ÇOK ÇEKİRDEKLİ GÖMÜLÜ İŞLEMCİLER ÜZERİNDE GERÇEK ZAMANLI SİSTEMLER İÇİN ÇOKLU GÖREV ZAMANLAYICI TEKNİĞİ

MULTI-SCHEDULING TECHNIQUE FOR REAL-TIME SYSTEMS ON EMBEDDED MULTI-CORE PROCESSORS

ABDULKADİR YAŞAR

Assist. Prof. Dr. Kayhan M. İmre
Supervisor

Submitted to Institute of Sciences of Hacettepe University
as a Partial Fulfillment to the Requirements
for the Award of the Degree of Master of Science
in Computer Engineering

2014

This work named “Multi-Scheduling Technique For Real-Time Systems On Embedded Multi-Core Processors” by Abdulkadir Yaşar has been approved as a thesis for the Degree of Master of Science in Computer Engineering by the below mentioned Examining Committee Members.

Assoc. Prof. Dr. Ali Ziya ALKAR
Head ……..……

Assist. Prof. Dr. Kayhan M. İMRE
Supervisor ……..……

Assist. Prof. Dr. Ahmet Burak CAN
Member ……..……

Assist. Prof. Dr. Sevil Şen AKAGÜNDÜZ
Member ……..……

Assist. Prof. Dr. Murat AYDOS
Member ……..……

This thesis has been approved as a thesis for the Degree of Master of Science in Computer Engineering by Board of Directors of the Institute for Graduate Studies in Science and Engineering.

Prof. Dr. Fatma SEVİN DÜZ
Director of the Institute of
Graduate Studies in Science

DEDICATED

To

My deceased uncle

Kadir Yaşar

Never seen and met with him but I know him very well from his memories he left behind

I grow up with his adventures, stories and books

His leadership and farsightedness inspired me

I am very proud of carrying his name..

ETHICS

In this thesis study, prepared in accordance with the spelling rules of Institute of Graduate Studies in Science of Hacettepe University,

I declare that

- all the information and documents have been obtained in the base of the academic rules
- all audio-visual and written information and results have been presented according to the rules of scientific ethics
- in case of using others Works, related studies have been cited in accordance with the scientific standards
- all cited studies have been fully referenced
- I did not do any distortion in the data set
- And any part of this thesis has not been presented as another thesis study at this or any other university.

25/06/2014

ABDULKADİR YAŞAR

ABSTRACT

MULTI-SCHEDULING TECHNIQUE FOR REAL-TIME SYSTEMS ON EMBEDDED MULTI-CORE PROCESSORS

Abdulkadir YAŞAR

Master of Science in Department of Computer Engineering

Supervisor: Assist. Prof. Dr. Kayhan M. IMRE

July 2014, 77 pages

Recent studies have shown that today's embedded systems require not only real-time ability but also general functionality. In order to provide these two functionalities on same system, many researches, techniques and frameworks have been developed. Integrating multiple operating systems on a Multi-core processor is one of the most favorite approaches for system designers. However, in this heterogeneous approach, failure in one of the operating systems can cause the whole system to come down. Moreover, in recent years many scheduling techniques such as external and partition-based scheduling have been developed to provide real-time ability for general purpose systems in single operating system without using heterogeneous approach.

This thesis introduces Multi-scheduling method for Multi-core hardware platforms without running heterogeneous operating systems concurrently. In this technique, there are two schedulers in single operating system. One of them is for real-time applications and the other is for general or non-real-time applications. In heterogeneous operating systems approach, a real time operating system services real-time functionality such as low interrupt latency while a versatile operating system processes IT applications. Unfortunately, Real-time and IT

applications are isolated and run on different operating system environments. This may cause some problems in system design and Inter-Process-Communication (IPC). In Multi-scheduling approach, Real-time and IT applications run in the same operating system environment so the implementation and maintenance of the system become easier.

We implemented our work on Linux, widely used general purpose operating system for embedded and industrial systems. By modifying Symmetric-Multiprocessing (SMP) technique in Linux, two schedulers are enabled to run on same kernel and each of them runs on different CPU cores. Our proposed technique is tested by real-time de-facto test tools and programs accepted all over the world. The most important characteristic of a real-time application such as low interrupt latency and responsiveness were benchmarked. The results show that Multi-scheduling technique can be profitable to bring the real-time functionality to general operating system as in heterogeneous approach.

Keywords: Multi-core, Multi-Scheduling, Embedded Systems, Real-time, Scheduling, Operating System, Symmetric Multi-processing, Linux

ÖZET

ÇOK ÇEKİRDEKLİ GÖMÜLÜ İŞLEMCİLER ÜZERİNDE GERÇEK ZAMANLI SİSTEMLER İÇİN ÇOKLU GÖREV ZAMANLAYICI TEKNİĞİ

Abdulkadir Yaşar
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Danışmanı: Yrd. Doç. Dr. Kayhan M. İMRE
Temmuz 2014, 77 sayfa

Son yıllarda yapılan çalışmalar ve ortaya çıkan ihtiyaçlar günümüz gömülü sistemlerinin hem gerçek zamanlama yeteneğine hem de genel yeteneklere aynı anda ihtiyaç duyduğunu göstermektedir. Bu iki yeteneği aynı sistem üzerinde gerçekleştirebilmek için birçok araştırmalar, çalışma altyapıları ve farklı teknikler geliştirilmiştir. Çok çekirdekli işlemciler üzerinde birden fazla ve farklı işletim sistemlerinin bir arada çalışmasını sağlayan yaklaşım sistem geliştiriciler arasında en yaygın olanıdır. Yine de heterojen yaklaşımda işletim sistemlerinden bir tanesinde meydana gelecek kritik bir hata tüm sistemin çalışmasını engelleyebilir. Bu yaklaşımın dışında, son zamanlarda harici ve parça tabanlı olmak üzere farklı görev zamanlayıcı teknikleri genel işletim sistemlerine gerçek zamanlama yeteneğini heterojen yaklaşım kullanmadan kazandırmak amaçlı geliştirilmiştir.

Bu tez çalışmasında çok çekirdekli donanımlarda heterojen işletim sistemleri çalıştırmayan Çoklu görev zamanlama (Multi-scheduling) ismini verdiğimiz yöntem sunulmaktadır. Bu yöntemde bir işletim sistem içerisinde iki farklı görev zamanlayıcı farklı işlemci çekirdekleri

üzerinde çalıştırılmaktadır. Görev zamanlayıcılardan bir tanesi gerçek zamanlı görevler, diğerİ ise genel ya da gerçek zamanlı olmayan görevler içindir. Heterojen sistemlerde işletim sistemlerinden bir tanesi düşük kesilme gecikmesi gibi gerçek zamanlama yetenekleri sunarken, diğer genel amaçlı işletim sistemi de genel amaçlı görevlerde kullanılmaktadır. Ne yazık ki heterojen yaklaşımda gerçek zamanlı ve genel görevler birbirlerinden ayrılmış farklı işletim sistemlerinde çalıştığından sistem tasarımda ve görevler arası iletişimde bazı sorunlara ve zorluklara sebep olmaktadır. Bu çalışmada sunulan çoklu görev zamanlayıcı yönteminde gerçek zamanlı ve genel görevler aynı işletim sisteminde dolayısıyla aynı çevrede çalıştığından bütün sistemin bakımı ve geliştirimi daha kolay olmaktadır.

Bu çalışma gömülü sistemlerde ve endüstride birçok kullanım alanına sahip olmasından ve iyi belgelenmesinden dolayı Linux işletim sistemi üzerinde gerçekleştirilmiştir. Linux işletim sisteminin Simetrik çoklu işleme (Symmetric-Multiprocessing) özelliğinin olduğu bazı kesimler değiştirilerek iki farklı görev zamanlayıcının aynı işletim sistemi üzerinde farklı çekirdeklerde çalışması sağlanmıştır. Ayrıca kullanıcılar için bu tekniği yöneten bir de uygulama hazırlanmıştır.

Çalışmada geliştirilen yöntem gerçek zamanlı uygulamaların ihtiyaçlarını baz alan ve dünya genelinde kabul görmüş test araçları ve uygulamaları kullanılarak değerlendirilmiş ve yorumlanmıştır. Özellikle kesilme gecikmelerinde gözlemlenen yöntemin kullanılmadığı standart sistemlere göre yaklaşık iki katı iyileştirme ve sistemin ani olaylara daha kararlı cevap vermesi sunduğumuz yöntemin faydalı ve kullanışlı olabileceğini ortaya koymaktadır.

Anahtar Sözcükler: Çok çekirdekli işlemciler, Çoklu görev zamanlama, Gömülü sistemler, Gerçek zamanlama, Görev zamanlayıcı, İşetim sistemi, Simetrik çoklu işleme, Linux

ACKNOWLEDGEMENT

First and foremost, I have to thank my research supervisor Assist. Prof. Dr. Kayhan M. İmre. Without their assistance and dedicated involvement in every step throughout the process, this thesis would have never been accomplished. I would like to thank him very much for his support and understanding over these past four years. I want to thank to Dr. Ahmet Burak Can and Dr. Ersin Eroğlu for their material and moral support since my undergraduate years. I feel lucky to be one of their graduate students, possibly benefiting from their teaching skills one of the most, and thank them one more time for the opportunities they have provided me.

Second, I am very grateful to all of my workmates in Aselsan, specifically Alper Yıldırım and Alaadin Köroğlu for their help and insights. I also would like to thank Çağlar Kılcıoğlu, Mehmet Emre Şahin and Ozan Küsmen for sharing his studies and ideas with me generously. I am also thankful to Canan Balcı Göbekli and Ayşen Balbaş Bulu for feeding us with their cakes, muffins and cookies during studies and workings in the lab, Ali İbrahim Bostancıoğlu for being a good roommate and workmate and his special encouragements, Ahmet Orunç and Tuğser Kutlu for their technical support about hardware, Yaşar Kalay for sharing his endless and priceless ideas especially about nature with me.

I am grateful to my current directors Cemal Kırlılar and Ali Büke for showing tolerance during the period of writing this thesis.

I am also appreciative of the support from my company, Aselsan, for its encouragement and financial support to study the Master of Science degree while working.

I also would like to thank all members of Linux Kernel mailing list, specifically to Steven Rostedt, for sharing their ideas with me, which helped me very much in solving problems related with this thesis.

I am also thankful to Mehmet Akif Akkuş and Ahmet İlhan Ayşan, for being a perfect colleague and friend, helping me any time when I became boring and dispirited by riding a bike. He also advised and helped me to study Master of Science in Hacettepe University.

Most importantly, none of this could have happened without my parents, Emine Yaşar and Fikret Yaşar. And, of course, special thanks for my brother, Mustafa Kemal Yaşar, and my sister-in-law, Çiğdem Yaşar, especially for their endless dessert support; baklava. It would be an understatement to say that, as a family, we have experienced some ups and downs in the past. Every time I was ready to quit, they did not let me and I am forever grateful. This dissertation stands as a testament to their unconditional love and encouragement.

CONTENTS

LIST OF TABLES

Page

LIST OF LISTINGS

Page

LIST OF FIGURES

Page

LIST OF SYMBOLS AND ABBREVIATIONS

Symbols

µs	microsecond

Abbreviations

RT	Real-Time
OS	Operating System
RTOS	Real-Time Operating System
AMP	Asymmetric Multi-processing
SMP	Symmetric Multi-processing
rt-task	Real-time task
non-rt-task	Non Real-time task
rt-core	Real-time CPU core
Msched	Multi-scheduling
IPC	Inter-process Communication
MPI	Message Passing Interface
SoC	System on a Chip
CPU	Central Processing Unit
API	Application Programming Interface
GCC	GNU Compiler Collection
CFS	Completely Fair Scheduling
EDF	Earliest Deadline First

1. INTRODUCTION

In the last decade, the processor manufacturers place multiple processor cores in a single chip called System on Chip (SoC) to speed up the computation, to improve the performance and to reduce the cost. These processors may be composed of two or more independent cores based on symmetric or asymmetric multiprocessing architectures [1]. In addition to desktop or server PCs, multi-core processors are used in embedded systems for performance and economic reasons.

A typical embedded system is dedicated to perform functions such as real-time data control and digital signal processing. Unsurprisingly, embedded systems also require general non-real-time functionality as well as real-time (RT) functionality. Combination of these two functionalities is one of the most challenging problems for embedded and RT system developers. In order to overcome this problem, processor manufacturers usually produce heterogeneous multi-core processors [1], [2]. In heterogeneous processors, each core runs a different type of operating system (OS) to perform required functionality. For example, in a dual-core processor, a real-time operating system (RTOS) runs on one core, and a versatile or general purpose OS runs on the other. Memory and peripherals are isolated by hardware, or a low-level software called hypervisor. On the other hand, in homogenous processors, each core runs the same OS code, and share the main memory, peripherals and other resources [2].

Migration from single-core to multi-core processor brings a discussion about how to manage OS code over the cores in SoC. There are two suggested modes; asymmetric-multiprocessing (AMP) and symmetric-multiprocessing (SMP). In AMP mode, each core has its own copy of OS kernel code, and the codes are generally different from each other (heterogeneous OSes). On the contrary, in SMP mode, the same kernel code runs on each core synchronously (homogenous OSes). SMP OS dynamically balances the work between processor cores, and controls the resource sharing, e.g., main memory, between the cores [3], [4].

Despite the fact that the heterogeneous OSes in AMP mode are difficult and costly to maintain, they are widely used in embedded RT systems [1], [5]. The reason behind this fact is that reserving the processor time for RT tasks in SMP mode is not trivial. In AMP mode, one or more cores run a RTOS in which RT application get the total control of the core(s) easily. The RTOS has a special scheduler, e.g., EDF scheduler, to meet strict timing constraints [6],

[7]. On the other hand, a general purpose OS has a scheduler, e.g., CFS, which gives tasks a fair share of a CPU's time.

It is essential that a RT system must respond actions or produce results within predefined timeframes [7]. In a RTOS, a task must gain immediate access to the processor to produce a timely response to an interrupt or action [2], [4], [7]. Therefore, processor should be waiting in idle state for the most of its time. Although the processing speed is important for the quality of a RT system, it is not the primary purpose of an RT system. The primary purpose of an RTOS is to eliminate the surprises [4]. In other words; RTOS must provide a solid infrastructure to guarantee the response time of a task. However, in a general purpose OS, time for the completion of a task is unpredictable and may diverge [8].

In this work, we propose a new technique called Multi-scheduling for SMP multi-core embedded processors to enable to run RT tasks along with the general purpose non-real-time tasks. We have implemented our approach in Linux since it is the most widely used OS in embedded systems. In our approach, there are two schedulers running in a single OS environment. After booting on SMP system, one or more cores, selected in kernel configuration, change their scheduling policy to an appropriate RT scheduler. Therefore, RT tasks and general tasks are run on separate cores. We have measured the interrupt latencies and average task completion times of the multi-scheduling policy on a system containing an ARM Cortex-A9 dual-core processor. We have also carried out the same measurements for the standard kernel on the same hardware. Our results show that multi-scheduling technique can be used to bring RT functionality to SMP homogenous multi-core processors.

The remainder of this paper is organized as follows: Chapter 2 reviews related works; Chapter 3 gives necessary information as background; Chapter 4 introduces the Multi-scheduling technique, and then its implementation on Linux will be detailed in Chapter 5 and 6; Chapter 7 identifies the benchmark and comparison results; and finally conclusions and future works are drawn in Chapter 8.

2. RELATED WORK

Heterogeneous operating systems are widely used in embedded systems to integrate real-time and non-real-time functionality together. Low-level software called hypervisor is used to partition the hardware resources between OSes [9]. Moreover, physical partitioning techniques have been developed to run RTOS and general OS simultaneously on same system [1], [10]. However, the physical partitioning techniques require hardware modifications on SoC to control the access lists to resources.

Linux is widely used OS in not only servers and desktops but also embedded systems. However, it suffers from lack of hard RT functionality. Although it is originally developed as general purpose OS, several RT infrastructures have been adopted to Linux kernel in recent years [3], [11], [12]. The RT-patch provides several modifications such as low latency support and preemption into the standard Linux kernel to yield hard RT support [4], [13]. Nowadays, the standard Linux also can be used in RT application but it provides soft RT infrastructure [2], [3]. Researchers in [13] made first experimental analysis of RT performance of the standard Linux primitives on multi-core platforms.

In the recent years, numerous scheduling methods have been suggested for homogenous multi-core processors. Authors in [14] implemented a hybrid scheduling method to make the parallelism by partitioning an application into some parallel tasks. In [6] and [14], the authors implement a task splitting semi-partitioned scheduler for multi-core embedded systems. They show that semi-partitioned scheduling has better performance and low overhead than other partition-based scheduling methods. Moreover, authors in [16] have developed a loadable RT scheduler suite to enable multi-core platforms to run different scheduling algorithms simultaneously. In [7], the researchers propose a new technique to support to run soft real-time periodic tasks in high performance asymmetric multi-core platforms that are running Linux. In a different work, a new real-time scheduling method for multi-core platforms that runs threads of a multi-threaded real-time task on different cores synchronously is developed in [17]. In the approach proposed in [8], real-time tasks are grouped in clusters. A task's cluster cannot be changed later and tasks are scheduled with their own cluster by using EDF scheduling algorithm. This hybrid approach enables large-scale multi-core platforms with hierarchical shared caches to run real-time tasks.

The proposed work in this thesis is a kernel-level approach to bring real-time functionalities to a general operating system such as Linux. Rt-patch [4] is also a kernel-level method that provides necessary timing features and modifications for the standart Linux kernel. Rt-patch provides lower latency in mutual exclusion mechanism such as spinlock and a better preemption in task scheduling. Multi-scheduling method is also implemented in kernel level like rt-patch, but it creates two isolated environment in kernel space for rt and non-rt tasks. On the other hand, rt-patch is able to run all tasks in same environment thanks to its highly reliable timing extensions and lower latency feature.

The work in [1] also targets same problem, running real-time and general tasks together. An hardware isolation layer that seperates rt and non-rt environments is proposed. We have been inspired from this work and decided to implement this work in kernel level with software.

To sum up, there are some methods and techniques to similar our proposed Multi-scheduling technique to provide both real-time and general functionalities. Our target is to show the fact that Multi-scheduling technique can be evaluated as a new technique to isolate real-time and non-real-time partitions or environments in same operating system.

3. BACKGROUND

The proposal scheduling technique presented in this thesis involves the integration of various hardware and software components. This chapter deals with the basic background knowledge required for a clear understanding of the Multi-scheduling technique details. The topics covered in this section are quite complex, and it is not possible to cover all details. Therefore, only brief summaries of the topics are presented. Firstly, we will brief what real-time and real-time systems are. Secondly, RTOS and its differences between general OSes will be explained. Following that, importance of scheduling in real-time and most known scheduling algorithms for RT systems will be summarized. Famous RTOSes will be briefed in Section 3.5. Then, we will cover embedded systems and multi-core requirements in Section 3.6. Finally, the chapter ends with a discussion between asymmetric and symmetric processor architectures.

3.1. Real Time Systems

In Computer Science, Real-time means a time period that a computer system must produce a result or respond in. In other words, it can be considered as a *deadline* that a result or respond must be produced [4], [7]. In the real world, the goal of a real-time system is to have a physical effect within a given time period. A real-time system consists of two main components; computer and environment. Computer system interacts with its environment to get information and responds the environment according to collected data from outside. Sensors are used to sense and collect data from environment in real-time embedded systems. The computer or controlling mechanism processes the data and produces a respond to outside. For example, a thermostat has a temperature sensor that senses the environment and a processor that controls an output device such as cooler. Any change in temperature of the environment is processed and cooler is managed if necessary. In some real-time systems, unexpected changes or unforeseen events may occur and it must be dealt with immediately for example in defense critical or medical devices. In this situation, computer system of real-time system must demand necessary timings of the task handling the critical events [7]. According to the type of real-time system, not demanding or insufficient in timing constraints can cause different cost-level results. A failed timing result leads to severe problems for safety critical real-time systems. For some systems, failed result is negligible or meeting necessary timings

are not so critical. To sum up, an rt-task that needs timing features in a real-time system must be handled carefully. In other words, a real-time system must provide accurate and sufficient timing features for rt-tasks. These timing properties are very related with task scheduling in a real-time system [7].

In real-time concept, the correctness of the result that is produced for an event is not only consequence. Respond time to an event or meeting necessary timings is also important. If the timings are not met, the system can be considered as unsuccessful. The fact that a real-time system must provide highly reliable timing features for rt-tasks increases the system's predictability. The higher predictability a real-time system has, the more possible to meet timings [12].

Real-time systems can be separated into 3 groups according to the criticality of the application area which the real-time system serves in [7], [18].

- Hard real-time system; if the system fails to meet the deadline even once then the system is considered to be failed. In other words, not meeting the deadline causes to potential loss of life or big economic damage. These systems are considered to be safety or mission critical. A cost function associated with hard real-time systems is depicted in Figure 3.1.

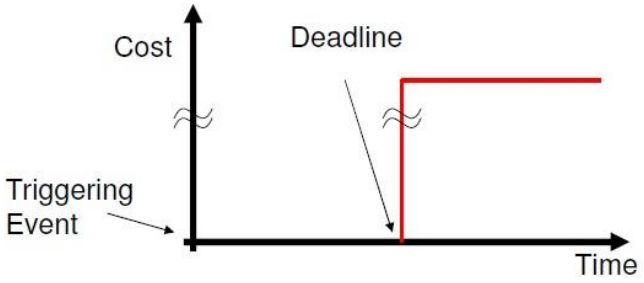

Figure 3.1. Cost Function associated with hard Real-time Systems

- Soft real-time system is the system that not meeting the deadline can be tolerable but not preferred. There will be no vital results if one or more deadlines are missed or met. However, the results are considered as worthless when it is produced after its deadline. This causes to increase the cost of the system as figured in Figure 3.2 below for example Audio or video stream processes.

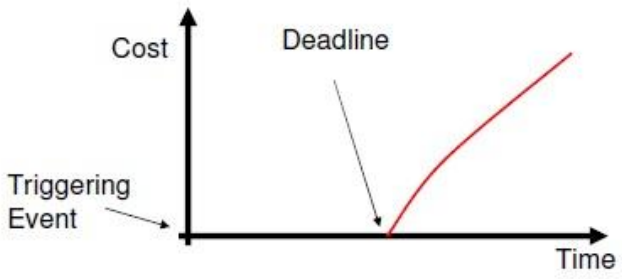

Figure 3.2. Cost function of soft Real-time systems

- Firm real-time system; the respond is obsolete if the deadline is missed. For example, a fore-casting system is a good example of Firm real-time systems where meeting deadline is desired but not critical or vital. Producing a result before deadline is a gain and provides to reduce the cost as depicted in Figure 3.3.

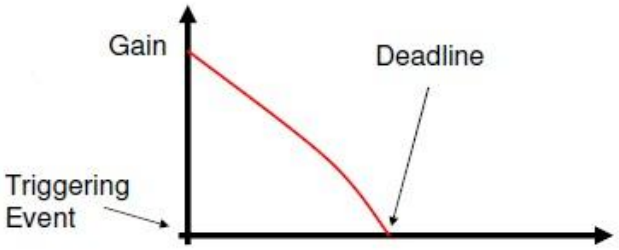

Figure 3.3. Gain function of firm Real-time systems

3.2. Real Time Operating Systems (RTOS)?

An operating system is a computer program that is responsible for managing and sharing hardware resources or components of a computer and provides services to user applications to run on the computer. The most important object of an OS is to run application or tasks with providing them for the resources such as memory and CPU fairly. On the other hand, A RTOS is a special OS provides the tasks with not only the services that an OS provides but also a very exact timing constraints and high reliability [2]. In other words, a RTOS is an OS that obey the Real-time requirements such as meeting deadline that mentioned on previous section.

A RTOS must guarantee to not to exceed a maximum time for every critical or urgent process. OS calls and interrupt handling can be given as some examples of these operations. These are time-bounded operations and not meeting required timings of these operations can cause critical problems. In practice, RTOSes are generally grouped in 2 categories [7]; hard and soft.

For example, failure or not meeting the timings of an anti-brake system, known as ABS, of a car can cause catastrophic results or maybe deaths. Therefore, a hard real-time system is needed; you need assurance as the system designer that no single operation will exceed certain timing constraints. On the other hand, small failures or latencies in timings of a VOIP transceiver device can be negligible or does not cause fatal problems. For this case, a soft RTOS may suffice [7], [17].

The most important point is that a RTOS must provide rt-tasks with high consistent timing as much as possible. Consequently, RTOSes have some opportunities such as task prioritization or different scheduling algorithms or policies to make the system to meet desired [7], [18]. On the other side, the most popular operating systems which are used in personal computers; for example Windows, are called general-purpose operating systems. Operating systems like Windows are designed to maintain user responsiveness with many programs and services running fairly, while RTOSes are designed to run safety and mission critical applications reliably and with determined timing constraints.

3.2.1. Monolithic kernel vs. Microkernel

Kernel is the most important and central part of the modern operating systems. It consists of the fundamental properties such as scheduling, process management and device drivers of an operating system. Operating systems are separated into two groups according to their kernel's structure; monolithic kernel and microkernel in Figure 3.4.

Monolithic kernel is a single large process running in a single address space. All kernel services and threads exist and execute in kernel address space and it provides to invoke functions directly. If one of the services or threads is crashed, it causes the whole system to come down [19]. In order to add some features or fix a problem in kernel, the whole kernel source code must be recompiled. It lacks of extensibility and maintainability. However, operating system designers have developed kernel module feature to add or remove kernel parts online without compiling the whole source code. Linux and UNIX are good examples for monolithic kernel based Oss.

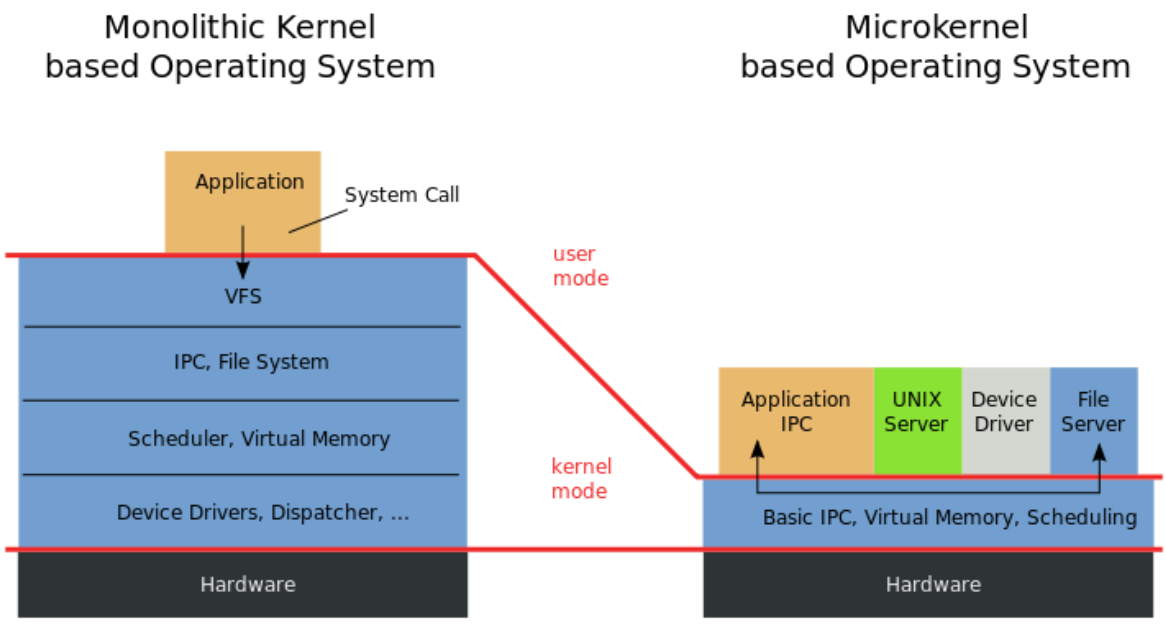

Figure 3.4. Monolithic kernel vs. Microkernel Architectures

The kernel is divided into separate threads, known as servers in Microkernel approach. All servers are isolated from each other and run in different memory address spaces. Moreover, these servers can execute in kernel-space or userspace. Message passing interfaces (MPI) or Inter-process Communication (IPC) are used to communicate between services in microkernels. Servers uses IPC messages or signals to call other services to carry out chained jobs [19]. However, these messaging between services cause to decrease operating system performance and this is the most considering disadvantage in this approach. One of the most important advantages of this approach is that if one of the servers in OS fails, other servers can continue to work. Mac OS X and Windows NT are good examples for Microkernel based OSes.

3.2.2. RTOS vs. General OS

General operating systems such as Microsoft Windows are designed to run tasks that do not need very precise timing constraints. They are suitable for normal use or daily applications such as document editing, games and internet browsers. These applications are not critical to respond actions or events and being late may not cause fatal results. On the other hand, some applications require precise timing constraints to respond to an action or produce a result before a deadline. Especially in medical and military areas, applications can need very reliable

timing properties to run. Consequently, using a general OS in such situations is not a good idea because a general OS cannot provide necessary timing features for a critical task. In this section, the major differences between a RTOS and a general OS will be explained.

Real-time operating systems are featured operating systems with reliable timing properties to provide an environment for real-time tasks. In order to satisfy timings, some components of operating systems must be enhanced such as latencies and scheduling mechanism. Latency is the period between the time when an event is occurred and the time when it is being handled. There are mainly two important types of latencies for RTOSes; scheduling latency and interrupt latency. Scheduling latency is the time between an rt-task needs to be woken up and the time it actually gains to control and run. It occurs in context-switching of the tasks when a higher priority task is scheduled or the current task relinquishes the processor. The less the scheduling latency is, the much more time remains to meet the deadline for an rt-task. Interrupt latency, on the other hand, is the time elapsed between an event occurred and when it is actually handled or processed. Interrupt latency is the most important latency to reduce for real-time systems [18], [20].

The other main difference is in scheduling mechanism. RTOSes have generally more deterministic scheduler algorithm than general OSes have. Determinism of a scheduler means that a task always has an opportunity to switch to CPU, in other words, RTOS provides execution for a task when it needs to run. General OSes share the CPU time between tasks fairly. On the other hand, RTOSes can not only share equal time slots between tasks but also delivers the CPU on event occurring, in other words they can have event-driven scheduling method.

To sum up, RTOSes does not have to have better performance than other general-purpose OSes [4], [7], [20]. It is one of the common misconceptions in Real-time concept. Focus of a general/regular OS is providing high computing , but a RTOS generally tries to respond events as much as possible [20]. OSes are designed to run general applications such as games and document editing, while embedded and critical tasks are run in RTOSes.

3.3. Importance of Scheduling

Scheduler is one of the operating system's kernel functions that run the tasks or processes in order according to its algorithm. When the processor is free or a task yields the processor or finishes, the scheduler chooses one of tasks from the task list and switches it to the processor. Scheduler can be considered as the engine of an OS. For this reason, it is the most basic and important part of an OS [7], [19].

Schedulers can be separated into two groups; preemptive and non-preemptive (cooperative). In preemptive scheduling, tasks can be suspended periodically, then the scheduler chooses new task to gain the processor. For example, imagine that there are 4 tasks in the task list of an OS and time period is 100ms. Every task will run at least 100ms in every 400ms cycles. On the other hand, tasks themselves decide to relinquish the processor when they finish. For example, tasks call ***sched_yield()*** system function to release the processor in Linux. This invocation will suspend the current task and yields the processor to other tasks in task queue. In spite of the easiness in implementation, cooperative scheduler is not suitable for RTOSes due to its lack of interrupts in scheduling. In other words, deadlines have to be met in real-time systems so rt-tasks must gain the processor to finish critical operations before the deadline. It cannot wait other tasks to yield the processor.

```
void Task_preemptive() {
   Init_task();
   while(1) {
      functions();
   }
}
```

```
void Task_Non_preemptive() {
   Init_task();
   while(1) {
      functions();
      if (status())
         sched_yield();
   }
}
```

Figure 3.5. Preemptive task and non-Preemptive task

Due to the requirement for high resolution timing and time constraints, rt-tasks must be scheduled or behaved demandingly [1]. Consequently, the scheduler needs to have some

special features to handle rt-tasks. The following objectives should be considered in scheduling a real-time system;

- Providing timing constraints and meeting the deadlines of the tasks
- Speed context switches in process/task changing
- Preventing concurrent access to shared resources, devices/peripherals
- Fast respond to soft/hard interrupts
- CPU is kept in idle as much as possible. It means that CPU should be ready to run an immediate task. This provides fast responsiveness and it is preferred in real-time systems [7].

To sum up, the scheduler or scheduling algorithm is the most fundamental and important part of a RTOS. The scheduling algorithm should be chosen according to required timing properties of rt-tasks running in the system. In the next section, we will discuss the most known and used scheduling algorithms for real-time systems and basic properties of a real-time scheduling algorithm will be given.

3.4. Real-time Scheduling Algorithms

The most important and imperative goal for scheduling in RTOSes is providing reliable timing features to complete tasks before deadlines and preventing to access to same shared resource, a hardware component or a device simultaneously as mentioned in previous section [18], [21]. There are some decisions that a RTOS has to enable to run rt-tasks to meet their timing requirements. The set of these decisions are composed in the scheduling algorithm. Many of real-time scheduling algorithms need timing features of tasks such as total duration and remaining time to finish. Real-time scheduling algorithms should behave according to rt-task properties. Each task occurring in a real-time system has some timing properties in Figure 3.6. These timing properties can be valuable for scheduling algorithm to choose the task which should be scheduled or gain the processor on a real-time system. The timing properties of a task are given in the following list;

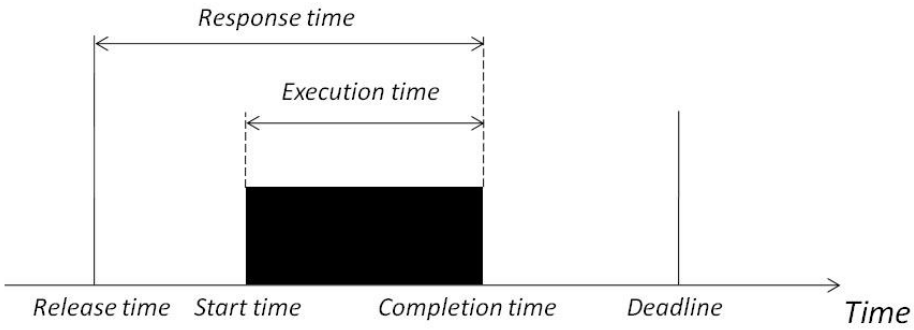

Figure 3.6. The basic attributes of an rt-task [18]

- *Release time (or ready time)*: It is the time that task is ready to go or run.
- *Deadline*: The last time that the task must finish before.
- *Start time*: Actual time that the task is being processed.
- *Execution time*: It is total amount of time that task is being processed or switched to processor.
- *Completion time*: It is time that the processing of task is finished.
- *Response time*: Time between release time and completion time. It is a very important timing constraint for an rt-task. Environment deals with the response time and tests for real-time systems measure this duration for performance evaluations.

The scheduling algorithms for real-time systems can be grouped according to the number of cores in the processor; uniprocessor or multiprocessor, or runtime decisions; on-line or off-line; or task priority; static or dynamic, as figured in Figure 3.7. In this section, the most known and used algorithms will be explained briefly.

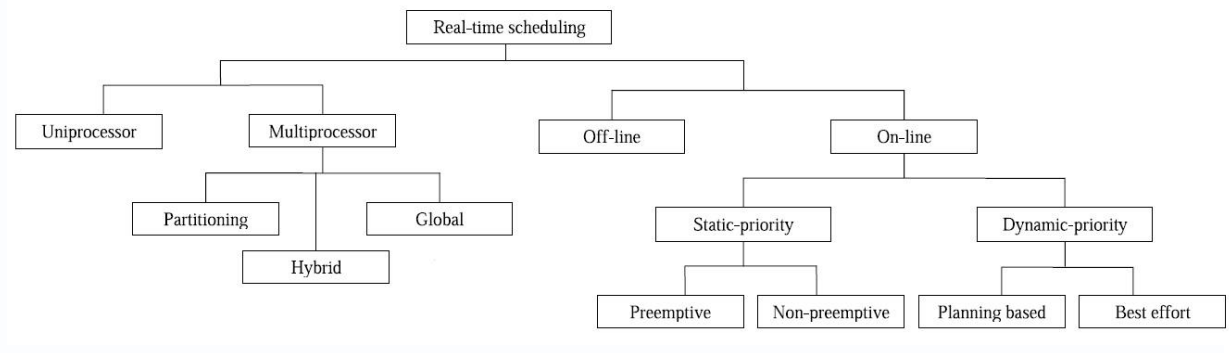

Figure 3.7. Classification of real-time scheduling algorithms [7]

3.4.1. Rate Monotonic (RM) Algorithm

Rate Monotonic is a fixed priority algorithm. In this approach, the higher priority a task has, the more CPU usage it gets. The scheduler using this algorithm always gives the processor to task having highest priority. Unique priorities are assigned to tasks at release time with respect to the cycle duration of the job and the task having shorter cycle duration or little jobs to get the higher priority. It is a preemptive and has deterministic guarantees for respond times. See [7] for detailed explanation and example about RM algorithm.

3.4.2. Earliest Deadline First (EDF) Algorithm

Dynamic priority algorithm is the most used and important for Real-time systems. The priority of a task can be changed while it is running. Contrary to Rate Monotonic algorithm, Earliest Deadline First (EDF) algorithm is based on priority changing. The priority of a task is inversely proportional to its deadline, that is, if a task's deadline is approaching, its priority also increases. In the same deadline of one or more tasks exception, priorities are delivered randomly. See [22] for detailed explanation and example about EDF algorithm.

3.5. Widely Used RTOSes

There are many different RTOSes provided by not only microprocessor manufacturers but also critical software companies in the community. In this section, we will cover the most widely known and used RTOSes briefly.

3.5.1. VxWorks

Wind River's VxWorks is one of the most popular RTOSes widely used in robotics, communications, avionics, flight simulation and other critical control applications. It provides reliability and scalability with multi-core support, including AMP and SMP operating system configurations [23].

3.5.2. QNX Neutrino RTOS

QNX software is preferred to develop solutions for life-critical systems such as air traffic control, surgical equipment and automobiles. It provides multi-tasking, preemptive scheduling, multi-threading and fast context-switching in a very small scalable size.

Furthermore, POSIX Standard API is also delivered for application developers. It is based upon on microkernel architecture and message-based inter-process communication [24].

3.5.3. FreeRTOS

FreeRTOS is a popular RTOS for embedded devices, being ported to many microcontrollers. It is distributed under the GPL. It is designed to be small and simple. The kernel is more readable and easy to port and it is only written in C. It provides multiple threading, mutexes, semaphores and software timers. Moreover, tick-less option is provided for low-power applications. OS scheduler is configurable for both preemptive and cooperative.

3.5.4. Windows CE

Windows CE is maintained by Microsoft and designed for mobile phone applications. Its real-time performance is not good enough to compare with other commercial or free RTOSs. In addition to this, because of being not POSIX compliant is not widely used in Real-time systems. It is designed for real-time developers familiar with Windows operating systems.

3.5.5. Linux for Real-time

Linux is a UNIX-like general purpose monolithic operating system kernel and distributed under GPL. It is most widely used operating system on the world in many areas such as servers, work-stations, personal computers and mobile phones. Thanks to being developed and maintained by a big community all over the world, it supports many CPU architecture and devices. This makes the Linux kernel widely used OS [12].

Many Linux kernel developers have suggested real-time enhancement modifications for the last decade. These modifications to provide the real-time responsiveness for the standard Linux kernel can be grouped in three basic approaches. Each of them is distributed as patches to the standard kernel.

- *Micro-kernel Approach*: In this approach, a new software layer is added between hardware and operating system. This additional software layer is called micro-kernel and responsible for all real-time tasks and operations such as interrupt handling, real-time scheduling and highly precise timing. This micro-kernel runs the real general

purpose operating system as normal task and dispatches all non-real-time operations to it in Figure 3.8 (b).

- *Nanokernel approach* is an alternative software layer and it can run more than one operating system simultaneously in Figure 3.8 (a). It only dispatches the interrupts. RTLinux and Xenomai are the most popular examples of this approach. RTLinux is based on micro-kernel approach, while Xenomai is based on nanokernel approach [4], [12].

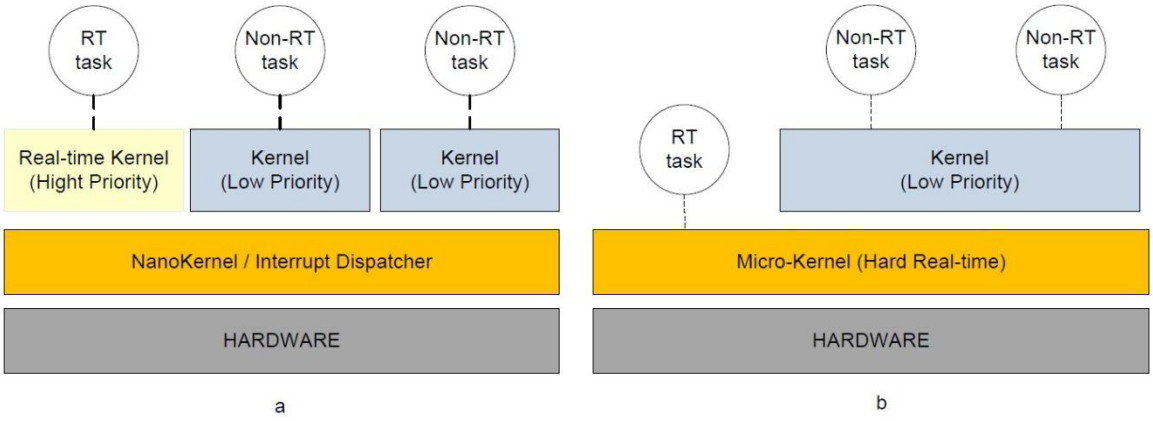

Figure 3.8. Nanokernel and Microkernel architectures

- *Kernel-level approach*: Implementing kernel extensions is the second approach. Real-time extensions such as high resolution timers, preemptive scheduling policies and high respond interrupt handling mechanism are added to the standard Linux kernel source code. The RT-patch maintained by RedHat Inc. and licensed GPL is the best and known real-time kernel-level extension for the standard Linux. Kernel-level approach is very popular and many real-time developers create and add their own extensions to the standard kernel. It is easy and more efficient way to bring real-time ability into the kernel.

Our work is a kernel-level approach. We have created some patches for the scheduling of the standard Linux kernel to handle real-time tasks and threads specially. We will explain it in details in later sections.

3.6. Embedded Systems and Multi-core Processors

An embedded system is a device with a microprocessor that is dedicated to specific tasks for special purpose. An embedded system is composed of one or more microcontrollers that is configured or programmed to perform for special tasks. In contrast to general-purpose systems, embedded systems processes special tasks according to the application area [1], [25].

In the hardware design of an embedded system, there may be some peripheral devices depending on the application that embedded system runs. For example, if the embedded system is needed to perform audio operations, there will be an audio codec microchip to perform codec conversion processes. These peripheral devices may be one or more in an embedded system, it depends on the application. In figure 2.1, the hardware block diagram of a typical embedded system is shown. In the center, there is a microcontroller generally a System on Chip (SoC) and other peripherals or components are connected to it via system buses such as SPI and I^2C [26].

A system on a chip or system on chip (SoC or SOC) is an integrated circuit (IC) that consists of all components such as USB, I^2C, DMA and MMU of a computer system inside a same single packaged chip [27]. SoCs are generally used in the area of embedded systems.

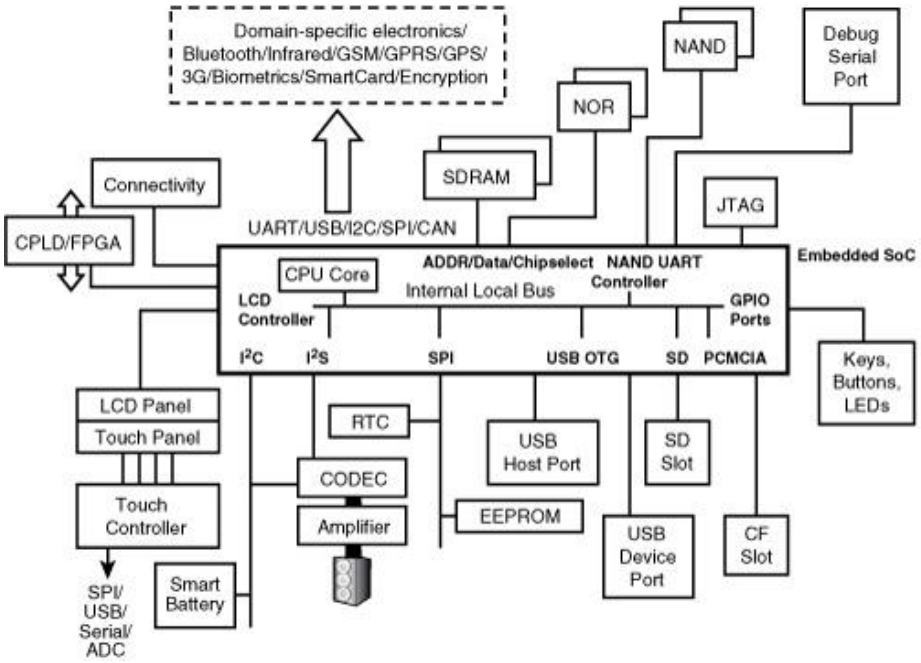

Figure 3.9. Example hardware block diagram of an embedded system [26]

Many embedded systems in the real world acquire real-time properties to operate. In other words, many of the real-time systems are embedded systems. A real-time system guarantees a worst case maximum time to complete an operation or response to an action. Embedded systems are dedicated to operate specific tasks and these are generally real-time tasks that need time constraints and precise timing. Execution of a periodic task that many embedded systems provide is one of the requirements of a real-time system.

Application areas of embedded systems are rapidly growing and their functionality and ability are rising up. Unsurprisingly, many embedded systems need not only real-time control functionality and precise timing features but also general IT functions for running non real-time applications such as database management and networking. Combination of these two functionalities is one of the most challenging problems for embedded and RT system developers. In order to overcome this problem, multi-core processors which have one or more CPU core in same SoC are begun to use in embedded systems. Moreover, processor manufacturers, nowadays, produce heterogeneous and homogenous multi-core processors to solve this problem. In heterogeneous processors, each core can run a different type of operating system to perform required functionality. For example, in a dual-core processor, a RTOS runs on one core, and a versatile or general purpose OS runs on the other. On the other hand, each core runs the same OS code, and share the main memory, peripherals and other resources in homogenous multi-core processors.

3.6.1. ARM Architecture

Advance RISC Machine (ARM) is the leader company in providing of 16/32-bit embedded RISC microprocessor design solutions. The company sells its high-performance, low-cost, power-efficient designs of their processors, peripherals and system-chip to other electronic component manufacturers such as Texas Instruments and Freescale. The Company, best known for its processor designs, does not produce physical integrated circuits (IC). ARM grant license of core to different silicon vendors like ATMEL, Texas Instruments, Samsung etc. ARM-based microprocessors are used in many areas such as handhelds, mobile phones, automation, robotics and consumer electronics [11], [28].

3.7. Asymmetric-multiprocessing (AMP) and Symmetric-multiprocessing (SMP)

Migration from single-core to multi-core processor brings a discussion about how to manage OS code over the cores in SoC. There are two suggested modes; asymmetric-multiprocessing (AMP) and symmetric-multiprocessing (SMP). In AMP mode, each core has its own copy of OS kernel code, and the codes are generally different from each other (heterogeneous OSes). On the contrary, in SMP mode, the same kernel code runs on each core synchronously (homogenous OSes). SMP OS dynamically balances the work between processor cores, and controls the resource sharing, e.g., main memory, between the cores. System developers choose appropriate form of multi-processing approach according to their application requirements.

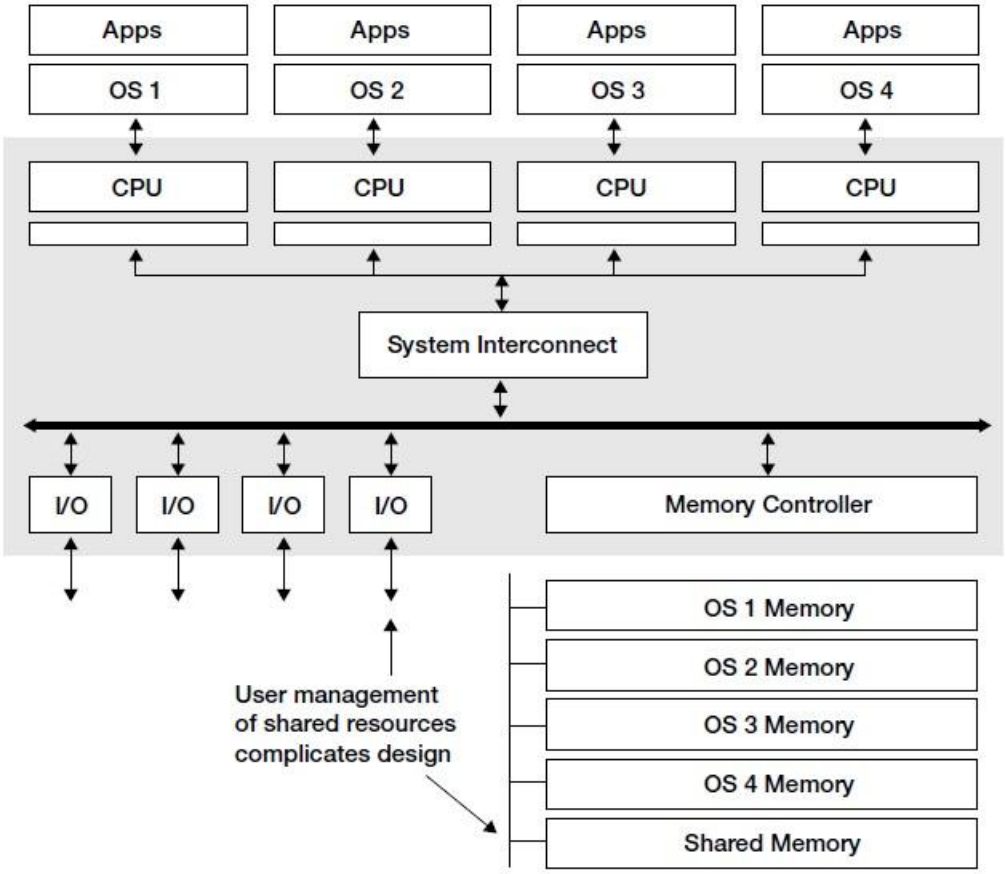

Figure 3.10. AMP Multi-core System Structure [29]

In AMP systems, each core can run not only same type OS image but also different type OS image. Therefore, in heterogeneous OSes approach where each core can run different OS

images simultaneously, processors that have AMP support are preferred [29]. In a homogeneous environment, the cores in the processor should be grouped to run different OS images efficiently in Figure 3.10.

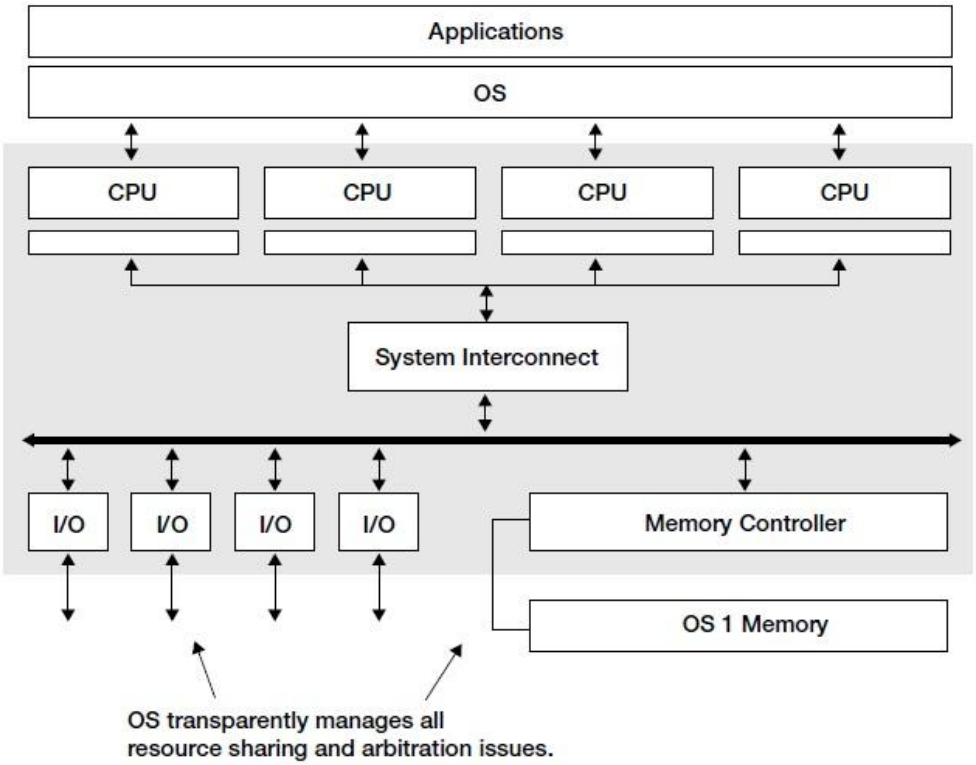

Figure 3.11. SMP Multi-core System Structure [29]

Sharing hardware resources or components in a multi-core processor can be hard to do. Especially in AMP systems where each core runs different type of OS kernel image and unawares of other OSes. SMP solves many of the issues in allocating and sharing resources by running same copy of an OS in all processor cores [29]. In SMP systems, all resources are managed and controlled from only same software that decides to give a resource to a task running in one of the cores without awareness and any input from user. SMP systems can allocate or share all resources rather than a CPU core to a specific task a shown in Figure 3.11.

4. MULTI-SCHEDULING TECHNIQUE

Multi-scheduling technique, proposed in this thesis, is a kernel-level approach to bring real-time functionality to the standard Linux kernel. Multi-scheduling is developed for SMP operating systems (OS), where each core runs the same kernel code synchronously as if the system has a single-core processor [29].

Most of the modern OSes support the SMP system. In SMP systems, one of the cores, generally CPU core-0 called primary-core is responsible for initialization of the hardware and all subsystems at boot time. After successful initialization, the same kernel code is copied to the other cores, called secondary-cores, on the SoC. Then, the tasks are assigned to cores to be run and load-balancing mechanism balances the work on the cores running the same scheduling policy. Briefly, multi-scheduling technique enables the OS to run different scheduling policies on different CPU cores of the processor.

In this chapter, an overview of the Multi-scheduling is given in Section 4.1. On the following sections, implementation details of Multi-scheduling technique in the Linux kernel will be explained. In some of the sections followed, we will brief about Linux kernel framework and functions used in the modification.

4.1. Multi-scheduling Overview

In many embedded systems, real-time tasks (rt-task) and non-real-time tasks are required to be run together. The process scheduler of an OS where rt-tasks running have to care the deadline of the processes [7]. These time sharing operations are in the concern of the scheduling algorithm. For example, in a RT scheduling algorithm, if the deadline of an rt-task is set to 10 nanoseconds, the scheduler must allow the task to run the critical operation in 10 nanoseconds before the deadline arrives. On the other hand, in general purpose OSes, it is not so critical to give the processor time to non-rt-tasks. Consequently, same scheduler system or scheduling algorithm cannot be use in both RTOS and general purpose OS. This causes to use real-time system designers to use heterogeneous operating systems approach where generally two different type of operating systems run. One of them is a RTOS for rt-tasks and the other is general purpose OS for general tasks or non-real-time tasks. However, this approach causes problems in maintenance and reliability of the system. They will be detailed in later sections.

Multi-scheduling technique is proposed to overcome these problems. It provides real-time system designers to run different schedulers in the same OS environment. Therefore, there is no need to run and strive at developing on different type of OSes. Topics covered in this chapter are about fundamentals of the Multi-scheduling technique.

4.1.1. Problem in Heterogeneous Systems

Multi-scheduling technique is developed for SMP based operating systems. The main problem is running different type of operating systems for rt and non-rt tasks in a computer system. This not only causes the system developers to spend much more time on different OS environments and APIs but also hardens the communication between rt and non-rt tasks. Although working with heterogeneous systems is hard to maintain, it is the most suitable way to bring real-time and general IT functionalities together. Multi-scheduling technique is designed to be an alternative way to do that. Thanks to this technique, system developers deal with only one operating system that supports necessary functionalities for rt and non-rt tasks. Moreover, both of them run in the same operating system so there is no need to find out an IPC mechanism between different operating systems. In this section, Multi-scheduling technique will be explained in details. Some explanations are based on the Linux kernel and the reader will be warned about that.

Most of the modern OSes support the SMP system. In SMP systems, one of the cores, generally CPU core-0 called primary-core is responsible for initialization of the hardware and all subsystems at boot time. After successful initialization, the same kernel code is copied to the other cores, called secondary-cores, on the SoC. Then, the tasks are assigned to cores to be run and load-balancing mechanism balances the work on the cores running the same scheduling policy while online. This feature is special to Linux kernel. Other operating systems which support SMP use different methods to balance the work on CPU cores.

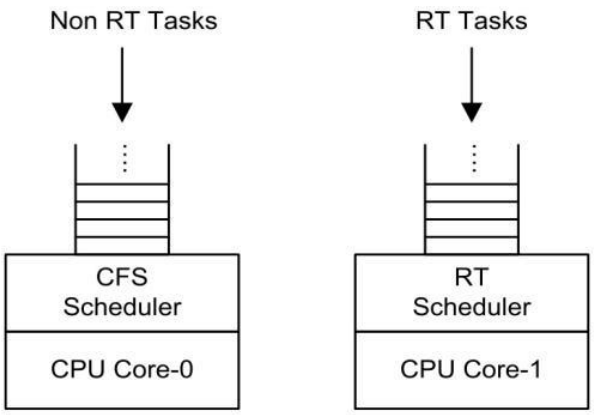

Figure 4.1. Multi-scheduling overview

Linux kernel is composed of threads a.k.a. kernel threads such as interrupt handlers and kernel services. They are also handled by the system scheduler, running periodically depended on the CPU architecture and triggered by the CPU timer. In a multi-core platform, each CPU is triggered by its timer and runs the same scheduler code. For SMP systems, all kernel threads share the same context in the main memory. Therefore, additional synchronization codes, i.e., spinlocks, are used to provide consistency between multiple threads. All tasks stored in the memory are handled by schedulers in CPUs synchronically. In Multi-scheduling technique, the shared context is copied and modified for one or more of the secondary-cores. We called these cores rt-cores where RT tasks will be handled on. Rt-cores will change its scheduling policy for RT task scheduling in their copied context. Moreover, load-balancing mechanism which balances the work between cores does not interfere with rt-cores; in other words, these cores are isolated from the other cores. For example, in Figure 4.1, CPU core-0 of a dual-core embedded system initializes and configures the hardware, and then, the core-1 runs a secondary startup code to initiate itself. In this secondary startup code for core-1, the scheduling policy of the second kernel image is changed to an RT scheduler and the load-balancing mechanism becomes aware of this. These procedures will be explained detail in next sections of this chapter.

In the following sections, the modifications to the standard Linux kernel for Multi-scheduling support will be covered. In each section, the most important modifications and additions to the standard kernel functions will be explained with code snippets, flowchart and figures. In this work, all modifications are applied to Linux Kernel version 3.4.

4.2. Linux SMP and Booting

As mentioned in previous sections, the primary-core or primary CPU, generally core-0, is responsible for booting in the standard Linux kernel on a multi-core processor. Once the power is enabled for the system, bootloader which is the first software running detects the Linux kernel and loads it to a known address on the main memory (RAM). Then, the Linux kernel is compressed itself and begins the initialization. All of these works are done by primary CPU and many of the SMP operating systems are booted by the primary-core.

Every necessary initialization are executed in *start_kernel()* function located in init/main.c file in the standard Linux kernel source code and it is executed by primary core. The most important part of this function is the invocation of *sched_init()* method residing in kernel/sched/core.c file. In this method, the settings of the scheduler such as choosing the scheduling policy are held. After all initializations are done, *kernel_init()* function is called. This function initiates the SMP feature and call *smp_prepare_cpus()* function where *secondary_startup_kernel* code is executed by each secondary core. Lastly, this function invokes *init_post()* method to run the first userspace task in Linux kernel so the system starts to service. Both *kernel_init()* and *init_post()* functions are located in init/main.c source file. These invocations in initialization are figured in Figure 4.2 with Linux kernel source file names.

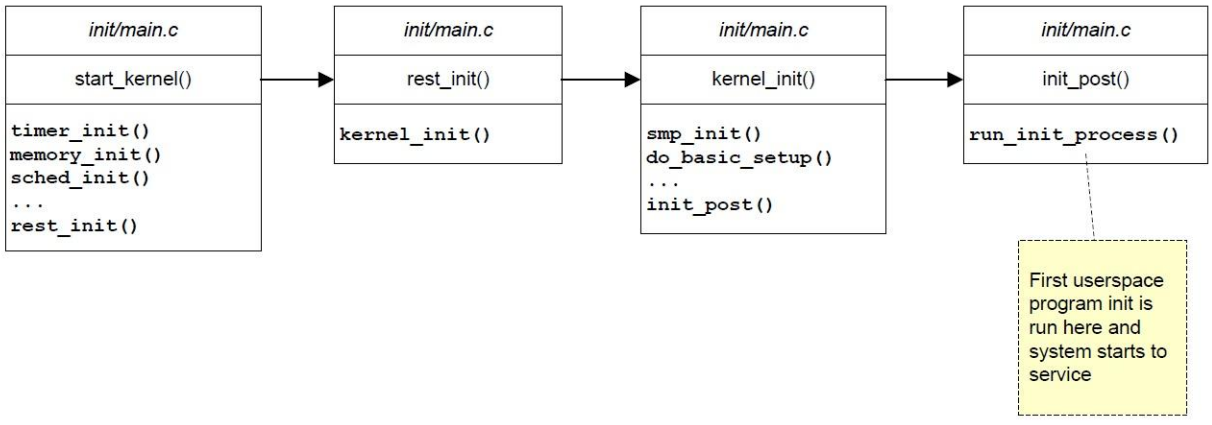

Figure 4.2. Linux kernel boot sequence

Main parts of the Linux kernel are initialized by the primary-core such as timers, scheduler and process management. Other cores, secondary-cores, run in idle while the main

initialization is in progress. When it is done, the kernel running on the primary-core signals secondary-cores to run a special minimal code called *smp_secondary_init()* to be initialized. In this function, related CPU core is just prepared to get tasks. In Figure 4.3, SMP initialization is figured on a quad-core processor. While primary CPU carries out the initialization processes, secondary cores executes NOP (No OPeration) code and waits. When the basic kernel setup is finished, secondary cores are signaled to execute *secondary_start_kernel()* function. This function is the secondary CPU boot entry. It is executed by each secondary CPU separately and some special setup and initialization are held. After successful execution, corresponding secondary CPU is switched to idle state and waits for task execution. In this stage, CPUs are added to *possible_cpus* list which available CPUs to run a task are kept in the kernel.

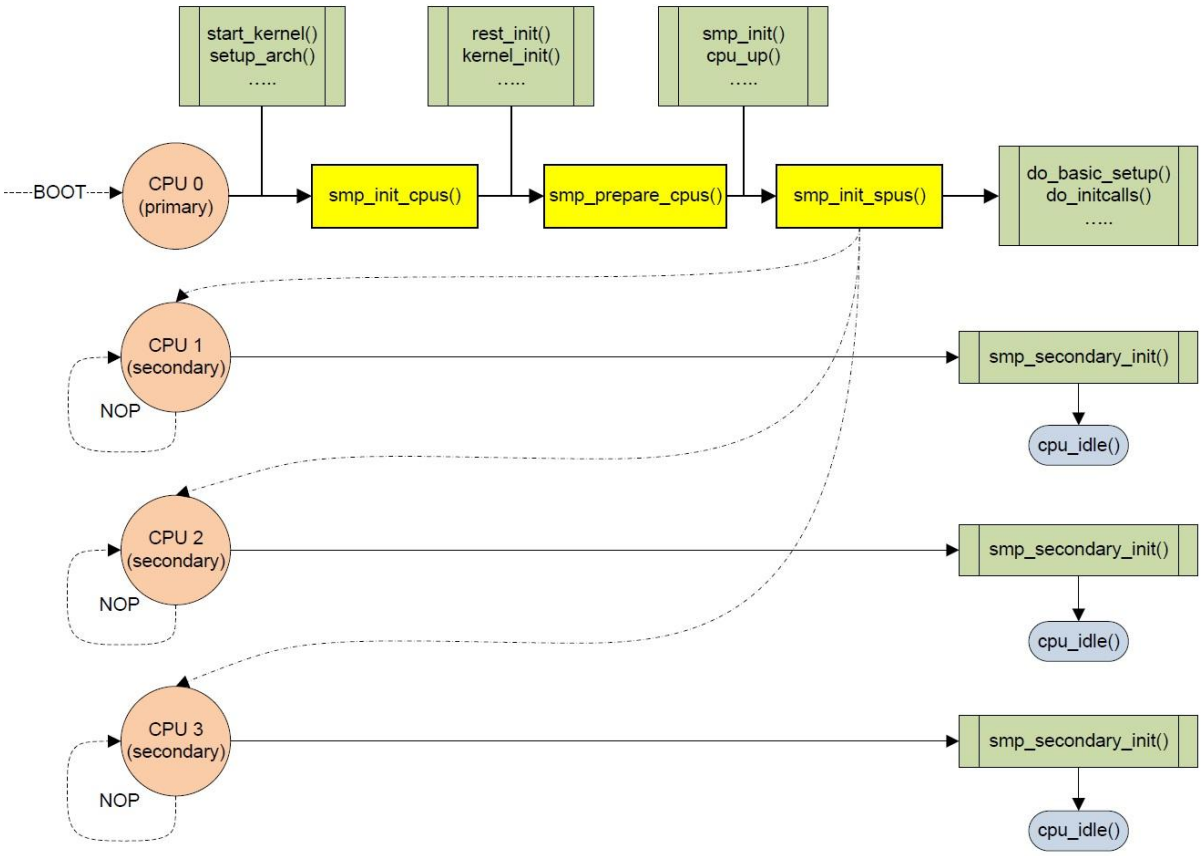

Figure 4.3. Linux SMP Boot sequence

As you see, CPU 0 takes all of the initialization jobs alone and secondary CPUs are just prepared to run tasks. The functions about SMP in Linux kernel will be worked out in details later. After all of these initialization works, every CPU core is get a task from task list of the

scheduler and processes it. Each CPU is triggered by its own timer periodically. This leads to run the timer interrupt routine which is addressed in CPU interrupt vector map according to CPU architecture. In this interrupt routine, operating system scheduler function is invoked by corresponding CPU to decide which the task will be switched on the task list. From this workflow, it is deduced that every CPU core runs the kernel scheduler function specially to switch between tasks. Every CPU core run the same kernel image and works with shared kernel data structures such as tasklist where all task data structures are stored. In order to prevent conflictions in seeking these shared data structures, mutex and semaphore based mechanisms, explained in details later, such as *spin_lock()* are used in Linux kernel [19], [28].

4.3. Secondary Startup Kernel

In our proposed approach, one or more of the secondary-cores runs a modified *secondary_startup_kernel()* code which enables to change some parts of the main kernel image. In multi-scheduling technique, one or more CPU cores are separated to run and schedule rt-tasks only. These CPUs are called as rt-cpus or rt-cores. On boot time of the Linux kernel, all CPU cores are initialized and run *secondary_startup_kernel()* function. However, rt-cores are not push to possible_cpus list. This is the main idea to separate rt-cores from other CPU cores. Consequently, the scheduler and load-balancing mechanism become not to be aware of them. This isolation method is added to *secondary_startup_kernel()* function as it is shown in green colored code in Listing 4.1.

```
asmlinkage void __cpuinit secondary_start_kernel(void)
{
      struct mm_struct *mm = &init_mm;
      unsigned int cpu = smp_processor_id();
      printk("CPU%u: Booted secondary processor\n", cpu);
      ...
      /* Do not give the rt-cores to scheduler */
      if (!is_rt_core(cpu)) {
         set_cpu_online(cpu, true);
      }
      ...
      if (is_rt_core(cpu)) {
         cpu_idle();
      }
}
```

Listing 4.1. Multi-scheduling secondary_startup_kernel code changes

Other Linux kernel functions used in this modification for Multi-scheduling are listed and explained below;

Function	smp_processor_id
Prototype	notrace unsigned int debug_smp_processor_id(void);
Description	Returns the CPU number on which the code is executed right now
File	lib/smp_processor_id.c

Function	cpu_idle
Prototype	void cpu_idle(void);
Description	It is an infinite loop used to wait the executed CPU in idle state. It also makes the CPU possible for scheduling.
File	arch/<arch = arm>/kernel/process.c

In standard SMP Linux, all kernel threads share the same memory context, in other words; every kernel data structures and variables are unique for all threads. However, the same kernel image is copied for all CPU cores and stored in different part of the main memory (RAM). For example in Figure 4.4, the copied kernel images are shown in different part of the main memory for a quad-core microprocessor. Each CPU core fetches its kernel instructions from its own memory partition. On the other hand, the kernel variables and data are shared and concurrent access can cause failures. Therefore, some software lock mechanisms such as mutex are used to prevent concurrent access in standard Linux SMP. For example, access to the tasklist data structure which the tasks are stored in is encapsulated with Linux kernel special locks; *spinlock*. Spinlock is a mutex-like mechanism that prevents the thread which is trying to get it if it is acquired by another thread before. It is widely used in Linux kernel to prevent problems that occurs in concurrent access to same resource. If a thread needs to access to a hardware or software resource that is protected by a spinlock, it must gain the spinlock at first. And, if the resource is in use by another thread, spinlock keeps it waiting. In Multi-scheduling, spinlocks are used to control to access some shared kernel data structures like CPU runqueues.

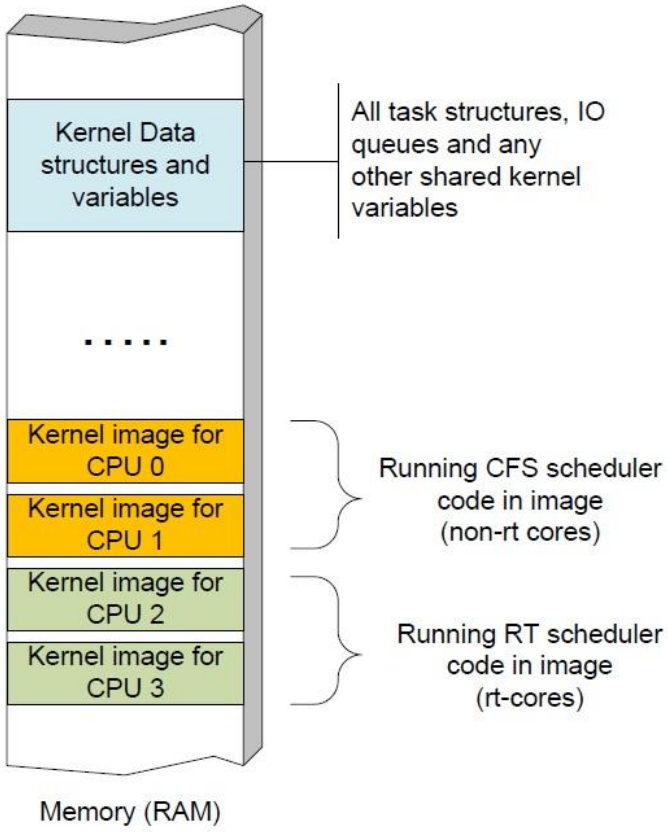

Figure 4.4. Multi-scheduling enabled Linux SMP on the main memory

The modifications in secondary startup kernel code provide to isolate rt-core to be used for all tasks in the system. *is_rt_core(int cpu)* function checks whether the given CPU is a separated CPU core for rt-task or not. Rt-cores are selected in kernel configuration which will be discussed later.

4.4. Scheduling Structures in Multi-scheduling

Multi-scheduling technique does not provide a new scheduler algorithm, it provides to run different type of schedulers in same system instead of running different type of operating systems. In standard Linux kernel, Completely Fair Scheduler (CFS) is the default scheduler for all tasks. The Completely Fair Scheduler (CFS) is added to Linux kernel in version 2.6.23 to provide fair sharing of CPU resources between tasks [4] and it is default scheduling policy/algorithm.

In Linux, processes are scheduled for execution from a doubly-linked list of processes, called the runqueues (rqs). See [19], for *rq* data structure details. This data structure is unique for

each CPU cores. As you can see from the code snippet of this structure, some valuable information's such as number of switching between tasks, number of running tasks and the next runnable task are also stored. One of the most important data stored in this structure is *load_weight*. This information is used in load-balancing mechanism which will be discussed later. This runqueue structure holds the task structures running on corresponding CPU core. Every task is stored with a task data structure *task_struct* in the Linux kernel. It keeps all information such as memory context, priority and allowed CPUs to run on about a task. *cpus_allowed* data field in task structure is important for the implementation of Multi-scheduling technique. In Figure 4.5, all of these structures and its relationships are given.

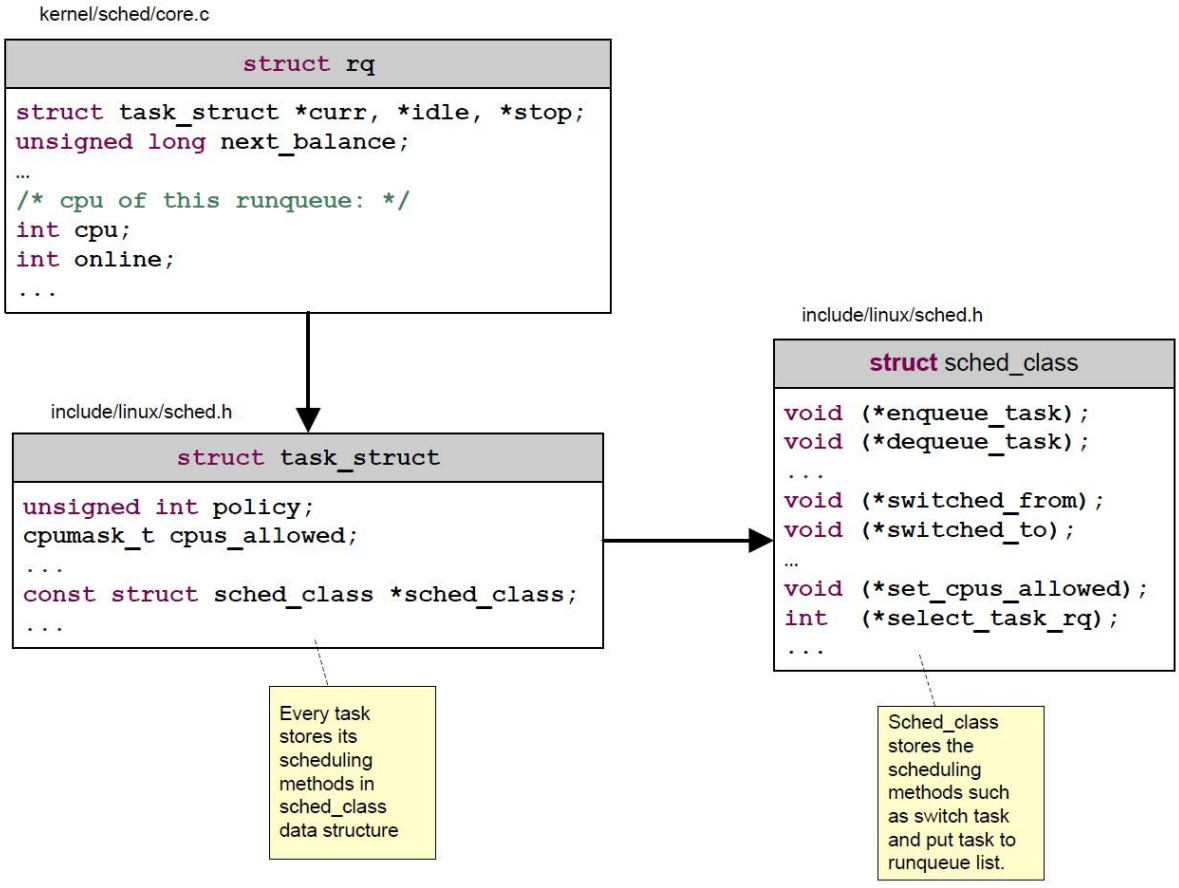

Figure 4.5. Scheduling structures and relationships used in Multi-scheduling

Some extra information like allowed CPU numbers are added to these task related structures in Linux kernel. Because of belonging to each CPU core privately, CPU type, rt-core or non-rt-core, is stored in *struct rq.* Moreover, *cpus_allowed* list is modified in *struct task_struct* to determine CPUs in which a task will run.

4.4.1. Scheduling Policy of a Task

Each task has a scheduler class which shows the scheduler algorithm or policy is used in scheduling the task. In the standard Linux kernel, there are three type of scheduling policies. SCHED_FIFO is a First-In, First-Out real-time scheduler policy. In this policy, when a task is assigned to a CPU, it is being executed until a new higher-priority task arrives. The task with highest priority always gains the CPU usage and does not relinquish it. The other policy is SCHED_RR, Round Robin real-time scheduler. This policy shares the CPU with same time duration between tasks. For example, if there are 4 tasks with same priority and execution duration is defined as 100ms, each task will gain the CPU control for 100ms in each 400ms loops. However, tasks generally have different priorities in practice. In this situation, the higher priority a task has, the more CPU time it gains by this policy. Shortly, this policy guarantees to share CPU usage between tasks fairly. SCHED_OTHER or SCHED_CFS is conventional, time-shared and default scheduler policy. It is also known as SCHED_CFS. Linux scheduler supports three different types of scheduling algorithms and it can be chosen and configured in the kernel configuration.

4.5. Real-Time Task Creation

Task creation request usually comes from userspace via system calls to Linux kernel. Linux provides *fork()* and *execv()* system calls to create new tasks on the system form userspace. *fork()* function invokes *do_fork()* kernel function which performs error checking and initial setup for the fork in process management. In this function, new task is copied from its caller task and it is ready to schedule. *wake_up_new_task()* call inserts new task to a runqueue to be scheduled. The process creation order is showed in Figure 4.6.

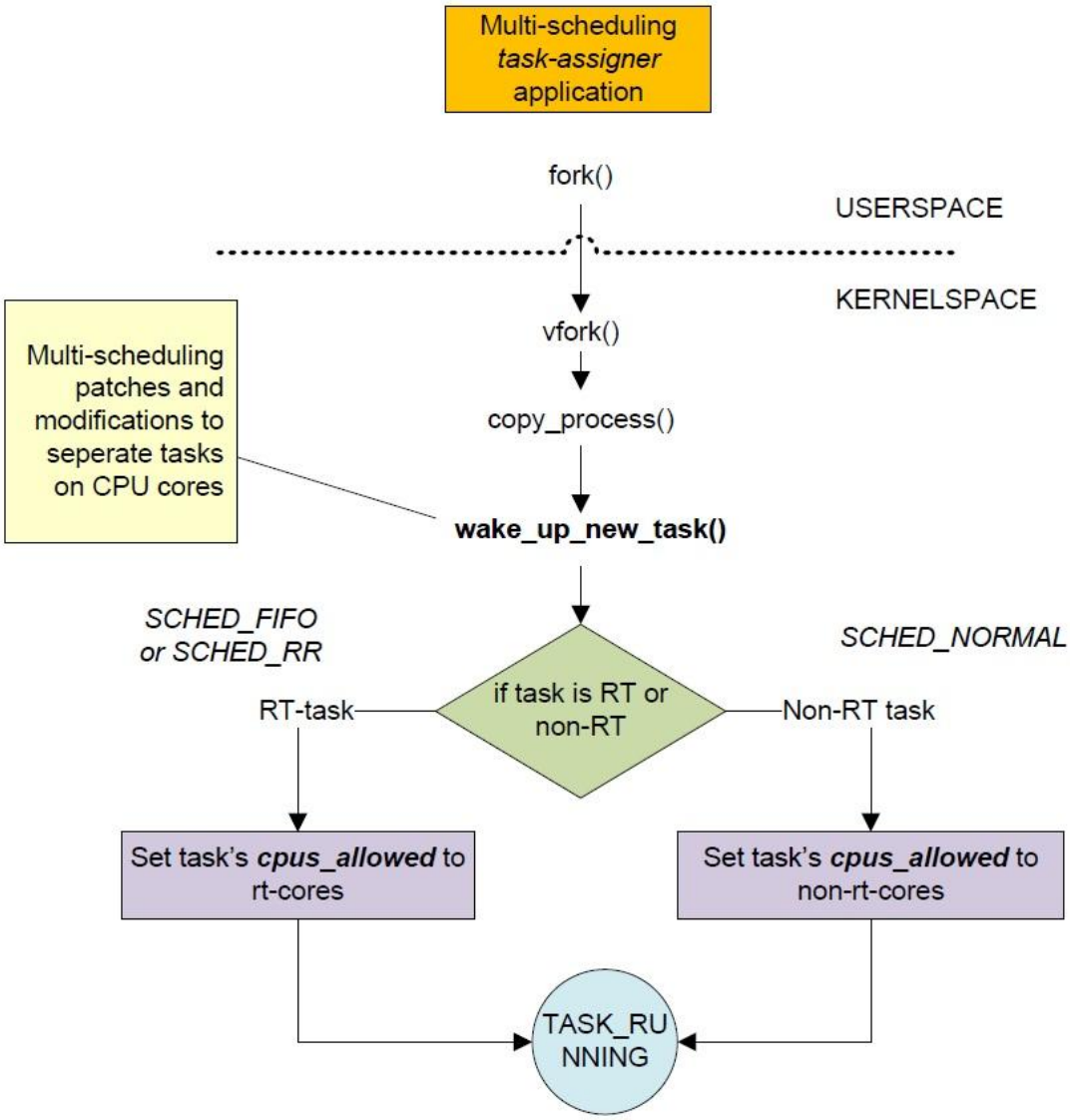

Figure 4.6. Real-time or non-real-time task creation flow in Multi-scheduling

Scheduling policy of new task is same with its parent and *init* program is the first and parent of all tasks in Linux. In standard Linux, all tasks have the same scheduling policy SCHED_CFS or SCHED_NORMAL by default. In Multi-scheduling technique, process creation is done by a special application. We have implemented a *task-assigner* utility to assign tasks either to RT or non-RT partition. The utility runs a task first and then changes its scheduling policy to SCHED_RR if the task is for RT partition; otherwise the policy is set to SCHED_NORMALs for non-RT tasks. The RT tasks are assigned to rt-cores, and the rest assigned to the other cores. In addition to forking the task structure and changing the task's scheduling policy, the *task-assigner* updates the allowed core(s) list for the task. Therefore, the tasks assigned to RT

partition are scheduled by the RT scheduler in the kernel space. See the Appendix A for more details about task-assigner application.

In kernel side, some modifications are done in *wake_up_new_task()* function executed when a new task is created. If a task is an rt-task; in other words, it is created by task-assigner as rt-task, it will be put on a runqueue of one of the rt-cores. New *wake_up_new_task()* function is given in Listing 4.2. The coded highlighted with green color checks the task type; rt or non-rt. The scheduling policy of rt-tasks is set to SCHED_FIFO or SCHED_RR policy, and SCHED_CFS for non-rt-tasks by task-assigner application. In kernel level, appropriate wakeup CPU which executes the task for the first time.

```
void wake_up_new_task(struct task_struct *p)
{
     unsigned long flags;
     struct rq *rq;

     raw_spin_lock_irqsave(&p->pi_lock, flags);
#ifdef CONFIG_SMP
     /*
      * Fork balancing, do it here and not earlier because:
      *  - cpus_allowed can change in the fork path
      *  - any selected cpu disappear through hotplug
      */
     cpumask_t mask;
     if (p->sched_class == SCHED_RR ||
          p->sched_class == SCHED_FIFO){
           set_task_cpu(p,select_task_rq(p,
               SD_MULTI_SCHEDULING, 0));

            cpus_clear(mask);
            cpu_set(get_rt_cores(), mask);
      }
      else {
          set_task_cpu(p,select_task_rq(p,SD_BALANCE_FORK,0));

          cpus_clear(mask);
          cpu_set(get_non_rt_cores(), mask);
      }
      p->cpus_allowed = mask;
```

```
#endif
     rq = __task_rq_lock(p);
     activate_task(rq, p, 0);
     p->on_rq = 1;
     trace_sched_wakeup_new(p, true);
     check_preempt_curr(rq, p, WF_FORK);
#ifdef CONFIG_SMP
     if (p->sched_class->task_woken)
          p->sched_class->task_woken(rq, p);
#endif
     task_rq_unlock(rq, p, &flags);
}
```

Listing 4.2. Task creation patch to Linux kernel

In Multi-scheduling technique, all new tasks are created and switched to non-rt-cores which are responsible to execute non-rt task regardless of being rt or non-rt task. In *wake_up_new_task()* function, if newly created task is an rt-task, it is assigned to one of the rt-cores. This dispatching is implemented in this function as shown in code Listing 4.2. In standard Linux kernel, every task has a field named *cpus_allowed* and it represents the cpus which the task can be executed on. Multi-scheduling technique uses this feature to separate tasks between CPU cores. See the next section to get more information about the functions used in the task creation process.

4.6. Load-balancing

In SMP based operating systems, it is desired that each CPU's work should be the same. In order to provide this fairness, there is a load-balancing mechanism to balance the total jobs between the CPUs on the system. In Linux kernel, the balancing is done by a kernel thread called *load_balance*. This thread, invoked by the scheduler in process operations such as task creation new process creation, senses and calculates the jobs on the CPUs and move the jobs or tasks between CPUs' runqueues to balance the total work [19].

To isolate rt and non-rt tasks in the same operating system, one of the important modifications in kernel-space have been applied to Linux load-balancing mechanism working on the runqueues to balance the tasks between CPU cores on the system. The load balancing mechanism for rt-cores does not interfere with the corresponding mechanism for the cores

reserved for non-real-time tasks. Consequently, the whole environment is partitioned into two separate environments; RT and non-RT. The tasks are also split into two groups, and each task runs on a corresponding core depending on its type whether it is RT or non-RT.

load_balance() function checks given CPU's workload whether there is an imbalance with other CPUs' workload or not. If there exists, It can move some tasks to other CPU's run queues. In Multi-scheduling technique, this function is needed to be modified. It must balance total real-time work between rt-cores and non real-time work between non rt-cores as it is shown in the flowchart diagram with some code details in Figure 4.7.

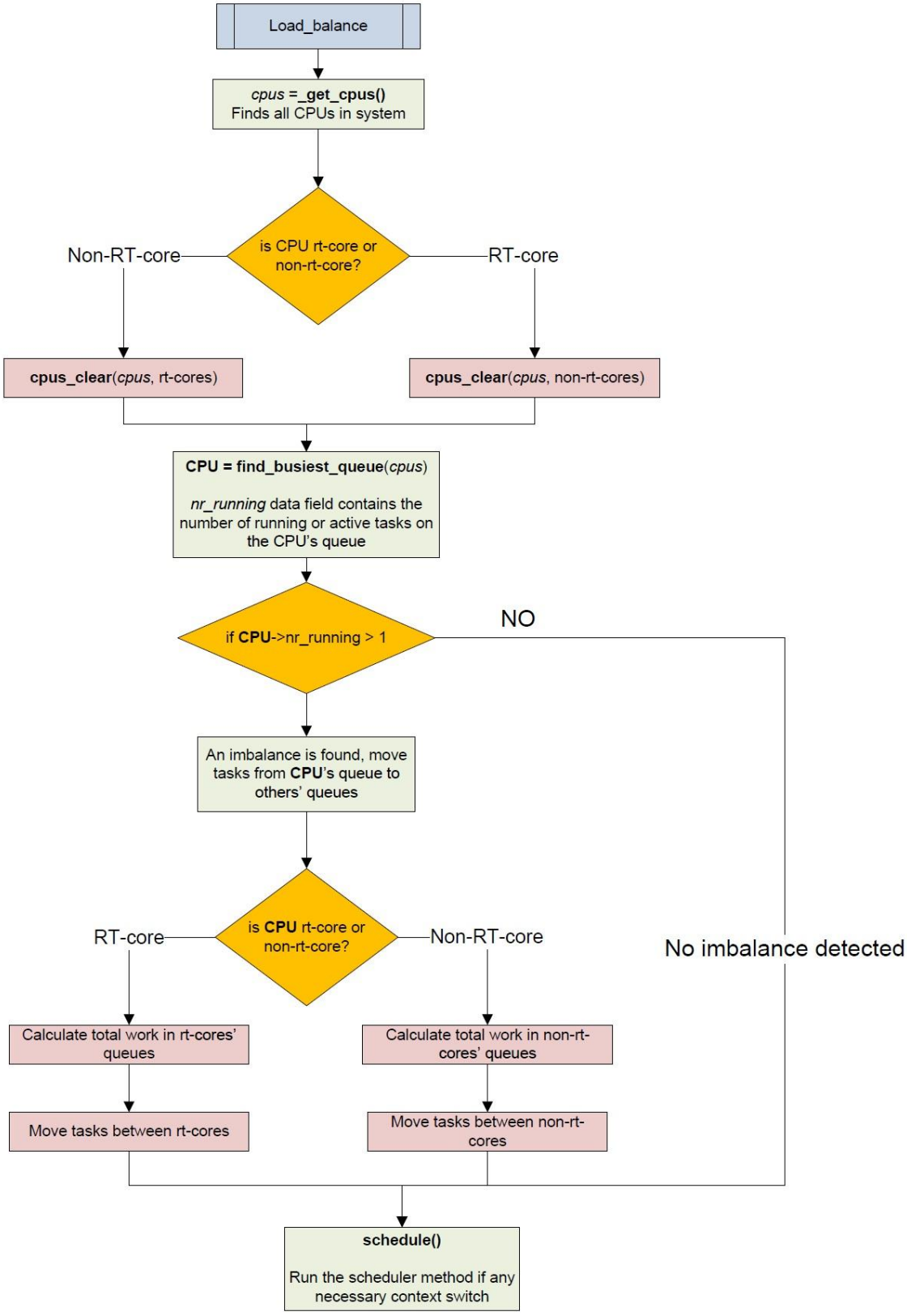

Figure 4.7. Load-balancing mechanism in Multi-scheduling

4.7. Other Extensions for Multi-scheduling

Multi-scheduling technique is developed for Linux. It is initialized and started to run in the boot process. The boot process of embedded systems is different from desktop or server PCs. When the power button is pressed, a small boot-loader software finds the OS image and loads it to the main memory and then OS initialization process begins. In the Linux SMP environment, core-0 is primary-core for initialization. When the secondary cores execute special secondary initialization kernel code, each secondary core prepares its specific resources such as MMU and caches, and then they wait for task processing in idle state while executing the idle process with PID 0. There is no need to reinitialize all resources in system because they have already been configured by primary-core. Consequently, each core on the system has its own environment containing a scheduling policy triggered by a timer specific to the core.

We have carried out some modifications to the Linux operating system in both user-space and kernel-space. In kernel-space, *secondary_start_kernel()* code has been modified to run a different scheduling policy for RT functionality. First of all, rt-cores are selected in kernel configuration for multi-scheduling and the selected core list is stored to allow tasks to run on them later. In *secondary_start_kernel(),* the shared context is copied for rt-cores and the rt-cores re-initialize the scheduling mechanism. Each rt-core changes its scheduling policy to SCHED_RR or SCHED_FAIR policies, defined in the Linux Kernel for RT applications, rather than SCHED_OTHER, aka CFS (Completely Fair Scheduling) default policy in Linux.

In Linux, each core has its own task queue (*runqueue*) for keeping the task to be run on that core. The other modifications in kernel-space have been applied to Linux *load-balancing* mechanism working on the runqueues to balance the tasks between CPU cores on the system. The load balancing mechanism for rt-cores does not interfere with the corresponding mechanism for the cores reserved for non-real-time tasks. Consequently, the whole environment is partitioned into two separate environments; RT and non-RT. The tasks are also split into two groups, and each task runs on a corresponding core depending on its type whether it is RT or non-RT. This is depicted in Figure 4.8 below.

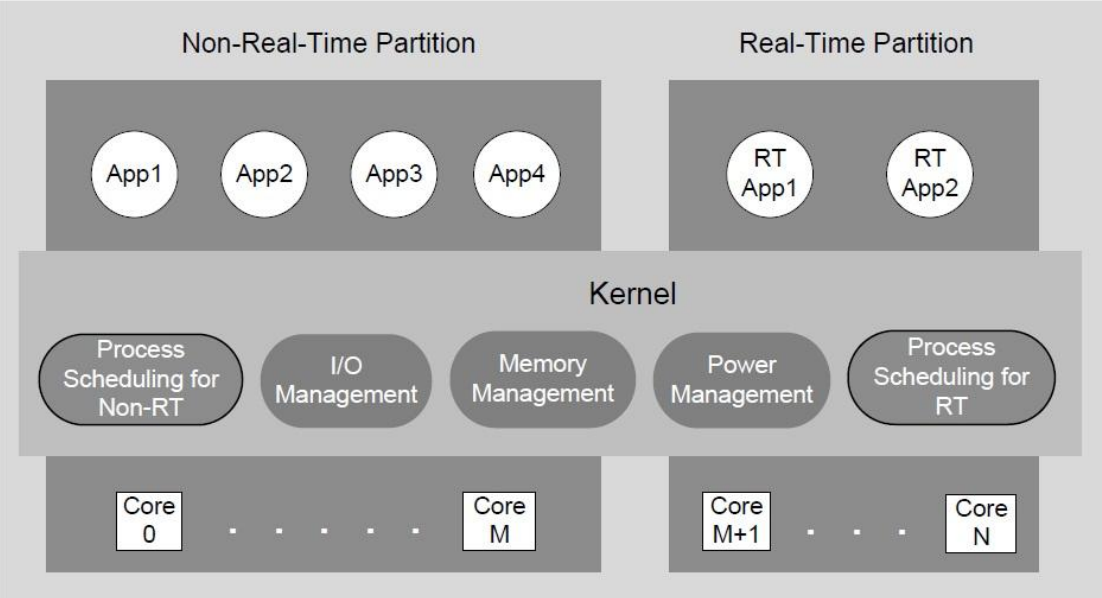

Figure 4.8. Isolation of real-time and non-real-time environments in Multi-scheduling

4.7.1. Support in Userspace

Apart from these modifications in kernel-space, some additions more are needed in user-space for Multi-scheduling. We have implemented a *task-assigner* utility to assign tasks either to RT or non-RT partition. The utility runs a task first and then changes its scheduling policy to SCHED_RR if the task is for RT partition; otherwise the policy is set to SCHED_FIFO for non-RT tasks. The RT tasks are assigned to rt-cores, and the rest assigned to the other cores. In addition to forking the task structure and changing the task's scheduling policy, the *task-assigner* updates the allowed core(s) list for the task. Therefore, the tasks assigned to RT partition are scheduled by the RT scheduler in the kernel space.

4.8. Standard Linux Kernel API Functions used in Multi-scheduling

Multi-scheduling uses the API that Linux kernel provides to develop kernel level implementations. In this section, the most important API functions which are used in the implementation of the Multi-scheduling technique will be explained.

Function	set_task_cpu
Prototype	**void** set_task_cpu(**struct** task_struct *p, **unsigned int** new_cpu);
Description	Assigns a task to a CPU. Task is linked to the chosen CPU's runqueue and

	executed on it.
File	kernel/sched/core.c

Function	select_task_rq
Prototype	**int** select_task_rq(**struct** task_struct *p, **int** sd_flags, **int** wake_flags);
Description	Finds and return CPU which is appropriate for the execution of given task. sd_flags parameter defines the CPU selection settings or sched domains. This flag is used in load-balancing to find the CPU. For example, if this flag is set to SD_BALANCE_WAKE sched domain, this task needs to be scheduled on wakes.
File	kernel/sched/core.c

Function	__cpu_clear
Prototype	static inline void __cpu_clear(int cpu, volatile cpumask_t *dstp)
Description	It gets a cpumask_t data which holds the runnable CPU numbers of a task and clears the given CPU from the task's allowed run CPU list.
File	include/linux/cpumask.h

Function	__cpu_set
Prototype	static inline void __cpu_set(int cpu, volatile cpumask_t *dstp)
Description	It gets a cpumask_t data which holds the runnable CPU numbers of a task and sets the given CPU to the task's allowed run CPU list.
File	include/linux/cpumask.h

In this chapter, we have covered the implementation details of Multi-scheduling technique in the standard Linux Kernel. In the next chapter, test and development environment will be covered.

5. DEVELOPMENT ENVIRONMENT

In this chapter, hardware and software development environment for this thesis work will be covered. Development environment will be explained in the view of two main aspects; target and host. Target is the platform that development is done for; and host is the platform that development is done on. At first, we will talk about the hardware platform that is used as target platform to develop and test Multi-scheduling technique on. Then, host platform and connections with target will be detailed. After the hardware is detailed, software components of the development environment will be covered. Firstly, we will explain what cross-compiling and toolchain are, and then Linux kernel compilation and configuration will be shown. Lastly, our patch-test loop will be presented.

5.1. Target and Host Platform

In Embedded Systems, target platform is the embedded hardware board for that the developments are done. On the other hand, host is the platform where the development such as compilation and configuration for target platform are carried out. Multi-scheduling technique is developed for Linux and embedded systems [30]. For this reason, the main development platform is Embedded Linux systems. In Figure 5.1, target and host systems and connections between them are shown. There are two connections between host and target system; Ethernet and serial port (UART). Ethernet is mainly used for downloading the generated code to target; and the serial or RS232 connection is for debugging and observing the behavior of the target system. In order to communicate or send commands to target board via RS232 or UART interface, serial terminal program is needed. The most widely used serial terminal applications are GtkTerm and Minicom and they are available on all desktop Linux distributions. On the other hand, Host platform's operating system is Ubuntu version 12.04 LTS, a widely used Linux distribution.

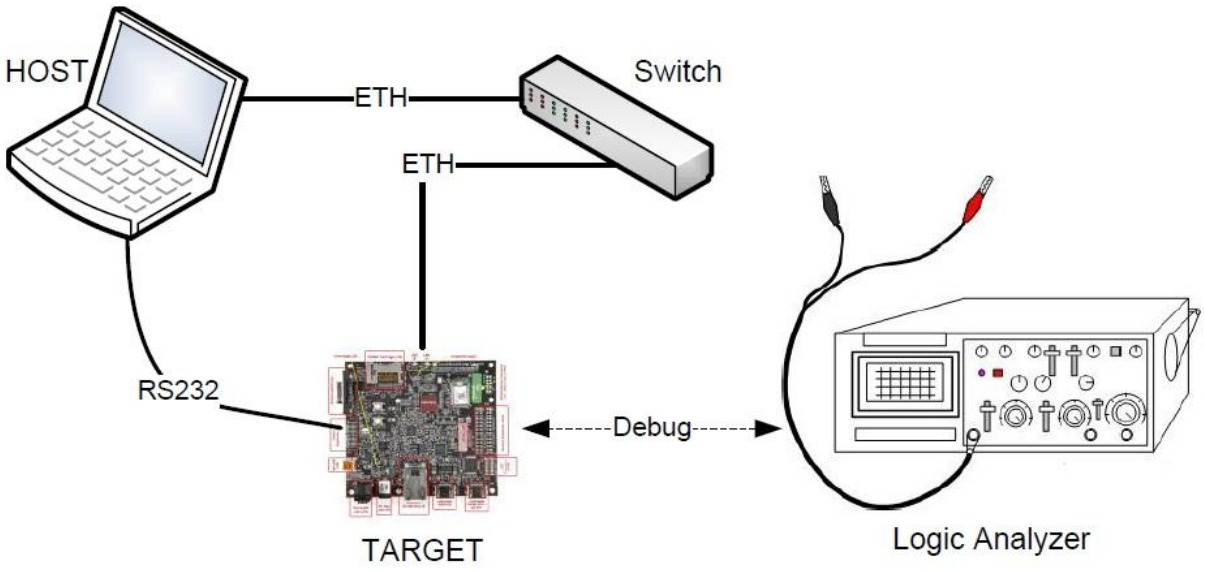

Figure 5.1. Target and Host platforms on Development environment

5.2. Target Platform

In this thesis work, our target platform is Pandaboard ES, widely used embedded Linux development board. In this section, the main features and specifications of this board will be covered. Pandaboard ES is a low-cost embedded development board that provides many I/O and media features and powered by Texas Instrument's OMAP4460 microprocessor. See Table 5.1 for a listing of the Pandaboard ES features.

Feature		
Processor	OMAP4460	
Memory (RAM)	Elpida 8Gb LPDDR2 (EDB8064B1PB-8D-F)	
PMIC	TI (TWL6030 Power Management Companion IC)	
DEBUG	14-pin JTAG	GPIO Pins
	UART via DB-9 connector	LEDs
PCB	4.5" x 4.0" (114.3 x 101.6 mm)	8 layers
Indicators	3 LEDs (two user-controlled, one overvoltage indicator)	
HS USB 2.0 OTG Port	Mini-AB USB connector, sourced from OMAP USB Transceiver	
HS USB Host Port	Four USB HS Ports, up to 500mA current out on each, two to	
Audio Connectors	3.5mm, L+R out	3.5mm, Stereo In
SD/MMC Connector	6 in 1 SD/MMC/SDIO	4/8 bit support, Dual voltage

User Interface	1-User defined button	Reset Button
	SYSBOOT3 switch	
Video	DVI-D or HDMI	Optional user provided plug-in
Display	Power Connector	USB Power

Table 5.1. Pandaboard Hardware specifications table [31]

5.2.1. Pandaboard ES Architecture

Shown in Figure 5.2 is the Architectural Block Diagram of the OMAP4460 Pandaboard ES. The Platform also includes connectors that can be used for additional functionality and/or expansion purposes. These connectors are not populated on the platform, but can be installed by the user. They are indicated by the blue blocks in Figure 5.2.

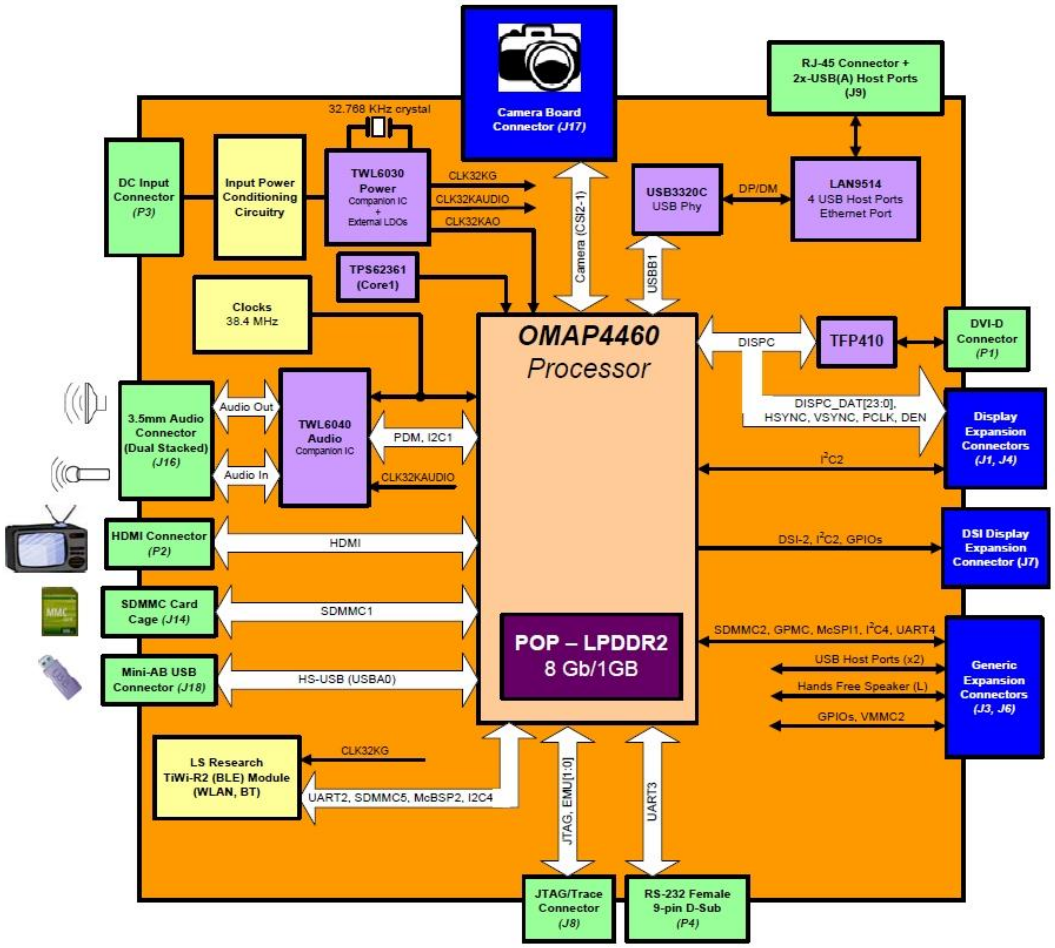

Figure 5.2. Architectural Block Diagram of Pandaboard ES [31]

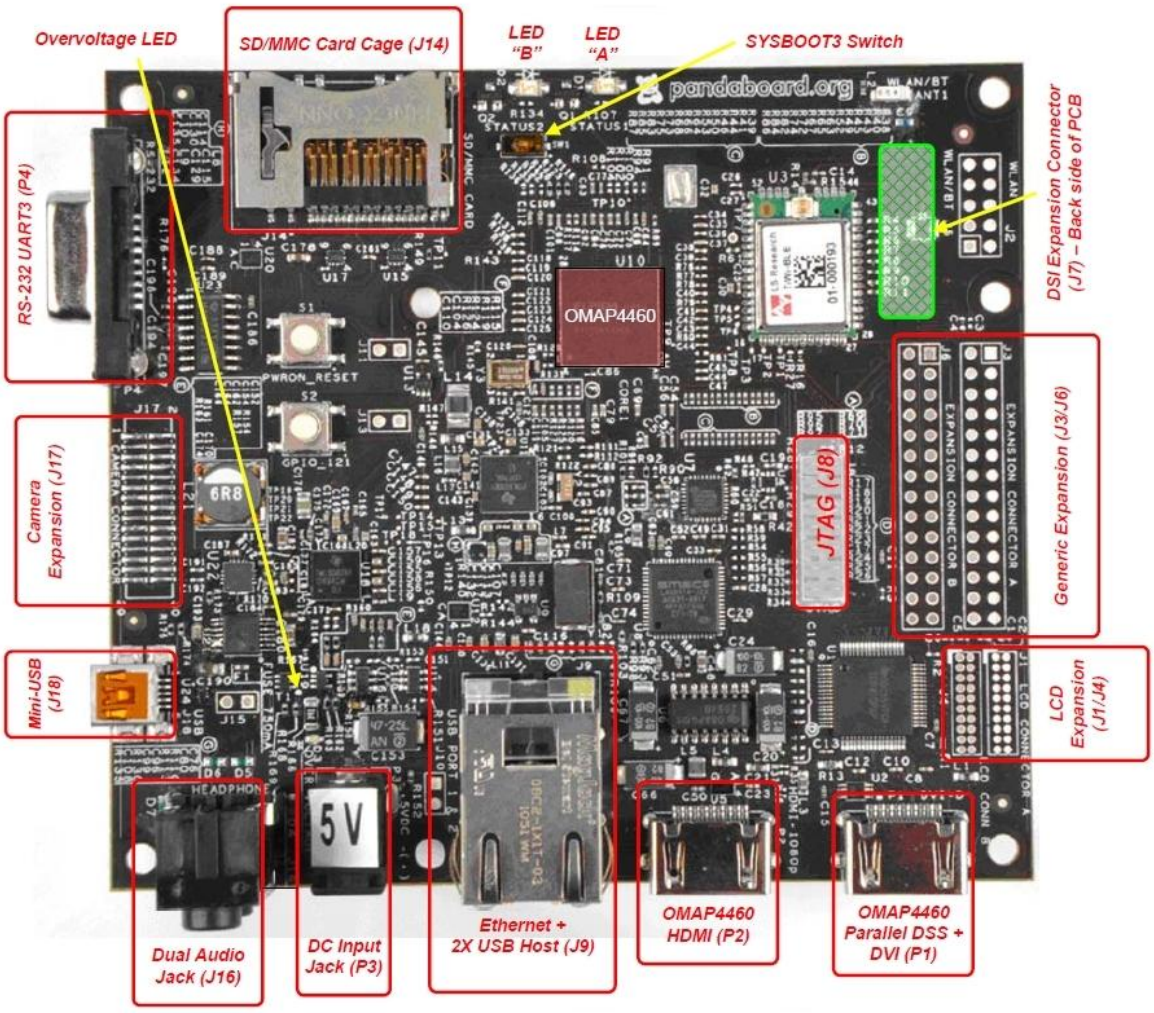

Figure 5.3. Top Real View of Pandaboard ES [31]

5.2.2. OMAP4460 Processor

Microprocessor of the target platform is the main part for Multi-scheduling technique because all execution of tasks and scheduling are handled in it. Therefore, the microprocessor of target board will be covered only in this section. OMAP4460 processor is the main component of Pandaboard ES. OMAP4460 is based on enhanced TI's OMAP architecture and uses 45-nm technology. For more and detailed information, see the OMAP4460 Technical Reference Manual (TRM) [31].

The device supports the following functions:

The device is composed of the following subsystems:
• Cortex™-A9 microprocessor unit (MPU) subsystem, including two ARM® Cortex-A9
• cores capable of operation at 1.2GHz
• Digital signal processor (DSP) subsystem
• Image and video accelerator high-definition (IVA-HD) subsystem
• Cortex™-M3 MPU subsystem, including two ARM Cortex-M3 microprocessors
• Display subsystem
• Audio back-end (ABE) subsystem
• Imaging subsystem (ISS), consisting of image signal processor (ISP) and still image
coprocessor (SIMCOP) block
• 2D/3D graphic accelerator (SGX) subsystem
The device supports high-level operating systems (OSs) such as:
• Windows™ CE, WinMobile™
• Symbian OS™
• Linux®
• Palm OS™
Video processing features
• Streaming video up to full high definition (HD) (1920 × 1080 p, 30 fps)
• 2-dimensional (2D)/3-dimensional (3D) mobile gaming
• Video conferencing
• High-resolution still image (up to 16 Mp)
Other features
• On-chip memory
• External memory interfaces
• Memory management

• Level 3 (L3) and level 4 (L4) interconnects
• System and connecting peripherals

Table 5.2. OMAP4460 processor features [31]

5.3. Cross-Compilation and Toolchain

In computer science, compiling is the converting a source code written in a computer language such as C and JAVA into executable or binary code for target machine. A special program called Compiler is designed to make this conversion. The computer where the compiler runs and source code resides on is called the host, and the computer where the compiled or generated executable program runs on is called the target. If the host and target machine are in same type, the compiling is called native compilation and the compiler is called native-compiler. On the other hand, if they are different in type, the compiling is called cross-compiling and the compiler is called cross-compiler. For example, if the host machine's architecture is x86 and the target is x86 again, native-compiler is used to generate executable program and it runs on all x86 targets. If the host is x86 machine and target is an ARM-architecture machine, cross-compiler will be used and generated program cannot run on the x86 host [30].

In embedded systems, cross-compiling method is used not only to generate applications running in embedded but also to compile the embedded operating system image. The reason of cross-compiling is that an embedded system has limited resources such as low CPU power and memory for compiling. Target naturally has limited hardware resources to make heavy compilation jobs. Therefore, these processor bounded jobs should be done in a different computer that has high power processor or host pc. However, host pc and target have in different CPU architecture types so compiled binary in a host pc, powered by x86-based processor generally, will not work in target. In order to overcome this problem, cross-development toolchains are developed to enable the host pc to compile or generate binary for computers having different type CPU architecture, for example target with ARM powered CPU.

Embedded microprocessor manufacturers provide cross-development toolchain that includes all necessary compilers, debug tools and libraries. Moreover, many featured toolchains can be found in the Linux community.

5.3.1. Toolchain components

A toolchain is composed of binary tools, compilers, libraries and some header files to generate executable images. From now on, we will use toolchain instead of cross-compiler toolchain.

5.3.1.1. Binutils

Binary utilities are the first component of a toolchain. They can run on host pc and produce or modify executable binaries for target. They are used to analyze or strip the generated binary as well as debugging and building it. The most important utilities are explained below;

- *as*, the assembler, converts assembly code to binary
- *ld*, the linker, links object code with libraries

The other tools can be given as objcopy, objdump, nm, readelf, strip, and so on.

5.3.1.2. Compilers

The most important and major part of a toolchain is the compiler. GNU Compiler Collection, shortly GCC, is the most widely used compiler collection. It supports C++, Java, Fortran, Objective-C, Ada as well as C. Moreover, it is designed to support many different CPU architectures [32].

5.3.1.3. C library

C library consists of traditional necessary functions used to develop userspace applications. It interfaces with the kernel via system calls [32]. For example, the actual implementation of *printf()*, widely used function in userspace application development, is in C library. For Embedded Linux, there are many C Library options according to size and POSIX compliant:

- *glibc* is the C library from the GNU project.
- *Embedded GLIBC* (EGLIBC) is embedded variant of the GNU C Library (GLIBC). It is optimized in size and supports for cross-compiling toolchains.
- *uclibc* is an alternate C library, which features a much smaller footprint.

5.3.2. CodeSourcery Toolchain

Toolchain selection is one the most important task before the development. There are many ways and choices to select a toolchain such as toolchain with board support package (BSP) provided by microprocessor manufacturer and prebuilt toolchains in the community. In other way, toolchain can be produced from source code according to special needs and configurations. It is also known as building a toolchain on your own. However, this can be a real pain. The easiest solution is using a prebuilt toolchain, because it has supposedly been tested by the vendor [30].

CodeSourcery is one the most widely used toolchain in Embedded Linux community. It develops Sourcery G++, an Eclipse based Integrated Development Environment (IDE) that incorporates the GNU Toolchain (gcc, gdb, etc.) for cross development for numerous target architectures. CodeSourcery provides a lite version for ARM, Coldfire, MIPS, SuperH and Power architectures. The toolchains are always very up to date. CodeSourcery contributes enhancements it makes to the GNU Toolchain upstream continually, making it the single largest (by patch count) corporate contributor [32].

In this thesis, CodeSourcery toolchain is used to compile and build Linux kernel with Multi-scheduling technique and any other userspace extensions. In the next section, downloading and installing of CodeSourcery toolchain to the host platform will be covered.

5.3.3. Downloading and Installing Toolchain

CodeSourcery ARM Compiler should be used for building different kernel distribution and software releases on ARM architecture platforms. CodeSourcery toolchain 2010-q1 release is used. Downloading and installing the toolchain to the host pc is explained step-by-step below.

- Download tarball from the link and untar to an appropriate directory in the host pc

```
kadir@kadirpc:~$ wget -c
http://www.codesourcery.com/sgpp/lite/arm/portal/package6488/
public/arm-none-linux-gnueabi/arm-2010q1-202-arm-none-linux-
gnueabi-i686-pc-linux-gnu.tar.bz2
```

```
kadir@kadirpc:~$ mkdir -p ${HOME}/opt

kadir@kadirpc:~$ tar -C ${HOME}/opt -jxf arm-2009q1-203-arm-
none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2
```

- After the extraction is complete, the new location is needed to be added to your PATH persistently. In your home directory type:

```
kadir@kadirpc:~$ gedit /home/kadir/.bashrc
```

- Add the following lines to the bottom of the file:

```
export PATH=/<toolchain_folder>/bin:$PATH

export CROSS_COMPILE=”arm-none-linux-gnueabi-“
```

- Then the ARM toolchain is ready to use from command-line. And on the last line the used GCC version for the cross-compiler can be checked.

```
kadir@kadirpc:~$ arm-none-linux-gnueabi-gcc -v

Using built-in specs.

Target: arm-none-linux-gnueabi

Configured              with:/home/kadir/2010q1-release-linux-
lite/src/gcc-4.4-2010q1/configure --build=i686-pc-linux-gnu –

...

Thread model: posix

gcc version 4.4.1 (Sourcery G++ Lite 2010q1-202)
```

6. DEVELOPMENT OF MULTI-SCHEDULING IN LINUX KERNEL

6.1. Why Linux Kernel?

Our proposal Multi-scheduling technique is an extension for a SMP-featured operating system. It enables to run rt and non-rt tasks without using heterogeneous operating systems. In this thesis, we have implemented Multi-scheduling in the standard Linux kernel version 3.4.67, the most stable version when the project is started to develop. Linux is the most widely used and well documented operating system especially in embedded systems. However, the standard Linux lacks of hard real-time features and functionalities. Many frameworks and techniques have been developed to provide hard real-time ability for Linux by its wide community [12].

Heterogeneous operating systems is the most widely used approach to provide both hard real-time and general functionalities in the same system and the Linux kernel is the most preferred general operating system in this approach. However, maintaining two different operating systems in same hardware and difficulties in IPC make heavy weather of developing the system. Multi-scheduling technique is developed as an alternative to bring real-time ability to general operating system. For the reasons mentioned before, Linux is chosen for the development and implementation of the Multi-scheduling technique.

Linux kernel supports many of the CPU architectures and peripheral devices. It is written in C and provides well documented framework to make developments in kernel level. In the following sections, Linux kernel sources and building/compiling methods will be covered.

6.2. Linux Kernel Source Tree

The Linux kernel is a Unix-like operating system kernel used by a variety of operating systems based on it, which are usually in the form of Linux distributions. Linux kernel project is a good example of free and open source software and released under the GNU General Public License version 2 (GPLv2) and it is developed by contributors worldwide. At the top of the source code, there are directories containing different sub-systems which are listed;

arch subdirectory is made up of the architecture specific kernel codes. There are sub-directories for each CPU architecture type.

include directory is composed of header files that are necessary to build/compile kernel source code.

init contains the initialization procedures for the kernel

mm contains all of the memory management code. Physical/Logical address conversion is done in the code relays here.

drivers contains as all functions to manage or control I/O devices. All devices are divided into classes. For example, keyboard and mouse are classed into input sub-directory

ipc directory contains the kernel inter-process communications code.

modules directory holds built modules.

fs contains supported file systems codes. In each sub-directory, supported file system types for example ext4 and ubifs are included.

kernel contains the main kernel code or is the heart of source code. Many of the modifications for Multi-scheduling are implemented under this directory.

net contains network frameworks and implementations such as TCP/IP and IPv6.

lib is library directory that holds many useful functions and methods used in the codes in other kernel directories.

scripts directory contains the scripts to configure or build kernel.

6.3. Patching

In Computer science, a patch is a collection of changes line-by-line in a source code. Instead of releasing whole source code version by version, releasing only the changes or differences between versions is more effective and easier. Our proposal technique is also prepared as a patch to the standard Linux kernel. We have prepared two patches and applied the Linux Kernel version 3.4.67. One of the patches called Msched-P1 runs RT tasks with SCHED_RR scheduling policy and the other called Msched-P2 runs them with general Linux scheduling policy SCHED_OTHER in Table 6.1.

Msched-P1	Msched_P2
SCHED_RR scheduling policy	SCHED_OTHER or SCHED_CFS scheduling policy

Table 6.1. Multi-scheduling patches

6.3.1. Patching the Linux Kernel

Patch command is used to apply patches to source code. It is shown below that multi-scheduling patches, Msched-P1 and Msched-P2, are applied to the standard Linux kernel.

```
kadir@kadirpc:~$ patch -p1 < Msched_P1.patch
```

6.4. Linux Kernel Configuration

In this section, configuration of the Linux kernel according to the Multi-scheduling technique will be covered. Linux kernel has a special configuration environment. Source code can be configured and compiled for many types of CPU architectures and machines.

Firstly, development embedded board, Pandaboard ES, must be defined and selected in the configuration. Thanks to the Pandaboard and opensource community, the board's hardware is defined in the standard Linux source code. It is ready to use. In Listing 6.1, in the top directory of the source code, configuration for the compiler is created according to a given board name predefined in source code. It generates a .config file in the top of the source code directory and contains some definitions or variables for the compiler.

```
kadir@kadirpc:~$  make omap2plus_defconfig

kadir@kadirpc:~$  make menuconfig
```

Listing 6.1. Linux kernel configuration over command-line

In Listing 6.1, Linux kernel configuration over the command-line interface is shown. In the root or top directory of the kernel source code, a configuration menu can be opened by typing the *make menuconfig* command. The *make menuconfig* command will launch a text-based user interface with default configuration options as shown in the Figure 6.1. This user interface, developed by using *ncurses*, helps to configure the Linux kernel by selecting many software components dependent to hardware such as CPU architectures, device drivers and memory utilizations. After the configuration is done, *.config* file is saved to the top directory automatically.

6.4.1. Configuration for Multi-scheduling

Multi-scheduling technique is also added to the Linux kernel configuration pages. Msched-P1 and Msched-P2 patches create a configuration page for real-time CPU selection under the *Kernel Features* category in Figure 6.1. As mentioned before, Multi-scheduling is dependent to SMP feature of an operating system. Therefore, in Linux kernel, SMP feature must be enabled first, as shown in Figure 6.2. After enabling the SMP, Multi-scheduling technique configuration line appears in Figure 6.3.

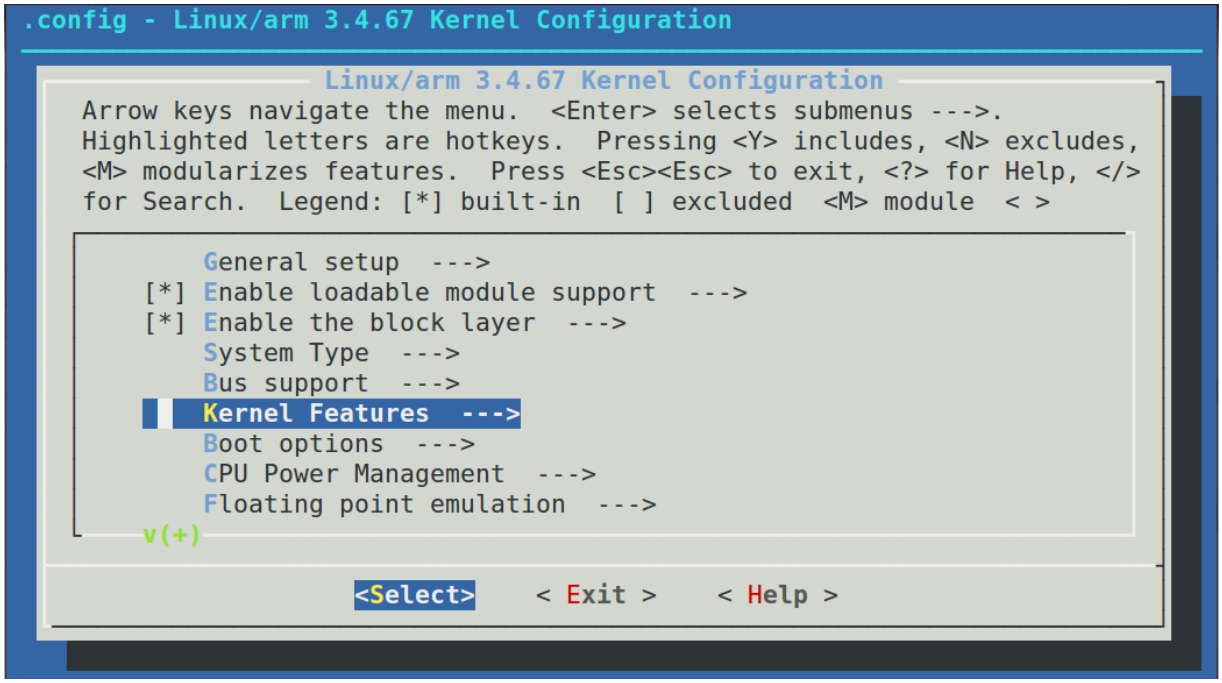

Figure 6.1. Linux Kernel Configuration Menu

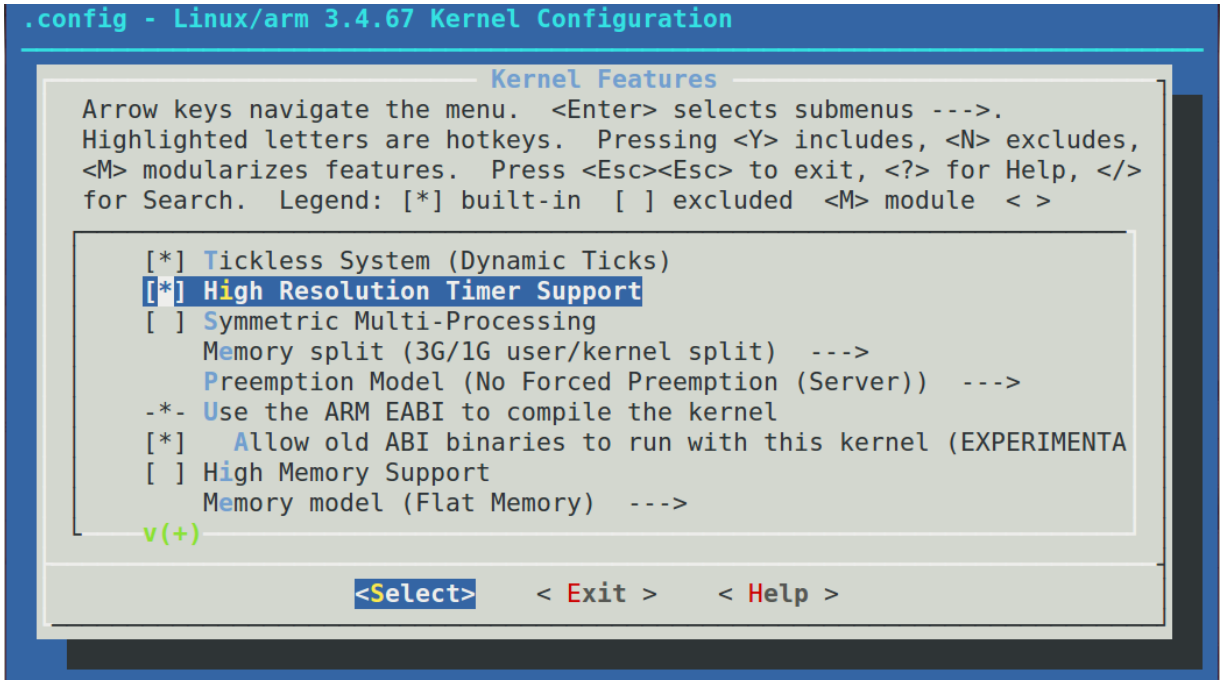

Figure 6.2. SMP kernel feature not enabled

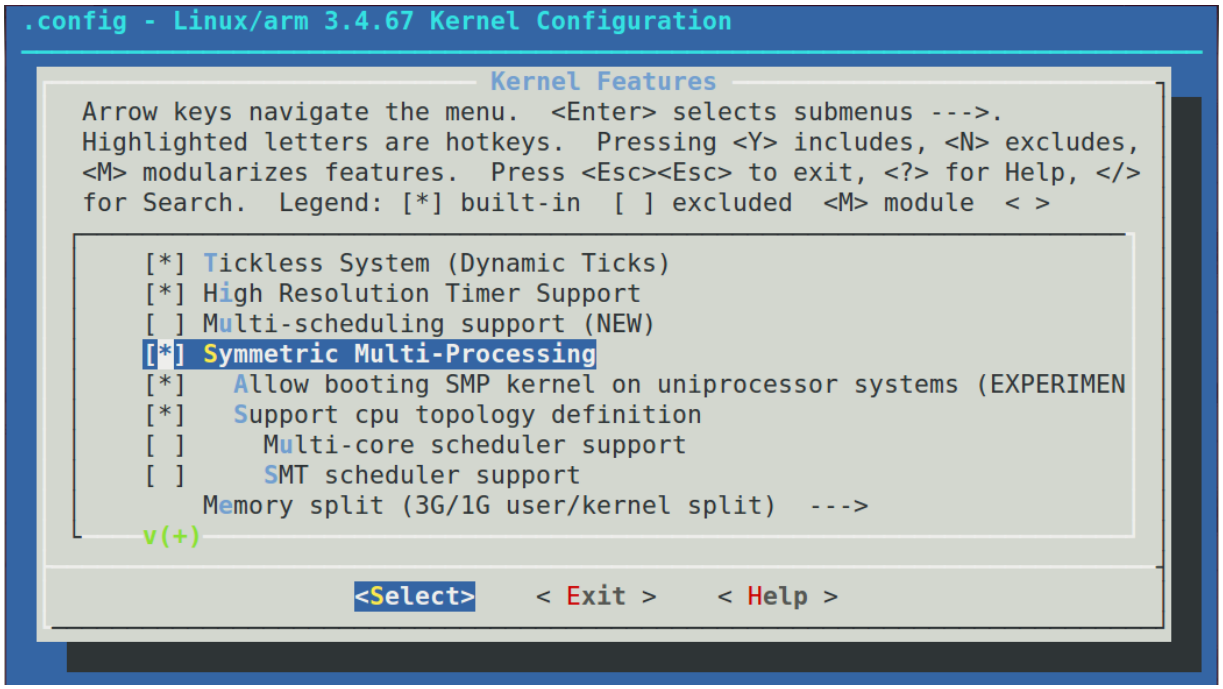

Figure 6.3. SMP enabled and Multi-scheduling support appears

In Figure 6.4, Multi-scheduling support is added to the configuration and CPU1 is chosen as real-time core automatically. Multi-scheduling patches to the configuration detects the number of the CPUs of the system by reading the configuration field CONFIG_NR_CPUS in .config

file and chooses appropriate CPU or CPUs for real-time. In dual-core processor, because of being a primary CPU, CPU0 is left and CPU1 is chosen as rt-core but it is changeable. The user can select the real-time CPUs manually; for example in Figure 6.5.

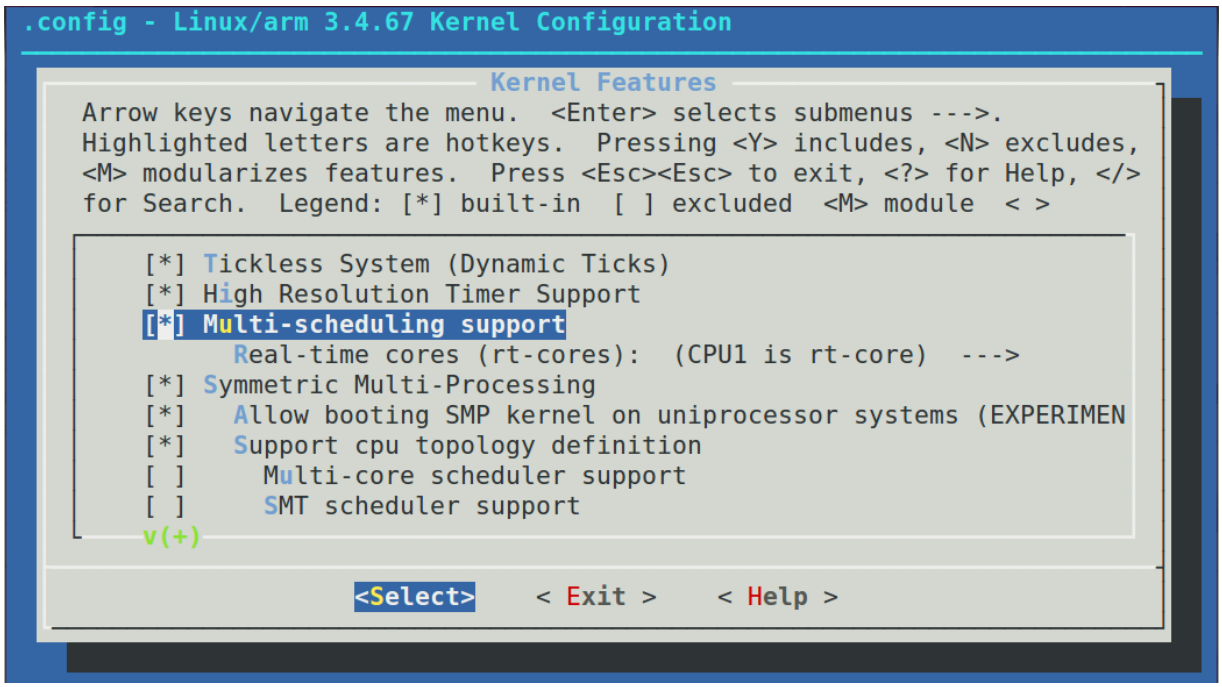

Figure 6.4. Multi-scheduling is enabled and CPU1 is chosen as rt-core automatically

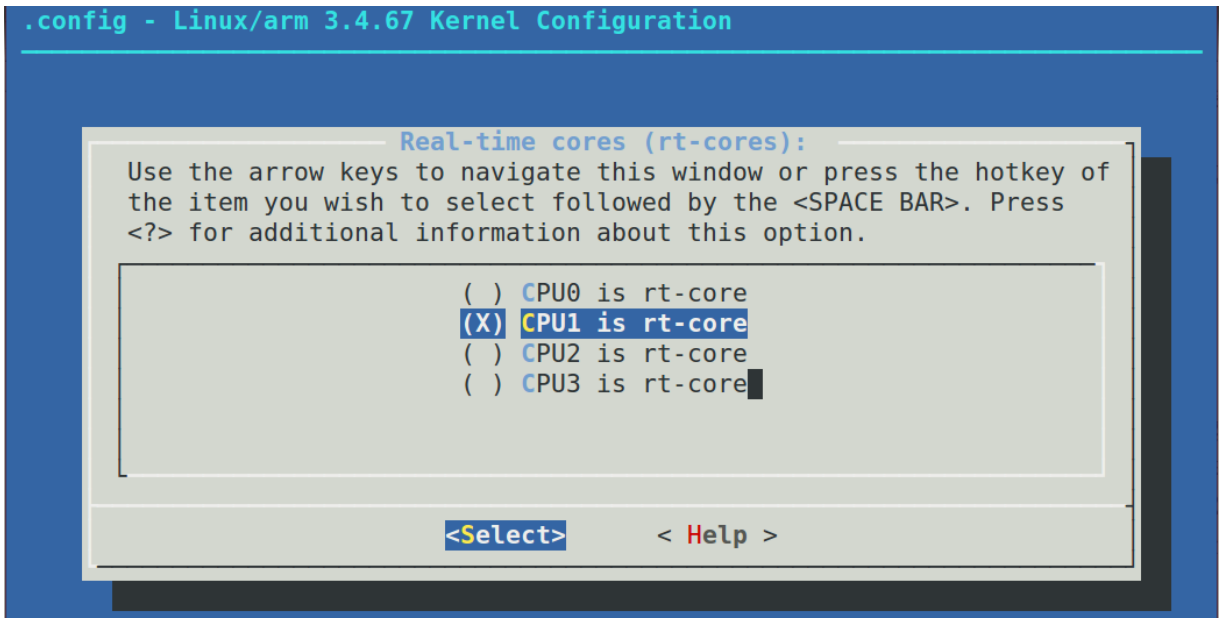

Figure 6.5. Rt-core selection in a quad-core processor

After the Multi-scheduling support is enabled, the configuration is saved .config file to be given to the compiler. In Listing 6.2, some important fields, which are relevant with the jobs done above, are shown.

```
kadir@kadirpc:~$  cat .config
....
#
# Kernel Features
#
CONFIG_TICK_ONESHOT=y
CONFIG_NO_HZ=y
CONFIG_HIGH_RES_TIMERS=y
CONFIG_GENERIC_CLOCKEVENTS_BUILD=y
CONFIG_HAVE_SMP=y
CONFIG_MULTI_SCHEDULING=y
# CONFIG_MSCHED_RT_CORE_CPU0 is not set
# CONFIG_MSCHED_RT_CORE_CPU1 is not set
CONFIG_MSCHED_RT_CORE_CPU2=y
# CONFIG_MSCHED_RT_CORE_CPU3 is not set
CONFIG_SMP=y
CONFIG_SMP_ON_UP=y
....
CONFIG_NR_CPUS=4
....
```

Listing 6.2. Content of .config file

6.5. Compiling the Kernel

In Chapter 4, development environment including cross-compiling tools is presented. In this section, the compilation process of the Multi-scheduling enabled Linux kernel will be shown. In previous section, the source code is configured and ready to compile or build. In Listing 6.3, the given command "*make ARCH=arm*" compiles the source code according to the configuration (stored in .config file in the top directory) for ARM architecture. The generated file is a compressed image of the kernel called *zImage* as shown in the Listing 6.3.

```
kadir@kadirpc:~$  make ARCH=arm
  CHK     include/linux/version.h
```

```
  CC      init/main.o
  CHK     include/generated/compile.h
  CC      init/version.o
  ....
  LD      vmlinux
  SYSMAP  System.map
  SYSMAP  .tmp_System.map
  OBJCOPY arch/arm/boot/Image
  Kernel: arch/arm/boot/Image is ready
  GZIP    arch/arm/boot/compressed/piggy.gzip
  AS      arch/arm/boot/compressed/piggy.gzip.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
```

Listing 6.3. Compiling Linux kernel

The generated kernel image can be loaded to main memory and triggered to run by a boot-loader. In this thesis, the U-Boot is used as boot-loader, so the generated image must be converted to *uImage* which includes U-Boot headers in addition to *zImage*, as shown in Listing 6.4. For more information, see [33].

```
kadir@kadirpc:~$  make ARCH=arm uImage
…
UIMAGE  arch/arm/boot/uImage
Image Name:   Linux-3.4.67-gef651f0-dirty
Created:      Sat Jun 21 21:11:26 2014
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    3682472 Bytes = 3596.16 kB = 3.51 MB
Load Address: 80008000
Entry Point:  80008000
  Image arch/arm/boot/uImage is ready
```

Listing 6.4. Generating bootable Linux kernel image

6.6. Running the compiled Kernel image

In the development environment of this thesis work, the host and target platforms are connected over the Ethernet. The generated kernel image is sent to the target via TFTP protocol. U-Boot boot-loader searches for a tftp server over the Ethernet and tries to fetch the

uImage from the tftp server. Therefore, a tftp server is needed in the host. U-Boot fetches the uImage from the server and loads it to predefined address in the main memory, and then runs it. Then, the controls are passed to the Linux kernel. In Listing 6.5, kernel boot messages are shown.

```
UBoot> TFTP from server 192.168.2.1; our IP address is
192.168.2.2
Filename 'uImage'.
Load address: 0x82000000
Loading:#################################################################
#################################################################
#################################################################
## ##############################################################
done
Bytes transferred = 3723832 (38d238 hex)
## Booting kernel from Legacy Image at 82000000 ...
   Image Name:   Linux-3.4.67-gef651f0-dirty
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3723768 Bytes = 3.6 MiB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
   Loading Kernel Image ... OK
OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
[0.000000] Booting Linux on physical CPU 0
[0.000000] Linux version 3.4.67-gef651f0-dirty (kadir@kadirpc)
(gcc version 4.4.1 (Sourcery G++ Lite 2010q1-202) ) #3 SMP Tue
Apr 8 13:40:55 EEST 2014
[0.000000] CPU: ARMv7 Processor [412fc09a] revision 10 (ARMv7),
cr=10c53c7d
[0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT
aliasing instruction cache
[0.000017] Machine: OMAP4 Panda board
[0.000017] Truncating RAM at 80000000-bfffffff to -af7fffff
(vmalloc region overlap).
```

```
[0.000103] Memory policy: ECC disabled, Data cache writealloc
[0.000119] OMAP4460 ES1.0
[0.000179] PERCPU: Embedded 8 pages/cpu @c126a000 s11520 r8192
d13056 u32768
[0.000208] Built 1 zonelists in Zone order, mobility grouping
on.  Total pages: 192784
[0.000234] Kernel command line:
ip=192.168.2.2:192.168.2.1:192.168.2.1:255.255.255.0:kadirpc:et
h0:off console=ttyO2,115200n8 root=/dev/nfs rw
nfsroot=192.168.2.1:/home/kadir/garage/tez/pandanfs
...
```

Listing 6.5. Linux kernel boot logs on Pandaboard ES

In first part of the output log of Linux kernel booting, kernel decompresses itself to the different part of the memory. Then, kernel checks its processor and architecture type for valid, and then creates initial memory page tables to store kernel structures and data. Following that, kernel enables the processor's memory management unit (MMU) to process physical/logical address conversion. Then, execution jumps to the start of the kernel's main components, *start_kernel()* in kernel/main.c. In this function, all of the initializations are carried out and *init*, first userspace program, is executed lastly as mentioned in previous chapters.

7. PERFORMANCE ANALYSIS OF THE MULTI-SCHEDULING TECHNIQUE

This chapter presents the Real-time performance and analysis techniques in the state of the art firstly, and then explains the most useful and well known tools in Linux community for the Real-time performance analyzing. Lastly, the result and discussions will be given about the performance and test results of our proposed technique.

7.1. Real-time Characteristics to Evaluate

The main purpose of an RTOS is to provide a predictable and deterministic environment for the tasks. As discussed in Chapter 3, Real-time system does not mean a high speed system to produce responds to actions it increases the quality of the system. The aim of a RTOS is eliminating the surprises and meeting the deadlines [4], [34]. In this section, the characteristics and main features expecting from a RTOS will be explained.

7.1.1. Responsiveness

A Real-time task generally has a deadline to finish or produce a result to an action and the respond must meet the deadline. Therefore, a real-time task must be switched to processor to be executed enough to meet the deadline. This depends on the real-time responsiveness of a RTOS [35]. A real-time task must be run to produce a respond as much as possible. A RTOS should give the control to a real-time task that needs to be executed as early as possible to increases its responsiveness. There are two main ways to increase the responsiveness. One of them is the fact that the real-time system is kept on idle state as much as possible. The other is reducing or minimizing the operating system's inner latencies. If the number of tasks is increasing in real-time system where rt-tasks and non-rt-tasks run on same environment, CPU idleness reduces and it causes to decrease the responsiveness [35].

Multi-scheduling technique provides a special and isolated hardware and software environment for rt-tasks. Rt-tasks are always scheduled to the rt-cores. Therefore, the real-time partition of the system is not affected by general stuff. In the next sections of this chapter, the measurement techniques and tools of responsiveness will be covered.

7.1.2. Latencies

Latency is the period between the time when an event is occurred and the time when it is being handled. There are mainly two important types of latencies for RTOSes; scheduling latency and interrupt latency. Scheduling latency is the time between an rt-task needs to be woken up and the time it actually gains to control and run. It occurs in context-switching of the tasks when a higher priority task is scheduled or the current task relinquishes the processor. The less the scheduling latency is, the much more time remains to meet the deadline for an rt-task [36]. Therefore, it directly affects the real-time performance.

Interrupt latency, on the other hand, is the time elapsed between an event occurred and when it is actually handled or processed. Interrupt latency is the most important latency to reduce for real-time systems [36], [37]. The reason is the fact that a real-time task generally runs when a hardware event occurred such as button press, or a scheduled timer event. In each situation, an interrupt service routine or function is invoked to handle the event. In Linux, interrupts are also run as a kernel thread or task. Therefore, a new task structure is created for each interrupt registration and the related thread is switched to a CPU on an event occurring.

7.1.3. Eliminating the Surprises

One of the most important features of a RTOS is to respond same timing results to events [34]. In other words, the time needed to respond does not change dramatically. It must be robust to catch the same and best timings. For example, if respond time to an event is 200 ms for an rt-task, the same time must be caught in the next occurring of the same event. There must be no surprises in a RTOS. It actually depends on the other tasks and jobs on the system. The number and load of tasks can change in a general operating system so it causes the differences in the respond time to same event.

Thanks to Multi-scheduling technique, all real-time jobs and load are isolated and do not affected by the load of general processes. This provides to decrease the surprises.

7.2. Test Results

RT performance of Multi-scheduling technique is tested on Pandaboard widely used in embedded community as reference design as discussed earlier. It has a dual-core ARM cortex-A9 powered by TI’s OMAP4460 microprocessor as discussed in Chapter 5. Therefore, one of

the CPUs runs rt-tasks and the other is for general tasks and general OS operations. We have prepared two patches and applied the Linux Kernel version 3.4. One of the patches called Msched-P1 runs rt-tasks with SCHED_RR scheduling policy and the other called Msched-P2 runs with general Linux scheduling policy SCHED_OTHER or SCHED_CFS. The standard Linux kernel is used as reference to compare the results and observe the improvements.

7.2.1. Cyclictest

Cyclictest are the most widely and frequently used real-time metric in Linux [38]. The core concept of Cyclictest is to calculate the average latency of response to a stimulus or interrupt. Cyclictest has many parameters and options to detect and calculate latencies for an operating system. For Multi-scheduling, we deal with only interrupt latency. Because of having many unrelated options, we simplified the source code to test the latency by using hardware timer interrupts via Linux high-resolution timer API. The pseudocode of our modified real-time test program called *rttest* is given in Listing 7.1. *Rttest* program tests the interrupt latency over high-resolution timers and scheduling latency. See the Appendix A for the whole source code.

```
clock_gettime((&now))
next = now + par->interval

while (!shutdown) {
    clock_gettime((&start))
    clock_nanosleep((&next))
    clock_gettime((&stop))

    diff = calcdiff(start, stop)
    # update stat-> min, max, total latency, cycles
    # update the histogram data and calculate average
    next += interval
}
```

Listing 7.1. cyclictest pseudocode

Latencies can be varied according to the work load in the system. Therefore, the standard Linux, Msched-P1 and Msched-P2 are tested on different CPU load levels. We developed a *cpustress* program to load fake jobs to CPUs. It a shell script and mainly used *cpulimit* application, widely used in Linux to stress CPU. See the Appendix D.1 for *cpustress* script. In

this latency test, CPU stress level are changed step-by-step and observe the latencies of each type of kernel in the latency test.

First, test results and program outputs are given for CPU stress level 0 as an example. Then, CPU stress level is raised by 20% for each step.

7.2.1.1.CPU Stress Level 0

For standard Linux, the average interrupt latency is 79 µs.

```
target # /root/rttest
Clock resolution (ns): 1
Measurement, please wait 1 minute...
Samples: 331275
Min latency: 52 us
Max latency: 815 us
Average latency: 84 us
target #
```

For Multi-scheduling MSched_P1 patch, the average latency is 35 µs.

```
target # task-assigner -r /root/rttest
Clock resolution (ns): 1
Measurement, please wait 1 minute...
Samples: 427399
Min latency: 22 us
Max latency: 83 us
Average latency: 44 us
```

For Multi-scheduling MSched_P2 patch, the average latency is 37 µs.

```
target # task-assigner -r /root/rttest
Clock resolution (ns): 1
Measurement, please wait 1 minute...
Samples: 428455
Min latency: 22 us
Max latency: 83 us
Average latency: 46 us
```

All results for CPU stress level = 0 are given in Table 7.1.

	Standard Linux Kernel	**Msched_P1**	**Msched_P2**
Average Latency (μs)	87	34	37

Table 7.1. Latency results in CPU stress level 0

7.2.1.2.Raising CPU Stress Level

At the second stage of Latency test, CPU stress level is increased 20% for each step. *Cpustress* tool is used to load work on CPUs shown in Listing 7.2 as example. For detailed usage of *cpustress* tool, see Appendix D.1.

```
target # cpustress.sh  20 120
-----------------------------------------------------------
CPU load script.
-----------------------------------------------------------
Date: Tue Jun 24 15:43:50 EEST 2014
-----------------------------------------------------------
Host: kadirdev
Number of CPU cores: 4.
CPU load per core: 20%.
CPU load duration: 120 seconds.
-----------------------------------------------------------
This script will run for 122 seconds.
-----------------------------------------------------------
[2014-06-24--15:43:50] => Creating
CPU_Load_kadirdev__20140624_154350.log.
[2014-06-24--15:43:50] => Starting stress for 2 seconds.
Time left: 00:00:00
[2014-06-24--15:43:53] => Running 20% CPU load for 120
seconds.
Time left: 00:00:00
[2014-06-24--15:45:54] => Log data saved in
CPU_Load_kadirdev__20140624_154350.log.
[2014-06-24--15:45:54] => This is the end!
-----------------------------------------------------------
target #
```

Listing 7.2. CPU stress level 20% for 120 seconds

The all results for each CPU stress level step are given in Table 7.2. As seen in the results, increasing in the CPU load does not affect the average latencies in both Multi-scheduling patches. On the other hand, it has dramatically increased with respect to CPU load in the standard Linux kernel. In Figure 7.1, Comparison results of Multi-scheduling patches in different CPU stress levels are given. This result says that using a Real-time scheduling algorithm such as EDF and RR instead of a normal algorithm like CFS provides a better performance, lower latency, in Multi-scheduling technique.

CPU Stress Level (%)	**Standard Linux Kernel (µs)**	**Msched_P1 (µs)**	**Msched_P2 (µs)**
20	84	44	46
40	3382	43	47
60	6570	43	47
80	10370	44	47
100	14706	46	48

Table 7.2. cyclictest Latency results in different CPU stress levels

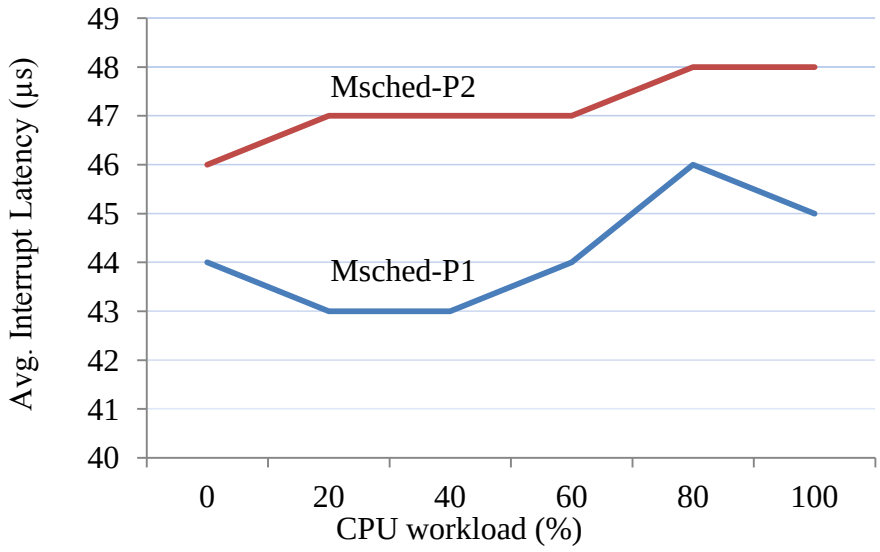

Figure 7.1. Comparison of Interrupt Latency results of Multi-scheduling patches over CPU workload

7.2.2. Responsiveness and Eliminating the Surprises

In this section, responsiveness and stability of the system to the actions are tested. Real-time applications generally wait for a signal or interrupt from outside as mentioned before. Handling these trigging actions should be as soon as possible for good responsiveness of Real-time system. Moreover, respond and execution times of a task to a trigging event/interrupt must be determined. In other words, the timings of a real-time task must not change dramatically.

In order to observe the responsiveness to hardware interrupts and changes in timings of a real-time task, *gpio-toggle* test program is developed. *Gpio-toggle* program toggles a GPIO pin on the board and estimates the duration. It raises a GPIO pin on development board, Pandaboard, and executes some mathematical commands as a work, and then pulls down the pin. See Appendix C for more information and source code of this test.

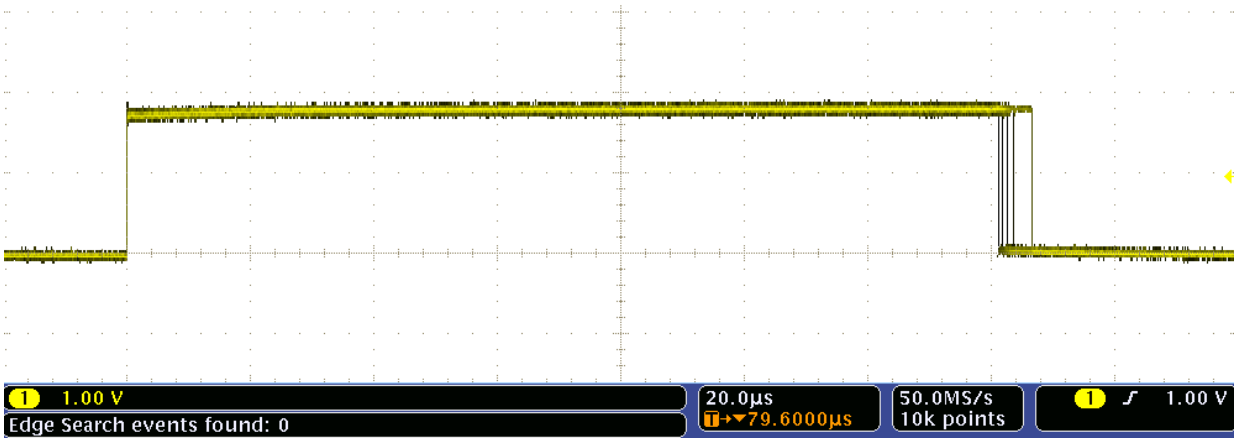

Figure 7.2. gpio-toggle test in the standard Linux Kernel

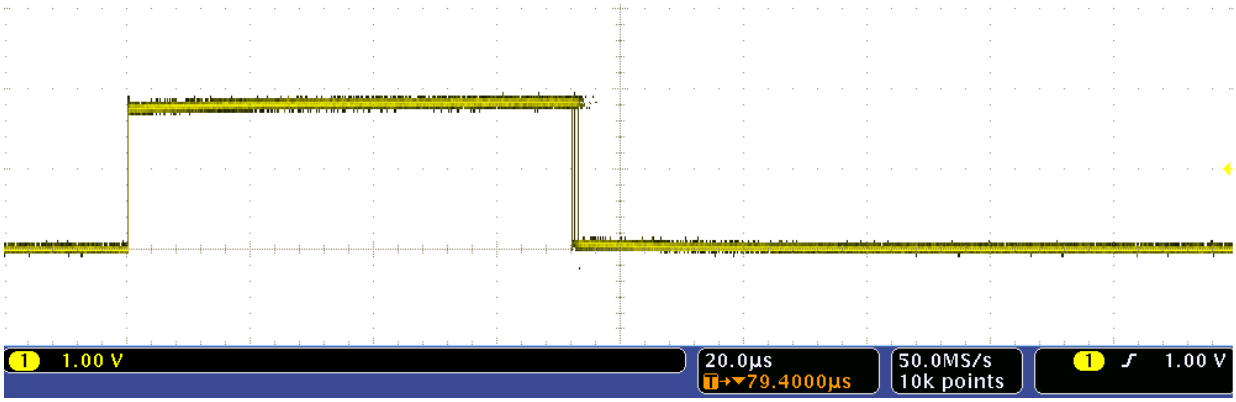

Figure 7.3. gpio-toggle test in Multi-scheduling enabled Linux Kernel

In this test, we compared the standard Linux kernel and Msched_P1 only on same CPU stress level. In Figure 7.2, the duration of toggling in the standard Linux is not stable. Time for the toggling changes in some cases as you can see on the falling edge of the output signal. On the other hand, in Figure 7.3, the duration for GPIO toggling in Multi-scheduling technique is more stable and shorter about two times than the Standard Linux. As we mentioned before, the stabilization of processing a task in any case is more important for RT systems. This toggling test shows that the processing time of RT tasks is more stable in Multi-scheduling technique than the Linux kernel.

8. CONCLUSION, DISCUSSION AND FUTURE WORK

In this thesis, we proposed a new approach Multi-scheduling to run RT and non-RT tasks in a single operating system or in the same environment. It is based on the partition of the cores in the multi-core processor into two groups, and two different environments are maintained for RT and non-RT tasks. Multi-scheduling technique only separates the cores on the system not the other resources such as main memory, USBs, GPIOs and other controllers. In order to provide a better RT and non-RT environment partition in the single OS, all resources on the system must be separated. For example, if a RT task and a non-RT task want to use the same resource, e.g., USB0, on the same time, it will reduce the overall performance and may cause the deadlocks. The current Multi-scheduling technique does not separate the other resources. In this work, we just want to show that creating two different environments may provide both RT and IT functionalities in the single OS. For future work, partitioning entire system with all resources is considered.

Multi-scheduling technique is designed for SMP systems and because of being widely used in embedded systems and well documented we decided to implement it in Linux kernel as a kernel-level real-time approach. Moreover, Linux community provides many tools and programs to test and observe real-time performance of the system. Our test platform, Pandaboard, is one of the most widely used evaluation board in the world. It has dual-core microprocessor, and it is suitable to test our proposal technique on it.

Our tests are based on most important real-time characteristic of a real-time task. We compared Multi-scheduling enabled Linux kernel and the standard Linux kernel. The interrupt latency and stability results have shown that Multi-scheduling technique can be a good approach to provide RT functionality for general OSes without using heterogeneous OSes. On the contrary of heterogeneous approach, a single OS environment is used for all tasks. This provides two main advantages to system developers. One of them is inter-process communication between RT and non-RT tasks. The other and more important advantage is about the system development and maintenance. In heterogeneous approach, system developers configure two different OSes; a general OS and a RTOS. A Failure in one of the heterogeneous OSes causes the whole system come down. Moreover, developers spend more time to learn different OS environments. This may increases the costs for production.

There are some userspace applications are developed during this work. Task-assigner application is used to create new tasks for Multi-scheduling enabled systems. In addition, a CPU stress application which compels CPU in given work levels by loading heavy jobs.

As a consequence, Multi-scheduling is a valuable technique to provide RT functionality for general purpose operating systems. It provides nearly two times better latency performance and is more robust for surprises. It may be considered as one of the most major approaches for Real-time systems in multi-core embedded systems. For future work, we want to extend Multi-scheduling technique to cover all resources in the system. Moreover, we will provide tools to control the Multi-scheduling from userspace easily.

BIBLIOGRAPHY

[1] T. Nojiri, Y. Kondo, N. Irie, M. Ito, H. Sasaki, R. Technology, and H. Maejima, Domaın Partıtıonıng Technology For Embedded Multıcore, *Journal of IEEE Computer Society*, **2009**.

[2] H. Tomiyama, S. Honda, and H. Takada, Real-time operating systems for multicore embedded systems, *2008 International SoC Designs Conference*, pp. I–62–I–67, Nov. **2008**.

[3] N. Vun, H. F. Hor, and J. W. Chao, Real-time enhancements for embedded Linux, *Proc. Inernational Conference of Parallel and Distributed Systems - ICPADS*, pp. 737–740, Dec. **2008**.

[4] S. Rostedt and D. Hart, Internals of the RT Patch, *Proceedings of Linux Symposium*, **2007**.

[5] P. Kohout, B. Ganesh, and B. Jacob, Hardware support for real-time operating systems, *Hardware/software codesign Systems*, pp. 45–51, **2003**.

[6] Y. Zhang, N. Guan, Y. Xiao, and W. Yi, Implementation and empirical comparison of partitioning-based multi-core scheduling, *Industrial Embedded Systems*, pp. 1–8, **2011**.

[7] A. Mohammadi and S. Akl, Scheduling algorithms for real-time systems, *Sch. Comput. Queen's Univ. Tech.* , **2005**.

[8] P. Regnier, G. Lima, and L. Barreto, Evaluation of interrupt handling timeliness in real-time Linux operating systems, *ACM Special Interest Group on Operating Systems (SIGOPS) Oper. Syst. Rev.*, vol. 42, no. 6, p. 52, Oct. **2008**.

[9] K. Song and L. Yan, Improvement of real-time performance of Linux 2.6 kernel for embedded application, *IFCSTA 2009 Proc. - 2009 Int. Forum Computer-Science and Applications*, vol. 2, pp. 71–74, **2009**.

[10] I. H. Say, A Reconfıgurable Computıng Platform For Real Tıme Embedded, no. September, **2011**.

[11] C. Y. Bi, Y. P. Liu, and R. F. Wang, Research of key technologies for embedded Linux based on ARM, *ICCASM 2010 - 2010 International Conference on Computer Application System Modeling*, vol. 8, no. Iccasm, pp. 373–378, **2010**.

[12] F. Proctor, Introduction to Linux for Real-Time Control, pp. 1–78, **2002**.

[13] Y. Zhang, C. Gill, and C. Lu, Real-Time Performance and Middleware for Multiprocessor and Multicore Linux Platforms, *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, no. 1, pp. 437–446, Aug. **2009**.

[14] P. Tan, Task Scheduling of Real-time Systems on Multi-Core Architectures, *2009 Second International Symposium on Electronic Commerce and Security*, pp. 190–193, **2009**.

[15] M. Shekhar, A. Sarkar, H. Ramaprasad, and F. Mueller, Semi-Partitioned Hard-Real-Time Scheduling under Locked Cache Migration in Multicore Systems, *2012 24th Euromicro on Conference Real-Time Systems*, pp. 331–340, Jul. **2012**.

[16] S. Kato, R. Rajkumar, and Y. Ishikawa, A loadable real-time scheduler suite for multicore platforms, **2009**.

[17] J. Anderson and J. Calandrino, Parallel Real-Time Task Scheduling on Multicore Platforms, *2006 27th IEEE International Real-Time Systems Symposium*, pp. 89–100, **2006**.

[18] F. Lindh, T. Otnes, and J. Wennerstrom, Scheduling algorithms for real-time systems, *Sch. Comput. Queen's Univ. Tech.* , **2005**.

[19] R. Love, *Linux Kernel Development, 3rd Edition*. **2010**, p. 441.

[20] National Instruments, What is a Real-Time Operating System (RTOS)?, Available: http://www.ni.com/white-paper/3938/en/. (Jun, **2014**).

[21] J. H. Anderson, J. M. Calandrino, and U. C. Devi, Real-Time Scheduling on Multicore Platforms, *12th IEEE Real-Time and Embedded Technology and Applications (RTAS'06) Symposium*, pp. 179–190, **2006**.

[22] D. Faggioli and F. Checconi, An EDF scheduling class for the Linux kernel, *Real-Time Linux*, **2009**.

[23] Wind River VxWorks Platforms 6 . 9, Wind River Documents, **2014**.

[24] B. R. Krten, QNX Neutrino, **2009**.

[25] D. Yun, S. Kim, and S. Ha, A parallel simulation technique for multicore embedded systems and its performance analysis, *IEEE Transactions on Computer-Aided Design Of Integrated Cırcuits And Systems*, vol. 31, no. 1, pp. 121–131, **2012**.

[26] Sreekrishnan Venkiteswaran, *Essential Linux Device Drivers [Hardcover]*. Prentice Hall; 1 edition, **2008**, p. 744.

[27] R. Rajesvari, G. Manoj, and A. P. M, System-on-Chip (SoC) for Telecommand System Design, vol. 2, no. 3, **2013**.

[28] J. He, Y. Li, W. Zhang, F. Fang, and H. Xu, Real-Time Optimization and Application of the Embedded ARM-Linux Scheduling Policy, *2011 International Conference on Information Technology, Computer Engineering and Management Sciences*, pp. 134–138, Sep. **2011**.

[29] K. Advantages, Running AMP , SMP or BMP Mode for Multicore Embedded Systems QNX Software Systems, 2012.

[30] C. Hallinan, *Embedded Linux Primer: A Practical Real-World Approach*, Prentice Hall; 2 edition, **2010**, p. 656.

[31] Pandaboard.org, Pandaboard ES Hardware System Reference Manual, **2013**.

[32] Elinux, Toolchains, **2012**, Available: http://elinux.org/Toolchains. (Jun, 2014).

[33] Denx Software, UBoot Bootloader, **2014**, Available: http://www.denx.de/wiki/view/DULG/UBoot. (Jun, 2014).

[34] R. Gumzej and W. Halang, Performance Metrics for Real-time Systems, *Life-cycle of Real-time Systems*, **2010**.

[35] N. Ward, Evaluating real-time responsiveness in dialog, *in InterSpeech*, pp. 9–10, **2006**.

[36] S. Rostedt and R. Hat, Finding Origins of Latencies Using Ftrace, **2009**.

[37] N. Hillary, Measuring performance for real-time systems, *Freescale Semiconducters Report, Novemb.*, **2005**.

[38] T. Knutsson, Performance evaluation of GNU/linux for real-time applications, December, **2008**.

Appendix A

Task-assigner Application Details

In Multi-scheduling technique, kernel-space must know the task is an rt-task or non-rt-task. For this reason, a userspace application is needed to create and manage the task creation operations. It will sign the task with rt or non-rt by using the sched_policy. If a task will be executed in Real-time domain, the sched_policy of the task will be SCHED_FIFO or SCHED_RR, if it is a normal or general task, its sched_policy will be SCHED_NORMAL. *Task-assigner* application does this stuff and it is the userspace extension of Multi-scheduling technique. It uses the system calls provided by *libc.* The usage and code of the application is given below.

A.1. Task-assigner Source Code

```
/*
Task-assigner application for Multi-scheduling
*/

#include <stdio.h>
#include <stdlib.h>
#include <sched.h>

static const struct {
      int policy;
      char name[sizeof("SCHED_OTHER")];
} policies[] = {
      {SCHED_OTHER, "SCHED_OTHER"},
      {SCHED_FIFO, "SCHED_FIFO"},
      {SCHED_RR, "SCHED_RR"}
};

static void show_min_max(int pol)
{
      const char *fmt = "%s min/max priority\t: %u/%u\n";
      int max, min;

      max = sched_get_priority_max(pol);
      min = sched_get_priority_min(pol);
      if ((max|min) < 0)
            fmt = "%s not supported\n";
      printf(fmt, policies[pol].name, min, max);
}

void perror_msg_and_die(char *msg)
{
      printf(stderr,"%s", msg, (int)pid);
             exit(EXIT_FAILURE);
```

```
}

#define OPT_m (1<<0)
#define OPT_p (1<<1)
#define OPT_r (1<<2)
#define OPT_f (1<<3)
#define OPT_o (1<<4)

int main(int argc, char **argv)
{
      pid_t pid = 0;
      unsigned opt;
      struct sched_param sp;
      char *pid_str;
      char *priority = priority; /* for compiler */
      const char *current_new;
      int policy = SCHED_RR;

      opt = getopt32(argv, "+mprfo");
      if (opt & OPT_m) { /* print min/max and exit */
            show_min_max(SCHED_FIFO);
            show_min_max(SCHED_RR);
            show_min_max(SCHED_OTHER);
            fflush_stdout_and_exit(EXIT_SUCCESS);
      }
      if (opt & OPT_r)
            policy = SCHED_RR;
      if (opt & OPT_f)
            policy = SCHED_FIFO;
      if (opt & OPT_o)
            policy = SCHED_OTHER;

      argv += optind;
      if (!argv[0])
            bb_show_usage();
      if (opt & OPT_p) {
            pid_str = *argv++;
            if (*argv) {
                  priority = pid_str;
                  pid_str = *argv;
            }
            // else
            pid = xatoul_range(pid_str, 1,((unsigned)(pid_t)ULONG_MAX)>>1);
      } else {
            priority = *argv++;
            if (!*argv)
                  bb_show_usage();
      }

      current_new = "current\0new";
      if (opt & OPT_p) {
            int pol;
 print_rt_info:
            pol = sched_getscheduler(pid);
            if (pol < 0)
                 perror_msg_and_die("cant get policy”);
```

```
            printf("pid %d's %s scheduling policy: %s\n",
                        pid, current_new, policies[pol].name);
            if (sched_getparam(pid, &sp))
                 perror_msg_and_die("cant get attributes");

            printf("pid %d's %s scheduling priority: %d\n",
                        (int)pid, current_new, sp.sched_priority);
            if (!*argv) {
                  /* Either it was just "-p <pid>",
                   * or it was "-p <priority> <pid>" and we came here
                   * for the second time (see goto below) */
                  return EXIT_SUCCESS;
            }
            *argv = NULL;
            current_new += 8;
      }

sp.sched_priority=xstrtou_range(priority,0,policy!=SCHED_OTHER?1:0,99);

      if (sched_setscheduler(pid, policy, &sp) < 0)
            perror_msg_and_die("cant set scheduler");

      return 0;
}
```

The most important system call used in task-assigner is sched_setscheduler() function:

	sched_setscheduler
Function	**int sched_setscheduler**(pid_t pid, int policy, const struct sched_param *param)
Description	Sets the scheduling policy and parameters of a task given its PID (Process ID)

A.2. Task-assigner Usage and Examples

- Run given program as a real-time task (-r)

```
target # task-assigner -r <program> <args..>
```

- Move the program given by PID (Process ID) to real-time partition

```
target # task-assigner -r -p <PID>
```

- Run given program as normal task

```
target # task-assigner <program> <args..>
```

Appendix B

B.1. Rttest Source Code

```
/* Small program to test high-resolution timers
 * and scheduling latency in Unix / Linux
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

#define MAX(a,b)  (((a) > (b)) ? (a) : (b))
#define MIN(a,b)  (((a) < (b)) ? (a) : (b))

unsigned long long int timespec_diff(struct timespec *t2, struct timespec
*t1)
{
      /* Computes the time difference between 2 timespecs */

      return(t2->tv_sec-t1->tv_sec)*1000000000ULL+t2->tv_nsec-t1->tv_nsec;
}

void timespec_add_ns(struct timespec *ts, unsigned ns)
{
      ts->tv_nsec += ns;
      if (ts->tv_nsec >= 1000000000) {
            ts->tv_nsec -= 1000000000;
            ts->tv_sec++;
      }
}

int main (int argc, char **argv)
{
      struct timespec start_time, time1, time2;
      unsigned long long int jitter;
      unsigned long long int min_jit = 999999999999999ULL;
      unsigned long long int max_jit = 0ULL;
      unsigned long long sum_jit = 0ULL;
      unsigned samples = 0;
      int samps[200];

      mlockall(MCL_CURRENT | MCL_FUTURE);

      /* Display clock resolution */
      clock_getres(CLOCK_MONOTONIC, &time1);
      printf("Clock resolution (ns): %lu\n", time1.tv_nsec);
```

```
        /* Initialize the timer that will be used in nanosleep(),  */
        /* to a value of 100 us                                    */

        printf("Measurement, please wait 1 minute...\n");
        fflush(stdout);
        clock_gettime(CLOCK_MONOTONIC, &start_time);

        do {
              /* Get the date before sleeping */
              clock_gettime(CLOCK_MONOTONIC, &time1);

              /* Compute the wake-up date */
              timespec_add_ns(&time1, 100000);

              /* Sleep */
              clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &time1, NULL);

              /* Get the wake-up date */
              clock_gettime(CLOCK_MONOTONIC, &time2);

              /* skip the first second for warmup */
              if (samples >= 1) {
                    /* Compute the sleep time */
                    jitter = timespec_diff(&time2, &time1);
                    min_jit = MIN(min_jit, jitter);
                    max_jit = MAX(max_jit, jitter);
                    sum_jit += jitter;
                    samps[samples-1] = jitter/1000;
              }
              ++samples;
        } while (timespec_diff(&time2, &start_time) < 60000000000ULL);

        /* Display sleeping statistics */
        printf ("Samples: %u\n", --samples);
        printf ("Min latency: %llu us\n", min_jit / 1000);
        printf ("Max latency: %llu us\n", max_jit / 1000);
        printf ("Average latency: %llu us\n", (sum_jit / samples) / 1000);

        exit(EXIT_SUCCESS);
}
```

B.2. Rttest Usage

```
target # rttest
```

Appendix C

C.1. Gpio-toggle Source Code

```
/* compile using  "arm-linux-gcc gpio-toggle.c -lrt –Wall –o gpio-toggle"
*/
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <sched.h>
#include <sys/io.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

#define NSEC_PER_SEC    1000000000
#define GPIO32_DIRECTION      "/sys/class/gpio/gpio32/direction"
#define GPIO32_VALUE          "/sys/class/gpio/gpio32/value"
#define GPIO_EXPORT           "/sys/class/gpio/export"

/* using clock_nanosleep of librt */
extern int clock_nanosleep(clockid_t __clock_id, int __flags,
      __const struct timespec *__req,
      struct timespec *__rem);

static inline void tsnorm(struct timespec *ts) {
    while (ts->tv_nsec >= NSEC_PER_SEC) {
        ts->tv_nsec -= NSEC_PER_SEC;
        ts->tv_sec++;
    }
}

double stress(char *argv) {
   int i, len=10;
   double sum = 0;

   if (argv != NULL)
      len = atoi(argv);

   for (i=1; i<len; i++) {
      sum += pow(i, i);
   }
   return sum;
}

int main( int argc, char** argv )
{
      struct timespec t;
      struct sched_param param;
      int interval=50000;  // 50000ns = 50us, cycle duration = 100us
```

```
        int fd;
        char zero_string[] = "0";
        char one_string[] = "1";
        char buffer[32];
        unsigned char value = 0;

        // GPIO_32, pin 18 of J6.
        if ((fd = open(GPIO_EXPORT, O_WRONLY | O_NDELAY, 0)) == 0) {
            printf("Error: Can't open gpio export.\n");
            exit(1);
        }
        strcpy( buffer, "32" );
        write( fd, buffer, strlen(buffer) );
        close(fd);
        printf("Added GPIO 32.\n");


        if ((fd = open(GPIO32_DIRECTION, O_WRONLY | O_NDELAY, 0)) == 0) {
             printf("Error: Can't open gpio direction.\n");
             exit(1);
        }
        strcpy( buffer, "out" );
        write( fd, buffer, strlen(buffer) );
        close(fd);
        printf("Direction set to out.\n");

        if ((fd = open(GPIO32_VALUE, O_WRONLY | O_NDELAY, 0)) == 0) {
            printf("Error: Can't open gpio value.\n");
            exit(1);
        }
        printf("Value opened for writing.\n");

        write( fd, one_string, 1 );
        int ret = stress(argv[1]);
            //printf("...value set to 1...\n");
        write( fd, zero_string, 1 );
            //printf("...value set to 0...\n");
        return ret;
}
```

C.2. Gpio-toggle Usage

```
target # gpio-toggle
```

Appendix D

D.1. Cpustress script

CPU load script generates a desired CPU load and forces it per each core on machines running Linux. It requires stress and cpulimit to be installed on the target machine. It depends on cpulimit utility program.

```
target # ./cpuload.sh [load in percent] [duration in seconds]
target # ./cpuload.sh 25 10
```

D.2. SAR Commandline Tool

Sar collects, reports, or saves system activity information. It is used to investigate CPU activities per core. The command below prints the idleness, load and other activities on each CPU core.

```
target # sar –P ALL <interval> <count>
target # sar -P ALL 1 3
Linux 3.4.67-gef651f0-dirty (kadir) 01/01/00 _armv7l_ (2 CPU)
02:08:54  CPU  %user  %nice %system  %iowait  %steal     %idle
02:08:55  all   0.00   0.00    0.50     0.00    0.00     99.50
02:08:55    0   0.00   0.00    1.00     0.00    0.00     99.00
02:08:55    1   0.00   0.00    0.00     0.00    0.00    100.00
```

D.3. CPU Affinity (cpus_allowed)

Thanks to /proc filesystem, we can observe on which CPU core a task is running currently. For example, the command output below says that the task can run only CPU1.

```
target # cat /proc/<task's PID>/status | grep –i cpus
Cpus_allowed : 2
Cpus_allowed_list: 1
```

CURRICULUM VITAE

Credentials

Name, Surname: Abdulkadir Yaşar

Place of Birth: 29.04.1987

Marital Status: Not married

E-mail: kyasar07@gmail.com, ayasar@aselsan.com

Address: ODTU Teknokent, SATGEB-1 Bolgesi, Aselsan, Ankara/Turkey

Education

High School: Muratpaşa Lisesi, Antalya

BSc. : Computer Engineering, Hacettepe University

MSc. : Computer Engineering, Hacettepe University

Foreign Languages

English (Advanced)

Work Experience

Embedded System Designer for 3 years in Aselsan Inc., Ankara/Turkey

Areas of Experiences

Embedded and Real-time Systems, VOIP, Computer Vision, Internet of Things

Projects derivated from the thesis

-

Publications related with the thesis

A. Yasar, K. M. Imre, Multi-Scheduling Technique for Real-time Systems on Embedded Multi-core Processors, International Conference on Advances in Computing, Electronics and Electrical Technology - CEET 2014, 2-3 August 2014 (not published yet)

Meetings attended with a poster or a paper derivated from the thesis

-