SMART REAL-TIME OPERATING SYSTEM

CHEN HUITING
(B. Eng., Shanghai Jiaotong University, P. R. China)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2007

ACKNOWLEDGEMENTS

I would like to thank many people who have made it possible for me to complete my Ph.D study in NUS. I wish to express my greatest and sincerest gratitude to my supervisor, Associate Professor Kenneth Ong Kong Wee, for his guidance, warm encouragement and considerate understanding throughout the courses of the research work. It is because his invaluable advice that I can accomplish this work. I would appreciate for his friendly and professional approach.

I would like to thank all of my friends and colleagues who contributed in various ways to this work, especially Mr. Ganasa for his useful advices. I wish to thank the examiners who kindly sent me useful advices to improve my presentation.

I wish to thank to thousands of volunteers of open-source development who contributes to the Linux Kernel. I would like to thank many of them for helping me clarify some doubt in my work.

Special thanks go to my family members for their greatest support and encouragement. I am most grateful to my mother and father for their understanding.

TABLE OF CONTENTS

SUMMARY

Presently, Linux becomes more and more popular because it can work on various hardware platforms. Many applications such as media processing and 3D games have the requirement of real-time response; however, Linux kernel is less flexible when scaling to real-time applications.

The aim of this study was to develop a smart real-time operating system that improves the system performance and enhances the real-time properties of standard Linux with high compatibility. Firstly, this system is built with preemptive patch, long-latency patch and queue manager mechanism to improve the response accuracy. Besides, a real-time control subsystem is built into the operating system to deploy real-time tasks and scale to real-time applications. In this hybrid operating system, the real-time tasks share all the primitives in the standard Linux kernel, which helps the tasks to access the full range of Linux facilities. On the other hand, the real-time tasks have some privilege priorities over the other non-real-time processes. In the real-time control subsystem, some commonly used scheduling algorithms are built-in and a flexible scheduling framework is presented to optimize its compatibility. Moreover, our system also targets on support of acyclic task execution. A hybrid process model is presented to investigate the task scheduling of acyclic task execution. The schedulability analysis in this model is conducted to provide a theoretical basis for the real-time scheduling. A new scheduling algorithm to deploy aperiodic tasks is presented.

Experimental results associated with system performance evaluation, cyclic and acyclic execution has been presented in this thesis. Results of system evaluation in terms of response latency showed an optimized performance of timing response accuracy is achieved in our system. Results of cyclic and acyclic execution also proved that this real-time operating system has the capability to deploy the real-time tasks and guarantee the timing constraints using various fixed-built scheduling policies or using a flexible scheduling framework. Results also showed the effect of Queue Manager (QM) mechanism on response accuracy and schedule precision. In the comparison of these two cases, the results show QM mechanism improves the schedule precision. Thus this study has developed a hybrid real-time operating system and provides a good platform that achieves optimized timing response accuracy and realizes the real-time task scheduling.

NUMERATION

Symbols

jiffies	The number of time slices
Laxity	The remaining execution time of a real-time task
cur_dd	The deadline of a real-time task
C_i / T_i	The CPU utilization by i^{th} task
GCD	Greatest Common Divisor
B_n	Schedulable bound of *n* tasks
L	Laxity
$U_{\max}$	Maximum fraction of processor utilization
NR_UDS	Maximum amount of tasks registered in UDS scheduling framework
QM	Queue manager is a component of RTS driver
OSCR	OS Timer Count Register for StrongARM SA-1110
RL_i	Response Latency of sample i
C_i	The execution time of i^{th} task
D_i	The deadline of i^{th} task
AP_SHED	Acyclic execution of sporadic task
IMP.	Improvement of Schedule Precision

Abbreviations

RTS	Real-time System
RTOS	Real-time Operating System
RTS-Linux	Real-time Supported Linux
POSIX	Portable Operating System Interface
I/O	Input/Output
API	Application Program Interface
CPU	Central Processor Unit
RM	Rate Monotonic
EDF	Earliest Deadline First
MLF	Minimum Laxity First
UDS	User-defined Scheduler
SoC	System on-Chip
MMU	Memory Management Unit
JFS	Journalized File Systems
NFS	Network File Systems
FAT	File Allocation Table
VFS	Virtual file system
IPC	Inter-process communication
SSL	Secure Sockets Layer
FSS	Flexible Sporadic Server
NP	Nondeterministic Polynomial
QM	Queue Manager
RTS(non-QM)	Usual Real-time Task Scheduling without Queue Manager supported .

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

Presently more and more user applications like 3D games, networking communication and media players have the requirement of good response accuracy to the external event. According to the constraints to response accuracy, the applications can be fit into two groups: soft and hard real-time applications. The first group is the applications with coarse real-time constraint, while the second group does not produce any predicted result if its timing constraints are violated.

The satisfaction of the response time requirements relies on the cooperation of applications and Operating systems. Among various operating systems, Linux has drawn more and more attention as a general-purpose operating system that can work on many hardware platforms. The good reliability, scalability and low-cost makes Linux an attractive operating system. A wealth of development tools and open-source applications helps to develop the kernel and applications conveniently. Besides, the compatibility of Linux makes it to be easily ported on various hardware platforms.

In order to make this general-purpose operating system to realize the real-time controls, various hybrid real-time operating systems (RTOS) have been proposed. Two approaches are applied to build such a hybrid RTOS: making use of a pre-emptive patch and using dual-kernel. In order to understand the real-time controls in Linux system, an overview of RTOS is presented in the following section.

1.1 Real-Time Systems

There have been many studies on real-time systems and real-time operating systems. Martin [1] describes the concept of Real-time System (RTS). A System is considered to be real-time system if it responds to the external events and performs functions within guaranteed time. In such a system, a real-time kernel offers support for the real-time applications. For example, MARS system [2] controls the timing response for distributed applications. A more effective example is *Spring kernel* [3], which offers the real-time controls for both multiprocessor and distributed systems.

In the real-time system for both uni-processor and multi-processor system, there are over 200 real-time systems specifically for embedded platform [4]. Inside these systems, the famous commercial real-time systems include Vxworks developed by Wind River system co., VRTX made by Mentor Graphics co, OS-9 by Microware and so on. Similar to the real-time applications, real-time systems are categorized into soft real-time systems and hard real-time systems according to their performance of response time. A hard real-time system fails when the timing constraints are violated. A hard real-time system has to work cooperatively with specific hardware as well as specific applications. One example is cruise control system that was designed by Hassan Gomaa in 1989 [19]. A soft real-time system takes just temporal and temporary failure when the timing constraints are violated, such as an online media player. When one packet is lost, a media player may fail for a short time. Then the player resumes to normal and continues to process the packets. Such a media player may be one application in mobile audio machine or one component of a complex operating system.

A Real-time Operating System (RTOS) is an operating system that performs the real-time controls and thus is more complex than a real-time system. RTOS provides more functions and contains more software like file systems and GUI windows that make the operating system friendly to the users. RTOS is an operating system that executes programs within a guaranteed upper bounded time. Depending on specific operating systems and applications, the response time of a certain task varies from scale of milliseconds to scale of minutes [6].

According to the development approaches, RTOS can be categorized into two groups. Some types of RTOS are modified or optimised from some timing-sharing operating systems. The modified examples are QNX [6] and LynxOS [7], and they are compatible with UNIX. Another group of RTOS is the completely "new" operating systems that are developed from clean state. One example of "new" RTOS is Vxworks commercial RTOS. A "new" RTOS is incompatible with UNIX, and it has more specific utilities and a smaller size than "modified" RTOS.

Some studies have shown that a Real-time Operating System (RTOS) has many important features such as interrupt handling, process management cached memory, and so on [4, 12]. These features make it possible to support the facilities of operating system and the control of real-time events. In order to respond to the external asynchronous events, RTOS must have the capability of interrupt handling. Besides, to make the interrupt handling predictable, RTOS adopts a pre-emptive scheduling in the process management. In the memory management, RTOS presents the facility of cached memory to keep a part of software and avoid the frequent accesses to the hard disk.

There are several focuses of research interest in the development of a hybrid RTOS [2, 15, 17]. One subject of research work is to develop a hybrid RTOS with Application Programming Interfaces (APIs) that are compatible with POSIX [20] and allow the developers to create their applications. Another trend of research work is to extend the real-time controls to the networking traffic control [14]. The study on the security and real-time scheduling of network traffic in the hybrid Linux system becomes a new hot-point of research on hybrid RTOS. There have been many studies of the hybrid real-time Linux ported on various platforms, especially on the embedded platforms [12, 13]. In order to illustrate the hybrid Linux system, we will introduce the architecture and some characteristics of Linux system in the following section.

1.2 Linux Operating System

Linux is a general-purpose operating system designed to provide an open source operating system and achieve good balanced performance. The developers all over the world have optimised its system performance. With the efforts of these developers, currently the management functions and characteristics of Linux system become mature. As Linux system is very comprehensive, many papers and books have introduced the implementation of Linux system. Michael and David introduced the main mechanisms of Linux and showed their merits and disadvantages [20-23]. Linux is a multi-process system, that is, many processes can be deployed in the system and share the processor resources.

Linux already provides all of the capabilities expected by a general-purpose operating system with multi-process. These include extensive support for multi-threading, multi-processing, simultaneous users, memory management and protection, architecture-independent features, POSIX support, multiple file systems, network support etc. However, like other multi-processes operating systems, Linux contains many structural elements that severely limit its ability to meet response time constraints.

As Linux is a timing sharing OS, its structural elements limit its ability to meet the response time constraint of the tasks [11, 25]. Thus Linux has some drawbacks in task scheduling:

1. Linux timer mechanism has several drawbacks. First, the frequency of periodic timer is only 100Hz, which cannot meet with real-time requirements. Second, the soft real-time is implemented with timer mechanism. If there are frequent soft timers being called, the conflicts between timers sharing may happen. Third, the interrupt handler is not schedulable. But in real-time systems, we expect that all interrupt handlers can be scheduling in the full set of interrupt handlers. Therefore we can determinate the priorities of tasks. For these reasons, the solution of shorten time slice is not a good solution to enhance real-time property.
2. Linux provides round-robin scheduling algorithms for real-time processes. This scheduling algorithm can only achieve the response time at a scale of seconds. If a real-time process cannot run within specific interval, its priority will be decreased and makes the process miss deadline.
3. Although Linux provides real-time processes with the higher-priorities than other processes, this scheduling only deploys the real-time tasks with

only First In First Serve (FIFS) scheduling algorithm. On the other hand, Linux did not assign the real-time tasks with timing constraints, such as deadline, period etc. Meanwhile, a large amount of non real-time processes may block the execution of the real-time processes, which makes the real-time requirements cannot be satisfied.

1.3 The Objective of Study

The aim of the study was to improve the timing response accuracy and develop a smart real-time operating system that supports real-time control with high compatibility in a general-purpose operating system. The objectives of this study were as follows:

- To develop a configurable real-time kernel for multiple real-time applications and a loadable kernel module (LKM) that can choose compactable facilities and deploy real-time tasks.

- To improve the system performance of Linux in terms of timing response accuracy. This improvement of response accuracy is dependent on the reduction of response latency and guaranteed timing constraints. In order to reduce the response time, the preemptive patch and the LLP patch were inserted in the standard Linux kernel. To meet the timing constraints of real-time tasks, a real-time scheduler inside LKM is used to deploy real-time tasks. Our proposed Queue Manager (QM) mechanism is used to optimize the schedule precision.

- To realize the cyclic and acyclic execution of real-time tasks and present application programming interfaces (APIs) that can interact with the kernel and the applications.

- To present a flexible scheduling framework that allows the developers to design their own scheduling disciplines. The proposed user-defined scheduler (UDS) includes some APIs of writing and applying some scheduling policies.

- To analyze the response times in the synchronous model and the asynchronous model and derive the formulation of response time of real-time tasks. This analysis is made to determine the bound of the workload and inter-arrival time of aperiodic tasks. In order to verify the formulation of response time, some simulations of task scheduling were proposed.

- To investigate the improvement of timing response in the hybrid Linux system. Thus some experiments to measure the response latency are conducted in the environment of light and stress system load. Furthermore, some experiments to investigate the task execution were conducted to show the optimization of timing response.

In the objectives of this study, the cyclic and acyclic execution of real-time tasks is the central part of developing a real-time operating system. Some scheduling mechanism is proposed in the execution of real-time tasks. In the designed hybrid operating system, a real-time process is proposed to hold a higher priority than a standard Linux processes. This real-time process is proposed to share all the primitives with the other

Linux process, which enables it to access the full range of facilities of Linux. Therefore RTS-Linux is compatible with Linux-based open source applications. This study only concentrates on the task scheduling and response accuracy of kernel processes, and does not present the real-time control of continuous networking traffic flows.

This research may provide a compact and configurable system that allows users to set up a kernel compatible with their utilities. The flexible scheduling framework may help developers to design and use alternative scheduling disciplines. The implementation of LKM may make it easy to port our real-time execution on updated versions and other platforms. Besides, the analysis of response time may serve as a theoretical base for a more efficient schedulability test. To provide a foundation for the study of a hybrid real-time operating system, some research work on real-time systems and real-time Linux is reviewed in chapter 2.

1.4 Thesis Outline

This thesis consists of 8 chapters. The contents of each chapter are highlighted as follows. Chapter 1 is a brief introduction to our research works. For real-time systems, it provides a thorough review on its relative terminology and development trend. For the real-time property under Linux, it briefly introduces the background and disadvantages of Linux.

In Chapter 2, we further investigate some previous works related to our research. As an UNIX-like operating system, Linux is a multi-processes OS. We adopt the standard Linux as a basic platform for embedded real-time application. Thus this chapter introduces several add-on options that bring real-time capabilities to Linux system as well as a wealth of commercial real-time systems.

Chapter 3 describes some commonly used real-time scheduling algorithms, including the cyclic executives (particularly RM and EDF algorithms) and some schemes to scheduling aperiodic tasks. Besides this, the process management and scheduling paradigm in the standard Linux are illustrated.

Chapter 4 details the system design of RTS-Linux. Section 4.1 introduces the requirement and assumption of the designation. In section 4.2, we discuss two approaches applied on the real-time control on the Linux. Section 4.3 describes the basic concepts and services in RTS-Linux.

Chapter 5 defines a new process model for acyclic task execution and presents a simplified approach for worst-case response time (WCRT) in real-time process model. A new process model composed of periodic tasks and aperiodic tasks is defined in section 5.1. Section 5.2 and 5.3 introduces the new approach for WCRT prediction and validate this approach with some simulation. Using the hybrid process model, it is presented the schedulability analysis of hybrid process model in section 5.4. One scheduling algorithm for the acyclic execution is presented in section 5.5.

Chapter 6 presents the system implementation of RTS-Linux. In this chapter, the content focuses on system architecture, the real-time scheduling policy and the important facilities of real-time properties. Chapter 6 also briefs the queue manager (QM) mechanism and user interface in RTS-Linux.

The experimental results of performance evaluation and measurement of RTS-Linux are shown in chapter 7. The performance evaluation mainly focuses on the response latency, scheduling performance of RTS & flexible scheduling framework and schedule jitter.

In Chapter 8, we summarize our research work and present some suggestions for the future research work.

CHAPTER 2
RELATED WORKS

This chapter briefly introduces the present solutions of real-time operating systems. Several top-level commercial real-time operating systems (RTOSs) are discussed in short. The features of Linux are introduced from the viewpoint of operating system in details. Finally two approaches to enhance the real-time control properties of Linux and some real-time add-on options of Linux are described.

2.1 Existing Real-time Operating Systems

A market survey performed by Real-Time Magazine [3] shows that Vxworks; windows CE, QNX, and VRTX pSOSystem are five popular real-time operating systems. These five operating systems are introduced below briefly.

2.1.1 Vxworks

VxWorks [4] is the most widely adopted RTOS developed by Wind River. It has been widely applied in the fields of robotics, process control and flight simulation control. It has also been used in the applications in the area of telecommunications, consumer electronics, data networks and bioengineer simulation.

The micro-kernel of VxWorks (Wind River) makes use of multiple functional modules. The wind provides the functions of multi-process scheduling, interrupt handling, inter-process communication and timer management. In the process scheduling, wind supports both pre-emptive and round robin scheduling. The priority inheritance algorithm and the priority ceiling algorithm have been deployed in wind to take care of the priority inversion. VxWorks is also POSIX compliant and supports real-time extensions such as asynchronous Input/Output (I/O) control, semaphore, signal and memory management.

2.1.2 pSOSystem

Another RTOS presented by Wind River is pSOSystem [5]. pSOSystem is a multitasking system designed for network applications on the embedded systems. It provides the components of memory management and resource monitor. pSOSystem runs under protected mode and adopts efficient exception management to avoid system crash. Wind River also presents a full set of debug tool and development environment. In pSOSystem, a priority-driven scheduler supports preemptive scheduling and external interrupt handling. Additionally, event driven operations are also offered by allowing tasks to wait for multiple shared resources simultaneously.

Compared with VxWorks, pSOSystem provides more efficient and powerful network facilities including TCP/IP stacks, LAN/WAN protocol, RPC, NFS client/server protocol and HTTP etc.

2.1.3 Windows CE

Microsoft's Windows CE [6] is a real-time operating system designed for the handheld platform and applications requiring a small footprint. It supports wireless technologies and secures sockets layer. It provides 256 levels of thread priority, wrapped interrupts and mechanism of priority-inversion.

Windows CE supports many hardware platforms including ARM720T, ARM920T, ARM1020T, StrongARM, MIPS II/32 with FP, X86 and Pentium processes. However, Windows CE does not support the POSIX APIs. As Windows CE is a real-time version of Windows, Microsoft presents powerful development tools and environment. Windows CE developers can build and test the design on their Windows 2000 and Windows XP workstation.

2.1.4 QNX Neutrino RTOS

QNX Neutrino RTOS [7, 8] is built on microkernel architecture targets (small footprint, real-time executives, and high reliability). Neutrino architecture is similar to Linux and UNIX. It can be built to run under x86, PowerPC, and MIPS processors. Neutrino is a highly modular and scalable OS.

The QNX microkernel provides multiple components such as thread scheduling and inter-process communications. In QNX, message passing is more commonly used than

a form of inter-process communication (IPC). Such a mechanism synchronizes the execution of cooperating components and encourages its maintenance. All the OS services run in the protected memory space. For example, if a device driver requires to access memory outsides its process space. To avoid the system crash, QNX will terminate the process in error and release all the resources allocated.

Another RTOS presented by Wind River is pSOSystem [5]. pSOSystem is a multitasking system designed for network applications on the embedded systems.

2.1.5 VRTX

Mentor Graphics makes Virtual Real-time Execution (VRTX) operating system. VRTX is developed by enabling operations for the real-time environment. The real-time executive is essential in the time-critical processors. VRTX provides 255 levels of priority and 350 ms context switching. VRTX is also integrated with a communication subsystem based on ISO standards.

VRTX RTOS provides an advanced, high-performance solution for System on-Chip (SoC) and traditional board-based systems [9]. VRTX product family comprises two distinct solutions: VRTXmc and VRTXsa. VRTXmc is a single and compact real-time executive. Comparatively, VRTXsa is suitable for complex system and supports dynamic task and resource control. VRTXsa supports a complete range of embedded applications through a modular architecture that enables user to select only those necessary components. VRTX offers the best possible combination of overall

performance and reliability. VRTX presents preemptive system calls and priority inheritance mutex. In VRTX, any number of tasks can be rescheduled even when the kernel code is executing. That is, a new task can be scheduled as soon as it is ready to execute. Thus an application will not stopped by the lengthy context switching. For the processors with MMU support, VRTX allows designers to access the memory in fine-grain cache control and other ways. VRTX also offers powerful and complicated development tools to help testing and debugging.

2.2 Feature of Linux

Linux is a general-purpose operating system designed to provide an open source operating system and achieve good balanced performance. The developers all over the world have optimized its system performance. With the efforts of these developers, currently the management functions and characteristics of Linux system are mature. As Linux system is quite comprehensive, many papers and books have been published to introduce the implementation of Linux system. Michael and David introduced the main mechanisms of Linux and showed their merits and disadvantages [23-25]. One mechanism in Linux is the timing sharing mechanism, in which the CPU time is divided into several slips and is assigned to the processes according to some policies. Besides, Linux is also a multi-process system, that is, many processes can be deployed in the system using a timing sharing mechanism.

This section describes the architecture and features of Linux. This section is divided into three sub-sections: architecture of Linux, functions of Linux Kernel and timing-Sharing Feature of Linux.

2.2.1 Architecture of Linux

Linux is a free, open source and has all the important feature of operating system. It is quite similar to Unix system in the process management. Linux operating system is composed of four major subsystems:

1. **User Applications** - the set of applications in use, which is different depending on what the computer system is used for, typically they include a text editor and a web-browser.
2. **O/S Services** - these are services that are typically considered part of the operating system (such as X-window and shell); the programming interface to the kernel (compiling tool chain and library) is included.
3. **Linux Kernel** - this abstracts and mediates access to the hardware resource.
4. **Hardware Controllers** - this subsystem is comprised of all the possible physical devices, such as the CPU, memory hardware, hard disks, and network hardware.

The architecture of Linux is detailed in Figure 2.1. Linux kernel is always resident in memory and provides the interfaces between user programs and the computer hardware. The kernel is the heart of Linux, which includes the abstraction layer of hardware, protective layers around kernel such as user authorization and interaction with user space.

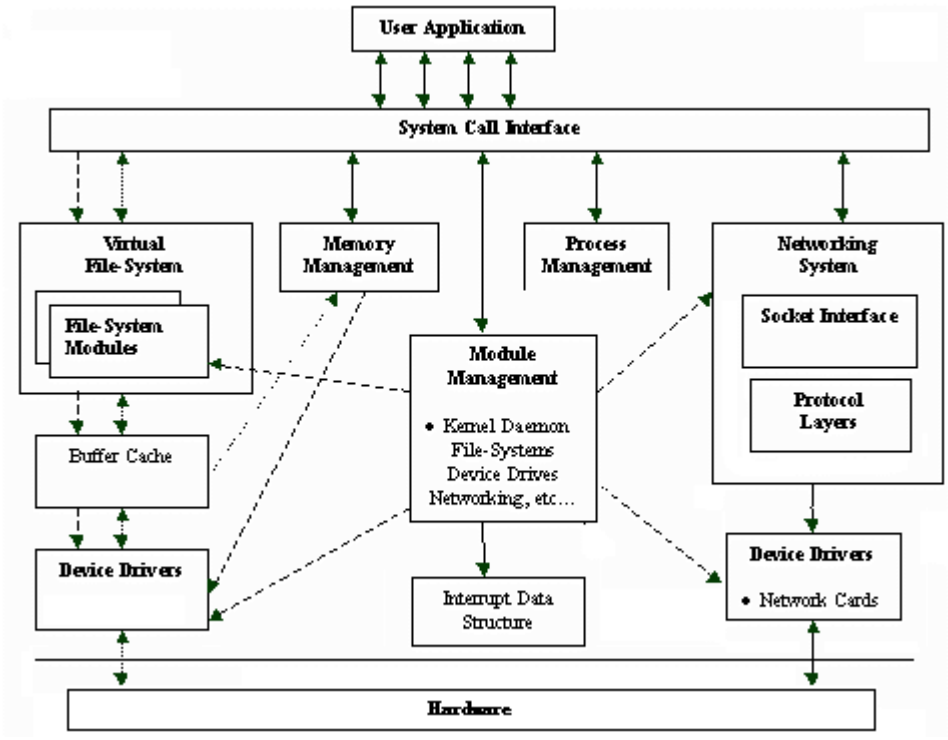


Figure 2.1 Architecture of the Standard Linux

Linux partitions the physical memory into user space and kernel space. In the kernel code, all the kernel code is running under protected mode. In the user space, Linux can manage multiple user programs running simultaneously in user space as a multi-tasking OS. The user space interacts with the kernel space using system calls, which can access the system resources such as hard disk, parallel ports and other peripheral equipments. One unique feature of Linux is that all the physical resources appear to work as files and to be easily controlled with system calls. All the input/output controls are fulfilled in the kernel space so that the application programs need not to be concerned with the details of sharing physical resources. The kernel and the device drivers manage the resources together.

Linux supports a mechanism of loadable modules to extend the kernel functionalities. These loadable kernel modules can be loaded into or removed from the kernel space by commands in the user space. One major advantage of this mechanism is developers of

kernel drivers do not need to repeatedly reboot the computer and reload the complete kernel to test the kernel's modification. Another advantage is that the memory space used by the kernel is reduced.

A general Operating System (OS) provides a good wealth of common services beyond the above components. Therefore the distribution of Linux also presents some utilities such as file browsers, editors, compilers, e-mail service and disk management.

2.2.2 Functions of Linux Kernel

The Linux kernel consists of several high level components. These components are responsible for the following: process, memory and module management, hardware device drivers, file-system drivers and management, network management, and various other tasks. Figure 2.1 gives an overview of the system. Components such as the Virtual File-System (VFS) and the Networking system are composed of a layered structure. These components will be introduced briefly.

2.2.2.1 Memory Management

Memory management is the mechanism to allocate memory requested by a process and de-allocated memory when a process terminates. Another function is to ensure that memory previously allocated by some processes will not be corrupted and is available

until the process releases the memory. To do this, the Linux memory manager manages a number of tables that list existing pages of virtual memory.

Linux uses demand paging to load executable images into a processes virtual memory. In this demand-paging scheme, only the first part of an executable image is brought into physical memory. The rest of the image is left on the disk so that further execution of the process generates page faults and causes the kernel to regain control. After the page fault, the kernel uses a memory map to determine which parts of the image are to be brought into physical memory. Like most operating systems, the Linux kernel supports high-volume physical address and presents page table and address translation.

The memory management is responsible for the management of four main caches:

- The Page Cache is used to speed up access to images and data on disk.
- A Swap Cache saves the modified or dirty pages in the swap file.
- The Hardware Caches is to cache the translation of processor page table and other caching required by hardware.
- The Buffer Cache contains data buffers that are used by file system and block device drivers. The buffer updates kernel daemon that attempts to maintain the file systems Buffer Cache.

2.2.2.2 Process Management

The important responsibility of kernel code is the execution and scheduling of application programs. Linux processes exist in the style of classic UNIX processes.

Linux is a multiprocessing operating system and thus each process has a separate virtual address space and cannot interact with another process except through kernel-management mechanisms.

The execution of process need allocate many resources, such as CPU, memory and files. Linux applies some scheduling algorithms to fulfill the resource management. Scheduler will select the outstanding process to run. Linux uses a priority based scheduling algorithm to choose between the current processes in the system. When it chose a new process to run, it saves the state of the current process, and other context in the process's data structure (*task_struct*).

2.2.2.3 Hardware Abstraction layer

The Linux kernel abstracts the handling of physical devices to control hardware. Linux supports three types of device drivers: character, block and network. Device drivers take on a special role in the Linux kernel. They make a particular piece of hardware respond to a well-defined internal programming interface. With such a hardware abstraction layer that hides the details of how the device works, the kernel code and user applications are interacted according to a set of standardized calls being independent of the drivers. Mapping those calls to specific operations that control on real hardware is also the role of device drivers. The hardware abstraction layer makes it possible that device drivers can be built separately from kernel, and plugged in at runtime. This modularity feature makes Linux drivers easy to write, to the point that there are now hundreds of them available.

2.2.2.4 File Management

Linux provides the persistent storage of information created in memory by the execution of a process. The information is typically stored as files on the file systems. Linux keeps track of the space on a disk to store the information that comprises a file. The user need not be aware of whether the file is stored in non-contiguous space or not.

Linux support a variety of other file system types such as ext2, ext3, swap, JFS, NFS and FAT etc. Linux uses a virtual file system (VFS), which abstracts the file system so that many heterogeneous file systems may be mounted by the system. Support for various file systems is achieved by loading modules that support the particular file system. Then all the file systems loaded are incorporated into the single VFS tree. The various file system modules communicate with the buffer cache that in turn communicates with the appropriate disk drivers.

2.2.2.5 Inter-Process Communication

Inter-process communication (IPC) is the communication between two processes. Linux supports it through three mechanisms:

- Messages: exchanges messages with any process or server.
- Semaphores: allows unrelated processes to synchronize execution.
- Shared memory: allows unrelated processes to share memory.

Once a share resource is created, access to it is assigned only when a permissions is set up. A resource consists of message queues, a semaphore set or a shared memory segment. A creator must first allocate the resource before it use the resource. The creator can assign a different owner. After use, the creator or owner must explicitly release the resources.

2.2.2.6 Network Management

Data communication between processes in Linux is commonly achieved with socket system calls. The idea of a socket is to plug-in to the other process and sends the message. The socket layer can be opened using a number of different protocols including TCP socket etc. For example, the SSL protocol, now known as the Transport Layer Security protocol, is the heart of efficient transactions on the Internet.

2.2.3 Time-Sharing Feature of Linux

As an Unix-like system, Linux is originally designed as a timing-sharing OS [10, 11]. Timing-sharing feature means that each process is assigned a certain amount of time quantum. This important feature allows multiple processes appear to execute simultaneously in kernel space.

As a timing-sharing OS, Linux is concerned with the maximizing CPU utilization and overall throughput, and minimizing the scheduling overhead, waiting time and

response time. Linux uses some algorithms of conventional timing-sharing scheduling. These algorithms are introduced briefly in section 3.3.

2.3 Linux Real-time Add-on Options

There have been some attempts to improve the response accuracy or make Linux real-time. In this section, we will introduce some Linux patches and three hybrid real-time Linux systems: Real-time Linux (RTLinux) [12], KU Real-Time (KURT) Linux [13] and Linux RK [14].

2.3.1 Preemptive Patch and LPP patch

Some researchers have worked on converting Linux into a preemptive system and reducing the response latency. MontaVista developed the preemptive patch to establish a preemptive Linux kernel based on Linux 2.4 and X 86 platforms [25]. This patch can set up and release the *spinlock*, so that task pre-emption is supported. In this modified preemptive kernel, if a process enters TASK_RUNNING state, the kernel checks whether its priority is greater than that of a currently running process. If so, the execution of the current task is pre-empted and the kernel scheduler is invoked to select an eligible process. Another way to promote a process's timing response was shown by Ingo Molnar. He developed the Low Latency Patch (LPP) of Linux specified for Multimedia Applications [27]. There are some sources of long latency such as calls to disk buffer cache and creation of processes. In order to reduce these latencies, some

pre-emption points are inserted into the system calls. This patch has been ported on i386 platforms and has been proved to reduce the response time of long latency events. The response latency of a system built separately with a preemptive patch, a LLP patch and their combination are investigated by Clark Williams [51]. He discovered that a system applying two patches showed a better performance than a system applying only one single patch.

2.3.2 Real-time Linux (RTLinux)

Real-time Linux (RTLinux) is an implementation of real-time task management using loadable kernel module [11]. RTLinux puts an emulation layer between the Linux kernel and interrupt controller. The emulator catches all hardware interrupts. In RTLinux, the non-real-time process in the standard Linux holds the lowest priority compared with real-time process. This approach converts the standard Linux kernel into a predictable kernel and realizes real-time scheduling in the loadable module. However, its drawback is the execution is non preemptive.

2.3.3 KU Real-Time (KURT)

KU Real-Time (KURT) is an extension of Linux built with a high-resolution timer and an event-driven task scheduling [12]. KURT can schedule the events at a resolution of microseconds and has a separate scheduling interface specified for real-time tasks. It provides a mechanism for transiting from the standard Linux scheduling to several

scheduling paradigms. KURT Linux supports conventional tasks as well as real-time tasks, while its performance of non-real-time task scheduling is not improved.

Both RTLinux and KURT use the loadable kernel module that has independent scheduling. The scheduling may be in three modes: real-time, non-real-time and hybrid mode, thus there can be frequent switches of modes that cause a long latency and introduce some unpredictability [11, 12]. Thus a system with a "dual-kernel" is easy to be broken and its response latency is increased. Besides, the safety, security and response time of system are more dependent on the applications.

2.3.4 Linux RK

Linux RK is an implementation of resource kernel developed by Oikawa and Rajkumar [13]. The most important feature of resource kernel is CPU reservation and there are some abstractions in it: *reserve* and *reserve set, where reserve* represents a shared resource and *reserve set* represents a group of *reserve*. When a process allocates a *reserve set*, it obtains a guaranteed execution corresponding to this *reserve set*. When an application works on a kernel of Linux RK, it can reserve a certain amount resources and the kernel can guarantee that the reservation of resource is safe. However, this kernel is treated as an enhancement of resource management rather than a real-time solution and its kernel is not preemptive.

2.3.5 Current Challenges

From the introduction of hybrid Linux systems, we can see that the attempts to enhance real-time property of Linux have some drawbacks more or less. The studies of

MontaVista and Ingo Molnar can convert Linux into a preemptive system and reduce the response latency. However, the two patches are ported onto scarce platforms such as X86 machines and cannot realize the real-time task scheduling. In the hybrid real-time Linux systems, Linux RK cannot present the real-time task scheduling. RTLinux and KURT realize the real-time task scheduling using a dual kernel. However, their kernels are non-preemptive and they only present fixed task scheduling that is not adaptive to various real-time applications. Although studies have proposed some approaches to improve the timing response accuracy, none has considered the potential of combining of pre-emption facility and the real-time task scheduling for deploying the real-time tasks effectively.

CHAPTER 3
REAL-TIME AND LINUX SCHEDULING

The role of scheduling algorithm is to provide the disciplines of the real-time task execution for the scheduling in an operating system. In this chapter, a survey on the real-time scheduling concepts and algorithms is conducted. In addition, the process management and scheduling in Linux are introduced in some details.

3.1 Survey on Real-time Scheduling

The real-time scheduling paradigm targets on satisfy the timing requirement of processes by coordinating the resources. The real-time scheduling algorithms can be categorized into cyclic executive and aperiodic executive. Many real-time systems use a simple cyclic executive for scheduling. Under this scheduling paradigm, the periodic tasks are executed in an order defined during the system design phase. Its major advantage is its low scheduling overhead and predictability, while its major drawback is its inflexibility. However, there are many situations where the tasks have irregular arrival times that are unpredictable. Presently there are various strategies for managing these aperiodic executives.

This section introduces certain concepts and algorithms of real-time scheduling. Section 3.1.1 describes three priority-driven scheduling algorithms for periodic tasks. Section 3.1.2 covers the scheduling paradigms specified for aperiodic tasks.

3.1.1 Cyclic Executive

Liu and Layland explored the priority-driven algorithms [16-18] that soon became the basis for the research in the field of real-time. This section illustrates three priority-driven scheduling algorithms to schedule periodic tasks: Rate Monotonic (RM), Earliest Deadline First (EDF) and Least Laxity First (LLF).

3.1.1.1 Rate Monotonic Algorithm

The most commonly used priority-driven scheduling algorithms are Rate Monotonic (RM) & Earliest Deadline First (EDF) [16]. The RM algorithm places a static priority based on the tasks' periods. A task with a shorter period is assigned the higher priority. The scheduling scheme is an optimal static-priority scheme. Static priority is attractive because a task's priority is assigned once it arrives and does not have to be inspected with it being aged if its period kept unmodified. The RM algorithm is applicable to periodic tasks only. Its bound of processor's utilization is always less than 1.

The strength of Rate Monotonic (RM) scheduling is that it is a static task scheduling. That is, when its priority is given with the execution rate, the task can be scheduled

with its priority. Its strength is the static scheduling decreases the complexity of task scheduling. The weakness is that the processor could not be fully utilized when the schedulability condition is satisfied.

3.1.1.2 Earliest Deadline First Algorithm

The Earliest Deadline First (EDF) [16] is a good algorithm that implements real-time scheduling by examining the deadline of each task. The task set is ordered by the deadline. The task with the closest deadline and an unfulfilled request is assigned with the processor. A task is thrown away if it does not complete its execution before its deadline expired. If it is waiting for some resources when its deadline is expired, the task is exited; if it is in service when its deadline is expired, it is aborted and then exit. In both cases, the task is lost in the system. Particularly as a dynamic priority assignment, EDF is applicable to both periodic tasks and aperiodic tasks. EDF algorithm is prosperous in minimizing the fraction of tasks that miss their deadlines. In every time interval, the deadline of each task is flashed and examined. The task's deadline cannot be expired before the context switching.

The strength of this algorithm is that it can make fully use of the processor resources with satisfaction of schedulability. Compared with RM algorithm, the priorities of tasks in EDF algorithm change with time. Thus this dynamic scheduling scheme brings a drawback: the higher scheduling and transient overhead in comparison with the RM algorithm.

3.1.1.3 Minimum Laxity First Algorithm

The Minimum Laxity First (MLF) algorithm is also a dynamic priority scheduling. Its model can be found in [19, 20]. In this algorithm, the task set is sorted according to the laxity times. The laxity is the remaining time of each task to complete in the current period. The highest priority is assigned to the task having the least laxity. In case of EDF, the dynamic priority is based on the deadline, where the closest deadline has the highest priority. While the MLF scheduler executing at every instance selects the tasks with the least laxity to be assigned with the resource and run on the processor.

As a dynamic tasks scheduling, MLF scheduling algorithm has the strength in making fully use of the processor resources with satisfaction of schedulability. Compared with EDF, one more strength of MLF algorithm is it takes the service demand into consideration to make the scheduling more reliable to the prediction of execution time. The drawback of MLF scheduling is the complexity of task scheduling increased.

3.1.2 Scheduling of Aperiodic Tasks

Under the situations that the arrival time of the task is irregular, the dynamic scheduling algorithms manifest unpredictability. Therefore several aperiodic executives are adapted to schedule the aperiodic tasks [18, 21, and 22]. Various algorithms dealing with such tasks are described here.

3.1.2.1 Polling Server Algorithm

Polling Server algorithm is the simplest executive that sporadic task can be dealt with by a periodic scheduling algorithm. Under this algorithm, a periodic task, named as polling server (PS), is run with relative higher priority to service on the aperiodic tasks. At each instance when it executes, the aperiodic task is executed during the PS's execution period. The server is preempted by higher-priority process until the time slice is used up or there are not aperiodic tasks existing. This algorithm is non-ideal and wasteful in that an aperiodic task arriving just after the polling server finished the period has to wait for the server's next period.

3.1.2.2 Deferrable Server (DS) Algorithm

Lehoczky, Sha, and Strosnider propose the Deferrable Server algorithm [22] that can deal with sporadic tasks. This approach guarantees timing critical for aperiodic tasks within the static priority policy. The DS algorithm is similar to a polling server in allotting a periodic server tasks for servicing aperiodic requests. However, unlike polling, the deferrable server preserves its execution time allocated for aperiodic service even when no aperiodic tasks are existed. The deferrable server task does not give up the timing slice until the period is exhausted; and new timing slice is assigned to the server in the next period.

3.1.2.3 Sporadic Server Algorithm

Lehoczky, Sha and Strosnider also proposed another algorithm called the Sporadic Server (SS) algorithm [22]. In this algorithm, a high-priority task to serve aperiodic tasks is called Sporadic Server (SS). When a sporadic task reaches the system, the sporadic server assigns the time slice to the sporadic task. The time slice is pre-allocated to the sporadic server. When the execution time is exhausted, the sporadic task stops to continue executing; thus other real-time task can achieve timing critical. The sporadic server algorithm controls the aperiodic tasks with two important attributes: execution budget and replenish period. The sporadic server's execution capacity is replenished periodically. The SS algorithm is used for dynamic priority scheduling as well as fixed priority scheduling.

Compared with DS algorithm, one difference is that the sporadic server can preserve its unused high-priority execution time indefinitely. Another difference is that the replenishment of execution time used by the SS is scheduled in a way that execution is deployed more evenly, and is not always at the beginning of a period as DS algorithm. In this respect, the SS is an improvement over the DS.

3.2 Process Model

3.2.1 Cyclic Process Model

There is two district forms of processes that are isolated from each other: periodic tasks and aperiodic tasks. Periodic tasks, as the name implies, task whose invocation is triggered within a regular time interval. Aperiodic tasks are the tasks that are triggered randomly. We first define and analyze a process model made up of only n periodic tasks. We make some assumptions and constraints to simplify this process model. In the synchronous process model, all the aperiodic tasks are created at the same time. Assuming the execution of a task does not depend upon the execution of other tasks, a synchronous process model is defined. Liu and Layland [16] presented the cyclic process model made up of n purely periodic tasks. These timing attributes are illustrated graphically in Figure 3.1.

$$\prod = \{\tau_i \mid \tau_i = (r_0, C_i, T_i, D_i), 1 \le i \le n, 0 \le C_i \le D_i \le T_i\}$$

Where, r_0*, task trigger time.*

C, task worst-case execution time.
T, task period, time of a cycle that a periodic task is activated.
D, task relative deadline.

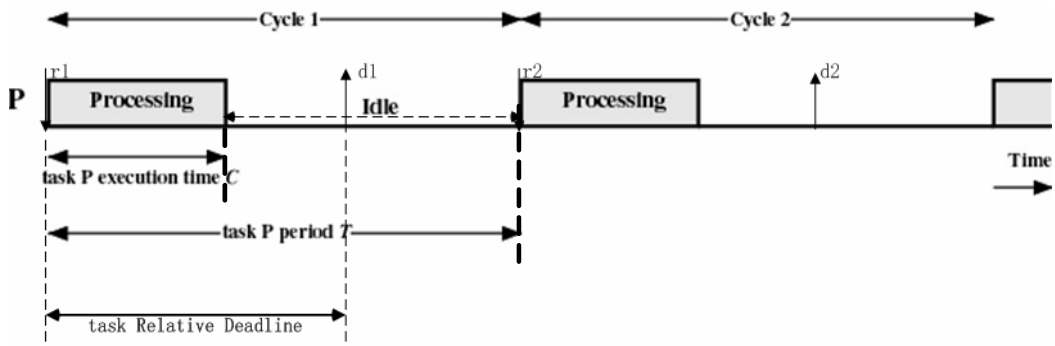


Figure 3.1 Process Model of Periodic Tasks[1]

[1] *In figure 3.1, d_1 is the absolute deadline of frist invocation (cycle 1) of the periodic task; d_2 d1 is the absolute deadline of second invocation (cycle 2) of the periodic task; meanwhile r_1 is the the task*

We make some simplistic constraints upon this process model as follows:

1. $C_i \leq D_i = T_i$, i.e. for a given task τ_i, its deadline is set to equal to its period.
2. Computation times for a give task are constant.
3. All processes are periodic; that is, period of a given task are constant.
4. All the processes are independent.
5. All the processes are executed on a single processor.

The tasks for which the scheduling paradigm and response time are repeated cyclically are closed. We call a system of such a process model *closed system.*

Closed Task is a task τ_i, *whose response time at* k^{th} *invocation is denoted as* $R_i(k)$. *Task* τ_i *is closed with interval of* X_i *cycles if*

$\forall i,\ 0 \leq i \leq n,\ \forall k \geq 1,\ \forall r \geq 0,\ R_i(k) = R_i(k + r \bullet X_i)$
Where $X_i = LCM^2(T_1, T_2, \ldots\ldots, T_n)/T_i$.

DEFINITION 1 (I*(w)*). *The idle time of the task execution in a given task set within the interval [0,w].*

DEFINITION 2 ($R_i(k)$). *The response time of task* τ_i *at* k^{th} *invocation. It is given by*

$$\forall i, 0 \leq i \leq n, k \geq 1,\ R_i(k) = F_i(k) - S_i(k) \qquad (3.1)$$

Where, $F_i(k)$ is the finalization instant of task τ_i at k^{th} invocation; $S_i(k))$ is the trigger instant of task τ_i at k^{th} invocation and is given by:

$\forall i, 0 \leq i \leq n, k \geq 1,\ S_i(k) = r_{0,i} + (k-1) \cdot T_i$

---

activation time of frist invocation (cycle 1) of the periodic task; r_2 *is the task activation time of second invocation (cycle 2) of the periodic task.*

3.2.2 Schedulability in Cyclic Process Model

In this section we analyze the schedulability condition and response time in the static and dynamic task scheduling in the cyclic process model. To simplify the problem, we can study the schedulability condition of cyclic process model. There is some research work that studied the schedulability of cyclic execution. Liu and Layland [16] present the theorem that offers a necessary and sufficient condition of the rate-monotonic scheduling. Since C_i/T_i is the fraction of CPU time spent in the task τ_i, the total CPU utilization of n tasks is:

$$U = \sum_{i=1}^{n} (C_i / T_i)$$

Theorem 3.1 A set of *n* periodic tasks is schedulable by the rate-monotonic algorithm if where

$$U \le n(2^{1/n} - 1) \qquad (3.2)$$

This theorem offers a sufficient condition of the rate-monotonic schedulability of a given periodic task set. Above theorem was proved by Liu and Layland [16]. This schedulable bound decreases monotonically from 0.83 with *n=2* to 0.693 with *n* reaching infinity.

The necessary and sufficient condition for the feasibility of a task set with EDF is given below:

$$\sum_{i=1}^{n} \left\lceil \frac{C_i}{T_i} \right\rceil \le 1$$

where, n is the total number of tasks.

---

[2] *LCM(T_1, T_2, ……,T_n) is the shorthand for the function of Least Common Multiple of T_1, T_2, ……,T_n.*

3.3 Process Management in Linux

The most central concept in Linux is the process: an abstraction of a running program. As we introduce in chapter 2, Linux must make sure that the processes get all the resources they need on time. Then it has to keep trace of the various processes. The fundamental data in the kernel is task_struct that is abstracted to keep some information including specific registers and other context. This data structure includes some main functional fields as following [23]:

- ***State Information*:** four process states are used in Linux.
- **S*cheduling Information***: Used to determine which process in the system will be the next to run.
- ***Inter-Process Communication***: Linux supports classic Unix IPC (pipes and semaphores) and System V IPC.
- ***Links***: The family relationship between other processes (parent and child).
- ***Times and Timers***: Process creation time and time consumed by the process.
- ***Virtual Memory***: Information concerning the virtual memory mapped by the process.
- ***Context:*** the registers and stack allocated by the process and other state of the system.

Only a single process can be running at any given instance. The scheduler is responsible for selecting which process to be run next and switching the CPU from one

process to another. Thus the scheduler can be called under two main conditions: one is the process return from the system call; while another condition is after the time quantum of a process is used up. The scheduler requires that the information of the out-going process must be saved. This entire procedure is called a context switch.

The consistency of above processes' information is achieved using mutual exclusion and synchronization. There are a number of synchronous primitives [23]:

- ***Mutual Exclusion***: objects that ensure that only a single process has access to a shared resource at any time.
- ***Semaphores***: similar to mutual exclusions but may include counters allowing only a certain amount of threads access a shared resource at any one time
- ***Atomic operations***: This mechanism ensures that an atomic transaction is completed by a process before access the atomic operations to a shared resource. The process entering an atomic operation maybe have uninterruptible access to the CPU until the operation is completed.

Processes can be in one of four states: Running (combining ready and running), Waiting (interruptible and uninterruptible), Stopped and Zombie. The running state means that the process has all the resources it need for execution or it has been given permission by the operating system to use the processor. Only one process can be in the running state at any time instance. The remaining processes are either in a waiting state (waiting for some external event or some shared resources) or a ready state (waiting for permission to allocate the processor). Linux process scheduler makes the processes transit among the four states. And it implements the following functions:

- To allow processes to create new copies of themselves
- To determine which process is eligible to allocate the CPU and switch between the original running process and idle process
- To receive interrupts and route them to the associated kernel subsystem
- To terminate process and release the resources when a process is ZOMBIE.
- To provide support for loadable kernel modules(LKMs)

3.4 Scheduling Paradigms in Linux

Linux actually unifies all three terms: task, process and thread into process in the kernel. When a new process is created, various scheduling policy and the associated parameters are assigned. Currently, the following three scheduling policies are supported under Linux: SCHED_FIFO, SCHED_RR and SCHED_OTHER [23, 24]. SCHED_FIFO and SCHED_RR are applicable for special time-critical applications that need precise control over the way in which outstanding process is selected for execution. Processes scheduled with SCHED_OTHER must be assigned the static priority 0, processes scheduled under SCHED_FIFO or SCHED_RR can have a static priority in the range 1 to 99. Only processes with super-user privileges can get a static priority higher than 0 and can be scheduled under SCHED_FIFO or SCHED_RR. Three scheduling algorithms correlated with respective semantics are described in the following sub-sections.

3.4.1 Multi-Processes Scheduling

Multi-Processes Scheduling in Linux is called as “Timing-sharing” Scheduling. The scheduling policy SCHED_OTHER applies the default universal time-sharing scheduler paradigm used by most processes. Under the scheduling scheme, the processes are scheduled by examining “dynamic priority”. First of all, each created process is assigned with a certain time quantum that decreases with its age, and the process’ priority is defined as its remaining time quantum. If a process’s quantum expires before its termination, it is switched to the next eligible process in the ready queue. Moreover, a process’s time quantum is only renewed after all the other processes in the ready queue have expired. The important role of scheduler is to select the process with highest priority from the ready queue. And if an executing process need wait for some shared resources, the scheduler will block the process temporarily and change its state to STOPPED (interruptible/uninterruptible). Once the resources the process need is available, the scheduler will wake up the process and insert it into ready queue.

3.4.2 FCFS Scheduling

The scheduling policy SCHED_FIFO adopts First Come First Served scheduling (FCFS) algorithm [23]. Under FCFS scheduling, the created processes are inserted into a first in first out (FIFO) queue. Under this scheduling paradigm, the process with a highest priority will continue to execute till it either terminates or blocks itself.

3.4.3 Round-Robin Scheduling

The scheduling policy SCHED_RR represents Round Robin scheduling algorithm. For the processes using Round Robin, a newly created process is always added to the end of ready queue. A new process will preempt the current process after running for some time slices and be moved to the end of ready queue. This is a fair assignment of CPU time and shared resource among all the SCHED_RR processes.

3.4.4 Summary

The three scheduling paradigms in Linux have been introduced in above sections. From the depiction of FIFS scheduling in standard Linux system, we could find that FIFS is not suitable for the realistic requirement of timing-critical applications. If a real-time application deployed by FIFS scheduling discipline has a long execution time, it may preempt all the other tasks, which may cause other task to be delayed for a long time.

CHAPTER 4
SYSTEM DESIGN

Real-Time Supported Linux is designed to enhance the real-time property of standard Linux kernel. This chapter is an overview of the system designation of RTS-Linux. Section 4.1 introduces the requirement and assumption of the designation. In section 4.2, the two approaches applied on the real-time executives are discussed. Section 4.3 describes the basic concepts and services of RTS-Linux

4.1 Requirement and Assumption

Since Linux is a multi-process operating system, many of its components do not perform well under real-time constraints. In RTS, the real-time events are predictable behaviors under all circumstances of system load. The requirements of "RTS-Linux" architecture are summarized as follows.

- The existing open-source application running on the standard Linux can be reused on RTS-Linux.
- The real-time property is pre-built and optionally configured. Real-time events are predictable behaviors under all circumstances of system load.
- The required modification on the standard Linux kernel should be minimized.
- An effective user interface design.

The last two requirements are important for the embedded system. They are also helpful to port the real-time extension to more platforms and advanced Linux.

When designing the “RTS-Linux”, in addition to assuming that the real-time task running on RTS-Linux has a privilege level over the other processes, we also assume that all the real-time tasks are constrained by their timing attributes.

4.2 Description of Two Approaches

In order to enhance Linux performance, there are some approaches to improve real-time performance. These approaches can be briefly categorized as building a preemptive kernel and using dual-kernel.

The first approach is to produce a pre-emptive Linux kernel with a real-time scheduler [25]. This allows the kernel to be preempted at any time when it is not locked. Using this model, when a task with the higher priority is ready to be executed, the system will preempt the current executing task and run the higher priority task.

The advantage of this approach is that real-time programming can be performed at the user application level and all standard Linux services are available to the application. The memory protection is also available. The disadvantage is that the worst-case response latency is unknown and overall system throughput is reduced.

Another common-used approach is known as dual-kernel approach. The real-time kernel inserts a very thin layer between the interrupt-control hardware and the standard Linux kernel, and a task will be run under the real time kernel [12, 26]. When Linux issues a request to enable or disable an interrupt, the real-time kernel receives the request first and thus controls all interrupt vectors and scheduling. Instead of dealing with actual interrupt-control hardware, the real-time kernel writes the request into an internal data structure and returns control to standard Linux. Using this approach, standard Linux is completely isolated from the interrupt-control hardware. Instead, the real-time kernel emulates that particular hardware with a virtual machine layer.

In the dual-kernel approach, the standard processes hold the lowest priority task compared with the real-time processes. The advantage is that the real-time kernel works as a real-time OS that can suspend the execution of Linux processes at any state. It does not take care of what Linux is doing the moment an interrupt arrives; it immediately switches context and passes control to a real-time task very quickly. A deterministic scheduling can also be achieved. The disadvantage of dual-kernel approach is that it increases the overhead of context switch and loadable module. The developers must take care of the real-time abstraction layer and know how the kernel works clearly. Finally, any improper modification of kernel will make the system crash, which would make the debugging quite difficult.

4.3 RTS-Linux Design

Before introducing the system designation of RTS-Linux, we intend to inspect the effects of tasks scheduling in alternative address space. The processes can be classified into kernel space processes and user space processes according the space where they are located. Similarly, the real-time execution could be run externally or internally to the kernel. Some kinds of real-time execution are run in the kernel space. For the scheduling scheme insides Linux Kernel, all the real-time processes are pre-initialized in kernel space. The advantage of this internal execution is to allow real-time processes to communicate directly with non-real-time processes without increasing the scheduling overhead. An alternative execution is external. The real-time processes can be created and deleted by the user applications. The real-time execution is run in kernel space as a loadable module. The advantage is this execution brings more flexibility to the scheduling paradigm. Meanwhile the version update of Linux is more convenient. Its advantage is the schedule overhead of real-time processes is increased as these processes have to switch between the kernel and the modules. Considering the fast optimization and version upgrading of Linux, we propose to adopt the second execution.

In the RTS-Linux, the RTS driver coexists with the standard Linux kernel. The modification is mostly contained in RTS driver. Thus it would be easy to port RTS extension on more hardware platforms and new Linux version. RTS driver is a virtual layer including the data abstraction of tasks and the control operations on the tasks. It also provides the APIs to interact with the kernel and control the real-time executive. We decompose the Linux system according to the conceptual dependencies of the

subsystem. Figure 4.1 shows the architecture of the RTS-Linux kernel. Linux kernel is composed of five major subsystems: the scheduler, the memory manager, the file system, the network interface and the inter-process communication. The real-time scheduling is the role of a virtual device driver-RTS driver.

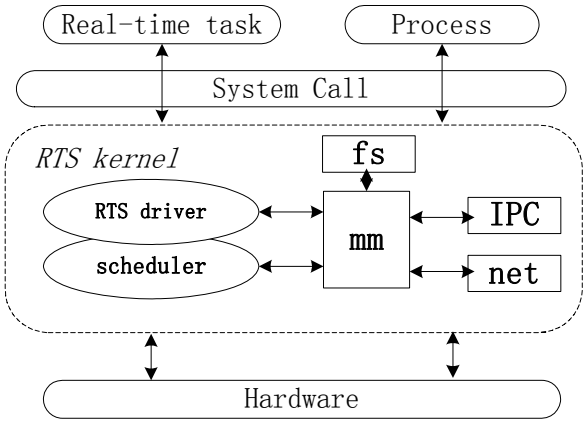


Figure 4.1 Block Diagram of RTS-Linux[3]

In the RTS-Linux, all real-time processes have access to all the services and APIs presented by the standard Linux. Figure 4.2 illustrates the relationship of APIs shared by real-time tasks and the standard Linux kernel. Standard Linux kernel and real-time subsystem sharing the main primitives such as synchronous and asynchronous I/O control, mutex mechanism, interrupt handling and timer/clock mechanism. The asynchronous I/O control may be used in soft-real-time control. In hard real-time control, synchronous I/O control or Directly Memory Access (DMA) is more suitable to its timing requirements.

[3] In this figure, net represents the network management function in the kernel; mm represents the memory management function in the kernel; fs is the file system management in the kernel; IPC is inter-process communication function n the kernel.

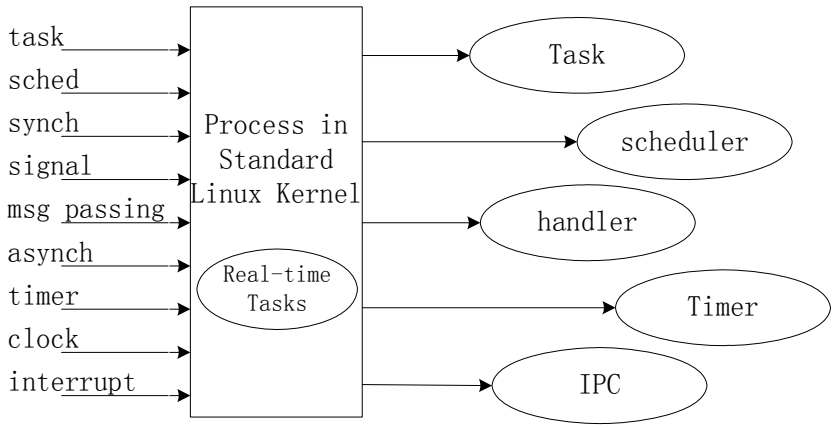


Figure 4.2 Shared APIs and IPC between two parts of RTS-Linux

The RTS driver holds a scheduler to schedule multiple real-time tasks using multiple flexible scheduling policies. The main role of real-time scheduler is to deploy the task execution of real-time tasks and make this execution to meet the real-time requirements. There are many ways to manifest the timing constrains and scheduling policies. By default, RTS-Linux provides a priority-driven scheduler where each task is assigned a specified priority. The scheduler chooses an eligible task from all the tasks that are ready to execute by examining their priorities. If a task becomes ready and has a higher priority than the executing task, it will preempt the executing task to shorten the response time. The scheduler supports aperiodic tasks as well as periodic tasks. For a periodic task, its period and offset (starting time) are specified. In order to improve the scheduler, a set of new features is optional to be inserted. For example, no single policy is appropriate for all applications. RTS-Linux allows developers to write their own scheduler.

The tasks queue keeps track of the status of multiple tasks. Tasks having the same status are queued into the correlated list. When a scheduler plans, the first task in the

list will be taken. The queue management also provides the necessary insert and removal primitives.

In this chapter, the approach and system design of RTS-Linux is introduced in short. In addition, the system architecture is introduced. The main role of scheduling algorithm is to define the disciplines of the real-time task execution. In chapter 5, the process model and response time in real-time scheduling will be investigated. In chapter 6, the implementation the real-time scheduling will be introduced in some detail.

CHAPTER 5
HYBRID PROCESS MODEL AND RESPONSE TIME

Many literatures deal with the problems in the field of real-time execution and task scheduling that include periodic deterministic tasks and acyclic task executions [16, 20 and 31]. The task execution to meet all their deadlines is determined by the scheduling algorithms. Scheduling algorithms can be characterized into two approaches. One is static scheduling such as Rate Monotonic (RM) [31]. Another approach is dynamic scheduling such as Earliest Deadline First (EDF) [31] and Minimum Laxity First (MLF) [20]. The schedulability analysis and feasibility test of tasks set are presented by Liu and Layland [16]. Mok and Chen et. proposed the Multiframe Model for Real-Time Tasks [40-42]. Thomadakis analyzed the response time and simulation of task execution in the synchronous system [43]. Some study of the response time is about the real-time control in networking communication [53-55, 57], particularly the fault-tolerant and failure. However, a lot previous research work of task scheduling draws attention on the analysis of cyclic process model. The analysis of the integration of cyclic and acyclic execution is lacked. Our research work construct a hybrid process model and analyze the schedulability in this process model.

On the other hand, Bernat analyzed the response time of asynchronous system by deriving its upper and lower bound in Bernat 2003 [50]. However, the computation of response time is quite complex. An approach of worst-case response time is proposed to decrease the computation complexity.

Based on the cyclic process model shown section 3, we present a new process model composed of real-time periodic tasks and aperiodic tasks in the chapter. Considering the task schedulability of aperiodic tasks, a new scheduling algorithm is also presented. In this flexible sporadic scheduling (FSS) algorithm, a server with variable period and execution budget is used to deploy the aperiodic real-time tasks.

The rest of the chapter is organized as follows. The new process model is described in section 5.1. The computation of worst-case response time prediction and a new approach of its prediction are introduced in section 5.2 and 5.3. The schedulability of hybrid process model is analyzed in section 5.4. The of flexible sporadic server (FSS) algorithm for aperiodic tasks is introduced in section 5.5

5.1 Hybrid Process Model

The process model composed of purely periodic tasks has been described in section 3.2. The realistic real-time system is more complex than a cyclic process model. One realistic real-time system is composed of periodic tasks as well as aperiodic tasks. Therefore a hybrid process model can be defined as below.

DEFINITION 5.1. *A hybrid Process model is a union of cyclic task set* $\prod$ *and acyclic task set* $\Re$. *A hybrid process model* is composed of N a*periodic tasks characterized by their main timing attributes (trigger time, computation time and period)*

$\Re = \{\varsigma_l \mid \varsigma_l = (w_{0,l}, C_l^a, T_l^a, D_l^a), 1 \le l \le N, 0 \le C_l^a \le T_l^a\}$ Where,

- $w_{0,l}$, stands for task trigger time of aperiodic task ς_l.
- C_l^a, stands for execution budget of each invocation of aperiodic task.
- T_l^a, stands for inter-arrival time of aperiodic tasks.
- D_l^a, stands for the Service Demand of aperiodic tasks

The lifetime of one aperiodic task can be obtained by $\left\lceil D_l^a / C_l^a \right\rceil \cdot T_l^a$.

When the hybrid process model has *n* task corresponding to *n* task in cyclic process model with the constraints of $D_l^a = mC_l^a$ (T denotes the observation duration being long enough, $m = T / T_l^a$) and $w_{0,l} = r_{0,l}$, the hybrid process model can be treated solely as a cyclic process model defined in section 3.2.

$\prod = \{\tau_i \mid \tau_i = (r_{0,i}, C_i, T_i, D_i), 1 \le i \le n, 0 \le C_i \le D_i \le T_i\}$

Where attributes are same as those defined in section 3.2.

5.2 Computation of Worst Case Response Time that

Assuming there is one task set in cyclic process model being schedulable, the response time and worst-case response time (WCRT) prediction is introduced in this section. To

simplify the process model, we make this assumption to compute the response time: for a given task τ_i in the process model of synchronous system, it has the property: $r_{0,i} = 0$. In the simplified synchronous process model, we investigate the response time of static scheduling. Given a specific task in a task set, we consider a subset composed of the tasks whose priorities is not less than that of the specific task (including itself). In the cyclic process model, the elapsed time denotes the summation of request time and idle time. Therefore we investigate the function of idle time first.

5.2.1 Response Time of Static Scheduling

In order to compute the response, it is necessary to compute the idle time in a task set in synchronous cyclic process model. To compute the function of idle time, we construct a function which denotes the deviation between the elapsed time and request: $f(x) = x - Req(x)$.

The function of request is the summation of request that has been submitted within the interval [0, w], it is denoted as $Req(x)$.

$$\mathrm{Re}\, q(x) = \sum_{j=1}^{n} \mathrm{Re}\, q_j(x) = \sum_{j=1}^{n} \left[(\lfloor x / T_j \rfloor + 1) \cdot C_j \right] = \sum_{j=1}^{n} \sum_{m \in N, m=0}^{M_j} \left[U(x - mT_j) \cdot C_j \right] \qquad (5.1)$$

where $U(x)$ represents the unit step function, equal to 0 for $x < 0$ and 1 for $x \geq 0$, and $\lfloor x \rfloor$ represents the ceiling function. Given an observation duration T that is long enough to investigate the worst-case response time (WCRT), M_j represents the max invocation number of task j.

Thus the function of difference between elapsed time and request is given by:

$$f(x) = x - \sum_{j=1}^{n} \sum_{m \in N, m=0}^{M_j} \left[U(x - mT_j) \cdot C_j \right] \tag{5.2}$$

Bernat 2003 [50] has shown the distribution using some specific examples of process model and defined an upper and lower bound of the idle time present the formulation to compute the worst case response time. Our formulation *f(x)* inherits the lower bound function of idle time shown in Bernat 2003 [50]. Based on his work, we derive an approach to WCRT computation and get the WCRT prediction from the computation of idle time.

Based on the distribution of idle time, the derivative of the function *f(x)* is given by:

$$\frac{d}{dx} f(x) = 1 - \sum_{j=1}^{n} \sum_{m \in N, m=0}^{M_j} \left[\delta(x - mT_j) \cdot C_j \right] \tag{5.3}$$

where $\delta(x)$ is the delta function.

The delta function has the fundamental property that $x\delta'(x) = -\delta(x)$ [46] Thus we have the second derivative of function f(x):

$$\frac{d^2}{d^2 x} f(x) = \sum_{j=1}^{n} \sum_{m \in N, m=0}^{M_j} \frac{\delta(x - mT_j) \cdot C_j}{x - mT_j} \tag{5.4}$$

Compared with the function f(x) and function of idle time, the function of idle time $I_i(x)$ is monotonic non-decreasing function. It can be defined as

$$I_i(x) = I_i(t_i), \forall i \in N, i \geq 0, t_i \leq x < t_i';$$

$$I_i(x) = f(x), \forall i \in N, i \geq 0, t_i' \leq x < t_{i+1}; \quad (5.5)$$

where $t_0 = 0$,

$t_i = \min(y_0, ...y_k)$, where $y_m|_{0 \leq m \leq k} \in \{y \mid y > t_{i-1}', f''(y) < 0\}$.

$t_i' = \min_Y(y_0', ...y_k')$, where $y_m'|_{0 \leq m \leq k} \in \{y \mid f(y) = f(t_i), f''(y) \geq 0\}$

In the computation of idle time, each time slice t_i' gives the finish time *Fi(k)* that task τ_i at k^{th} invocation ($S_i(k) < t_i' \leq S_i(k+1)$). Therefore we can compute the response time according to equation 3.1. Thus worst-case response time (WCRT) can be obtained from the maximum value of the response time in each invocation.

5.2.2 Response Time of Dynamic Scheduling

Based on the previous investigation of response time of static scheduling, we conduct a theoretic analysis of response time in dynamic real-time task scheduling. Given a task set which is known to be feasible to schedule in cyclic process model, we now analyze the response time and worst case of task execution. In the simplified process model, the dynamic scheduling discipline is applied. Thus we need investigate the response

time of a specified task in the whole set of the real-time tasks. But whatever scheduling discipline is applied, we can get the same formulation of the idle time.

We define the function of request as $Req_i(x)$, which has been submitted within the interval [0, w] by the tasks whose priorities are not less than that of the specified real-time task τ_i.

$$\mathrm{Re}\, q_i(x) = \sum_{j=1}^{n} (\lfloor x / T_j \rfloor \cdot C_j) + \sum_{j=1, T_j \in hp}^{n} C_j \quad [4]$$

For example, given a simplified process model, with the deadline of the real-time task τ_i at k^{th} invocation $D_i(k) = S_i(k) + T_i$, if the real-time tasks are scheduled according to the earliest-deadline first (EDF) scheduling disciplines, the request time function can be given by

$$\mathrm{Re}\, q_i(x) = \sum_{j=1}^{n} \sum_{m \in N, m=0}^{M_j} \left[U(x - mT_j) \cdot C_j \right] - \sum_{j=1}^{n} \left\{ \left[U(x - (S_i(k) + T_i - T_j)) \right] \cdot C_j \right\} \qquad (5.6)$$

Similarly, the function of difference between elapsed time and request is given by:

$$f(x) = x - \mathrm{Re}\, q_i(x)$$

$$f(x) = x - \sum_{j=1}^{n} \sum_{m \in N, m=0}^{M_j} \left[U(x - mT_j) \cdot C_j \right] + \sum_{j=1}^{n} \left\{ \left[U(x - (S_i(k) + T_i - T_j)) \right] \cdot C_j \right\} \qquad (5.7)$$

The derivative of the function f(x) is given by:

[4] In this equation, *hp* represents the subset of tasks that hold the priorities that are not less than that of task τ_i

$$\frac{d}{dx}f(x) = 1 - \sum_{j=1}^{n}\sum_{m\in N, m=0}^{M_j}\left[\delta(x - mT_j)\cdot C_j\right] + \sum_{j=1}^{n}\left\{\left[\delta(x - (S_i(k) + T_i - T_j))\right]\cdot C_j\right\} \quad (5.8)$$

where $\delta(x)$ is the delta function.

The second derivative of function f(x) is

$$\frac{d^2}{d^2x}f(x) = \sum_{j=1}^{n}\sum_{m\in N, m=0}^{M_j}\frac{\delta(x - mT_j)\cdot C_j}{x - mT_j} + +\sum_{j=1}^{n}\frac{\delta(x - (S_i(k) + T_i - T_j))}{x - (S_i(k) + T_i - T_j)}\cdot C_j \quad (5.9)$$

Similarly in dynamic task scheduling, we can get the function of idle time $I_i(x)$ according to equation 5.5. In the computation of idle time, we can obtain the time slice t_i' that is the finish time $F_i(k)$ of task τ_i at k^{th} invocation ($S_i(k) < t_i' \le S_i(k+1)$). Therefore we can compute the response time according to equation 3.1. The WCRT can be given by the maximum value of the response time in each invocation.

5.2.3 Response Time of Asynchronous Process Model

In a similar way, we can extend the analysis of extension of synchronous model to asynchronous process model. The main difference between the synchronous model and asynchronous model is that not all the task is triggered at the same time instant. We consider a more complex process model that all the tasks are not created at $t = 0$. Therefore for a given task τ_j, it has a specified offset of $r_{0,j}$ time units.

We define n_j *(w)* as the number of invocations of task τ_j that lay completely inside the interval [0,w]. It is given by

$$n_j(w) = \lfloor w / T_j \rfloor$$

The release instant of its last invocation can be computed by $n_j(w) \cdot T_j$.

In asynchronous model, let $n_j(w)$ *function* ($\lfloor w / T_j \rfloor$ in synchronous model) change to:

$$n_j(w) = \lfloor (w - r_{0,i}) / T_j \rfloor + 1$$

5.3 Worst-Case Response Time Prediction and Computation

In this section, we verify the worst-case response time (WCRT) prediction using the approach in previous section.

In the first experiment, we compute the idle time and find the points where the idle time begins to increase. This time gives the finish time of current invocation, thus the response time is obtained. The maximum value of response time is WCRT prediction.

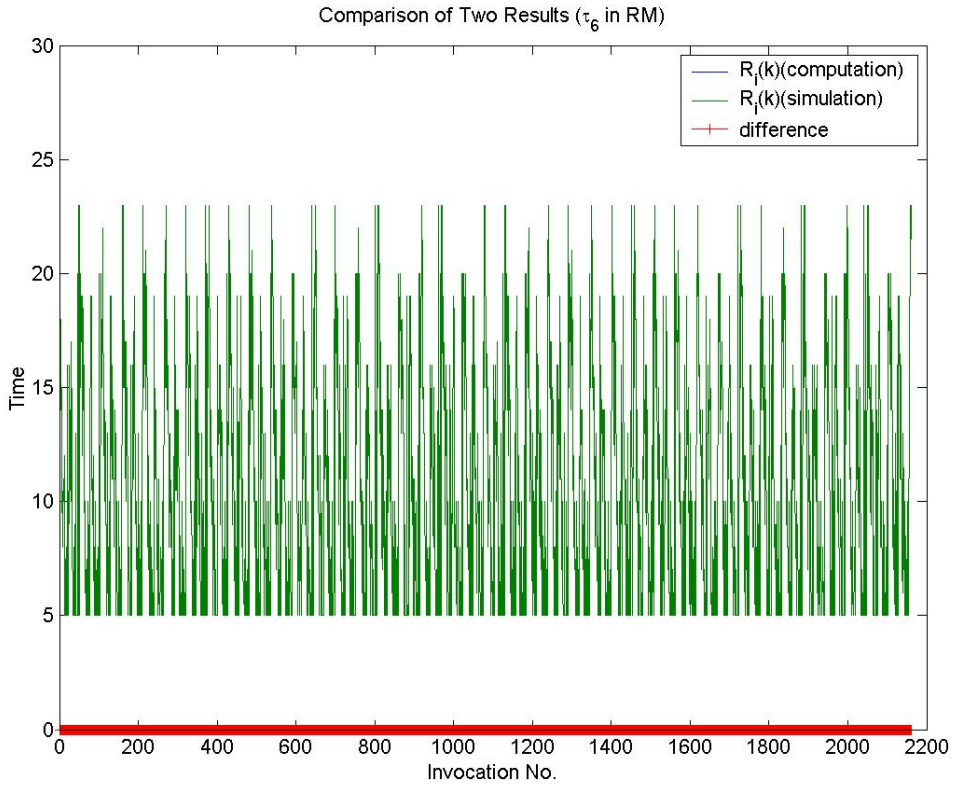
In the second experiment, we developed a program to simulate the task scheduling in some common-used algorithms. Each task is characterized with its timing attributes. When all tasks are initialized in the system, a scheduler deploys the tasks according to the scheduling policies. The periodic tasks are assigned with the priorities according to

the execution rate and distributed with their priorities. We measure the response time at every invocation and get the worst-case response time.

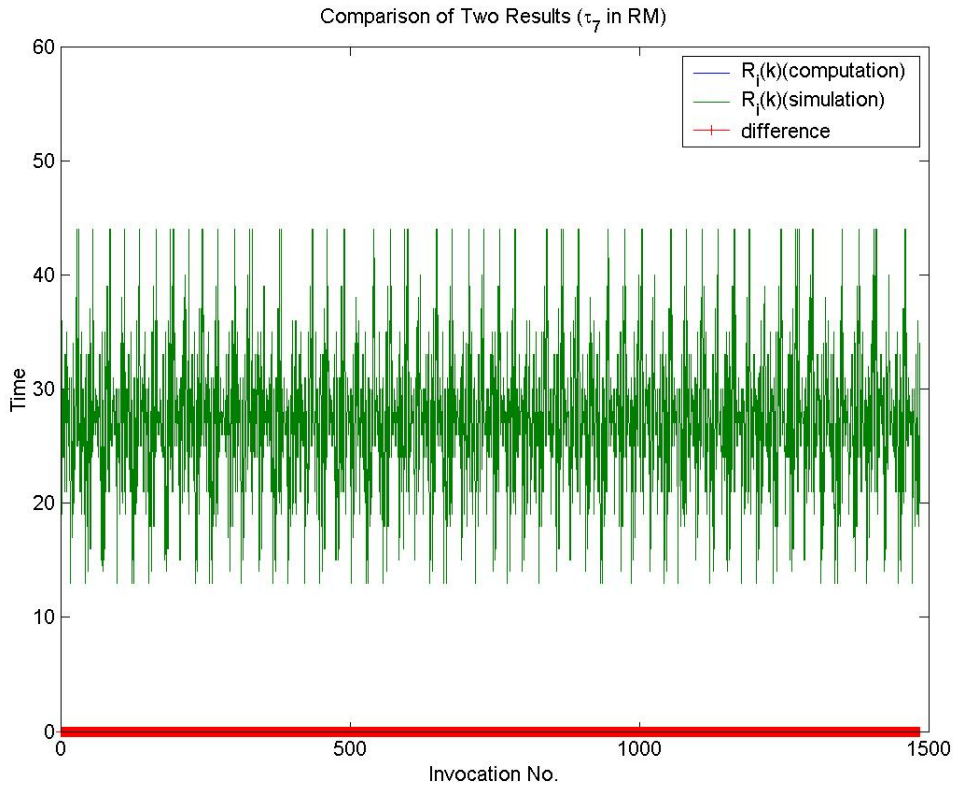
We compared the result of WCRT prediction and actual WCRT computation at process model. The timing attributes and the computation results such as worst response time are shown in Table 5.1. The fraction of processor usage of the task set is 0.727, while the schedulable bound of the task set ($n(2^{1/n}-1)$) is 0.729. This task set is schedulable under RM algorithm. Using the equation (3.1) and (5.5), we can compute the response time of every task at each point with idle time increasing. Figure 5.1 shows the response time of task τ_6 and task τ_7. In this figure, the blue line is response time obtained by simulation; while the green line is response time obtained in WCRT prediction; the red line is the deviation between these two value at identical time. The zero deviation shows that the response time using in WCRT prediction gives a no-bias prediction. In the closed process model, we can show all the possible response time in the certain cycles of task execution. According to the formulation of computing X_i, the computation of response time of task τ_6 and τ_7 within around 2200 invocations. The no-bias WCRT prediction proved that our prediction approach is functioning well.

Table 5.1 Example task set: time attributes and WCRT

τ_i	*Ci*	*Di*	*Ti*	*Ri*
τ_1	2	25	25	2
τ_2	6	27	27	8
τ_3	3	40	40	11
τ_4	3	50	50	14
τ_5	4	54	54	18
τ_6	5	55	55	23
τ_7	10	80	80	44



(a) task τ_6



(b) task τ_6

Figure 5.1 Response Time of the task τ_6 and τ_7 (task set in Table 5.1)

The experimental result shows worst-case response time may not take place at each invocation. The response time varies with the number of invocation and the relationship with other tasks. $R_i(1)$ always equals to the worst-case response time of task τ_i in synchronous cyclic process model. The distribution of the worst case response time interval when a task suffers is not even. This distribution depends on the execution time and task periods of higher-priority subset.

5.4 Schedulability of Hybrid Process model

We have proposed the hybrid process model in section 5.1. The schedulability is a condition that a task set can meet its real-time requirements in the process model. Our research about the schedulability of hybrid process model can be described as follows.

Theorem 5.1. If and only if a periodic task is schedulable in a task set, when the periodic task is replaced by multiple aperiodic tasks with same utilization fraction whose lifetime does not interact with each other, the new task set including the multiple aperiodic tasks are also schedulable.

Proof: Let Π (τ_1, τ_2,…, τ_s) be a set of s tasks including the periodic task τ_m. Let τ_m to be replaced with periodic tasks $\tau_{m,o}$ holding the timing attributes ($C_{0,m}$ $T_{0,m}$), $C_{0,m}/T_{0,m} = U_m$

and GCD[5] ($C_{0,m}$ $T_{0,m}$)=1. We construct a new task set Π′, as $C_{0,m}/T_{0,m} = U_m = C_m/T_m$, if task set Π is schedulable; the new task set Π′ is also schedulable.

Let τ_m be replaced with multiple aperiodic tasks that each aperiodic task $\tau_{m,i}'$ has a constant utilization faction U_m whose lifetime does not interact with each other and which is activated continuously. We construct a new task set Π′. In k^{th} invocations of τ_m, we have these multiple aperiodic tasks $\tau_{m,i}'(w_{m,i}'$ $C_{m,i}'$ $T_{m,i}'$ $C_{m,i}'$ where $w_{m,i}'$ is the activation time of aperiodic task. For each aperiodic task $\tau_{m,i}'$ ($1 \le i \le l$) with $n_{m,i}$ invocations and timing attributes ($n_{m,i}$, $C_{m,i}$ $T_{m,i}$) where $C_{m,i}/T_{m,i} = U_m$. Let us replace each aperiodic task $\tau_{m,i}'$ by $n_{m,i}'$ invocations of tasks $\tau_{0,m}'$ ($C_{0,m}$ $T_{0,m}$) as $n_{m,i}'=T_{m,i}/T_{0,m}$. We have

$$U' = U - C_m/T_m + \sum_l C_{m,i}/\sum_l T_{m,i}$$

$$= U - C_m/T_m + \sum_l n_{m,i}' \cdot C_{0,m}/\sum_l n_{m,i}' \cdot T_{0,m} = U$$

Therefore we draw a conclusion that if a task set is schedulable, and let one periodic task to be replaced with multiple aperiodic tasks with same processor utilization fraction, the new task set including these aperiodic tasks is also schedulable.

This provides a theoretical base for the flexible sporadic server algorithm.

[5] *GCD($C_{0,m}$ $T_{0,m}$) is the shorthand for the function of Greatest Common Divisor of $C_{0,m}$ $T_{0,m}$.*

5.5 Flexible Sporadic Server Algorithm

We proposed a flexible sporadic server (FSS) algorithm to deploy the aperiodic real-time tasks. The FSS server is characterized by a fixed utilization, variable service time and period. When an aperiodic task is activated in the system, it will be registered in the flexible sporadic server and its execution budget and period is also defined. Then the aperiodic task can be scheduled just as other real-time tasks. When one aperiodic task finishes execution in the current invocation of the FSS server, the task scheduler will transfer to task execution of other real-time tasks. When the aperiodic task finishes all execution, it will be unregistered in the flexible sporadic server and release the FSS server. One flexible sporadic server can serve only one aperiodic task at one time. The period of FSS server can be adjusted according to the actual service time of the aperiodic tasks and the utilization of FSS server.

For example, we consider a process model with static task scheduling. Assuming the aperiodic task has an identical priority, the period of the task is assigned with its priority. When an aperiodic task attempts to allocate the server, the period of the server is modified to the same as that of aperiodic task. As a flexible sporadic server have a utilization fraction, the execution budget of the server can be adjusted to the multiply of period and utilization.

Lemma 5.1. Given a task τ_m with utilization faction $U_m=C_m/T_m$ in a task set Π, if the task set is schedulable, let τ_m to be replaced with a flexible sporadic server τ_m' with a

constant utilization faction U_m, then the task set Π′ including the flexible sporadic server is also schedulable.

Proof: Let τ_1, τ_2,…, τ_s be a set of s tasks including the periodic task τ_m. Let τ_m to be replaced with a flexible sporadic server τ_m' with a constant utilization faction U_m. We assume a special case that the laxity between the invocations of flexible sporadic server is zero. This special case gives the maximum utilization of flexible sporadic server. The schedulability of this special case has been proved in Theorem 5.1. Therefore we conclude that if a task set is schedulable, and let one periodic task to be replaced with a flexible sporadic server with same processor utilization fraction, the new task set including the periodic tasks and aperiodic tasks served by this FSS server is also schedulable.

This Lemma gives one sufficient condition that flexible sporadic server can deploy the aperiodic tasks in hybrid process model. Thus we can derive a necessary and sufficient schedulable condition of a flexible sporadic server in a hybrid process model.

Theorem 5.2 If a flexible sporadic server is schedulable, all the tasks allocated within the execution budget of the flexible sporadic server are schedulable.

Proof: Let τ_1, τ_2,…, τ_s be a set of s tasks including the flexible sporadic server τ_m. Let τ_s to be replaced by τ_m and let τ_1, τ_2,…, τ_m be a set of m tasks that fully utilization the processor. Thus $U_m=C_m/T_m$ can give the upper bound of utilization fraction.

For i^{th} task registered in the flexible sporadic server with inter-arrival time L_i', service demand, execution period T_i' and execution budget C_i', it has the property that $U_i'=n_i'C_i'/(n_i \cdot T_i'+L') \leq U_m$, thus we obtain the execution time:

$$\sum_{j=1}^{l}[q_i(T)TU_i'] \leq \sum_{j=1}^{l}[q_i(T)T]U_m = TU_m$$

where $q_i'(x)$ is the faction of task allocation in the full lifetime of sporadic server, we assume the lifetime T is long enough and have $q_i(x) \leq 1$.

Thus we have the actual utilization U_s: $U_s = [\sum_{j=1}^{l}(n_i T_i')]/T \leq U_m$. Therefore we conclude that the flexible sporadic server is schedulable if all the tasks registered in a flexible sporadic server are schedulable.

Lemma 5.2 Given a flexible sporadic server ($w_{m,i}$, $C_{m,i}$, $T_{m,i}$) which is executed on the background of cyclic process model with a utilization fraction U', the flexible sporadic server is schedulable if $C_{m,i}/T_{m,i} + U' \leq U_b$ where U_b is denoted as the upper bound of processor utilization in specified scheduling discipline.

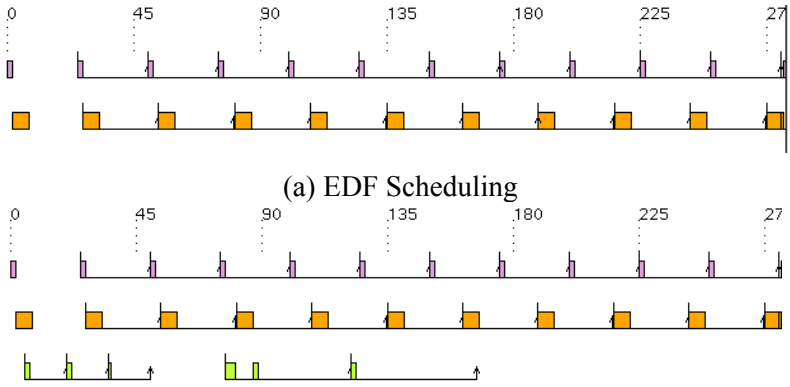
Lemma 5.3. Given s flexible sporadic servers τ'_1, τ'_2,…, τ'_s on an acyclic task execution system and each server τ'_m has its own constant processor utilization fraction U_m. Let we define a task set of multiple periodic tasks τ_1, τ_2,…, τ_s with each task has a same processor utilization U_m as that of relative FSS server τ'_m, if this task set of cyclic task execution is schedulable, then the acyclic task execution system is schedulable.

Proof: Given τ_1, τ_2,…, τ_s is schedulable τ_m. Let τ_1 to be replaced with FSS server τ'_1 with processor utilization fraction U_1. According to Lemma 1, we can construct a new task set Π_1 with FSS server $\tau_{1,o}$ and tasks τ_2,…, τ_s and conclude this task set is schedulable. In the schedulable subset Π_1' of τ_2,…, τ, let us replace τ_2 with FSS server τ'_2 (processor utilization fraction U_2), we can construct a new task subset Π_2' composed of FSS server $\tau_{1,o}$, $\tau_{2,o}$ and tasks τ_3,…, τ_s. According to Lemma 5.1, We can conclude the task subset Π_2' is schedulable. In the schedulable subset Π_m' of τ_{m+1},…, τ, let us replace τ_3 with FSS server τ'_{m+1} with processor utilization fraction U_{m+1}, we can construct a new task subset Π_{m+1} compose of FSS servers $\tau_{1,o}$, $\tau_{2,o}$…, $\tau_{m,o}$ and periodic tasks τ_{m+1},…, τ_s. According to Lemma 5.1, the task subset Π_m is schedulable. Consequently, we have the task set composed of FSS servers $\tau_{1,o}$, $\tau_{2,o}$…, $\tau_{m,o}$ is schedulable.)

This Lemma gives the necessary and sufficient condition that a flexible sporadic server can schedule the aperiodic tasks. It defines the formulation to set up the timing attributes of flexible sporadic server.

We use an example to show how to schedule the aperiodic tasks using a FSS server (see Figure 5.2). In this example, we apply EDF scheduling discipline to deploy all the real-time tasks. We assume a task set composed of two periodic tasks with (C, T) branch to (2, 25) and (6, 27) and a FSS server with its Processor Utilization Fraction constrained by GCD set (2, 15). In this case, aperiodic task ζ_1 is invoked at time instant 5, its execution budget at each invocation is 2, and its service demand is 5. When aperiodic task ζ_1 has (C, T) branch to be (2, 15) and its pre-defined period to be 15, it

holds a higher priority over that of task 1 and 2. Thus task 2 is preempted by task ζ_1 when t=5. Another aperiodic task ζ_2 is invoked at time instant 77 and assigned with (C, T) branch to be (6, 45). According to the scheduling algorithm, task 2 has a privilege over the aperiodic task ζ_2 deployed by FSS server. When t = 81, the aperiodic task ζ_2 is preempted by task 2.



(a) EDF Scheduling

(b) FSS Server in EDF Scheduling

Figure 5.2 Computation of response time

In the hybrid process model, we consider all the aperiodic tasks are deployed by flexible sporadic servers constrained by ($C_{0,m}$ $T_{0,m}$) branch and period. If a sporadic server is schedulable in a hybrid process model, the aperiodic tasks that can be deployed by FSS server are also schedulable in hybrid process model.

Specifically, we compare the Sporadic Server algorithm and Flexible Sporadic Server algorithm. In Sporadic Server algorithm, the server executes like other real-time process and is scheduled like a periodic task. While in Flexible Sporadic Server algorithm, a server is characterized by the utilization; its service demand and period are variable. It executes like an aperiodic task. Its utilization could also be variable with satisfaction of the schedulability condition of hybrid process model.

5.5.1 Performance of FSS Server

We have done some simulations to compare the performance of Sporadic Server algorithm and Flexible Sporadic Server. In the test, one task set is executed as a cyclic execution background and one server is executed to server aperiodic task scheduling (timing attributes are shown in Table 5.2). The scheduling policy is rate monotonic (RM) task scheduling. We assume that the aperiodic tasks have their inter-arrival times and service times exponentially distributed using parameter λ and μ where 1/λ is the mean inter-arrival time and 1/μ is the mean service time. The tasks deployed by these two servers have identical arrival distribution. We denote U_m as the utilization fraction of FSS server. Then the serving rate of FSS server μ′ is given by μ/Um. If there is no interference between other tasks, the average waiting time of aperiodic to allocate FSS server is given by ρ′/[μ′(1-ρ′)] from the M/M/1 queuing model (where ρ′ is λ/μ′).

In our experiments, the mean inter-arrival time of aperiodic task is 1sec, and the utilization of FSS server is always 1/10. We increase the mean service demand of the aperiodic tasks from 5ms to 50ms. Then we obtain the performance of FSS and SS server under variable system load (Figure 5.3). Figure 5.3a shows the comparison of average number of aperiodic tasks waiting for FSS and SS server. Figure 5.3b shows the comparison of average waiting time of aperiodic task competing the FSS and SS server.

We compare the average waiting time and utilization of the two systems and make some observations. When the mean load of aperiodic task increases, both the average number in waiting queue and the average waiting time of aperiodic tasks increase in these two servers. It is shown that the average number in queue and average waiting time of FSS server are less than that of SS server. This is because a less service demand generates a higher execution rate in FSS server algorithm. FSS server holds higher priorities with less service demand and it is quickly deployed in the system, which makes its response time is short. When the service demand is long, FSS server has a lower execution rate, which makes it holding lower priority. Thus a long aperiodic task will not preempt other real-time task and affect other real-time tasks. Sporadic Server (SS) is a periodic task and its priority cannot be adjusted dynamically like a FSS server.

With variable workload, we compute and compare the average number in queue and waiting time of the aperiodic tasks deployed by SS and FSS servers. We draw the conclusion that FSS server algorithm can optimize the system performance by reducing the response time and the average number of tasks in waiting queue compared with SS server.

Table 5.2 Task Set of Cyclic Execution

Task (i)	Ci	Di
Task 1	4	49
Task 2	10	54
SS Server	9	135
FSS Server	i (Service demand)	i * 15

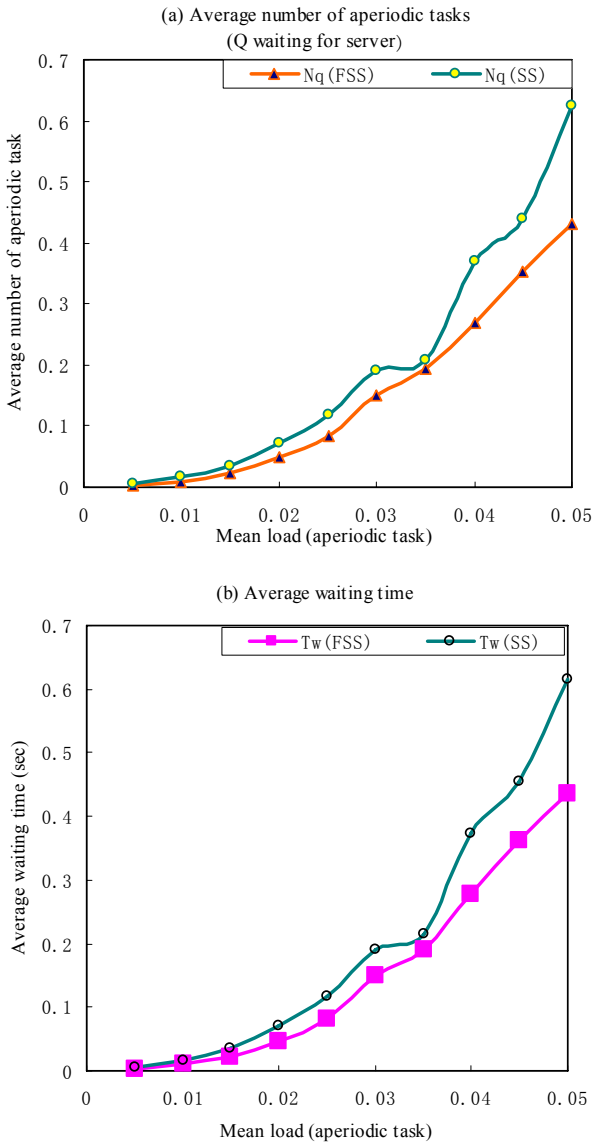


Figure 5.3 Queue and Waiting Time of FSS server and SS server

In our analysis, we investigate the performance of the two servers by modeling the aperiodic task as Poisson process and take some approximation in the relationship of arrival ratio and utilization ratio. The real system is more comprehensive compared with our model. If the model can be adaptive to the real world, the analysis would be more practical.

Compared with sporadic server, the flexible sporadic server has an important feature that it holds variable execution budget and period. The FSS algorithm provides a theoretical base for the task scheduling of complex real-time tasks. With a set of flexible sporadic server running in the system, we could deploy the real-time tasks in a complex process model.

5.5.2 Cyclic and Acyclic Execution

To investigate the task scheduling in the hybrid process model, we simulated the task execution of FSS server at the background of cyclic execution. In this simulation, we first construct a cyclic task set with (*C, T*) branch to be {(2, 49), (5, 54)} and a FSS server with ($C_{0,3}$ $T_{0,3}$) to be (1, 15), the system load is 20.0%. Then this FSS server have (*C, T*) branch to be ($1*n_i$, $15*n_i$) in this model. We increase n_i from 1 to 6 and obtain the WCRT of each task in each case. Table 5.3a and Figure 5.5a show the WCRT in RM scheduling. Table 5.3b and Figure 5.5b show the WCRT in EDF scheduling. We could find that the three tasks have equivalent WCRT in these two scheduling algorithms. We can see that the WCRT of FSS server is monotonic increasing with n_i increasing.

We increase the execution budgets by doubling the amount in previous simulation and the system load is changed to 40%. The timing attributes of periodic tasks and FSS server is shown in Figure 5.5c and 5.5d. In this simulation, the response time of the three tasks in RM and EDF scheduling is different. It shows a same property as that of

previous simulation: the WCRT of FSS server is monotonic increasing with n_i increasing. From the simulation of system load being 20% and 40%, the WCRT of task 1 and task 2 is not monotonic increasing or decreasing. This finding implies that a periodic task is possible to obtain a good response time if we can choose a suitable (*C, T*) branch for the FSS server.

Table 5.3 Example task set: time attributes and WCRT

a. WCRT in Hybrid Process model (load: 20.0%, RM scheduling)

task	τ_3:(C, T)	(1,15)	(2,30)	(3,45)	(4,60)	(5,75)	(6,90)
Cyclic	τ_1: (2,49)	3	4	5	2	2	2
	τ_2: (5,54)	8	9	10	7	7	7
Acyclic	τ_3	1	2	3	11	12	13

b. WCRT in Hybrid Process model (load: 20.0%, EDF scheduling)

task	τ_3:(C, T)	(1,15)	(2,30)	(3,45)	(4,60)	(5,75)	(6,90)
Cyclic	*τ1: (2,49)*	3	4	5	2	2	2
	τ2: (5,54)	8	9	10	7	7	7
Acyclic	*τ3*	1	2	3	11	12	13

c. WCRT in Hybrid Process model (load: 40.0%, RM scheduling)

task	τ_3:(C, T)	(2,15)	(4,30)	(6,45)	(8,60)	(10,75)	(12,90)
Cyclic	*τ1: (4,49)*	6	8	10	4	4	4
	τ2: (10,54)	18	18	20	14	14	14
Acyclic	*τ3*	2	4	6	22	24	26

d. WCRT in Hybrid Process model (load: 40.0%, EDF scheduling)

task	τ_3:(C, T)	(2,15)	(4,30)	(6,45)	(8,60)	(10,75)	(12,90)
Cyclic	*τ1: 4,49)*	12	12	14	10	8	8
	τ2: (10,54)	18	18	20	14	14	14
Acyclic	*τ3*	2	4	11	22	24	26

e. WCRT in Hybrid Process model (load: 26.7%, RM scheduling)

task	τ_3:(C, T)	(2,15)	(4,30)	(6,45)	(8,60)	(10,75)	(12,90)
Cyclic	*τ1: (2,49)*	*4*	*6*	*8*	*2*	*2*	*2*
	τ2: (5,54)	*9*	*11*	*13*	*7*	*7*	*7*
Acyclic	*τ3*	2	4	6	15	17	19

f. WCRT in Hybrid Process model (load: 26.7%, EDF scheduling)

task	τ_3:(C, T)	(2,15)	(4,30)	(6,45)	(8,60)	(10,75)	(12,90)
Cyclic	τ_1: (2,49)	4	6	8	2	2	2
	τ_2: (5,54)	9	11	13	7	7	7
Acyclic	τ_3	2	4	6	15	17	19

Considering the cyclic execution of task 1 (2, 49) and task 2 (5, 54) as the background environment, we only increase the execution budget of FSS server. Then the new FSS

server have (C, T) branch to be (2*ni, 15*ni) in this model. We can obtain the WCRT in figures 5.5e and 5.5f. In this case, the WCRT of FSS server is monotonic increasing with ni increasing.

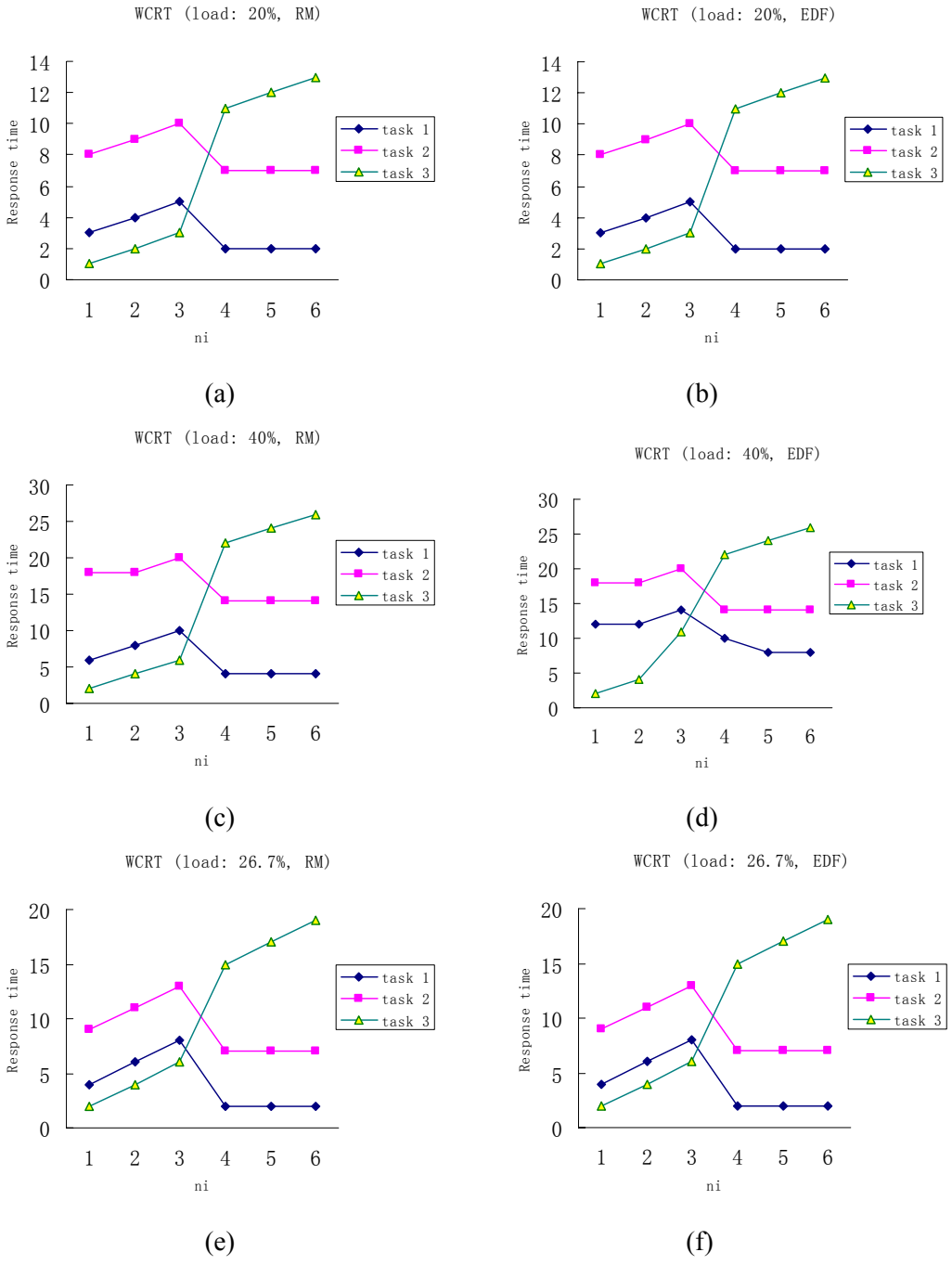


Figure 5.4 Execution of aperiodic task

To investigate the WCRT of periodic task and sporadic server, we set up a background of cyclic execution with task 1 (4, 49) and task 2 (10, 54). We decrease only the period of FSS server. The new FSS server is set up to have a (*C, T*) branch to be (1*2, T_i*) (50 $\leqslant T_i \leqslant 90$). This condition accords with the schedulability of hybrid process model. The WCRT of periodic task and FSS server in RM and EDF scheduling algorithm is shown in figures 5.6a and 5.6b. It is observed that the WCRT of task 3 decreases while the WCRT of task 2 increases when the period of FSS server is decreased to 53 under RM scheduling algorithm. This is because that at this point, FSS server has a period longer than that of task 2. Thus FSS server has a privilege level over task 2. When the period of FSS is larger than 53, FSS server always runs after other task execution. If the period of FSS is less than 53, sometimes task 2 is preempted by FSS server. Such a task-preemption makes the response latency of task 2 become longer.

Under EDF scheduling algorithm, the WCRT of task 2 and FSS server (task 3) is not monotonic increasing with the load increasing. This implies that it is possible to reduce WCRT of acyclic and cyclic execution in EDF scheduling algorithm by choosing a good setting of scheduling parameters for FSS server. As the response time computation in real-time scheduling is non-deterministic polynomial (NP) problem, the parameter of FSS server cannot be defined by a simple regulation. To solve the problem, the approach is to make predict WCRT prediction of FSS server at different utilization after the execution budget of periodic tasks can be predicted.

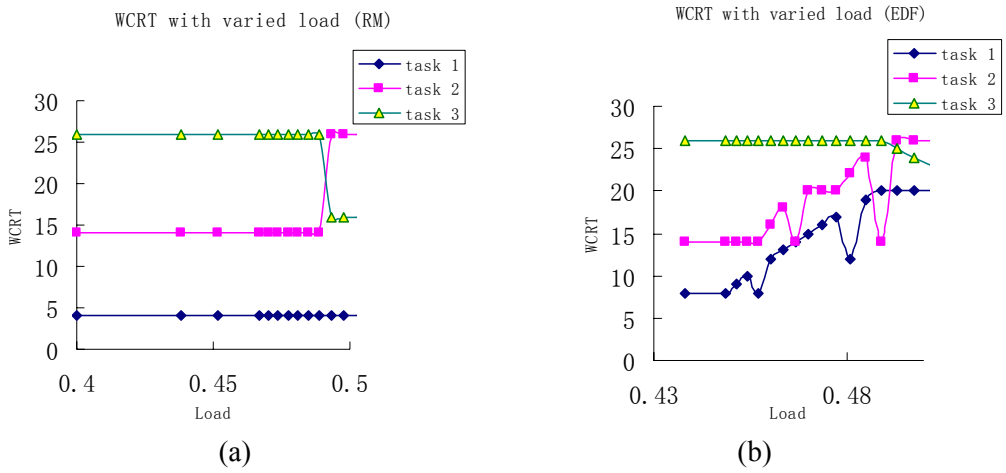


Figure 5.5 WCRT of Periodic Tasks and FSS server with Varied Load

5.5.3 Discussion

In this chapter we have defined a new process model composed of periodic tasks and aperiodic tasks. In such a model, aperiodic tasks are executed on the background of cyclic execution. Using this hybrid process model, we have presented the schedulability analysis and formulation of response time of static and dynamic task scheduling. This chapter also presents the simulation of cyclic and acyclic task execution in hybrid process model.

A realistic method to deploy the aperiodic tasks is to make use of a flexible sporadic server that will replenish its execution budget and define its period when an aperiodic task is registered in the FSS server. Then the aperiodic task can be scheduled just as other real-time task in a hybrid process model. When the aperiodic task finishes all its execution, it will be unregistered in the flexible sporadic server and release the FSS server. One flexible sporadic server can serve only one aperiodic task at one time. We

have investigated some task execution in a hybrid process model. For a FSS server, it is constrained by an upper bound of processor utilization fraction U_m, which can be given by (C_m, T_m) branch with $U_m = C_m/T_m$ and GCD(C_m, T_m)=1. The actual utilization fraction of FSS server can be less than the bound U_m.

CHAPTER 6
IMPLEMENTATION OF RTS-LINUX

6.1 Introduction

As an important part of the thesis, this chapter gives an overview of the implementation of real-time control in real-time supported Linux (RTS-Linux). This real-time operating system (RTOS) has a preemptive kernel and makes use of a control module to establish a real-time control solution. It combines the preemption patch and the low-latency patch to establish a preemptive system. Based on the preemptive kernel, RTS-Linux realizes real-time control and is compatible with standard Linux. As the task scheduling of Linux system cannot support the predictable activities to meet the real-time requirements of application, RTS-Linux provides a real-time control sub-system to deploy real-time tasks. This subsystem is mainly implemented within a loadable module: RTS driver. The driver is a virtual layer between the standard Linux and user applications, which contains the data abstraction and the input/output control of tasks. The RTS driver provides the Application Programming Interfaces (APIs) between the standard kernel and user applications.

This system provides flexible scheduling capabilities for user to define a flexible scheduling framework. It also provides a low-latency-patched kernel so that real-time applications will not suffer long delays. Moreover, it provides queue manager (QM) mechanism to decrease schedule jitter.

6.2 Mechanism to Improve Response Latency

6.2.1 Preemption Patch

For the two approaches of real-time executives investigated in section 4.2, their performances are balanced to achieve an optimal performance. The preemption patch designed by Monta Vista [25] and the low-latency patch by Ingo Molnar [27] are ported onto the standard Linux. This section reveals a practical method of building the standard Linux to a preemptive kernel.

As we introduce in section 6.2, in order to reduce the response time, the primary work is to modify the standard Linux into a preemptive scheduling system. The real-time requirements of user applications require the OS to be preemptive, that is, a running process will be preempted by a higher priority process and switched to this process very quickly. In the preemptive kernel ported with the pre-emption patch [25], all unlocked functions and kernel threads are preemptive except the following:

- Handling interrupt, in *softirq*, or performing 'bottom half' processing;
- Holding a *spinlock*, *readlock* or *writelock*;
- Scheduler is executing in kernel mode;
- A newly created process is in initialization.

In the RTS-Linux, Linux processes become preemptive. The process currently allocated the processor has the highest priority. If a process enters the TASK_RUNNING state, the kernel checks whether its priority is greater than that of

the currently running process. If so, the current task execution is preempted and the scheduler is invoked to select the other process to run.

For all above states, a counter labels the process being preemptive or not. When the kernel enters into the state that is not preemptive, the counter is incremented, indicating that current process is not preemptive. The function of *preempt_enable* and *preempt_disable* is called when several bottom handlers are allocated and released. The scheduler enables preemption at the beginning of *schedule_tail* function and disables preemption at the end of *schedule_tail*. The kernel processes are set to be preemptive when many lock/unlock functions are called. This preemptive is set by preempt_enable. Particularly, the *preemption_goodness* function compares the value of *goodness*(). If the task preemption happens, the scheduler selects the outstanding task and performs a task switch. As a result, the execution of the preempted task is resumed very quickly.

6.2.2 Long-latency Points (LLP) in Linux

There are some sources of long latencies on current hardware in the Linux kernel [27], such as calls to the disk buffer cache, creation and termination of processes and so on. Moreover, when a process returns from interrupt, and if the processor is in kernel mode, the scheduler will not run until the system call is finished. All these will possibly cause a long latency.

In order to reduce such kernel latencies, some preemption points are inserted into the system calls of the kernel [27]. The new scheduler will check task-preemption and optimize the execution of long-latency task. The flow of preemptive kernel is shown in Figure 6.1:

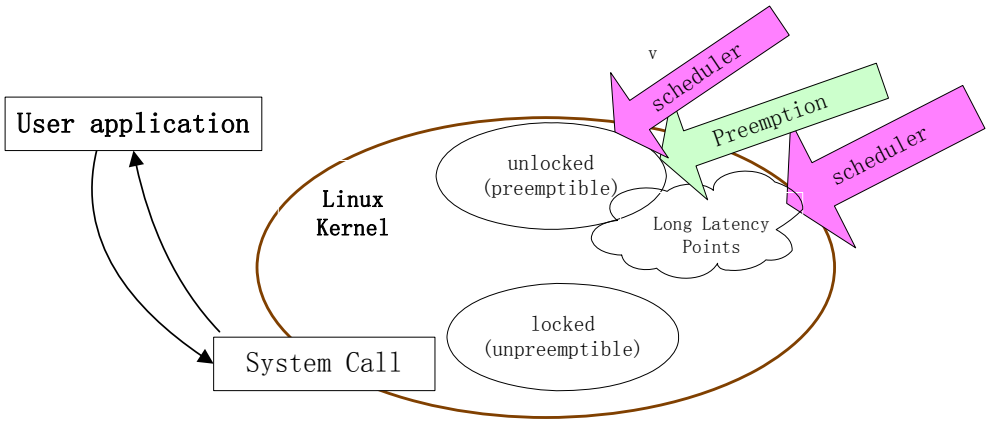


Figure 6.1 Preemptive RTS-Linux Kernel

The response latency is an important metric to evaluate the system performance. Therefore a comparison of the response latency on the standard Linux kernel and the preemptive kernel is explored in section 7.1.

6.3 Real-Time Control Subsystem

6.3.1 Virtual Device driver

As we introduce in section 4.1, RTS driver is the heart of the real-time control and implemented as a loadable kernel module. RTS driver begins to work in the kernel space once the module is loaded into kernel [23, 24]. How the RTS module interacts

with Linux kernel and the applications is shown in Figure 6.2. The driver and the Linux kernel share the hardware control to minimize the kernel's modification. RTS driver has these important features as follows.

- All real-time features are implemented as kernel modules.
- Required modification on the standard Linux kernel is minimized.
- APIs of RTS driver is simpler compared with Linux OS. It is because RTS driver shares all the primitives with the standard Linux kernel. Therefore RTS solution is more flexible and compatible.

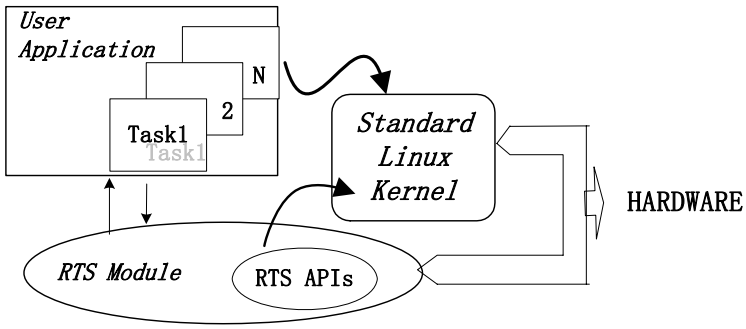


Figure 6.2 RTS driver cooperating with Standard Kernel

This driver for real-time control is divided into two modules. Module *rts-mod.o* contains a RTS scheduler for all real-time the applications. The source codes of module *rts.o* are placed in directiory *drivers/rts/* and source codes of *uds* module are in directory *drivers/rts/uds*. Module (*uds.o*) supports a user-defined scheduler (UDS). The header files could be found in *include/linux/* directory. RTS driver can be accessed and controlled by a character device "*/dev/rts*". For writing the device driver, the book "*Linux Device Drivers"* is the major reference [28].

Beyond the scheduling paradigm and tasks management, some add-on components are presented in RTS driver: a queue manager, proc file suppor and UDS scheduling framework. Several macros are defined when configuring the kernel's components (Figure 6.3), which would determine whether these facilities in RTS driver are inserted or not.

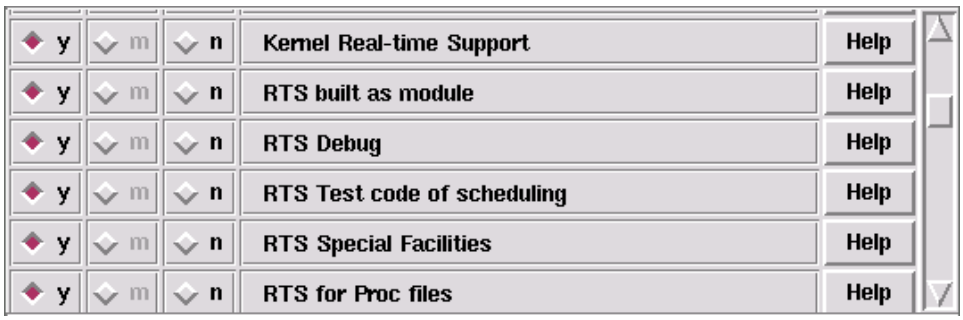


Figure 6.3 Configure Options for RTS-Linux

6.3.2 Admission Controller

As the capacity of processor is limited, it can only meet the real-time requirement under specific scheduling condition. When a real-time task is created in RTS system, the facility of admission control checks the condition of schedulability and determines whether a task can be activated. The schedulability is quite important to realize the real-time control and has been discussed in chapter 5.

6.3.3 Flexible Scheduling Framework

In the real-time scheduling in this system, there is one important assumption to make: the relative deadline of a task is designed to equal to its period. This is a weakness in

the practical using of RTS-Linux. Thus one flexible scheduling framework is established as an appendix action to treat this weakness.

As the events in the real world are sophisticated, a scheduling discipline is defined according to the requirement of user applications. It makes the task scheduling more efficiently. User-defined Scheduler (UDS) is a facility required by the applications. This facility provides a scheduling framework that is adaptive to the various controls of user application. This framework is shown in Figure 6.5. UDS scheduler could completely implement the job of RTS scheduler. It holds its structures and methods that can be used to implement flexible scheduling interface. UDS scheduler contains its own APIs that allow applications to use their own special scheduling in a way compatible with RTS interface.

UDS scheduler works as a loadable kernel module. To minimize the size of UDS scheduler, UDS scheduler shares some ioctl functions with RTS scheduler. The virtual device */dev/rts* is used as the standard real-time scheduling interface. Some /proc is used as interface between kernel space and user space [34]. Additionally, a system call is used by the user application to control the UDS scheduling policy. Finally, the code being used by the UDS schedulers is built in RTS-Linux with the specified configuration option.

The real-time tasks can be deployed by UDS scheduler when the task registers the scheduling policy as *UDS_SCHED*. After the real-time task has defined a policy

presented in RTS-Linux, if the policy need to be adjusted, the *ioctl* function *ioctl_set_policy* is used to set up the policy.

6.3.3.1 Correlation with RTS Scheduler

UDS scheduler does not affect the current scheduling policy such as round-robin, FIFS/RM/EDF and Linux multi-process scheduling that assigns certain execution budget to each process. UDS scheduler is an alternative scheduler that can be treated as a component at the same level with RTS scheduler. Each real-time task in the RTS system has its scheduling priority. For UDS scheduler, the system priority may be predefined and changed by UDS interface. The scheduling policy in UDS scheduler also makes use of priority-driven task scheduling.

If the policy UDS_SCHED is registered in RTS system, UDS scheduler is called up when kernel scheduler probed this policy. The user application invokes ioctl function to change the scheduling parameters. When a real-time task is created in RTS system, the admission controller checks whether the task can be admitted to run. After that, the user application can change the scheduling parameters to achieve a predictable and efficient scheduling. Thus UDS scheduler chooses and activates the eligible task from all the ready real-time tasks with UDS_SCHED using a customized scheduling policy.

UDS scheduler shares the ioctl function in RTS driver as well as the data types. The ioctl functions mainly include:

- Accept or grant the initialization of task and task's termination

- Activation and suspension of a real-time task
- Interrupt handler and transferring data between the device managers.

6.3.3.2 Data Structure of UDS

The following data structure is defined in <*linux/uds.h*>. These data types are only used in UDS scheduler to schedule real-time tasks.

```
struct udssched_tsk{
    int index;
    int pid;
    struct rts_task *owner;
    int sched_priority;
};

struct uds_queue{
    struct udssched_tsk tsk[MAX_UDS];
    int uds_nicer;
    int uds_flag;
};
```

The structure *udssched_tsk* represents a task that run on UDS scheduler. It contains the parameters associated with flexible scheduling policy. All the symbols are included in <*linux/uds.h*>.

6.3.3.3 Example UDS Scheduler

The patch of UDS scheduler presents an example loadable scheduler encapsulated in the files <drivers/uds/my_sched.c> and <include/linux/uds.h>. Using this framework, a scheduler or algorithm designed by the developer could be created. The function related with module is listed as below.

i. *sys_uds_scheduler* is the controlling/monitoring interface for UDS scheduler.

ii. *my_scheduler* is the dispatcher to choose the next eligible task to run.

iii. *my_change_priority* is the function to set up scheduling parameters of the tasks running on RTS environment.

iv. *my_admission_add_task* is the admission controller called when new task is registered in RTS environment.

v. *init_module* calls *uds_init* when loading the module, set up relative UDS control functions.

vi. *cleanup_module* releases all UDS scheduler's functions at module remove time.

6.4 Real-Time Scheduling

This section intends to highlight the details of real-time task scheduling. In the RTS driver, the real-time tasks have their own scheduler. This scheduler can choose an outstanding task only from the real-time tasks, which minimizes the scope that scheduler and explores decrease the scheduling overhead. The job of scheduler is to switch from one task to another and set the interrupt for the next time. Figure 6.4 shows a simplified schematic of how RTS scheduler is internally implemented. In this figure, UDS scheduler provides a scheduling framework.

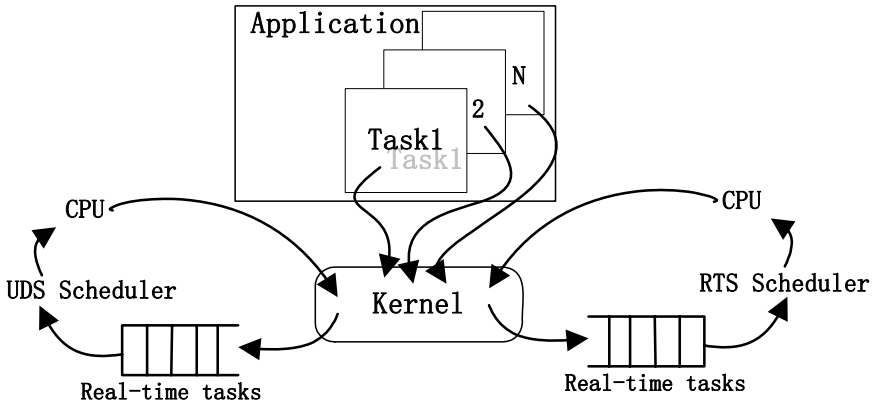


Figure 6.4 RTS and UDS Scheduler

The following capabilities are presented in the real-time execution:

- Periodic Real-time Tasks: Periodic execution of task is commonly required in real-time property. The RTS driver allows real-time processes to register as periodic processes and executed as a periodic model.
- Sporadic Real-time Tasks: The sporadic tasks coexist with the periodic task and run just like a periodic task. When the priority is assigned to a sporadic task, it is deployed by a server of aperiodic task scheduling.
- Priority-driven Scheduling: A priority-based scheduling policy is adopted in the RTS-Linux. When a real-time task is registered in RTS driver, its priority is defined based on its time attributes (such as period or deadline depending on the specific scheduling algorithm). In this case, the task can be scheduled with a priority.
- User-defined Scheduler: A scheduler with its data structure and scheduling parameters is used to implement an adaptive scheduling framework. Thus the application-developer can define a flexible scheduling-policy.

6.4.1 Task Management

In RTS-Linux, One-to-One mode is adopted to deploy a set of real-time tasks. One-to-One mode is that one real-time task is attached to a standard Linux process. These two parts are run in RTS driver side and standard Linux side separately. As a real-time task get initialization or termination, the execution process that owns it may be resumed and suspended respectively. Under one-to-one mode, the real-time tasks have access to all the services and APIs of standard Linux.

Before illustrating the details of real-time task management, it is important to understand the internal data structure. This data structure is shown in Figure 6.5. Each process has a structure *time-param* containing all the information needed to perform the real-time control operations. It includes a set of parameters composed by four individual timing attributes. Besides, the user interface of real-time control contains the task management such as initialization, admission control, suspension, resuming and scheduling.

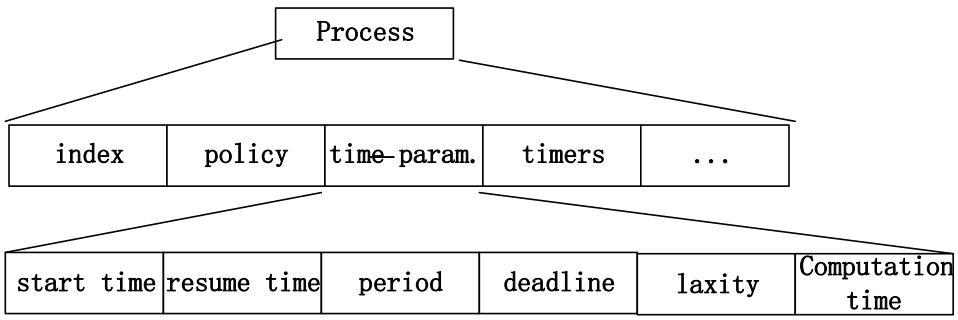


Figure 6.5 Data Structure of Real-Time Task

The real-time tasks have following five states.

1 *RUNNING* is the state that a task is assigned with the time slice and allocates CPU.

2 *READY* is the state that a task is waiting for CPU. Its condition is ready for execution.

3 *IDLE* is the state that a task used up the time slice and is waiting for the next period. Its condition is ready for execution except the time slice.

4 *DELAY* is the state that a task is suspended. A suspended task will be awakened by the time after the specified time.

5 *WAITING* is the state that a task is waiting to be initialized in RTS driver.

The RTS scheduler transits the tasks among the different states. The state transit diagram is shown in Figure 6.6. When a task is initialized in the system, its state is initialized as TASK_RUNNING(ready). Its state is refreshed to TASK_RUNNING (ready) when it is activated at the beginning of each invocation. When it is chosen by the scheduler and allocates the processor, its state is transited from TASK_RUNNING (ready) to TASK_RUNNING. When it uses up its execution budget, its state transits from TASK_RUNNING to TASK_IDLE. When it is preempted by another task holding a privilege over it, its state transits from TASK_RUNNING to TASK_DELAY. In a soft-real-time scheduling, the real-time tasks may not be independent from each other. One task needs to wait for some sharing resources that are released by other tasks, its state transits from TASK_RUNNING to TASK_WAIT.

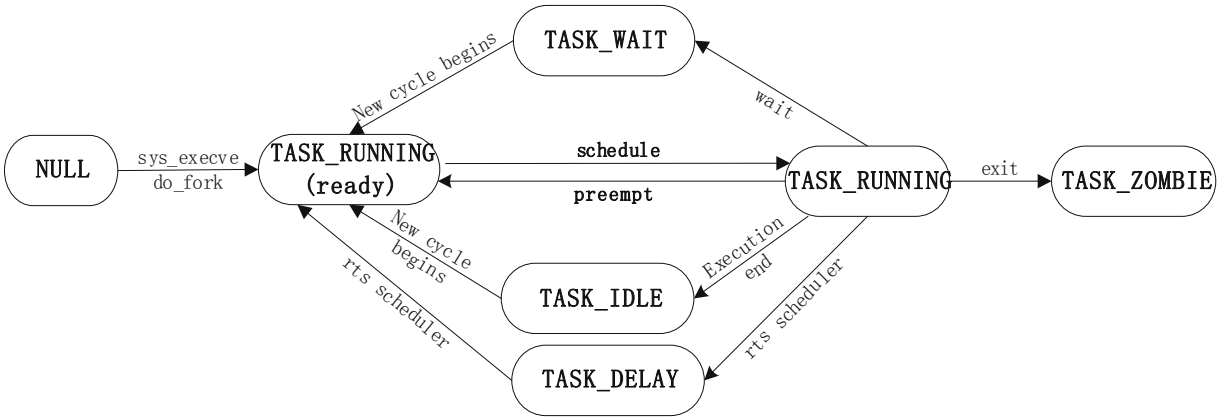


Figure 6.6 State Transition Diagram (RTS scheduler)

The scheduling policies in RTS extension are listed in Table 6.1. These policies are only used to schedule real-time tasks.

Table 6.1 Scheduling Policy in RTS module

Symbol	Description
RM_SCHED	A Scheduling Policy Adopted the RM algorithm
EDF_SCHED	A Scheduling Policy Adopted the EDF algorithm
MLF_SCHED	A Scheduling Policy Adopted the MLF algorithm
UDS_SCHED	User-Defined Scheduling Policy

6.4.1.1 Timer Mechanism

Linux provides a timer mechanism, that is, some events can be registered in a timer and they are scheduled sometime in the future [29]. In the timer structure, the expiry time of the events called up and a function of timer handling is defined for each timer registered.

The RTS driver provides two timers for each-time process. They manage the task with period and deadline. They are described as bellow:

- **PERIODIC TIMER**: it is applied to manage the periodic task. The timer's expiry value is set as *starttime+period*. When the timer is expired, the *resume_time* & *expires* of the real-time task are updated and a new timer is added into the timer list.
- **ONESHOT TIMER**: this timer is used to manage the deadline. When the timer is expired, a soft real-time task is moved to the last of task queue and waited for the next scheduler; if it is a hard real-time task, the missing deadline may cause the system death failure.

6.4.1.2 Task Synchronization

In our real-time control subsystem, the shared resources are managed by mutex exclusion mechanism. When a task proposes to allocate a shared resource, it must acquire its mutex lock first. Thus the conflict of shared resources could be avoided. RTS driver provides a set of locking protocols for the mutex variable.

One important problem that must be considered in RTS driver is priority inversion [36]. For instance, a high-priority task may be expected to execute when the low-priority tasks are holding a mutex lock. Thus the high-priority task may miss the deadline. To avoid the priority inversion, the priority inheritance protocol [38] is implemented in task synchronization. When a high-priority thread is blocked on a resource locked by a low-priority thread, the low-priority is boosted to that high-priority. Priority ceiling protocol [37] is another solution to priority inversion. The protocol has the following results. A priority monitor is initialized with a ceiling

priority. When a task requires the mutex and enters the mutex lock, its priority is raised to the ceiling priority. Thus other threads competing the mutex lock will not preempt the thread in execution. The mutual exclusion locks helps to realize the task synchronization.

In the mutex mechanism, RTS driver provides priority-driven supported scheduling and synchronization facilities. Two protocols have been implemented as priority-based protocol and priority inheritance protocol.

6.4.2 Scheduling Algorithms

A priority-based scheduling policy [30] is adopted in the RTS-Linux. When a real-time task is registered in RTS driver, its priority is defined based on its time attributes (such as period or deadline depending on the specific scheduling algorithm). It can be modified according to the importance of tasks using IOCTL function. The details about implementing scheduling algorithm and facilities are described in this section.

To make the tasks execute precisely and periodically, three scheduling algorithms are adopted: Rate Monotonic (RM), Earliest Deadline First (EDF) and Minimum Laxity First (MLF). Table 6.2 shows the scheduling elements in these algorithms in RTS-Linux. The actual time attributes, the priority and scheduling elements are listed for each algorithm.

Table 6.2 Scheduling Elements of Real-time Task [6]

algorithm	Computation time	period	deadline	priority	Scheduling element
RM	Y	Y	Y	*execution rate*	priority
EDF	Y	Y	Y	*MAXLAT + jiffies - cur_dd*	priority
MLF	Y	Y	Y	*MAXLAT+ cur_dd- laxity -jiffies*	priority

6.4.2.1 Rate Monotonic (RM)

Under the RM algorithm in RTS driver, the real-time tasks have their priorities respecting to their execution rates. The task with highest priorities is assigned with processor and other shared resources. Each task has an execution budget, when the task used up it execution budget, its state is changed to IDLE and releases the shared resources. The scheduler will switch to another eligible task.

6.4.2.2 Earliest Deadline First (EDF)

Working under the EDF algorithm, a task is assigned with its priority once it is registered in RTS driver. The priorities of tasks are updated at the beginning of each invocation. Its priority is dynamically changed respecting to its scheduling parameter. The task with closest deadline is assigned with the processor. Compared with RM algorithm, EDF algorithm improves the max utilization with satisfying the schedulability.

6 cur_dd is the deadline of a task; Y is the parameters is defined and valid in the scheduling paradigm jiffies is the number of clock tick; Laxity is the remaining execution time of a task.

6.4.2.3 Minimum Laxity First (MLF)

MLF algorithm is an optimized dynamic priority scheduling. In MLF algorithm, the priority of a task is assigned according to its remaining execution time. The highest priority is assigned to the task having the least laxity. The task with least laxity is assigned with the processor. In case of MLF, the priorities are dynamically updated once the deadline or remaining execution time is changed. Panwar and Towsley [33] show the minimum MLF algorithm maximizes the utilization fraction of tasks execution. The strengths and drawbacks of the above three periodic scheduling algorithm has been shown in section 3.1.1.

6.4.2.4 Sporadic Task Scheduling

In RTS scheduling, the FIFS scheduling can be used to schedule sporadic tasks. As this scheduling may make other real-time tasks miss deadline, we apply a flexible sporadic server algorithm to deploy the sporadic tasks (shown in section 5.5). In this algorithm, we adopt the FSS server to deploy the sporadic real-time task. For FSS server, its execution time and period can be varied but its utilization needs to meet an upper bound in the schedulability condition. According to the hybrid process model, a sporadic task is constrained by four important attributes: starting time, period and service demand. When a new sporadic task is created in the system and registered in FSS server, the execution time and period of the server are refreshed. After the sporadic task finishes the execution, it works as a server and waits for the next sporadic task coming into the system.

6.5 Queue Management

Queue manager (QM) mechanism targets to reduce the schedule jitter. It is applied in both a Linux task scheduling and RTS driver. There are four kinds of task scheduling in RTS-Linux system. Three kinds of disciples inherited from Linux system are first in first serve (FIFS), Round robin and Time-sharing scheduling. Another kind is the real-time scheduling in RTS driver. In these task scheduling usually the scheduler will explore the whole task queue to find the eligible to allocate the processor. Its computation complexity is *O(n)*.

However with the queue manager to manage the task queue, the proceeding of schedule is modified as follows. When the first task is initialized in the system, it is inserted next to the head of run queue (Figure 6.7a). Task 2 is initialized in the system. As it has a higher priority than that of task 1, task 2 is inserted before task 1 (Figure 6.7b). The tasks are queued in the order of decreasing priority. The ordering of task queue is resumed when there is any new tasks, state changes and priority modification. So when task 3 is generated in the system, queue manager will explore the task queue and insert task 3 between task 1 and its neighbor (task 1) (Figure 6.7c). The computation complexity of task exploration in queue manager is less than that of the normal task scheduling without Queue Manager (QM). With a queue manager to update the eligible task, the scheduler fetches the task directly; whose computation complexity is *O(1)*. Queue manager (QM) mechanism decreases the computation complexity in the task scheduling.

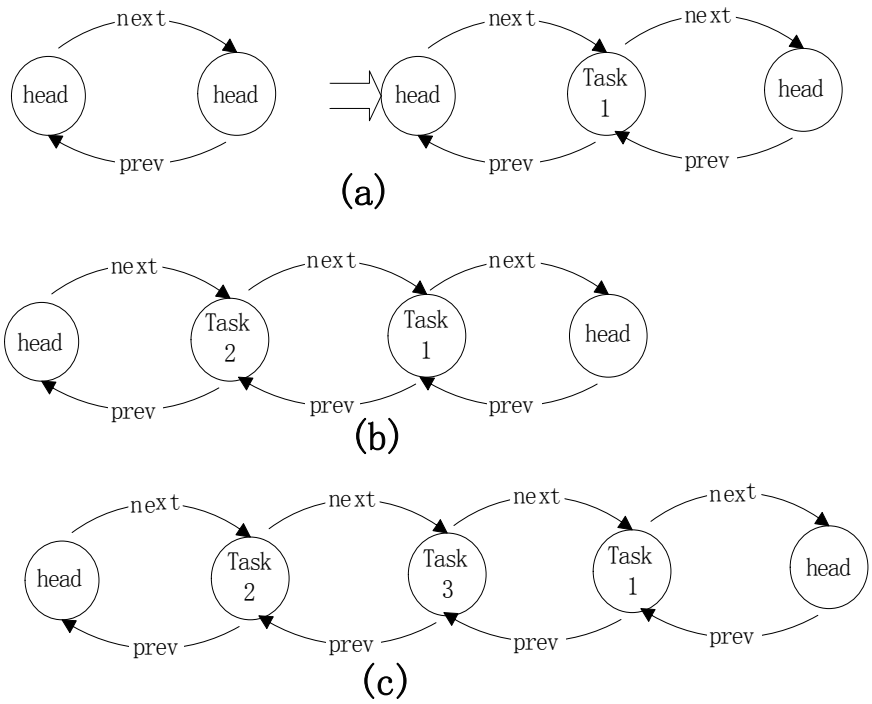


Figure 6.7 Task Queue and Task Management

An important parameter used to reveal the system performance is the response latency. The response latency accounts for interrupt latency and scheduling overhead. Interrupt Latency is the interval between the moment of an interrupt called and the beginning of the interrupt handler. Scheduling overhead is the interval between making scheduling decision and context switching. The context switch overhead depends on the number of windows to be saved and the memory to be switched.

The affect of scheduling overhead on the response latency is pictured in Figure 6.8, which shows the items in response latency. A task cannot be preempted or destroyed until the scheduler is invoked. At each timer interrupt, the scheduler executes to choose the next task if it found the execution budget is decremented to zero. If a higher priority task arrives, the previous active task is preempted. Thus the overhead can be

grouped in timer overhead, preemption overhead and exiting overhead. Timer overhead includes the overhead of interrupt handling, task scheduling overhead and the time returning to previous task when no task-preemption happens. Preemption overhead mainly comprises context-switching overhead, which is the time to save the states of previously task control blocks and load the eligible task. The exiting overhead describes the interval to handle the trap and load the next task from run queue. Respecting to the granularity of the timer interrupt, the scheduling overhead at every timer interrupt affect the worst-case response time.

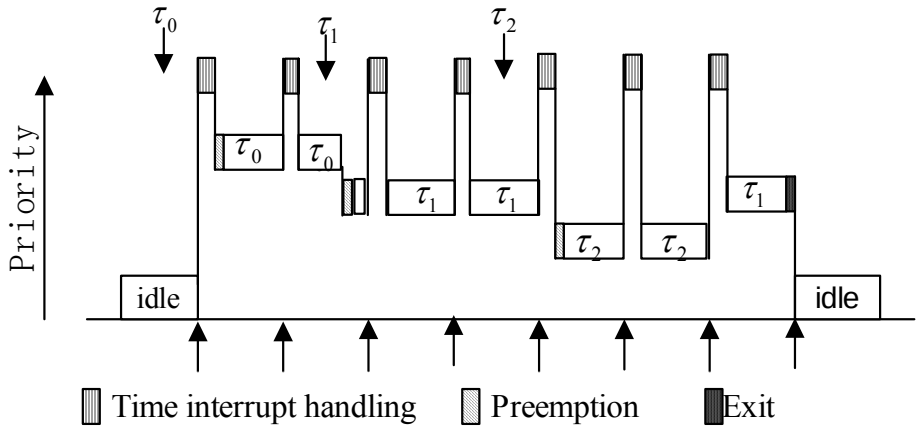


Figure 6.8 Timer-driven scheduling in Linux

6.5.1 QM Mechanism in Linux Scheduling

In the standard Linux scheduling, three scheduling policies are: FIFS scheduling, round-robin and a priority-driven scheduling. We introduce the implementation of queue manager (QM) in these scheduling policies in this section. In Queue Manager (QM) mechanism, three queues are used to sort the activated tasks. An activated task is

queued in an identical queue according to its scheduling policy. The task scheduling can be divided into two steps done by the scheduler and Queue Manager (QM) separately. When the scheduling condition including state or priority is changed, Queue Manager (QM) resort the task queues and adjust the eligible task accordingly. If the next eligible task is changed, the kernel scheduler is called up. For three classes of tasks in the system, the scheduler chooses the eligible tasks from the head of three queues: FIFS first, then Round-robin, and priority-driven finally.

6.5.2 Queue Manager in Real-Time Control

The facility of Queue manager (QM) is presented in RTS driver. In this scheduling mechanism, the job of real-time scheduler is to switch from the current executing process to the next eligible task. Similarly to QM in standard Linux scheduing, the queue manager in real-time control resorts the activated real-time tasks and selects the next eligible real-time task. It is executed when a priority or status of any real-time tasks is modified. Only when the eligible real-time task is changed, the kernel schedule is called up.

Queue Manager (QM) mechanism works on both static scheduling disciplines and dynamic scheduling disciplines. When scheduling condition is changed, queue manger is called up to update the eligible task and task queue. Only when the eligible real-time task is updated, the scheduler is called up to do the context switching. In a specified scheduling discipline, queue manager (QM) executes under these situations:

- When a new real-time task may be initialized in the system, its priority is defined. Then the queue manger appends it to task queue and resort the task queue. It also determines the next eligible task.

- When a task's priority is changed with the varied scheduling condition or user-defined, queue manger will resort the task queue.

- When a task's state is changed, queue manager update the eligible task and task queue as the eligible task can be chosen from only the tasks in the state of READY.

Queue manager (QM) is an optional facility in real-time control. Thus in the remaining parts of the thesis, *QM* represents the case that task scheduled with Queue Manger, while *non-QM* represents the case that the task scheduled under normal Task scheduling. The comparison of the two cases is shown in Table 6.3. In all the experiments shown in this thesis, if we do not specify the utility of queue manager (QM), the experiment is conducted without queue manager (QM) mechanism.

Table 6.3 Scheduling Activity of non-QM and QM

Scheduler (non-QM)	Scheduler (QM)
	One queue to keep all the active tasks;
If schedule condition changed;	If eligible task is changed;
Schedule called up;	Schedule called up;
Check all the active tasks;	If there is new task, add to the special location of active tasks;
Get the task with highest priority;	If there is any priority or state modification, re-locate the task in the active queue;
Context Switch.	Get the task with highest priority;
	Context Switch.

6.6 Application Programming Interfaces (APIs)

The driver is accessed using Input/Output control (IOCTL) APIs. The functions listed this section are enough to control the virtual driver and manage the real-time tasks. The driver can interact with the applications through some proc files so that the applications can inspect the status of real-time tasks. To illustrate the usage of RTS driver simply, the important I/O control APIs are introduced below.

6.6.1 Register and Un-register a Real-time Task

The user applications can register and release a real-time task using these functions:

- *int ioctl_register_tsk(int pid, struct rts_param par)* - *r*egister process as a real-time task and the real-time task's name is inherited from the process.
- *int ioctl_del_rts(int pid)* - delete a real-time task. It will free all resources allocated by the task and release the memory.
- *int ioctl_clear_events(void)* - kill all real-time tasks

6.6.2 Parameters of Real-time Tasks

The user applications can fetch and change the parameters of a real-time task using these I/O control functions:

- *int ioctl_get_rts(pid_t pid)* - get all the settings of real-time task
- *int ioctl_set_priority(pid_t pid, unsigned long priority)* – set the priority of a real-time task, this is available for the UDS scheduler
- *int ioctl_set_deadline (pid_t pid, unsigned long deadline)* - set the deadline of a

real-time task.

- *int ioctl_get_deadline(pid_t pid)* - get the deadline with real-time process's pid

6.6.3 Scheduling Policy in RTS

The user applications can set up the scheduling policy of a real-time task using these I/O control functions:

- *int ioctl_set_policy (int policy)* - setup the real-time scheduling policy. In the prototype, *policy* is the real-time scheduling policy. It is initialized as 0. The *policy* can be either EDF_SCHED or RM_SCHED. If *policy* is EDF_SCHED, then the real-time task will be scheduled by EDF algorithm.
- *int ioctl_get_policy ( )* - gets the current real-time scheduling policy.

6.6.4 Other IOCTL Function

- *int ioctl_suspend_rttask (pid_t pid, int sig)* - suspends a real-time task.
- *int ioctl_rts_getid_from_name (char *pname)* - returns a real-time process's id from its name.
- *int ioctl_resume_rttask(pid_t pid)* – wakes up a real-time task.
- *int ioctl_get_scheduler(pid_t pid, int policy, struct sched_param *param*) - fetches the schedule policy and parameters of the real-time process.
- *int ioctl_set_scheduler(pid_t pid, int policy, struct sched_param *param)* - sets the sched_param (*rts_priority*) of the real-time process.

6.6.5 APIs of Flexible Scheduling Framework

A set of scheduling actions are triggered in the UDS scheduler by the user application. These actions are:

- UDS interface to implement user-defined scheduling actions
- Modify the priority of the real-time tasks
- Choose an eligible task from the real-time tasks
- Implement the Admission control

6.7 Summary

We have shown the implementation of the sub-system of real-time control. Two patches are built into Linux kernel to improve the response accuracy. A Queue Manager mechanism is used to improve timing response accuracy by reducing the computation complexity of task scheduling. The real-time control is supported in RTS-Linux using a subsystem of real-time control. All the real-time control facility is supported using loadable kernel module to improve the compatibility of kernel and minimize the kernel modification. In this subsystem, several common-used algorithms of real-time task scheduling are supported. Besides this, a flexible scheduling framework provides a method for the developers of real-time applications to develop the adaptive scheduling disciplines.

CHAPTER 7
PERFORMANCE EVALUATION

In order to evaluate the performance and verify the basic functionality of the real-time extensive implementation, some tests are conducted on the modified kernel. The experiments investigate the characteristics of real-time scheduling in RTS-Linux, including the response latency, scheduling paradigm and scheduling precision. All the tests were run on IPAQ H3600; the hardware and software environment is introduced in Appendix A.

The rest part of the chapter is organized as follows. Section 7.1 shows the response latency in RTS-Linux. Section 7.2 presents multiple examples of real-time task scheduling using rate monotonic (RM), earliest deadline first (EDF) and minimum laxity first (MLF) task scheduling. Moreover, some evaluation in terms of schedule jitter, task preemption and missing-rate are revealed in section 7.3 and 7.4. The final section draws certain conclusion from the experimental result.

7.1 Response Latency

An important metric to reveal the timing performance is the response latency. The response latency depends on interrupt latency and scheduling overhead. Interrupt latency is the interval between the moments when an interrupt is called and the

interrupt handling begins. This metric also affects other aspects of system performance. Longer interrupt latency may cause a lower boundary of the scheduling overhead. Scheduling overhead is the interval required to make scheduling decision and the context switch. The time to make scheduling decision depends on the number of tasks in the system. The context switch overhead depends on the number of windows to be saved. Respecting to our implementation of real-time executive, the response latency of real-time FIFS scheduling and the scheduling paradigm of real-time process are investigated

In the experiment of response latency, a test program is developed to compare the response latency of the RTS-Linux kernel and standard Linux under various system load. The Rhealstone Benchmark [35] is a well-known benchmark for real-time operating systems proposed by Kar and Porter [35] to investigate the real time performance of multitasking systems. Based on this proposal, we build a program that target on ARM. For the purpose of comparison, the measurement program executes on the different Linux kernel. Its scheduler policy is set to SCHED_FIFO that is common to the standard Linux and RTS-Linux kernel. As FIFS task holds a higher priority over other processes, this setting ensures that this task is scheduled before all other tasks. Firstly, the program sets up a lot of parameters and calibrates a ticks-per-second of OSCR register and time stamp from *gettimeofday*( ). Secondly, the periodic interrupts are raised on the processor under a certain execution time. Finally, it reads OSCR register to get the elapsed time.

In the experiment, the amount of loops decides how long the test lasts. Under the light or heavy system load, five million loops (RTC clock interrupts) are run on the preemptive kernel.

Figure 7.1 shows the comparison of response latency distribution on these two kernels under light system load. Table 7.1 presents the details of the experimental results of response latency. This test is conducted under light system load. It only runs with the necessary daemon process such as file system management, process management and so on. The clock frequency is set to 1024Hz. In table 7.1 and 7.2, the samples < 0.1ms is given by the percentage of occurrence with latencies less then 0.1ms in the observation of 5000,000 samples.

After filtering the samples that have the latency over 0.2ms, the response latency is shown. In this figure, the blue line gives the pair of amount of samples and response latency of standard Linux; while the pink line gives that of RTS-Linux. It shows the occurrence of response over 0.2ms is largely reduced. It verifies RTS-Linux has a better performance with reduced response latencies. This is because that more preemption points are inserted in the task scheduling and long latency cases. In these points, the preemption condition is checked constantly. As a result, the short response latency is issued.

The statistic in Table 7.1 shows that in standard Linux, a majority latency happens at [0.1, 0.2) (99.82%) and a majority latency happens at (0. 0.1) (98.45%) in RTS-Linux. This is because the preemption points make the real-time system have a better response

accuracy. As we can see from table 7.1, a shorter mean latency is caught in RTS-Linux kernel. Compared with the experimental result of standard Linux, the number of samples whose latency less than 0.5ms in RTS-Linux is larger.

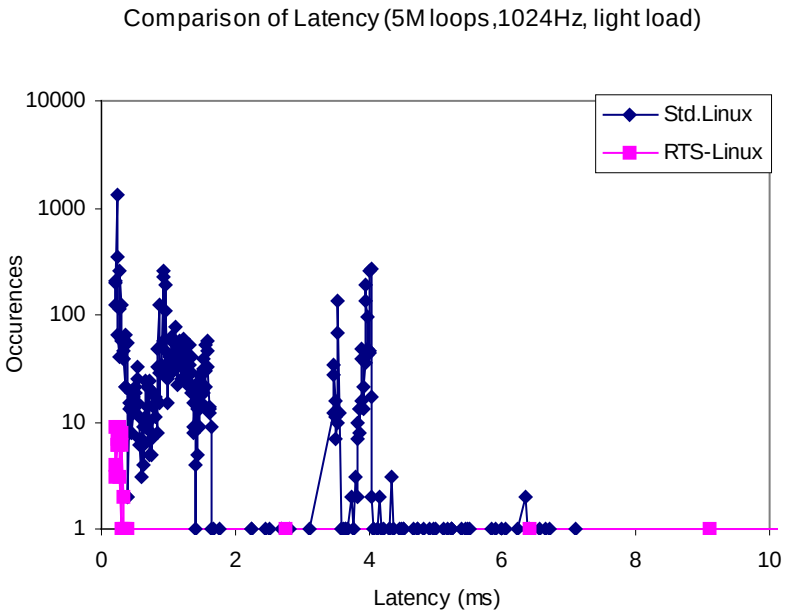


Figure 7.1 Response Latency (light load) [7]

Table 7.1 Response Latency (without load)

Kernel	Std. Kernel	RTS-Linux Kernel
Maximum Latency (ms)	11.88	39.26
Mean Latency (ms) [8]	0.0470	0.0438
Std. Deviation	0.0067	0.0005
P (samples < 0.1ms)	0.000020%	98.45%
P (samples < 0.2ms)	99.82%	~100.00%
P (samples < 15ms)	100.00%	100.00%

[7] Occurrence is amount of samples whose response latencies located in a specified latency scope. In this figure, only the samples with RL_i > 0.2ms are shown. It is because the occurrence (RL_i<= 0.2ms) is very large compared with the occurrence that the occurrence(RL_i > 0.2ms). If the distribution of response latency at each observation is shown, the comparison of this two scheduling is difficult to observe the difference. Thus we accumulate the occurrence in the latency scope. In figure 7.1, the scope is (0, 0.01), [0.01, 0.02),[0.02, 0.03),…….

[8] In this experiment, the observation of samples is accumulated within the scopes with a devisor to be 0.01ms. Because the mass latency in the scopes (0, 0.01) is computed as 0, the observation does not have obvious difference in mean latency. In table 7.1 and 7.2, P (samples < 0.1ms) is given by the percentage of occurrence with latencies less then 0.1ms in the observation of 5000,000 samples. Others are defined accordingly.

In order to reveal the response latency under heavy system load, previous test is repeated. In this test, a media player is executed on the background testing environment. The real-time clock frequency is set to 2048 Hz. Figure 7.2 shows the response latency distribution on the standard Linux kernel and RTS-Linux kernel under stressful system load. Table 7.2 presents the details of experimental result. It shows a reduction in mean response latency. The maximum latency is much less in the RTS-Linux kernel than that in standard Linux kernel.

When looking at the above results, it was obvious that the response latency distribution shows a big difference between RTS-Linux and the standard Linux. More samples in RTS-Linux are observed in the short response latency. This optimization is easy to be observed under the stress system load. It proves our preemptive kernel has a shorter response latencies compared with standard Linux.

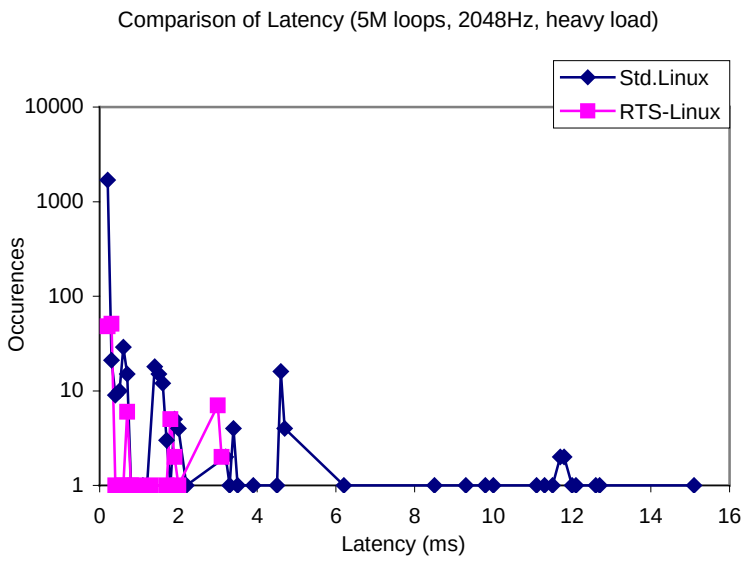


Figure 7.2 Response Latency (Stress Load) [9]

[9] In this figure, only the samples (RL_i > 0.1ms) are shown. It is because the occurrence (RL_i<= 0.1ms) is very large compared with the occurrence that the occurrence (RL_i > 0.1ms). If each observation scope is shown, it is difficult to observe the different between these two scheduling because the large difference of scale. Thus we accumulate the occurrence in the latency scope. In figure 7.1, the scope is (0, 0.1), [0.1, 0.2),[0.2, 0.3),……

Table 7.2 Response Latency (Stress Load)

Kernel	Std. Kernel	RTS-Linux Kernel
Maximum Latency (ms)	15.1	3.1
Mean Latency (ms) [10]	0.000679	0.000077
Std. Deviation	0.0254	0.0054
P (samples < 0.1ms)	99.96%	99.99%
P (samples < 0.2ms)	99.99%	~100.00%
P (samples < 15.1ms)	100.00%	

We compare the experimental results under light system load and heavy system load. It is observed that the response latency is lower under heavy system load. There are two reasons lead to this result. In our experiments, the occurrence is accumulated in difference scopes of latency. In the test of light system load, the latency in the scope (0, 0.01ms) is recorded as 0 with a devisor to be 0.01ms. While it is 0.1ms in the test of heavy system load because the disperse distribution of response latency under system load. It causes the mean latency is very short in heavy system load. Thus only the comparison of mean latency (non-QM and QM) drawn on the same scale can reveal the response accuracy situation (purely in light system load or in heavy system load).

On the other hand, the occurrence of latency over 0.1ms in the test of heavy system load is larger than that of light system load. The explanation of this observation is the high timing granularity leads to the reduction of the response latency. When the timing granularity is increased, the processor time assigned to each process is decreased and the kernel scheduler is called up more frequently. Thus a process with higher priority is easier to allocate the shared system resource.

[10] In this experiment, the divisor of observation scope is 0.1ms. Because the latencies less than 0.1 ms are recorded as 0, the mean latency only considers the latency scope ($RL_i > 0.1$ms). Thus mean value is much less than that of light load. So the comparison of mean latency can only be applied on the comparison of Standard Linux and RTS-Linux Kernel under same situation. The significant digit of mean latency is because the large observation and many observation happens within (0, 0.1ms).

7.2 Real-Time Scheduling Paradigm

In order to validate the performance of real-time scheduling paradigm, a set of experiments are conducted. These experimental results reveal how the real-time tasks are scheduled under different policies. Section 7.2.1 shows the examples of three common-used scheduling algorithm. Some examples of aperiodic task scheduling are presented in section 7.2.2. The flexible scheduling framework is also explored in section 7.2.3.

7.2.1 Task Scheduling of RM/EDF/MLF

Some real-time applications are executed to verify the task scheduling in RTS-Linux. Using these applications, some experimental results can be obtained to show the real-time task execution. It is transformed into some diagrams to reflect how the real-time tasks are activated and executed. The procedure of application is presented in List 7.1.

List 7.1 Procedure of Real-time Application

1. Initialize the real-time task scheduling discipline and set the scheduling policies;
2. Create child real-time process;
3. Initialize the real-time tasks with the timing attributes;
4. At the end of life time, kill all the processes, exit;

In the experiment, a set of tasks are executed in the system. The Real-time scheduler deploys all the tasks using various schedule policies. The timing attributes of the task set are presented in Table 7.3.a based on the assumption that the deadline of a task deadline equals to its period.

Presently in RTS-Linux, RTS scheduler can make use of three scheduling policies: RM, EDF and MLF. Figures 7.3-7.5 show the scheduling paradigms of RM, EDF and MLF policy under the normal scheduling. Figures[11] 7.3-7.8 reveal the scheduling paradigm of RM, EDF and MLF policy with QM configured. In these figures, the scheduling paradigm of the policy without QM is similar to that of the policy configured with QM. In order to investigate the difference between them, a comparison of task preemption and schedule jitter versus utilization is also investigated in section 7.4.

The example of tasks scheduling shows that the real-time task scheduling has been implemented in the RTS-Linux system. Using a certain scheduling policy, the real-time tasks can be scheduled as the scheduling disciplines expected. In above figures, we show the scheduling paradigm in various scheduling policies as well as the policies cooperating with queue manager (QM). Queue manager (QM) has been introduced in chapter 6. It makes use of a queue to keep the outstanding task and all the running tasks. When the task changes its state and plan to call up scheduling, queue manager (QM) will update the outstanding task and the queue of running tasks. Only when the outstanding task is not current running task, the scheduler is called up.

Table 7.3 Timing Attributes Of Real-time Task Set

a) Periodic tasks

Task (i)	Ci	Di
Task 1	14	70
Task 2	5	90
Task 3	5	100
Task 4	10	110

b) Acyclic Execution

Task (i)	Ci	Di
Task 1	14	70
Task 2	5	90
Task 3	5	100
Task 4	10	110
Task 5 (AP_SCHED)	10+*i*	/

i: stands for the index of aperiodic task.

[11] In Figures 7.3-7.11, the examples of real-time tasks execution that is scheduled under specified policies are shown. In the diagrams, the first row represents the Linux standard process, and the other tasks are task 1, task 2, task 3, task 4 and task 5 (If any). And task 5 is specified for Acyclic Execution.

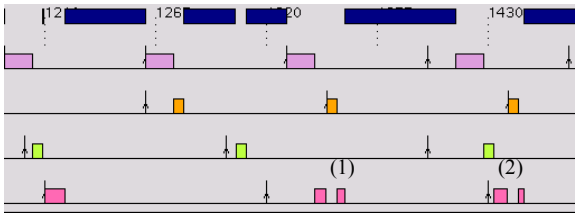


Figure 7.3 Task Execution
(RM scheduling, without QM)

In the cases (1) and (2), task 4 has a lower execution rate, so a higher-priority task (task 2) pre-empts it.

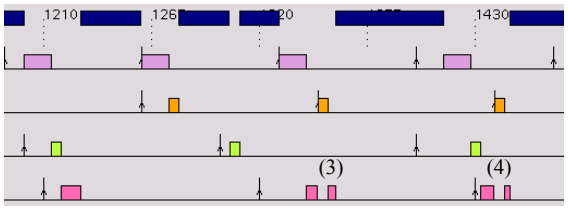


Figure 7.4 Task Execution
(RM scheduling, with QM)

In the cases (3) and (4), task 4 has a lower execution rate, so a higher-priority task (task 2) preempts it.

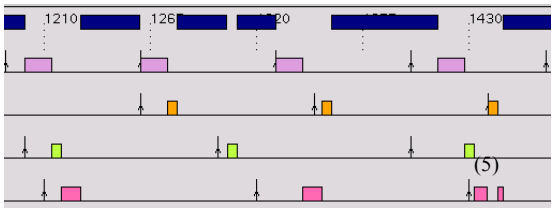


Figure 7.5 Task Execution
(EDF scheduling, without QM)

In the case (5), as task 4 has a longer deadline, a higher-priority task (task 2) preempts it.

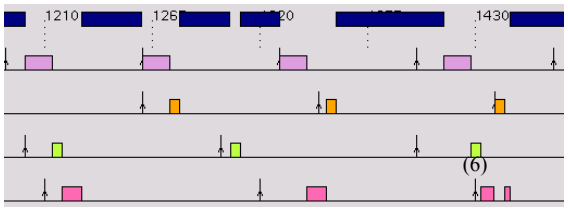


Figure 7.7 Task Execution
(EDF scheduling, with QM)

In the case (6), as task 4 has a longer deadline, a higher-priority task (task 2) preempts it.

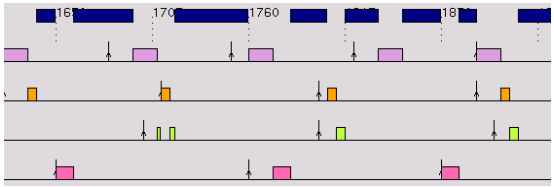


Figure 7.6 Task Execution
(MLF scheduling, without QM)

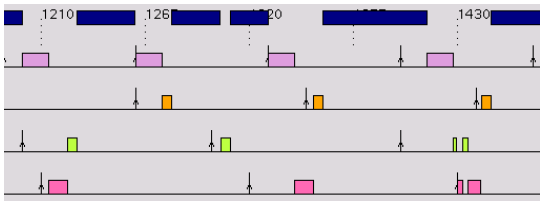


Figure 7.8 Task Execution
(MLF scheduling, with QM)

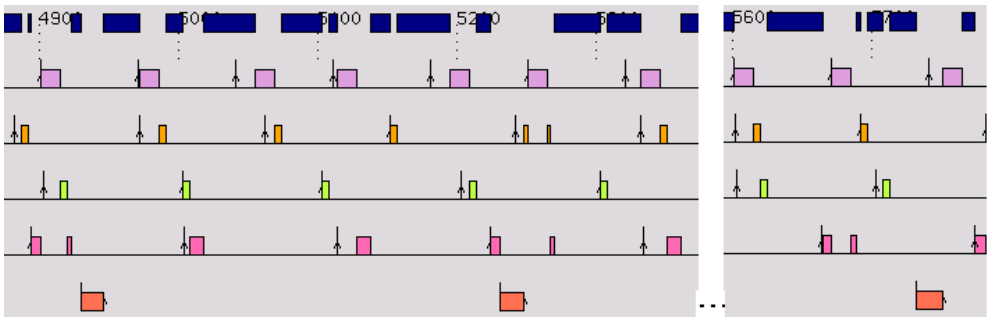
Compared with the task scheduling of rate monotonic (RM) with and without queue manager (QM) facility, the task execution is similar. It is because the schedule overhead is less compared with the execution time. Therefore, the reduction of schedule overhead is difficult to show in the graphic scheduling paradigm though queue manager (QM) helps to reduce the schedule overhead. These examples can only be used to show the scheduling disciplines and task preemption in various scheduling polices.

7.2.2 Acyclic Execution

Acyclic tasks scheduling is also supported in RTS scheduler. Some experiments are conducted to reflect how the aperiodic tasks are activated and executed. In the experiment, a set of tasks are executed in the system. Then an aperiodic task comes into the system randomly. The real-time scheduler deploys all the tasks using various schedule policies. In the test, the timing attributes of periodic task set is set as Table 7.3b; and the load of aperiodic tasks is increasing with incremental being 1.

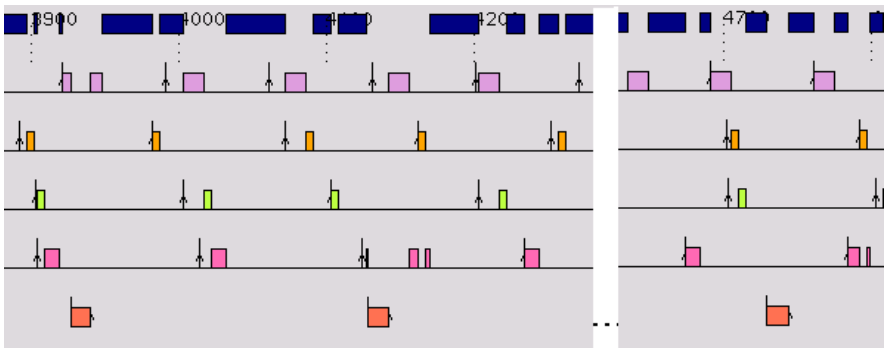
Figure 7.9 reveals the scheduling paradigm of the aperiodic tasks holding a higher priority over other real-time tasks. Three policies are investigated: RM, EDF and MLF. As the scheduling paradigm under schedule policy with QM configured is very similar with that under normal task scheduling, these diagrams of tests configure with QM are not shown here. The flexible sporadic server (FSS) has been implemented in RTS-Linux. The experimental result is shown in figure 5.2 of section 5.5.

In this test of aperiodic task execution, the aperiodic task works like a real-time task that served with FIFS policy in Linux. A long task execution may block other real-time tasks and cause a missing of deadline. Thus such a simple scheduling can be applied in the real-time control that do not have strict real-time requirement. For a hybrid system including periodic tasks as well as aperiodic task, there must be some more comprehensive algorithms to realize the real-time control.



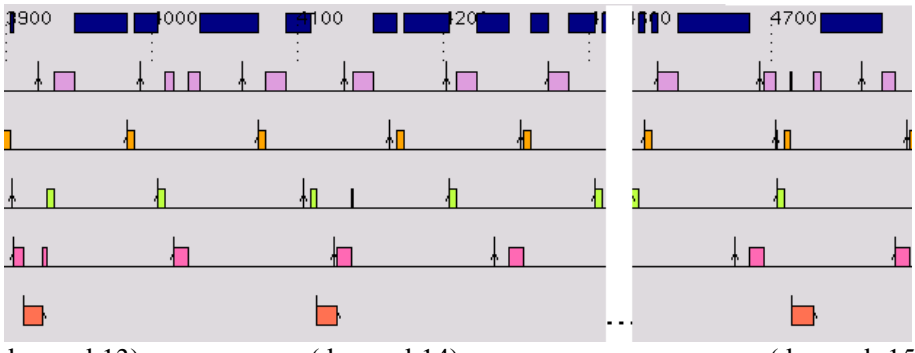
(demand:16) (demand:17) (demand: 18)

(a) Acyclic Task Execution of RM



(demand:13) (demand:14) (demand: 15)

(b) Acyclic Task Execution of EDF



(demand:13) (demand:14) (demand: 15)

(c) Acyclic Task Execution of MLF

Figure 7.9 Scheduling Paradigm of Acyclic Task Execution

7.2.3 Performance of Flexible Scheduling Framework

The UDS scheduler provides a flexible scheduling framework for the application developer. This facility allows the developers to write the customized scheduling algorithms. Using the UDS framework, two scheduling disciplines (RM static task scheduling and MLF scheduling) have been implemented. Figures 7.9 and 7.10 show how the tasks are executed under these scheduling disciplines.

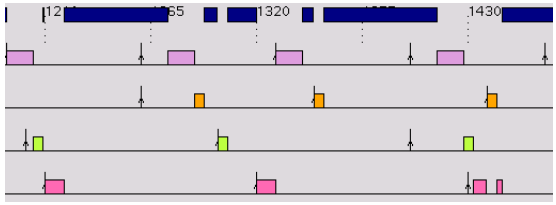


Figure 7.10 Static Task Scheduling in UDS framework

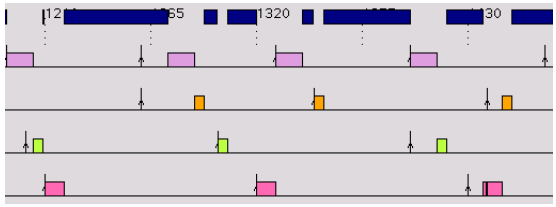


Figure 7.11 Dynamic Task Scheduling (MLF) in UDS Framework

Under the priority-driven scheduling, the priority increases with the execution rate increasing. So it is comparable to RM schedule policy pre-built in RTS driver. For MLF scheduling, we can draw a similar comparison of MLF in UDS scheduler and RTS scheduler. Compared with the scheduling paradigm in Figures 7.10 and 7.11, there is a slight difference between the flexible scheduling scheme and the scheme built in RTS driver. In a comparison of the original data of execution time generating the diagrams, a longer response latency is found on the flexible scheduling framework. As the delayed is caused by the famous loadable module mechanism in Linux system,

the evaluation is not investigated here. The more complex routing of scheduler generates the longer schedule overhead. A more complex routing in the user-defined framework makes the schedule overhead increased. Applying such a framework, the efficiency is decreased with the payback of compatibility. This also proves that a good routing, scheduling algorithm and code optimization may help to reduce the schedule overhead.

7.3 Results of Schedule Precision

Scheduling Precision is an important metric that measures the accuracy and predictability of real-time scheduling. It reveals how exactly a real-time task is execution as it predicted. The scheduling jitter depends on how often the scheduler is called and the granularity of timer. Scheduling jitter is investigated for the real-time scheduling of a periodic real-time task. Queue manager (QM) mechanism targets on reducing the computation complexity and improving the response latency. We analyze the computation complexity of schedule overhead and its affect on the response accuracy in section 6.5. Base on the above analysis, we conduct some experiments to measure the response latency and schedule jitter in Linux system.

7.3.1 Schedule Precision in FIFS and Priority-Driven Scheduling

Rhealstone Benchmark is proposed by Kar and Porter [13] to investigate the real-time performance of RTOSs. Based on this proposal, we build a benchmark application to

measure the response latency and schedule overhead with queue manager (QM). In this application, multiple system calls are submitted to the system to measure the response time on FIFS schedule policy (real-time). For the purpose of comparison, some tests are conducted within two cases: standard Linux system and FIFS in RTS-Linux with queue manager (QM). These tests are executed on the background environment of a media player. Here we show the experimental results of response latency obtained in Linux system.

Figure 7.12 shows a comparison of response latency distribution. Figure 7.12(a) shows the distribution of response latency under three cases. In standard Linux task scheduling assigned with execution budget, 8.27% of the samples have latencies over 0.05ms; while only 3.36% of the samples with latencies over 0.05 ms in multi-process scheduling with QM. It is shown the response latency of multi-process scheduling is longer than that of QM-supported schedule policy. For FIFS schedule policy, the mean response latency is 0.047 ms; for the normal timing sharing schedule policy, the mean response latency is 0.130 ms, for the QM-supported timing sharing schedule policy, the mean response latency is 0.055ms (refer to Table 7.4). Compared with the normal RM scheduling, the decrement of response latency of QM-supported RM scheduling is 57.7%. This comparison shows that Queue manager mechanism has a good optimization on the response latency. The main reason of the optimization is the short schedule overhead in queue manager (QM) mechanism.

As can be seen in the figure and table, queue manager (QM) has a larger decrement in priority-driven scheduling than that in FIFS scheduling and. The main reason is that

FIFS schedule policy has a privilege level over the other processes, the schedule will serve it as an outstanding task other than exploring the whole task queue. As computation complexity is less in FIFS scheduling, the optimization on response latency is not so obvious.

Table 7.4 Response Latency in Priority-Driven Scheduling

	Scheduler (non-QM)	QM-supported	Difference
Mean	0.13	0.055	57.7%
Deviation	0.16	0.018	88.6%
Relative Deviation	1.22	0.329	73.0%
Standard Deviation	1.11	0.130	88.4%

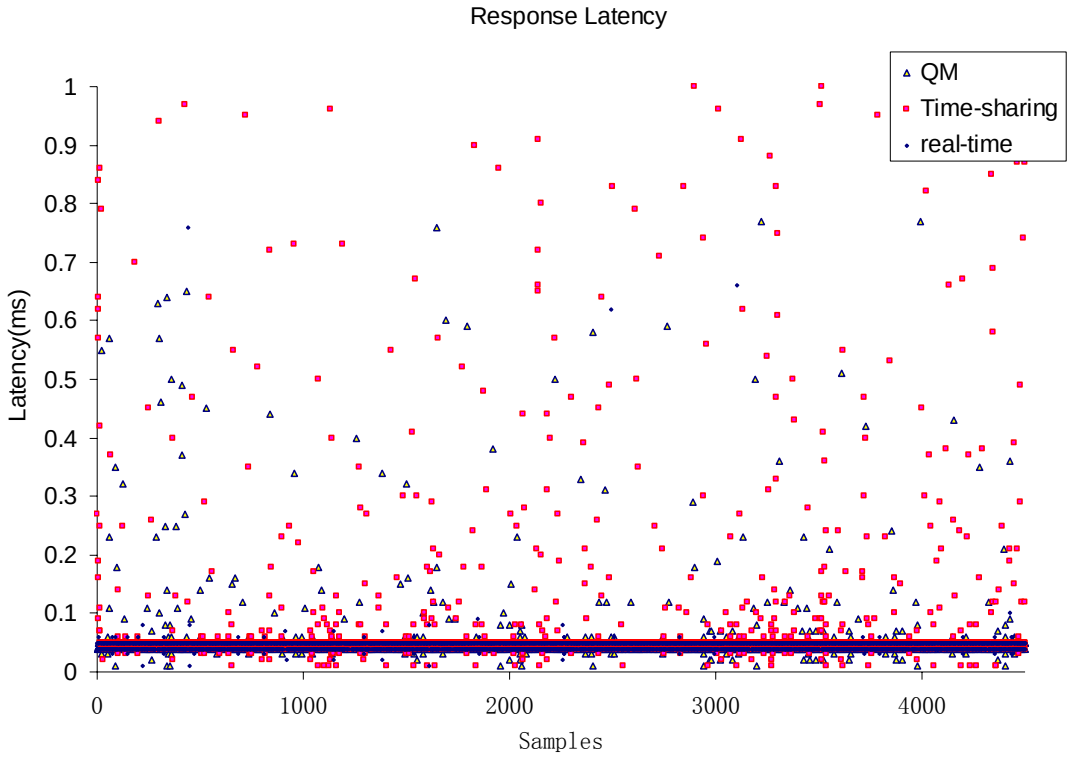
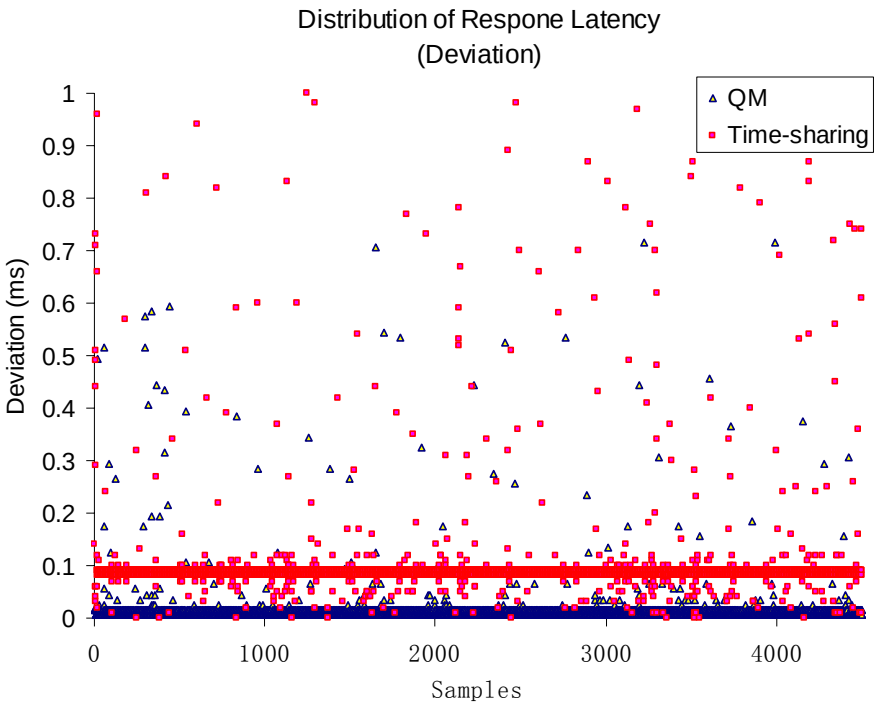
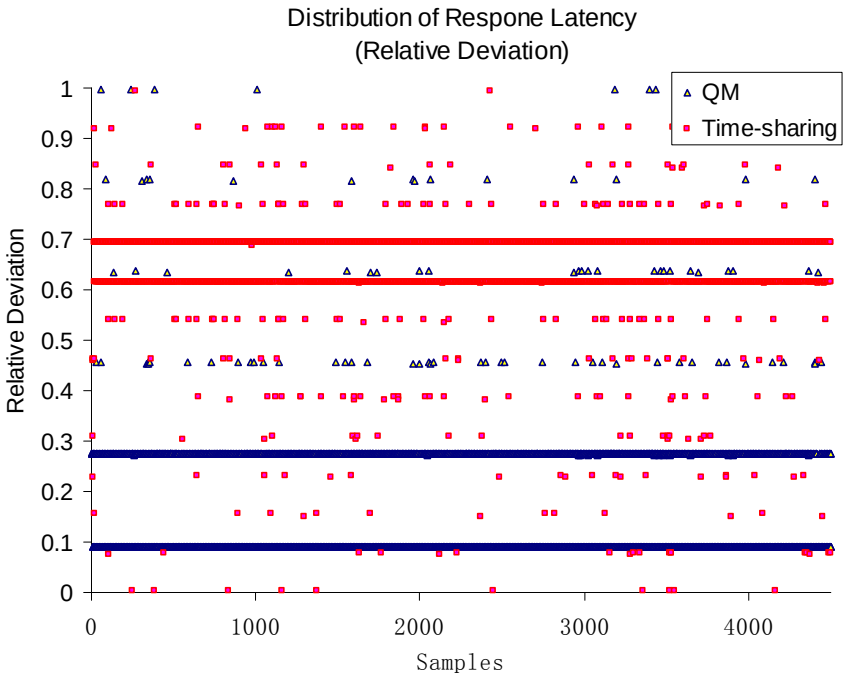


Figure 7.12 (a) Response Latency [12]

[12] In this figure, the serial (QM) represents the task scheduling with priority driven and QM supported, the serial (time-sharing) represents the task scheduling with priority-driven scheduling; and the serial (real-time) represents the task scheduling with FIFS scheduling.



(b) Deviation of Response Latency



(c) Relative Deviation

Figure 7.12 Distribution of Response Latency

As the deviation and relative deviation of the response latency help to evaluate the overall performance, these observations are shown in Figure 7.12b and 7.12c. Because the affect of Queue manager (QM) mechanism in priority-driven scheduling is obvious to observe, only the cases of priority-driven scheduling and priority-driven with Queue manager (QM) supported are presented. From the absolute deviation to mean in Figure 7.12 (b), the mess samples is in scope [0.04ms, 0.06ms) for priority-driven scheduling; while the mess sample in scope [0.05ms, 0.07ms) for priority-driven task scheduling with QM. The experimental results imply that Queue manager (QM) sharply reduces the response latency in priority-driven scheduling. Figure 7.3c shows the relative deviation having mass records at two values, which is observed as two horizontal lines. This is because the scale of latency measurement is 0.01ms. Then the latencies in scope [0.04ms, 0.05) are regarded as 0.04ms. Thus the mass occurrence of latency in priority-driven (QM) is recorded as 0.04ms and 0.05ms.

7.3.2 Schedule Jitter in Real-Time Scheduling

Some experiments are conducted to verify the real-time design and demonstrate the practicality of the design. The example coding is introduced in Appendix B. In this experiment, we create a set of periodic real-time tasks with different scheduling policies to investigate schedule jitter the real-time execution. When there are some tasks with a constant period in the system, some time interval happens between the actual execution time and their expected execution time. This deviation is known as Scheduling Jitter. The scheduling jitter can show how exactly the real-time task is executed in the expected time.

In this test, three tasks are created and executed in the system with different timing attributes (shown in Table 7.5). In this table, *load* denotes as the utilization of real-time task set. Schedule jitter is measured using the deviation of the desired timing and the actual timing. In order to stress the system at different levels, we run these experiments with utilization increasing from 10% to 40%. The execution of test lasts 50 minutes for each case.

Table 7.5 Timing Attributes Of Real-time Task Set (Various Load)

Task (i)	Ci	Di
Task 1	1**load**100	500
Task 2	2**load**100	590
Task 3	3**load**100	650

Figures 7.13-7.15 are the comparison of schedule jitter under the real-time control with and without Queue Manager (QM) supported (utilization: 10%). From the comparison drawn in a specific scheduling policy, it can be observed that queue manager (QM) can reduce the schedule jitter more or less. Under RM, EDF and MLF schedule policy, the percentage of reduction of Schedule Jitter are 35.77%, 21.94% and 9.47 %. The optimization is more obvious in rate monotonic (RM) scheduling algorithm. It is because that the computation complexity in dynamic scheduling (EDF and MLF) is higher than that of static scheduling. Under these two scheduling algorithms, even with Queue Manager (QM), the computation complexity cannot be reduced as much as that of static scheduling.

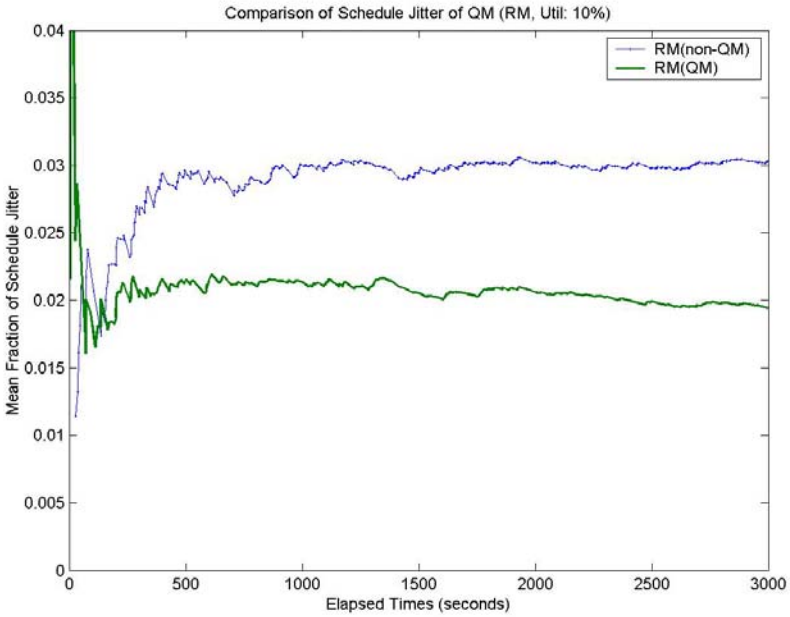


Figure 7.13 Schedule Jitter of QM (RM policy)

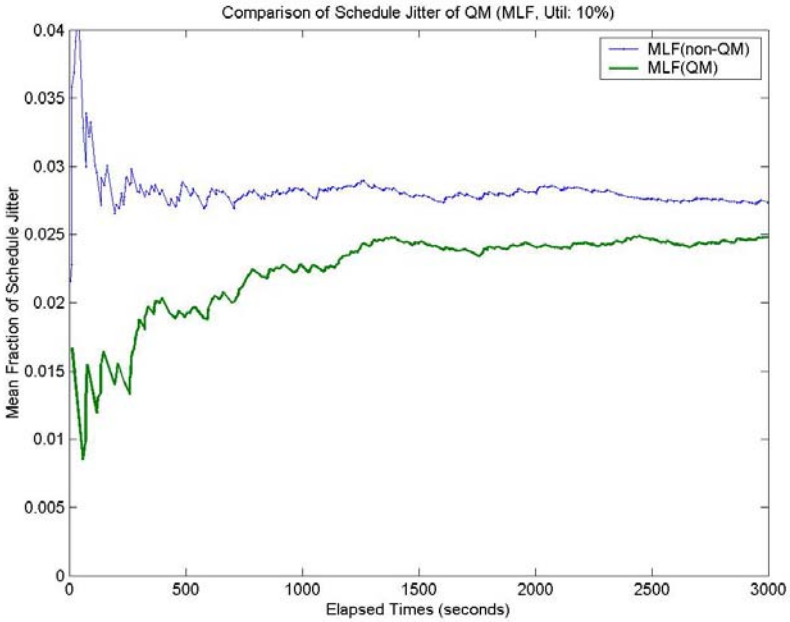


Figure 7.14 Schedule Jitter of QM (MLF policy)

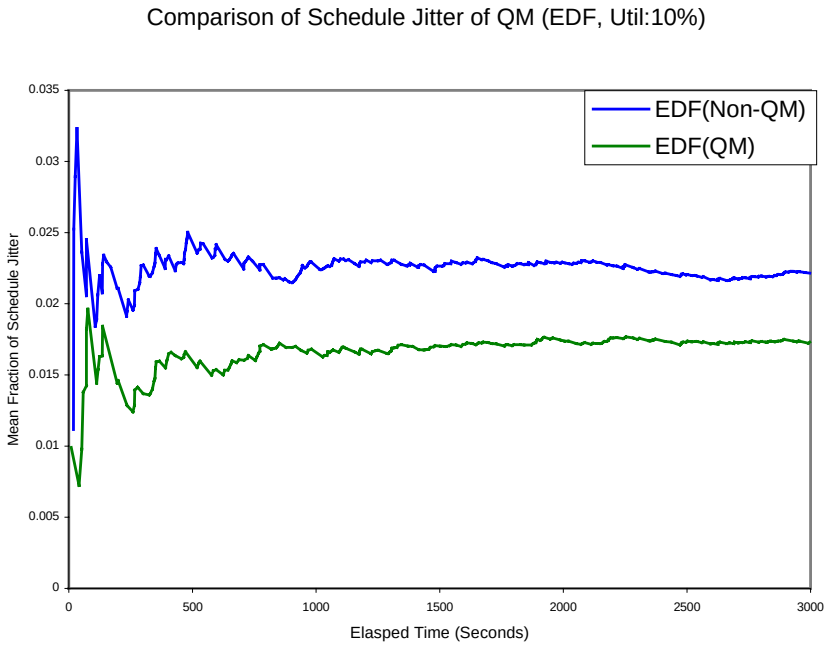


Figure 7.15 Schedule Jitter of QM (EDF policy)

Table 7.6 is the overview of schedule jitter on RM, EDF and MLF schedule policy. We make the following observation. Under the same schedule policy and utilization of real-time tasks, the queue manager reduces the schedule jitter more or less.

Table 7.6 Schedule Jitter[13]

Fraction of Utilization	RM			EDF			MLF		
	RTS (non-QM)	QM	IMP.[14]	RTS (non-QM)	QM	IMP.	RTS (non-QM)	QM	IMP.
10%	0.030	0.019	35.8%	0.022	0.017	21.92%	0.027	0.025	9.47%
20%	0.067	0.066	2.24%	0.072	0.066	8.88%	0.072	0.071	1.66%
30%	0.120	0.117	2.50%	0.117	0.112	4.50%	0.124	0.115	7.34%
40%	0.127	0.120	5.61%	0.124	0.121	2.36%	0.127	0.127	0.30%

[13] In this table, RTS (non-QM) stands for the usual task scheduling real-time scheduling. Respecting to the queue manager facility, this task scheduling did not use queue manager in the task scheduling.
[14] IMP.: Improvement Ratio of Schedule Precision.

$$IMP. = \frac{jitter_{non-QM} - jitter_{(QM)}}{jitter_{non-QM}}$$

It is observed that the schedule jitter increases with the system load increasing. It shows that the optimization fraction arose by Queue Manager (QM) is not linearly increasing with the load. It is because the real-time scheduling is a complex task scheduling. The task scheduling is not only correlated with the system load, but also the task set in terms of their timing attributes.

The increasing slope is larger when the system load is higher. It is because the kernel schedule has less laxity to do the task scheduling of other processes at the light system load. Thus the tasks waiting for the processor resource are more than that of light system load. It explains the increasing slope of schedule jitter at heavy system load is higher.

We evaluate the system performance in terms of schedule jitter in the task scheduling inherited from Linux system. The experimental results show that the queue manager mechanism can improve the timing response accuracy and balance the response on heavy and light system load. We also investigated the schedule jitter of three scheduling algorithms: RM, EDF and MLF. In comparison with schedule precision on various system load and multiple real-time scheduling algorithms, we conclude that queue manager can optimize the response accuracy more or less.

7.4 Other Evaluations of Real-Time System

To evaluate the real-time control properties, the probability of task preemption and missing deadline is also very important in the performance metrics. In the preemptive kernel, the kernel allows one executing process to be preempted by another process that holds higher priority. The occurrence of task preemption is recorded to reveal how often the scheduler is called, thus this metric is also investigated in the experiment. Different scheduling discipline takes different effect on the frequency of task preemption.

Figure 7.16 shows the comparison of pre-emption times among various system loads. Table 7.7 details of the happening of task preemption times on RM, EDF and MLF scheduling. The real-time scheduler and a scheduler with queue manger are both investigated. We make the following observation. Respecting to RM and MLF, the occurrence of task preemption increases with the increasing utilization. While for EDF policy, the occurrence of task preemption decrease when utilization increases from 30% to 40%. This only shows that the preemption is not monotonically increasing in EDF scheduling. It is difficult to draw some conclusions with the individual observation of task preemption. Thus we investigate the pair of preemption times and schedule jitter (Figure 7.17). The schedule jitter is detailed in Table 7.6.

We observe that the schedule jitter increases with the frequency of task preemption increasing. This can be explained by a frequent task preemption leads to frequent

calling up of task schedule. Thus the schedule overhead is increased, which makes the average schedule jitter increasing.

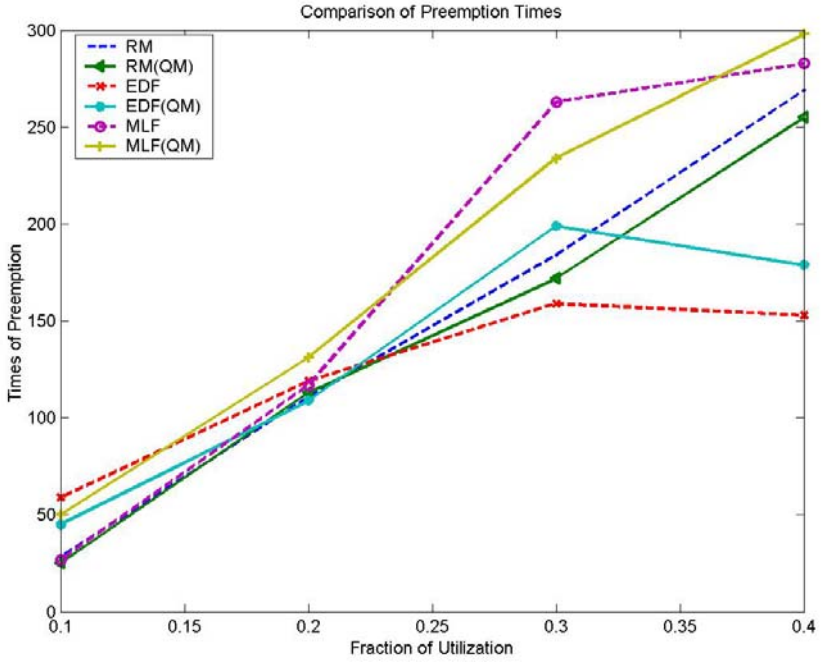
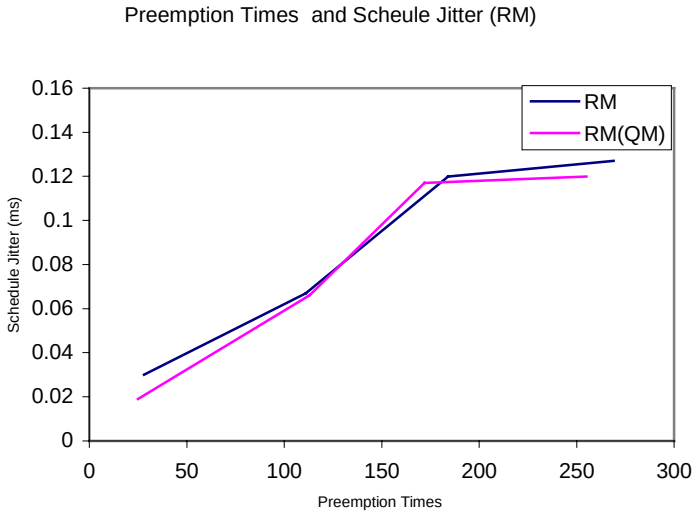


Figure 7.16 Overview of Task Preemption [15]

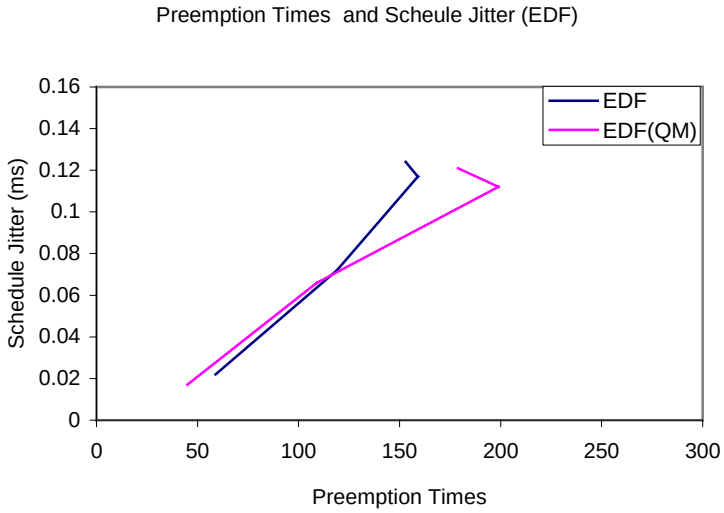
Table 7.7 Occurrence of Task Preemption

Fraction of Utilization	RM		EDF		MLF	
	Non-QM	QM	Non-QM	QM	Non-QM	QM
10%	28	25	59	45	27	50
20%	111	113	119	109	117	131
30%	184	172	159	199	263	234
40%	269	255	153	179	283	298

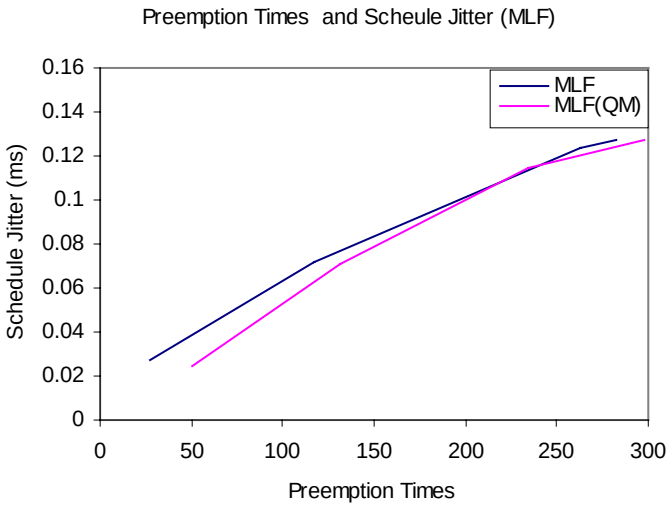
[15] In Figure 7.16, Times of Preemption stands for the occurrence of task preemption, that is the times that task preemption is invoked.



(a) RM



(b) EDF



(c) MLF

Figure 7.17 Preemption Times and Schedule Jitter

7.4.1 Missing Deadline

On the other hand, the missing deadline is also an important metric of real-time system. The missing deadline means the failure of system in a hard real-time system. It is very important to investigate how the missing-deadline is reduced in a good real-time system. The miss rate of specified schedule policy is also recorded in the experiment of schedule precision. Table 7.8 is the comparison of miss rate. In order to review how the timing requirement is met, the time when the missing deadline happens is also recorded. Figure 19 shows the correlation between missing deadline and time.

Table 7.8 Occurrence of Missing Deadline

Fraction of Utilization	RM		EDF		MLF	
	RTS(non-QM)	QM	RTS(non-QM)	QM	RTS(non-QM)	QM
10%	0	0	3	2	1	3
20%	0	0	3	3	3	3
30%	0	0	2	2	3	2
40%	0	0	2	2	1	2

Figure 7.19 shows that the missing-deadline happens only when the whole real-time task set is generated in the system. After the real-time execution becomes stable, the missing deadline does not happen. Tt is because there is an admission control to test the schedulability condition in RTS-Linux. Only after the real-time task set passes the test of schedulability condition, the new task can be initialized in the system. We apply the admission controller of real-time tasks. Thus the real-time requirement could be satisfied in RTS-Linux.

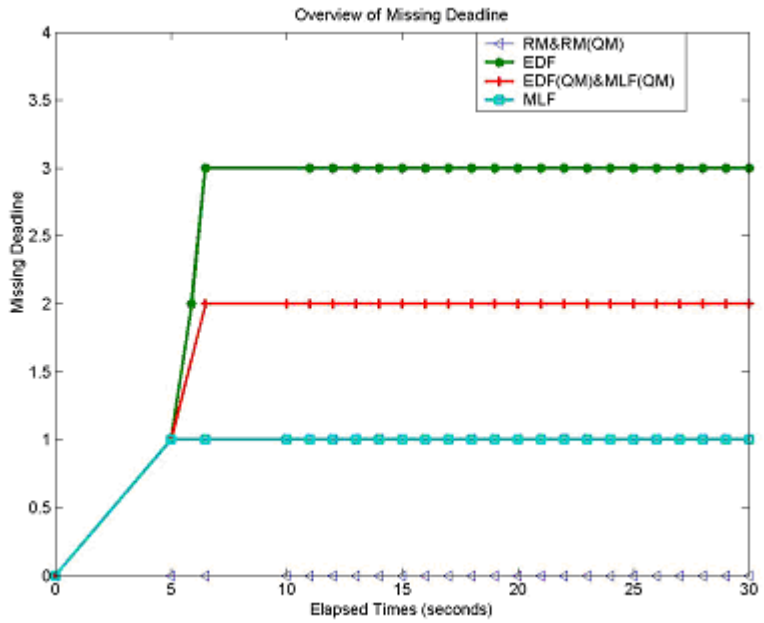


Figure 7.18 Missing Deadlines

In the experiments, the utilization of task is designed to meet the schedule condition. As only the tasks within processor's capacity are allowed to allocate the processor, the goal of meeting deadline can be met as we assumed.

7.5 Discussion and Conclusion

The functionality of RTS-Linux is tested by multiple experiments of performance evaluation. All the experiment is toggled on iPAQ. Implementing a system such as RTS-Linux that targets on the embedded hardware is more difficult than that on the i386 platform. The processing of debugging the kernel in the embedded system is complex and difficult. The resource such as the documents and cross-compiler tool-chains for iPAQ is limited. This situation makes the development a time-consuming work.

In the experiments of system performance evaluation, we examine the response timing accuracy, scheduling paradigm and other scheduling metrics. The evaluation leads to five primary conclusions:

- The basic functionality of standard Linux can work on RTS-Linux
- The experimental result shows that reduced response latency is achieved in RTS-Linux compared with the standard Linux.
- The scheduling policies based on some common-used algorithms are implemented on the system.
- Acyclic Execution is also supported in RTS-Linux.
- The component of Queue Manager optimizes the scheduling precision at the price of increasing occurrence of task preemption.
- The UDS scheduling framework works effectively and presents an open interface for developer to build flexible scheduling.

The experiments show that RTS-Linux improves the timing accuracy and enhances the real-time property of the standard Linux. The facilities including queue manager (QM), flexible scheduling framework and admission control enhance the performance of the real-time control. These facilities improve the schedule precision and the compatibility.

CHAPTER 8
CONCLUSIONS AND FUTURE WORK

This chapter summarizes the research work. Section 8.1 presents the main conclusions and contributions of the research work. Section 8.2 introduces some related fields that could be explored in the future.

8.1 Conclusions and Contributions

This study provides a solution to enhance the real-time properties and satisfy real-time control in Linux system. In RTS-Linux, a preemptive kernel and a real-time control module cooperate to satisfy the timing requirements of the applications. The modified platform is called RTS-Linux. In RTS-Linux, a certain fraction of processor utilization can be assigned to real-time tasks. Real-time tasks share all the primitives in the standard Linux kernel, thus RTS-Linux have the capability to access the full range of facilities in standard Linux. In the real-time control, some common-used scheduling policies are built in the RTS scheduler. The policies make use of rate monotonic (RM), earliest deadline first (EDF) and minimum deadline first (MLF) algorithms. Queue Manager (QM) mechanism is proposed to reduce the response latency of operating system.

In this thesis, we propose the hybrid process model based on cyclic process model. In this process model, a real-time task has the timing attributes to define its life cycle

duration. The schedulability analysis is conducted in the process model and provides a theoretical base for the task scheduling of aperiodic tasks. This study motivates the development of flexible sporadic server (FSS) algorithm. The flexible sporadic server (FSS) algorithm is applied to schedule aperiodic tasks in the hybrid process model. We have proposed a simplified approach for worst-case response time (WCRT) prediction. This approach decreases the computation complexity of worst-case response time (WCRT).

8.1.1 Hybrid Process Model

The schedulability analysis and formulation of response time of periodic tasks are reviewed. Based on the review, a hybrid process model is presented for the task scheduling of aperiodic tasks. The schedulability analysis in the process model serves as a theoretical base for the proposal of Flexible Sporadic Server (FSS) algorithm.

8.1.2 Response Time Prediction

We propose a simplified approach of worst-case response time (WCRT) prediction is a. For an identical task in a process model, a subset of the task and tasks holding higher priorities is investigated. In this subset, it is found the time when an idle time begins to increase. This moment also denotes the finish time in the current invocation. Then the response time can be computed from the time and invocation starting-time. From all the possible value of the response time, the worst-case response time (WCRT) can be

predicted. Obviously, compared with the computation of the execution demand at each invocation, this approach decreases the computation complexity.

8.1.3 Flexible Sporadic Server (FSS)

The experimental results of acyclic execution show that aperiodic task can be executed at the background environment of cyclic execution in this hybrid system. Besides the FIFS policy for aperiodic task, Flexible Sporadic Server (FSS) algorithm is applied to deploy the acyclic task execution. Some parameters such as execution budget and period of aperiodic server can be determined according to schedulability analysis of hybrid process model. Given a bound of utilization, the period of FSS server can be set with the prediction of execution budget. The bound is determined according to the schedulability of the hybrid process model. With some suitable settings of aperiodic server, an aperiodic task is schedulable in the hybrid operating system. The experimental result depicted in section 5.5 indicates that Flexible Sporadic Server (FSS) reduces the queue length and average waiting time of acyclic task.

8.1.4 Queue Manager Mechanism

The response accuracy of system is investigated in terms of response latency and schedule jitter. The experimental results of response latency show that a good optimization of timing response is achieved on either light system load or heavy system load. It is because more task preemption is triggered in the preemptive kernel and reduces the scheduling overhead caused by sharing resources and deadlock. These experimental results show the comparison of schedule jitter under various utilizations

and prove that QM scheduling mechanism optimized schedule precision. It can be explained by the fact that the reduction of task scheduling computation complexity contributes to the reduction of scheduling overhead.

8.1.5 Flexible Scheduling Framework:

The Flexible Scheduling Framework, User Defined Scheduler (UDS), is an API that enables applications to write and use their own scheduling scheme in a way compatible with RTS interface. UDS scheduler is implemented as a loadable kernel module. The loadable modules are employed by Linux to add optional facilities to the kernel without recompiling/rebooting the system, and this new technology is extended to support the User-defined Scheduling (UDS) scheduler.

This study introduces the implementation of a hybrid real-time operating system: RTS-Linux. RTS-Linux provides a good platform for real-time applications. It demonstrates optimized timing response accuracy and realizes the real-time task scheduling. It also presents many important facilities such as admission control UDS scheduler to improve its utilities and make it convenient.

8.2 Recommendations for Future Work

This study has taken one step in establishing a hybrid real-time operating system. However, the scheduling policies built in the system are limited to only three corresponding commonly used scheduling algorithms. This study does not present real-

time control of continuous networking traffic flows as it concentrates mostly on the task scheduling and response latency of kernel process.

Some related topics could be investigated in the future. One of them is to apply the flexible sporadic server algorithm in the network communication. The second direction for the future work concerns modifying RTS-Linux to support the real-time executive in the multi-processor system. The scheduling deployed among a number of processors may be taken into account. The mechanism such as admission control and flexible scheduling framework also need to be improved in their efficiency. On the other hand, building a distributed computation system is a difficult problem to be considered in the future research. The fine-grained distributed computation invokes the interests of researchers on the performance of distributed simulation and distributed process control. Distributed data collection and process control have real-time constraints associated with their timing accuracy. Besides, multimedia conference applications have the real-time constrains correlated with QoS requirements. It would be interesting to extend real-time control to QoS and provide a framework for QoS guarantee in network traffic and a distributed real-time application.

Finally, RTS-Linux can be extended to support communication middleware between the real-time extension and the networking management of Linux. The multimedia and high-speed networks require a good timing response. Thus this hybrid system could be reinforced with building the communication layer to interface with the network services and applied protocols.

REFERENCE

[1] Krithi Ramamritham, John A. Stankovic, Scheduling Algorithms and Operating Systems Support for Real-Time Systems, Proceedings of the IEEE, 82(1), pp. 55-66, 1994.

[2] Michael Barr, Special Report: Choosing an RTOS, http://www.embedded.com/story/OEG20021212S0061, Embedded.com.

[3] Martin Timmeman. RTOS Market Survey Preliminary Result, 1999. http://www.realtime-info.be/magazine/99q1/1999q1_p006.pdf

[4] Wind River System. VxWorks 5.x, 2002, http://www.windriver.com/products/vxwroks5/index.html

[5] Wind River System. pSOSystm 3 Homepage, 2002, http://www.windriver.com/products/psosystem_3/index.html

[6] Boling D.: Programming Windows CE, Microsoft Press, 1998

[7] Dan Hidebrand. An architectural overview of QNX. In USENIX Workshop on Microkernels and Other Kernel Architectures, pp 113-126, Seattle, WA, April 1992.

[8] QNX Neutrino RTOS MICROKERNEL OPERATING SYSTEM, QNX Software System LTD, http://www.qnx.com.pl/pliki/Neutrino_2003_EN.pdf

[9] VRTX-The Operating System for System-on-Chip, Mentor Graphics, http://www.mentor.com/embedded/brochures/vrtx.pdf

[10] Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel, Chap. 10: Process Scheduling, Oreilly, Oct. 2000

[11] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. Communications of the ACM, 26(1):84--89, January 1983

[12] Michael Barabanov and Victor Yodaiken. Introducing Real-Time Linux. Linux journal, Issue 34, Feburary 1997.

[13] Kansas University. KURT: Real Time Linux, 1997. http://www.ittc.ku.edu.kurt

[14] Shui Oikawa and Raj Rajkumar, Linux/RK: A Portable Resource Kernel in Linux, IEEE Real-Time Systems Symposium Work-In-Progress, Madrid, December 1998.

[15] Shui Oikawa and R. Rajkumar, Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In Real-Time Technology and Application Symposium, Vancouver, Canada, June 1999.

[16] C.L. Liu and James W. Layland, Scheduling algorithms for multiprogramming in a hard real time environment, ACM, 20(1):46--61, 1973

[17] C.L. Liu, Fundamentals of real-time scheduling, Real-time Computing, pp.1-7, Springer-Vedag Publishers, 1994.

[18] Liu Sha, Mark H. Klein, and John Goodenough. Rate monotonic analysis for real-time systems. In Andre M. van Tilborg and Gary M. Koob. Editors, Foundations of Real-Time Computing: Scheduling and Resource Management, chapter 5, pp. 120-156, Kluwer Academic Publishers, 1991.

[19] Lubomir Bic, Alan C. Shaw, The logical design of operating systems, pp. 129-139, Englewood Cliffs, N.J. : Prentice-Hall , c1988

[20] J. P. Hong and X. Tan and Donald F. Towsley, A performance analysis of minimum laxity and earliest deadline scheduling in a real-time system. IEEE TRANSACTIONS ON COMPUTERS, 38(12):1736--1744, December 1989.

[21] Ghazalie, T. M. and Baker, T.P., Aperiodic servers in a deadline scheduling environment. Real-Time Systems. 9, pp. 31-67, 1995.

[22] Brinkley Sprint, Liu Sha and John Lehoczky. Aperiodic task scheduling for hard real-time systems. Journal of Real-Time Systems, 1, pp27-60, 1989.

[23] Michael Beck, Linux kernel internals, 2nd edition. Addision-Wesley. 1999.

[24] Barbeau M., The scheduler of Linux, http://www.scs.carleton.ca/~barbeau/Courses/SETP/ALP/schedulerofLinux.pdf

[25] MontaVista, 2000, RT-Scheduler and Preemption-Patch for the Linux Kernel, http://www.mvista.com/realtime

[26] Victor Yodaiken and Michael Barabanov, A Real-Time Unix, Proc. Linux Applications Development and Deployment Conference (USELINUX), Jan. 1997.

[27] Ingo Molnar, Linux Low Latency Patch for Multimedia Applications, http://people.redhat.com/mingo/lowlatency-patches/

[28] Jon Roberts, Linux Device driver, O'Reilly 2001.

[29] GNC library, chapter 20, Date and Time, http://www.gnu.org/manual/glibc-2.2.3/html_chapter/libc_21.html

[30] David A Rusling, The Linux Kernel, January 1998, Linux LDP, Mar. chap. 7, 1998.

[31] John P. Lehoczky, Lui Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. Proceedings of IEEE Real-Time Systems Symposium, pp. 166-171, Dec. 1989

[32] Kevin Jeffay and Donald L. Stone, "Accounting for interrupt handling costs in dynamic priority task systems," In Proceedings of the 14th IEEE Real-Time Systems Symposium, pp. 212-221, Raleigh-Durham, NC, December1993.

[33] Mok A., Task management techniques for enforcing ED scheduling on a periodic task set. In Proceedings 5th IEEE Workshop on Real-Time Software and Operating Systems, pp. 42-46, Washington D.C. May 1988.

[34] Erik (J.A.K.) Mouw, Linux Kernel Procfs Guide, Delft University of Technology, http://www.kernelnewbies.org/documents/kdoc/procfs-guide/lkprocfsguide.html

[35] Kar R.P. and Porter K., "Rhealstone - A Real-Time Benchmarking Proposal". Dr Dobbs Journal, Vol. 14, No. 2, pp. 4-24, February 1989.

[36] Douglass Locke, Lui Sha, Ragunathan Rajkumar, John. Lehoczky, and Greg Burns. Priority Inversion and Its Control: An experimental investigation

[37] Goodenough JB and Sha L., The priority ceiling protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks. ACM Ada Letters, 1988,8(7):20~31.

[38] Sha, L., Rajkumar, R. and Lehoczky, J. P. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Technical Report, Department of Computer Science, CMU , 1987

[39] Burns, A. Scheduling hard real-time systems: a review, Software Engineering Journal, v.6 n.3, pp.116-128. 1991.

[40] Mok, A. and Chen, D. Multiframe Model for Real-Time Tasks, IEEE Real-Time Systems Symposium. 1996.

[41] Mok, A. and Chen, D. Multiframe Model for Real-Time Tasks, IEEE Transactions on Software Engineering, 23:10 - 635-645. 1997.

[42] Baruah, S., Chen D., Gorinsky S., and Mok, A. Generalized Multiframe Tasks. Real-Time Systems 17 (1), pp. 5-22. 1999.

[43] Thomadakis, M. and Liu, S. On the Efficient Scheduling of Non-Periodic Tasks. in Hard Real-Time Systems, in Proceedings of the 20th IEEE Real-Time Systems Symposium. Phoenix, USA. 1999.

[44] Puschner, P. and Koza, Ch. Calculating the Maximum Execution Time of Real-Time Programs. Real-Time Systems Journal, v.1 n.2, pp.159-176. 1989.

[45] Cheng, S.C., Stankovic, J.A. and Ramamritham, K. Scheduling Algorithms for Real-time Systems: A Brief Survey. Real-Time Systems, IEEE Press, pp. 150-173. 1993.

[46] Bracewell, R. "The Impulse Symbol." Ch. 5 in The Fourier Transform and Its Applications, 3rd ed. New York: McGraw-Hill, pp. 69-97, 1999.

[47] Baruah, S., Chen D., Gorinsky S., and Mok, A. Generalized Multiframe Tasks. Real-Time Systems 17 (1), pp. 5-22. 1999.

[48] Devi, U. C. An Improved Schedulability Test for Uniprocessor Periodic Task Systems. ECRTS 2003. IEEE Computer Society Press. 2003.

[49] Scott A. Brandt, Scott Banachowski, Caixue Lin, Timothy Bisson, "Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes," rtss, p. 396, 24th IEEE International Real-Time Systems Symposium (RTSS 2003), pages 396-407, Dec. 2003.

[50] Bernat, Guillem, Response Time Analysis of Asynchronous Real-Time Systems. Real-Time Systems 25(2-3): 131-156. 2003.

[51] Clark Williams, Linux Scheduler Latency, Red Hat, Inc. March 2002.

[52] Junsung Lim, Goodman and D.J., Evaluation of Time Scheduling for Real Time Services in High Speed Uplink Packet Access, Communications, 2006 IEEE International Conference on Volume 11, pp5258 – 5262, June 2006.

[53] Jeon, W.S., Jeong, D.G.;Combined Connection Admission Control and Packet Transmission Scheduling for Mobile Internet Services, Vehicular Technology, IEEE Transactions on Volume 55, Issue 5, pp. 1582 – 1593, Sept. 2006.

[54] Gozalvez, J., Lopez-Benitez, M.; Lazaro, O., Link Adaptation Algorithm for Improved Wireless Transmission of Delay-sensitive Packet Data Services Electronics Letters, Volume 41, Issue 14, pp. 813 – 815, July 2005.

[55] Maode Ma, Yongqing Zhu, Tee Hiang Cheng, A Systematic Scheme for Multiple Access in Ethernet Passive Optical Access Networks, Lightwave Technology, Journal of Volume 23, Issue 11, pp. 3671 – 3682. Nov. 2005.

APPENDIX

A. Implement Embedded Linux on PDA

The real-time support Linux is developed on Familiar Linux and Ipaq platform. Familiar Linux is presently available for the Compaq iPaq. Since the architecture of iPaq is quite different to the normal desktop computer, we have to develop the application programs for this Linux embedded system.

During the development of embedded Linux on PDA, the source code is written and modified in the Linux on desktop. The source code is compiled by the cross compiler for Arm Linux. It is a compiler used to compile a source code for arm target platform.

The Familiar Project is composed of a group of loosely-cooperation developers all contributing to creating the next generation of PDA Operating System. Currently, most of our development time is being put towards producing a stable, and full featured Linux distribution for the Compaq iPAQ h3600-series of handheld computers, as well as applications to run on the distribution. The hardware and software requirements are listed as following:

A.1. Hardware Requirement:

1. Compaq H3600 iPAQ
2. Serial cable
3. Serial cradle

A.2. OS/Software Requirement:

1. Red Hat Linux 7.3
2. Cross compiler for ARM used in Redhat Linux
3. GCC
4. Root Image – Familiar v.0.5.2 task-bootstrp.jffs2)
5. A terminal emulator capable of performing xmodem uploads – Hyperterminal
6. ActiveSync Software (on host pc)
7. Bootloader – ARM Bootldr 2.18.01
8. Some applications for PDA – qpe application

As we use Compaq iPAQ h3600 as the hardwire, and it is in the series of Compaq iPAQ of handheld computers, so Familiar Linux is surely an ideal OS platform that we place our target on.

Currently Familiar's Linux distribution supports some of the following key features:

- Qtopia GUI System.
- a good wealth of peripherals and GUI applications .
- JFFS2 file system supported.
- Binary and Library distribution compatible the ARM target.

The current software release versions are:

- Familiar Project Linux v0.5.1 for the IPAQ H3600.

For free download, please visit: http://familiar.handhelds.org/

A.2.1. Install pre-built toolchain

A.2.1.1. Download tool-chain

The toolchain actually consists of a number of components. The cross-compiler tools targeted on the for a development environment must be installed is listed as follows.

- arm-linux-gcc-*.rpm (The compiler itself gcc)
- arm-linux-binutils-*.rpm (A set of tools for manipulating binaries)
- arm-linux-glibc-*.rpm (C-library glibc)
- arm-linux-kernel-*.rpm (Kernel source code of ARM Linux)

The most convenient approach to install cross-compiler tool-chain is to use rpm. The download URL is ftp://ftp.netwinder.org/users/c/chagas/arm-linux-cross/RPMS/ . The packages are suitable for Linux version 2.4.

A.2.1.2. Install the tool-chain

If there is rpm package, please:

rpm -ivh arm*-binutils*.rpm arm*-gcc*.rpm arm*-glibc*.rpm arm*-*rpm

But sometime version conflicts happened. Thus to install it one by one could be safer. The installation path of arm-linux toolchain is /usr/local/arm-linux.

A.2.1.3. Setup environment variables

In order to make the toolchain work, please set a couple of variables:

PATH=/usr/local/arm-linux/bin:$PATH

LD_LIBRARY_PATH=/usr/local/arm-linux/lib:$LD_LIBRARY_PATH

CC=arm-linux-gcc

CXX=arm-linux-g++

PREPROCESSCMD="$CC -E"

LD=arm-linux-ld

PS1="[arm-linux] $PS1"

A.2.2. Cross Compiling C-program

This sub-section shows how to build C program for ipaq using the cross compiler.

For a sample hello.c , we'll make a new directory “hello” and do the following.

mkdir ~/hello

cp hello.c ~/hello

cd ~/hello-world

To build, call the compiler, with the right options, with the command

arm-linux-gcc -Wall -g -o hello hello.c

A.2.3. Compile your toolchain

This subsection introduces how to compile your toolchain. This is useful when the pre-built package is unavailable.

To download the binutils-2.10.tar.gz

tar zxvf binutils-2.10.tar.gz

cd binutils-2.10

mkdir build

cd build

../configure --target=arm-elf --prefix=/usr/local/gnu

make

make install [Note: write access to /usr/local/gnu is required]

To Build gcc, please download the gcc-2.95.3.tar.gz

tar zxvf gcc-2.95.3.tar.gz

We apply patch from Philip Blundell to change the Position Independent Code generation for ARM:

gzip -cd gcc-2.95.3-arm-pic.patch.gz | patch -p0

cd gcc-2.95.3

mkdir build

cd build

../configure --target=arm-elf --prefix=/usr/local/gnu --disable-threads

make LANGUAGES="c" [Note: be sure /usr/local/gnu/bin is in the PATH list]

Then we continue installing the C compiler:

cd gcc

make install

Build libc ...

Download the newlib.tar.gz

tar zxvf newlib.tar.gz

cd newlib-1.8.1

mkdir build

cd build

```
# ../configure --target=arm-elf --prefix=/usr/local/gnu
            --enable-target-optspace
# make
# make install
```

To build other languages supports in gcc:

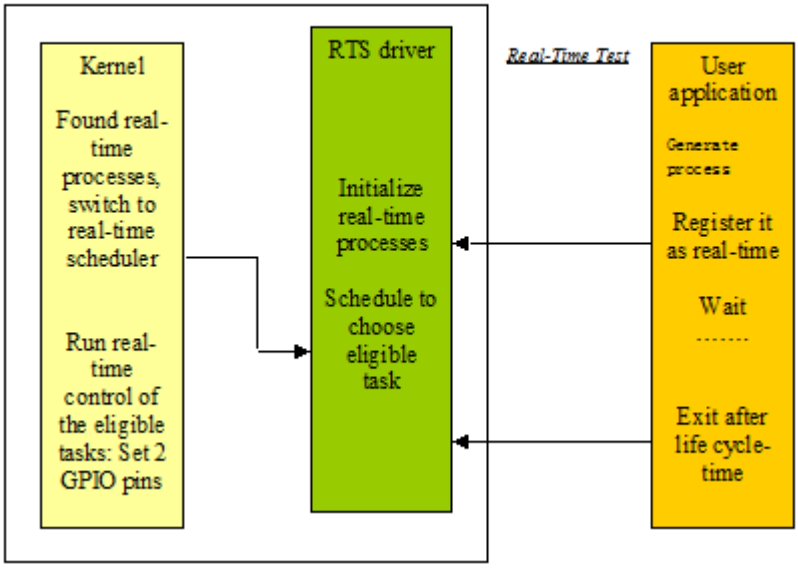
```
# cd gcc-2.95.3/build
# ../configure --target=arm-elf --prefix=/usr/local/gnu
# make LANGUAGES="c c++"
# make install
```

B. Introduction of Real-Time Test Application

The real-time support (RTS) Linux is developed based on Familiar Linux and Ipaq. It is designed to support real-time control in standard Linux system. We have introduced the design of real-time control module in RTS-Linux in section 4.2. Some target real-time applications are designed for verifying the design methodology and demonstrating the practicality of the design. Its purpose and description has been introduced in section 6.4.2. We introduce an example of target real-time application in this section.

In this example, there is one application that generates multiple real-time processes in the system. To avoid the long latency spending in the switching between user space and kernel space, the actual action taken by the real-time process is defined in the kernel. The action implemented by the real-time processes is to change two GPIO pins from (0,0)->(0,1) ->(1,1)->(1,0)->(0,0) periodically. If these two pins are physically connected to a motor controller, the real-time control can drive a motor turn step by step. Then motor is turned periodically.

In the application, there is multiple real-time tasks are generated in the system. An example coding of real-time application is shown in List B.1. The flow chart of real-time test application and kernel is shown as follows. rtstest is a test application that generates several processes and register them as real-time processes. Kernel Scheduler will check whether there is a real-time process and switch to real-time scheduler. RTS driver will initialize the real-time tasks according to relative IOCTL functions and schedule all the real-time tasks.



List B.1 Example Coding of Real-time Test Application

```
struct {
  int compute;
  int period;
  int deadline;
} Parameters[NTASKS];


int unregister(void)
{
        int retval, pid;
        retval = ioctl(file, IOCTL_CLEAR_EVENTS);
        if (retval) {
                perror("CLEAR_EVENTS");
                exit(0);
                }
        retval = ioctl(file, IOCTL_SET_POLICY, OTHER_SCHED);
        if (retval) {
                perror("set policy");
                exit(retval);
                }
        /* Now switch the kernel to realtime */
        retval = ioctl(file, IOCTL_UNREGISTER_RTS, 0);
        if (retval < 0) {
                perror("unregister rts");
                exit(0);
                }
        return 0;
}
int registerme(void)
{
        int retval;
        file=open("/dev/rts",O_RDWR| O_NOCTTY | O_NONBLOCK);
        fprintf(stderr,"open /dev/rts\n");
        if(file<0){
                fprintf(stderr,"open /dev/rts error %s\n", strerror(errno));
```

```
                close(file);
                return(1);
                }
        retval = ioctl(file, IOCTL_REGISTER_RTS, 0);
                if (retval) {
                                perror("register_rts");
                                exit(retval);
                        }
        retval = ioctl(file, IOCTL_SET_POLICY, EDF_SCHED);
                if (retval) {
                                perror("set policy");
                                exit(retval);
                        }
        return 0;
}
void init_timing(void)
{
 Parameters[0].compute=14;
 Parameters[0].period=70;

 Parameters[1].compute=5;
 Parameters[1].period=90;
……
}


/* t -- the fifo number */
void fun(int computo)
{
        int bucle,x;
 while(1){

   for (bucle=0; bucle < computo ; bucle++){
     for (x=0; x<1000; x++);
   }
}

static int rt_task_init(int i, void (*fn)(int), int compute, int period, int deadline)
{
        int childpid;
        struct rt_param rtpar1;


        rtpar1.pid = 0;
        rtpar1.starttime  = 0;
        rtpar1.period  = (unsigned long) period;
        rtpar1.compute  = (unsigned long) compute;
        rtpar1.dd  = (unsigned long) deadline;
        rtpar1.edf_policy = 0;
        if ((childpid =fork())<0) {
                perror("fork error");
                exit(1);
                }
        if (childpid ==0) {
                if (ioctl(file, IOCTL_REGISTER_TSK, &rtpar1)<0){
                                perror("join RTS error");
                                exit(1);
                                }
                        printf("child, real-time task %d forked\n",i);
                fn(compute);
```

```
                }
        else { /* parent */
                        sleep(2);
                        printf("parent, real-time task %d forked\n",i);
                        printf("parent, fork pid: %d\n",childpid);
                        pids[i] = childpid;
                }
        return 0;
}


int main(void)
{
    int ret, i;
    int fp;
    time_t t;
        u64 begintime, endtime;
        double max_delay = 0;
        struct timeval tv;
        gettimeofday(&tv,0);
        begintime = tv.tv_sec;
        //signal(SIGINT, IntProc);
        if(registerme()) {
                perror("open /dev/rts fail");
                exit(0);
                }
        init_timing();
        for (i = 0; i < ACTNO; i++) {
        rt_task_init(i, fun, Parameters[i].compute,
                Parameters[i].period, Parameters[i].period);
        }
        for (i = 0; i < ACTNO; i++) {
                fprintf(stderr,"forked ps, pid: %d\n", pids[i]);
        }

        fprintf(stderr,"begin sleep 600\n");
        sleep(6000);//wait everybody get ready
        fprintf(stderr,"<end>\n");
        gettimeofday(&tv,0);
        endtime = tv.tv_sec - begintime;
        fprintf("test: %d minutes %d seconds\n", (int)endtime/60, (int)endtime%60);
        fprintf(stderr,"start the interactive test\n");

        unregister();

        //send kill to all the childs
        for (i = 0; i < ACTNO; i++) {
                kill(pids[i],SIGUSR1);
        }
        close(file);
}
```