JAISTRepository
https://dspace.jaist.ac.jp/

Title	A Study on Practical Real-Time Task Scheduling
Author(s)	尹，傲彤
Citation	
Issue Date	2012-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/10766
Rights	
Description	Supervisor: 田中 清史，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

A Study on Practical Real-Time Task Scheduling

By Aotong Yin

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Kiyofumi Tanaka

September, 2012

A Study on Practical Real-Time Task Scheduling

By Aotong Yin (1010229)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Kiyofumi Tanaka

and approved by
Associate Professor Kiyofumi Tanaka
Professor Mineo Kaneko
Associate Professor Yasushi Inoguchi

August, 2012 (Submitted)

Copyright © 2012 by Aotong Yin

Abstract

In this research, four typical task scheduling algorithms, Rate Monotonic, Earliest Deadline First, Priority Exchange Server, and Total Bandwidth Server, are implemented on ITRON System, which is the widely used real-time embedded operating system. Their performances are tested by different task sets, and the results show they are coincident with their theories. Furthermore, two practical improvements separately on PES and TBS are raised according to the experiment results. By considering the overheads in practical systems, the improvements are obvious on responsive performance with smaller deadline missing number.

Contents

1 Introduction **1**

2 Related Algorithms **4**

3 Implementation **15**
3.1 Basic Implementation 15
3.2 Practical Realization 25

4 Evaluation **29**
4.1 Environment 29
4.2 Basic Results 34
4.3 Practical Results 43

5 Conclusion **46**

6 References **47**

7 Acknowledgement **48**

List of Figures

1.1 ITRON is embedded real-time operating system. 3
1.2 ITRON Kernel Structure. 3

2.1 Two periodic tasks. 6
2.2 Using Rate Monotonic to schedule two periodic tasks. 6
2.3 Three periodic tasks. 8
2.4 Using Rate Monotonic to schedule three periodic tasks. 8
2.5 Two periodic tasks. 9
2.6 Using Earliest Deadline First to schedule three periodic tasks. 9
2.7 Two aperiodic tasks. 10
2.8 Two periodic tasks. 10
2.9 Using Earliest Deadline First to schedule both aperiodic and periodic tasks at the same time. 10
2.10 Server's capacity and period. 12
2.11 Two periodic tasks. 12
2.12 Two aperiodic tasks. 12
2.13 Using Priority Exchange Server to schedule both aperiodic and periodic tasks with server's capacity. 12
2.14 Two aperiodic tasks. 14
2.15 Two periodic tasks. 14
2.16 Using TBS to schedule both aperiodic and periodic tasks. 14

3.1 In actual systems, the execution time can be off a multiple of a tick. 16
3.2 A brief structure of TCB. 17
3.3 A brief structure of kernel queue. 18
3.4 Important attributes of each task. 19
3.5 Flow of inserting tasks in ready queue using RM. 20
3.6 Flow of brief scheduling process of PES. 22
3.7 Flow for treating with aperiodic request using PES. 23
3.8 Flow for treating with aperiodic request if there is no server's capacity using PES. 24
3.9 Flow for treating with non-aperiodic request using PES. 26
3.10 Flow of calculating procedure for virtual deadlines of aperiodic tasks by using TBS. 27

4.1 10 periodic tasks for the 1st experiment task set. 30
4.2 10 periodic tasks for the 2nd experiment task set. 31
4.3 6 aperiodic tasks for the 3rd experiment task set. 32
4.4 10 periodic tasks for the 3rd experiment task set. 32
4.5 6 aperiodic tasks for the 4th experiment task set. 33
4.6 10 periodic tasks for the 4th experiment task set. 33
4.7 6 aperiodic tasks for the 5th experiment task set. 34
4.8 10 periodic tasks for the 5th experiment task set. 34
4.9 Result of RM on the 1st task set. 35
4.10 Result of EDF on the 1st task set. 35
4.11 Result of PES on the 1st task set. 36
4.12 Result of TBS on the 1st task set. 36
4.13 Result of RM on the 2nd task set. 37
4.14 Result of EDF on the 2nd task set. 37
4.15 Result of PES on the 2nd task set. 38
4.16 Result of TBS on the 2nd task set. 38
4.17 Result of EDF on the 3rd task set. 39
4.18 Result of TBS on the 3rd task set. 40
4.19 Result of EDF on the 4th task set. 41
4.20 Result of theoretical TBS on the 4th experiment task set. 42
4.21 Result of theoretical PES on the 5th experiment task set. 43
4.22 Result of practical TBS on the 3th experiment task set. 44
4.23 Result of practical PES on the 5th experiment task set. 45

Chapter 1

Introduction

In modern times, real-time systems, which focus on computing correctness both on the value and the responding time, are in a critical position in our society. In this case, task scheduling is extremely important for real-time systems. So in order to adapt and improve the real-time systems, various scheduling algorithms have been devised. However, these algorithms were either too theoretical to be used in actual system, or too heavy computing. In this research, our aim is to study the scheduling algorithms on real-time operating systems, and try to reach a solution to improve the actual used real-time operating system, ITRON.

Real-time systems are the computational systems aiming to react to events in the environment within the accurate time constraint. Time means the systems need to compute correctly relying not only on the logical results, but also at the time when the results are generated. And the word real indicates that the reaction of the system to external events is meaningful only during their processing time. That means a reaction that occurs too late could be useless or even dangerous.

The subject of real-time computing is to complete a task correctly and in time. Since an incredible increasing number of todays complex systems depend on computer control in part or completely, real-time computing occupies a key important position in our society. The rule of task scheduling of real-time systems includes timeliness, design for peak load, predictability, fault tolerance, and maintainability. To achieve this aim, various kinds of tasks have been adopted, such as hard real-time, soft real-time, aperiodic and periodic tasks in fact. So to judge whether a real-time system is good or not, we should focus on two main aspects: whether the task requests have good response, and whether the computation spends few system overheads.

In order to realize the crucial real-time systems, a great number of scheduling theories and relative algorithms have been devised. Even the existing typical task scheduling theories and their algorithms are well studied and have reasonable solutions for real-time task scheduling, they are not practical enough for the real used real-time systems. Their problems are shown in the following:

First, they are too theoretical to adjust to the actual systems; second, they require heavy computation with many overheads.

Although the existing typical scheduling algorithms are very classical to deal with certain kinds of tasks, they are not realistic because each strategy aims to solve its single type of tasks, while in the real computing environment, there should be different types of tasks running at the same time and in the same environment, like both periodic tasks and aperiodic tasks, hard real-time tasks and soft real-time tasks, etc. Meanwhile, other constraints are not taken into consideration, like time constraint, dependency, resource constraints, synchronization, etc.

For the second aspect, to solve the problems above, some other algorithms or solutions which are much more considerable, are raised. However, these solutions are still not suitable for actual real-time systems because when adopted in the practice machines, the algorithms turn to be so complicated that the systems run too slow. The main reason is that, in order to be as considerable as possible, the tasks sometime have to switch with each other very frequently during their execution time. The frequent task switching would cause a high number of overheads, which in the actual environment, is the main factor to let the computation inefficient and heavyweight.

So in this research, the aim is to try to devise a real-time task scheduling algorithm that is in consideration of all kinds of tasks, concerning practical constraints, and being achieved by lightweight computation. In order to achieve this goal, we need to choose a real-time operating system, as practical as possible, implement some typical existing scheduling algorithms on it, and evaluate performance on that. And according to the evaluation, on the foundation of these scheduling algorithms, the author obtains a practical and lightweight computing algorithm, also implement it on the experiment environment, evaluate its performance, then reach a compare among all of the algorithms.

In this research, our environment is called ITRON System[1]. ITRON is an abbreviation for Industrial TRON. It is a substructure of TRON, widely used all around the world, especially in Japan, and appearing on millions of electronic equipments, used for about 30 percents of the embedded systems (Figure 1.1), but only fixed priority scheduling policy is used in this system. The author chooses ITRON System as our research environment because of the following two reasons. On one hand, as our study is focusing on the actual real-time operating system, the author chooses the one in real use. Fortunately, ITRON System is not only right in actual use, but also well used by different fields and places, so these makes ITRON System a very suitable environment for studying real-time operating system. On the other hand, for our research is mainly about scheduling algorithm, the limitation of scheduling policy on ITRON System is right what the author can try and replace by a new strategy or algorithm (Figure 1.2). So the above two reasons let ITRON System be the best environment for our research.

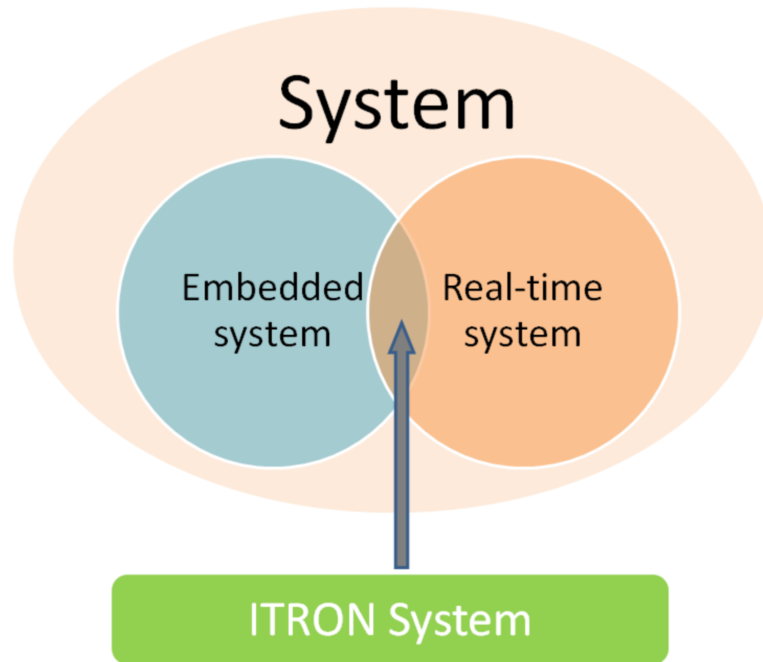

Figure 1.1: ITRON is embedded real-time operating system.

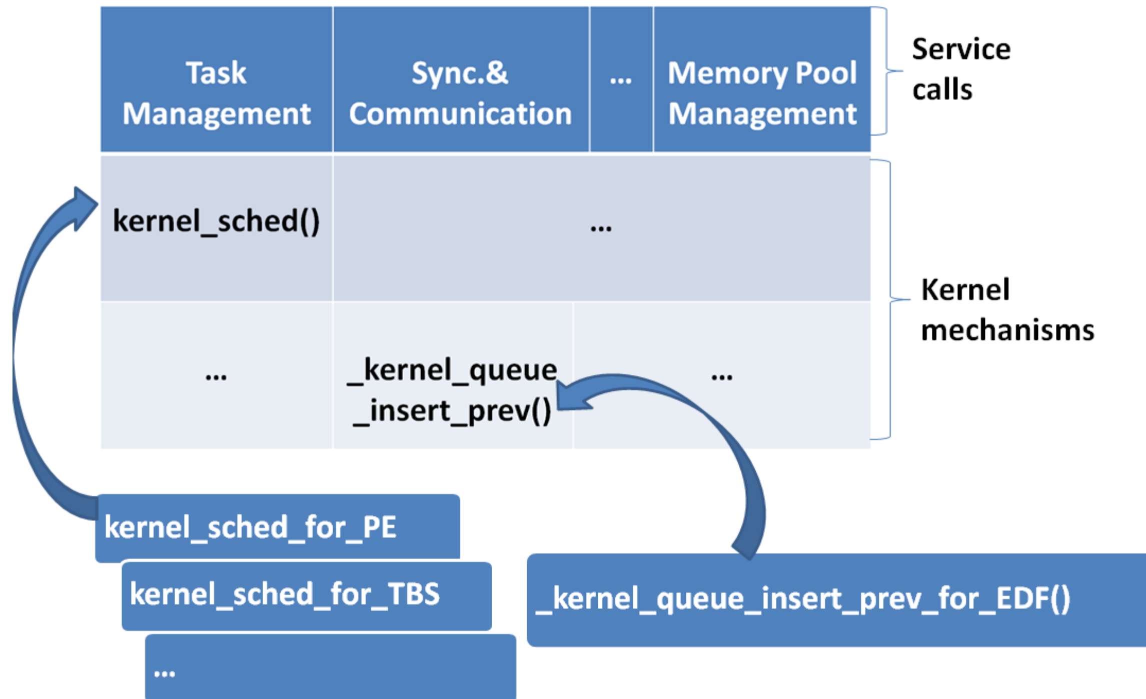

Figure 1.2: ITRON Kernel Structure.

Chapter 2

Related Algorithms

Some important conceptions for real-time system should be explained at the beginning.

Periodic task, is requested at every regular time period; while aperiodic task, to the opposite,is requested irregularly .

Hard real-time task means that if it misses its deadline, it will cause a serious consequence on a system, or even cause the whole system damaged. So usually, a hard real-time task should be guaranteed as a highest priority on execution. Soft real-time task is not as serious as hard real-time task. If the deadline is missed, it won't cause serious or dangerous problem, only sometimes causing bad response or showing bad performance. So in real-time task scheduling, missing hard real-time task deadline should be strictly avoided. Soft ones should be refrained as many as possible to achieve the perfect performance of real-time system.

Arrival time is the time that a task is getting ready to be executed. It can also be called as request time. While in this research, arrival time is more frequently used.

Computation time is time during which the task runs without any interruption.

Finishing time is the time the task finishes its execution.

Response time is the time between the task's request and its finishing time. This factor is very important for real-time system. It is the main value to evaluate the performance of real-time system.

Absolute Deadline is a time point before which a task should finish its execution in running environment. If a hard real-time task misses its absolute deadline, the system will face the risk of damage; if a soft real-time task misses its absolute deadline, the response will be bad, and the performance will be decreased.

Relative Deadline is the time between the task's request time and the absolute deadline.

Static is that the sequence of the task execution can be known by some constant arguments, and can be sure before the tasks start to run.

Dynamic is that the order of task execution can only be known at its run time. It depends on dynamic arguments, and might be changed by different system situation.

Processor utilization factor U is the accumulation of each periodic task's computation time divided by its own period. The value should be less than 1, so that the system is schedulable.

To realize real-time task scheduling, many typical and classical algorithms were devised. For example, Rate Monotonic[2] is specially for periodic task scheduling, focusing on periods of tasks; Earliest Deadline First[2] concerns absolute deadline of all tasks, considering both periodic tasks and aperiodic tasks; Priority Exchanged Server[3], which is one of the most complicated algorithms, leads a conception of capacity into computation to gain a good schedule bound of the periodic task set; Total Bandwidth Server[4], considerable but much simpler, does not introduce too heavy computation. The above four typical algorithms are what the author used in this research. The author implemented them on ITRON System by replacing the old strategy, and also evaluated their performance. So these four algorithms' policies are shown on the following.

Rate Monotonic is the simplest one among these four algorithms. It is specially dealing with periodic tasks. The main conception is that the periodic task which has the higher frequency request will have a higher priority in the run time. Higher priority means the task will be executed earlier than other tasks. That means, the scheduling is according to the period of tasks.If a task has the most rapid request (the shortest period), it will has a highest priority, then if the task has second most rapid request, it will be chosen as the second priority to be executed, then as the same sequence, until the task with the longest period will have the lowest priority.

Since the periodic tasks' period is a constant value, and only period is concerned in this algorithm, the priority is fixedly assigned: before the execution of all the tasks, the priority can be known in advance only by obtaining all the periods of these tasks. So this algorithm is static, not dynamic.

To explain Rate Monotonic algorithm in detail, here follows two examples.

The first example is a simple one, because there are only two periodic tasks $\tau 1$ and $\tau 2$ (Figure 2.2). In the figure, the horizontal axis indicates the time line. The vertical lines indicate the starts of every period of the two periodic tasks. The squares are the execution times of these tasks. And the dash lines indicate the switches of different task-executions.

From t=0, since $\tau 1$'s relative deadline is 3, shorter than $\tau 2$'s relative deadline 4, $\tau 1$

	C_i	T_i
τ_1	1	3
τ_2	2	4

Figure 2.1: Two periodic tasks.

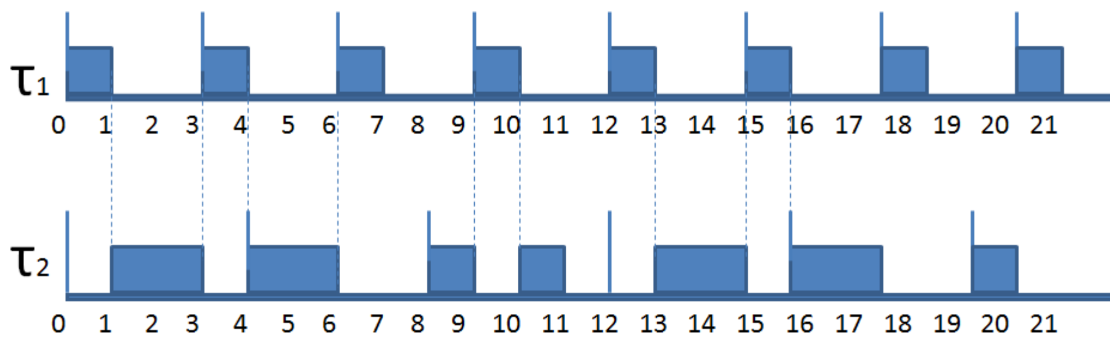

Figure 2.2: Using Rate Monotonic to schedule two periodic tasks.

will always execute first according to Rate Monotonic algorithm. So at t=9, when $\tau 2$ is executing in the way, since $\tau 1$ has a shorter period, it preempts $\tau 2$, running first. The same way, at t=12, although $\tau 2$ is activated, $\tau 1$ is chosen first.

In the second example, there are 3 periodic tasks (Figure 2.4). $\tau 3$ has the longest relative deadline so always has the lowest priority to execute. At t=0, $\tau 1$ executes, and after that, $\tau 2$ starts the execution. At t=3, $\tau 1$ starts again, then followed by $\tau 2$. At t=7, $\tau 1$ and $\tau 2$ are idle, and $\tau 3$ finally starts to execute, while t=8 is the deadline. Since the computation time should be 2 in one period, so $\tau 3$ missed the deadline in its first period. And from t=8, $\tau 1$ and $\tau 2$ still executes with higher priority than $\tau 3$ until t=11. Then $\tau 3$ starts to execute but is preempted at t=12 by $\tau 1$. Until t=16, $\tau 3$ hasn't executed again, so that it meets its deadline missing again. From the second example, we can find that, although Rate Monotonic is quick and simple enough to schedule the periodic tasks, it might be easy to cause deadline missing for low priority tasks which have longer periods.

Earliest Deadline First is a little more complicated than Rate Monotonic. It can deal with both periodic task and aperiodic task. It selects task to execute according to absolute deadline. The task which has earlier absolute deadline will execute at higher priority.

Here are two examples for Earliest Deadline First. One only deals with two periodic tasks, the other one treats both periodic tasks and aperiodic tasks.

For the first example (Figure 2.6),at first, $\tau 1$ has the earliest deadline 3, so it is the first to execute. After this execution, its absolute deadline alters to 6, so at t=3, $\tau 2$ that has an earlier absolute deadline 5 should be executed first. At t=6, this moment, $\tau 1$ has the deadline 9, earlier than $\tau 2$'s 10, so $\tau 1$ executes first, then here comes $\tau 2$, continuing to execute at t=7.

For the second example (Figure 2.9),the upwards arrows indicate the activation of each aperiodic task, and the downwards arrows indicate the absolute deadline of each.At first, according to the absolute deadlines, $\tau 1$ executes first and then $\tau 2$ follows. At t=3, aperiodic task ap1 is activated, but at this time, $\tau 1$'s absolute deadline is 6, earlier than ap1's 8. So at t=3, $\tau 1$ executes first, then t=4, ap1 executes. At t=6, since $\tau 1$'s deadline 9 is still earlier than $\tau 2$'s 10, $\tau 1$ will preempt to execute. At t=11, aperiodic task ap2 is activated, and since the absolute deadline 14 is also earlier than $\tau 2$, it will be executed first.

Priority Exchange Server is the most complicated algorithm among these four typical algorithms. It introduces a periodic server to service the aperiodic tasks, and the server capacity concept is also imported into the algorithm.

Different from other algorithms, it assumes the aperiodic tasks as soft real-time, while periodic tasks as hard real-time. That means, this algorithms is aiming to optimize the periodic task performance. Compared with other algorithms, Priority Exchange Server

	C_i	T_i
τ_1	1	3
τ_2	2	4
τ_3	2	8

Figure 2.3: Three periodic tasks.

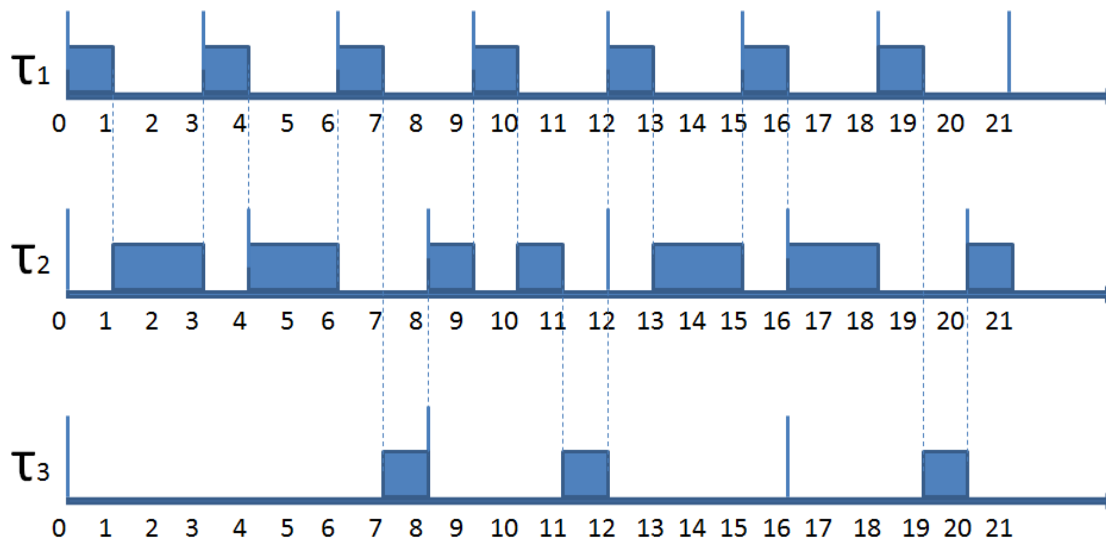

Figure 2.4: Using Rate Monotonic to schedule three periodic tasks.

	C_i	T_i
τ_1	1	3
τ_2	3	5

Figure 2.5: Two periodic tasks.

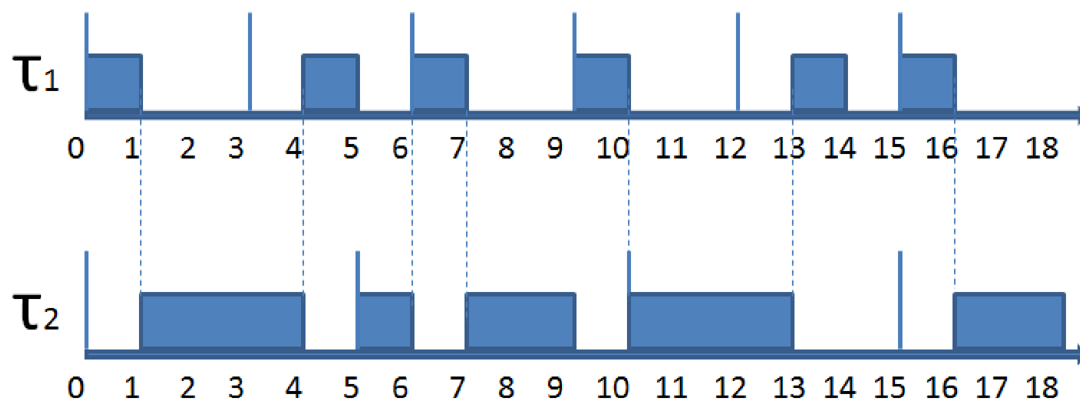

Figure 2.6: Using Earliest Deadline First to schedule three periodic tasks.

	A_i	C_i	A_dl_i
ap_1	3	1	8
ap_2	11	2	14

Figure 2.7: Two aperiodic tasks.

	C_i	T_i
τ_1	1	3
τ_2	2	5

Figure 2.8: Two periodic tasks.

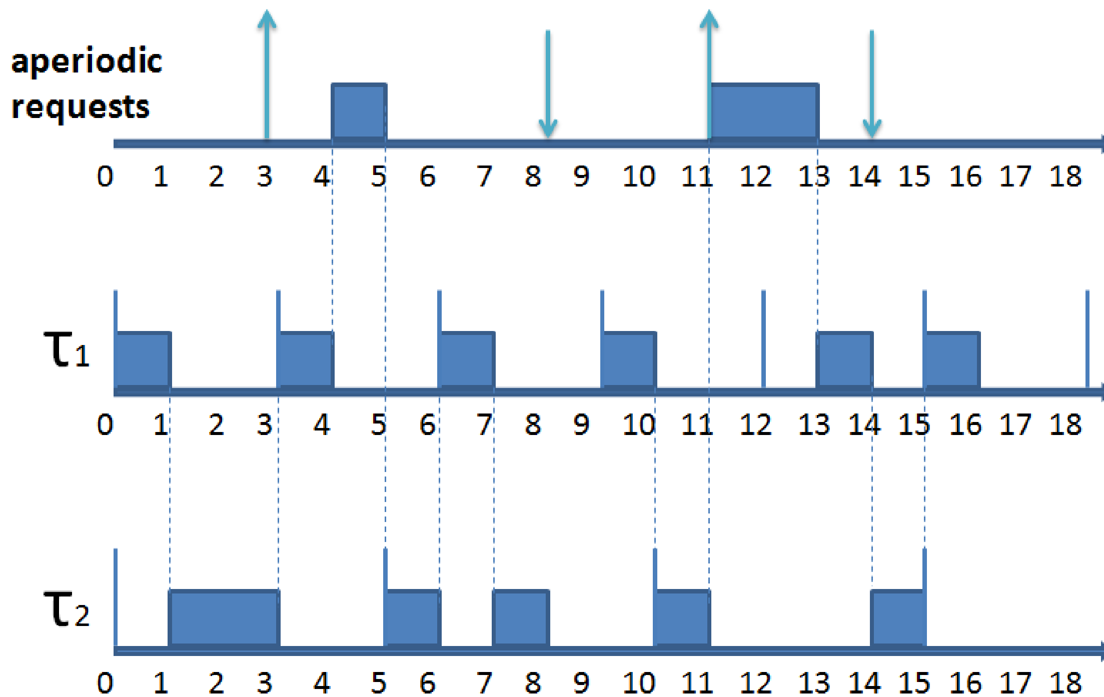

Figure 2.9: Using Earliest Deadline First to schedule both aperiodic and periodic tasks at the same time.

shows a little bit worse performance of aperiodic responsiveness but provides much better schedulability bound of periodic task set.

Although the priority sequence is fixed before the execution of all the tasks, the algorithm is still dynamic because when these tasks start executing, the absolute deadline can be sure, and then all the tasks' priority can be assigned during the execution time.

Priority Exchange Server depends on the idea of Rate Monotonic, and a periodic server is introduced to service the aperiodic tasks. The server capacity is introduced into this rule. Since the server's period is supposed shorter than any periods of periodic tasks, according to Rate Monotonic, server will have higher priority than any periodic tasks. Then aperiodic tasks are assigned the server's priority, so they have the highest priority. As each server's period starts, the server replenishes a full capacity.If no aperiodic task request exists but the server's capacity is being replenished, periodic tasks from the highest priority to the lowest will be scanned and the highest periodic task will be executed and preserve the capacity for the server, then the capacity will have the same priority as this periodic task. If there is an aperiodic task request, and there is still enough capacity, the aperiodic task is executed and consumes the capacity. If there is not enough capacity on server's priority, lower level's capacity will be looked for, and from the highest priority if there is any enough capacity on this priority, the aperiodic task is executed and consumes this priority's capacity. But if there is no capacity, and this priority's periodic task is activated and ready to run, this priority's periodic task will be executed, and the aperiodic task needs to wait.

Here is the example of Priority Exchange Server (Figure 2.13). Here, the server's period is 5, and the capacity for each period is 1. So at the beginning of each period, the server will be filled with the full capacity. At t=0, there is no aperiodic task, so $\tau 1$ is executed, so that the server's capacity is preserved at $\tau 1$'s priority. At t=5, aperiodic task ap1 whose computation time is 1 is activated, and the server is replenished by the full capacity at the beginning of the second period. This time, aperiodic task is executed using this capacity, and completes the execution. After that, when t=6, $\tau 2$ starts the execution, so it also starts to preserve the capacity instead of $\tau 1$. While at t=10, since PE Server is based on RM algorithm, $\tau 1$, which has the shorter period, has the higher priority to execute, so $\tau 1$ begins to execute at this moment. At the same time, server's capacity is replenished again, so $\tau 1$ preserves the capacity at its own priority. At t=11, the aperiodic task ap2 executes, consuming the capacity at $\tau 1$'s priority. But at t=12, although the highest priority task ap2 hasn't finished executing, and there is capacity at $\tau 2$'s priority, $\tau 1$ continues to execute. And t=15, the server's capacity is replenished again, and ap2 starts to execute again, using the server's capacity. From t=16, $\tau 1$ continues to execute, and at t=20, $\tau 1$ executes and preserves the capacity at its priority.

Total Bandwidth Server handles both periodic task and aperiodic tasks. It uses Earliest Deadline First algorithm as the basic and makes an improvement of it. The concept is to

Server
$C_i = 1$
$T_i = 5$

Figure 2.10: Server's capacity and period.

	C_i	T_i
τ_1	5	10
τ_2	7	20

Figure 2.11: Two periodic tasks.

	C_i	A_i
ap_1	1	5
ap_2	2	11

Figure 2.12: Two aperiodic tasks.

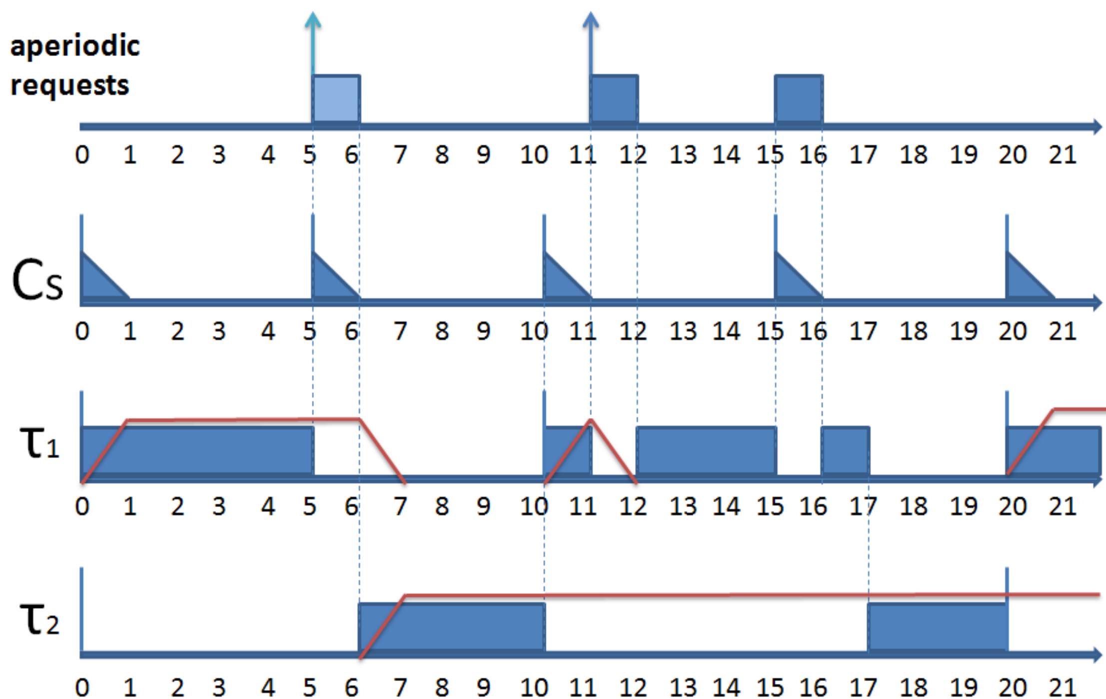

Figure 2.13: Using Priority Exchange Server to schedule both aperiodic and periodic tasks with server's capacity.

assign an virtual absolute deadline to aperiodic task, and schedule with EDF algorithm referring to aperiodic task's virtual absolute deadline and periodic task's real absolute deadline. A task that has the earlier absolute deadline will have the higher priority and will be executed first.

TBS is considerable for both of periodic and aperiodic tasks, but not as complex as PE Server. The virtual absolute deadline is the larger one between aperiodic task's request time and the virtual last aperiodic task's absolute deadline, plus the aperiodic task's computation time divided by the server's utilization.

The sequence relies on the absolute deadline, so it depends on the aperiodic tasks request, their computation time and server's utilization. So this algorithm is also dynamic. And the assigned absolute deadline to aperiodic tasks makes them having a better responsiveness performance than some other algorithms, and the simple scheduling rule realizes lightweight computation.

Suppose there are two periodic tasks $\tau 1$, $\tau 2$ and two aperiodic tasks ap1 and ap2 (Figure 2.16). At time t=0, according to EDF, $\tau 1$ has the earlier deadline 6, so $\tau 1$ will be executed first, then at t=2, $\tau 2$ executes. Ap1 is activated at t=6, which is also the beginning of $\tau 1$'s next period. This means $\tau 1$ is also activated. Referring $\tau 1$ and $\tau 2$, the utilization of periodic tasks is 0.67, so we suppose the server's utilization is 0.33(= 1 - 0.67). Then comes ap1's virtual absolute deadline, 9, which is earlier than $\tau 1$'s absolute deadline 12. So here ap1 precedes $\tau 1$ by using the virtual deadline. At t=11, ap2 requests, the virtual absolute deadline is 17, so it preempts $\tau 2$, whose absolute deadline is 18, and executes first.

	A_i	C_i
ap_1	6	1
ap_2	11	2

Figure 2.14: Two aperiodic tasks.

	C_i	T_i
τ_1	2	6
τ_2	3	9

Figure 2.15: Two periodic tasks.

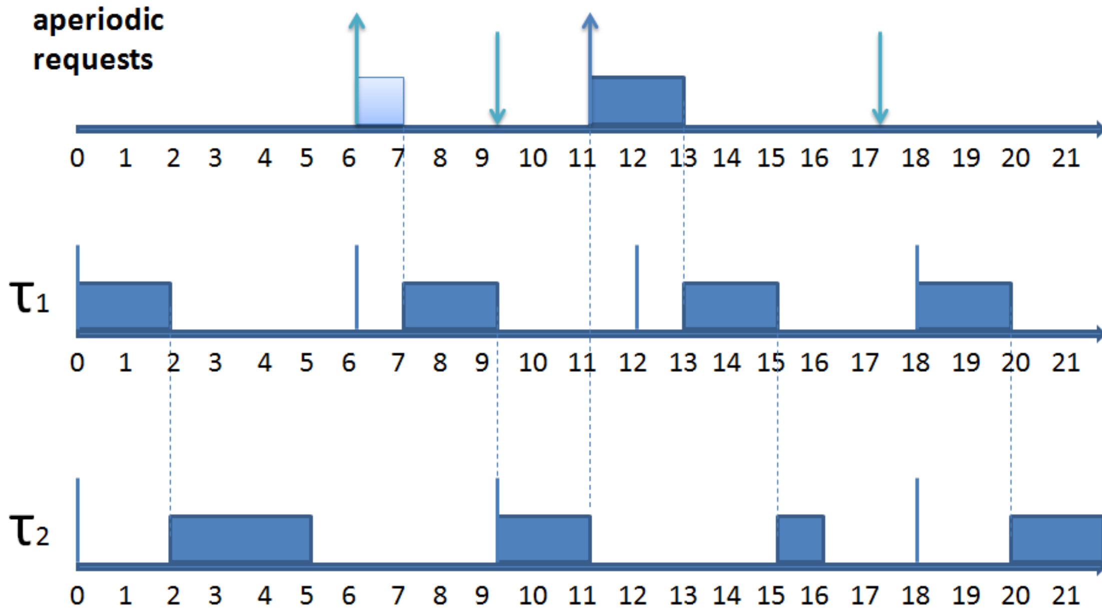

Figure 2.16: Using TBS to schedule both aperiodic and periodic tasks.

Chapter 3

Implementation

3.1 Basic Implementation

Here are two significant problems when these four algorithms are implemented into real use, especially being realized on ITRON System. The assumptions in the original algorithms are too simple and theoretical for actual real-time systems.

For the first one, in the description of the four algorithms in the previous chapter, the tasks' execution time is usually assumed as a multiple of system's tick which is the smallest measured time unit in the system. It is too theoretical that real task's execution would be exactly a multiple of a tick. In actual situations, the execution time can be off a multiple of a tick, and even be shorter than a tick. In addition, in actual environment, tasks can be activated at any timing (not at a multiple of a tick). In the following figure (Figure 3.1), the top line labeled "tick" indicates the measure of ticks, and J1 and J2 are two periodic tasks. We can find that, the two tasks' executions are not a multiple of a tick, and they are activated off the tick timing. Therefore, overheads of task activation and task switching happen not only at a tick timing but at any timing. To achieve practical real-time task scheduling, these overheads must be considered.

The second problem is that, in theoretical explanation of scheduling algorithms, no matter how complex one algorithm can be, the complexity is ignored. The explanation focuses only on the scheduling rule itself. On the other hand, in actual systems, overheads will appear when scheduling is performed and an execution is switched from one task to another. Overheads are the important factor for real-time system since they affect the performance of responsiveness. So in the implementation process, the author needs to concern which scheduling strategy is considerable and at the same time which strategy introduces fewer overheads.

Before introducing the process of the implementation, some important variables and structures will be explained at first.

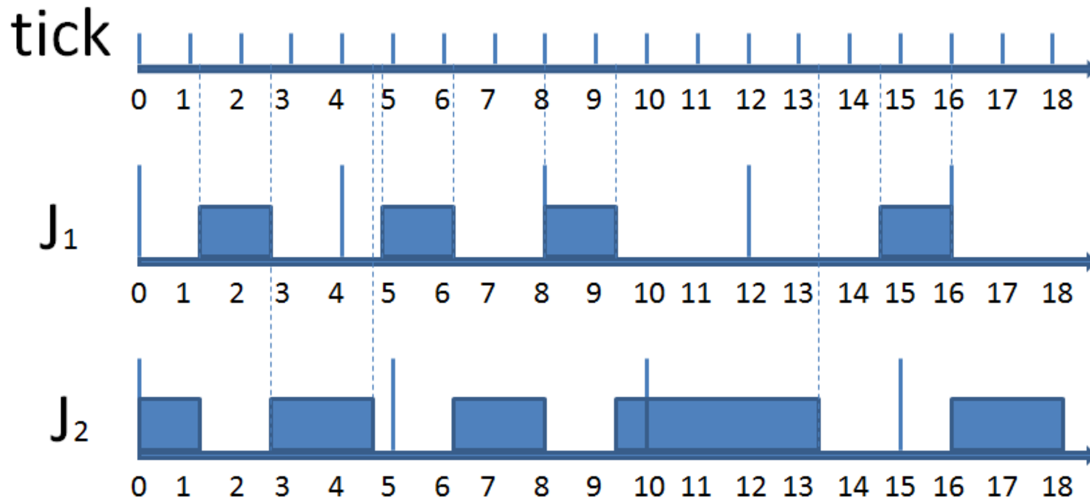

Figure 3.1: In actual systems, the execution time can be off a multiple of a tick.

The first important structure in the operating system is called TCB (Task Control Block) (Figure 3.2). The operating system manages a TCB for each task. It has member variables including a priority information. TCBs are connected with each other by using "prev" and "next" pointers, which can build several queues, such as a ready queue, as a double linked list. "prev" points to the top address of the previous task's TCB. If there is no previous task, it will point to the address of the corresponding entry of the queue structure. The other pointer is called "next". This points to the top address of the next task's TCB. If there is no next task, it will point to the top address of the corresponding entry of the queue. "self" pointer is used to reference other member variables in a TCB when a list is scanned.

Then the significant structure, ready queue, is introduced. It is an array and each element is a struct which consists of pointers (prev, next and self) to make a double linked list. The indexes of the array represent the priorities from 0 to the maximum number. 0 is the highest priority, and as increasing the value, the priority becomes lower. The elements are the heads of each priority and point to the top address of the first task's TCB to be executed at each priority. If at the priority, there is no task to be executed, next will point to the element's own top address. The same way, each element's prev points to the top address of the last task's TCB to be executed at the priority. Also if there is no task to be executed at this priority, next will point to the element's top address. In this array, priority 0 and priority 1 are preserved for the operating system. Since they two have the highest priority, their tasks can preempt any other application tasks. In our implementation, the author preserve priority 2 for the aperiodic tasks in the case of Priority Exchange Server and Total Bandwidth Server algorithms. Periodic tasks use priority 3 and the lower priorities (Figure 3.3).

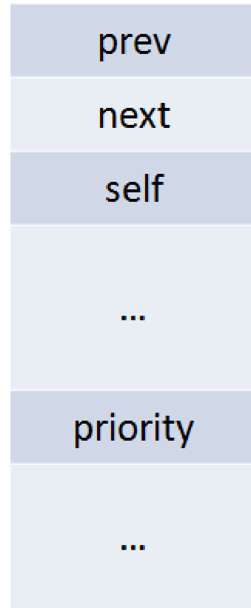

Figure 3.2: A brief structure of TCB.

Periodic tasks and, except for Rate Monotonic, aperiodic tasks are created to evaluate the implemented algorithms. So in this research, the author use appl.c, appl.h, and system.cfg to create periodic tasks and aperiodic tasks. In appl.c file, tasks are defined. In appl.h file, tasks' periods are defined. The system.cfg file includes important information (Figure 3.4). The figure shows several creation functions. By using the creation function, the attributes of tasks to be created are described. As the arguments of the function, two special members are introduced. "member3", defined in TCB's definition, indicates whether a task is periodic task or aperiodic task. If it indicates aperiodic task, the value will be set 0 in the task creating function. If it is periodic, the value will be set as 1. The other member is called member4, also defined in TCB's definition. It records the computation time of the task. These attributes will be used in the following algorithms' implementations.

Rate Monotonic algorithm focuses on tasks' relative deadline which is equal to the period, and there are no aperiodic tasks considered in this algorithm. So in this algorithm's implementation, the author omit the priority factor. Then all the periodic tasks are associated with the same priority 2, which is lower than the system tasks' priorities 0 and 1.

In order to implement Rate Monotonic algorithm, the ready queue needs to be managed. The ready queue decides the sequence of task execution in the priority. Usually, when a task is activated, the task's TCB is inserted to the corresponding ready queue. When scheduling, the scheduler selects a top task of a ready queue as one to be executed next. The selection is performed from the highest ready queue to the lowest until a task

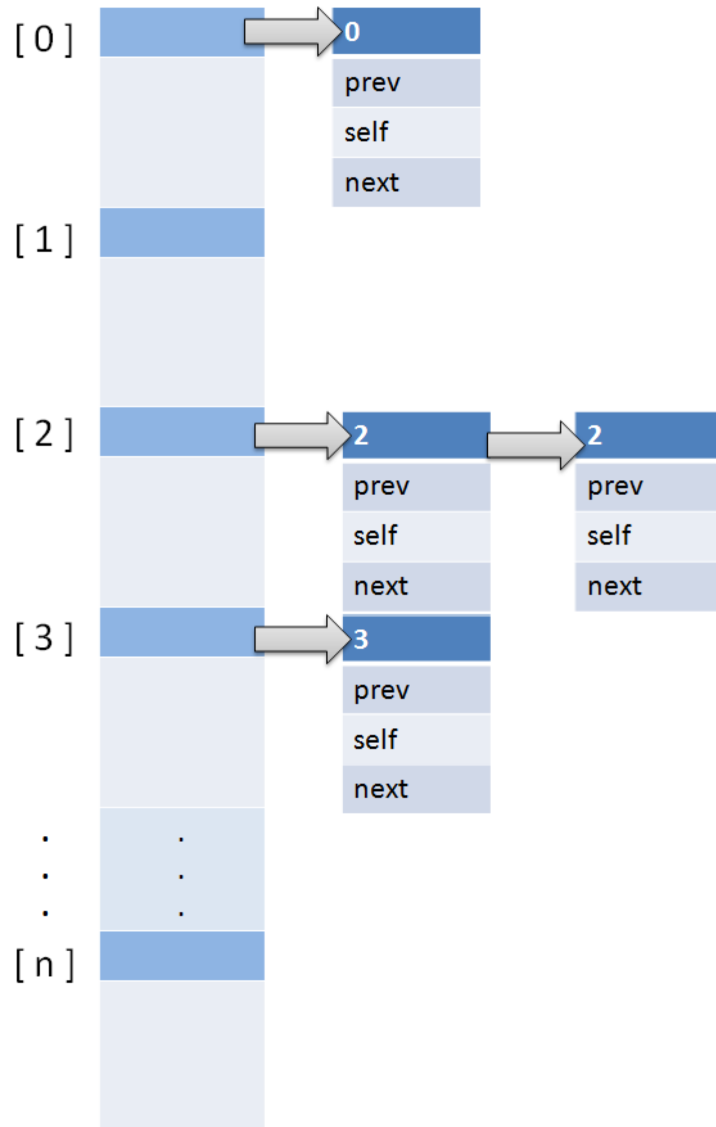

Figure 3.3: A brief structure of kernel queue.

```
CRE_TSK ( TSK_AP, { (TA_HLNG), 255, task_ap, 2, 4096, NULL, 0, 10, 0, 0, 1 } );
CRE_TSK ( TSK_AP2, { (TA_HLNG), 255, task_ap2, 2, 4096, NULL, 0, 10, 0, 0, 2 } );
CRE_TSK ( TSK_AP3, { (TA_HLNG), 255, task_ap3, 2, 4096, NULL, 0, 10, 0, 0, 2 } );
CRE_TSK ( TSK_AP4, { (TA_HLNG), 255, task_ap4, 2, 4096, NULL, 0, 10, 0, 0, 4 } );
CRE_TSK ( TSK_AP5, { (TA_HLNG), 255, task_ap5, 2, 4096, NULL, 0, 10, 0, 0, 6 } );
CRE_TSK ( TSK_AP6, { (TA_HLNG), 255, task_ap6, 2, 4096, NULL, 0, 10, 0, 0, 9 } );
CRE_TSK ( TSK_P3, { (TA_HLNG | TA_ACT), 255, task_p3, 2, 4096, NULL, 0, 15, 0, 1, 1 } );
CRE_TSK ( TSK_P4, { (TA_HLNG | TA_ACT), 255, task_p4, 2, 4096, NULL, 0, 25, 0, 1, 1 } );
CRE_TSK ( TSK_P5, { (TA_HLNG | TA_ACT), 255, task_p5, 2, 4096, NULL, 0, 35, 0, 1, 3 } );
CRE_TSK ( TSK_P6, { (TA_HLNG | TA_ACT), 255, task_p6, 2, 4096, NULL, 0, 45, 0, 1, 7 } );
CRE_TSK ( TSK_P7, { (TA_HLNG | TA_ACT), 255, task_p7, 2, 4096, NULL, 0, 50, 0, 1, 15 } );
CRE_TSK ( TSK_P8, { (TA_HLNG | TA_ACT), 255, task_p8, 2, 4096, NULL, 0, 60, 0, 1, 3 } );
CRE_TSK ( TSK_P9, { (TA_HLNG | TA_ACT), 255, task_p9, 2, 4096, NULL, 0, 70, 0, 1, 1 } );
CRE_TSK ( TSK_P10, { (TA_HLNG | TA_ACT), 255, task_p10, 2, 4096, NULL, 0, 80, 0, 1, 3 } );
CRE_TSK ( TSK_P11, { (TA_HLNG | TA_ACT), 255, task_p11, 2, 4096, NULL, 0, 90, 0, 1, 2 } );
CRE_TSK ( TSK_P12, { (TA_HLNG | TA_ACT), 255, task_p12, 2, 4096, NULL, 0, 100, 0, 1, 3 } );
```

Figure 3.4: Important attributes of each task.

is found. So the insert function should be modified to implement Rate Monotonic.

When creating periodic tasks, the tasks should be made activated at regular intervals according to their periods. So in appl.c file, before defining the detail of each task, a function "cyclic handler" is defined to invoke periodic tasks when the periodic tasks arrive at the beginning of their period. The cyclic handler is called at the same frequency as a tick. The method is that, when the cyclic handler is called, it will begin to check from the first periodic task, using the current time divided by the task's period. If at this current moment, the periodic task meets its beginning of the period, the task will be invoked. By this way, periodic tasks are invoked at every regular time interval according their periods.

The figure (Figure 3.5) shows the flow of the task inserting operation. It explains the strategy of Rate Monotonic rule according to the relative deadline. As the picture shows, scanning the ready queues starts from the first priority. At the first step, it is necessary to check whether the ready queue of the current priority is empty. If it is empty, entry can be directly inserted after the head element of the ready queue. Up to now, the insert operation is finished. But if the ready queue is not empty, it is necessary to find the right place to insert the TCB. If at this time the pointed TCB is not the last one, and the inserted entry's relative deadline is equal to or larger than that of the pointed task, the next TCB is pointed to, that is, scanning goes next. If the inserted entry's relative deadline is not bigger than that of the pointed task, the entry is inserted at this position. Then the insert action is finished at this point.

When implementing Earliest Deadline First algorithm, the main method is quite the same as Rate Monotonic algorithm, since Earliest Deadline First chooses a task to be executed first only according to their absolute deadline, while Rate Monotonic algorithm chooses a task by using relative deadline. So the difference of the task selection strategy from Rate Monotonic is only that the comparison is done between the absolute deadlines of two tasks, instead of relative deadlines. So the author can use the same insertion method as Rate Monotonic with the modification of the deadline comparison part.

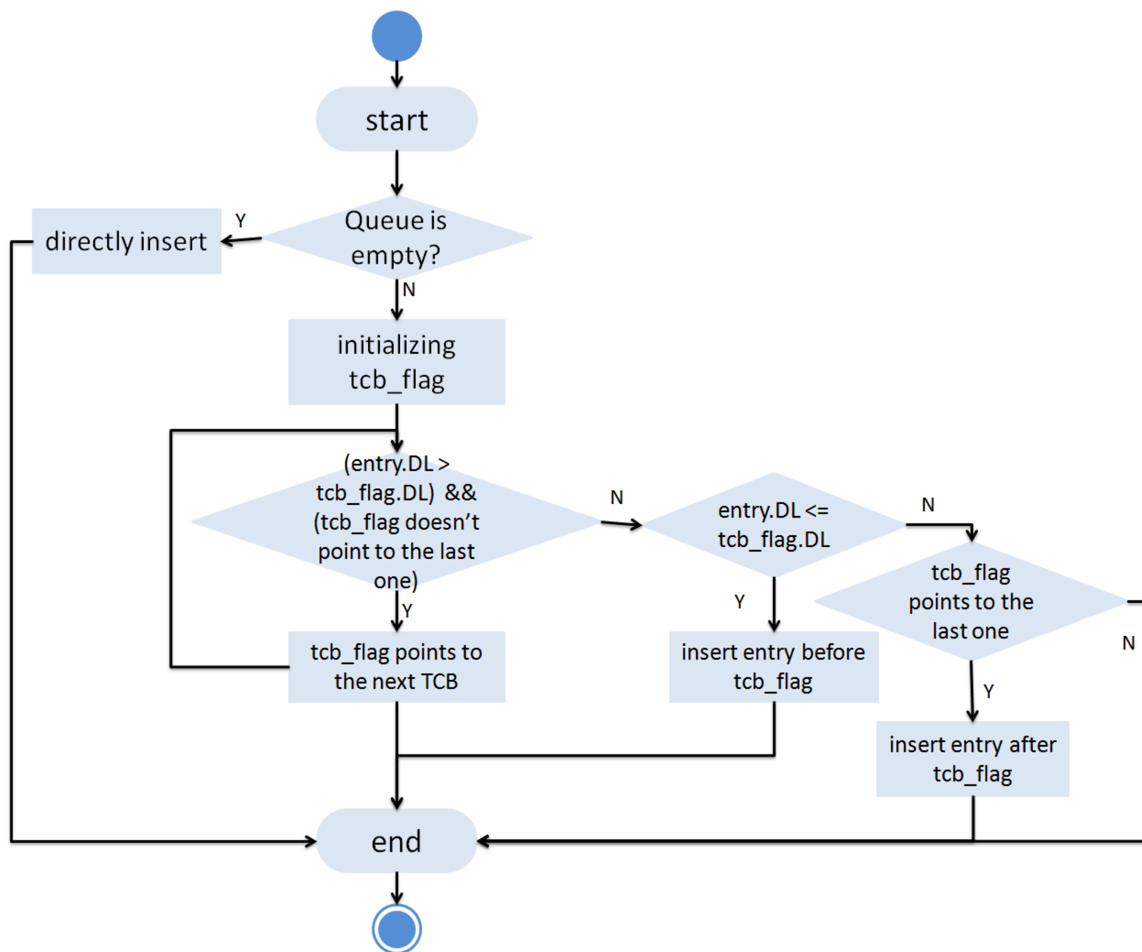

Figure 3.5: Flow of inserting tasks in ready queue using RM.

Another difference from Rate Monotonic is that Earliest Deadline First deals with both periodic and aperiodic tasks. Hence different from the method of Rate Monotonic, aperiodic tasks can be created in this algorithm. The definition of aperiodic tasks is also in appl.c file, but to achieve invoking the tasks in the right planned time, interrupt mechanisms are used to call aperiodic tasks. In the evaluation environment of this research, timing of interrupts is described in the file “irq_tab.c2”.

Priority Exchange is the most complex algorithm among these four algorithms, and most complicated to implement. The scheduler itself needs to be changed in this implementation. Since the capacity conception is imported into this algorithm, the management of capacity must be defined in detail. So the two most significant functions are the scheduling function and the capacity management function. The scheduling function is in the “kernl_sched.c” file, and the capacity management function is added in appl.c file.

The main procedure of Priority Exchange Server is shown in the following figure (Figure 3.6). The procedure happens at every tick timing. As the figure shows, at the beginning of every tick, whether there are kernel's tasks or not should be checked first. Their priorities are 0 and 1. If there is, this priority's request will be responded first. If there is no request of kernel tasks, then whether there is an aperiodic task request or not should be checked. If there is an aperiodic request, the aperiodic task should be disposed. The detail operation is shown in the next figure (Figure 3.7). If an aperiodic request has not been detected, a periodic request will be detected from the highest priority to the lowest. Once a periodic request is found, system will stop scanning and start to deal with this periodic response. The detailed action is shown in the next figure (Figure 3.9). Then this operation is finished.

To deal with aperiodic request, whether there is enough server's capacity should be checked first. If there is enough capacity for the aperiodic request, the aperiodic task is executed and it consumes this server's capacity. Then the aperiodic request's treatment is finished at this moment. If there is no server's capacity found, capacity preserved in the periodic tasks is looked for from the highest priority to the lowest one. After that, either aperiodic task or periodic task will be executed and the capacity information will be changed according to the scanning result as above. The detailed operation is shown in the following (Figure 3.8). Then, this aperiodic request response is finished.

To look for the periodic tasks' preserved capacity, it is necessary to start from the highest priority of periodic tasks. In this study, the highest priority for periodic tasks is priority 3. For each priority, if there is enough capacity found, the aperiodic task will be executed and consume the corresponding capacity in this priority. Then this aperiodic request's response job is finished at this moment. If no capacity is found in this priority, another information should be checked. Is there any unfinished periodic task in this priority? If there is, this periodic task should be executed first. If there is no unfinished

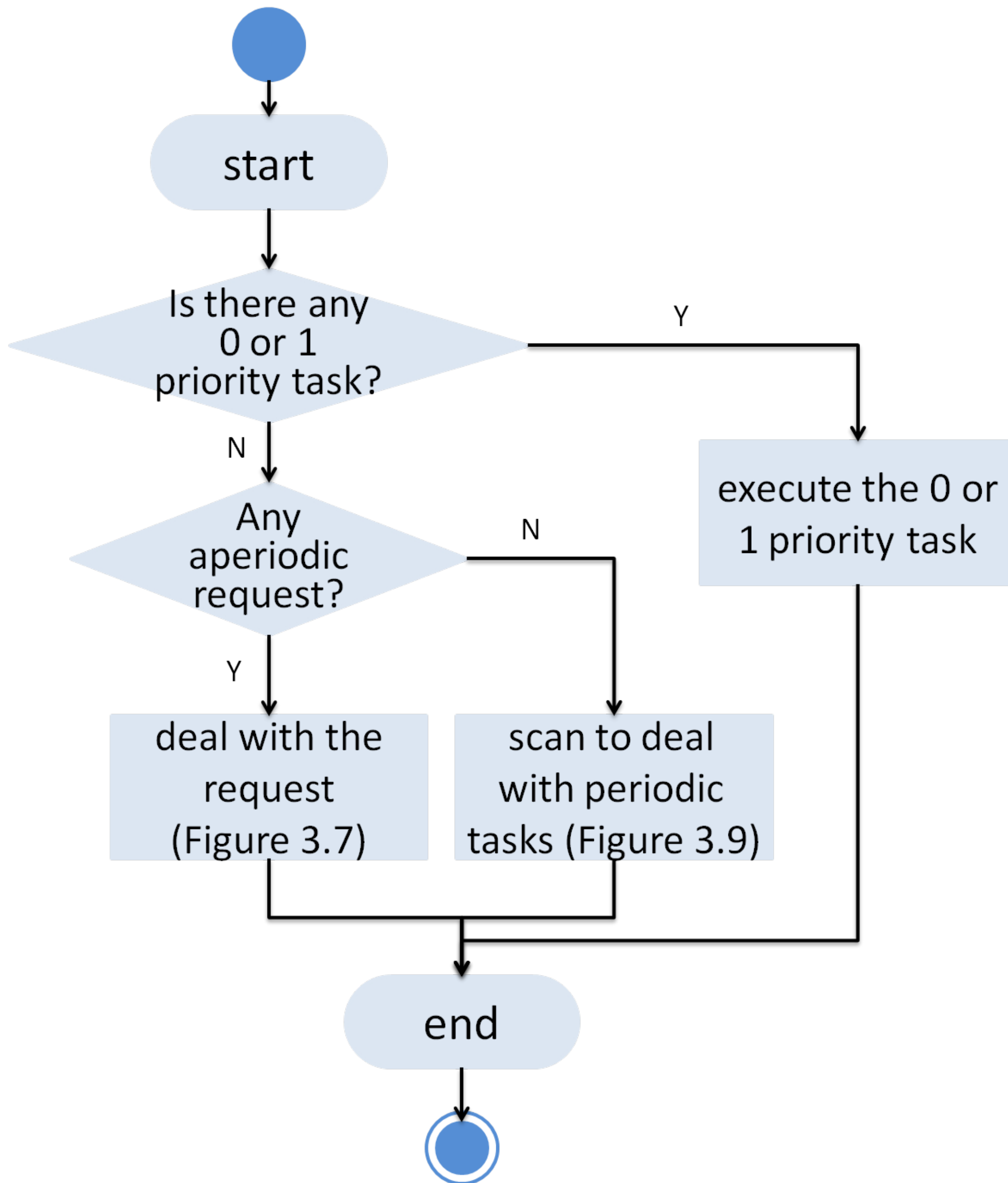

Figure 3.6: Flow of brief scheduling process of PES.

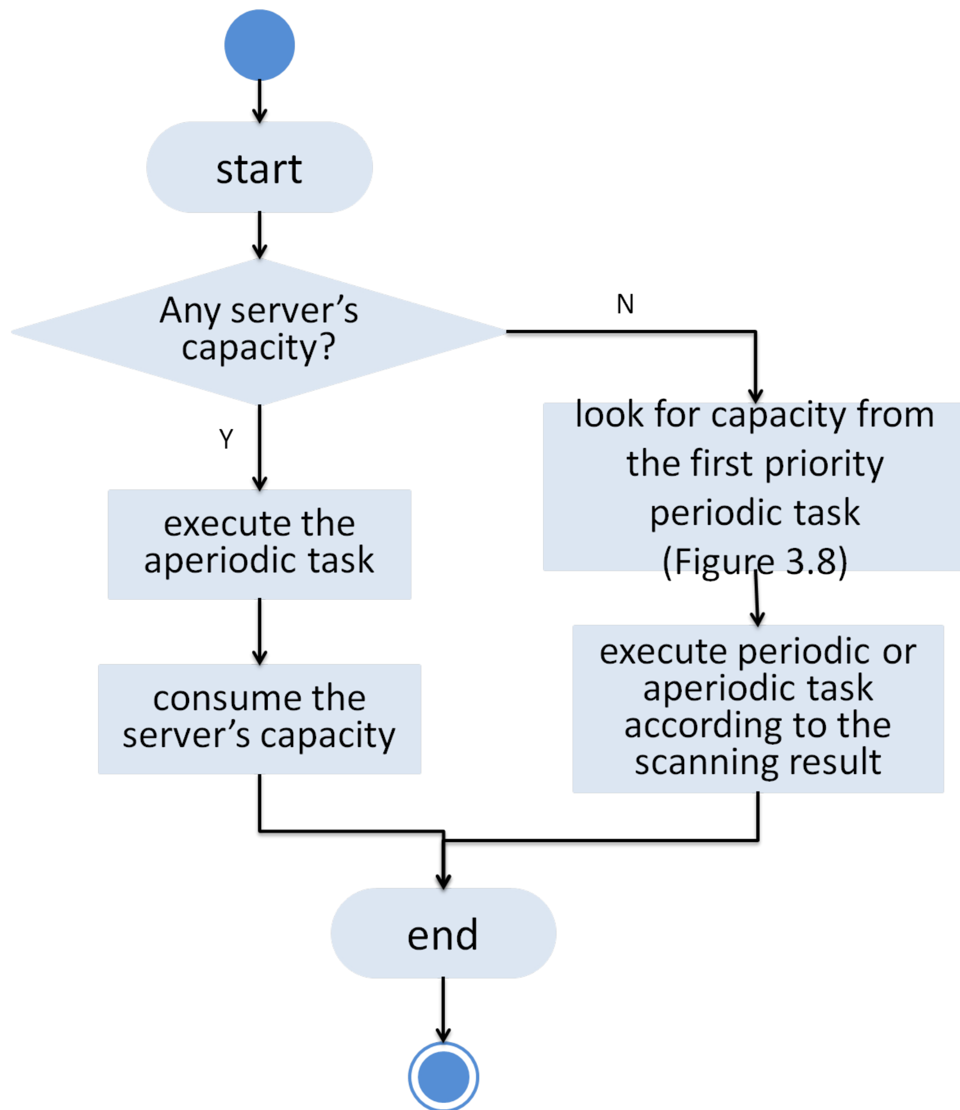

Figure 3.7: Flow for treating with aperiodic request using PES.

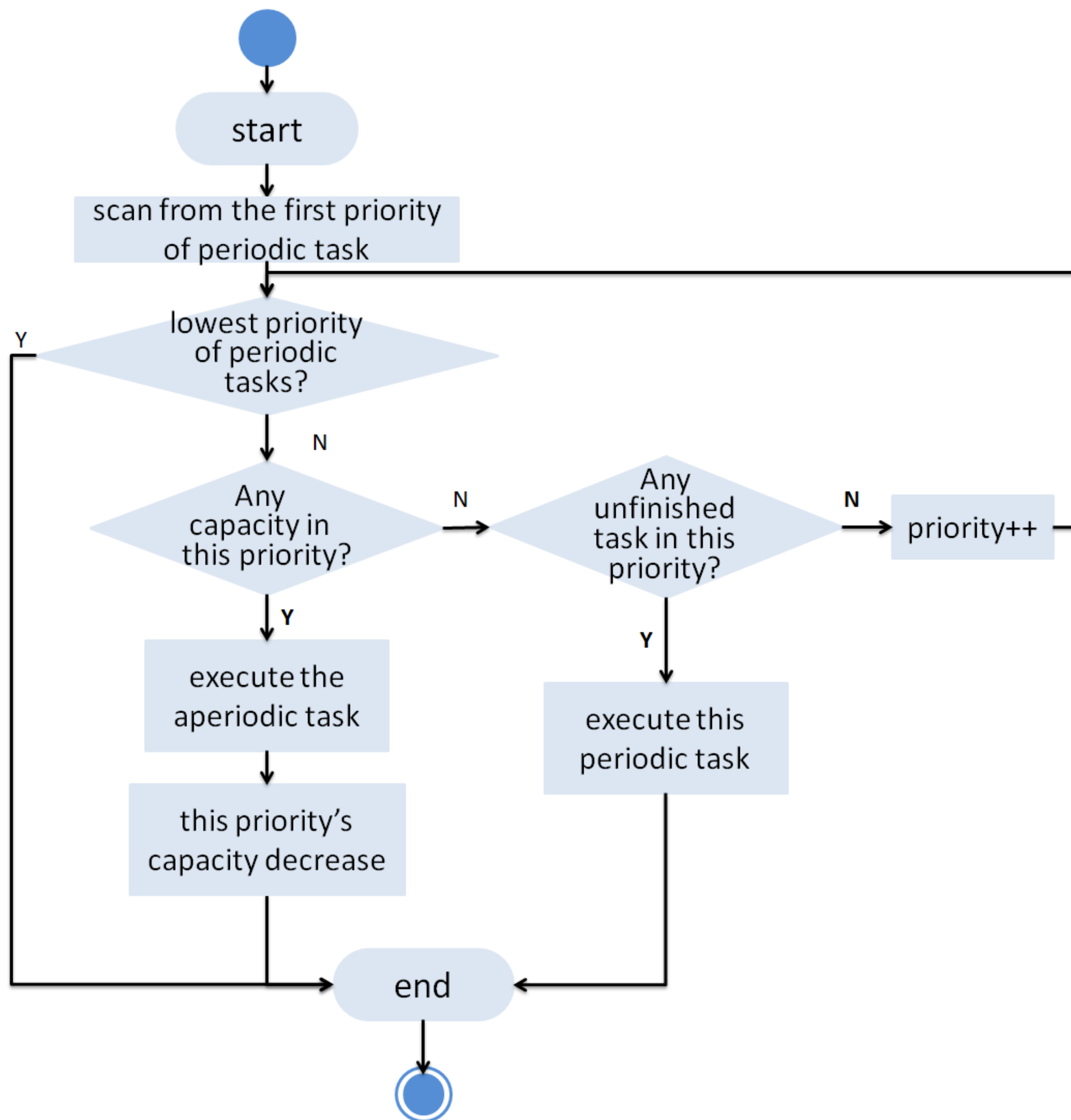

Figure 3.8: Flow for treating with aperiodic request if there is no server's capacity using PES.

periodic tasks found in this priority, the scanning goes to the next priority.

If at the tick timing there is no aperiodic requests activated, the periodic tasks are looked for in the ready queues and treated with (Figure 3.9). It is necessary to start to check from the highest priority of periodic tasks, priority 3. If until the lowest priority there is no periodic request found, the checked lowest priority's capacity, if there is any, should be decreased by one.

The procedure above shows how Priority Exchange Server works at the tick timing. At every beginning of the server's period, the capacity will be replenished. And the cyclic handler explained above will deal with periodic and aperiodic tasks according to the capacity information.

Total Bandwidth Server is based on Earliest Deadline First. The sequence of execution depends on the absolute deadline of each task. While originally, there is no deadline for aperiodic tasks. So to explain its implementation in detail, virtual absolute deadline is assigned to aperiodic tasks. Then the two special members in TCB's definition are used in this algorithm. They are member3 and member4. As the previous usage, the value 0 of member3 indicates the task is aperiodic task. Then the absolute deadline of it will be assigned. On the other hand, the value 1 indicates the task is periodic, then it will follow the EDF rule. How to assign a virtual absolute deadline is shown in the following.

The server's utilization should be calculated first and saved. Let this value be Us. The value Us plus the periodic tasks' total utilization, Up, should not be larger than 1. By accumulating all of the computation time divided by all of the periods for periodic tasks, Up can be obtained. Using 1 minuses Up, the value Us is calculated. Using Us, the virtual absolute deadline of the aperiodic tasks can be calculated. For the first aperiodic task's deadline, the request time plus the computation time divided by Us, becomes the virtual absolute deadline of the task. From the second aperiodic task, comparison between the last aperiodic task's virtual absolute deadline and the current request time is done and the bigger one is selected as the assumed request time. Then the assumed request time plus the computation time divided by Us, becomes the virtual absolute deadline of this aperiodic task (Figure 3.10). The system will schedule these periodic and aperiodic tasks by checking their (virtual) absolute deadlines based on EDF's rule.

3.2 Practical Realization

In the previous section, the author showed basic implementation of the four scheduling algorithms. The implementation was purely based on the scheduling algorithms, except that the scheduling timing, task activation, and task completion occur at any timing, while in their theory, they occur only at tick timing.

Using each algorithm, characteristics of the algorithm can be obtained. Is this true?

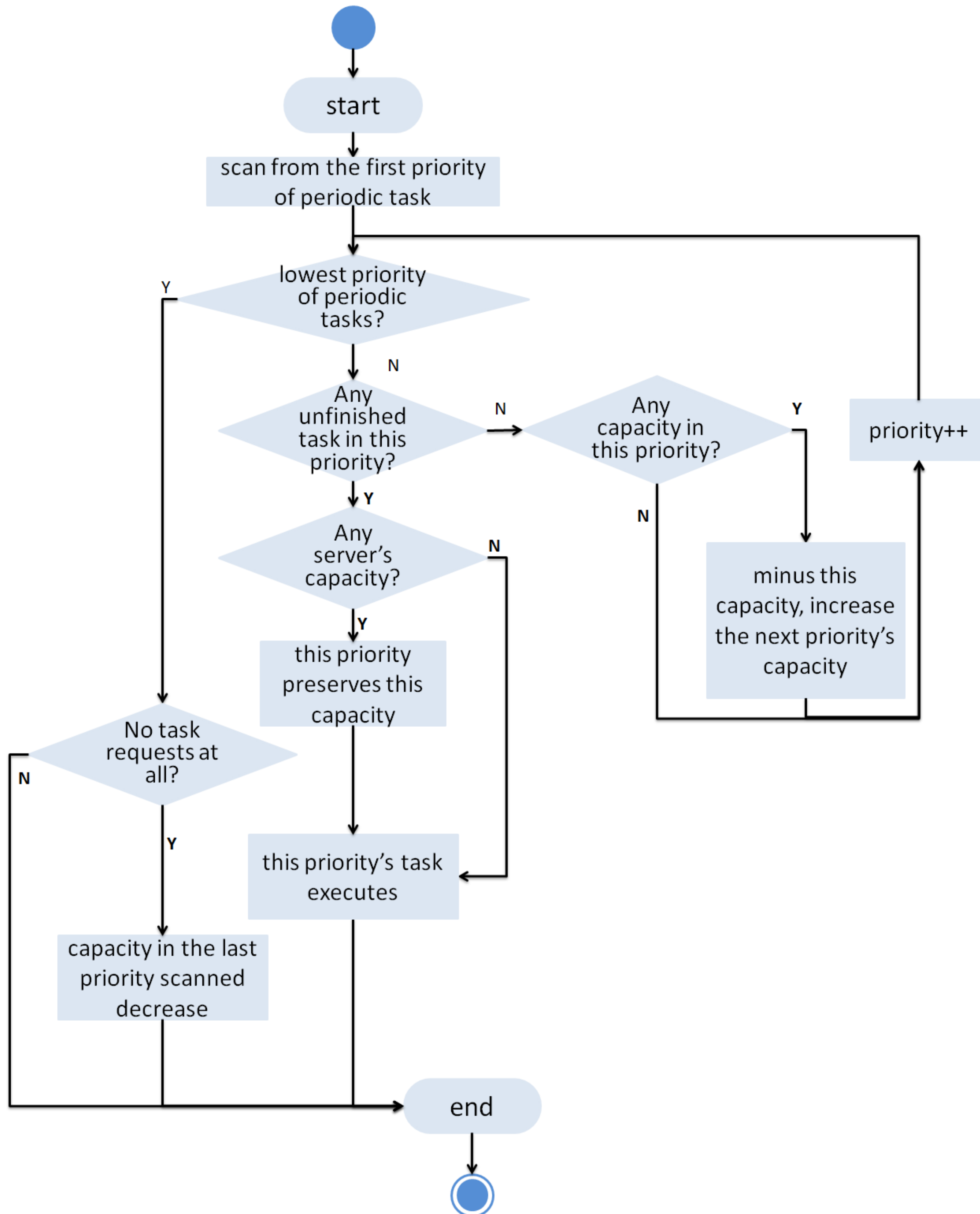

Figure 3.9: Flow for treating with non-aperiodic request using PES.

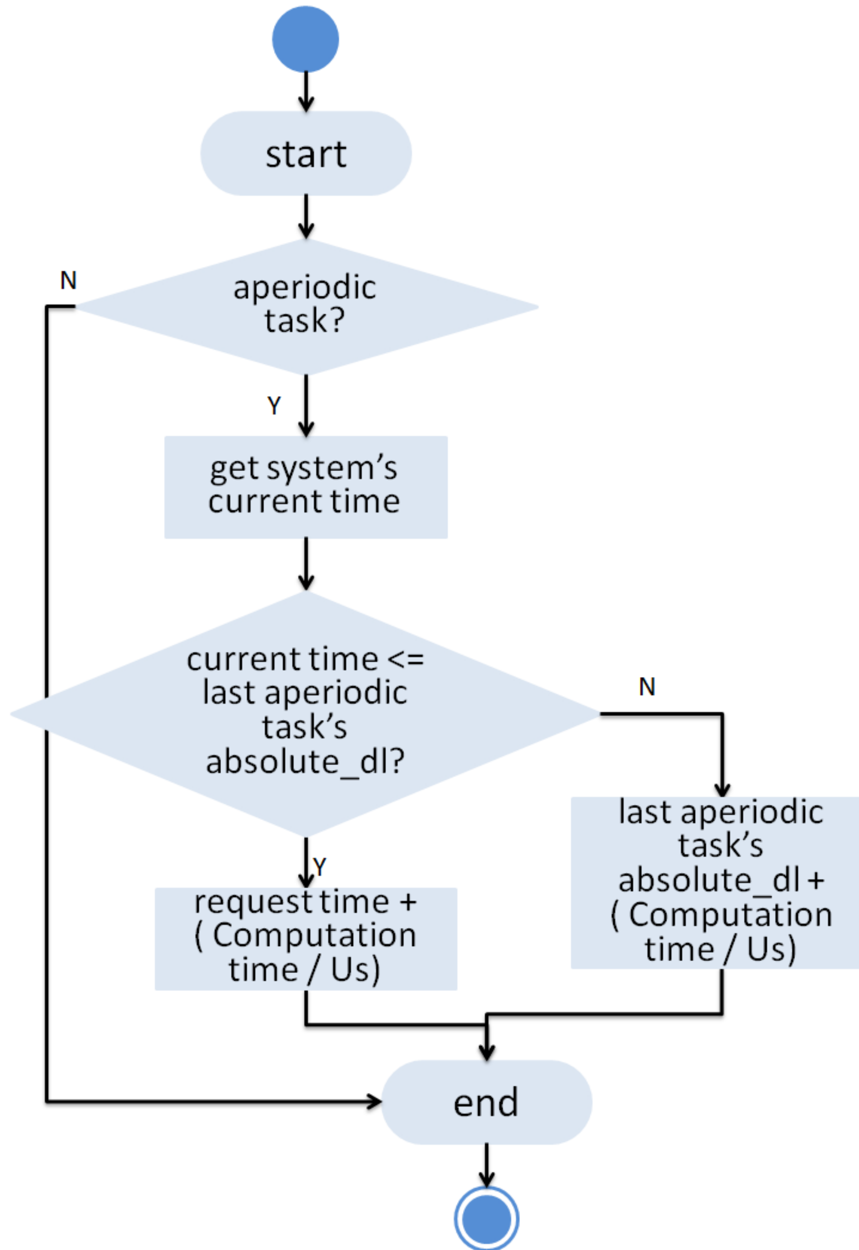

Figure 3.10: Flow of calculating procedure for virtual deadlines of aperiodic tasks by using TBS.

The answer is no. Let's consider the case of Rate Monotonic scheduling. Using Rate Monotonic means that schedulability is guaranteed if the CPU utilization is smaller than 0.69 [2]. It is said that in actual situation, 0.69 is too pessimistic and 0.88 is enough to guaranty the schedulability. The author uses 0.83 as the upper bound in this study. Here, schedulability means that all task executions satisfy their deadlines. Suppose that there are three periodic tasks, $\tau 1$, $\tau 2$, and $\tau 3$, their computation time is 1, and their periods are 3, 4, and 5, respectively. In this case, CPU utilization by tasks is Up = 1/3 + 1/4 + 1/5 = 0.78. This utilization seems small enough to satisfy the deadlines. However, in actual situation, there are various system overheads such as scheduling and context switching. If the total overheads get, for example, 0.08, then the total utilization becomes Up = 0.78 + 0.08 = 0.86, which cannot necessarily satisfy the schedulability. In this section, the author show the practical realization for Priority Exchange Server and Total Bandwidth Server.

In Priority Exchange Server, the server period should depend on CPU utilization by tasks. For example, when the total CPU utilization by tasks is Up = 0.7 and the server capacity is 1, the server period is obtained by $\lceil 1 \ / \ (0.83 - 0.7) \rceil = 8$. It seems that we can guarantee the schedulability with this server period. However, similar to the previous example, the overheads can affect the schedulability. Therefore, the author uses the following formula.

$$U_p + U_s + U_{ov} < 0.83$$

Here, Up is utilization by tasks, Us is utilization by the server, and Uov is utilization by system overheads. By assigning the values to Up and Uov, we can obtain Us (= Capacity/Ts). Then by assigning the server capacity, the server period is obtained.

Next, the system overheads are introduced into Total Bandwidth Server. In Total Bandwidth Server, virtual absolute deadlines are assigned to aperiodic tasks. The virtual absolute deadline is calculated by the following formula.

$$d_k = max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

Here, r_k is the request (arrival) time, d_{k-1} is the virtual deadline of the previous aperiodic task, C_k is the computation time of the aperiodic task, and U_s is the server utilization factor. Generally, U_s is obtained by the calculation $U_s = 1 - U_p$. Similarly to the case of Priority Exchange Server, the author considers the system overheads, that is, $U_s = 1 - U_p - U_{ov}$. By using this calculation, the virtual deadlines are obtained.

Chapter 4

Evaluation

4.1 Environment

After we make all the four existing algorithms usable on ITRON System, we also use the simulator to evaluate each performance of these algorithms. The evaluation environment is clock cycle based simulator. And the inputs are task binary codes and ITRON binary codes. The outputs are deadline missing information, the number of execution of each task, and system overheads.The configuration of the simulator is as follows.
The instruction set is SPARC Architecture version 8 [5].
Each instruction is basically executed in one clock cycle.
(Several instructions, such as multiply, spend three or more cycles.)
The instruction and data caches are separate, 4-way set associative, 16KBytes.
The access latency is one cycle for hit, and ten cycles for miss.
The author will show results of five different task sets as following. First of all, the task sets are explained.

For the first task set, there are only 10 periodic tasks, no aperiodic task, and their main relative information: computation time, periods, are shown as below (Figure 4.1).

	Computation time	Relative deadline
p3	1	15
p4	1	25
p5	3	35
p6	7	45
p7	15	50
p8	3	60
p9	1	70
p10	3	80
p11	2	90
p12	3	100

Figure 4.1: 10 periodic tasks for the 1st experiment task set.

All the four algorithms will be tested using this task set.

For the second task set, there are also only 10 periodic tasks, no aperiodic task, while the only difference from the first task set is that, each of the relative deadline is 5 ticks longer than the first task set 's tasks, as shown below (Figure 4.2).

	Computation time	Relative deadline
p3	1	20
p4	1	30
p5	3	40
p6	7	50
p7	15	55
p8	3	65
p9	1	75
p10	3	85
p11	2	95
p12	3	105

Figure 4.2: 10 periodic tasks for the 2nd experiment task set.

For the third task set, it includes 10 periodic tasks(Figure 4.4), which are the same tasks as the first example and also 6 aperiodic tasks(Figure 4.3).

	Computation time	Relative deadline	Activation time
ap	1	50	9
ap2	2	100	5
ap3	2	150	14
ap4	4	200	20
ap5	6	250	30
ap6	9	300	40

Figure 4.3: 6 aperiodic tasks for the 3rd experiment task set.

	Computation time	Relative deadline
p3	1	15
p4	1	25
p5	3	35
p6	7	45
p7	15	50
p8	3	60
p9	1	70
p10	3	80
p11	2	90
p12	3	100

Figure 4.4: 10 periodic tasks for the 3rd experiment task set.

In the Figure 4.3, computation time, relative deadline, and activation time for aperiodic tasks are all shown.

For the fourth task set, it includes 10 periodic tasks(Figure 4.6), which are the same tasks with the second example and also 6 aperiodic tasks(Figure 4.5) which are the same as the third example.

	Computation time	Relative deadline	Activation time
ap	1	50	9
ap2	2	100	5
ap3	2	150	14
ap4	4	200	20
ap5	6	250	30
ap6	9	300	40

Figure 4.5: 6 aperiodic tasks for the 4th experiment task set.

	Computation time	Relative deadline
p3	1	20
p4	1	30
p5	3	40
p6	7	50
p7	15	55
p8	3	65
p9	1	75
p10	3	85
p11	2	95
p12	3	105

Figure 4.6: 10 periodic tasks for the 4th experiment task set.

For the fifth task set, it is only for PES. The task set also consisted of 10 periodic tasks and 6 aperiodic tasks, while the relative deadlines are different from the previous task sets, which are shown as below (Figure 4.7 and Figure 4.8).

	Computation time	Relative deadline	Activation time
ap	1	50	9
ap2	2	10	5
ap3	2	11	14
ap4	4	12	20
ap5	6	13	30
ap6	9	14	40

Figure 4.7: 6 aperiodic tasks for the 5th experiment task set.

	Computation time	Relative deadline
p3	1	25
p4	1	32
p5	3	30
p6	7	35
p7	15	60
p8	3	75
p9	1	80
p10	3	96
p11	2	200
p12	3	120

Figure 4.8: 10 periodic tasks for the 5th experiment task set.

4.2 Basic Results

We record the deadline missing and overheads results of RM, EDF, PES, and TBS of each task set in order, shown as the following tables.

In the first task set test, we can see the results that, for non-periodic task set, EDF(Figure 4.10) and TBS(Figure 4.12) have no deadline missing, while RM(Figure 4.9) and PES(Figure 4.11) have.

	Deadline missing	Total execution	System overheads
p3	0	69	545082
p4	0	42	
p5	0	30	
p6	0	23	
p7	0	21	
p8	0	18	
p9	1	15	
p10	1	13	
p11	2	12	
p12	4	11	

Figure 4.9: Result of RM on the 1st task set.

	Deadline missing	Total execution	System overheads
p3	0	69	550172
p4	0	42	
p5	0	30	
p6	0	23	
p7	0	21	
p8	0	18	
p9	0	15	
p10	0	13	
p11	0	12	
p12	0	11	

Figure 4.10: Result of EDF on the 1st task set.

	Deadline missing	Total execution	System overheads
p3	0	69	820395
p4	0	42	
p5	0	30	
p6	0	23	
p7	0	21	
p8	1	18	
p9	1	15	
p10	2	13	
p11	4	12	
p12	10	11	

Figure 4.11: Result of PES on the 1st task set.

	Deadline missing	Total execution	System overheads
p3	0	69	551418
p4	0	42	
p5	0	30	
p6	0	23	
p7	0	21	
p8	0	18	
p9	0	15	
p10	0	13	
p11	0	12	
p12	0	11	

Figure 4.12: Result of TBS on the 1st task set.

And the deadline missing of RM is mainly around low priorities, so is PES. That's because TBS is based on EDF, and PES is based on RM. So through this result we can have an assumption that EDF has a better responsive performance than RM, and the fact is right the same. And for system overheads, we can find that, PES produces much more than other three algorithms. That's also because of the complexity of this scheduling strategy. The above results are coincident with the theories.

The second task set is to increase 5 on all periodic tasks' periods, shown as following. Then we also record these results with the same way, shown in the following tables. We can also find the result that EDF(Figure 4.14) and TBS(Figure 4.16) still have better responsive performance than RM(Figure 4.13) and PES(Figure 4.15) for only periodic

tasks.

	Deadline missing	Total execution	System overheads
p3	0	52	545477
p4	0	35	
p5	0	26	
p6	0	21	
p7	0	19	
p8	0	16	
p9	0	14	
p10	0	13	
p11	1	11	
p12	1	10	

Figure 4.13: Result of RM on the 2nd task set.

	Deadline missing	Total execution	System overheads
p3	0	52	547088
p4	0	35	
p5	0	26	
p6	0	21	
p7	0	19	
p8	0	16	
p9	0	14	
p10	0	13	
p11	0	11	
p12	0	10	

Figure 4.14: Result of EDF on the 2nd task set.

	Deadline missing	Total execution	System overheads
p3	0	52	819331
p4	0	35	
p5	0	26	
p6	0	21	
p7	0	19	
p8	0	16	
p9	0	14	
p10	1	13	
p11	1	11	
p12	1	10	

Figure 4.15: Result of PES on the 2nd task set.

	Deadline missing	Total execution	System overheads
p3	0	52	548623
p4	0	35	
p5	0	26	
p6	0	21	
p7	0	19	
p8	0	16	
p9	0	14	
p10	0	13	
p11	0	11	
p12	0	10	

Figure 4.16: Result of TBS on the 2nd task set.

While fewer deadline missings appeared than last task set for RM and PES. At the same time, with the periods increasing, expect for RM, all other three algorithms have fewer system overheads. But PES still has the most large system overheads among these algorithms.

After the first two task sets, we also need to test task sets including both aperiodic and periodic tasks. Also two task sets will be shown as below.

The third task set results are shown below (Figure 4.17, Figure 4.18).

	Deadline missing	Total execution	System overheads
ap	0	1	557081
ap2	0	1	
ap3	0	1	
ap4	0	1	
ap5	0	1	
ap6	0	1	
p3	0	69	
p4	0	42	
p5	1	30	
p6	2	23	
p7	4	21	
p8	3	18	
p9	2	15	
p10	1	13	
p11	2	12	
p12	3	11	

Figure 4.17: Result of EDF on the 3rd task set.

	Deadline missing	Total execution	System overheads
ap	0	1	557939
ap2	0	1	
ap3	0	1	
ap4	0	1	
ap5	0	1	
ap6	1	1	
p3	4	69	
p4	2	42	
p5	4	30	
p6	7	23	
p7	9	21	
p8	5	18	
p9	5	15	
p10	3	13	
p11	4	12	
p12	5	11	

Figure 4.18: Result of TBS on the 3rd task set.

Through the results, we can see that, with the inserting of aperiodic tasks, the deadline missing of both EDF and TBS either increased or appeared.

The fourth task set results are shown below (Figure 4.19, Figure 4.20).

	Deadline missing	Total execution	System overheads
ap	0	1	545948
ap2	0	1	
ap3	0	1	
ap4	0	1	
ap5	0	1	
ap6	0	1	
p3	0	52	
p4	0	35	
p5	0	26	
p6	0	21	
p7	0	19	
p8	0	16	
p9	0	14	
p10	0	13	
p11	0	11	
p12	0	10	

Figure 4.19: Result of EDF on the 4th task set.

	Deadline missing	Total execution	System overheads
ap	0	1	548740
ap2	0	1	
ap3	0	1	
ap4	0	1	
ap5	0	1	
ap6	0	1	
p3	0	52	
p4	0	35	
p5	0	26	
p6	0	21	
p7	0	19	
p8	0	16	
p9	0	14	
p10	0	13	
p11	0	11	
p12	0	10	

Figure 4.20: Result of theoretical TBS on the 4th experiment task set.

The fourth experiment results show that, with the increasing of the periods, the utility is decreasing, so that the deadline missing is smaller than the third task set.

For the fifth task set experiment, we tested it using theoretical PES algorithm. We can find that, there are several deadline missing from p9(Figure 4.21).

Task name	deadline missing	total execution	System overheads
ap	2	3	**783413 clock cycles (Uov = 7.613%)**
ap2	3	3	
ap3	2	2	
ap4	2	2	
ap5	2	2	
ap6	2	2	
p3	0	42	
p4	0	33	
p5	0	35	
p6	0	30	
p7	0	18	
p8	0	14	
p9	1	13	
p10	3	11	
p11	4	6	
p12	6	9	

Figure 4.21: Result of theoretical PES on the 5th experiment task set.

4.3 Practical Results

In the theoretical PES and TBS, we can find both of these two algorithms are good scheduling but complex, so in the practical implementation, the author uses the idea of taking all the system overheads into consideration to improve the responsive capability.

The author uses the third task set to do the experiment on the modified practical TBS algorithm.

Utilization of the server is decreased with considering the utilization because of the overheads. Compared with Figure 4.18, the deadline missing has obvious decreasing as shown below (Figure 4.22).

	Deadline missing	Total execution	System overheads
ap	0	1	558462
ap2	0	1	
ap3	0	1	
ap4	0	1	
ap5	0	1	
ap6	0	1	
p3	0	69	
p4	0	42	
p5	1	30	
p6	0	23	
p7	3	21	
p8	1	18	
p9	0	15	
p10	1	13	
p11	1	12	
p12	2	11	

Figure 4.22: Result of practical TBS on the 3th experiment task set.

The author can say the practical TBS works and does have an improvement for ITRON System.

With enlarging the absolute deadline of the aperiodic tasks, the periodic tasks have better responsive ability.

And for PES, the author uses the fifth task set to test. After the modification, compared with Figure 4.21, the responsiveness has much improvement especially for periodic tasks (Figure 4.23).

Task name	deadline missing	total execution	System overheads
ap	3	3	**804155 clock cycles** **(Uov = 7.815%)**
ap2	3	3	
ap3	2	2	
ap4	2	2	
ap5	2	2	
ap6	2	2	
p3	0	42	
p4	0	33	
p5	0	35	
p6	0	30	
p7	0	18	
p8	0	14	
p9	0	13	
p10	0	11	
p11	0	6	
p12	1	9	

Figure 4.23: Result of practical PES on the 5th experiment task set.

So, the assumption is true through experiment results.

Chapter 5

Conclusion

In this research, the task scheduling is studied in practical real-time operating system, ITRON. We first chose four typical existing algorithms: Rate Monotonic, Earliest Deadline First, Priority Exchange Server and Total Bandwidth Server, and implemented them on ITRON System for tasks' scheduling. Their performances are evaluated and compared with different task sets. The results show that the implemented algorithms on ITRON System have the coincident performance with their theories. For example, when testing RM algorithm, we can find that, lower priority tasks are easier to miss the deadline, and so is PES, because PES is based on RM. And another example, TBS is more lightweight computing than PES, but does not have better responsive capability than PES. That is because PES cost more decisions and task switches to achieve a better task scheduling.

And then two ideas of PES and TBS are raised to improve the scheduling strategy on ITRON System according to the evaluation results. These two strategies are also implemented on ITRON System and evaluated. Both PES and TBS's new ideas have shown obvious improvement on responsive performance. So the author can say that the idea works. In the future, the author will continue testing more task sets to prove the improvement of both PES and TBS.

Chapter 6

References

[1] ITRON Committee, TRON ASSOCIATION. ITRON4.0 Specification Ver.4.00.00.

[2] C.L.Liu and J.W.Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, Journal of ACM, 20(1), 1973.

[3] J.P.Lehoczky, L.Sha, and J.K.Strosnider, Enhanced Aperiodic Responsiveness in Hard Real-Time Environments, Proc. of IEEE Real-Teim Systems Synposium, 1987.

[4] M.Spuri and G.Buttazzo, Efficient Aperiodic Service Under Earliest Deadline Scheduling, Proc. of IEEE Real-Time Systems Symposium, 1994.

[5] SPARC International, Inc. The SPARC Architecture Manual Version 8. Prentice-Hall, Inc., 1992

Chapter 7

Acknowledgement

In this research, the author should express the gratefulness to every body related.

First of all, I need to thank my parents. They are my spiritual pillar for every thing. When ever I come up with any kind of difficulty, they are always very supportive and encourage me a lot. By the great positive factor, can I suffer from any barrier.

The next I must thank my great instructor Associate Professor Tanaka Kiyofumi. He is more than kind, patient, responsible and professional. I was once very lack of related knowledge. Tanaka Sensei was very nice to guide me reading and studying relative documents. During the implementing process, no matter when I come up with problem or how difficult the problem is, Tanaka Sensei did all he could do to help me and told me how to solve similar problem at the same time. Without his best effort, I could never gain such achievement of this research.

Then I need to thank my assistant professor and other members in my laboratory. They are always very nice and kind to help me, even as a foreigner, they are very hard to communicate with me. Im very thankful that they are always ready to help, and every time very patient.
At last, I need to thank other friends of mine and many JAIST faculties. My friends fulfilled much of my spare time as Im in another country, which is very far away from my home town. They gave me happiness and warmness. And the faculties did their best job to let every process go smoothly.

All in all, there must be someone which I missed to thank. Anyway, I need to thank every one related, and thank JAIST, and other relative environment to give me such a chance to study, discover the Information Science knowledge. Thank every one and every thing!