ABSTRACT	viii
CHAPTER	
I. INTRODUCTION	1
1.1 Motivation	1
1.2 Definition of the Problem	4
1.3 Terminology	8
1.4 Dissertation Contents	14
II. A SURVEY OF PREVIOUS WORK	16
2.1 Scheduling Tasks on a Single Processor System	16
2.2 Scheduling Tasks on multi-Processor Systems	26
2.2.1 Partitioning with respect to the Deadline Driven Scheduling Algorithm	32
2.2.2 Partitioning with respect to the Rate-Monotonic Scheduling Algorithm	34
III. PARTITIONING TASKS WITH RESPECT TO THE RATE-MONOTONIC SCHEDULING ALGORITHM	38
3.1 Introduction	38

3.2 Proof of NP-hardness..... 40
3.3 The First-Fit-Decreasing-Utilization-Factor Algorithm..... 44
3.3.1 Worst-Case Analysis of the FFDUF Algorithm..... 45
3.3.2 The Complexity of the FFDUF Algorithm..... 59
3.4 Summary..... 61
IV. ON-LINE ALGORITHMS..... 63
4.1 Introduction..... 63
4.2 The Algorithm NEXT-FIT-2..... 65
4.2.1 The Complexity of NEXT-FIT-2.. 67
4.2.2 Worst-Case Analysis of NEXT-FIT-2..... 68
4.3 The Algorithm NEXT-FIT-M..... 81
4.3.1 The Complexity of NEXT-FIT-M.. 84
4.3.2 Worst-Case Analysis of NEXT-FIT-M..... 85
4.4 A Special Case..... 96
4.5 Summary..... 105
V. CONCLUSION..... 106
5.1 Suggestions for Future Research..... 109

REFERENCES..... 110
APPENDIX A..... 112

LIST OF TABLES

Table		Page
2.1.	Results of Bin-Packing Algorithms.....	33
4.1.	The Upper Bounds for Worst-Case Performance Ratio of NEXT-FIT-2 for Different Values of x..	76
4.2.	Values of K_i in Worst-Case Analysis of NEXT-FIT-M.....	89
4.3.	Values of S_M in Worst-Case Analysis of NEXT-FIT-M.....	93
4.4.	Values of K_i in Worst-Case Analysis of NEXT-FIT-M, for the Special Case.....	97
4.5.	Values of S_M in Worst-Case Analysis of NEXT-FIT-M, for the Special Case.....	99
5.1.	A Summary of all the Known Results.....	108

LIST OF FIGURES

Figure		Page
1.1.	A timing diagram representing the schedule of a periodic-time-critical task on a single processor.....	6
1.2.	Schedules for two tasks on a single processor with different priority assignment.....	11
2.1.	A feasible schedule of three tasks, produced by the rate-monotonic scheduling algorithm.....	18
2.2.	An example of two tasks with total utilization factor of 1 which are not feasible under the rate-monotonic scheduling algorithm.....	19
2.3.	An example of a 3-task set, with utilization factor greater than $3(2^{1/3}-1)$, which is not feasible under the rate-monotonic algorithm....	21
2.4.	A feasible schedule of two tasks under the deadline driven algorithm.....	23
2.5.	An example of a set of 3 tasks which can not be feasibly scheduled on two processors under the rate-monotonic scheduling algorithm.....	28
2.6.	An example of a set of 3 tasks which can not be feasibly scheduled on two processors under the deadline driven scheduling algorithm.....	29
2.7.	A feasible schedule of a set of 3 tasks on two processors under a fixed priority algorithm....	31

ABSTRACT

The problem of allocating a set of periodic-time-critical tasks to processors in a multiprocessor system is considered. A periodic-time-critical task consists of a certain number of requests, arising periodically, each of which has a prescribed deadline. The allocation problem is to use a minimum number of processors subject to the condition that the tasks allocated to any processor must be feasibly schedulable according to some specified algorithm, i.e. the schedule provided by the algorithm must guarantee that the deadline of each request is honored. We first prove that this problem is NP-hard, and then present three heuristic algorithms and analyze their complexity and worst-case performance. One of the algorithms presented is an off-line algorithm and the other two are on-line. Two heuristic off-line algorithms for this problem are available in literature. The worst-case performance of our off-line algorithm is shown to be better than that of the two existing off-line algorithms. The on-line algorithms presented here are the only on-line algorithms presented for this problem to date. The time and space complexity of the presented on-line algorithms are shown to be better than those of the available off-line algorithms, and their

worst-case performance are shown to be comparable to that of the available off-line algorithms. Finally, it is shown that if the set of tasks to be scheduled does not contain any task with utilization factor in the range $(2^{1/2}-1, 1/2]$ then the worst-case performance of one of the on-line algorithms will improve considerably.

SCHEDULING PERIODIC-TIME-CRITICAL TASKS ON MULTIPROCESSOR SYSTEMS

CHAPTER I

INTRODUCTION

1.1 Motivation:

Computer systems are now being used to control and monitor a wide variety of real-time processes. In many of these real-time applications, the computer is required to execute a certain number of time-critical control and monitoring functions, in response to periodic external signals. It is essential that each such function is completely executed within a specified interval of time following the occurrence of the signal that causes the initiation of the function. Failure to do so may result in an irreparable loss. Some examples of such systems are[1,2]:

- a system to control the operation of a chemical plant.
- a system for monitoring a defense network.
- a system for tracking a missile and predicting its impact point.
- a system for monitoring space flights.

The computers dedicated for this purpose have been termed "process control computers". A process control computer performs a certain number of, periodically occurring, control and monitoring functions. We call each such function a periodic-time-critical task.

A periodic-time-critical task, in a real-time environment, consists of an infinite sequence of requests arising at fixed intervals of time. Associated with each request there is an initiation time, a computation time, and a deadline for the completion of its computation. The requests of a task are initiated periodically by means of some external signals. The computation for any request cannot start before the occurrence of the signal that initiates it. The deadline of each request of a task can be no later than the initiation of the next request of the same task. Failing to meet the deadline of a request causes irreparable damage, categorizing the environment as "hard-real-time"[3] in contrast to "soft-real-time" where a statistical distribution of response times is acceptable. The pointing of an antenna to track a spacecraft in its orbit is an example of such a task. In this example the goal is to have the antenna continuously pointing at the spacecraft, which is moving in its orbit at a known speed. In order to accomplish this, requests to adjust the antenna must be made periodically, one request every some fixed interval of time. Each such request adjusts the pointing

of the antenna with respect to the previous adjustment. Therefore, the computation of each request must finish before the next request arrives.

Recent progress in hardware technology and computer architecture has led to the design and construction of computer systems that contain a large number of processors. Because of their capability of executing several tasks simultaneously, it is both of practical and theoretical importance to investigate how to make best use of multi-processor computing systems for the type of tasks being considered.

Efficient utilization of computers in this type of environment can only be achieved by a careful scheduling of these periodic-time-critical tasks. This fact motivated our interest to work on this problem.

1.2 Definition of the Problem:

A periodic-time-critical task in general can be characterized by the quadruple (c, t, s, d), with $0 < c \leq d \leq t$. In this characterization, task T makes a request for c units of computation every t units of time. The first request of task T is made at time s, and thereafter at times $s + Kt$, ($K = 1, 2, \ldots$). The deadline for the Kth request of task T is $s + (K-1)t + d$.

A set of m periodic-time-critical tasks can be characterized by the quintuple $(\{T_i\}, \{c_i\}, \{t_i\}, \{s_i\} \{d_i\})$, where each T_i ($1 \leq i \leq m$) is a periodic-time-critical task.

The scheduling problem for a set of such tasks is to produce a schedule according to which all requests of all tasks in the set can be executed to meet their respective deadlines.

The results obtained, so far, for the general problem of scheduling periodic-time-critical tasks are not that significant[4-6]. But, an interesting variation of this problem is obtained by making the assumption that $d_i = t_i$ for all $1 \leq i \leq m$ [7-10].

We make the following assumptions about the type of tasks to be considered:

1. The requests of each task are periodic, with constant intervals between requests.

2. Deadlines consist of runability constraints only, i.e. each request must be completed before the next request of the same task.
3. The tasks are independent in that the requests of a task do not depend on the initiation or the completion of the requests of the other tasks.
4. Computation time for the requests of a task is constant for the task. Computation time here refers to the time a processor takes to execute the request without interruption.

These assumptions allow the complete characterization of a task by two numbers: its request period and its computation time. We will denote a task T by the ordered pair (c,t), where c is the computation time and t is the request period. The ratio 1/t is called the request rate, and the ratio c/t, denoted by u, is called the utilization factor of the task.

The utilization factor of a set of periodic-time-critical tasks is the sum of the utilization factor of all tasks in the set. Notice that the utilization factor of a task is the fraction of the processor time taken up by the task.

Figure 1.1 shows a timing diagram representing the schedule of the first 4 requests of a periodic-time-critical task. This task makes a request for 2 units of computation every 5 units of time. The deadline of each

T = (2, 5)

u = 2/5

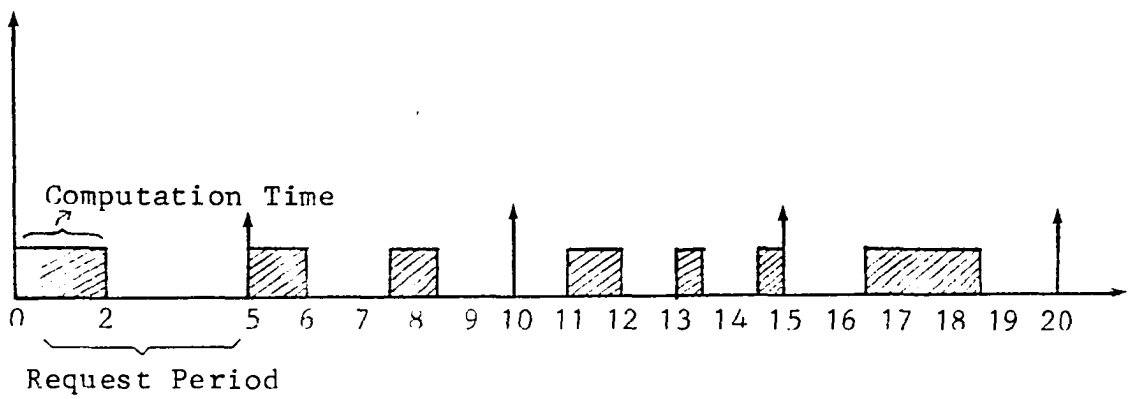


Figure 1.1: A timing diagram representing the schedule of a periodic-time-critical task on a single processor

request is the initiation of the next request. The service to each request can be given at any time and in any form within the span of the period, as shown in Figure 1.1. The request rate of this task is 1/5 and the utilization factor of this task is 2/5.

1.3 Terminology:

A schedule for a set of tasks specifies the time interval(s) during which the various requests of the tasks will be executed. A schedule which meets the deadlines of all the requests of all the tasks is called a feasible schedule for the set of tasks. Another way of describing a schedule is to spell out the rules which determine where the requests of any task in the set will be executed. A scheduling algorithm is a set of rules that determines a schedule for any set of tasks to which the algorithm is applicable. We say that a set of tasks can be feasibly scheduled by a scheduling algorithm(or algorithm), if the algorithm produces a feasible schedule for the set. A set of tasks is said to fully utilize a processor according to a certain scheduling algorithm, if the set of tasks can be feasibly scheduled by that algorithm and increasing the computation time of any one of the tasks in the set would cause the algorithm not to produce a feasible schedule for the set. For a given algorithm, the minimum achievable processor utilization is the minimum of the utilization factor over all task sets that fully utilize the processor. This means that any set of tasks whose utilization factor is less than or equal to the minimum achievable utilization can always be scheduled by the corresponding algorithm. Sets of tasks with larger utilization factor may or may not be scheduled by the corresponding algorithm. Thus, a

possible measure of the "effectiveness" of a scheduling algorithm is the minimum achievable processor utilization of the algorithm. Other things being equal, an algorithm which has higher minimum achievable processor utilization is naturally more effective, since it enlarges the set of feasibly scheduleable task sets.

The problem of devising scheduling algorithm for these type of tasks has attracted the attention of many researchers[4-10]. Scheduling algorithms considered for this problem have been restricted to preemptive priority driven algorithms. In these algorithms, a currently running task of lower priority will be taken off the processor whenever there is a request from a higher priority task, even though the request of the lower priority task has not been completed. The interrupted task is resumed later on. These priority driven scheduling algorithms can be classified into two categories: static priority algorithms, and dynamic priority algorithms. A static(or fixed) priority algorithm is one in which priorities are assigned to tasks first, according to some mechanism, and then scheduling begins. During the process of scheduling, all requests of a task of higher priority have precedence over all requests of a task of lower priority. To get some feeling about a static priority algorithm, let us consider an example. Suppose we have a set of two tasks T_1 and T_2 to be scheduled on a single-

processor computing system, with c_1=1, t_1=2, and c_2=2, t_2=5. If we let T_1 be the higher priority task, then all requests of task T_1 have a priority over any request of T_2. From Figure 1.2(a) we see that such a priority assignment is feasible. But if we let T_2 be the higher priority task, then from Figure 1.2(b) we see that such a priority assignment is not feasible.

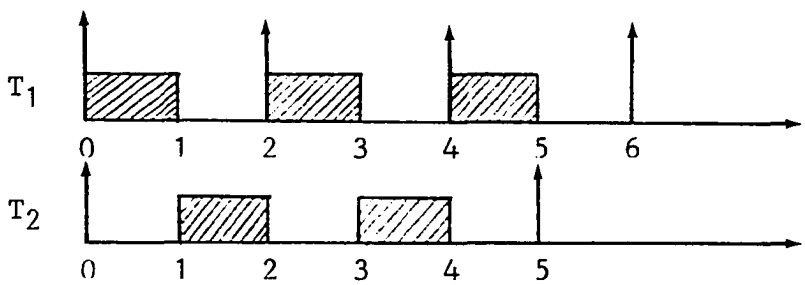
A dynamic priority algorithm, on the other hand, is one in which the priority of a task is a function of time and hence it may vary from one request to another, and even during different times of the same request. We will see one such algorithm in the next Section.

As we mentioned before, each periodic-time-critical task makes an infinite number of requests. By having this fact in mind, how can we tell whether a particular schedule meets the deadline of all the requests of all the tasks in the set? In other words, how can we decide whether the schedule produced by a particular priority algorithm is feasible? Liu and Layland[7] answered this question. Their result is described in Theorem 1.1. But before we get to the theorem, let us define some terms.

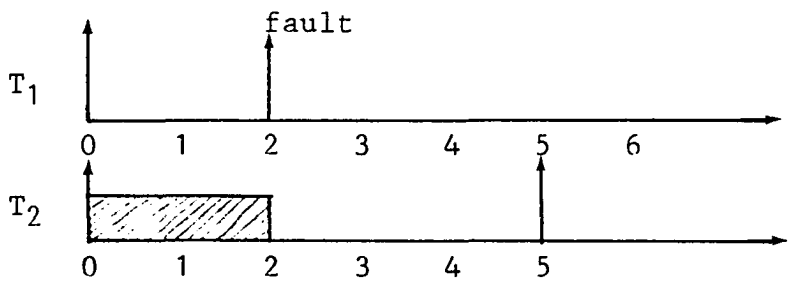
The response time of the request of a task is defined to be the time span between the time the request is made and the time at which the computation of the request has just finished. A critical instant for a task is defined to be an instant at which the request for that task will have

$T_1 = (1, 2)$

$T_2 = (2, 5)$



(a) If T_1 is the higher priority task.



(b) If T_2 is the higher priority task.

Figure 1.2: Schedules for two tasks on a single processor with different priority assignment.

the largest possible response time. Therefore, the schedule produced by a priority algorithm will meet the deadline of all the requests of a task if and only if it meets the deadline of a request made at a critical instant of that task.

Theorem 1.1: [7] A critical instant for any task occurs whenever the task is requested simultaneously with requests for all higher priority tasks.

The proof of Theorem 1.1 follows from the fact that, in a priority driven algorithm, the processing of a request of a task can only be delayed by requests of higher priority tasks. Thus, the maximum delay occurs when a request of a task is made simultaneously with a request of all higher priority tasks.

If all the tasks make their first request at time zero, then time zero will be a critical instant for all the tasks in the set. Therefore, when all tasks make their first request at time zero, if the schedule produced by a priority algorithm meets the deadline of the first request of all the tasks in the set, that schedule is feasible. Because of this fact, the schedule shown in Figure 1.2(a) will meet the deadline of all the requests of both tasks.

We call a scheduling algorithm on-line, if it schedules tasks as they arrive. In other words, the tasks are available one at a time, and the algorithm schedules

each task before the next one becomes available. In contrast, we call a scheduling algorithm off-line, if it has to have all the tasks available beforehand. By this definition, all the scheduling algorithms that have to sort the set of tasks before scheduling them, are in the category of off-line algorithms.

1.4 Dissertation Contents:

In Chapter II we give a survey of previous work on scheduling periodic-time-critical tasks. This Chapter actually prepares us for presenting the results given in chapters III and IV. We first give a survey of the algorithms available for scheduling this type of tasks on single processor systems, and then talk about the problem of scheduling this type of tasks on multiprocessor systems. It is shown in this chapter that the scheduling algorithms which worked well for single processor systems do not perform as well for multiprocessor systems. An alternative approach would be to first partition tasks into different groups, and then schedule the tasks in each group on one processor using one of the scheduling algorithms available for single processor systems. We show that the scheduling algorithm to be used to schedule each group of tasks influences the partitioning process.

In Chapter III we first prove that the problem of partitioning a set of tasks with respect to the best static priority driven algorithm, available for scheduling tasks on single processor systems, is NP-hard. We then present an off-line heuristic algorithm for this problem and analyze its performance and complexity.

In Chapter IV we present two on-line heuristic algorithms for the same problem, and analyze their performance and complexity.

Finally, in Chapter V we give some concluding remarks about our work.

CHAPTER II

A SURVEY OF PREVIOUS WORK

2.1 Scheduling Tasks on a Single-Processor System:

This problem was first studied by Liu and Layland[7] and Serlin[8]. Liu and Layland came up with two scheduling algorithms. One was called the rate-monotonic scheduling algorithm which is a static priority algorithm. This algorithm was referred to as intelligent fixed priority algorithm by Serlin[8]. According to this algorithm priorities to tasks are assigned in decreasing order of their request rates(or in increasing order of their request period). A task with higher request rate is assigned higher priority over a task with lower request rate, regardless of their computation time. Ties are broken arbitrarily. As an example, consider a set of three tasks T_1, T_2, and T_3 with $t_1=3$, $t_2=7$, $t_3=5$, and $c_1=1$, $c_2=2$, $c_3=1$, to be scheduled on a single-processor computing system. Since T_1 makes a request every 3 units of time, T_2 makes a request every 7 units of time, and T_3 makes a request every 5 units of time, therefore the priority order according to the rate-monotonic algorithm would be (T_1, T_3, T_2).

A feasible schedule produced by the rate-monotonic algorithm for this set of tasks is shown in Figure 2.1. All of these three tasks make their first request at time zero. Since the schedule meets the deadline of the first request of all three tasks, therefore, by Theorem 1.1, the schedule shown in Figure 2.1 is a feasible schedule.

Recall that the utilization factor of a task is the fraction of the processor time taken by the task. Therefore, for a set of tasks to be feasibly schedulable on a single processor, it is necessary that its utilization factor be less than or equal to 1. One may ask if this condition is sufficient for the rate-monotonic algorithm to produce a feasible schedule. Unfortunately, as the following example shows, this is not the case. Consider a set of two tasks T_1 and T_2 with $c_1=1$, $t_1=2$, and $c_2=2.5$, $t_2=5$. The total utilization factor of these two tasks is 1. As shown in Figure 2.2, this set of tasks is not feasible under the rate-monotonic scheduling algorithm, simply because we are not able to meet the deadline of the first request of task T_2. Thus, utilization factor of 1 is not a sufficient condition for a set of tasks to be feasible under the rate-monotonic scheduling algorithm. What, then is the sufficient condition? Liu and Layland[7] answered this question by the following theorem.

T_1 = (1, 3)

T_2 = (2, 7)

T_3 = (1, 5)

priority order: (T_1, T_3, T_2)

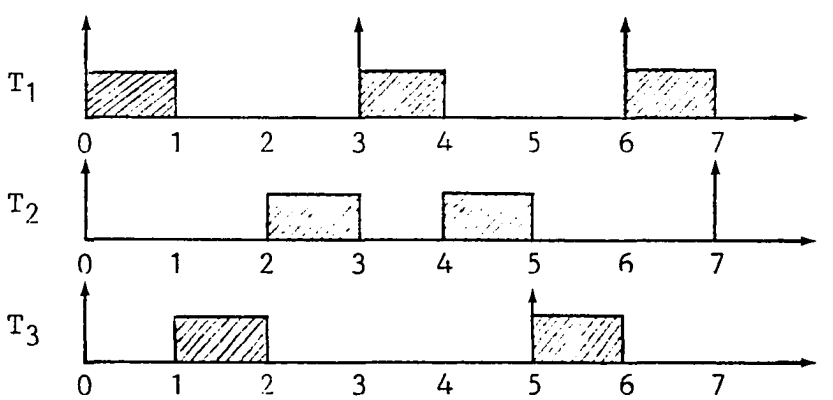


Figure 2.1: A feasible schedule of three tasks, produced by the rate-monotonic scheduling algorithm.

T_1 = (1, 2)

T_2 = (2.5, 5)

priority order: (T_1, T_2)

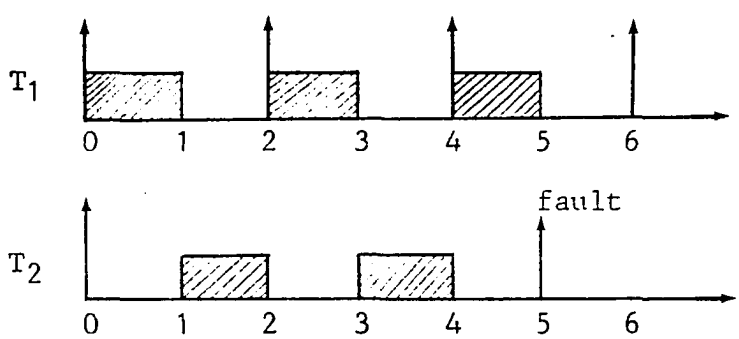


Figure 2.2: An example of two tasks with total utilization factor of 1 which are not feasible under the rate-monotonic scheduling algorithm.

Theorem 2.1: [7] A set of m periodic-time-critical tasks can be feasibly scheduled on a single-processor computing system by the rate-monotonic scheduling algorithm, if the utilization factor of the set is less than or equal to $m(2^{1/m} -1)$, and this bound is tight in the sense that for each m, there exists a set of m tasks with utilization factor $m(2^{1/m} -1)$ which fully utilizes the processor.

By Theorem A.1 of Appendix A, the value of $m(2^{1/m} -1)$ approaches Ln2 when m approaches infinity. It should be noted that Theorem 2.1 provides only a sufficient condition for a set of m tasks to be feasibly scheduled by the rate-monotonic scheduling algorithm. Sets of m tasks with utilization factor greater than $m(2^{1/m} -1)$ may or may not be feasibly scheduled by the rate-monotonic scheduling algorithm. The example in Figure 2.1 shows that a 3-task set with utilization factor greater than $3(2^{1/3} -1)$ can be scheduled feasibly, while the example in Figure 2.3 shows that a 3-task set with utilization factor greater than $3(2^{1/3} -1)$ can not be scheduled feasibly.

Liu and Layland[7] also proved the following theorem about the static priority algorithms.

T_1 = (1, 3)

T_2 = (1, 4)

T_3 = (1.1, 5)

priority order: (T_1, T_2, T_3)

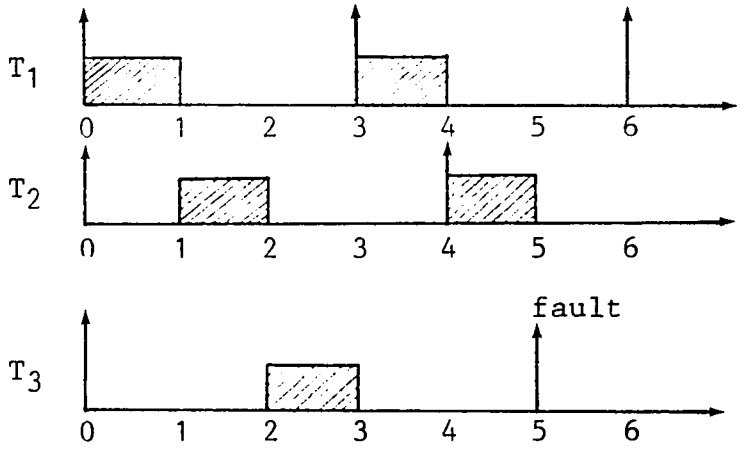


Figure 2.3: An example of a 3-task set, with utilization factor greater than $3(2^{1/3} - 1)$, which is not feasible under the rate-monotonic algorithm.

Theorem 2.2: [7] Among all static priority scheduling algorithms for scheduling a set of periodic-time-critical tasks on a single-processor computing system, the rate-monotonic scheduling algorithm is a best one in the sense that if a set of periodic-time-critical tasks can be feasibly scheduled by any static priority algorithm, then this set can also be scheduled by the rate-monotonic scheduling algorithm.

The other scheduling algorithm considered by Liu and Layland[7] was called the deadline driven scheduling algorithm. This algorithm assigns priorities to requests of tasks according to their deadlines, with the highest priority being given to the request whose deadline is the earliest. If several requests have the same deadlines, then the tie is broken in an arbitrary manner.

As an example consider a set of two tasks T_1 and T_2 with $t_1=2$, $t_2=5$, and $c_1=1$, $c_2=2.5$. A feasible schedule produced by the deadline driven scheduling algorithm for these two tasks is shown in Figure 2.4. This is the same set of tasks which was not feasible under the rate-monotonic scheduling algorithm, shown in Figure 2.2. The reason it is feasible in this case is that when the third request of T_1 arrives at time 4, its deadline is further than the deadline of the first request of T_2. Therefore,

$T_1 = (1, 2)$

$T_2 = (2.5, 5)$

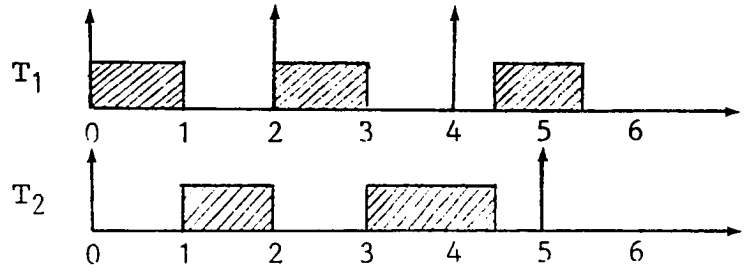


Figure 2.4: A feasible schedule of two tasks under the deadline driven scheduling algorithm.

at time 4, unlike in the previous case, the execution of task T_2 is not interrupted. By doing so, the first request of T_2, as well as the third request of T_1, meet their deadlines, hence making the schedule shown in Figure 2.4 a feasible schedule.

They also came up with the following result regarding the deadline driven scheduling algorithm.

Thoerem 2.3: [7] A set of periodic-time-critical tasks can be feasibly scheduled on a single-processor computing system by the deadline driven scheduling algorithm, if and only if the sum of the utilization factors of all the tasks in the set is less than or equal to 1.

The deadline driven scheduling algorithm is optimal because of the fact that for a set of tasks to be feasibly scheduled on a single processor computing system by any scheduling algorithm, its utilization factor must be less than or equal to 1.

The above two algorithms are the best results obtained, so far, for the problem of scheduling periodic-time-critical tasks on single processor computing systems.

Although dynamic priority algorithms are in general a more effective way of scheduling tasks than static priority algorithms, it is easy to see that they are also more difficult to implement and hence can incur higher system overhead than static priority algorithms. Moreover, it is possible to implement static priority algorithms at the hardware level by the use of priority-interrupt mechanism. Thus, the overhead involved in scheduling tasks can be reduce almost to zero.

2.2 Scheduling tasks on multi-processor systems:

One would naturally hope that a simple extension of the algorithms developed for single-processor computing systems would give satisfactory results when applied to scheduling tasks on multi-processor computing systems. Unfortunately, as the following example shows, this does not turn out to be the case.

A preemptive priority driven scheduling algorithm for a multi-processor computing system works as follows. Priorities are assigned to requests of tasks in the set. Any time a processor is free, it is assigned to an active request with the highest priority, and also if a request is made at an instance when all the processors are busy and there are one or more requests with lower priority being executed at that instance, then this request preempts the request with the lowest priority. Ties are broken arbitrarily.

As in the case of single-processor computing systems, the rate-monotonic scheduling algorithm assigns higher priorities to requests of a task with a higher request rate over requests of a task with a lower request rate, and the deadline driven scheduling algorithm assigns higher priority to a request whose deadline is the earliest. In each case ties are broken arbitrarily.

Example: Consider a set of three tasks T_1, T_2, and T_3 with $c_1=c_2=2e$, $c_3=1$, and $t_1=t_2=1$, $t_3=1+e$, for some $0< e \leq 1/4$, to

be scheduled on a 2-processor computing system. Figure 2.5 shows that this set of tasks is not feasible on a two-processor computing system under the rate-monotonic scheduling algorithm. The reasons being: At time zero when all three tasks make their first requests, both processors P_1 and P_2 are free. Since T_1 and T_2 have higher priorities over T_3, we assign their first requests to P_1 and P_2 at time zero, and their second requests to P_1 and P_2 at time 1. Consequently, both processors will be occupied by tasks T_1 and T_2 during the time intervals [0,2e] and [1,1+2e], leaving a maximum of 1-2e units of time, on each processor, for the first request of T_3 before its deadline at 1+e. Since at any given time only one processor can be working on a given request of a task, the deadline of the first request of T_3 cannot be met. Thus, this set of tasks is not feasible on a two-processor computing system under the rate-monotonic scheduling algorithm.

Figure 2.6 shows that this set of tasks is not feasible on a two-processor computing system under the deadline driven scheduling algorithm either. The reason being: The first request of T_1 and T_2, each with deadline 1, have higher priority over the first request of T_3, with deadline 1+e. Consequently, both processors will be occupied by tasks T_1 and T_2 during the time interval [0,2e], leaving a maximum of 1-e units of time, on each processor, for the first request of T_3 before its deadline

T_1 = (2e, 1)

T_2 = (2e, 1)

T_3 = (1, 1+e)

priority order: (T_1, T_2, T_3)

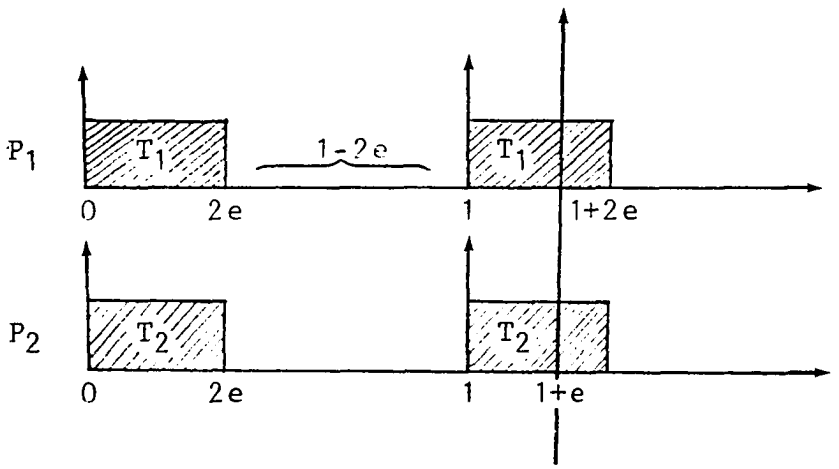


Figure 2.5: Example of a set of 3 tasks which can not be feasibly scheduled on two processors under the rate-monotonic scheduling algorithm.

T_1 = (2e, 1)

T_2 = (2e, 1)

T_3 = (1, 1+e)

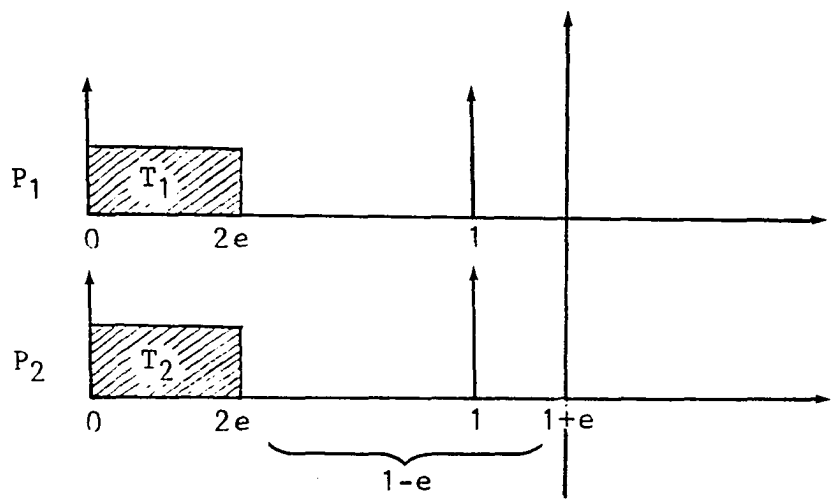


Figure 2.6: Example of a set of 3 tasks which can not be feasibly scheduled on two processors under the deadline driven scheduling algorithm.

at 1+e. Therefore, this set of tasks is not feasible on a 2-processor computing system under the deadline driven scheduling algorithm either.

However, if T_3 is given the highest priority, then this set of tasks can be feasibly scheduled by a static priority algorithm, as shown in Figure 2.7.

This example shows that: (1) the rate-monotonic scheduling algorithm, when applied to scheduling a set of tasks on a multi-processor computing system, is no more optimal amongst all fixed priority scheduling algorithms, and (2) the deadline driven scheduling algorithm is no more optimal amongst all scheduling algorithms when applied to multi-processor computing systems. Therefore, it is desirable to look for better scheduling strategies that will lead to more efficient use of multi-processor computing systems.

The problem of devising optimal algorithms to schedule a set of periodic-time-critical tasks on a fixed number of processors turns out to be a difficult one.

An alternative to this approach is to first partition tasks into different groups, and then schedule the tasks in each group on one processor according to either one of the above algorithms. The scheduling algorithm to be applied to the individual groups in the partition will influence the partitioning process, because each group of tasks must be feasibly schedulable on a single processor according to

T_1 = (2e, 1)

T_2 = (2e, 1)

T_3 = (1, 1+e)

priority order: (T_3, T_1, T_2)

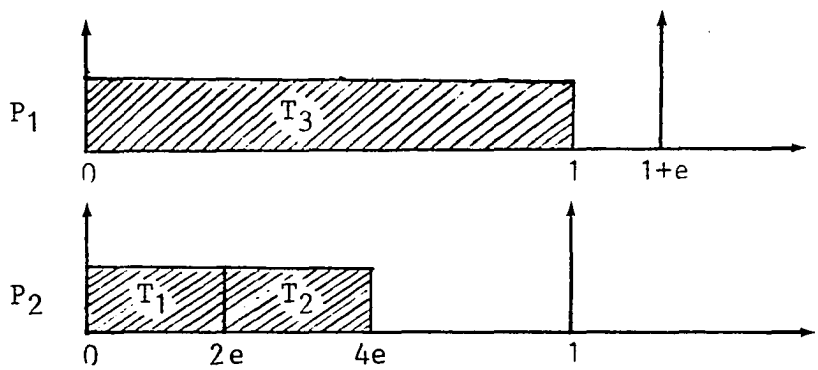


Figure 2.7: A feasible schedule of a set of 3 tasks on two processors under a fixed priority algorithm.

the designated algorithm. This reduces the problem to finding a good partitioning scheme in order to use a minimum number of processors.

2.2.1 Partitioning with respect to the Deadline-Driven Scheduling Algorithm:

Recall that if the utilization factor of a set of tasks is less than or equal to 1, then the set can be feasibly scheduled on a single processor computing system according to the deadline-driven scheduling algorithm. Hence, the problem of partitioning a set of tasks with respect to the deadline-driven scheduling algorithm reduces to the famous bin-packing problem[11-17], where it is desired to pack a set of packages into bins of fixed size so that the sum of the sizes of the packages in a bin does not exceed the size of the bin. This can be seen by imagining a task as a package of size equal to its utilization factor and a processor as a bin of size 1. Thus, all of the results known about the bin-packing problem can also be applied in this case.

The bin-packing problem has been studied extensively since the early 1970s. Since this problem has been shown to be NP-complete[11,12], various heuristic algorithms have been proposed[13-17].

Let N* and N(A) denote, respectively, the number of bins needed by an optimal algorithm and the number of bins

needed by a heuristic algorithm A to pack a given list of packages. Then, the worst-case performance ratio of algorithm A is defined as $\lim_{N^* \to \infty} N(A)/N^*$. This ratio is customarily used to evaluate the performance of a heuristic bin-packing algorithm.

Table 2.1 summarizes all the results known about one-dimensional bin-packing problem.

Table 2.1: The Results of Bin-Packing Algorithms.
(n = number of packages in the list.
W.P.R. = Worst-case Performance Ratio.)

algorithm	time-complexity	W.P.R.
Next-Fit [13]	O(n)	2
First-Fit [13-15]	O(nlogn)	17/10
Best-Fit [13-15]	O(nlogn)	17/10
First-Fit-Decreasing [13-15]	O(nlogn)	11/9
Best-Fit-Decreasing [13-15]	O(nlogn)	11/9
Refined-First-Fit [16]	O(nlogn)	5/3
Refined-First-Fit-Decreasing [16]	O(nlogn)	11/9 - ε
Harmonic [17]	O(n)	1.692
Refined-Harmonic [17]	O(n)	1.636

2.2.2 Partitioning with respect to the Rate-Monotonic Scheduling Algorithm:

The problem of partitioning a set of tasks with respect to the rate-monotonic scheduling algorithm is no longer the same as the bin packing problem. The reasons being:

(1) The total utilization factor of the set of tasks which can be feasibly scheduled on a single processor by the rate-monotonic scheduling algorithm depends on the number of tasks in the set, as well as their relative request periods.

(2) The bound in Theorem 2.1 is only a sufficient one.

The problem of partitioning with respect to the rate-monotonic scheduling algorithm may be considered as a bin-packing problem where the capacity of the bin is not fixed but varies within a certain range. In Chapter 3 we prove that this problem is NP-hard.

The deadline driven scheduling algorithm is a dynamic priority algorithm and, as mentioned before, the implementation of a dynamic priority algorithm requires more overhead than the implementation of a static priority algorithm. Therefore, it will be interesting to investigate into the performance of some "heuristic" algorithms for the partitioning problem with respect to the rate-monotonic scheduling algorithm, which is the best

static priority algorithm.

Two heuristic algorithms for this problem were considered by Dhall and Liu [10]. They called the first one the Rate-Monotonic-Next-Fit algorithm and the second one the Rate-Monotonic-First-Fit algorithm.

According to the Rate-Monotonic-Next-Fit algorithm, tasks are first arranged in non-decreasing order of their request period, and renumbered, if necessary, as T_1, T_2, ..., T_m. Then, task T_1 is assigned to processor P_1, and for each remaining task T_i, $2 \leq i \leq m$, the following actions are taken:

> Assuming that task T_{i-1} is assigned to processor P_j, if T_i can be feasibly scheduled on P_j along with all the tasks already assigned to P_j under the rate monotonic scheduling algorithm, then T_i is assigned to P_j, otherwise T_i is assigned to P_{j+1}.

They proved the following theorem concerning the Rate-Monotonic-Next-Fit algorithm:

Theorem 2.4: [10] Let N be the number of processors required to feasibly schedule a set of tasks by the Rate-Monotonic-Next-Fit algorithm, and N^* be the minimum number of processors required to feasibly schedule the same set of tasks. Then,

$$2.4 \leq \lim_{N^* \to \infty} N/N^* \leq 2.67$$

While partitioning tasks according to the Rate-Monotonic-Next-Fit algorithm, only one processor is considered at a time. Once a task cannot be placed on the processor under consideration, that processor is not used for placing any further task on it, even though some tasks may still be feasibly scheduled on the processor. When this restriction is removed from the Rate-Monotonic-Next-Fit algorithm, then the result is a new algorithm, called the Rate-Monotonic-First-Fit algorithm.

They proved the following theorem concerning the Rate-Monotonic-First-Fit algorithm:

Theorem 2.5: [10] Let N be the number of processors required to feasibly schedule a set of tasks by the Rate-Monotonic-First-Fit algorithm, and N^* be the minimum number of processors required to feasibly schedule the same set of tasks. Then,

$$2 \leq \lim_{N^* \to \infty} N/N^* \leq (4(2^{1/3}))/(1+2^{1/3})$$

It was conjectured that the upper bound can be improved to 2.

The bounds obtained in these algorithms are not very efficient. Moreover, the upper bound in both of these algorithms is not tight, in the sense that no examples are provided which achieve these bounds.

Since these were the only two algorithms available for

the problem of partitioning with respect to the rate-monotonic scheduling algorithm, we decided to pursue this problem further, in search of finding hopefully some "better" heuristic algorithms.

In Chapter III we present a new algorithm which has a better and a tight worst-case performance ratio.

The algorithms given by Dhall and Liu and the algorithm presented in Chapter III are all off-line algorithms. In Chapter IV we present two O(n)-time and O(1)-space on-line algorithms (n is the number of tasks in the set of tasks to be scheduled). The worst-case performance ratio of these algorithms are comparable to those of the off-line algorithms.

CHAPTER III

PARTITIONING TASKS WITH RESPECT TO THE RATE-MONOTONIC SCHEDULING ALGORITHM

3.1 Introduction:

As mentioned in Chapter II, the scheduling algorithms which perform well in the case of single-processor computing systems do not necessarily work as well for multi-processor computing systems. An alternative approach for scheduling periodic-time-critical tasks on multiprocessor systems would be to first partition the set of tasks into different groups, and then schedule the set of tasks of each group on a single processor using one of the algorithms for single-processor systems.

The partitioning problem will depend mainly on the type of scheduling algorithm to be used on each processor. If it is desired to use the deadline-driven algorithm(a dynamic priority algorithm) to schedule the tasks of each group, then the partitioning problem is the same as the famous bin-packing problem[13-17]. But, the problem of partitioning with respect to the rate-monotonic scheduling algorithm is not the same as the bin-packing problem, an

obvious reason being, that a set of tasks with utilization factor of 1 may or may not be feasible on a single processor under the rate-monotonic scheduling algorithm. The total utilization factor of a set of tasks that is feasible under the rate-monotonic algorithm, depends on the number of tasks in the set. The fact that the implementation of a static priority algorithm requires less overhead than the implementation of a dynamic priority algorithm, motivated us to pursue the problem of partitioning tasks with respect to the rate-monotonic scheduling algorithm(the best static priority algorithm). In the next section we will prove that this problem is NP-hard.

3.2 Proof of NP-hardness:

By the following Theorem, we prove that the problem of partitioning with respect to the rate-monotonic scheduling algorithm is NP-hard.

Theorem 3.1: Given a set of periodic-time-critical tasks $S = \{T_1, T_2, \ldots, T_m\}$, with utilization factors $U = \{u_1, u_2, \ldots, u_m\}$, respectively, and a positive integer K, the problem of partitioning this set of tasks into K disjoint subsets $S_1, S_2, \ldots, S_K$ so that each S_i can be feasibly scheduled on a single processor by the rate-monotonic scheduling algorithm is NP-hard.

Proof: To prove this Theorem, we follow the technique described in [11]. We will restrict this problem to the bin-packing problem. The bin-packing problem is known to be NP-complete[11,12], and it is stated as follows:

> Given a finite set $A = \{a_1, a_2, \ldots, a_m\}$ of items with sizes $A_s = \{s(a_1), s(a_2), \ldots, s(a_m)\}$, respectively, a positive integer bin capacity B, and a positive integer K. Is there a partition of A into disjoint sets $A_1, A_2, \ldots, A_K$ such that the sum of the sizes of the items in each A_i is B or less.

According to Theorem 2.1, if the total utilization factors of a set of tasks is less than or equal to Ln2, the set can be feasibly scheduled on a single processor by the rate-monotonic scheduling algorithm. Remember that this

condition was only a sufficient one, meaning if the total utilization factors of a set of tasks is greater than Ln2 and less than or equal to 1, the set may or may not be feasible under the rate-monotonic algorithm. Therefore, if the utilization factor of the set is greater than Ln2 but less than or equal to 1, we need to make some extra checks to find out whether or not it is feasible.

Consider a restricted version of our problem where $0 \leq u_i \leq Ln2$, for all $1 \leq i \leq m$, and the feasibility test for a set of tasks is successful if the utilization factor of the set is less than or equal to Ln2 and it is a failure otherwise. This restricted version of our problem is equivalent to the bin-packing problem by assuming A = S, A_s = U, and B = Ln2.

Since we have shown that a restricted version of our problem is equivalent to the bin-packing problem, therefore our problem must be at least as hard as the bin-packing problem, which is known to be NP-complete. Hence, according to the definition given in [11], our problem is NP-hard.

Q.E.D.

The class of NP-hard problems is such that if there is a method for solving any member of the class that takes time bounded by a polynomial in the size of the input, then every member of the class has such a polynomial time solution. Since problems famous for their computational

intractability, such as the HAMILTONIAN CIRCUIT and TRAVELING SALESMAN problem, are members of this class, it seems probable that in fact no member of this class can be solved in polynomial time.

It is customary to look for heuristic algorithms for NP-hard problems whose goal is only to generate near optimal solutions. The effectiveness of these algorithms is measured by analyzing their worst case behavior. Some of the first work in bounding the worst case behavior of near-optimal algorithms was done as early as 1966 by Graham [19,20], for a different problem in multiprocessor scheduling.

To analyze the worst case behavior of the heuristic algorithms for this problem, we will use the following notations:

Let N* and N(A) denote, respectively, the number of processors needed by an optimal algorithm and the number of processors needed by an algorithm A to schedule a given set of tasks. Then, the worst-case performance ratio of algorithm A, denoted by r(A), is defined as:

$$r(A) = \lim_{N^* \to \infty} \frac{N(A)}{N^*}$$

As it was mentioned in Chapter II, Dhall and Liu[10] considered two heuristic algorithms for this problem. The

worst-case performance ratio of these algorithms were:

$$2.4 \leq r(A) \leq 2.67, \text{ and}$$

$$2 \leq r(A) \leq \frac{4(2^{1/3})}{(1 + 2^{1/3})}$$

These bounds are not tight. In the next section we present a new algorithm whose bound is tight and better than the above bounds.

3.3 The First-Fit-Decreasing-Utilization-Factor Algorithm:

According to the First-Fit-Decreasing-Utilization-Factor(FFDUF) algorithm, the tasks are first arranged in non-increasing order of their utilization factors. Let these rearranged tasks be labelled as T_1, T_2, ..., T_m, and imagine the set of processors as a semi-infinite sequence P_1, P_2, ... The assignment procedure is as follows:

```
1)  set i = 1; N = 1;
2)  while i ≤ m do
        set j = 1; assigned = false;
        repeat
            if T_i is feasible on P_j according to the
            rate-monotonic scheduling algorithm along
            with all the tasks (if any) already assigned
            to P_j then
                assign T_i to P_j;
                set i = i+1; N = max(N,j);
                assigned = true
            else
                set j = j+1
        until assigned
    end-while;
```

The final value of N is the number of processors required to schedule the given set of tasks according to this algorithm.

3.3.1 Worst Case Analysis of the FFDUF Algorithm:

Before we analyze the behavior of FFDUF algorithm, let us prove the following theorem about the rate-monotonic scheduling algorithm. This result will be used in the proof of Theorem 3.3.

Theorem 3.2: Let $T_1 = (c_1, t_1)$ and $T_2 = (c_2, t_2)$ be a set of two tasks with utilization factors $u_1 = c_1/t_1$ and $u_2 = c_2/t_2$, respectively. If $t_1 \leq t_2$ and $u_2 \leq (1-u_1)/(1+u_1)$, then these two tasks can be feasibly scheduled on a single processor by the rate-monotonic scheduling algorithm.

Proof: If T_1 and T_2 make their first request at time zero, then time zero will be the critical instant of T_1 and T_2, by Theorem 1.1. Therefore, if the schedule produced by the rate-monotonic scheduling algorithm meets the deadline of the first request of T_1 and T_2 then, it will also meet the deadline of all the requests of T_1 and T_2.

Let $t_2 = nt_1 + k$.

We may have one of the following two cases:

Case 1: $k \leq c_1$. In this case the largest possible value of c_2 is

$$c_2 = n(t_1 - c_1) = nt_1(1-u_1)$$

$$\text{Thus, } u_2 = \frac{c_2}{t_2} = \frac{nt_1(1-u_1)}{nt_1 + k} = \frac{1 - u_1}{1 + \frac{k}{nt_1}}$$

$$\text{Since } k \leq c_1 \text{ and } n \geq 1, \text{ therefore, } \frac{k}{nt_1} \leq \frac{c_1}{t_1}$$

$$\text{Thus, } u_2 \geq \frac{1-u_1}{1+u_1}$$

Case 2: $k > c_1$. In this case the largest possible value of c_2 is

$$c_2 = n(t_1-c_1) + (k-c_1) = nt_1(1-u_1) + (k-c_1).$$

Lemma 3.1: If $a \geq b > 0$ and $a > x > 0$ then,

$$\frac{b}{a} \geq \frac{b-x}{a-x}$$

Proof: Since $a \geq b$ and $x > 0$, we have

$$ax \geq bx$$

$$(ab-bx) \geq (ab-ax)$$

$$b(a-x) \geq a(b-x)$$

$$\text{or} \quad \frac{b}{a} \geq \frac{b-x}{a-x}$$

Q.E.D.

Since $t_2 \geq c_2 > 0$ and $t_2 > (k-c_1) > 0$, by Lemma 3.1, we have:

$$\frac{c_2}{t_2} \geq \frac{c_2-(k-c_1)}{t_2-(k-c_1)}$$

By substituting the values of c_2 and t_2 in the above inequality, we get

$$u_2 = \frac{c_2}{t_2} \geq \frac{nt_1(1-u_1)}{nt_1+k-(k-c_1)} = \frac{1 - u_1}{1 + \frac{c_1}{nt_1}}$$

Since $\frac{c_1}{nt_1} \leq \frac{c_1}{t_1}$, therefore

$$u_2 \geq \frac{1-u_1}{1+u_1}$$

Q.E.D.

We now proceed to analyze the behavior of FFDUF algorithm by proving the following theorem.

Theorem 3.3: $r(FFDUF) = 2$.

Proof: We define a function f mapping the utilization factors of tasks into the real interval [0,1]. If u is the utilization factor of a task, let

$$f(u) = \begin{cases} 2u, & 0 \leq u \leq 1/2 \\ 1, & 1/2 \leq u \leq 1 \end{cases}$$

Lemma 3.2: Let N^* be the minimum number of processors required to feasibly schedule a set of tasks $\{T_1, T_2, \ldots, T_m\}$ with utilization factors $\{u_1, u_2, \ldots, u_m\}$, respectively. Then,

$$N^* \geq 1/2 \sum_{i=1}^{m} f(u_i)$$

Proof: Let $\{T_{r,1}, T_{r,2}, \ldots, T_{r,k_r}\}$ be the set of tasks with utilization factors $\{u_{r,1}, u_{r,2}, \ldots, u_{r,k_r}\}$, respectively, assigned to processor P_r, $1 \leq r \leq N^*$. Then,

$$\sum_{i=1}^{k_r} f(u_{r,i}) \leq \sum_{i=1}^{k_r} 2u_{r,i} = 2 \sum_{i=1}^{k_r} u_{r,i}, \quad 1 \leq r \leq N^*.$$

Since $\sum_{i=1}^{k_r} u_{r,i} \leq 1$, we have $\sum_{i=1}^{k_r} f(u_{r,i}) \leq 2, \quad 1 \leq r \leq N^*$.

Hence, $\sum_{i=1}^{m} f(u_i) = \sum_{r=1}^{N^*} \sum_{i=1}^{k_r} f(u_{r,i}) \leq 2N^*$.

Which means $N^* \geq 1/2 \sum_{i=1}^{m} f(u_i)$.

Q.E.D.

In what follows we assume the following definitions:

- Let N be the number of processors used by the FFDUF algorithm to schedule a set of m tasks $\{T_1, T_2, \ldots, T_m\}$ with utilization factors $\{u_1, u_2, \ldots, u_m\}$, respectively.

- Let $(P_1, P_2, \ldots, P_N)$ be the set of processors used by the FFDUF algorithm.

- Let $(T_{r,1}, T_{r,2}, \ldots, T_{r,k_r})$ be the set of tasks assigned to processor P_r, $1 \leq r \leq N$.

- Let α_r be the utilization factor of the task with the highest utilization factor amongst all the tasks assigned to processor P_r.

Lemma 3.3: Suppose N processors are used by the FFDUF algorithm to schedule a set of m tasks. if $\alpha_N \leq \text{Ln2-.5}$, then for each processor P_r, $1 \leq r \leq N-1$, we have

$$\sum_{i=1}^{k_r} f(u_{r,i}) \geq 1.$$

Proof: By Theorem 2.1, as long as the utilization factor of a set of tasks is less than or equal to Ln2, it can be scheduled on a single processor by the rate-monotonic scheduling algorithm. Therefore, if there is a task $T_{N,j}$, with $u_{N,j} \leq \text{Ln2-.5}$, assigned to processor P_N by the FFDUF algorithm, then for all the processors P_r, $1 \leq r \leq N-1$, we must have $\sum_{i=1}^{k_r} u_{r,i} > .5$. Otherwise, the task $T_{N,j}$ must have been assigned to the first processor, P_r, found with $\sum_{i=1}^{k_r} u_{r,i} \leq .5$. On the other hand, if for all the processors P_r, $1 \leq r \leq N-1$, we have $\sum_{i=1}^{k_r} u_{r,i} > .5$, then, by the definition of function f, we must have

$$\sum_{i=1}^{k_r} f(u_{r,i}) \geq 1, \quad 1 \leq r \leq N-1.$$

Q.E.D.

Lemma 3.4: Suppose N processors are used by the FFDUF algorithm to schedule a set of tasks. If $Ln2-.5 < \alpha_N \leq 2^{1/3}-1$, then for each processor P_r, $1 \leq r \leq N-1$, we have

$$\sum_{i=1}^{k_r} f(u_{r,i}) \geq 1.$$

Proof: Since the FFDUF algorithm assigns tasks in non-increasing order of their utilization factors, therefore, at the time when the task with utilization factor α_N is being assigned to processor P_N, no task with utilization factor less than or equal to Ln2-.5 has been assigned to any one of the processors in use. At the time when the task with utilization factor α_N is being considered for assignment, let a processor P_r be placed in group 1 if $\alpha_r > 2^{1/3}-1$, and in group 2 if $\alpha_r \leq 2^{1/3}-1$.

For any processor P_r in group 1, for which $\alpha_r \geq .5$, we obviously have $\sum_{i=1}^{k_r} f(u_{r,i}) \geq 1$.

For a processor P_r in group 1, with $2^{1/3}-1 < \alpha_r < .5$, which has at least 3 tasks assigned, we obviously have

$$\sum_{i=1}^{k_r} f(u_{r,i}) = 2 \sum_{i=1}^{k_r} u_{r,i} > 2(2(Ln2-.5) + 2^{1/3}-1) > 1.$$

We next consider those processors P_r in group 1 for which we have $2^{1/3}-1 < \alpha_r < .5$ and each such processor has at most 2 tasks assigned to it.

By Theorem 2.1, if the utilization factor of a set of 3 tasks is less than or equal to $3(2^{1/3}-1)$, it can be scheduled on a single processor by the rate-monotonic scheduling algorithm. Since $.5 + \alpha_N < 3(2^{1/3}-1)$, the utilization factor of the tasks assigned to any such processor must have been greater than .5. For otherwise, the task with utilization factor α_N must have been assigned to one of these processors. Therefore, for any processor P_r of this group we must have

$$\sum_{i=1}^{k_r} f(u_{r,i}) \geq 1.$$

This will exhaust group 1 processors.

We next consider group 2 processors. Since for each task T_i assigned to any processor P_r of this group we have $\text{Ln}2-.5 < u_i \leq 2^{1/3}-1$ and since $\alpha_N \leq 2^{1/3}-1$, we must have had at least 3 tasks assigned to each processor of this group. For otherwise, according to Theorem 2.1, the task with utilization factor of α_N must have been assigned to the first processor found from this group which has two or less tasks assigned to it. On the other hand, if we have at least 3 tasks assigned to every processor of group 2, then for each processor P_r from this group we must have

$$\sum_{i=1}^{k_r} f(u_{r,i}) = 2 \sum_{i=1}^{k_r} u_{r,i} \geq 2*3*(\text{Ln}2-.5) > 1.$$

Q.E.D.

Lemma 3.5: Suppose N processors are used by the FFDUF algorithm to schedule a set of tasks. If $2^{1/3}-1 < \alpha_N \leq 1/3$, then for each processor P_r, $1 \leq r \leq N-1$, we have

$$\sum_{i=1}^{k_r} f(u_{r,i}) \geq 1.$$

Proof: Since the FFDUF algorithm assigns tasks in non-increasing order of their utilization factors, at the time when the task with utilization factor α_N is being assigned to processor P_N, for any task $T_{r,i}$ assigned to any processor P_r, $1 \leq r \leq N-1$, we must have $u_{r,i} > 2^{1/3}-1$.

For all processors P_r, $1 \leq r \leq N-1$, with $\alpha_r \geq .5$ we obviously have $\sum_{i=1}^{k_r} f(u_{r,i}) \geq 1$.

Therefore, we are left to consider only those processors P_r, $1 \leq r \leq N-1$, for which we have $2^{1/3}-1 < \alpha_r < .5$. We claim that any such processor must have at least 2 tasks assigned to it. Suppose there is a processor P_r with only one task $T_{r,1}$ assigned to it. We know that $u_{r,1}$ is less than .5. Since (1-.5)/(1+.5) = 1/3

and since $\alpha_N \leq 1/3$, Therefore by Theorem 3.2, the task with utilization factor α_N must have been assigned to processor P_r. Hence, our claim is true. But if any processor P_r of this group has at least 2 tasks assigned to it, then we obviously have:

$$\sum_{i=1}^{k_r} f(u_{r,i}) = 2 \sum_{i=1}^{k_r} u_{r,i} \geq 2*2*(2^{1/3}-1) > 1.$$

Q.E.D.

Corollary 3.1: Let N be the number of processors used by the FFDUF algorithm to schedule a set of m tasks. If $\alpha_N \leq 1/3$ then,

$$N \leq \sum_{i=1}^{m} f(u_i) + 1.$$

Proof:

Since $\sum_{r=1}^{N-1} \sum_{i=1}^{k_r} u_{r,i} < \sum_{i=1}^{m} u_i$,

therefore, $\sum_{r=1}^{N-1} \sum_{i=1}^{k_r} f(u_{r,i}) \leq \sum_{i=1}^{m} f(u_i)$. (1)

But since $\alpha_N \leq 1/3$, for any processor P_r, $1 \leq r \leq N-1$, by Lemmas 3.3, 3.4, and 3.5, we have

$$\sum_{i=1}^{k_r} f(u_r) \geq 1. \quad (2)$$

From equations (1) and (2) we get

$$N-1 \leq \sum_{i=1}^{m} f(u_i).$$

Which means

$$N \leq \sum_{i=1}^{m} f(u_i) + 1.$$

Q.E.D.

Lemma 3.6: Let N be the number of processors used by the FFDUF algorithm to schedule a set of m tasks. Let K be the number of tasks in the set, each with utilization factor greater than 1/3. If $\alpha_N > 1/3$ Then, $N \leq K$.

Proof: Since the (FFDUF) algorithm schedules tasks in non-increasing order of their utilization factors, at the time when the task with utilization factor α_N is being assigned to processor P_N, for any task $T_{r,i}$ assigned to any processor P_r, $1 \leq r \leq N-1$, we must have $u_{r,i} > 1/3$. This means that at the end of the scheduling process any one of the N processors has at least one task with a utilization factor greater than 1/3 assigned to it. Since we have only K of such tasks and since no task is assigned to more than one processor, therefore, we must have $N \leq K$.

Q.E.D.

Proof of Theorem 3.3: We can have one of the following two cases:

Case 1: $\alpha_N \leq 1/3$. If this is the case, then by corollary 3.1 we have

$$N \leq \sum_{i=1}^{m} f(u_i) + 1,$$

and by Lemma 3.2 we have

$$N^* \geq 1/2 \sum_{i=1}^{m} f(u_i).$$

Therefore, we have

$$\lim_{N^* \to \infty} \frac{N}{N^*} \leq 2.$$

Case 2: $\alpha_N > 1/3$. Let S be the set containing all tasks T_i with $u_i > 1/3$, and $K = |S|$. Then, by Lemma 3.6 we have $N \leq K$. Since no other algorithm can schedule more than 2 tasks in S on a single processor, we have $N^* \geq K/2$. Therefore, in this case also we have

$$\lim_{N^* \to \infty} \frac{N}{N^*} \leq 2.$$

Thus, so far we have shown that $r(FFDUF) \leq 2$.

To finish the proof of Theorem 3.3, we are going to show that for a given $\varepsilon > 0$, There exists an arbitrary large set of tasks for which we have $N/N^* > 2-\varepsilon$.

For a given N, let us choose a set of N tasks as follows, where the first number in each parenthesis is the run-time of the task, and the second number is the request-period of the task:

$$
\begin{aligned}
T_1 &= (1,\ 1 + 2^{1/N}) \\
T_2 &= (2^{1/N} + \delta,\ 2^{1/N}(1 + 2^{1/N})) \\
T_3 &= (2^{2/N} + \delta,\ 2^{2/N}(1 + 2^{1/N})) \\
&\vdots \qquad\qquad \vdots \\
T_N &= (2^{(N-1)/N} + \delta,\ 2^{(N-1)/N}(1 + 2^{1/N}))
\end{aligned}
$$

Where δ is such that no task has utilization factor greater than 1/2.

This set of tasks when scheduled according to the FFDUF algorithm, will require N processors, because no two of these tasks can be feasibly scheduled on a single processor according to the rate-monotonic scheduling algorithm. However, since the utilization factor of each task in this set is less than 1/2, any pair of these tasks can be feasibly scheduled on a single processor according to the deadline driven scheduling algorithm. Thus, all these tasks can be feasibly scheduled on $\lceil N/2 \rceil$ processors. Thus $N^* = \lceil N/2 \rceil$, and so $N/N^* = N/\lceil N/2 \rceil$. Taking N sufficiently large, we can make the ratio $N/N^* > 2 - \varepsilon$.

Therefore,

$$\lim_{N^* \to \infty} \frac{N}{N^*} = 2.$$

Since we had already shown that $r(FFDUF) \leq 2$, we conclude that

$$r(FFDUF) = 2.$$

Q.E.D.

Note that the example given in the proof of Theorem 3.3 proves two facts:

(1) the bound obtained for the worst-case performance ratio of FFDUF algorithm is a tight bound.

(2) No other partitioning scheme based on the rate-monotonic scheduling algorithm can have a better worst-case performance ratio.

We will analyze the time and space complexity of the FFDUF algorithm in the next section.

3.3.2 The Complexity of the FFDUF Algorithm:

The algorithm FFDUF is a two part algorithm. In the first part it sorts the set of tasks to be scheduled in non-increasing order of their utilization factors. In the second part, for each task T_i, $1 \leq i \leq n$ (n = number of tasks in the set), it searches among processors to find one processor P_j to which T_i is feasible along with all the tasks(if any) already assigned to P_j. In the feasibility test, when considering two tasks the result in Theorem 3.2 can be used, and when considering more than two tasks the result in Theorem 2.1 can be used. In either case it needs to make only one comparison. In order to be able to use the results of Theorems 2.1 and 3.2, all it needs to do is to keep track of the number of tasks assigned to each processor, and the total utilization factors of all the tasks assigned to each processor.

The time complexity of the first part(sorting) is O(nlogn) [21].

In the second part(searching) the maximum possible number of processors it may consider would be less than n, the number of tasks in the set. Therefore, the time complexity of searching for each task would be O(n) and for n tasks would be $O(n^2)$.

Thus, the time complexity of FFDUF algorithm is $O(n^2)$.

The space complexity of each part is O(n). The reason being: in the fist part it needs n storage spaces in order to sort a set of n tasks, and in the second part it needs at most n storage spaces, one for each active processor.

Therefore, the space complexity of the FFDUF algorithm is O(n).

3.4 Summary:

In this chapter we first proved that the problem of partitioning tasks with respect to the rate-monotonic algorithm is NP-hard. We then presented a new algorithm (FFDUF) and showed that its worst-case performance ratio is 2, which is an improvement over the worst-case performance ratio of the previously known algorithms (RMNF and RMFF). We also showed that the time complexity of FFDUF algorithm is $O(n^2)$, and that its space complexity is $O(n)$.

Like the FFDUF algorithm, the RMFF algorithm is also a two part algorithm. In the first part it sorts the set of tasks to be scheduled in non-decreasing order of their reqest period, and in the second part for each task it makes a search in First-Fit manner, similar to the search made by the FFDUF algorithm. Therefore, the RMFF algorithm is also an $O(n^2)$-time and $O(n)$-space algorithm.

The RMNF algorithm also sorts the set of tasks to be scheduled in non-decreasing order of their request period. Unlike the other two algorithms, it does not have the complexity of searching. The reason being: when the feasibility test fails on one processor, it will not consider that processor any longer. But since it has to sort, its time complexity is $O(n\log n)$ and its space complexity is $O(n)$.

All of these three algorithms have to sort the set of tasks before scheduling them. Therefore, all of these three algorithms are off-line algorithms. In the next chapter we give two O(n)-time and O(1)-space on-line algorithms for this problem.

CHAPTER IV

ON-LINE ALGORITHMS

4.1 Introduction:

The heuristic algorithms for the problem of partitioning a set of tasks with respect to the rate-monotonic scheduling algorithm given by Dhall and Liu[10] and the one presented in the previous chapter, are all off-line algorithms. For these algorithms to proceed, it is necessary that all the tasks to be scheduled be available before hand. The space complexity of these three algorithms is O(n). Two of these algorithms have a time complexity of $O(n^2)$, and the third one has a time complexity of O(nlogn), where n is the number of tasks in the set of tasks to be scheduled.

In this chapter we present two on-line algorithms. We call these algorithms NEXT-FIT-2 and NEXT-FIT-M. An on-line scheduling algorithm is more difficult than an off-line algorithm because of the fact that the nature of the arriving tasks in an on-line processing is unpredictable. In general, The performance of an on-line algorithm is substantially affected by the permutation of the tasks in a

given set. We will show that the time complexity of these algorithms is O(n) and their space complexity is O(1). We will also show that the worst-case performance ratio of NEXT-FIT-2 is less than 2.4143, and that the worst-case performance ratio of NEXT-FIT-M is less than 2.2838.

In Section 4.4, we show that if the set of tasks to be scheduled does not contain any task with utilization factor in the range $(2^{1/2}-1, 1/2]$, then the worst-case performance ratio of NEXT-FIT-M would be less than 1.911.

4.2 The Algorithm NEXT-FIT-2:

Let $\{T_1, T_2, \ldots, T_n\}$ be the set of tasks with utilization factors $\{u_1, u_2, \ldots, u_n\}$, respectively. Divide this set of tasks into 2 different classes as follows. Let any task T_i belong to class-1 if $u_i > (2^{1/x}-1)$, where x is a positive integer greater than 1, and let it belong to class-2 if $u_i \leq (2^{1/x} - 1)$, as shown below.

class of task	range of utilization factor
1	$(2^{1/x}-1 , 1]$
2	$(0 , 2^{1/x}-1]$

Similarly divide the set of all processors into 2 different classes. A processor designated to process class-k tasks exclusively is referred to as a class-k processor, $1 \leq k \leq 2$. For convenience, let a processor of class-k, $1 \leq k \leq 2$, be called "filled" if it has been used and it is not intended to assign any more tasks to it. Let a processor be called "active" if it is the processor to which the next class-k task will be assigned.

Algorithm NEXT-FIT-2

/* $P_{k,j}$ = the jth processor of class-k */

/* U_k = the total utilization factors of all the tasks assigned to the active processor of class-k */

/* m_k = the number of tasks assigned to the active
processor of class-k */

1. for k = 1 to 2 do
set N_k = 1
set U_k = m_k = 0;
end-for;
2. set i = 1;
3. while i $\leq$ n do
if $u_i > (2^{1/x}-1)$ then /* T_i is a class-1 task */
set k = 1
else /* T_i is a class-2 task */
set k = 2
end-if;
if $U_k > (m_k+1)(2^{1/(m_k+1)}-1) - u_i$ then
set $N_k = N_k+1$;
set $U_k = m_k = 0$
end-if;
assign T_i to P_{k,N_k};
set $U_k = U_k + u_i$;
set $m_k = m_k + 1$;
set i = i + 1
end-while;
4. if $m_k = 0$, $1 \leq k \leq 2$, then
set $N_k = N_k - 1$;

The final values of N_k, $1 \leq k \leq 2$, would be the number of class-k processors used by the algorithm.

4.2.1 The Complexity of NEXT-FIT-2:

For each task T_i, $1 \leq i \leq n$, this algorithm first determines its class by a single test, and then by an additional test it determines whether or not it is feasible on the active processor of its class. If the task is feasible on the active processor of its class, the algorithm assigns it to the processor. Otherwise, it picks a new processor to be the active one. Therefore by a constant amount of computation this algorithm assigns a task to a processor. Hence, the time complexity of NEXT-FIT-2 is O(n).

If we consider a filled processor as the output of the algorithm, then NEXT-FIT-2 needs only 2 storage spaces for two active processors. Therefore, the space complexity of NEXT-FIT-2 is O(1).

4.2.2 Worst-Case Analysis of NEXT-FIT-2:

The upper bound for the worst-case performance ratio of NEXT-FIT-2, denoted by r(NF2), is obtained in two parts. In part I we calculate the upper bound for r(NF2) when x = 2, and in part II we calculate the bound for x > 2. We will assume the following definitions throughout this section.

- Let N^* and N(NF2) denote, respectively, the number of processors needed by an optimal algorithm and the number of processors needed by NEXT-FIT-2 to schedule a given set of tasks.
- Let N_1 and N_2 denote, respectively, the number of class-1 processors and the number of class-2 processors needed by NEXT-FIT-2 to schedule the given set of tasks.
- Let S, S_1, and S_2 denote, respectively, the sum of the utilization factors of all tasks, the sum of the utilization factors of all class-1 tasks, and the sum of the utilization factors of all class-2 tasks in the given set.

Part I: x = 2.

Lemma 4.1: For x = 2, we have $N_1 < \frac{S_1}{(2^{1/2}-1)} + 1$.

Proof: Since the utilization factor of any class-1 task in this case is greater than $(2^{1/2}-1)$ and since any

filled class-1 processor must have at least one class-1 task assigned to it, we conclude that

$$N_1 < \frac{S_1}{(2^{1/2}-1)} \leq \frac{S_1}{(2^{1/2}-1)} + 1.$$

Q.E.D.

Lemma 4.2: For x =2, we have $N_2 < \frac{2S_2}{Ln2} + 2.$

Proof: By Theorem A.1 of Appendix A, the value of $m(2^{1/m}-1)$ approaches Ln2 when m approaches infinity. Therefore, the total utilization factors of the tasks assigned to any two adjacent class-2 processors must be greater than Ln2. For otherwise, the next processor must have been used illegally. Thus, we conclude that:

$$\frac{N_2}{2} < \frac{S_2}{Ln2} \leq \frac{S_2}{Ln2} + 1$$

and

$$N_2 < \frac{2S_2}{Ln2} + 2.$$

Q.E.D.

Corollary 4.1: For x = 2, we have

$$N(NF2) < \frac{2S}{Ln2} + 2.$$

Proof:

$$N(NF2) = N_1 + N_2 < \frac{S_1}{(2^{1/2}-1)} + \frac{2S_2}{Ln2} + 2$$

$$= \frac{2S_1}{2(2^{1/2}-1)} + \frac{2S_2}{Ln2} + 2$$

$$< \frac{2S_1}{Ln2} + \frac{2S_2}{Ln2} + 2$$

$$= \frac{2S}{Ln2} + 2.$$

Q.E.D.

Theorem 4.1: For $x = 2$, we have $r(NF2) \leq \frac{2}{Ln2}$.

Proof: We obviously have $N^* \geq S$, and by Corollary 4.1 we have $N(NF2) < \frac{2S}{Ln2} + 2$. Thus,

$$r(NF2) = \lim_{N^* \to \infty} \frac{N(NF2)}{N^*} \leq \frac{2}{Ln2}.$$

Q.E.D.

Part II: $x > 2$.

Lemma 4.3: For $x > 2$, we have

$$N_1 < \frac{2S_1}{(x-1)(2^{1/(x-1)}-1)} + 4.$$

Proof: Divide all the filled class-1 processors into two groups as follows. Let all filled class-1 processors which have less than (x-1) tasks assigned to them along with their next immediate neighbors belong to group-1 and the rest of filled class-1 processors belong to group-2. The reason for this categorization will become clear shortly. Before we continue further, let us make some additional definitions.

- Let $S_{1,i}$, $1 \leq i \leq 2$, denote the sum of the utilization factors of class-1 tasks assigned to all group-i processors,
- and let $N_{1,i}$, $1 \leq i \leq 2$, be the number of processors in group-i.

Since the utilization factor of any class-1 task is greater than $(2^{1/x}-1)$ and since each group-2 processor has at least (x-1) class-1 tasks assigned to it, we have

$$N_{1,2} < \frac{S_{1,2}}{(x-1)(2^{1/x}-1)} + 1.$$

We next consider group-1 processors. Relabel group-1 processors in increasing order of their index as P_1, P_2 ..., $P_{N_{1,1}}$. Let U_i be the total utilization factors of the tasks assigned to processor P_i, $1 \leq i \leq N_{1,1}$. For now let us assume that $N_{1,1}$ is an even number. If we consider these processors as a group of $N_{1,1}/2$ adjacent pairs, then, by definition, the first processor in each pair has less than (x-1) tasks assigned to it. Let P_i and P_{i+1} be one such pair, where P_i has $m < (x-1)$ tasks assigned to it. Then, we must have $U_i + U_{i+1} > (m+1)(2^{1/(m+1)}-1)$. For otherwise, the assignment of tasks to P_{i+1} is made illegally. Since, by Theorem A.2 of Appendix A, the value of $(m+1)(2^{1/(m+1)}-1)$ decreases as the value of m increases, by substituting the maximum possible value of m in terms of x we get $U_i + U_{i+1} > (x-1)(2^{1/(x-1)}-1)$. Since this is true for any such pair of group-1 processors, we have

$$\frac{N_{1,1}}{2} < \frac{S_{1,1}}{(x-1)(2^{1/(x-1)}-1)} + 1.$$

Therefore, $$N_{1,1} < \frac{2S_{1,1}}{(x-1)(2^{1/(x-1)}-1)} + 2.$$

If $N_{1,1}$ is odd then,

$$\frac{N_{1,1}-1}{2} < \frac{S_{1,1} - U_{N_{1,1}}}{(x-1)(2^{1/(x-1)}-1)} + 1.$$

$$< \frac{S_{1,1}}{(x-1)(2^{1/(x-1)}-1)} + 1$$

Therefore, $$N_{1,1} < \frac{2S_{1,1}}{(x-1)(2^{1/(x-1)}-1)} + 3.$$

We proceed to finish the proof of Lemma 4.4.

$$N_1 = N_{1,1} + N_{1,2}$$

$$< \frac{2S_{1,1}}{(x-1)(2^{1/(x-1)}-1)} + \frac{2S_{1,2}}{(x-1)(2^{1/x}-1)} + 4$$

$$= \frac{2S_{1,1}}{(x-1)(2^{1/(x-1)}-1)} + \frac{2S_{1,2}}{2(x-1)(2^{1/x}-1)} + 4$$

We next show that $(x-1)(2^{1/(x-1)}-1) < 2(x-1)(2^{1/x}-1)$.
By Theorem A.2 of Appendix A, $(x-1)(2^{1/(x-1)}-1)$ has its maximum value when x has its minimum possible value, and, by Theorem A.3, $(x-1)(2^{1/x}-1)$ has its minimum value when x has its minimum possible value. Then, by substituting the minimum possible value of x, which is 3, in both sides of the inequality the proof of our claim becomes obvious.
As a result, we have

$$N_1 < \frac{2S_{1,1}}{(x-1)(2^{1/(x-1)}-1)} + \frac{2S_{1,2}}{(x-1)(2^{1/(x-1)}-1)} + 4$$

$$= \frac{2S_1}{(x-1)(2^{1/(x-1)}-1)} + 4.$$

Q.E.D.

Lemma 4.4: For $x > 2$, we have $N_2 < \frac{S_2}{Ln2-(2^{1/x}-1)} + 1$.

Proof: By Theorem A.1 of Appendix A, as long as the utilization factors of a set of tasks is less than or equal to Ln2 the set can be scheduled on one processor by NEXT-FIT-2. Since any class-2 task has a utilization factor of at most $(2^{1/x}-1)$, the total utilization factors of the tasks assigned to each class-2 processor must be greater than $Ln2-(2^{1/x}-1)$. The proof of lemma then follows from this fact.

Q.E.D.

Corollary 4.2: For $x > 2$, we have

$$N(NF2) < \frac{2S}{(x-1)(2^{1/(x-1)}-1)} + 5$$

Proof:

$$N(NF2) = N1 + N2$$

$$< \frac{2S_1}{(x-1)(2^{1/(x-1)}-1)} + \frac{S_2}{Ln2-(2^{1/x}-1)} + 5$$

$$= \frac{2S_1}{(x-1)(2^{1/(x-1)}-1)} + \frac{2S_2}{2(Ln2-(2^{1/x}-1))} + 5$$

We next show that $(x-1)(2^{1/(x-1)}-1) < 2(Ln2-(2^{1/x}-1))$. The right hand side of this inequality has its minimum value when x has its minimum possible value, and, by

Theorem A.2 of Appendix A, the left hand side has its maximum value also when x has its minimum possible value. By substituting the minimum possible value of x, which is 3, in both sides we prove our claim. As a result, we have

$$N(NF2) < \frac{2S_1}{(x-1)(2^{1/(x-1)}-1)} + \frac{2S_2}{(x-1)(2^{1/(x-1)}-1)} + 5$$

$$= \frac{2S}{(x-1)(2^{1/(x-1)}-1)} + 5$$

Q.E.D.

Theorem 4.2: For $x > 2$, we have

$$r(NF2) \leq \frac{2}{(x-1)(2^{1/(x-1)}-1)}.$$

Proof: We obviously have $N^* \geq S$, and by Corollary 4.2 we have $N(NF2) < \frac{2S}{(x-1)(2^{1/(x-1)}-1)} + 5.$

Thus, $r(NF2) = \lim_{N^* \to \infty} \frac{N(NF2)}{N^*} \leq \frac{2}{(x-1)(2^{1/(x-1)}-1)}.$

Q.E.D.

The numerical values of bounds in Theorems 4.1 and 4.2 for different values of x are shown in Table 4.1. By looking at the values shown in Table 4.1, we see that the best choice for x is 3.

Table 4.1: The upper bounds for worst-case performance ratio of NEXT-FIT-2 for different values of x.

x	upper bound
2	2.8853...
3	2.4142...
4	2.5648...
5	2.6426...
6	2.6900...
⋮	⋮
∞	2.8853...

To establish a lower bound for r(NF2), let us consider the following two examples.

Example-1:

In this example the set of tasks to be scheduled consists of a combination of three different types of tasks:

Type-1: tasks with utilization factors $(2^{1/2}-1)$.

Type-2: tasks with utilization factors $Ln2-(2^{1/2}-1)$.

Type-3: tasks with utilization factors $1/N^2$, where N is a sufficiently large integer.

The appearance of tasks in the list is as follows:

- N repetition of
    - one type-1 task
    - N type-3 tasks
    - one type-2 task
    - N type-3 tasks

An optimal algorithm can schedule this set of tasks on 3N/4 processors as follows:

- N/2 processors each with
    - one type-1 task
    - two type-2 tasks
    - 2N type-3 tasks
- N/4 processors each with
    - two type-1 tasks
    - 4N type-3 tasks

When x = 2, the NEXT-FIT-2 algorithm would use 2N processors as follows:

- N processors each with
  - one type-1 task
  - N type-3 tasks
- N processors each with
  - one type-2 task
  - N type-3 tasks

Therefore, for x = 2, we have $\frac{N(NF2)}{N^*} = \frac{8}{3} = 2.6666...$

When x > 2, the NEXT-FIT-2 algorithm would use N+3 processors as follows:

- N processors each with
  - one type-1 task
  - one type-2 task
- 3 processors for type-3 tasks.

Therefore, for x > 2, we have $\frac{N(NF2)}{N^*} = \frac{4}{3} + \frac{1}{4N}$.

Example-2:

In this example the set of tasks to be scheduled consists of a combination of two different types of tasks:

Type-1: tasks with utilization factors 1/2.

Type-2: tasks with utilization factors 1/3.

The appearance of tasks in the list is as follows:

- N repetition of
    - one type-1 task
    - one type-2 task

An optimal algorithm can schedule this set of tasks on 5N/6 processors as follows:

- N/2 processors each with two type-1 tasks
- N/3 processors each with three type-2 tasks

When x = 2, the NEXT-FIT-2 algorithm would use 3N/2 processors as follows:

- N processors each with one type-1 task
- N/2 processors each with two type-2 tasks

Therefore, for x = 2, we have $\frac{N(NF2)}{N^*} = 1.8$

When x > 2, the NEXT-FIT-2 algorithm would use 2N processors as follows:

- N processors each with one type-1 task
- N processors each with one type-2 task

Therefore, for x > 2 we have $\frac{N(NF2)}{N^*} = 2.4$

The first example establishes a lower bound for r(NF2) when x = 2, and the second example establishes a lower bound for r(NF2) when x > 2.

As we can see from these two examples, the best choice of x will depend on the type of tasks to be scheduled. But in general, when the tasks to be scheduled are not always a specific type, then the choice of x = 3 is the best. The lower bound and the upper bound of r(NF2) for x = 3 (from Table 4.1 and Example-2) are: $2.4 \leq r(NF2) < 2.4143$.

In the next section we present another on-line algorithm, which has the same complexity as NEXT-FIT-2, but its worst-case performance ratio is better than that of NEXT-FIT-2.

4.3 The Algorithm NEXT-FIT-M:

Let $\{ T_1, T_2, \ldots, T_n \}$ be the set of tasks with utilization factors $\{ u_1, u_2, \ldots, u_n \}$, respectively. Let M be a positive integer greater than 2. Divide the set of tasks into M different classes as follows. Let any task T_i belong to class-k if $(2^{1/(k+1)} - 1) < u_i \leq (2^{1/k} - 1)$, for $1 \leq k < M$, and let it belong to class-M if $0 < u_i \leq (2^{1/M} - 1)$, as shown below.

class of task	range of utilization factor
1	$(2^{1/2}-1 , 1]$
2	$(2^{1/3}-1 , 2^{1/2}-1]$
3	$(2^{1/4}-1 , 2^{1/3}-1]$
M	$(0 , 2^{1/M}-1]$

Similarly divide the set of all processors into M different classes. A processor designated to process class-k tasks exclusively is referred to as a class-k processor. Note that since the utilization factor of a task in class-k is less than or equal to $(2^{1/k}-1)$, by Theorem 2.1, at least k class-k tasks can be scheduled by the rate-monotonic scheduling algorithm on one processor. The following algorithm assigns exactly k class-k tasks to each processor(except possibly the last processor) used from class-k, for $1 \leq k < M$. The algorithm assigns class-M tasks to class-M processors so that the total utilization factors of

all the tasks assigned to each class-M processor does not exceed Ln2. For convenience, let a processor of class-k, $1 \leq k \leq M$, be called "filled" if it has been used and it is not intended to assign any more tasks to it. Let a processor be called "active" if it is the processor to which the next class-k task will be assigned. Since NEXT-FIT-M assigns k class-k tasks to each class-k processor and since the utilization factor of any class-k task is greater than $(2^{1/(k+1)} -1)$, $1 \leq k < M$, therefore the total utilization factors of all the tasks assigned to any filled class-k, $1 \leq k < M$, processor is greater than $k(2^{1/(k+1)}-1)$. Also, since any class-M task has a utilization factor of at most $(2^{1/M} -1)$, therefore the total utilization factor of all the tasks assigned to each filled class-M processor must be greater than $Ln2-(2^{1/M}-1)$. Another important property of NEXT-FIT-M is that the number of class-k processors, $1 \leq k < M$, used by the algorithm is independent of the order of arrival of the tasks. In other words, except for class-M processors, any permutation of the tasks in the original list will result in the same number of processors used by NEXT-FIT-M. As we will see, these properties make the analysis of the worst-case performance of NEXT-FIT-M relatively easy.

Algorithm NEXT-FIT-M

/* Let $P_{k,i}$ refer to the ith processor of class-k */

1. for k = 1 to M do set N_k = 1;
2. set i = 1;
3. while i $\leq$ n do
   if T_i is a task from class-k ,$1 \leq k < M$, then
   assign T_i to P_{k,N_k};
   if P_{k,N_k} has currently k tasks assigned to it then
   set $N_k = N_k + 1$
   end-if
   else /* T_i is a task from class-M */
   if the total utilization factors of all the tasks assigned to P_{M,N_M} is greater than Ln2-u_i then
   set $N_M = N_M + 1$
   end-if;
   assign T_i to P_{M,N_M}
   end-if;
   set i = i + 1
   end-while;
4. if P_{k,N_k} ,$1 \leq k \leq M$, has no task assigned to it then
   set $N_k = N_k - 1$;

The final values of N_k, $1 \leq k \leq M$, would be the number of class-k processors used by the algorithm.

4.3.1 The Complexity of NEXT-FIT-M:

For each task T_i ,$1 \leq i \leq n$, this algorithm first determines its class, and then it assigns the task to the active processor of its class. Since the class of a task can be determined in O(logM) time and there is only one active processor in each class at any time, the time-complexity of this algorithm is O(nlogM). As we will see later, the worst-case performance ratio of this algorithm is insensitive to M; a reasonable and a practical range of M is $3 \leq M \leq 12$. Therefore M can be considered as a constant. Hence, the time-complexity of NEXT-FIT-M is O(n).

If we consider a filled processor as the output of the algorithm, then the algorithm needs only M storage spaces for M active processors. Therefore, the space complexity of NEXT-FIT-M is O(1).

4.3.2 Worst-Case Analysis of NEXT-FIT-M:

For a given set of tasks, the total number of processors used by NEXT-FIT-M, denoted by N(NFM), is $\sum_{k=1}^{M} N_k$, where N_k is the number of class-k processors used. Let n_k, $1 \leq k < M$, denote the number of class-k tasks in the set of tasks to be scheduled. then,

$$N(NFM) = \sum_{k=1}^{M} N_k = n_1 + \left\lceil \frac{n_2}{2} \right\rceil + \left\lceil \frac{n_3}{3} \right\rceil + \ldots + \left\lceil \frac{n_{M-1}}{M-1} \right\rceil + N_M$$

Let U_M denote the total utilization factors of all the class-M tasks in the set of tasks to be scheduled. Since the total utilization factors of the tasks assigned to each filled class-M processor is greater than $Ln2-(2^{1/M}-1)$, we have

$$N_M < \frac{U_M}{Ln2-(2^{1/M}-1)}.$$

Thus,

$$N(NFM) < n_1 + \frac{n_2}{2} + \frac{n_3}{3} + \ldots + \frac{n_{M-1}}{M-1} + \frac{U_M}{Ln2-(2^{1/3}-1)} + (M-1) \quad (1)$$

The algorithm NEXT-FIT-M assigns 1 class-1 task to each class-1 processor it uses, and it assigns k class-k tasks to each class-k processor (except possibly the last one) it

uses, for $2 \leq k < M$. Therefore, asymptotically, each class-k task costs NEXT-FIT-M $1/k$ processors, for $1 \leq k < M$, and each class-M task T_i with utilization factor u_i costs NEXT-FIT-M at most $u_i/(\text{Ln2}-(2^{1/M}-1))$ processors. We therefore define a cost function f as:

$$f(u_i) = \begin{cases} \dfrac{1}{k}, & \text{if } T_i \text{ is a class-k task and } 1 \leq k < m \\ \dfrac{u_i}{\text{Ln2}-(2^{1/M}-1)}, & \text{if } T_i \text{ is a class-M task} \end{cases}$$

Thus, in terms of the cost function f we can rewrite (1) as

$$N(NFM) < \sum_{i=1}^{n} f(u_i) + (M-1) \qquad (2)$$

The term (M-1) in (2) accounts for the last processors used from class-k, $2 \leq k \leq m$, for which we may not have enough class-k tasks to assign to them.

Furthermore, if T_i is a class-k task ,for $1 \leq k < M$, then we have $(2^{1/(k+1)} - 1) < u_i \leq (2^{1/k} - 1)$ and $f(u_i) = 1/k$. therefore,

$$\frac{f(u_i)}{u_i} = \frac{1}{ku_i} < \frac{1}{k(2^{1/(k+1)} - 1)}.$$

Since, by Theorem A.3 of Appendix A, $\dfrac{1}{k(2^{1/(k+1)} - 1)}$ is

monotonically decreasing with k, we have the following inequality:

$$\frac{f(u_i)}{u_i} < \frac{1}{k(2^{1/(k+1)}-1)}, \quad \text{if } u_i \leq (2^{1/k}-1) \text{ and } 1 \leq k < M \qquad (3)$$

Consider an optimal algorithm that uses N* processors to schedule a set of n tasks. Let $\{T_{i1}, T_{i2}, \ldots, T_{it_i}\}$ be the set of tasks, with utilization factors $\{u_{i1}, u_{i2}, \ldots, u_{it_i}\}$, assigned to the ith processor by the optimal algorithm, for $1 \leq i \leq N^*$. Since the total utilization factors of all the tasks assigned to each processor by any algorithm can not exceed 1, we have

$$\sum_{j=1}^{t_i} u_{ij} \leq 1, \quad 1 \leq i \leq N^*.$$

In terms of the cost function f, the number of processors used by NEXT-FIT-M, for a given M, to schedule $\{T_{i1}, T_{i2}, \ldots, T_{it_i}\}$ would be

$$F_{i,M} = \sum_{j=1}^{t_i} f(u_{ij}), \quad 1 \leq i \leq N^*.$$

Let $T' = \{T'_1, T'_2, \ldots, T'_t\}$ be a set of tasks with utilization factors $\{u'_1, u'_2, \ldots, u'_t\}$, respectively, with the following properties:

(1) $u'_m > 0, \quad 1 \leq m \leq t$

(2) $\sum_{m=1}^{t} u'_m \leq 1$

$$(3)\quad F_M = \sum_{m=1}^{t} f(u'_m) \geq F_{i,M},\ \text{for } 1 \leq i \leq N^*$$

Then, we can rewrite (2) as:

$$N(NFM) < \sum_{i=1}^{N^*} \sum_{j=1}^{t_i} f(u_{ij}) + (M-1)$$

$$\leq N^* F_M + (M-1)$$

This implies that F_M is the worst-case performance ratio of NEXT-FIT-M since,

$$r(NFM) = \lim_{N^* \to \infty} \frac{N(NFM)}{N^*} \leq F_M$$

Therefore, we will concentrate on finding the bound on the number $F_M = \sum_{m=1}^{t} f(u'_m)$, given $\sum_{m=1}^{t} u'_m \leq 1$.

Since any permutation of the tasks in the original list will result in the same number of class-k, $1 \leq k < m$, processors used by NEXT-FIT-M, we assume, without loss of generality, that $u'_1 \geq u'_2 \geq u'_3 \geq \cdots \geq u'_t$.

Define a sequence K_i, as follows:

$$K_1 = K_2 = 1$$

and to calculate K_i, for $i > 2$, find the smallest integer S $(=K_i)$ so that the following inequality holds:

$$(2^{1/(S+1)} - 1) < 1 - \sum_{j=1}^{i-1} (2^{1/(K_j+1)} - 1) \qquad (4)$$

Some of the values of K_i so calculated are shown in table (4.2). It is not hard to see that, for $i > 2$

$$(2^{1/K_i} - 1) \geq 1 - \sum_{j=1}^{i-1} (2^{1/(K_j+1)} - 1) \qquad (5)$$

Table 4.2: Values of K_i in Worst-Case Analysis of NEXT-FIT-M.

i	K_i
1	1
2	1
3	4
4	30
5	2,635
6	7,145,847
.	.
.	.
.	.

Theorem 4.3: For $M > 2$ and $K_i < M \leq K_{i+1}$, Where K_i is given by (4) we have

$$F_M = \sum_{m=1}^{t} f(u'_m) < \sum_{j=1}^{i} \frac{1}{k_j} + \frac{1 - \sum_{j=1}^{i} (2^{1/(k_j+1)} - 1)}{Ln2 - (2^{1/M} - 1)}$$

Proof: Let $S_M = \sum_{j=1}^{i} \frac{1}{k_j} + \frac{1 - \sum_{j=1}^{i} (2^{1/(k_j+1)} - 1)}{Ln2 - (2^{1/M} - 1)}$.

The numerical values of S_M for various values of M are listed in Table (4.3). We may have one of the following three cases:

Case 1: T'_1 is not a class-1 task. Then, $u'_m \leq (2^{1/2} - 1)$, $1 \leq m \leq t$.

By (3) we have $f(u'_m) < \frac{u'_m}{2(2^{1/3} - 1)}$, $1 \leq m \leq t$.

Therefore, $\sum_{m=1}^{t} f(u'_m) < \frac{\sum_{m=1}^{t} u'_m}{2(2^{1/3} - 1)}$

$$\leq \frac{1}{2(2^{1/3} - 1)}$$

$$< 1.9237 < S_M$$

Case 2: T'_1 is a class-1 task but T'_2 is not a class-1 task. Then,

$$\sum_{m=2}^{t} u'_m < 1 - (2^{1/2} - 1), \quad \text{and} \quad u'_m \leq (2^{1/2} - 1), \; 2 \leq m \leq t.$$

By (3) we have $f(u'_m) < \dfrac{u'_m}{2(2^{1/3} - 1)}, \quad 2 \leq m \leq t.$

$$\text{Therefore,} \quad \sum_{m=1}^{t} f(u'_m) < f(u'_1) + \frac{\sum_{m=2}^{t} u'_m}{2(2^{1/3} - 1)}$$

$$< 1 + \frac{1 - (2^{1/3} - 1)}{2(2^{1/3} - 1)}$$

$$< 2.1269 < S_M$$

Case 3: Both T'_1 and T'_2 are class-1 tasks. First, we consider the case where $T'_1 \in$ class-K_1, $T'_2 \in$ class-K_2, ..., $T'_i \in$ class-K_i. Then,

$$\sum_{m=i+1}^{t} u'_m < (1 - \sum_{j=1}^{i} (2^{1/(K_j+1)} - 1)).$$

Since $M \leq K_{i+1}$, we have $T'_m \in$ class-M for all $i+1 \leq m \leq t$. Therefore,

$$F_M = \sum_{m=1}^{t} f(u'_m) < \sum_{j=1}^{i} \frac{1}{kj} + \frac{1 - \sum_{j=1}^{i} (2^{1/(k_j+1)} - 1)}{\text{Ln}2 - (2^{1/M} - 1)} = S_M$$

We next show that F_M will have its maximum value when $T_j \in$ class-K_j, for $1 \leq j \leq i$.

Suppose $T_j \notin$ class-K_j for some $j < i$. Then we have $u_m \leq (2^{1/(K_j+1)}-1)$, for all $j \leq m \leq t$. This would reduce the above value of F_M by $1/K_j$, and it would increase the value of F_M by at most $\dfrac{(2^{1/(K_j+1)}-1)}{(K_j+1)(2^{1/(K_j+2)}-1)}$. We will show that

$$\frac{(2^{1/(K_j+1)}-1)}{(K_j+1)(2^{1/(K_j+2)}-1)} < \frac{1}{K_j}.$$

Since, by Theorem A.3 of Appendix A, $K(2^{1/(K+1)}-1)$ is monotonically increasing with K, we have

$$K_j(2^{1/(K_j+1)}-1) < ((K_j+1)(2^{1/(K_j+2)}-1))$$

If we divide both sides of the above inequality by $K_j(K_j+1)(2^{1/(K_j+2)}-1)$, we get

$$\frac{(2^{1/(K_j+1)}-1)}{(K_j+1)(2^{1/(K_j+2)}-1)} < \frac{1}{K_j}.$$

Since this is true for any $j \leq i$, therefore, F_M has its maximum value when we have $T_j \in$ class-K_j, for all $1 \leq j \leq i$.

Q.E.D.

Table 4.3: Values of S_M in Worst-Case Analysis of NEXT-FIT-M

M	S_M
3	2.3960...
4	2.3404...
5	2.2920...
6	2.2900...
7	2.2888...
8	2.2879...
9	2.2873...
10	2.2868...
11	2.2864...
12	2.2860...
13	2.2858...
30	2.2841...
31	2.2837...
∞	2.2837...

By looking at the values of F_∞ and F_{12} (Table 4.3), we see that little improvement on performance is achieved beyond $M = 12$. Therefore, the region $3 \leq M \leq 12$ is recommended in practice.

To show how close the bound in Theorem 4.3 is to a tight bound, we consider the following example:

- Let $M = K_{i+1}$, for some $i \geq 2$.
- Let N be an integer divisible by M, and e be a sufficiently small quantity.
- Consider a set of tasks consisting of:

  N tasks each with utilization factor

  $u_j = (2^{1/(K_j+1)} - 1 + e)$, for $1 \leq j \leq i$, and

  N tasks each with utilization factor

  $$u_{i+1} = (1 - \sum_{j=1}^{i} u_j - ie).$$

Since $M = K_{i+1}$, by (4) and (5) we have

$$(2^{1/(M+1)}-1) < u_{i+1} < (2^{1/M}-1).$$

Therefore, NEXT-FIT-M assigns exactly M tasks, each with utilization factor u_{i+1}, to each class-M processor it uses, and it assigns K_j tasks each with utilization factor u_j to each class-K_j processor it uses, for $1 \leq j \leq i$. Thus, the total number of processors needed by NEXT-FIT-M would be

$$N(NFM) = \sum_{j=1}^{i} \frac{N}{K_j} + \frac{N}{M}$$

Since $\sum_{j=1}^{i+1} u_j = 1$, the number of processors needed by an optimal algorithm is N.

Therefore, $$\frac{N(NFM)}{N^*} = \sum_{j=1}^{i} \frac{1}{K_j} + \frac{1}{M} = \sum_{j=1}^{i+1} \frac{1}{K_j}.$$

Let $Q_M = \sum_{j=1}^{i+1} \frac{1}{K_j}$. The numerical values of Q_M for various values of M, along with those of S_M in Theorem 4.3, is shown below.

M	Q_M	S_M
3	2.2500...	2.3960...
4	2.2500...	2.3404...
30	2.2833...	2.2841...
2635	2.2837...	2.2837...
⋮	⋮	⋮
∞	2.2837...	2.2837...

Therefore, the bound in Theorem 4.3 is very close to the tight bound for small values of M, and it is tight for large values of M.

4.4 A Special Case:

In this section we show that if the set of tasks to be scheduled does not contain any task with utilization factor in the range of $(2^{1/2}-1 , 1/2]$, then $r(NFM) < 1.911$.
The classification of tasks in this case are:

class	range of utilization factors
1	$(1/2 , 1]$
2	$(2^{1/3}-1 , 2^{1/2}-1]$
3	$(2^{1/4}-1 , 2^{1/3}-1]$
4	$(2^{1/5}-1 , 2^{1/4}-1]$
⋮	⋮
M	$(0 , 2^{1/M}-1]$

It is not hard to see that every thing we said in section 4.3.2, up to the Theorem 4.3, is also true for this case, except the values of K_i which are defined as follows:

$K_1 = 1$

$K_2 = 2$

and to calculate K_i, for $i>2$, find the smallest integer S $(=K_i)$ so that the following inequality holds:

$$(2^{1/(S+1)} -1) \quad < \quad 1/2 - \sum_{j=2}^{i-1} (2^{1/(K_j+1)} -1) \qquad (6)$$

Some of the values of K_i so calculated are shown in table (4.4). It is not hard to see that, for $i > 2$,

$$(2^{1/K_i} -1) \quad \geq \quad 1/2 - \sum_{j=2}^{i-1} (2^{1/(K_j+1)} -1) \qquad (7)$$

Table 4.4: Values of K_i in Worst-Case Analysis of NEXT-FIT-M, for the special case.

i	K_i
1	1
2	2
3	3
4	13
5	6,017
.	.
.	.
.	.

Let T' be the set of tasks satisfying all the conditions given in Section 4.3.2 for T', except that it does not contain any task with utilization factor in the range of $(2^{1/2}-1, 1/2]$.

We next prove the following Theorem:

Theorem 4.4: For $M > 2$ and $K_i < M \leq K_{i+1}$, where K_i is given by (6), we have

$$F_M = \sum_{m=1}^{t} f(u'_m) < \sum_{j=1}^{i} \frac{1}{K_j} + \frac{\frac{1}{2} - \sum_{j=2}^{i} (2^{1/(K_j+1)} - 1)}{Ln2 - (2^{1/M} - 1)}$$

Proof: Let $S_M = \sum_{j=1}^{i} \frac{1}{K_j} + \frac{\frac{1}{2} - \sum_{j=2}^{i} (2^{1/(K_j+1)} - 1)}{Ln2 - (2^{1/M} - 1)}$

The numerical values of S_M for various values of M are listed in Table (4.5).

Table 4.5: Values of S_M in Worst-Case Analysis of NEXT-FIT-M, for the special case.

M	S_M
3	2.0541...
4	1.9342...
5	1.9267...
6	1.9224...
7	1.9196...
8	1.9177...
9	1.9163...
10	1.9152...
11	1.9143...
12	1.9136...
13	1.9130...
14	1.9104...
∞	1.9104...

We may have one of the following three cases:

Case 1: $T'_1 \in$ class-K, for some $K \geq 3$.

Then, $u'_m \leq (2^{1/3} - 1)$, for all $1 \leq m \leq t$.

By (3) we have $f(u'_m) < \dfrac{u'_m}{3(2^{1/4} - 1)}$, $1 \leq m \leq t$.

Therefore, $F_M = \sum_{m=1}^{t} f(u'_m) < \dfrac{\sum_{m=1}^{t} u_m}{3(2^{1/4} - 1)}$

$$\leq \frac{1}{3(2^{1/4} - 1)}$$

$$< 1.762$$

$$< S_m.$$

Case 2: $T_1 \in$ class-2. First, if $T_2 \notin$ class-2 then, $u_m \leq (2^{1/3} - 1)$, for all $2 \leq m \leq t$, and

$$\sum_{m=2}^{t} u_m < 1 - (2^{1/3} - 1).$$

By (3) we have $f(u'_m) < \dfrac{u'_m}{3(2^{1/4} - 1)}$, $2 \leq m \leq t$.

$$\text{Therefore, } F_M = \sum_{m=1}^{t} f(u'_m) < f(u'_1) + \frac{\sum_{m=2}^{t} u'_m}{3(2^{1/4}-1)}$$

$$< \frac{1}{2} + \frac{1 - (2^{1/3}-1)}{3(2^{1/4}-1)}$$

$$< 1.804$$

$$< S_M.$$

Next, if T'_2 also $\in$ class-2 then,

$$\sum_{m=3}^{t} u'_m < 1 - 2(2^{1/3}-1), \quad \text{and} \quad u'_m < (2^{1/4}-1), \quad 3 \leq m \leq t.$$

By (3) we have $f(u'_m) < \dfrac{u'_m}{4(2^{1/5}-1)}$, $3 \leq m \leq t$.

Therefore,

$$F_M = \sum_{m=1}^{t} f(u'_m) < f(u'_1) + f(u'_2) + \frac{\sum_{m=3}^{t} f(u'_m)}{4(2^{1/5}-1)}$$

$$< \frac{1}{2} + \frac{1}{2} + \frac{1 - 2(2^{1/3}-1)}{4(2^{1/1/5}-1)}$$

$$< 1.874$$

$$< S_M$$

Case 3: $T'_1 \in$ class-1. Let us first consider the case where $T'_m \in$ class-K_m, for all $1 \leq m \leq i$.

Then, $\sum_{m=i+1}^{t} u'_m < (\frac{1}{2} - \sum_{j=2}^{i} (2^{1/(K_j+1)} - 1))$.

Since $M \leq K_{i+1}$, we have $T'_m \in$ class-M, for all $i < m \leq t$.

By (3) we have

$$F_M = \sum_{m=1}^{t} f(u'_m) < \sum_{m=1}^{i} \frac{1}{K_m} + \left(\frac{1}{2} - \frac{\sum_{m=2}^{i} (2^{1/(K_m+1)} - 1)}{Ln2 - (2^{1/M} - 1)}\right)$$

$$= S_M.$$

By an argument similar to the one given in section 4.3.2, it is not hard to see that in this case, also, F_M will have its maximum value when $T_m \in$ class-K_m, for all $1 \leq m \leq i$.

Q.E.D.

To show how close the bound in Theorem 4.2 is to a tight bound, we consider the following example:

- Let $M = K_{i+1}$
- Let N be an integer divisible by M, and e be a sufficiently small quantity.
- Consider a set of tasks consisting of:

  N tasks each with utilization factor

$$u_1 = (\frac{1}{2} + e),$$

N tasks each with utilization factor

$u_j = (2^{1/(K_j+1)}-1 + e)$, for $2 \leq j \leq i$, and

N tasks each with utilization factor

$$u_{i+1} = \frac{1}{2} - \sum_{j=2}^{i} (2^{1/(K_j+1)}-1 - ie).$$

By (6) and (7) we have $(2^{1/(M+1)} -1) < u_{i+1} < (2^{1/M} -1)$. Therefore, NEXT-FIT-M assigns exactly M tasks each with utilization factor u_{i+1} to each class-M processor it uses, and it assigns K_j tasks to each class-K_j processor it uses, for $1 \leq j \leq i$. Therefore the total number of processors needed by NEXT-FIT-M for the above set of tasks would be

$$N(NFM) = \sum_{j=1}^{i} \frac{N}{K_j} + \frac{N}{M}.$$

Since $\sum_{j=1}^{i+1} u_j = 1$, the number of processors needed by an optimal algorithm is N.

Therefor, $$\frac{N(NFM)}{N^*} = \sum_{j=1}^{i} \frac{1}{K_j} + \frac{1}{M} = \sum_{j=1}^{i+1} \frac{1}{K_j}.$$

Let $Q_M = \sum_{j=1}^{i+1} \frac{1}{K_j}$. The numerical values of Q_M for various

values of M, along with those of S_M in Theorem 4.4, is shown below.

M	Q_M	S_M
3	1.8333...	2.0541...
13	1.9102...	1.9130...
6017	1.9104...	1.9104...
⋮	⋮	⋮
∞	1.9104...	1.9104...

Therefore, the bound in theorem 4.4 is very close to the tight bound for small values of M, and it is tight for large values of M.

4.5 Summary:

In this chapter we have presented two O(n)-time and O(1)-space on-line algorithms for the problem of partitioning a set of periodic-time-critical tasks with respect to the rate-monotonic scheduling algorithm. These algorithms are less complex than the existing off-line algorithms, and their worst-case performance ratio are better than that of the existing off-line algorithm RATE-MONOTONIC-NEXT-FIT, and comparable to those of the off-line algorithms RATE-MONOTONIC-FIRST-FIT and FIRST-FIT-DECREASING-UTILIZATION-FACTOR.

We also showed that if the set of tasks to be scheduled does not contain any task with utilization factor in the range $(2^{1/2}-1, 1/2]$, then the worst-case performance ratio of NEXT-FIT-M is better than that of any existing algorithm for this problem.

The algorithms presented in this chapter are the only on-line algorithms known for this problem to date.

CHAPTER V

CONCLUSION

In this dissertation we studied the problem of partitioning a set of periodic-time-critical tasks into different groups, subject to the conditions that: i) each group of tasks can be feasibly scheduled on a single processor using the rate-monotonic scheduling algorithm (which is the best static priority driven algorithm available for scheduling this type of tasks on a single processor system); and, ii) the number of processors required is minimum.

In Chapter III we first proved that this problem is NP-hard, and then presented a new heuristic off-line algorithm, called First-Fit-Decreasing-Utilization-Factor (FFDUF). The time complexity of FFDUF was shown to be $O(n^2)$, and its space complexity was shown to be $O(n)$, where n is the number of tasks in the set of tasks to be scheduled. The worst-case performance ratio of FFDUF was shown to be 2, which is an improvement over the worst-case performance ratio of the two existing off-line algorithms.

There exists no on-line algorithm for this problem. Since the nature of the arriving tasks in an on-line processing is unpredictable, an on-line scheduling algorithm is expected to be more difficult than an off-line scheduling algorithm, and in general the performance of an on-line scheduling algorithm is substantialy affected by the permutation of the tasks in a given set.

In Chapter IV we presented two O(n)-time and O(1)-space on-line algorithms for this problem. We called these algorithms NEXT-FIT-2 and NEXT-FIT-M. These algorithms are less complex than the off-line algorithms. The worst-case performance ratio of NEXT-FIT-2 was shown to be less than 2.4143, and the worst-case performance ratio of NEXT-FIT-M was shown to be less than 2.2838. These ratios are comparable to those of the existing off-line algorithms. We further showed that if the set of tasks to be scheduled does not contain any tasks with utilization factor in the range $(2^{1/2}-1, 1/2]$, then the worst-case performance ratio of NEXT-FIT-M would be less than 1.911 which is better than those of the available off-line algorithms.

Table 5.1 summarizes our results and the results of the two previously known heuristic algorithms for this problem.

Table 5.1: A Summary of all the Known Results.
n = number of tasks to be scheduled.
S.C. = Space Complexity.
T.C. = Time Complexity.
W.C.P.R. = Worst-Case Performance Ratio.

Algorithm	Type	S.C.	T.C.	W.C.P.R.
(RMNF)	off-line	O(n)	O(nlogn)	$2.4 \leq r(RMNF) \leq 2.67$
(RMFF)	off-line	O(n)	$O(n^2)$	$2 \leq r(RMFF) < 2.24$
(FFDUF)	off-line	O(n)	$O(n^2)$	$r(FFDUF) = 2$
(NF2)	on-line	O(1)	O(n)	$r(NF2) = 2.4142...$
(NFM)	on-line	O(1)	O(n)	$r(NFM) = 2.2837...$ $r(NFM) = 1.9104...$*

* When the set of tasks does not contain any task with utilization factor in the range $(2^{1/2}-1, 1/2]$.

5.1 Suggestions for Future Research:

In Chapter III we showed by an example that no partitioning algorithm with respect to the rate-monotonic algorithm can have a better worst-case performance ratio than 2. Therefore, the bound obtained for FFDUF can not be improved. But, one may try to look for a less complex off-line algorithm, or look for an algorithm that distributes tasks evenly among processors.

The worst-case performance ratio of NEXT-FIT-2 is not shown to be tight in this study. It will be interesting to see whether this bound is really tight. If not, can it be improved further? It may also be interesting to look for other equally good or better on-line algorithms.

It would also be interesting to analyze the performance of NEXT-FIT-2 and the off-line algorithms when they are applied to task sets that do not contain any task with utilization factor in the range $(2^{1/2}-1, 1/2]$.

All of the scheduling algorithms considered for this problem, so far, are preemptive algorithms. It would be interesting to investigate the behavior of non-preemptive algorithms for this problem.

REFERENCES

1. Martin, J., "Programming Real-Time Computer Systems," Prentice-Hall, Englewood Cliffs, N.J., 1965.

2. Duncan A. M., "Real-Time Computing with Applications to Data Acquisition and Control," Van Nostrand Reinhold Company Inc., New York, N.Y. 10020.

3. Manacher, G.K., "Production and Stabilization of Real-Time Task Schedules," J. ACM 14, 3 (July 1967), pp. 439-465.

4. Labetoulle, J., "Some Theorems on Real-Time Scheduling," in Computer Architecture and Networks, pp. 285-298, E.Gelenbe and R. Mahl (Eds.), North Holland Publishing Co., 1974.

5. Leung, Y.T., and Merrill, M.L., "A Note On Preemptive Scheduling of Periodic, Real-Time Tasks," Inform. Process. Lett., Vol. 11, no. 3 (Nov. 1980) pp. 115-118.

6. Lawler, L., and Martel, C.U., "Scheduling Periodically Occuring Tasks On Multiple Processors," Inform. Process. Lett., Vol. 12, no. 1 (Feb. 1981) pp. 9-12

7. Liu, C.L. and Layland, J.W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," J. ACM 20, 1 (jan. 1973), pp. 46-61.

8. Serlin, Omri, "Scheduling of Time Crirical Processes," Proc. of the Spring Joint Computer Conference, (1972), pp. 925-932.

9. Dhall, S.K., "Scheduling Periodic Time-Critical Jobs on Single Processor and Multiprocessor Computing Systems," University of Illinois Technical Report no. UIUCDCS-R-77-859 (April 1977).

10. Dhall, Sudarshan K., and Liu, C.L., "On a Real-Time Scheduling Problem," Operations Research, vol. 26, no.1, 1978.

11. Garey, M.R., and Johnson, D.S., "Computers and Intractability: A Guide to the Theory of NP-completeness," W.H. Freeman & Co., San Francisco, Calif., 1979.

12. Karp, R.M., "Reducibility among Combinatorial Problems," in Complexity of Computer Computations, R.E. Miller and J.M. Thatcher, Eds. Plenum Press, New York, 1972, pp. 85-103.

13. Johnson, D.S., "Near Optimal Bin Packing Algorithms," Ph.D. dissertation, MIT, Cambridge, Mass., June 1973.

14. Johnson, D.S., "Fast Algorithms for Bin-packing," J. Comput. System Sci., vol. 8, pp. 272-314, 1974.

15. Johnson, D.S., Demers, A., Ullman, J.D., Garey, M.R. and Graham, R.L., "Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms," SIAM Jornal on Computing, vol. 3, pp. 299-325, 1974.

16. Yao, A., "New Algorithms for Bin-Packing," J. ACM 27 (1980), pp. 207-227.

17. Lee C.C., Lee D.T., "A Simple On-Line Bin-Packing Algorithm," J. ACM, Vol. 32, No. 3 (July 1985), pp. 562-572.

18. Aho, A.V., Hopcroft, J.E., and Ullman, J.D., "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, Mass., 1974.

19. Graham, M.R., "Bounds for Certain Multiprocessing Anomalies," Bell. Sys. Tech. Jour. 45, no. 9(1966) 1563-1581.

20. Graham, M.R., "Bounds on Multiprocessing Timing Anomalies," SIAM JOUR. of APP. Math. 17, no. 2(1969) 416-429.

21. Knuth, D.E., "The Art of Computer Programming," Vol. 3 Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

22. Davari, S., and Dhall, S.K., "On a Real-Time Task Allocation Problem," 19 Annual Hawaii International Confrence On Syste Sciences, Jan. 8-10 1986, Honolulu, Hawaii.

APPENDIX A

Theorem A.1: Let x be a positive integer. Then,

$$\lim_{x \to \infty} x(2^{1/x} - 1) = \mathrm{Ln}2$$

Proof:

$$\lim_{x \to \infty} x(2^{1/x}-1) = \lim_{x \to \infty} \frac{2^{1/x}-1}{1/x}$$

$$= \lim_{k \to 0} \frac{2^{k}-1}{k}$$

$$= \lim_{k \to 0} \frac{2^{k}\mathrm{Ln}2}{1}$$

$$= \mathrm{Ln}2$$

Q.E.D.

Corollary A.1: Let x be a positive integer. Then,

$$\lim_{x \to \infty} (x-1)(2^{1/x}-1) = \mathrm{Ln}2.$$

Theorem A.2: Let x be a positive integer. Then, the value of function $f(x) = x(2^{1/x}-1)$ is monotonically decreasing with x.

Proof: We will prove this theorem in two different ways:

(I) Show that $f(x) - f(x+1) > 0$:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

$$2^{1/x} = e^{(Ln2)/x}$$

$$= 1 + \frac{Ln2}{x} + \frac{(Ln2)^2}{2!x^2} + \frac{(Ln2)^3}{3!x^3} + \ldots$$

$$f(x) = x(2^{1/x}-1) = Ln2 + \frac{(Ln2)^2}{2!x} + \frac{(Ln2)^3}{3!x^2} + \ldots$$

$$f(x+1) = Ln2 + \frac{(Ln2)^2}{2!(x+1)} + \frac{(Ln2)^3}{3!(x+1)^2} + \ldots$$

$$f(x) - f(x+1) = \frac{(Ln2)^2}{2!}\left(\frac{1}{x} - \frac{1}{x+1}\right) +$$

$$\frac{(Ln2)^3}{3!} - \left(\frac{1}{x^2} - \frac{1}{(x+1)^2}\right) + \ldots$$

$$> 0$$

(II) Show that $f'(x) < 0$:

$$f'(x) = 2^{1/x}\left(1 - \frac{Ln2}{x}\right) - 1$$

Since $e^{-x} > 1 - x$,

$$2^{-1/x} = e^{-(Ln2)/x} > (1 - \frac{Ln2}{x})$$

$$1 > 2^{1/x}(1 - \frac{Ln2}{x})$$

Therefore, $f'(x) < 0$

Q.E.D.

Theorem A.3: Let x be a positive integer. Then, the value of function $g(x) = (x-1)(2^{1/x}-1)$ is monotonically increasing with x.

Proof: We will prove this theorem in two ways:

(I) Show that $g(x+1) - g(x) > 0$:

$g(x) = (x-1)(2^{1/x}-1) = x(2^{1/x}-1) - (2^{1/x}-1)$

$g(x+1) = x(2^{1/(x+1)}-1) = (x+1)(2^{1/(x+1)}-1) - (2^{1/(x+1)}-1)$

From the proof of theorem A.2, we have:

$$(2^{1/x}-1) = \frac{Ln2}{x} + \frac{(Ln2)^2}{2!x^2} + \frac{(Ln2)^3}{3!x^3} + \ldots$$

$$(2^{1/(x+1)}-1) = \frac{Ln2}{x+1} + \frac{(Ln2)^2}{2!(x+1)^2} + \frac{(Ln2)^3}{3!(x+1)^3} + \ldots$$

$$x(2^{1/x}-1) = Ln2 + \frac{(Ln2)^2}{2!x} + \frac{(Ln2)^3}{3!x^2} + \ldots$$

$$(x+1)(2^{1/(x+1)}-1) = \mathrm{Ln}2 + \frac{(\mathrm{Ln}2)^2}{2!(x+1)} + \frac{(\mathrm{Ln}2)^3}{3!(x+1)^2} + \ldots$$

$$g(x+1) - g(x) = (x+1)(2^{1/(x+1)}-1) - x(2^{1/x}-1) + (2^{1/x}-1) - (2^{1/(x+1)}-1)$$

$$= \frac{(\mathrm{Ln}2)^2}{2!(x+1)} + \frac{(\mathrm{Ln}2)^3}{3!(x+1)^2} + \ldots$$

$$- \frac{(\mathrm{Ln}2)^2}{2!x} - \frac{(\mathrm{Ln}2)^3}{3!x^2} - \ldots$$

$$+ \frac{\mathrm{Ln}2}{x} + \frac{(\mathrm{Ln}2)^2}{2!x^2} + \frac{(\mathrm{Ln}2)^3}{3!x^3} + \ldots$$

$$- \frac{\mathrm{Ln}2}{x+1} - \frac{(\mathrm{Ln}2)^2}{2!(x+1)^2} - \frac{(\mathrm{Ln}2)^3}{3!(x+1)^3} - \ldots$$

$$= \frac{(\mathrm{Ln}2)^2}{2!}\left(\frac{1}{x+1} - \frac{1}{x}\right) + \frac{(\mathrm{Ln}2)^3}{3!}\left(\frac{1}{(x+1)^2} - \frac{1}{x^2}\right) +$$

$$+ \mathrm{Ln}2\left(\frac{1}{x} - \frac{1}{x+1}\right) + \frac{(\mathrm{Ln}2)^2}{2!}\left(\frac{1}{x^2} - \frac{1}{(x+1)^2}\right) + \ldots$$

$$= \mathrm{Ln}2\left(\frac{1}{x} - \frac{1}{x+1}\right)\left(1 - \frac{\mathrm{Ln}2}{2!}\right) + \ldots$$

$$+ \frac{(\mathrm{Ln}2)^2}{2!}\left(\frac{1}{x^2} - \frac{1}{(x+1)^2}\right)\left(1 - \frac{\mathrm{Ln}2}{3}\right) + \ldots$$

$$> 0$$

(II) Show that $g'(x) > 0$:

$$g(x) = (x-1)(2^{1/x}-1) = x(2^{1/x}-1) - 2^{1/x} + 1$$

$$g'(x) = 2^{1/x}\left(1 - \frac{Ln2}{x} + \frac{Ln2}{x^2}\right) - 1$$

Since $e^{-x} < 1 - x + \frac{x^2}{2}$,

$$2^{-1/x} = e^{-(Ln2)/x} < 1 - \frac{Ln2}{x} + \frac{(Ln2)^2}{x^2} < 1 - \frac{Ln2}{x} + \frac{Ln2}{x^2}$$

$$1 < 2^{1/x}\left(1 - \frac{Ln2}{x} + \frac{Ln2}{x^2}\right)$$

Therefore, $g'(x) > 0$.

Q.E.D.