İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

UNINTRUSIVELY LINUX KERNEL REAL-TIME PERFORMANCE MEASUREMENT

M.Sc. Thesis by
Mustafa ÇAYIR, Bs.

Department : **Defense Technologies**

Programme: **Information Technologies**

JUNE 2006

İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

UNINTRUSIVELY LINUX KERNEL REAL-TIME PERFORMANCE MEASUREMENT

M.Sc. Thesis by

Mustafa ÇAYIR, Bs.

(514031101)

Date of submission : 8 May 2006

Date of defence examination : 14 January 2006

JUNE 2006

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

LINUX ÇEKİRDEĞİNİN GERÇEK ZAMAN PERFORMANSININ ÖLÇÜM YÜKÜ GETİRİLMEKSİZİN ÖLÇÜLMESİ

YÜKSEK LİSANS TEZİ
Müh. Mustafa Çayır
(514031101)

Tezin Enstitüye Verildiği Tarih :	**8 Mayıs 2006**
Tezin Savunulduğu Tarih :	**14 Haziran 2006**

Tez Danışmanı :	**Prof. Dr. Bülent ÖRENCİK**
Diğer Jüri Üyeleri :	**Prof.Dr. Çoşkun SÖNMEZ (Y.T.Ü.)**
	Yrd. Doç. Dr. Turgay ALTILAR (İ.T.Ü.)

HAZİRAN 2006

FOREWORD

I am indebted to my supervisor Prof. Dr. Bülent Örencik for being helpful and sharing experiences throughout my research.

I would like to thank to my friends from Tübitak MRC Information Technologies Research Institute Bülent Özumut, Isa Taşdelen, Orhan Baykan, Engin Kurt, Veysi Öztürk and Ferhat Uçan.

Finally, I would like to dedicate my thesis to my wife Betül Çayır. I owe much to him for all her self-sacrifice, patience and support during all my education.

May 2006 Mustafa ÇAYIR

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

OS	: Operating Systems
RTOS	: Real Time Operating Systems
ISR	: Interrupt Service Routine
CPU	: Central Process Unit
POSIX	: Portable Operating System Environment Standard
OBEX	: Object Exchange Protocol
MMU	: Memory Management Unit
PIC	: Programmable Interrupt Controller
ADEOS	: Adaptive Domain Environment for Operating Systems
RTC	: Real Time Clock
PWM	: Pulse Width Modulation
AIO	: Asynchronous Input Output
HRT	: High Resolution Timer

LIST OF TABLES

LIST OF FIGURES

UNINTRUSIVELY LINUX KERNEL REAL-TIME PERFORMANCE MEASUREMENT

SUMMARY

Using of embedded systems in defense technologies is increasing. Moreover, embedded systems are getting more complicated. The simplest solution to implement a complicated embedded system is using a general purpose operating system that has extensive software support, network libraries and rich set of device drivers. Linux has many advantages to use as an embedded operating system. Linux provides lowest cost solution for the embedded systems. In addition, Linux is an open source, multi-vendor and free operating system that has the richest device driver support. As a result, Linux is getting more popularity in embedded operating systems market.

Embedded systems generally have timing constraints. Therefore, the embedded system designer must measure Linux kernel latencies and understand Linux real-time performance before use Linux as real-time embedded operating systems.

Real-time performance of a system, basically, is amount of time between when an interrupt occurs and when corresponding handler or task start execution. So, the most important thing to figure out Linux real-time performance and latencies are based on interrupt handling mechanisms.

A number of methods, referred to as measurements methods, have been developed to measure Linux kernel latencies. These existing methods run on top the target machine and use system calls or functions of measured systems. In addition, these methods have no chance to adjust measurement system timing parameters according to real world interrupt or events. Therefore, measurements of these existing methods are not give good understanding of systems real-time performance.

In this thesis, Linux kernel mechanisms that are influence on real-time performance and all available approaches to make Linux more real-time were analyzed. In addition, we developed a method to perform unintrusively measurement of Linux real-time performance. To achieve unintrusively measurement, an embedded system is developed to generate interrupts to Linux machine with adjustable timing via host machine user interface program to simulate real world events most accurately.

LINUX ÇEKİRDEĞİNİN GERÇEK ZAMAN PERFORMANSININ ÖLÇÜM YÜKÜ GETİRİLMEKSİZİN ÖLÇÜLMESİ

ÖZET

Savunma sistemlerinde gömülü sistemlerin kullanımı giderek artmaktadır ve bu gömülü sistemler giderek daha karmaşıklaşmaktadır. Karmaşık bir gömülü sistemin gerçekleştiriminde en basit çözüm genel amaçlı bir işletim sistemi kullanılmasıdır. Çünkü genel amaçlı bir işletim sistemindeki mevcut ağ kütüphaneleri, aygıt sürücüleri ve diğer birçok yazılım gömülü sistemin tasarımında hazır olarak kullanılabilecektir. Linux gömülü sistemlerde kullanımında birçok avantajı olan genel amaçlı bir işletim sistemidir. Linux açık kaynak kodlu yapısı, zengin aygıt sürücü desteği ve bir üreticiye bağlı olmamasıyla beraber düşük maliyetli bir çözüm de sunmaktadır. Bu avantajlarından dolayı, gömülü işletim sistemi pazarındaki ünü giderek artmaktadır.

Gömülü sistemlerin başarısı çoğu zaman görevlerini tamamlama sürelerine de bağlıdır. Bu nedenle, gerçek zaman isterleri olan bir gömülü sistemde Linux kullanılacak ise Linux çekirdeğinin gecikmeleri ve Linux işletim sisteminin gerçek zaman performansı ölçülmelidir.

Bir sistemin gerçek zaman performansı en genel olarak bir kesme işaretinin gelme zamanı ve ilgili işlevin çalışmaya başlaması arasında geçen süredir. Linux işletimin sisteminin gerçek zaman performansını da Linux çekirdeğinin kesme kotarıcıları ve iş çizelgeleyicisinin cevap süresi belirler. Bu nedenle, Linux işletim sisteminin gerçek zaman performansının ölçülmesi için kesme kotarıcıları ve iş çizelgeleyicisinin cevap süreleri ölçülmelidir.

Linux çekirdeğinin gerçek zaman performansını ölçmek için birçok yöntem geliştirilmiştir. Bu mevcut yöntemler, hedef sistemin üzerinde çalışmakta olup hedef sistemin kaynaklarını kullanmaktadır. Bu ölçüm verilerinin doğruluğu olumsuz etkilemektedir. Ayrıca bu ölçüm yöntemleri gerçek çalışma koşullarına en yakın ölçüm yapmak için ayarlanabilir kesme işaretleri oluşturma özellikleri yoktur. Sonuç olarak mevcut ölçüm yöntemleri ile Linux sisteminin gerçek zaman performansı tam anlaşılamamaktadır.

Bu tez çalışmasında, Linux çekirdeğindeki gerçek zaman performansını etkileyen mekanizmalar incelenmiş, Linux'un niçin katı gerçek zamanlı olmadığı ortaya konmuştur. Linux işletim sisteminin gerçek zaman performansını geliştiren yaklaşımlar incelenmiştir. Linux gerçek zaman performansını ölçüm yükü getirmeksizin ölçebilen bir sistem tasarlanmıştır. Ayrıca, mikroişlemci tabanlı bir devre geliştirilmiş olup, bu devre sayesinde gerçek koşullara uygun ayarlanabilir kesme işaretleri oluşturabilmektedir ve bu kesme işaretlerine göre Linux gerçek zaman performansı ölçülebilmektedir.

1. INTRODUCTION

Nowadays, functions of embedded systems are getting more complicated while pressure on time to market and cost is increasing. To simply achieve this, embedded systems are designed with general purpose operating systems that Linux is one of them. Linux as multi-vendor open standard free embedded operating system has extensive software support, network libraries and rich set of device drivers. So, Linux is the most popular embedded operating systems [1].

Embedded systems are generally some kind of real-time system that correctness of the system depends on also the time at which the results are produced. Linux was developed as a general purpose operating system and Linux isn't a hard real-time operating system. Therefore, real-time performance of Linux must be tested before use it in the real-time embedded systems to decide is it possible to use Linux in the system.

Real-time performance of a system, basically, is amount of time between when an interrupt occurs and when corresponding handler or task start execution. So, the most important thing to figure out Linux real-time performance and latencies is based on interrupt handling mechanisms. Interrupt handlers must be finish up quickly and not keep interrupts disabled. On the other hand, Interrupt handlers may perform lengthy tasks within a handler. These two needs conflict with each other. Linux resolves this problem by splitting the interrupt handler into two halves that are the top half responds to the interrupt and the bottom half is a routine that is scheduled by the top half to be executed later, at a safer time.

In this thesis, a measurements system is developed to unintrusively measure response time of Linux kernel interrupt handling mechanisms and Linux kernel scheduler. Linux real-time performance measurement results in understanding of Linux real-time performance.

2. EMBEDDED SYSTEMS AND REAL-TIME KERNELS

2.1. What is Real-Time?

Different definitions exist for real-time. These definitions are mostly given by specific application field. One of definitions is independent of an application domain is that a real-time system responds in a timely predictable way to all unpredictable stimuli arrivals [2]. Real-time doesn't mean real fast. Real-time means fast enough to meet its deadlines. "A real-time system is one in which the correctness of system depends on not only on the logical results, but also on the time at which the results are produced" [3]. A real-time system must satisfy bounded response-time constraints; otherwise risk severe consequences, including failure. A real-time system should response in a timely and predictable manner to its external events and the system has a bounded worst case response times [4].

Ordinary operating systems are designed to give fairness to its task and average high performance. Temporary delays and sometimes freezes of the system are acceptable. But in real-time system these occurrences are not allowed and any occurrences of them may cause degrade of correctness of the system functions [5].

Real-time systems have requirements in terms of both deadlines and jitter. For an example of defining requirements of real-time system [6], consider an industrial control application. The application consists of a sensor, a paint nozzle and valve. When the part that must painted is just in the right position the sensor alerts the system that the valve on the paint nozzle should open to allow paint to be sprayed onto the part. It would be nice that this valve open at just the right time on every time? But it is only possible on average time. Deadline of the application is the latest possible time to open the valve and still accomplish the proper painting. In this case if we miss the deadline, we won't paint the part properly. The system cannot start painting exactly predefined amount of time after the interrupt, with infinite precision.

Software for real time system can be structured [7]:

Monolithic system**:** All software for the system includes a single block of code. This structure is usually practical only for very simple systems.

Kernel based system: Kernel based system uses a real time kernel, available from a vendor or developed in house, to manage such real time entities as task and interrupts. The logic for the real time application is coded separately from the kernel, and then linked with it to form the complete real time system.

OS based system: OS based systems differ from Kernel-based systems only in the range of functions provided by an OS compared to a kernel. A real time OS provides normal OS functions (file system, User interface, etc.) in addition to the real time functions supplied by a kernel.

2.1.1. Soft real-time

Soft real-time system has average manages on its deadline. Deadlines are still important but any missing of a deadline doesn’t cause catastrophic failures.

For example, consider the playback of a motion video on your screen, during background Linux compilation job. You may notice occasional frame loss or image freeze and even longer periods if you try to start your web browser at same time. This is because of soft real-time behavior of video application that cannot provide any guarantees timing [5].

2.1.2. Hard real-time

In hard real-time systems average case performance is not enough. A hard real-time system has to satisfy all deadlines. Any missing of a deadline causes catastrophic failures. Missing of a deadline is considered to be disastrous and lead to loss vast amount of money or even human life. Examples of hard real-time systems are the shut down sequence of a nuclear reactor, life support system in medical use, engine control etc [5].

2.2. What is Real-Time Operating System?

A real-time operating system supports real-time applications and embedded systems. Real-time applications have the requirement to meet task deadlines in addition to the logical correctness of the results. A RTOS manages the system to ensure that all time critical events are processed before deadlines. An RTOS allows the system to be divided into multiple independent elements called tasks. A task is a simple program, which thinks it has the microprocessor all for itself. Each task is given a priority based on its importance. An RTOS also provides valuable services to application such as time delays, system time, message passing, synchronization, mutual-exclusion and it should be rich in inter process communication capabilities [7].

There are two types of RTOS [7]: non-preemptive RTOS and preemptive RTOS.

Non-preemptive RTOS: Each task explicitly gives up control of the CPU. Asynchronous events are handled by ISR. An ISR can make a higher-priority task ready to run but the ISR always returns to the interrupted task. The new higher-priority task will gain control of the CPU only when the current task gives up the CPU.

Preemptive RTOS: Control of the CPU is always given to the ready highest-priority task. A preemptive RTOS is used when system responsiveness is important. A preemptive RTOS can ensure that time critical tasks are performed first. When a task makes a higher-priority task ready to run, the current task is preempted and the higher-priority task is immediately given control of the CPU. If an ISR makes a higher-priority task ready, when the ISR completes, the interrupted task is suspended and the new higher-priority task is resumed.

An example execution path of preemptive RTOS:

1. A low priority task is executing.
2. An asynchronous event interrupts the microprocessor.
3. The microprocessor services the event (ISR), which makes a higher priority task ready for execution.
4. The ISR invokes the RTOS, which decides to run the more important task.
5. The higher priority task executes to completion unless it also gets interrupted by another more important task.
6. At the end of the higher priority task, the RTOS resumes the lower priority task.
7. The lower priority task continues to execute.

2.3. Requirements of Real-Time Operating Systems?

The following are basic requirements of a RTOS [4]:

- RTOS must be multi-threaded.
- The scheduler should be able to preempt any thread in the system.
- RTOS should handle multiple levels of interrupts.

- RTOS should be preemptible at interrupt level as well as thread level.
- RTOS should be able to determine which thread needs a resource the most. Deadline information is converted to priority levels and the OS allocates resources according to the priority levels of threads.
- For multiple threads to communicate among each other, in a timely fashion, predictable inter-thread communication and synchronization mechanisms are required.
- RTOS should be the ability to lock/unlock resources to achieve data integrity.
- RTOS can prevent priority inversion called priority inheritance by giving the lower priority task the same priority as the higher priority task that is being blocked.
- When using priority scheduling, it is important that the RTOS has a sufficient number of priority levels, so that applications with stringent priority requirements can be implemented. Unbounded priority inversion occurs when a higher priority task must wait on a low priority task to release a resource while the low priority task is waiting for a medium priority task.
- An OS that supports a real-time application needs to have information about the timing of its system calls.

2.4. What is Embedded Linux?

Linux consists of three major parts: a kernel, a system, and a distribution. No such thing exists to make embedded Linux kernel from vanilla Linux kernel. This doesn't mean the kernel can't be embedded. It only means you do not need a special kernel to create an embedded system. One of the official kernel releases can be used to build your system. A Linux kernel used in an embedded system differs from a kernel used on a workstation or a server by its build configuration. An embedded Linux system is based on the Linux kernel and does not imply the use of any specific library or user tools with this kernel. Development host system may include various development tools that special source browser, cross compilers, debuggers, project management software, boot image builders, and so on [8].

Building Embedded Linux system includes these steps:

- Hardware Development
- Boot Loader development

- Kernel Porting
- Device driver development
- Application program development
- System integration and packaging

2.5. What is Real-Time Linux?

Real-time Linux represents the expanding industrial and scientific use of Linux operating systems for real-time applications and real-time systems [5].

Most general-purpose operating systems are optimized for average case performance. Fairness is key objective of these operating systems. Each process of these operating systems is given fair amount of CPU time. However, this approach is not optimal for real-time systems. In real-time systems, precise timing and predictable performance is more vital than average case performance. Linux was originally designed as a general-purpose desktop operating system. In last several years, there have been many optimizations the Linux kernel to use it as a real-time embedded operating system. There are mainly two approaches to achieve these objectives for Linux operating systems. One of them uses small real-time kernel that runs the Linux kernel on it as lowest priority task. Second approach is reworking on Linux kernel to add more preemptions points, to increase timer resolution and to reduce overall latency on system [5].

2.6. Overview of Available Real-Time OS

2.6.1. VxWorks

Wind River's VxWorks is widely used in telecommunications, consumer electronics, data networking, medical system, robotics, process control, avionics, CNC equipment and light simulation control applications.

The thread based kernel core of VxWorks provides multitasking, interrupt support and both preemptive and round robin scheduling. The intertask communications mechanisms supported by VxWorks includes shared memory, message queues, semaphores, events and pipes, sockets and remote procedure calls and signals. VxWorks is POSIX compliant, implements the POSIX pThreads and the POSIX Real Time Extension. The Tornado embedded development platform provides a full suite of cross development tools. The Tornado auto scaling feature analyses the application code and incorporates optional subsystems [9, 10].

2.6.2. Windows CE

Microsoft's Windows CE .NET is designed for mobile applications and applications requiring a small footprint that can be configured to be 200 Kb with limited kernel functions. It supports wireless technologies such as Bluetooth, 802.1x, IPv6, Object Exchange Protocol (OBEX), Media Sense, and Real Time Communication (RTC)/Session Initiation Protocol (SIP). Windows CE .NET supports the Kerberos Security Protocol and the Secure Sockets Layer (SSL). It provides 256 levels of thread priority, nested interrupts, and priority inversion.

Applications developers on Windows CE .NET can use about 10% of the Win32API's available on the Windows 2000 O/S. Windows CE .NET is not POSIX compliant. Windows CE .NET applications may be created with Microsoft Visual Studio .NET and native code created with Microsoft embedded Visual C++ 4.0. Windows CE .NET developers can build and test the design on their Windows 2000 or Windows XP Professional workstations [9, 10].

2.6.3. QNX

QNX Neutrino RTOS is designed for embedded mission and life-critical systems, medical instruments, aviation and space systems, process-control systems, and in-car devices by QNX Software Systems Ltd.

QNX complies with the IEEE 1003.1b real-time standard. The process based kernel core of QNX provides services such as thread scheduling, inter-process communication and uses synchronous message passing to communicate with other OS modules. All OS modules run in their own memory protected address space.

The intertask communications mechanism supported by QNX Neutrino includes conventional synchronization services like semaphores, conditional variables, and spin locks. The QNX Neutrino RTOS architecture provides several modular components. The QNX Photon microGUI module is integrated with Internet capabilities, international fonts, and sophisticated user interfaces. QNX Neutrino distributed processing Qnet module provides networking support such as Tiny TCP/IP stack, and NetBSD 1.5 IPv4 or IPv6 stack [9 , 10].

2.6.4. RTOS comparison with execution models

There are available three types of kernel according to execution model that are thread based, processes based and microkernel based kernels.

Thread based kernels: There is typically no protection inherent in the operating system. It is possible that errant pointers corrupting data or code between separate threads. The thread paradigm usually supports a flat memory model where all memory is accessible by all threads at all times. This makes it extremely easy to share memory between threads. Simply pass a pointer and, you now have access the memory. Threads also tend to have very small context data structures. This makes for very fast context switching. Some of the strengths of the thread-based model are also its weaknesses. Namely, because of the flat memory model, a thread’s pointer can point into the operating system itself [11].

Process based kernels: Each process has its own memory address space and that memory address space is isolated from other processes and from the kernel. Any attempt to stray outside of the process’s address space will abort the process. In order to enforce this process separation, the hardware MMU is called into action. Each process has an additional piece of context that corresponds to the list of valid addresses and their mapping to physical pages of memory via the MMU. Each access to memory is validated against this memory map. Because of the additional context added by this memory map, the process-to-process context switch time is considerably longer than thread-to-thread context switches [11].

Microkernel based kernel: Only kernel is in privileged mode and everything else is a process in non-priviledge mode. This results in very secure system software. But, context switch and message passing are slowest.

RTOS comparison with execution models is shown in Table 2.1

Table 2.1: RTOS comparison with execution models

OS	Execution model
Linux	Process + Thread
QNX	Microkernel
VxWorks	Thread
Win CE	Process + Thread

2.6.5. RTOS comparison with processor modes

If processor is in privileged mode, it can execute all privileged instructions and can access all memory. If processor is in non-privileged mode, it is restricted to safe instructions and can only access memory that is allowed by MMU.

RTOS comparison with processor mode is shown in Table 2.2

Table 2.2: RTOS comparison with processor mode

OS	Processor Mode
Linux	Privileged + non-privileged
QNX	Privileged + non-privileged
VxWorks	Privileged
Win CE	Privileged + non-privileged

2.6.6. RTOS comparison with interrupt modes

RTOS comparison with interrupt modes is shown in Table 2.3

Table 2.3: RTOS comparison with interrupt modes

OS	Interrupt Mode
Linux	Nested interrupts + Multiple interrupts per vector
QNX	Nested interrupts + Multiple interrupts per vector + Partial masking at all levels
VxWorks	Nested interrupts + Multiple interrupts per vector + Partial masking at all levels
Win CE	-

2.6.7. RTOS comparison with scheduling policies

RTOS comparison with scheduling policies is shown in Table 2.4

Table 2.4: RTOS comparison with scheduling policies

OS	Scheduling Polices
Linux	O(1) , non real-time
QNX	Priority
VxWorks	Priority + Round Robin + Priority Inversion Prevention
Win CE	Priority + Priority Inversion Prevention

2.6.8. RTOS comparison with licensing

RTOS comparison with licensing is shown in Table 2.5

Table 2.5: RTOS comparison with licensing

OS	Licensing Models
Linux	GNU Public License
QNX	License + royalty
VxWorks	License + royalty
Win CE	License + royalty

2.6.9. RTOS comparison with performance

Average and maximum latencies on a 200 MHz Pentium machine are shown in Table 2.6. Latencies for Linux depend on which interrupt handling mechanism is used. So, Linux average and maximum latency are not given in the Table 2.6.

Table 2.6: RTOS comparison with performance [11]

OS	Average Latency	Maximum Latency
QNX	1,6 us	4,1 us
VxWorks	1,7 us	6,8 us
Win CE	2,4 us	5,6 us
Linux	-	-

2.7. Reasons For Choosing Linux For Embedded Applications

Embedded applications are getting more complexity, although pressure for a shorter time-to-market is accelerating. It is clear that standalone and roll-your-own kernels are inadequate to support all the expected complexity. So, reduced development costs needs using platforms provide rich software and device driver support. Using general-purpose software platform such as Linux is the best way to implement a complex embedded system. In addition, Linux is open source, this results better control and ability to adapt the entire system to specific needs. Linux is also efficient, robust, modular, and configurable all important characteristics for mission-critical embedded systems. Moreover, Linux is royalty free.

There are various motivations for choosing Linux over a traditional embedded OS [12]:

- Modularity: Linux can be extended at runtime that means you can add functionality to the kernel while the system is up and running. Each piece of code that can be added to the kernel at runtime is called a module. Modules can be taken as completely independent programs, which can be loaded into the system at run time and removed from it again later. Hardware drivers are generally kernel modules. If a driver is to be used for a self-developed card, the driver can be used for another card without Linux kernel has to be re-converted. This means it is easy to change driver and easy to port existing application to another system.
- Ease of fixing: The code should be easy to fix for whoever understands its internals. It is possible, because codes are open.
- Extensibility: Adding features to the code should be fairly straightforward.

- Configurability: It is possible to select which features from the code should be part of the final application. Linux and GCC provide the required flexibility.
- Predictability: Upon execution, the program's behavior is supposed to be within a defined framework and should not become erratic.
- Longevity: The program will run unassisted for long periods of time and will conserve its integrity regardless of the situations it encounters.
- Open source development model: Many people all over world are contributing to projects, identifying existing problems, debating possible solutions, and fixing problems effectively.
- Mailing lists: Mailing list is a good option to solve problems. It is possible to get a solution from someone that he or she solved the same problem before.
- Availability of code: Kernel is distributed under the (GPL). Kernel source code can be downloaded freely. Availability of code results possibility of fixing the code without exterior help, the capability of digging into the code to understand its operation and fixes for security weaknesses.
- Hardware support: Linux supports different types of hardware platforms and devices. Large number of drivers are maintained by the Linux community itself confidently use hardware components without fear of obsolete. No other OS provides this level of portability. CPU and a platform, you can reasonably expect that Linux runs on it or that someone else has gone through a similar porting process and can assist you in your efforts. The software that you write for special architecture can be easily ported to another architecture Linux runs on. Device drivers that are written as module can be transparently used on different architectures.
- Communication protocol and software standards: Linux also provides broad communication protocol and software standards support. Linux origins lie in the network communication of the classic Unix computer. Drivers and all-important protocols for Ethernet are already included. All standard protocols such as HTTP, ftp, Telnet etc. can be used. The integration of other field buses has now become a reality and there are now drivers available for almost all the major field buses such as Profibus, CAN, RS485. An embedded Linux can be interface between classic field bus technology and modern industrial Ethernet [13].
- Available tools: The variety of tools existing for Linux make it very versatile.

- Vendor independence: You do not need to rely on any sole vendor to get Linux or to use it [13].
- Cost: Three components of software cost are initial development setup, additional tools, and runtime royalties. With Linux, all development tools and OS components are available free of charge, and no royalties [8].

3. LINUX KERNEL MECHANISMS THAT INFLUENCE ON REAL-TIME PERFORMANCE

3.1. Interrupt Handling Mechanisms

Interrupts allow hardware communicate with processor. The use of interrupts is the most effective way to deal with external events. Instead of polling, the CPU can work on other tasks. Therefore, most systems use interrupt-driven input/output. A number of steps have to be performed by hardware and software to handle an interrupt.

3.1.1. Interrupt handling by the interrupt controller

In x86 based PC-hardware, main processor does not receive interrupt signals directly from the devices. An interrupt controller is located between the CPU and the I/O devices in the system. An interrupt is electronic signals originating from devices and directed into input pins on an interrupt controller. Different devices are associated with unique interrupts by means of a unique value associated with each interrupt. These interrupt values are often called interrupt request (IRQ) lines. All existing IRQ lines are connected to the input pins of a hardware circuit called the Programmable Interrupt Controller. A system using 8259A PIC is shown in Figure 3.1. Interrupt controller performs the following actions [14]:

- Monitors the IRQ lines
- Checks for raised signals
- Accept interrupt request signals from the I/O devices in the system.
- If two or more IRQ lines are raised, selects the one having the lower pin number. Thereby, resolve priorities among these requests.
- Forward the highest priority request to the CPU.
- Converts the raised signal received into a corresponding vector to tell the CPU on which input line the interrupt occurred. Since PCI devices are capable of sharing interrupts, it is left to CPU to find out which device exactly needs attention.

- Stores the vector in an Interrupt Controller I/O port, thus allowing the CPU to read it via the data bus.
- Sends a raised signal to the processor INTR pin that is, issues an interrupt
- Waits until the CPU acknowledges the interrupt signal by writing into one of the Programmable Interrupt Controllers I/O ports
- Clears the INTR line after the CPU acknowledges the interrupt signal.

For the CPU to know which device or devices need to be serviced, the interrupt controller maps each of its interrupt input lines to an interrupt request number. The CPU gathers this number from the controller and executes the particular handler .

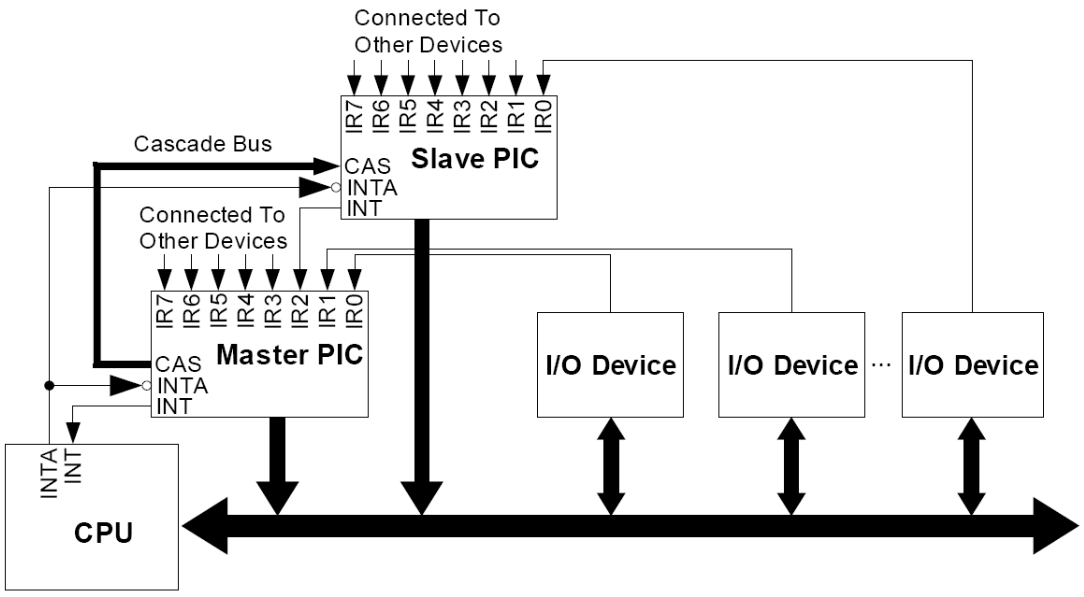


Figure 3.1 : A system using the 8259A PIC [14]

3.1.2. Interrupt handling by the CPU

There are two mechanisms that the processor may be interrupted in its instruction flow. The mechanisms are interrupts and exceptions. Interrupt is usually used for asynchronous events from hardware devices and delivered to the processor by an interrupt controller. Exceptions are events generated within the instruction flow and are thus synchronous such as division by zero or page fault. Linux service calls are implemented with using these exceptions.

Each type of event are handled same way. Each individual event is mapped to a vector ranging from 0 to 255. Acceptance of interrupt can be enabled or disabled by setting corresponding bit in CPU registers.

Intel processors interrupt handling sequence [14, 15, 16]:

1. Interrupt is accepted only at instruction boundary to save a valid instruction pointer.
2. The interrupt vector is determined from the data bus in PIC systems or from the local APIC in APIC systems.
3. This vector is used as an index in the Interrupt Descriptor Table (IDT). It contains up to 256 entries, one for each possible vector.
4. If the current privilege level of the processor is lower than the privilege level of the service routine to execute, a stack switch occurs.
5. The EFLAGS, CS, and EIP registers are pushed on the stack.
6. The segment and instruction pointer to the interrupt service routine as found in the IDT are loaded.
7. For interrupt gates, the processor disables interrupts locally.
8. The processor is executing the handler code.
9. After the handler has finished, the iret instruction instructs the CPU to restore the saved context and resume the interrupted task.

3.1.3. Interrupt handling by the Linux

3.1.3.1. Interrupt handler:

The function the kernel runs in response to a specific interrupt called an interrupt handler or interrupt service routine (ISR). Interrupt handler for a device is part of the device driver. Handler have to run quickly to resume execution of the interrupted code as soon as possible When an I/O device raises an interrupt, the first instructions of the corresponding kernel control path save the contents of the CPU registers in the Kernel Mode stack, and the last instructions restore the contents of the registers. Regardless of the kind of circuit that caused the interrupt, all I/O interrupt handlers perform the same four basic actions:

1. Save the IRQ value and the register's contents on the Kernel Mode stack
2. Send an acknowledgment to the PIC that is servicing the IRQ line, thus allowing it to issue further interrupts.
3. Execute the interrupt service routines associated with all the devices that share the IRQ.

4. Terminate by jumping to the ret_from_intr( ) kernel function address.

3.1.3.2. Nesting interrupt handler:

Kernel control paths may be arbitrarily nested. An interrupt handler may interrupt an interrupt another handler. The price to pay for allowing nested kernel control paths is that an interrupt handler must never block and no process switch can take place until an interrupt handler is running [17]

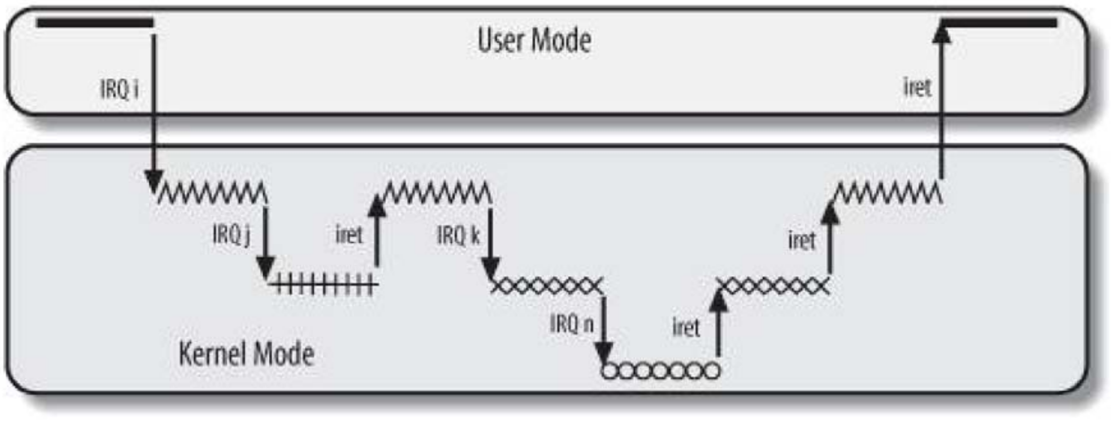


Figure 3.2 : Nesting interrupt handlers [17]

3.1.3.3. Sharing IRQ lines:

An I/O interrupt handler must be flexible enough to service several devices at the same time. Several devices may share the same IRQ line. This means that the interrupt vector alone does not tell the whole story. An I/O interrupt handler must be flexible enough to service several devices at the same time. Several devices may share the same IRQ line. This means that the interrupt vector alone does not tell the whole story [18]. The kernel must discover which I/O device corresponds to the IRQ number before enabling interrupts. Otherwise, for example, how could the kernel handle a signal from a SCSI disk without knowing which vector correspond to the device? The correspondence is established while initializing each device driver. There are two ways to achieve interrupt handler flexibility.

IRQ sharing: The interrupt handler executes several interrupt service routines. Each ISR is a function related to a single device sharing the IRQ line [17].

IRQ dynamic allocation: An IRQ line is associated with a device driver at the last possible moment; for instance, the IRQ line of the floppy device is allocated only when a user accesses the floppy disk device. In this way, several hardware devices may use the same IRQ vector even if they cannot share the IRQ line. This way works only when the hardware devices cannot be used at the same time [17].

3.1.3.4. Deferring interrupt handler operations:

Interrupts can come anytime, when the kernel may want to finish something else it was trying to do. The kernel's goal is therefore to get the interrupt out of the way as soon as possible and defer as much processing as it can. Not all actions to be performed when an interrupt occurs have the same urgency. In fact, the interrupt handler itself is not a suitable place for all kind of actions. Long non-critical operations should be deferred, because while an interrupt handler is running, the signals on the corresponding IRQ line are temporarily ignored. Most important, the process on behalf of which an interrupt handler is executed must always stay in the TASK_RUNNING state, or a system freeze can occur. Therefore, interrupt handlers cannot perform any blocking procedure such as an I/O disk operation. The activities that the kernel needs to perform in response to an interrupt are thus divided into critical, noncritical and noncritical deferrable parts [17].

Critical: Actions to be performed following an interrupt are called critical. These can be executed quickly and are critical, because they must be performed as soon as possible. Critical actions are executed within the interrupt handler immediately, with maskable interrupts disabled. Some of critical actions are executed within the interrupt handler:

1. Acknowledging an interrupt to the PIC.
2. Reprogramming the PIC or the device controller.
3. Updating data structures accessed by both the device and the processor.

Non-critical: These actions can also finish quickly, so the interrupt handler executes them immediately, with the interrupts enabled. Actions such as updating data structures that are accessed only by the processor are examples of this type [17].

Non-critical deferrable: These actions may be delayed for a long time interval without affecting the kernel operations such as copying a buffer's contents into the address space of a process. These may be delayed for a long time interval without affecting the kernel operations The interested process will just keep waiting for the data. Non-critical deferrable actions are performed by means of separate functions that are Softirqs, Tasklets, and Workqueues [17].

3.1.4. Journey of interrupt in the kernel

The journey of interrupt in the kernel are consist of both architecture dependent and independent functions. These functions are shown in Figure 3.3

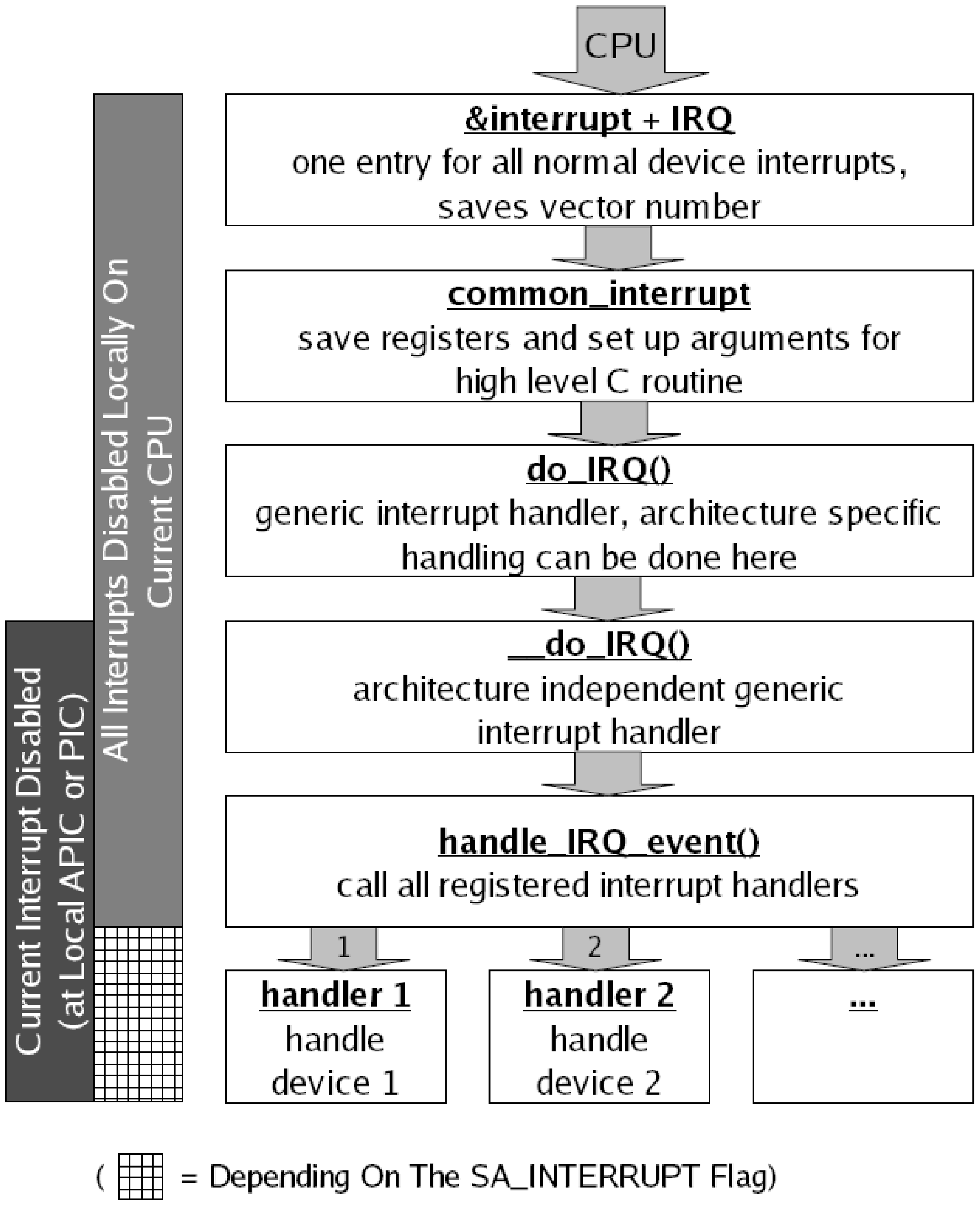

CPU
&interrupt + IRQ
one entry for all normal device interrupts, saves vector number
common_interrupt
save registers and set up arguments for high level C routine
do_IRQ()
generic interrupt handler, architecture specific handling can be done here
__do_IRQ()
architecture independent generic interrupt handler
handle_IRQ_event()
call all registered interrupt handlers
1
2
...
handler 1
handle device 1
handler 2
handle device 2
...
All Interrupts Disabled Locally On Current CPU
Current Interrupt Disabled (at Local APIC or PIC)
(▦ = Depending On The SA_INTERRUPT Flag)

Figure 3.3 : Journey of interrupt in Linux kernel [14]

3.1.4.1. The initial entry point:

The journey of interrupt in the kernel begins at a predefined entry point. For each interrupt line, the processor jumps to a unique location. For doing this, kernel gets

IRQ number of coming interrupt from interrupt controller. The initial entry point save this value and stores the current register values which belongs to the interrupted task and then kernel calls do_IRQ().

3.1.4.2. The do_IRQ() kernel function:

The do_IRQ() is written in C programming language but still architecture dependent and lives in \arch\i386\kernel\irq.c. The function flow diagram is shown in Figure 3.4

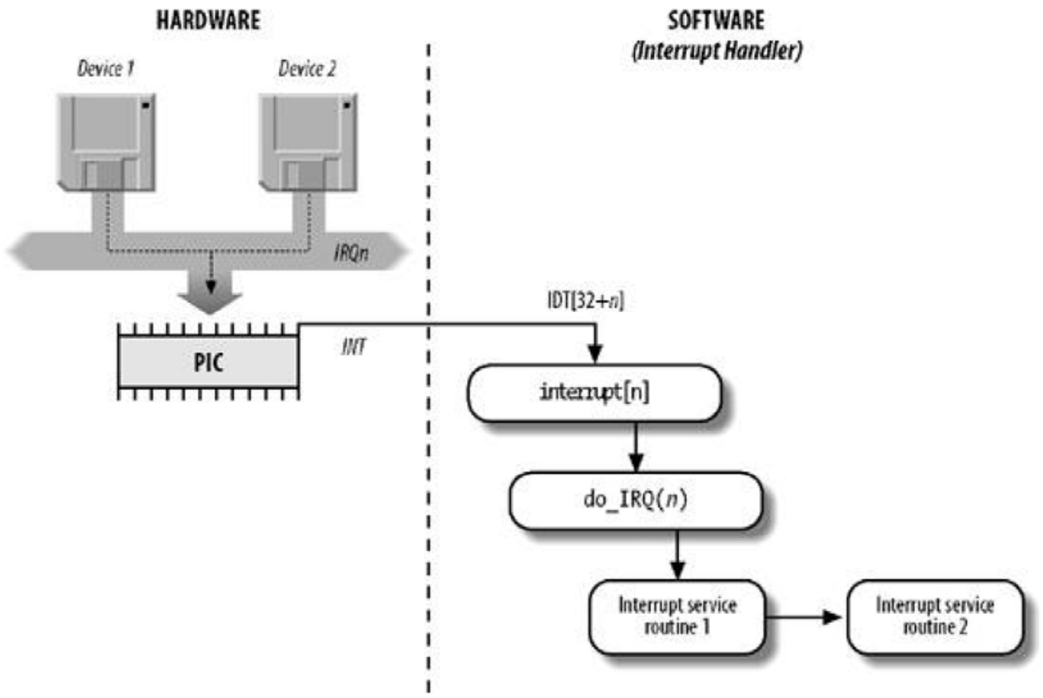


Figure 3.4 : Interrupt handling by do_IRQ() [17]

Code fragment of the function is following:

```
fastcall unsigned int do_IRQ(struct pt_regs *regs)
{
        int irq = regs->orig_eax & 0xff;
        irq_enter();
        __do_IRQ(irq, regs);
        irq_exit();
}
```

The do_IRQ( ) function executes the following actions:

1. Executes the irq_enter( ) macro, which increases a counter representing the number of nested interrupt handlers. The counter is stored in the preempt_count field of the tHRead_info structure of the current process.
2. The IRQ number obtained from the regs->orig_eax field.

3. Invokes the __do_IRQ( ) function passing to it the pointer regs .
4. Executes the irq_exit( ) macro, which decreases the interrupt counter and checks whether deferrable kernel functions are waiting to be executed.
5. Terminates the interrupt service routine by the ret_from_intr( ) function.

3.1.4.3. The __do_IRQ() kernel function:

The function is implemented in \kernel\irq\handle.c. There are two important data structures used in the function that are 'struct irq_desc' and 'struct irqaction'. Fields of 'struct irq_desc' are shown in Table 3.1 and fields of 'struct irqaction' are shown in Table 3.2

Table 3.1 : Fields of 'struct irq_desc' [17]

Field	Descriptions
handler	Points to the hw_irq_controller descriptor that services the IRQ line.
handler_data	Pointer to data used by the PIC methods.
action	Identifies the interrupt service routines to be invoked when the IRQ occurs. The field points to the first element of the list of irqaction descriptors associated with the IRQ.
status	A set of flags describing the IRQ line status
depth	Shows 0 if the IRQ line is enabled and a positive value if it has been disabled at least once.
irq_count	Counter of interrupt occurrences on the IRQ line (for diagnostic use only).
irqs_unhandled	Counter of unhandled interrupt occurrences on the IRQ line (for diagnostic use only).
lock	A spin lock used to serialize the accesses to the IRQ descriptor and to the PIC

Table 3.2: Fields of 'struct irqaction' [17]

Field	Descriptions
handler	Points to the interrupt service routine for an I/O device. This is the key field that allows many devices to share the same IRQ.
flags	This field includes a few fields that describe the relationships between the IRQ line and the I/O device
mask	Not used.
name	The name of the I/O device
dev_id	A private field for the I/O device. Typically, it identifies the I/O device itself
next	Points to the next element of a list of irqaction descriptors. The elements in the list refer to hardware devices that share the same IRQ.
irq	IRQ line.
dir	Points to the /proc/irq/n directory associated with the IRQn.

Using of ‘struct irq_desc’ and ‘struct irqaction’ are shown in Figure 3.5 .

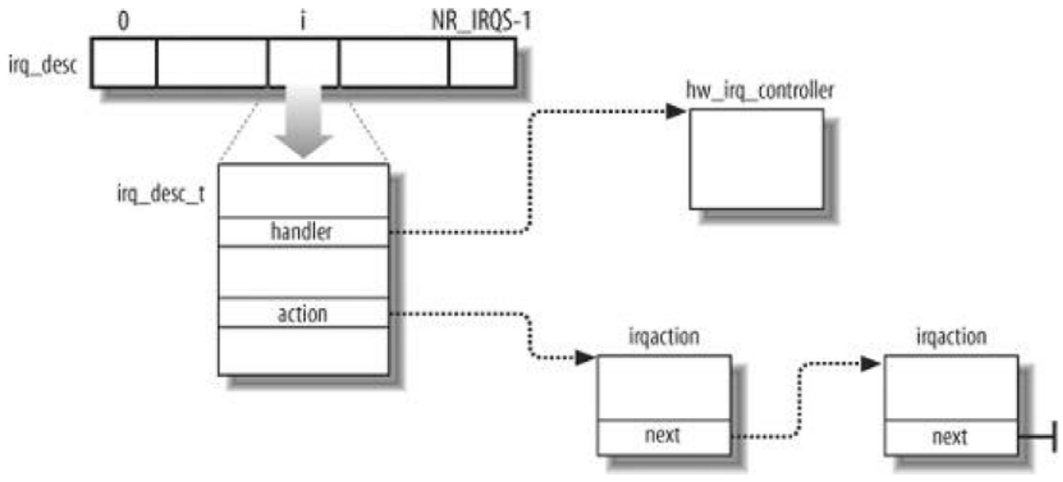


Figure 3.5 : IRQ descriptor [17]

Code fragment of the function is following:

```
fastcall unsigned int __do_IRQ(unsigned int irq, struct pt_regs *regs)
{
        irq_desc_t *desc = irq_desc + irq;
        struct irqaction * action;
        spin_lock(&desc->lock);
        desc->handler->ack(irq);
        for (;;) {
                irqreturn_t action_ret;
                spin_unlock(&desc->lock);
                action_ret = handle_IRQ_event(irq, regs, action);
                spin_lock(&desc->lock);
                if (!noirqdebug)
                        note_interrupt(irq, desc, action_ret, regs);
                if (likely(!(desc->status & IRQ_PENDING)))
                        break;
                desc->status &= ~IRQ_PENDING;
        }
        desc->status &= ~IRQ_INPROGRESS;
        desc->handler->end(irq);
        spin_unlock(&desc->lock);
}
```

The __do_IRQ( ) function executes the following actions:

1. The __do_IRQ( ) function receives as its parameters an IRQ number
2. Invokes the ack method of the main IRQ descriptor. When using the old 8259A PIC, the corresponding mask_and_ack_8259A( ) function acknowledges the interrupt on the PIC and also disables the IRQ line. Masking the IRQ line ensures that the CPU does not accept further occurrences of this type of interrupt until the handler terminates.
3. It sets the IRQ_PENDING flag because the interrupt has been acknowledged, but not yet really serviced it also clears the IRQ_WAITING and IRQ_REPLAY flags.
4. Now __do_IRQ( ) checks whether it must really handle the interrupt. There are three cases in which nothing has to be done. These cases are IRQ_DISABLED is set, IRQ_INPROGRESS is set, and irq_desc[irq].action is NULL.
5. Let's suppose that none of the three cases holds, so the interrupt has to be serviced. The _ _do_IRQ( ) function sets the IRQ_INPROGRESS flag.
6. Starts a loop. The function clears the IRQ_PENDING flag,
7. Releases the interrupt spin lock
8. Executes the interrupt service routines by invoking handle_IRQ_event( ).
9. Acquires the spin lock again and checks the value of the IRQ_PENDING flag. If it is clear, no further occurrence of the interrupt has been delivered to another CPU, so the loop ends.
10. If IRQ_PENDING is set, do_IRQ( ) performs another iteration of the loop, servicing the new occurrence of the interrupt
11. The function invokes the end method of the main IRQ descriptor. When using the old 8259A PIC, the corresponding end_8259A_irq( ) function reenables the IRQ line.
12. Finally, __do_IRQ( ) releases the spin lock

3.1.4.4. The handle_IRQ_event kernel function:

An interrupt service routine handles an interrupt by executing an operation specific to one type of device. When an interrupt handler must execute the ISR, it invokes the handle_IRQ_event( ) function. The function is implemented in

\kernel\irq\handle.c

Code fragment of the function is following:

```
fastcall int handle_IRQ_event(int irq, struct pt_regs *regs, struct irqaction *action)
{
        int ret, retval = 0, status = 0;
        if (!(action->flags & SA_INTERRUPT))
                local_irq_enable();
        do {
                ret = action->handler(irq, action->dev_id, regs);
                if (ret == IRQ_HANDLED)
                        status |= action->flags;
                retval |= ret;
                action = action->next;
        } while (action);
        if (status & SA_SAMPLE_RANDOM)
                add_interrupt_randomness(irq);
        local_irq_disable();
        return retval;
}
```

This function essentially performs the following steps:

1. Enables the local interrupts if the SA_INTERRUPT flag is clear.
2. Executes each interrupt service routine of the interrupt.
3. At the start of the loop, action points to the start of a list of irqaction data structures that indicate the actions to be taken upon receiving the interrupt
4. Disables local interrupts.

3.1.5. Registering an interrupt handler

Drivers can register an interrupt handler and enable a given interrupt line for handling via the function request_irq() that locate in \kernel\irq\manage.c

```
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long irqflags,
                const char * devname,
                void *dev_id   )
```

The first parameter, irq, specifies the interrupt number to allocate. For some devices, this value is typically hard coded such as system timer or keyboard. For most other devices, it is probed or otherwise determined dynamically.

The second parameter, handler, is a pointer to the actual interrupt handler that service this interrupt. This function is invoked whenever the interrupt is received by the operating system.

The third parameter, irqflags, may be either zero or a bit mask of one or more of the fallowing flags.

SA_INTERRUPT: run with all interrupts disabled on local processor.

SA_SAMPLE_RANDOM: the timing of interrupts from this device will be fed to the poll of entropy.

SA_SHIRQ: This flag specifies that the interrupt line can be shared multiple interrupt handlers.

The fourth parameter, devname, is a text representation of the device.

The fifth parameter, dev_id, is used primarily for shared interrupt lines. When an interrupt handler is freed, dev_id provides a unique cookie to allow removal of only the desired interrupt handler from the interrupt line [17].

3.1.6. Deferring work and bottom halves

Several tasks that are executed by the kernel are not critical. These non-critical tasks can be deferred to execute for a long period of time. Remember that the interrupt service routines of an interrupt handler are serialized, and often there should be no occurrence of an interrupt until the corresponding interrupt handler has terminated. Conversely, the deferrable tasks can execute with all interrupts enabled. Taking them out of the interrupt handler helps keep kernel response time small. This is a very important property for many time-critical applications that expect their interrupt requests to be serviced in a few milliseconds. Linux 2.6 answers such a challenge by using three kinds of non-urgent interruptible kernel functions: softirqs, tasklets and work queues.

3.1.7. Softirqs

Softirqs are statically allocated. Softirqs can run concurrently on several CPUs, even if they are of the same type. Thus, softirqs are reentrant functions and must explicitly protect their data structures with spin locks.

Linux 2.6 uses a limited number of softirqs. Only the six kinds of softirqs are currently defined as shown in Table 3.3 .

Table 3.3: Available softirqs in Linux 2.6 kernel [12]

Softirq	**Index (priority)**	**Description**
HI_SOFTIRQ	0	Handles high priority tasklets
TIMER_SOFTIRQ	1	Tasklets related to timer interrupts
NET_TX_SOFTIRQ	2	Transmits packets to network cards
NET_RX_SOFTIRQ	3	Receives packets from network cards
SCSI_SOFTIRQ	4	Post-interrupt processing of SCSI commands
TASKLET_SOFTIRQ	5	Handles regular tasklets

3.1.7.1. Implementations of softirqs:

Softirq codes are located in kernel\softirq.c. Basically, registered softirq must be marked before it will execute. This is called raising the softirq. Usually, an interrupt handler marks its softirq for execution before returning. The kernel should be periodically checks for pending softirqs. If there are any pending softirq, means that it is suitable to execute softirqs, the kernel executes softirqs. The softirq pending check is performed in a few points of the kernel code. Number and position of the softirq checkpoints may change both with the kernel version and with the supported hardware architecture. Most significant points are followings:

- The kernel invokes the local_bh_enable( ) function to enable softirqs on the local CPU.
- When the do_IRQ( ) function finishes handling an I/O interrupt and invokes the irq_exit( ) macro.
- By code that explicitly checks and executes pending softirqs.
- When one of the special ksoftirqd/n kernel threads is awakened.

If pending softirqs are detected at one such checkpoint (local_softirq_pending() is not zero), the kernel invokes do_softirq( ) to take care of them.

```
asmlinkage void do_softirq(void)
{
        __u32 pending;
        unsigned long flags;
        if (in_interrupt())
                return;
        local_irq_save(flags);
        pending = local_softirq_pending();
        if (pending)
                __do_softirq();
        local_irq_restore(flags);
}
```

do_softirq() function performs the following actions:

1. If in_interrupt() yields the value one, this function returns. This situation indicates either that do_softirq() has been invoked in interrupt context or that the softirqs are currently disabled.
2. Executes local_irq_save to save the state of the IF flag and to disable the interrupts on the local CPU.
3. Invokes the __do_softirq() function
4. Executes local_irq_restore to restore the state of the IF flag saved in step 2 and returns.

__do_softirq() function reads the softirq pending bit mask of the local CPU and executes the deferrable functions corresponding to every set bit. New softirqs pending may come before ending the loop that executes pending. So, this can cause latency especially in heavy network loads because of thousand of RX, TX events per second. __do_softirq() function limit the number of execution and use kernel thread to prevent this type of latency.

Main data structure used in softirqs is 'struct softirq_action'. Fields of the data structure is following:

```
struct  softirq_action
{
        void    (*action)(struct softirq_action *);
        void    *data;
};
static struct softirq_action softirq_vec[32];
```

Code fragment of __do_softirq() is following:

```
asmlinkage void __do_softirq(void)
{
        pending = local_softirq_pending();
restart:
        local_softirq_pending() = 0;
        local_irq_enable();
        h = softirq_vec;
        do {
                if (pending & 1) {
                        h -> action(h);
                }
                h++;
                pending >>= 1;
        } while (pending);
        local_irq_disable();
        pending = local_softirq_pending();
        if (pending && --max_restart)
                goto restart;
        if (pending)
                wakeup_softirqd();
}
```

__do_softirq() function performs the following actions:

1. Initializes the iteration counter, max_restart, to a predefined value.
2. Copies the softirq bit mask of the local CPU by local_softirq_pending( ) in the pending local variable.
3. Invokes local_bh_disable( ) to increase the softirq counter. Because the deferrable functions mostly run with interrupts enabled, an interrupt can be raised in the middle of the __do_softirq( ) function. When do_IRQ( ) executes the irq_exit( ) macro, another instance of the __do_softirq( ) function could be started. This has to be avoided, because deferrable functions must execute serially on the CPU. Thus, the first instance of __do_softirq( ) disables deferrable functions, so that every new instance of the function will exit at step 1 of do_softirq( ) .
4. Clears the softirq bitmap of the local CPU, so that new softirqs can be activated.
5. Executes local_irq_enable( ) to enable local interrupts.
6. For each bit set in the pending local variable, it executes the corresponding softirq function; recall that the function address for the softirq with index n is stored in softirq_vec[n]->action.

7. Executes local_irq_disable() to disable local interrupts.
8. Copies the softirq bit mask of the local CPU into the pending local variable and decreases the iteration counter one more time.
9. If pending is not zeroat least one softirq has been activated since the start of the last iteration and the iteration counter is still positive, it jumps back to step 4.
10. If there are more pending softirqs, it invokes wakeup_softirqd( ) to wake up the kernel thread that takes care of the softirqs for the local CPU.

3.1.7.2. The ksoftirqd kernel threads:

Softirq functions may reactivate themselves. Particularly, the networking softirqs and the tasklets do this. Moreover, external events, such as packet flooding on a network card, may activate softirqs at very high frequency. The potential for a continuous high-volume flow of softirqs creates a problem that is solved by introducing kernel threads. Without them, developers are essentially faced with two alternative strategies. The first strategy consists of ignoring new softirqs that occur while do_softirq( ) is running. This solution is not good enough. Suppose that a softirq function is reactivated during the execution of do_softirq( ). In the worst case, the softirq is not executed again until the next timer interrupt, even if the machine is idle. As a result, softirq latency time is unacceptable for networking developers. The second strategy consists of continuously rechecking for pending softirqs. The __do_softirq( ) function could keep checking the pending softirqs and would terminate only when none of them is pending. While this solution might satisfy networking developers, it can certainly upset normal users of the system: if a high-frequency flow of packets is received by a network card or a softirq function keeps activating itself, the do_softirq( ) function never returns, and the User Mode programs are virtually stopped [18].

The ksoftirqd/n kernel threads represent a solution the critical trade-off problem. The do_softirq( ) function determines what softirqs are pending and executes their functions. After a few iterations, if the flow of softirqs does not stop, the function wakes up the kernel thread and terminates. When awakened, the kernel thread checks the local_softirq_pending() softirq bit mask and invokes, if necessary, do_softirq( ). If there are no softirqs pending, the function puts the current process in the TASK_INTERRUPTIBLE state and invokes then the cond_resched() function to perform a process switch if required by the current process. The kernel thread has low priority, so user programs have a chance to run; but if the machine is idle, the pending softirqs are executed quickly [18]. The equivalent function of ksoftirqd/n kernel threads is following code fragment:

```
while (!kthread_should_stop()) {
            preempt_disable();
            if (!local_softirq_pending()) {
                  schedule();
            }
            __set_current_state(TASK_RUNNING);
            while (local_softirq_pending()) {
                  do_softirq();
                  cond_resched();
                  preempt_disable();
            }
            preempt_enable();
            set_current_state(TASK_INTERRUPTIBLE);
      }
```

3.1.7.3. Using softirqs:

Softirqs using steps are as follows:

1. Create a new softirq,
2. Add a new entry to the enum.
3. Registering your handler by the open_softirq() function. The function takes care of softirq initialization. By the way, the registered handler run with interrupt enabled and cannot sleep. For example: open_softirq(NEW_SOFTIRQ, new_handler, NULL)
4. Raising your softirq by the raise_softirq() function. For example, raise_softirq (NEW_SOFTIRQ).

Code fragment of he raise_softirq() function is following

```
void fastcall raise_softirq(unsigned int nr)
{
      unsigned long flags;
      local_irq_save(flags);
      local_softirq_pending() |= 1UL << (nr);
      if (!in_interrupt())
            wakeup_softirqd();
```

```
        local_irq_restore(flags);
}
```

The function raise_softirq, which receives as its parameter the softirq index, performs the following actions:

1. Executes the local_irq_save macro which saves the state of the IF flag and disable interrupts on the local CPU.
2. Marks the softirq as pending by setting the bit in the softirq bit mask of the local CPU.
3. Invokes wakeup_softirqd() to wake up.
4. Executes the local_irq_restore macro to restore the state of the IF flag saved in step

3.1.8. Tasklets

Tasklets are based on softirq mechanism. Tasklets are good enough and are much easier to implement deferrable functions in I/O drivers than softirqs. Because, Tasklets are not reentrant and number of tasklets are not limited while softirq is. Tasklets can be allocated and initialized at runtime when loading a kernel module. Tasklets do not have to protect their data structures by spinlocks, because their execution is controlled more strictly by the kernel. Tasklets of the same type are always serialized that means the same type of tasklet cannot be executed by two CPUs at the same time. However, tasklets of different types can be executed concurrently on several CPUs. Serializing the tasklet simplifies the life of device driver developers, because the tasklet function needs not be reentrant [17].

3.1.8.1. Implementations of tasklets:

Tasklets are built on top of two softirqs named HI_SOFTIRQ and TASKLET_SOFTIRQ as shown in Table 3.3 .

Several tasklets may be associated with the same softirq, each tasklet carrying its own function. There is no real difference between the two softirqs, except that do_softirq() executes HI_SOFTIRQ's tasklets before TASKLET_SOFTIRQ's tasklets.

Data structure of tasklet is ‘struct tasklet_struct’ that is located in include\linux\interrupt.h. Fields of the structure are shown in Table 3.4 .

Table 3.4: The tasklet_struct data structure [17]

Field Name	Description
next	Pointer to next descriptor in the list
state	Status of the tasklet
count	Lock counter
func	Pointer to the tasklet function
data	An unsigned long integer that may be used by the tasklet function

3.1.8.2. Using tasklets:

Let's suppose you're writing a device driver and you want to use a tasklet: what has to be done? Tasklets using steps are as follows:

Declaring your tasklet: New tasklet_struct data structure should be allocated and initialized by invoking tasklet_init( ). This function receives as its parameters the address of the tasklet descriptor, the address of your tasklet function, and its optional integer argument.

```
void tasklet_init (      struct tasklet_struct *t,   //  address of the tasklet descriptor
                         void (*func)(unsigned long),  //  tasklet function
                         unsigned long data )  // optional integer argument
```

Writing tasklet handler: Tasklet handler cannot sleep and run interrupts enabled.

Scheduling your tasklet: To activate the tasklet, you should invoke either the tasklet_schedule() function or the tasklet_hi_schedule() function, according to the priority that you require for the tasklet. After a tasklet is scheduled, it runs in the near future. For example: tasklet_schedule(&mytasklet)

tasklet_schedule code fragment is following:

```
void fastcall __tasklet_schedule(struct tasklet_struct *t)
{
        unsigned long flags;
        local_irq_save(flags);
        t->next = __get_cpu_var(tasklet_vec).list;
        __get_cpu_var(tasklet_vec).list = t;
        raise_softirq_irqoff(TASKLET_SOFTIRQ);
        local_irq_restore(flags);
}
```

The function performs the following actions:

1. Checks the TASKLET_STATE_SCHED flag; if it is set that means the tasklet has already been scheduled, returns.
2. Invokes local_irq_save to save the state of the IF flag and to disable local interrupts.
3. Adds the tasklet descriptor at the beginning of the list pointed to by tasklet_vec[n] or tasklet_hi_vec[n], where n denotes the logical number of the local CPU.
4. Invokes raise_softirq_irqoff( ) to activate either the TASKLET_SOFTIRQ or the HI_SOFTIRQ softirq this function is similar to raise_softirq( ), except that it assumes that local interrupts are already disabled.
5. Invokes local_irq_restore to restore the state of the IF flag.

Tasklets are executed by the do_softirq( ) function. The softirq function associated with the HI_SOFTIRQ softirq is named tasklet_hi_action( ), while the function associated with TASKLET_SOFTIRQ is named tasklet_action( ) [18].

3.1.9. Work Queues

Deferrable functions such as softirqs and tasklets run in interrupt context while functions in work queues run in process context. Work queues defer work into a kernel thread. Functions that may need block or sleep are not allowed to execute in interrupt context, because no process switch can take place in interrupt context. If deferred work needs to sleep, work queues are only selection.

They allow kernel functions to be activated and later executed by special kernel threads called worker threads. The main difference is that deferrable functions run in interrupt context while functions in work queues run in process context. Work queue functions can block, the worker thread can be put to sleep and even migrated to another CPU when resumed.

Every worker thread continuously executes a loop inside the worker_thread() function. Most of the time, the thread is sleeping and waiting for some work to be queued. Once awakened, the worker thread invokes the run_workqueue( ) function.

3.2. Linux Kernel Scheduler

Scheduling algorithm of an operating system is designed to its market requirements. The market requirements of an operating system should be firstly determined to understanding of an operating system scheduling algorithm. These requirements are following [18]:

Efficiency: It must try to allow as much real work as possible. Efficiency and others goals such as interactivity have inverse relationship. Interactivity essentially requires having more frequent context switches, so this decreases to runtime of real work or efficiency. Also, scheduler own speed is an important factor in scheduling efficiency. The code making scheduling decisions should run as quickly as possible.

Interactivity: Schedulers should be designed to respond to user interaction within a certain time period.

Fairness and Preventing Starvation: Schedulers should be designed to prevent Starvation. Fairness does not mean that every thread should have the same degree of access to CPU time with the same priority, but it means that starvation must not be allowed. Starvation happens when a thread is not allowed to run for an unacceptably long period of time due to the prioritization of other threads over it.

3.2.1. Scheduling algorithm of the Linux kernel

The Linux 2.4.x scheduler has undesirable characteristics that are embedded in its design. So, during the Linux 2.5.x development period, The Linux 2.4.x scheduler was changed to a new one instead of making modifications to it. The Linux 2.4.x scheduling algorithm contains O(n) algorithms was perhaps its greatest lack, and subsequently the new scheduler's use of only O(1) algorithms was its most welcome improvement.

O(n) algorithm operates on input, and the size of that input usually determines its running time. Big-O notation is used to denote the growth rate of an algorithm's execution time based on the amount of input. For example, the running time of an O(n) algorithm increases linearly as the input size n grows. The running time of an O(nˆ2) grows quadratically. If it is possible to establish a constant upper bound on the running time of an algorithm, it is considered to be O(1). An O(1) algorithm is guaranteed to complete in a certain amount of time regardless of the size of the input [19].

The Linux 2.6.x scheduler operates in constant time and does not contain any algorithms that the size of that input increases its running time. So, the Linux 2.6.x scheduler algorithm is an O(1) time algorithm. That is, every part of the scheduler is guaranteed to execute in a certain constant amount of time regardless of how many tasks are on the system. This allows the Linux kernel to efficiently handle massive numbers of tasks without increasing overhead costs as the number of tasks grows [19].

3.2.2. Implementations of the Linux 2.6.x scheduler

There are key data structures in the Linux 2.6.x scheduler that allow for it to perform its duties in O(1) time. The data structures are run queues, priority arrays, process descriptor.

3.2.2.1. The runqueue data structure:

The runqueue data structure is defined as a struct in kernel/sched.c. A runqueue structure keeps track of all runnable tasks assigned to a particular CPU. So, one runqueue is created and maintained for each CPU in a system. The job of the runqueue is to keep track of a CPU's thread information to handle its two priority arrays. The fields of 'runqueue struct' shown in Table 3.5

Table 3.5: The runqueue data structure [19]

Type	Field	Description
spinlock_t	lock	This is the lock that protects the runqueue. Only one task can modify a particular runqueue at any given time.
unsigned long	nr_running	The number of runnable tasks on the runqueue.
unsigned long	cpu_load	The load of the CPU that the runqueue represents.
unsigned long long	nr_switches	The number of context switches that have occurred on a runqueue since its creation. This value isn't actually used for anything useful in the kernel itself. it is used in the proc filesystem as a statistic.
unsigned long	expired_timestamp	Time since last priority array swap (active <–> expired)
unsigned long	nr_uninterruptible	Number of uninterruptible tasks on the runqueue.
unsigned long long	timestamp_last_tick	Timestamp of last scheduler tick. Primarily used in the task hot macro, which decides whether a task should be considered cache hot or not.
task_t *	curr	Pointer to the currently running task.
task_t *	idle	Pointer to a CPU's idle task that runs when nothing else is running
struct mm_struct *	prev_mm	Pointer to the virtual memory mapping of the previously running task. This is used in efficiently handling virtual memory mappings in terms of cache hotness.
prio_array_t *	active	The active priority array. This priority array contains tasks that have time remaining from their time slices.
prio_array_t *	expired	The expired priority array. This priority array contains tasks that have used up their time slices.
prio_array_t	arrays[2]	The actual two priority arrays. Active and expired array pointers switch between these.
int	best_expired_prio	The highest priority of any expired task.
atomic_t	nr_iowait	The number of tasks on a runqueue waiting on I/O. Used for kernel stats.
struct sched_domain *	sd	The scheduler domain that a runqueue belongs to. Essentially this is a group of CPUs that can share tasks between them.
int	push_cpu	The CPU that a runqueue should be pushing tasks to when being balanced.
task_t	*migration_thread	A CPU's migration thread.

3.2.2.2. The priority arrays data structure:

This data structure is most critical structure in particularly Linux 2.6.x kernel O(1) time performance. Linux 2.6.x scheduler always schedules the highest priority task on a system, and if multiple tasks exist at the same priority level, they are scheduled round robin with each other. Priority arrays make finding the highest priority task in a system is a constant-time operation, and also makes round-robin behavior within priority levels possible in constant-time. Each runqueue contains two priority arrays. The arrays field of the runqueue is an array consisting of two prio_array_t structures. All tasks on a CPU begin in one priority array, the active one, and as they run out of their time slices they are moved to the expired priority array. During the move, a new time slice is calculated. When there are no more runnable tasks in the active priority arrays, it is simply swapped with the expired priority array. So, using two priority arrays makes transitions time between when all runnable tasks begin with a fresh time slice and when all runnable tasks have used up their time slices is constant-time. Fields of the priority arrays data structure are shown in Table 3.6 .

Table 3.6: The prio_array_t data structure [19]

Type	Field	Discription
unsigned int	nr_active	The number of active tasks in the priority array unsigned
long	bitmap[BITMAP_SIZE]	The bitmap representing the priorities for which active tasks exist in the priority array. For example - if there are three active tasks, two at priority 4 and one at priority 9, then bits 4 and 9 should be set in this bitmap. This makes searching for the highest priority level in the priority array with a runnable task is constant-time
struct list_head	queue[MAX_PRIO]	An array of linked lists. There is one list in the array for each priority level (MAX_PRIO). The lists contain tasks, and whenever a list's size becomes > 0, the bit for that priority level in the priority array bitmap is set. When a task is added to a priority array, it is added to the list within the array for its priority level. The highest priority task in a priority array is always scheduled first, and tasks within a certain priority level are scheduled round-robin.

Using of these data structures are shown in Figure 3.6

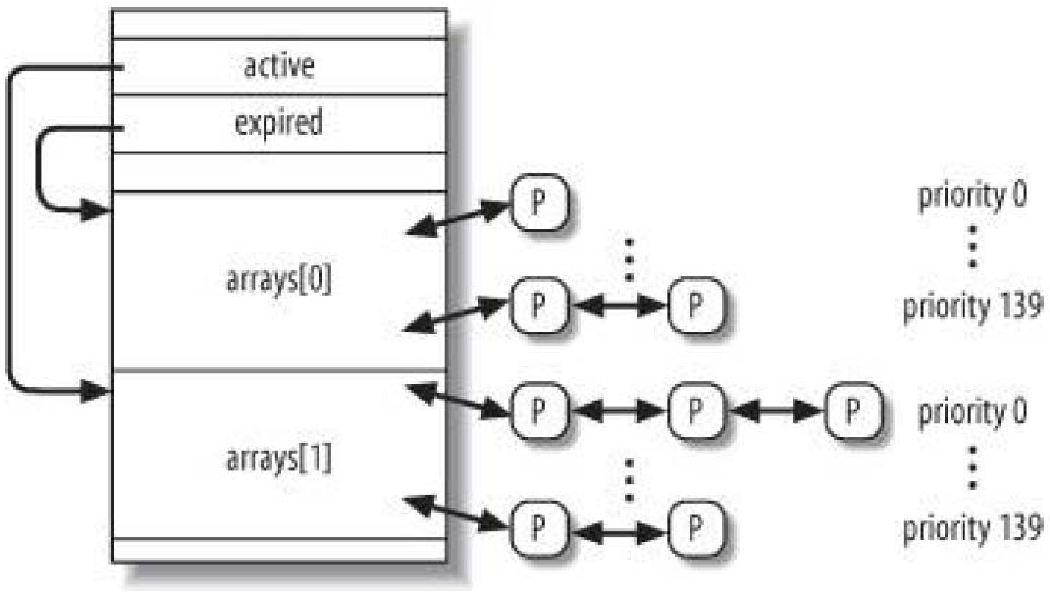


Figure 3.6 : The runqueue structure [17]

3.2.2.3. The process descriptor data structure:

Table 3.7: The fields of process descriptor related to scheduler [17]

Type	Name	Description
unsigned long	thread_info->flags	Stores the TIF_NEED_RESCHED flag, which is set if the scheduler must be invoked
unsigned int	thread_info->cpu	Logical number of the CPU owning the runqueue to which the runnable process belongs
unsigned long	state	The current state of the process
int	prio	Dynamic priority of the process
int	static_prio	Static priority of the process
struct list_head	run_list	Pointers to the next and previous elements in the runqueue list to which the process belongs
prio_array_t *	array	Pointer to the runqueue's prio_array_t set that includes the process
unsigned long	sleep_avg	Average sleep time of the process
unsigned long long	timestamp	Time of last insertion of the process in the runqueue, or time of last process switch involving the process
unsigned long long	last_ran	Time of last process switch that replaced the process
int	activated	Condition code used when the process is awakened
unsigned long	policy	The scheduling class of the process (SCHED_NORMAL, SCHED_RR, or SCHED_FIFO)
cpumask_t	cpus_allowed	Bit mask of the CPUs that can execute the process
unsigned int	time_slice	Ticks left in the time quantum of the process
unsigned int	first_time_slice	Flag set to 1 if the process never exhausted its time quantum
unsigned long	rt_priority	Real-time priority of the process

Process descriptor of each task includes several fields related to scheduling as shown in Table 3.7

When a new process is created, sched_fork(), invoked by copy_process( ), sets the time_slice field of both current (the parent) and p (the child) processes in the following way:

```
p->time_slice = (current->time_slice + 1) >> 1;
```

```
current->time_slice >>= 1;
```

In other words, the number of ticks left to the parent is split in two halves: one for the parent and one for the child. This is done to prevent users from getting an unlimited amount of CPU time by using the following method: the parent process creates a child process that runs the same code and then kills itself; by properly adjusting the creation rate, the child process would always get a fresh quantum before the quantum of its parent expires. This programming trick does not work because the kernel does not reward forks. Similarly, a user cannot hog an unfair share of the processor by starting several background processes in a shell or by opening a lot of windows on a graphical desktop [17].

4. LINUX FOR REAL-TIME APPLICATIONS

Linux was designed to give average performance and the throughput. Real-time operating systems must provide determinism and predictability that Linux wasn't developed to provide them. Therefore, Linux is not a real-time operating system.

4.1. Real-Time Features of Linux

Although Linux is not a real-time system, it has some features, already included in the mainstream source code to provide real-time to Linux.

These, already included in the mainstream source code features, are following [20]:

Scheduling: The Linux scheduler is real-time POSIX compatible. It supports the fixed priority SCHED_FIFO policy, which is the base feature to build a real-time system. It also provides the POSIX required: SCHED_RR and SCHED_OTHER. The range of priorities is [0..99]. In addition, Linux 2.6.x scheduler is able to manage a large number of processes with no overhead degradation. This new scheduler is "O(1) scheduler" developed by Ingo Molnar.

Virtual memory: It is not possible to build real-time applications on a system with virtual memory. The random and long delays are introduced when RAM is exhausted and swapping is required is intolerable in a real-time system. Linux provides the mlock() and mlockall() functions that disables paging for the specified range of memory, or for the whole process respectively. Therefore, all the "locked" memory will stay in RAM until the process exits or unlocks the memory. mlock() and mlockall() are included in the POSIX real-time extensions.

Shared memory: Linux processes can share memory with each other and with drivers the POSIX.1b call mmap(). Linux provides open shared memory objects, which is part of the POSIX real-time extensions.

Real-time signals: Signals take an important role in real-time as the way to inform the processes of the occurrence of asynchronous events like high-resolution timer expiration, fast interprocess message arrival, asynchronous I/O completion and explicit signal delivery. Linux fully supports the POSIX real-time signals standard.

POSIX asynchronous I/O: The standard way to access I/O devices defined by UNIX the read(), write() blocking sequence, where a next file access is performed only when the previous request has been completed. AIO mechanism provides the ability to overlap application processing and I/O operations initiated by the application. A process can start one or more IO requests to a single file or multiples files and continue its execution. Also, a single system call can start a sequence of I/O operation on one or several files, which reduces the overhead due to context switches. Linux-AIO provides this functionality to newer kernels.

POSIX threads: Current Linux implementation of POSIX threads (POSIX 1003.1c) known as LinuxThreads. LinuxThreads is now integrated in the glibc, and distributed as a part of it.

4.2. Reasons of Why Linux Is Not Real-Time

Linux kernel is not preemptable that is, while the processor executes kernel code, no other process or event can preempt kernel execution. Therefore, Standard Linux kernel is not a RTOS.

4.2.1. Non-preemptible kernel

The main handicap to consider Linux as a real-time system is that the kernel is not preemptible; that is, while the processor executes kernel code, no other process or event can preempt kernel execution. Kernel code refers to interrupt handlers, bottom halves, tasklets, system calls etc [21].

4.2.2. Non-deterministic response time

Response time is defined as the amount of time that elapses from when an interrupt is generated and to when the task that was waiting for it runs. This has to be deterministic and low values are preferred. There are four major factors that make up the response time of the Linux kernel and these are [21]:

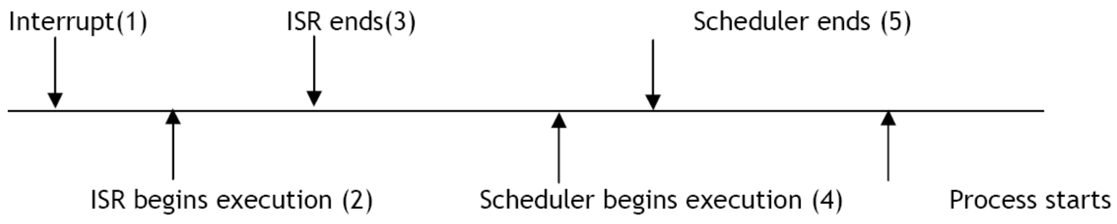


Figure 4.1 : Execution points of Linux kernel interrupt response [21]

Interrupt Latency: which is the time between when an interrupt is generated and running the appropriate interrupt service routine. Kernel disables interrupts (cli) at times to guarantee exclusive access to resources, which leads to increase and non-determinism in interrupt latency. Time elapsed between 1 and 2 in Figure 4.1.

Interrupt Duration: which is the amount of time spent in the interrupt handler, depends mostly on device drivers. Time elapsed between 2 and 3 in Figure 4.1.

Scheduler Latency**:** Scheduler Latency is the delay between the occurrence of an interrupt and the running of the scheduling routine. This is due to two reasons:

- Non-preemptible parts of Linux kernel to guarantee exclusive access.
- Scheduler being called at some fixed timeout (scheduler tick, 10ms in i386) and not at all arbitrary time.

In simple terms, it is delay in continuing a newly awakened high-priority task. Time elapsed between 3 and 4 in Figure 4.1.

Scheduler Duration: Scheduler duration is the amount of time actually spent in the scheduler routine. With Linux this depends on the number of tasks, since this of O(n) complexity. Time elapsed between 4 and 5 in Figure 4.1.

Even when the scheduler finishes execution it might choose some other task instead of user expected task depending on priority and time slice left. Assuming that task is chosen, it still has to do a context switch, which adds to the response time.

4.2.3. Priority inheritance

Priority inheritance protocol prevent occurrence of priority inversion. In priority inheritance protocol, if a high priority-task blocks for a critical section, a low-priority task that has holds the lock for the section gets a priority boost. It inherits the priority of the blocked task. This prevents the unbounded priority inversion that could occur if medium priority tasks would preempt the lower-priority task that holds the lock. Note that, with priority inheritance, the medium-priority tasks suffer priority inversion as well [22].

Real-time operating system must prevent occurrence of priority inversion to be deterministic. Linux has no mechanism to prevent priority inversion and no priority inheritance protocol.

4.3. The Real-Time Linux Solutions

Linux was originally designed as non-preemptive kernel and was not well suited for real-time applications, since processes may spend several milliseconds in Kernel mode while handling interrupts. The response time requirements for applications are directly tied to the kernel's ability to preempt a running process and switch to another process with a higher priority very quickly. The lack of kernel preemption in Linux means that long system calls can delay the execution of a user process with high priority for relatively long period of time, even tens of milliseconds in the Linux 2.4 kernel. The lack of kernel preemption results in kernel latency. This behavior is not acceptable for the applications that require predictable and low response times [23]. As a result, Linux Kernel latency is not acceptable for real-time application because of its nondeterministic behavior and possibility to exceed tens of milliseconds. Therefore, Linux kernel has been modified in several ways to reduce the kernel latency and to achieve deterministic response time.

4.3.1. Preemptable Linux

Linux kernel was designed as monolithic kernel means that at any given time there is an only one thread of execution in kernel. Nature of this monolithic design, any other task that is also high priority can not preempt the running thread. This results in non-deterministic scheduler latency and a big problem for real time applications. Basic idea behind the preemptable Linux is to remove the constraint of a single execution flow inside the kernel. Kernel data must be explicitly protected using mutexes or spinlocks to support full kernel preemptability. In preemptable Linux approach, kernel preemption is only disabled when a spinlock is held. In similar way, mutexes instead of spinlocks can be used to achieve priority inheritance [23].

In a preemptable kernel, the maximum latency is determined by the maximum amount of time for which a spinlock is held inside the kernel. There is an additional patch called lock-breaking that merges some of the low-latency rescheduling points into the preemptable kernel to decrease the maximum latency caused by amount of time for which spinlocks are held. Modifications in preemptable Linux are followings:

- A variable named preempt_count was added to the task structure that maintains state for each thread.
- Macros preempt_disable(), preempt_enable() and preempt_enable_no_resched() was added to modify the preempt_count. The preempt_disable() macro increments the preempt_count variable, while the

preempt_enable() macros decrement it. The preempt_enable() macro checks for a reschedule request by testing the value of need_resched and if it is true and the preempt_count variable is zero, calls the function preempt_schedule() .

- The scheduler has been modified to check preempt_count for need_resched.
- The macro spin_lock() was modified to first call preempt_disable() , then actually manipulate the spinlock variable.
- The macro spin_unlock() was modified to manipulate the lock variable and then call preempt_enable().
- The macro spin_trylock() was modified to first call preempt_disable() and then call preempt_enable() if the lock was not acquired.
- Interrupt return path code was modified to make the same test done by preempt_enable() and calls the preempt_schedule() routine if conditions are right.

The preemption modifications reduce the amount of time between when a wakeup occurs and sets the need_resched flag and when the scheduler may be run. Because, each time a spinlock is released or an interrupt routine returns, there is an opportunity to reschedule.

4.3.2. Low latency Linux

Main modification in this approach is focus on decreasing size of non-preemptable sections. When a thread is executing inside the kernel, it can explicitly decide to yield the CPU to some other thread.

The modifications in this approach add explicit preemption points in blocks of code that may execute for long stretches of time. The modifications are finding places that iterate over large data structures and figure out how to safely introduce a call to the scheduler. The low latency patches are not so simple to implement. Finding and fixing blocks of code that contribute to high scheduler latency is a time intensive debugging task [21].

As an example of preemption points, following code fragments show a block of code before low latency patch and after patch.

Before low latency Linux patch:

```
void example_block_of_code(int count)
```

```
{
spin_lock(&example_lock);
while (1) {
          do_somethings();
          if (!--count ){
                    break;
}
          }
spin_unlock(&example_lock);
}
```

After low latency Linux patch:

```
void example_block_of_code(int count)
{
DEFINE_RESCHED_COUNT;
redo:
spin_lock(&example_lock);
          while (1) {
                    if (TEST_RESCHED_COUNT(50)) {
                              RESET_RESCHED_COUNT();
                              if (conditional_schedule_needed()) {
                                        spin_unlock(&example_lock);
                                        unconditional_schedule();
                                        goto redo;
                              }
                    }
                    do_somethings();
                    if (!--count ){
                              break;
                    }
          }
spin_unlock(&example_lock);
}
```

The Macro DEFINE_RESCHED_COUNT defines a counter variable. The label redo is added. The TEST_RESCHED_COUNT(50) macro increments the counter variable, tests it against the argument and returns true if the variable is greater than or equal to the input argument. So, after 50 iterations of the loop, the variable will be true and the ‘if statement’ body will be executed. The body resets the counter value to zero, then checks to see if low latency is enabled and a rescheduling request is pending (current−>need_resched != 0). If a rescheduling pass is needed, the example_lock is dropped, the scheduler is called by unconditional_schedule() and the code then jumps to the label, which reclaims the example_lock and starts the process again. This style of lock breaking works because there is no order imposed on the list. If need_resched is just checked and count loop iteration is not performed, it would be possible for the system to do nothing but loop between the redo label and the test of need_resched by only testing need_resched after a set of loop iterations. Count loop iterations insure that some work gets done [24].

4.3.3. O(1) scheduler

Linux kernel 2.4 scheduler has O(n) type scheduler algorithm, increase its operation time while its load increase. O(1) type scheduler is a new Linux patch that is accepted into mainstream code and it is now part of Linux kernel 2.6, rewrites the Linux old scheduler. This patch provides O(1) scheduling algorithm, better SMP scalability and affinity, and handles extreme loads more smoothly.

The core of the new scheduler is two arrays. One of them contains all tasks that haven’t used up time slices are called active arrays. Other one contains all tasks that have used up time slices are called expired arrays. If a task used up its time slice, the task moved to expired array and new time slice of the task is recalculated. If all time silences of active tasks are finished, new time slices calculation is as simple as changing expired and active array pointer. So, it doesn’t require any additional time. In this way, Linux 2.6 kernel scheduler scales well with huge number of task [21].

4.3.4. Increasing timer resolution

Linux 2.6 kernel timer resolution was increased to 1 ms while Linux 2.4 kernel has 10ms timer resolution. Benefits of increasing timer resolution are following:

- Increasing timer resolution allows to occurring of process preemption more accurately. Because when a event made a task ready, it is necessary to call schedule to start the task execution. Higher timer resolution results to run scheduling more frequently and decreases scheduler latency.
- Increasing timer resolution allows better accuracy and finer resolution for kernel timers.

- System calls that use a timeout value can be executed with improved precision.
- System resource usage can be measured and recorded with finer resolution.

Increasing timer resolution also results in higher overhead, because the processor must spend more time executing the interrupt handler [21].

4.3.5. High resolution timers

Linux provides POSIX timers with a timing resolution around 10ms, which is not suitable of real-time applications. The High Resolution Timers provides microsecond resolution with lower overhead for applications that require better timing resolution and greater accuracy than the standard 10 millisecond Linux time base can provide, because The High Resolution Timers use of several timing and interrupt hardware sources such as 8254, the Pentium internal instruction TSC and the ACPI timers when available. HRT provides increased control over real-time application behavior, more predictable timer response with much less jitter. Programmers can implement time-bases and event driven algorithms with microsecond accuracy with HRT. In addition, using HRT eliminate the need for cycle-wasting polling and busy loops [21].

4.3.6. Dual kernel approaches

In dual kernel approach, there is a second kernel that is an interface layer between the standard kernel and the hardware layer as shown in Figure 4.2. The second kernel, or in this approach call it the real time kernel, treats the Linux operating system kernel as the idle task. Linux only executes when there are no real time tasks to run, and the real time kernel is inactive. The Linux task can never block interrupts or prevent itself from being preempted. Main design issue is the software emulation of interrupts control hardware. Linux is not permitted to really disable hardware interrupts, and it cannot add latency to the interrupt response time of the real time system. When an interrupt occurs, the real time kernel intercepts dispatch the interrupt. If there is a real time handler for the interrupt, the appropriate handler is invoked. If there is no real time interrupt handler, Linux interrupt handler invoked. Regardless of the state of Linux, the real-time system is always able to respond to an interrupt. Dual kernel architecture is shown in Figure 4.2 .

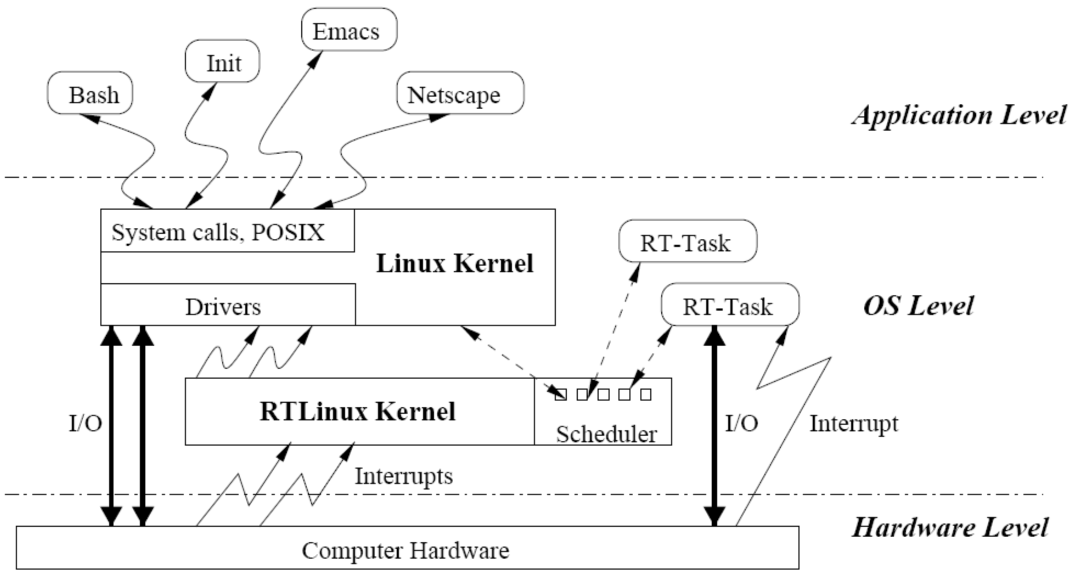


Figure 4.2 : Dual kernel real-time architecture [5]

In dual kernel approach, the real time kernel never waits for the Linux side to release any resources. Linux side and real-time side communicate with FIFO and data transfer between real time tasks and Linux processes are non-blocking on the real time side [25].

4.3.7. Nano kernel

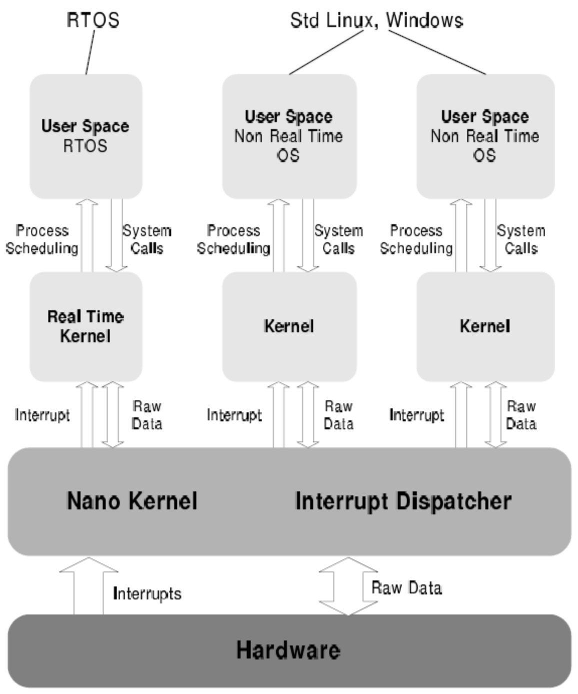


Figure 4.3 : Nano kernel architecture [9]

In the nano kernel approach, many operating systems can be run in parallel on top of the nano kernel layer as shown in Figure 4.3. One of the operating systems can be real time, the others can be non-real-time operating systems. The purpose of The Adaptive Domain Environment for Operating Systems is to provide a flexible environment for sharing hardware resources among multiple operating systems, or among multiple instances of a single OS [34]. ADEOS developers have called their hardware to kernel interface a nano-kernel rather than a micro-kernel to further distinguish their contribution from the patented work of RTLinux [9].

4.3.8. Resource kernel

Real-time systems are typically built using timeline based approaches, production/consumption rates or priority-based schemes, where the resource demands are mapped to specific time slots or priority levels. This mapping of resources to currently available scheduling mechanisms introduces many problems. Assumptions go undocumented, and violations go undetected with the end result that the system can become fragile and fail in unexpected ways. Resource Kernel argues for a resource-centric approach where the scheduling policies are completed subsumed by the kernel, and applications need only specify their resource and timing requirements. The kernel will then make internal scheduling decisions such that these requirements are guaranteed to be satisfied [26]. Resource kernel architecture is shown in Figure 4.4 .

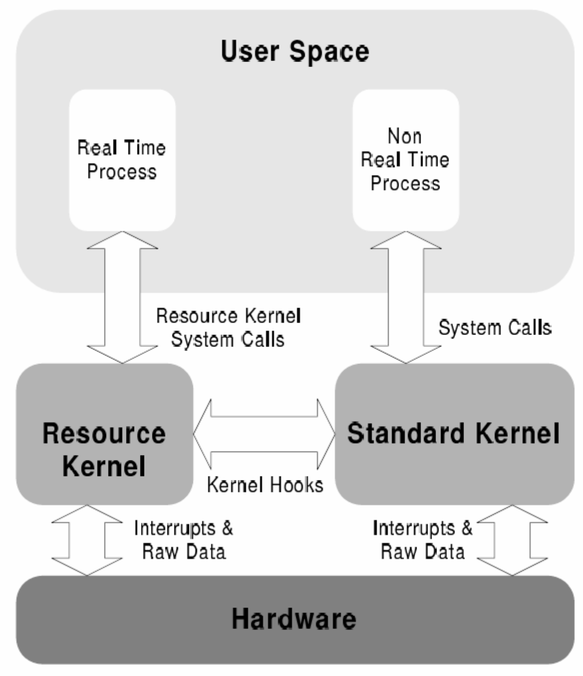


Figure 4.4 : Resource kernel architecture [9]

4.4. Performance Characteristics

The fundamental data points to figure out performance of a real-time operating system are event latency, scheduler latency and periodic jitter.

4.4.1. Scheduler latency

Scheduler latency is the delay between the occurrence of a wakeup signaling and the running of the scheduling routine. Wakeup signaling can be an interrupt generated by hardware external to the CPU or it can be a signal internal to the operating system such as a notification to start the next task on a ready-to-run queue. There are blocks of kernel code spends a long period of time in processing without giving a chance for the scheduler to run such as some device drivers. These kernel codes cause that scheduler not to be run more often and the kernel does not have the opportunity to perform scheduling calculations for a long period of time. Since the scheduler is the mechanism for determining what thread should run, it needs to be run with a relatively high frequency for the kernel to good real-time performance [24].

There are two data points to discuss scheduler latency. The first is the maximum latency, which is the largest value of scheduler latency measured in a test run. This value is almost always statistically insignificant. Maximum latency is most important for applications that something catastrophic will happen if the system is late. So the maximum latency is usually very interesting to system designers. Second data point is jitter in scheduler response. When system designers are designing a system that needs a scheduler to be good enough, and can tolerate some in jitter [24].

4.4.2. Event latency

Events can be either hardware generated interrupts or operating system generated software signals. In the case of a hardware interrupt, the latency is the time from when an interrupt requests service to the time when the first instruction of the interrupt-service routine is executed as shown in Figure 4.5. One consideration in measuring interrupt latency is that the measurement will include any delays introduced by the computer-system peripheral bus. Such delays could be due to other peripheral devices transferring data over the bus and preempting CPU bus access [9].

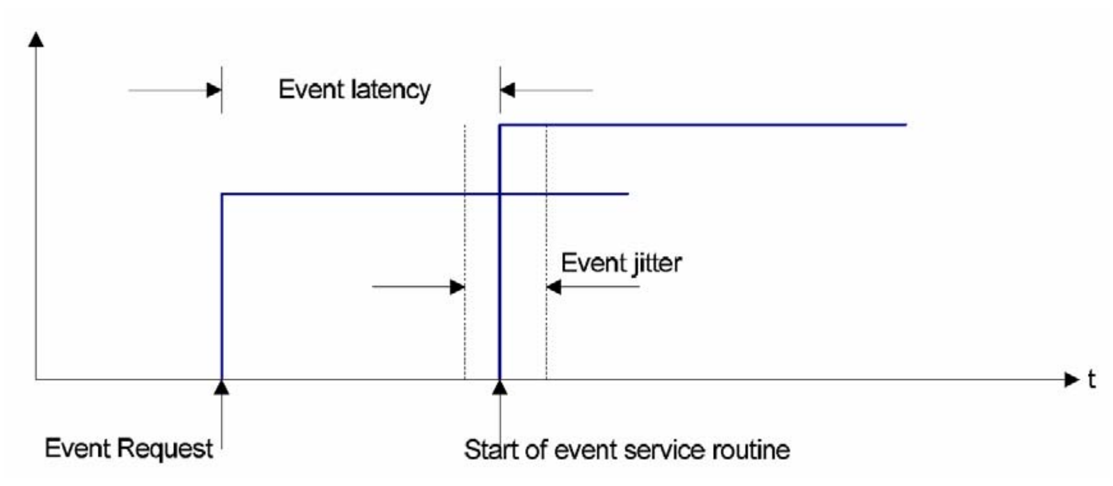


Figure 4.5 : Event latency [9]

4.4.3. Periodic jitter

Periodic jitter is amount of time variations that a repetitive task executes as shown in Figure 4.6. The repetitive task is at the heart of sampled data control of mechanical devices. Sampling data application has a sample time that is periodic and fixed. The control algorithms are, in turn, calculated from the device model, reinforcing the dependence on a known and stable sample time. Any jitter in the sample time leads to imprecision in the control-system performance [9].

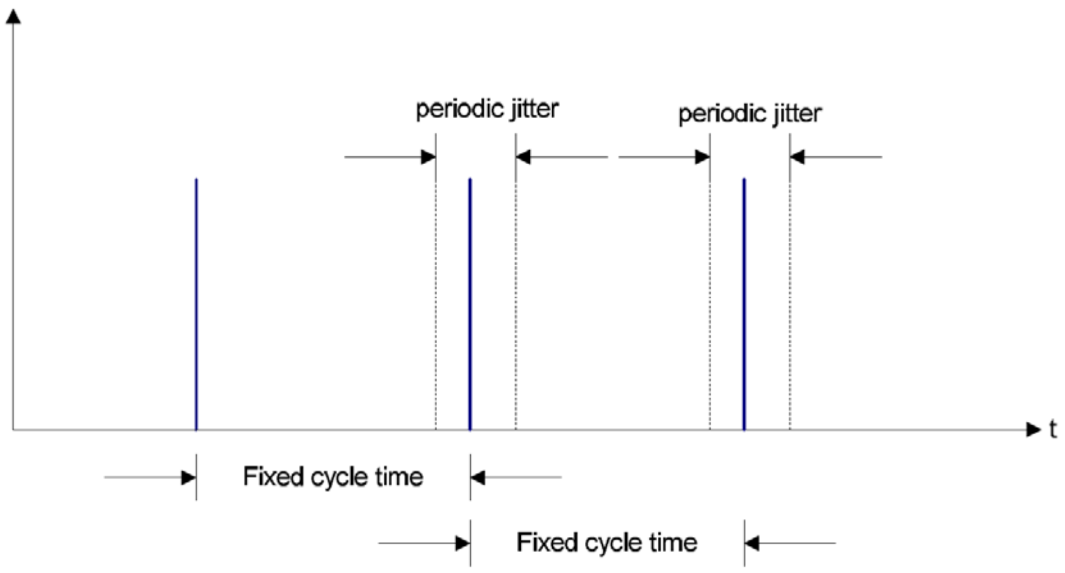


Figure 4.6 : Periodic jitter [9]

4.5. Comparison of Preemptable Linux and Low-latency Linux

Preemptable Linux and Low-latency Linux was designed to reduce scheduler response time. All these patches come with extra cost in terms of performance and throughput. They have different effect on average latency and maximum latency. Implementation and ability to use existing driver is another important selection criteria.

4.5.1. Performance comparison

In this chapter, scheduler latency measurement results of preemptible Linux and low-latency Linux are given to show performances of latency patches. All test results in this chapter were taken from “Linux Scheduler Latency” by Clark Williams, Red Hat [24]. In this measurement, a program called Realfeel Test [27] was used to specifically measure scheduler latency. The tests were run on a 700MHz AMD Duron system with 360MB RAM and Realfeel program runs for 5 million iterations at an interrupt frequency of 2048 interrupts per second.

Realfeel benchmarking test results of a vanilla 2.4.17 kernel are shown in Figure 4.7

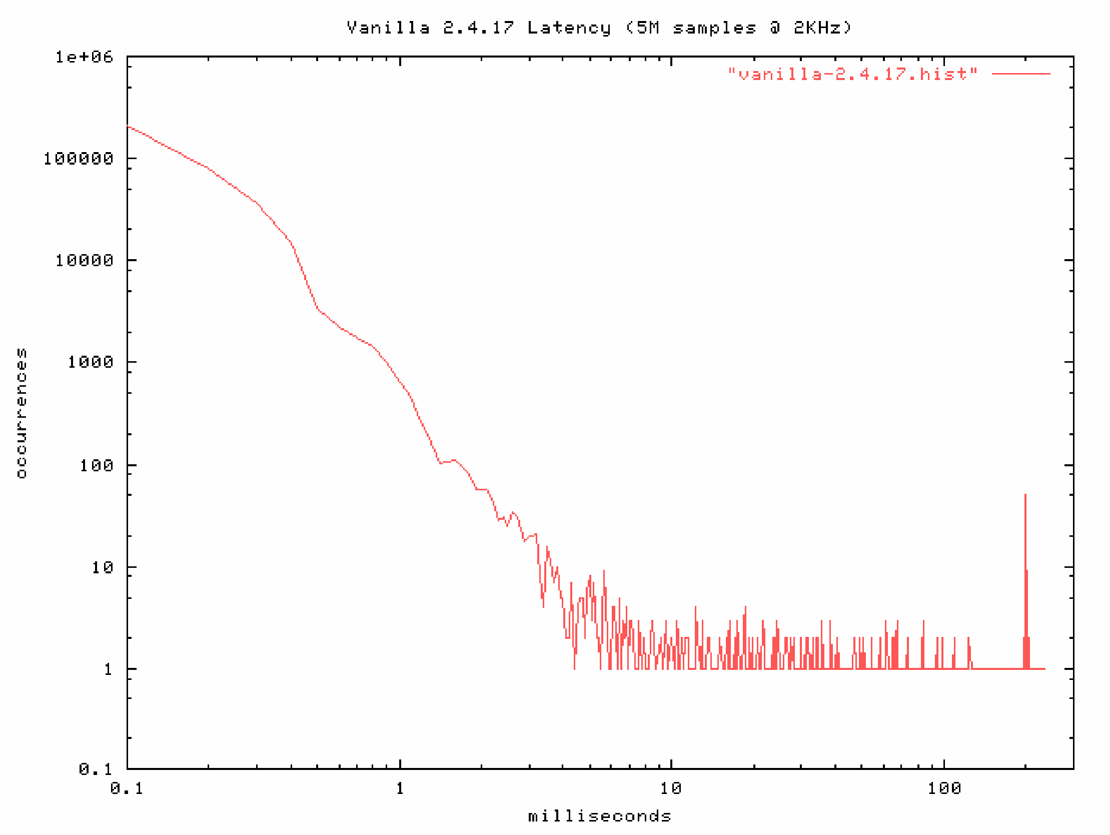


Figure 4.7 : Realfeel benchmarking test results of a vanilla 2.4.17 kernel [24]

Maximum latency: 232.6ms, Average: 0.088ms

Realfeel benchmarking test results of 2.4.17 kernel with preemption patch are shown in Figure 4.8

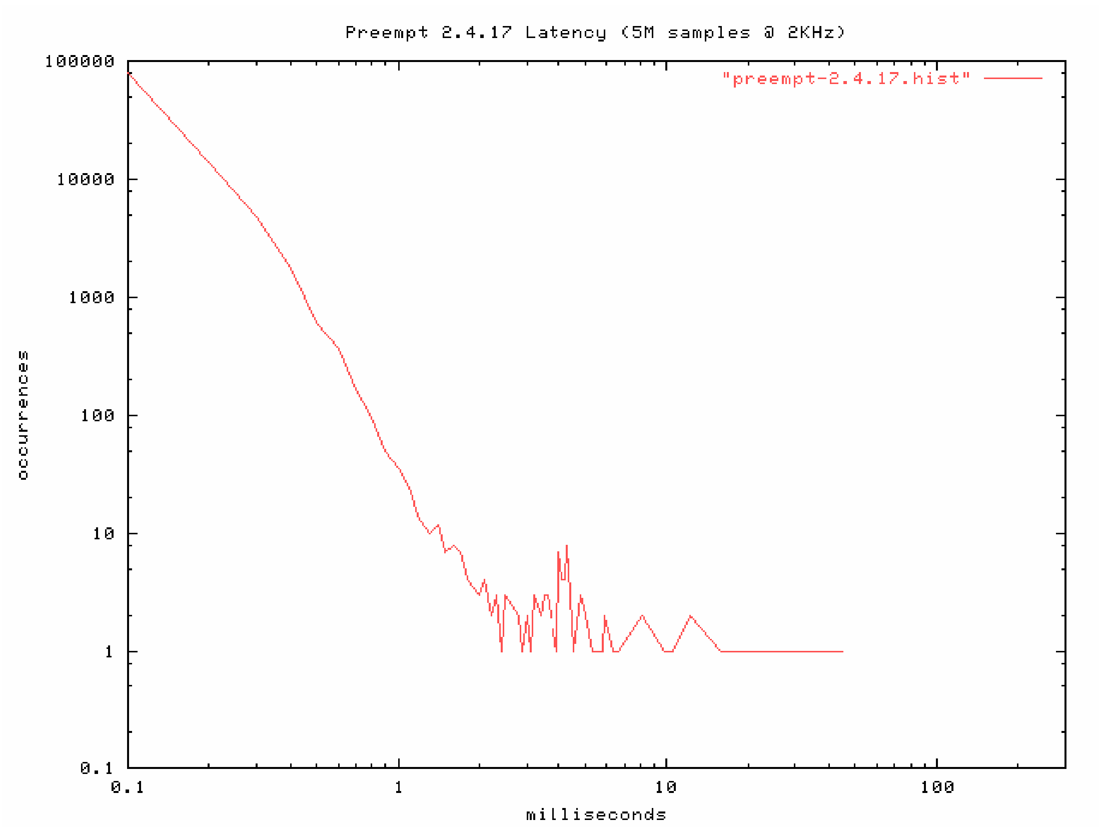


Figure 4.8 : Realfeel benchmarking test results of a preempt 2.4.17 kernel [24]

Maximum latency: 45.2ms, Average: 0.0528ms

Realfeel benchmarking test results of 2.4.17 kernel with low-latency patch are shown in Figure 4.9

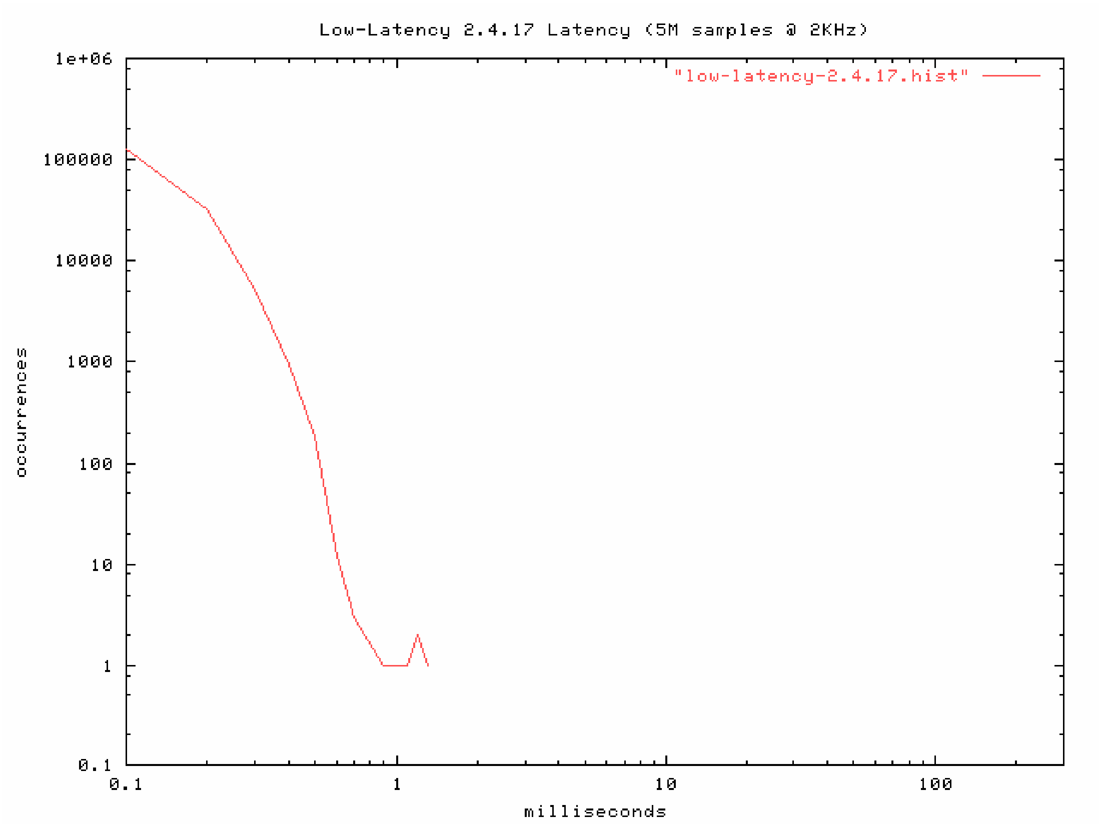


Figure 4.9 : Realfeel benchmarking test results of a low latency 2.4.17 kernel [24]

Maximum latency: 1.3ms, Average: 0.0542ms

Comparison of low-latency and preempt patches according to test results are shown in Figure 4.10

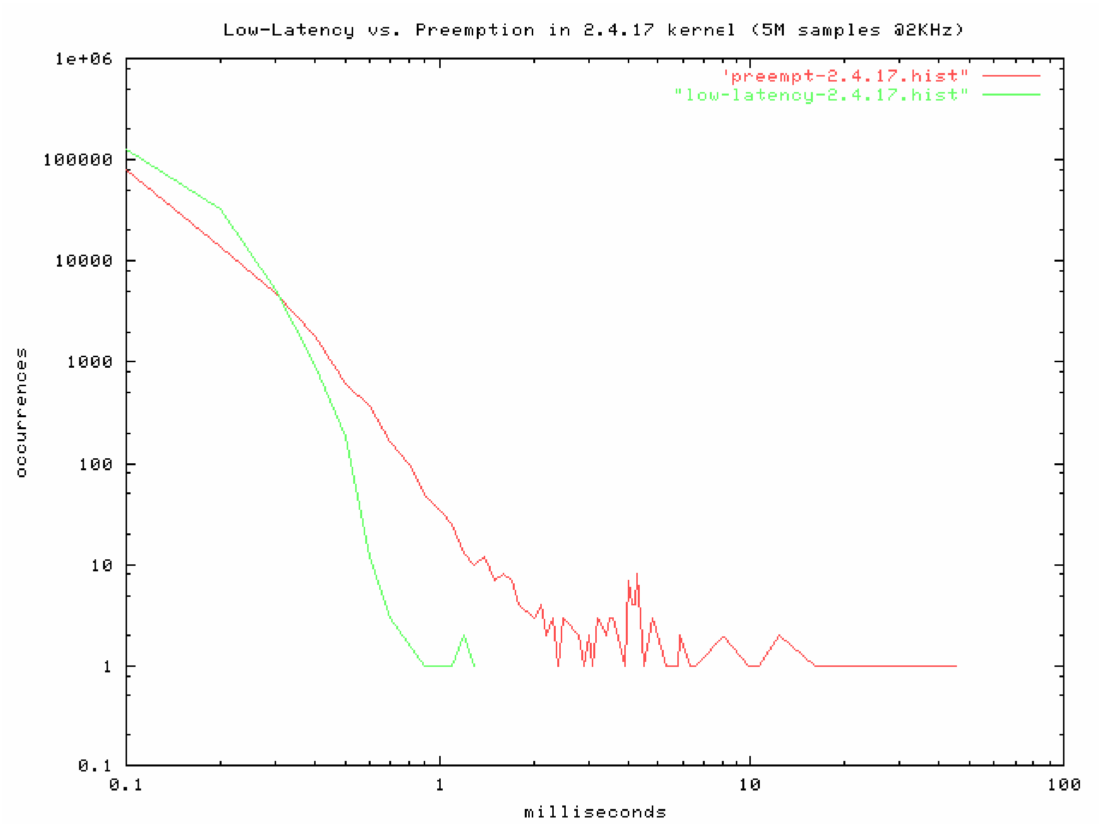


Figure 4.10 : A preempt 2.4.17 kernel vs low-latency 2.4.17 kernel [24]

According to Figure 4.10, it is possible to see that the low latency kernel is characterized by larger average latencies respect to the preemptable kernel, but reduces the worst-case latencies.

4.5.2. Implementation comparison

Preemptable Linux and Low-latency Linux have effectively similar scheduler response performance. Therefore, implementation and ability to use existing driver is more important selection criteria.

Low latency patch isn't interesting in coding of device drivers, because the low latency patch effectively adds preemption points within the kernel source code. If third party binary device drivers that source code is not available will be used, this is important. However, in the preemptable kernel patch, it is necessary to ensure that all device drivers have been written to perform proper use of spinlocks, semaphores and mutexes.

5. IMPLEMENTATION OF UNINTRUSIVELY LINUX KERNEL REAL-TIME PERFORMANCE MEASUREMENT

5.1. Related Work

There are a variety of measurements tools and methodologies available, yet none of them suited our needs perfectly. For the measurement of real-time performance very accurate and precise, functions of the measurement system should not use the system functions itself and the measurement data should be gathered in a way which influences the timing of the system functions as little as possible that is called unintrusively measurement.

Linux kernel has many interrupt handling mechanism and all of them provide solution to some special problems. To fully understand real-time performance of Linux, latency or response time of these distinct mechanisms must be measured.

The Linux Tracing Toolkit is a universal profiling and event tracking tool. It provides timing measurements for both user mode and kernel mode. To analyze the collected data, a set of powerful tools is available. Unfortunately, user mode measurement points can only be established using a system call, which violates our requirements [28].

The kernel profiler kit uses statistical profiling which is far too inaccurate for our project and lacks the opportunity to profile the interaction between user and kernel mode [29].

The Realfeel Test was written by Mark Hahn and modified by Andrew Morton. Realfeel Test uses periodic interrupts from RTC and read() file operation of RTC device. The read() file operation of RTC device blocks to current thread that called the read service if the data hasn’t been modified yet by RTC interrupt handler. RTC interrupt handler wake up the sleeping thread and then read() operation is finished. The difference between the time when RTC interrupt occurrence and when the read operation finished is scheduler latency that the realfeel program use for benchmarking test [24]. Following is code fragment of realfeel:

```
int main() {
   int fd = open("/dev/rtc",O_RDONLY);
```

```
    while (1) {
            u64 now;
            double delay;
            int data;
            if (read(fd, &data, sizeof(data)) == -1)
                fatal("blocking read failed");
            now = rdtsc();
            delay = secondsPerTick * (now - last);
            printf("%f\n",1e6 * (ideal - delay));
            last = now;
    }
}
```

The realfeel program is measure scheduler latency plus interrupt handling latency as scheduler latency. The test program doesn’t test linux interrupt handling response times [27].

The intlat program allows measuring how long the kernel spends with interrupts disabled. Kernel may block interrupts for a significantly longer time. The intlat program use the amount of time between the interrupt being taken by the __sti() or local_irq_save() and the amount of time between the __cli() or local_irq_restore() to represent periods when interrupts are blocked. But, the program doesn’t represent interrupt latencies [30, 31].

The schedlat program is a tool from Andrew Morton to measure intervals during which the kernel is ignoring reschedule requests. What schedlat does is to record the interval between a process entering the kernel and leaving the kernel [32].

LMBench is suite of performance tests written in C and is fairly portable. It includes Memory Latency, IPC Latency and Context switching tests. The test suite use system call, which violates our requirements [33].

Basically, all of these existing methods measure the performance intrusively. These existing methods run on top the target machine and they use system calls or functions of measured systems. In addition, these methods are lack of adjusting test system event timing parameters according to real external interrupt or events. Also, these methods can not measure different interrupt handling mechanisms of Linux kernel separately. As a result, data’s that obtained using these methods are not give good understanding of Linux system real-time performance.

5.2. Unintrusively Interrupt Latency Measuring Method

In our design, we develop a methodology to perform unintrusively measurement of Linux interrupt and scheduler latencies. In addition to unintrusively measuring, we develop an embedded system to generate interrupt to Linux machine with adjustable timing via host machine user interface program.

The unintrusively Linux kernel latencies measuring system has distinct parts. One part is host machine user interface program called measurement suite user interface. Second part is an embedded system called embedded interrupt generator. Third part is parallel port device driver called performx that is installed on target Linux machine. Last part is user space programs that run on target Linux machine called schedule_api and loads. The measurement system is shown in Figure 5.1 .

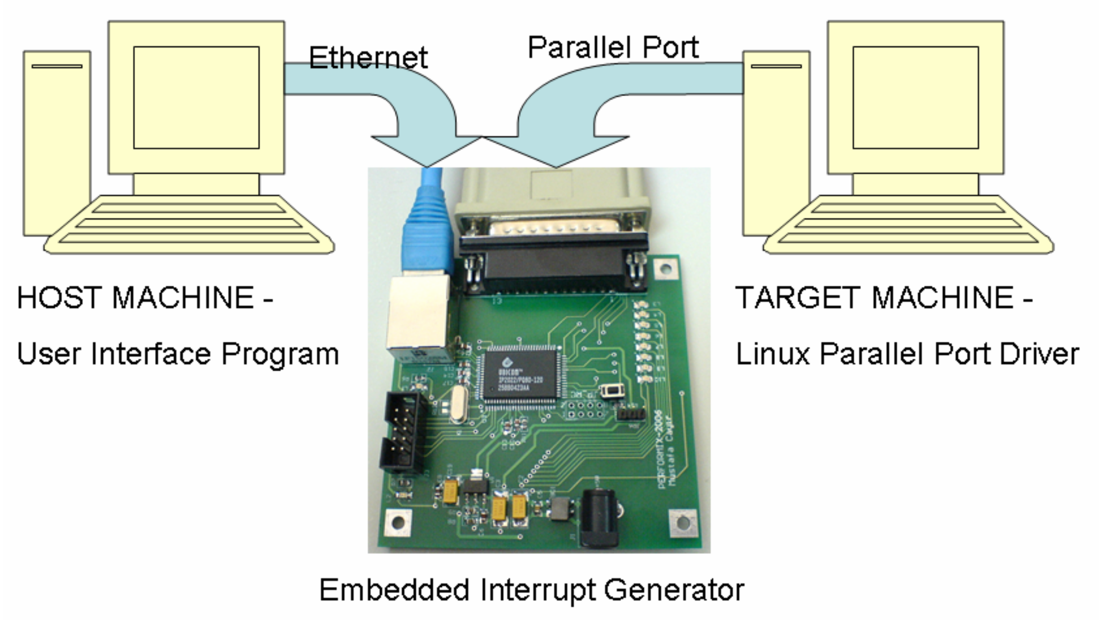
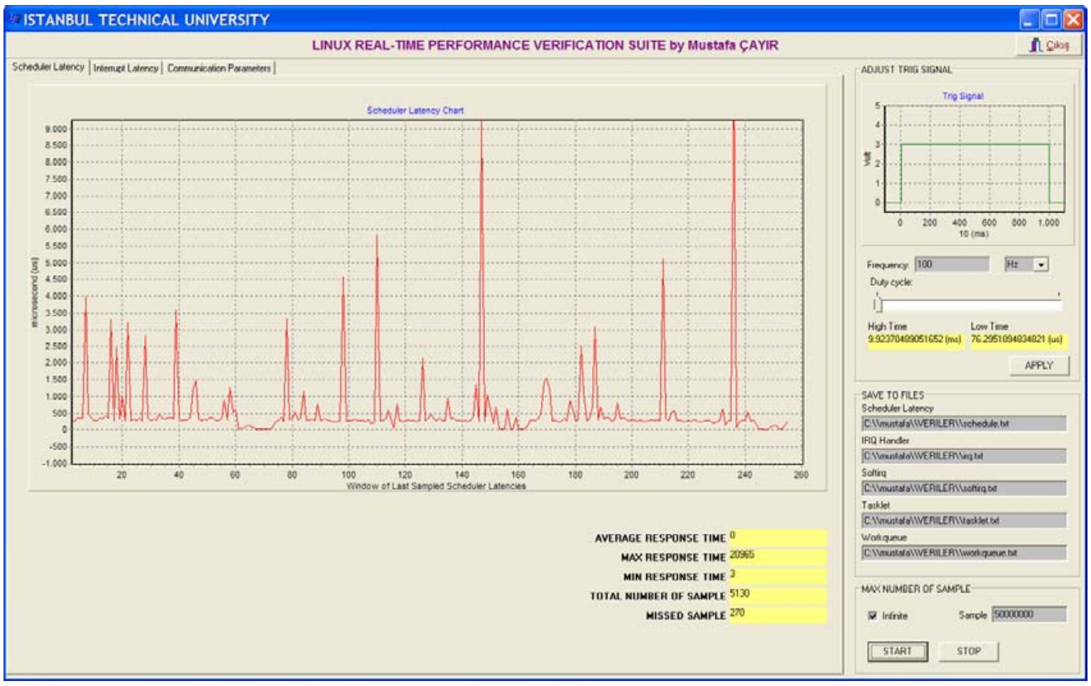


Figure 5.1 : The measurement setup

Basically, to measure the latency, the system uses parallel port of target Linux PC. This port has an eight data pins (D0-D7) and one acknowledge pin (ACK). A rising edge on acknowledge pin causes the parallel port logic to generate an interrupt to Linux kernel. Embedded interrupt generator generates an interrupt by activating the ACK pin. Parallel port device driver has service routines to response the interrupt including all type of interrupt handlers that are irq handler, softirq, tasklet and workqueue. These service routines change logic level of corresponding parallel port pin.

5.3. Part 1 - Measurement Suite User Interface Program

Measurement suite user interface program runs on Windows XP host machine. Basically, responsibilities of the program are communication to embedded part, logging received latency values, displaying results in graphical chart window and providing interface to adjust trig frequency and duty cycle.



Frequency: 100 Hz

Duty cycle:

High Time 9.92370489051652 (ms) Low Time 76.2951094834821 (us)

Scheduler Latency C:\\mustafa\\VERILER\\schedule.txt

IRQ Handler C:\\mustafa\\VERILER\\irq.txt

Softirq C:\\mustafa\\VERILER\\softirq.txt

Tasklet C:\\mustafa\\VERILER\\tasklet.txt

Workqueue C:\\mustafa\\VERILER\\workqueue.txt

☑ Infinite Sample 50000000

AVERAGE RESPONSE TIME 0
MAX RESPONSE TIME 20965
MIN RESPONSE TIME 3
TOTAL NUMBER OF SAMPLE 5130
MISSED SAMPLE 270

Figure 5.2 : Measurement suite user interface

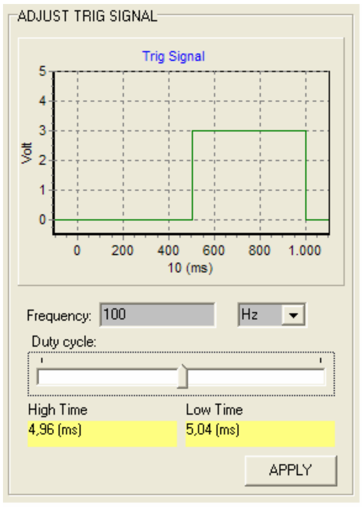
5.3.1. Communication to embedded interrupt generator:

Measurement suite user interface program and embedded interrupt generator communicate each other using TCP/IP socket connections. The user interface program act as a TCP/IP client. User must give IP and port number of embedded part and click to connect button. After the connection is established, embedded part starts to send latency values via this socket connection. The program parse received data and perform its operations.

The program can send three commands via TCP/IP socket connection. These commands are set trig frequency, set trig duty cycle and set number of measurement per packet.

5.3.2. Adjust trig signal

The program provides interface to adjust trig frequency and duty cycle as shown in Figure 5.3. Trig frequency can be adjusted 16 Hz to 120 MHz.



Frequency: 100 Hz

High Time 4,96 (ms)

Low Time 5,04 (ms)

Figure 5.3 : Interface to adjust trig signal

The program sends appropriate commands to embedded interrupt generator to adjust timing of trig signal. User can easily adjust frequency and duty cycle of trig signal that is generated by embedded interrupt generator using graphical interface shown Figure 5.3.

5.3.3. Display and log latency values

The program has distinct charts for every interrupt handling mechanisms and scheduler latency. In addition, the program shows maximum, minimum, average values and number of sample. All that displays are updated when a new network packet received. Received data's are saved to log files.

5.4. Part 2 - Embedded Interrupt Generator

Embedded Interrupt Generator is a 16 bit microcontroller based system that was developed with Ubicom IP20202 Network processor. The processors combine support for communication physical layer, Internet protocol stack and device-specific application. The chip is a RISC processor operates at 120 MHz. Schematics of embedded interrupt generator is shown in Figure 5.4

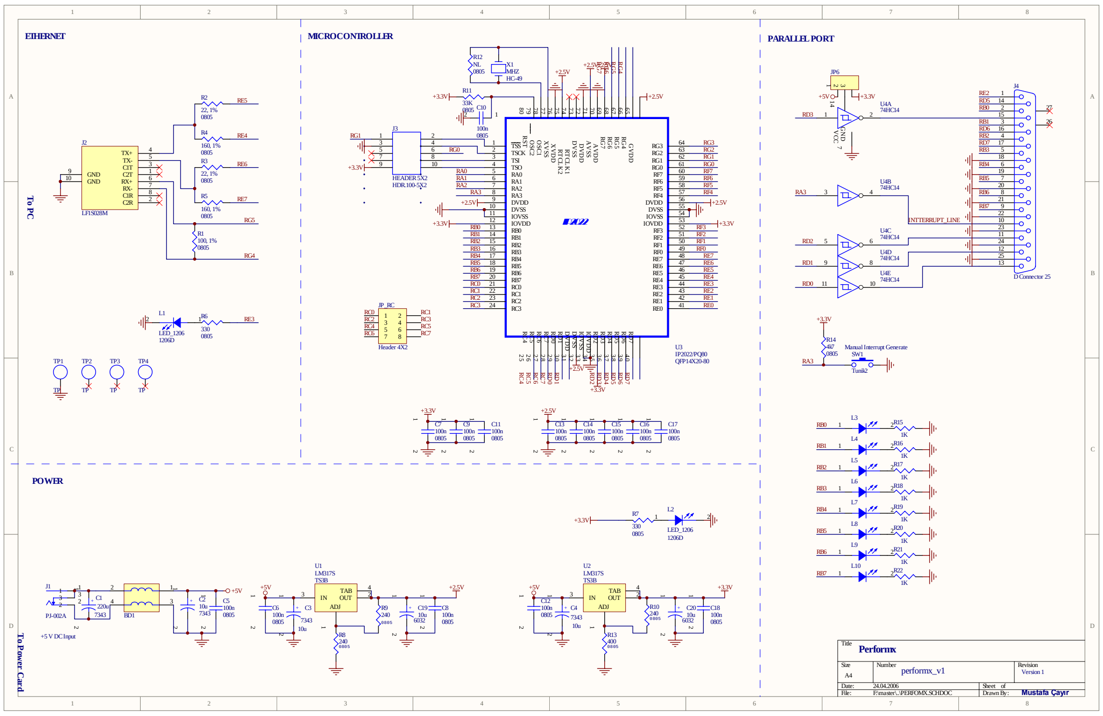


Figure 5.4 : Schematics of embedded interrupt generator

5.4.1. Generating trig signal:

Main job of this part is simulating real world interrupts. Embedded interrupt generator generates a rising edge on acknowledge pin causes the parallel port logic to generate an interrupt to Linux kernel. High and low times of ACK pin can be adjusted form tens of nanosecond to milliseconds to achieve more similarity to real world interrupts.

Trig signal generated using microcontroller PULSE WIDTH MODULATOR timer mode. Sources that responsible of PWM are pwm.c and pwn.h . Followings are the functions implemented in pwm.c

void pwm_set_duty_cycle(u16_t duty_cycle_percent)

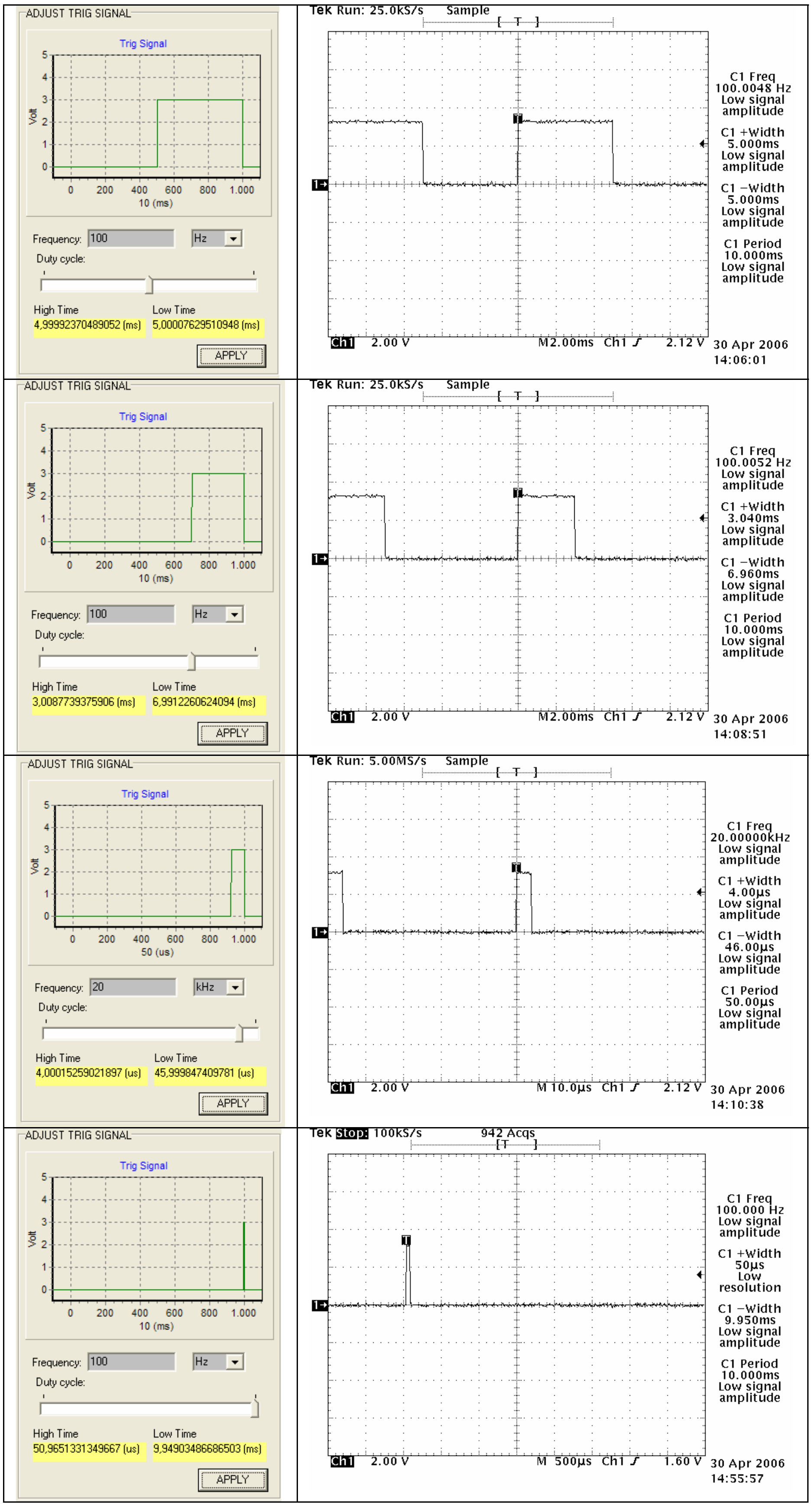
void pwm_set_frequency(u32_t freq)

void pwm_start(void)

void pwm_stop(void)

void pwn_init(void)

Adjusting trig signal example is shown in Figure 5.5. Snap shots of user interface program and corresponding output at embedded interrupt generator trig pin are shown in Figure 5.5.



Frequency: 100 Hz
Duty cycle:
High Time 4,99992370489052 (ms)
Low Time 5,00007629510948 (ms)

Tek Run: 25.0kS/s Sample
C1 Freq 100.0048 Hz Low signal amplitude
C1 +Width 5.000ms Low signal amplitude
C1 −Width 5.000ms Low signal amplitude
C1 Period 10.000ms Low signal amplitude
Ch1 2.00 V M2.00ms Ch1 ƒ 2.12 V 30 Apr 2006 14:06:01

Frequency: 100 Hz
Duty cycle:
High Time 3,0087739375906 (ms)
Low Time 6,9912260624094 (ms)

Tek Run: 25.0kS/s Sample
C1 Freq 100.0052 Hz Low signal amplitude
C1 +Width 3.040ms Low signal amplitude
C1 −Width 6.960ms Low signal amplitude
C1 Period 10.000ms Low signal amplitude
Ch1 2.00 V M2.00ms Ch1 ƒ 2.12 V 30 Apr 2006 14:08:51

Frequency: 20 kHz
Duty cycle:
High Time 4,00015259021897 (us)
Low Time 45,999847409781 (us)

Tek Run: 5.00MS/s Sample
C1 Freq 20.00000kHz Low signal amplitude
C1 +Width 4.00µs Low signal amplitude
C1 −Width 46.00µs Low signal amplitude
C1 Period 50.00µs Low signal amplitude
Ch1 2.00 V M 10.0µs Ch1 ƒ 2.12 V 30 Apr 2006 14:10:38

Frequency: 100 Hz
Duty cycle:
High Time 50,9651331349667 (us)
Low Time 9,94903486686503 (ms)

Tek Stop: 100kS/s 942 Acqs
C1 Freq 100.000 Hz Low signal amplitude
C1 +Width 50µs Low resolution
C1 −Width 9.950ms Low signal amplitude
C1 Period 10.000ms Low signal amplitude
Ch1 2.00 V M 500µs Ch1 ƒ 1.60 V 30 Apr 2006 14:55:57

Figure 5.5 : Adjusting trig signal

5.4.2. Measuring latency times:

Embedded interrupt generator generates trig signal and save the time of it. Linux machine that the parallel port drivers installed on respond the trig signal with changing level of corresponding parallel port data pins. These response times are also saved by interrupt generator in microsecond resolutions and difference between trig time and respond time gives us unintrusively measurement of latency times. All these values are saved in a data structure. Embedded interrupt generator send one or more data structures in one TCP packet to measurement suite user interface program via TCP/IP socket connection.

5.5. Part 3 - Parallel Port Device Driver

5.5.1. Implementation of interrupt handling mechanisms

The Linux kernel top half interrupt handling mechanism is the irq handler that is registered with request_irq. The Linux kernel bottom half interrupt handling mechanisms are tasklets which base on softirqs, and workqueues. Workqueues may have a higher latency but that are allowed to sleep. In a real-time application, one or more of these mechanisms have to be used and response times of these mechanisms must be figured out. For this purpose, we write a parallel port device driver on target Linux Machine with using all of top and bottom handlers, because interrupt handling codes are part of device driver at Linux. The device driver flow diagram is shown in Figure 5.6 .

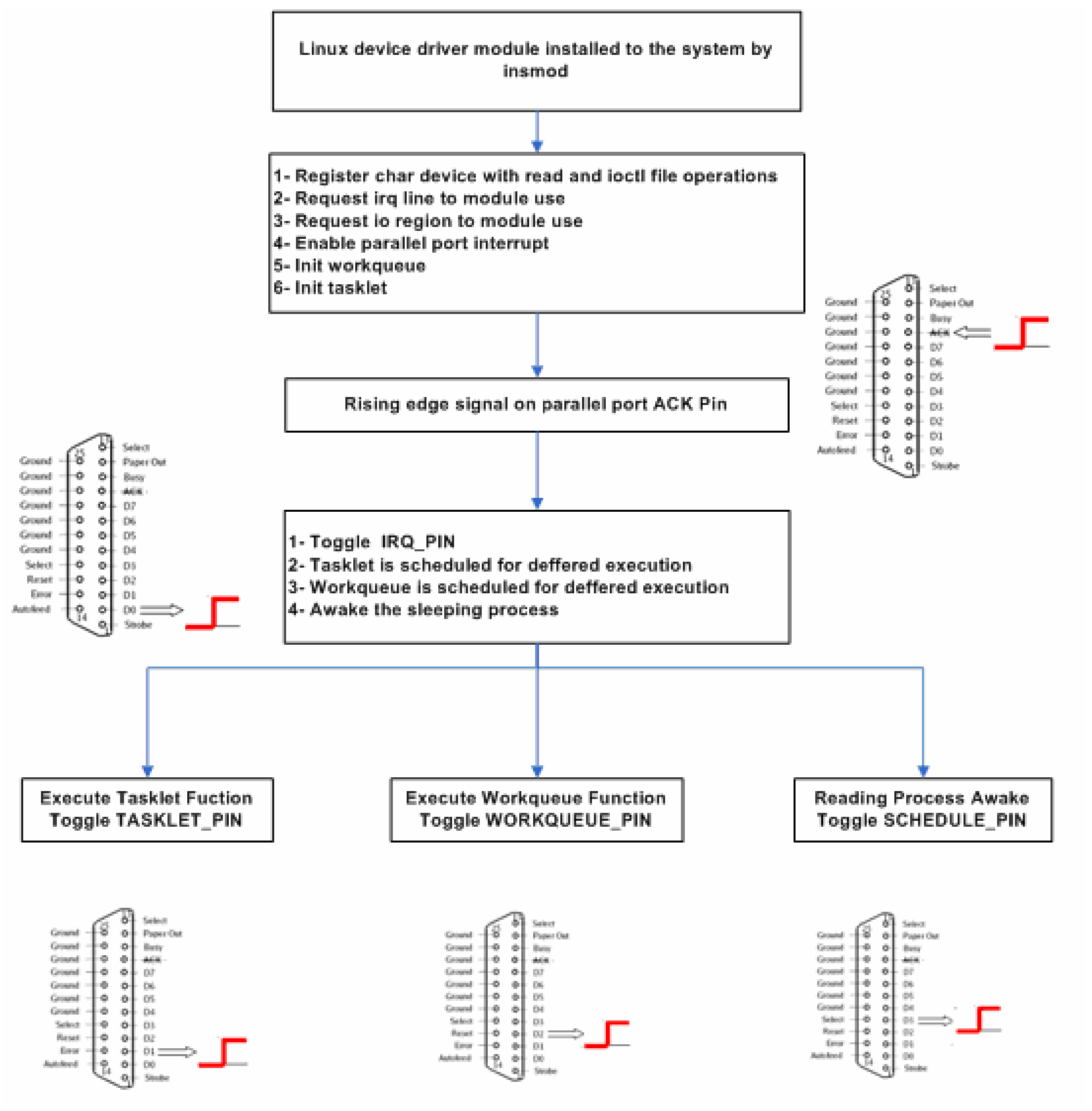

Linux device driver module installed to the system by insmod
1- Register char device with read and ioctl file operations
2- Request irq line to module use
3- Request io region to module use
4- Enable parallel port interrupt
5- Init workqueue
6- Init tasklet
Rising edge signal on parallel port ACK Pin
1- Toggle IRQ_PIN
2- Tasklet is scheduled for deffered execution
3- Workqueue is scheduled for deffered execution
4- Awake the sleeping process
Execute Tasklet Fuction
Toggle TASKLET_PIN
Execute Workqueue Function
Toggle WORKQUEUE_PIN
Reading Process Awake
Toggle SCHEDULE_PIN

Figure 5.6 : Flow chart of the device driver

5.5.2. Measuring scheduler latency method

To measure scheduler latency, we write read file operation for our parallel port device. When a thread issues the read file operation of the device, the read file operation handles the system call puts the thread on a wait queue. This puts the requesting thread to sleep so that it will not be eligible to run while waiting for the interrupt from our embedded interrupt generator. So at this point, the requesting thread is asleep and the kernel runs all threads except the thread that originated the read() requests. When embedded interrupt generator assert the interrupt line. This will cause an interrupt service routine to run which runs the device driver's interrupt handler. The driver's interrupt handler will remove the thread from the driver wait queue, making it ready to run. The read operation is shown in the following code fragment:

```
static ssize_t performx_read(struct file *file, char *u_buf, size_t count, loff_t *ppos)
{
        DECLARE_WAITQUEUE(wait, current);
        add_wait_queue(&performx_wait, &wait);
        do {
                __set_current_state(TASK_INTERRUPTIBLE);
                schedule();
        } while (1);
}
```

5.6. Part 4 - User Space Programs

5.6.1. Scheduler latency measurement user space program

Scheduler latency is another handicap for real-time systems. If some tasks may need sleep or blocking, these tasks cannot be handled within interrupt context. Basically, these tasks must be marked to schedule within interrupt context and executed deferred time within a scheduled thread. The task that marked to schedule starts execution after scheduler run and if the task has larger priority in ready queue. So, response time of the task is determined mainly by the scheduler latency. To measure scheduler latency, we develop a method starting from user space read call and finish with device driver. Code fragment of user space program as follows:

```
        while (1) {
                read( fptr, &dummy, sizeof(dummy) ;
                ioctl( fptr, XOR_SCHEDULE_PIN, dummy);
        }
```

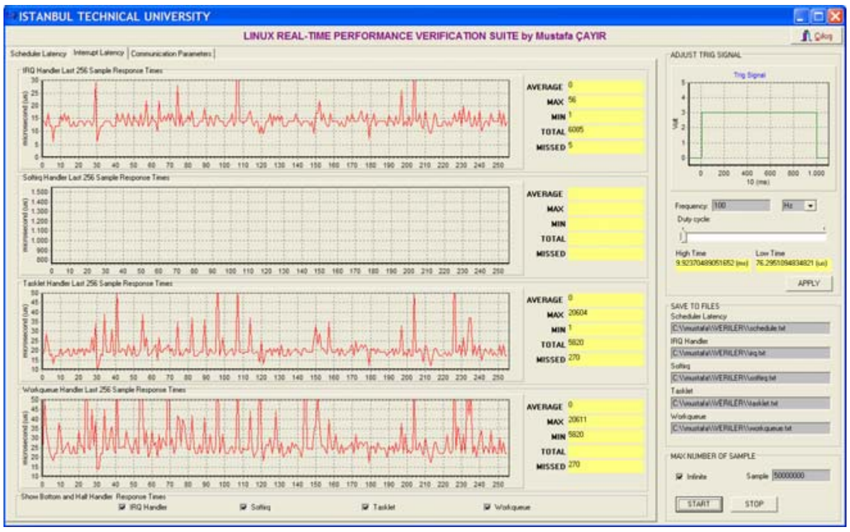
When the thread call request read system call, the device read file operation put the thread SLEEP. Only, the device driver’s interrupt handler makes the thread READY by removing the thread from the wait queue. As a consequence of completing interrupt service routine and removing the thread from the wait queue, the kernel will indicate that a scheduling pass needs to be made by setting the need_resched flag in the current task structure. When the kernel gets to a point where scheduling is allowed, it will note that need_resched is set and will call the function schedule(), which will determine what thread should run next. If the thread that issued the read() has a high enough priority, the kernel will context switch to it and when the kernel transitions back to user space the thread will run next operation ioctl() . The amount of time between when the ioctl() operation executed and time when the interrupt is asserted gives us time of scheduler latency. The ioctl() operation changes corresponding parallel port data pin logic. This makes possible to measure scheduler response time by embedded interrupt generator.

5.6.2. System load user space programs

Target Linux system must have a heavy system load to measure the worst case latencies. It is certain that test result is directly related to system load. Therefore, in our work, we load the target Linux system with repeated compilation of a Linux kernel, a file copy program from a network drive and media player program.

5.7. Measurements Results

The tests were run on a 2.00GHz Intel Pentium 4 system with 1GB RAM. Measurement user interface program snapshots are shown Figure 5.7 and Figure 5.8 .


ISTANBUL TECHNICAL UNIVERSITY
LINUX REAL-TIME PERFORMANCE VERIFICATION SUITE by Mustafa ÇAYIR
Çıkış
Schedule Latency
Interrupt Latency
Communication Parameters
IRQ Handler Last 256 Sample Response Times
Softirq Handler Last 256 Sample Response Times
Tasklet Handler Last 256 Sample Response Times
Workqueue Handler Last 256 Sample Response Times
microsecond (us)
ADJUST TRIG SIGNAL
Trig Signal
Volt
10 (ms)
APPLY
START
STOP

IRQ Handler: AVERAGE 0, MAX 56, MIN 1, TOTAL 6005, MISSED 5

Softirq Handler: AVERAGE, MAX, MIN, TOTAL, MISSED

Tasklet Handler: AVERAGE 0, MAX 20604, MIN 1, TOTAL 5820, MISSED 270

Workqueue Handler: AVERAGE 0, MAX 20611, MIN 5820, TOTAL, MISSED 270

Show Bottom and Half Handler Response Times: ☑ IRQ Handler ☑ Softirq ☑ Tasklet ☑ Workqueue

Frequency: 100 Hz

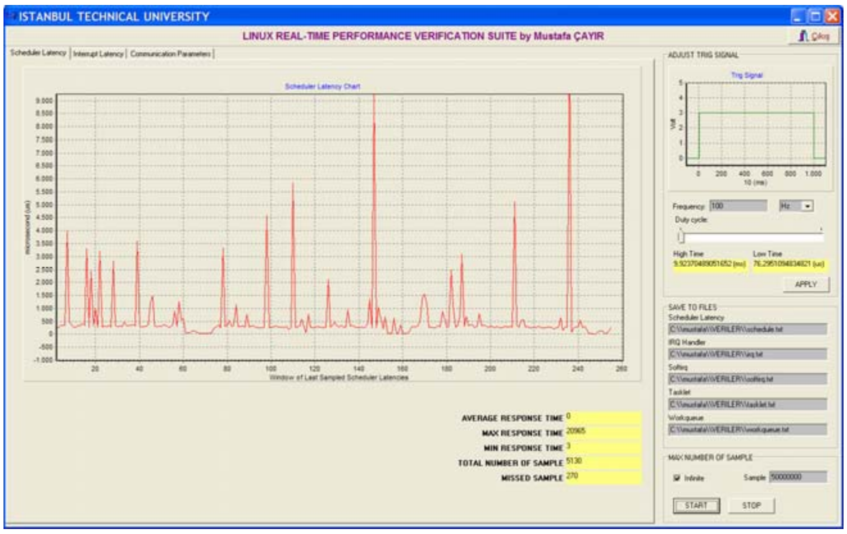
Duty cycle

High Time 9.92370489051652 (ms) Low Time 76.2951094834821 (us)

SAVE TO FILES
Scheduler Latency: C:\\mustafa\\VERILER\\schedule.txt
IRQ Handler: C:\\mustafa\\VERILER\\irq.txt
Softirq: C:\\mustafa\\VERILER\\softirq.txt
Tasklet: C:\\mustafa\\VERILER\\tasklet.txt
Workqueue: C:\\mustafa\\VERILER\\workqueue.txt

MAX NUMBER OF SAMPLE: ☑ Infinite Sample 50000000

Figure 5.7 : Snapshot of interrupt latencies measurements


ISTANBUL TECHNICAL UNIVERSITY

LINUX REAL-TIME PERFORMANCE VERIFICATION SUITE by Mustafa ÇAYIR

AVERAGE RESPONSE TIME 0
MAX RESPONSE TIME 20965
MIN RESPONSE TIME 3
TOTAL NUMBER OF SAMPLE 5130
MISSED SAMPLE 270

Frequency: 100 Hz
High Time 9.92370489051652 (ms)
Low Time 76.2951094834821 (us)

SAVE TO FILES
Scheduler Latency: C:\\mustafa\\VERILER\\schedule.txt
IRQ Handler: C:\\mustafa\\VERILER\\irq.txt
Softirq: C:\\mustafa\\VERILER\\softirq.txt
Tasklet: C:\\mustafa\\VERILER\\tasklet.txt
Workqueue: C:\\mustafa\\VERILER\\workqueue.txt

MAX NUMBER OF SAMPLE
Infinite [✓] Sample 50000000

Figure 5.8 : Snapshots of scheduler latencies measurements

IRQ handler response time values are given in Figure 5.9.

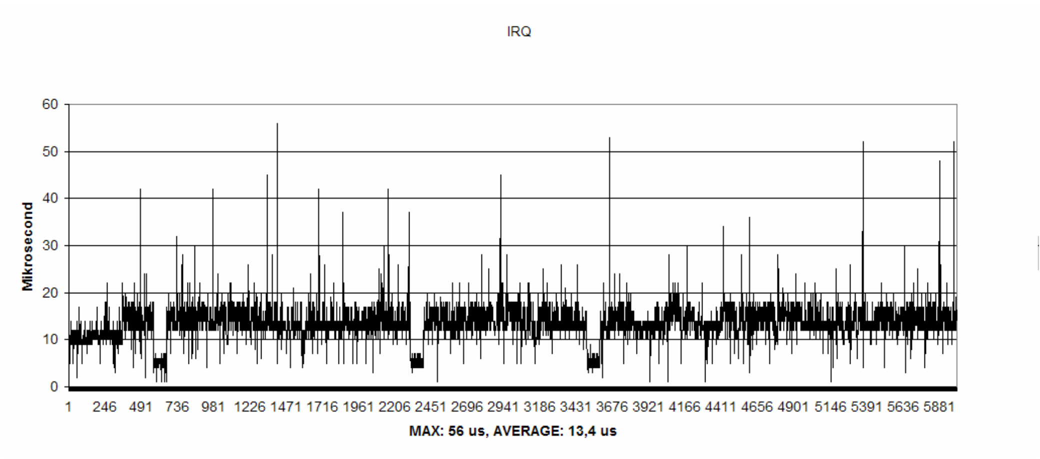


Figure 5.9 : IRQ measurements results

TASKLET response time values are given in Figure 5.10.

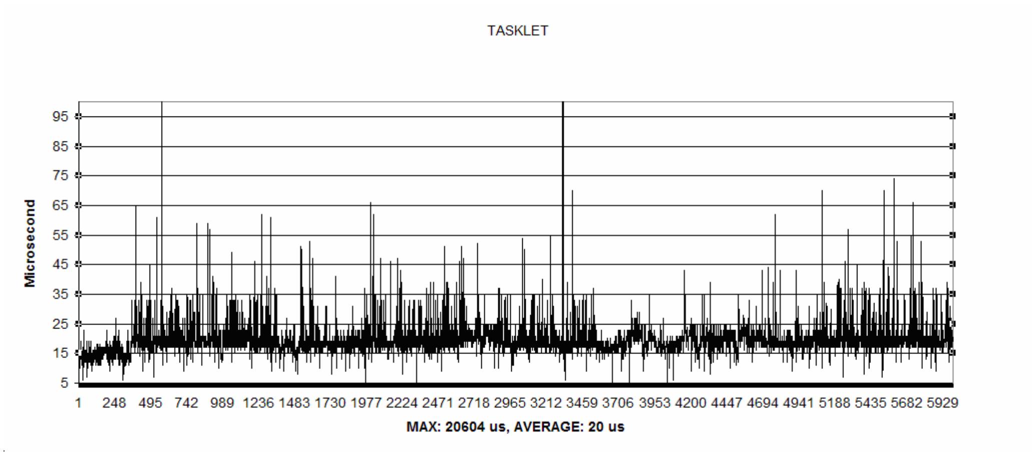


Figure 5.10 : TASKLETS measurements results

WORKQUEU response time values are given in Figure 5.11.

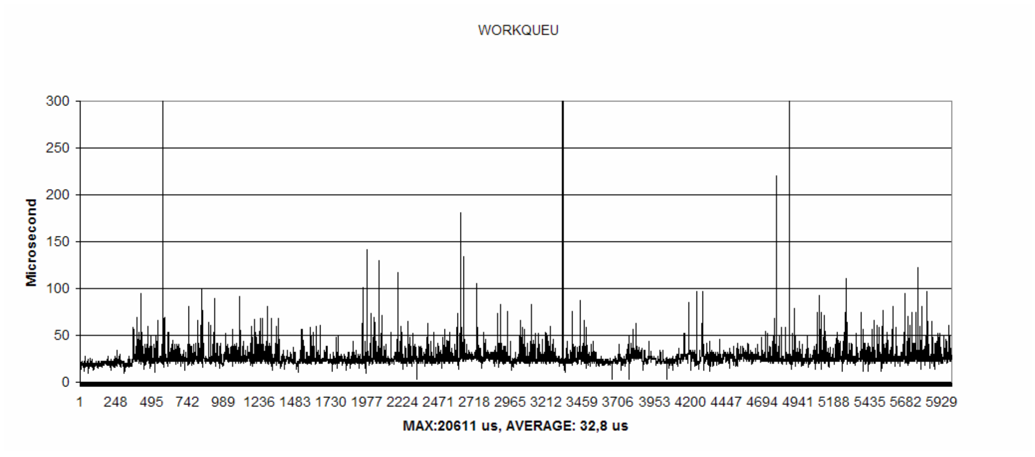


Figure 5.11 : WORKQUEUE measurements results

Scheduler latency values are given in Figure 5.13.

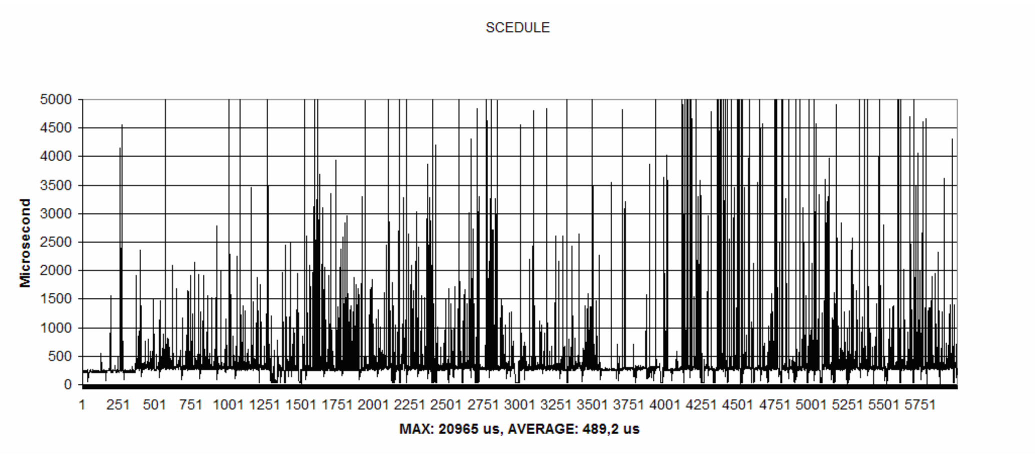


Figure 5.12 : Schedule measurements results

6. CONCLUSION

In this thesis, first of all, a method was developed to measure Linux kernel latencies unitrusively. The implementation of the method is build up with separate parts that are user interface program that runs on host machine, embedded interrupt generator to simulate real events and measure latencies microsecond resolution, Linux 2.6 kernel parallel port device that includes all types of bottom and top half interrupt handlers and Linux user space programs run on target Linux machine.

Linux kernel mechanisms that are influence on real-time performance was analyzed such as scheduler implementation and interrupt handling mechanisms.

In addition, all available approaches to make Linux real-time embedded operating systems were analyzed. Performance improvements and implementation of these methods were explained.

REFERENCES

[1] **Venture Development**, 2004. http://linuxdevices.com/news/NS2744182736.html

[2] **M. Timmerman, L. Pernell,** 2005. RTOS state of art. Dedicated system experts.

[3] **P.A.Laplante,** 1997. Real-Time Systems Design and Analysis: An Engineer's Handbook, Second edition, IEEE Press.

[4] **B. Mukherjee, K. Schwan, K. Ghosh,** 2002. A Survey of Real-Time Operating Systems Technical Report Georgia Tech. College of Computing.

[5] **J. Nilsson, D. Rytterlund,** 2000. Modular Scheduling in Real-Time Linux. Master's thesis, MdH, Mälardalen University.

[6] **Kevin Dankwardt,** 2002. ELJonline Real Time and Linux Part 1. www.linuxdevices.com/articles/AT5997007602.html

[7] **Cranes Software,** 2003. www.cranessoftware.com/press/articles/rtos.pdf

[8] **Karim Yaghmour**, 2003. Building Embedded Linux Systems. O'Reilly.

[9] **National Institute of Standards and Technology,** 2002. Introduction to Linux for real-time control. Aeolean Inc.

[10] **K. Engin, Ç. Mustafa,** 2006. GÖSİS projesi sistem yazılımları paketi literatur taraması sonuç raporu. TÜBİTAK MAM BTE.

[11] **Michael E. Anderson,** 2002. Selecting the Right RTOS, A Comparative Study. Embedded Systems Conference

[12] **Alessandro Rubini, Jonathan Corbet,** 2005. Linux Device Drivers 3rd. O'Reilly

[13] **SSV embedded systems**, 2000. Embedded Linux for DIL/NetPC DNP/1486-3V. SSV embedded systems.

[14] **T. Wiedemann**, 2005. How Fast Can Computer React. Chemnitz University of Technology, Seminar Paper

[15] **Intel Corporation,** 2005. Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture. Intel Corporation

[16] **Intel Corporation,** 2005. Intel Architecture Software Developer's Manual, Volume 3: system Programming Guide. Intel Corporation

[17] **Daniel P. Bovet, Marco Cesati,** 2005. Understanding The Linux Kernel 3rd. O'Reilly

[18] **R. Love,** 2005. Linux Kernel Development.

[19] **Josh Aas, 2005.** Understanding the Linux 2.6.8.1 CPU Scheduler. Silicon Graphics, Inc.

[20] **I. Ripoll, P. Pisa, L. Abeni, P. Gai, A. Lanusse, S. Saez, B. Privat,** 2002. WP1 – RTOS State of the Art Analysis: Deliverable D1.1 - RTOS Analysis. OCERA

[21] **Sp. Raja,** 2004. Linux for Real Time Requirements. Integrated SoftTech Solutions P Ltd

[22] **CE Linux Forum, 2000.** RTSpecDraft_R1

[23] **Abukar Mohamed,** 2002. Linux Low-Latency Patches and Alternative Schedulers

[24] **Clark Williams,** 2002. Linux Scheduler Latency. Red Hat, Inc.

[25] **DIAPM,** 2000. RTAI Programming Guide 1.0. The DIAPM RTAI homepage, http://www.rtai.org

[26] **R. Rajkumar, K. Juvva, A. Molano, S. Oikawa**, 2000. Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems.

[27] **Andrew Morton,** 2000. Realfeel Test Program. http://brain.mcmaster.ca/~hahn/realfeel.c

[28] **Opersys**, Inc., 2004. Linux Tracing Toolkit. http://www.opersys.com/LTT/

[29] **John Levon,** 2002. Profiling in Linux HOWTO

[30] **I. Molnar, A. Morton.** http://www.zipworld.com.au/~akpm/linux/#intlat.html

[31] **Andrew Morton,** 2000. Initlat Test Program.
http://www.zipworld.com.au/~akpm/linux/intlat.txt

[32] **I. Molnar, A. Morton.** http://www.zipworld.com.au/~akpm/linux/schedlat.html

[33] **Lmbench**, 2006. Tools for Performance Analysis.
http://www.bitmover.com/lmbench

[34] **R. Lehrbaum,** 2001. Adeos.
http://www.linuxdevices.com/articles/AT7788559732.html.

AUTOBIOGRAPHY

Mustafa Çayır was born in Sivas in 1981; he concluded his high school education in Sivas High School. He had graduated from the Electrical and Electronics Engineering Department of İstanbul Technical University at year 2003. He started his education for master degree at the Defense Technologies Programme of İstanbul Technical University. He is currently working as a researcher at Tübitak Marmara Research Center, Information Technologies Institute.