A Linux-based Real-Time Operating System

by

Michael Barabanov

Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in Computer Science

New Mexico Institute of Mining and Technology
Socorro, New Mexico

June 1, 1997

Abstract

This work describes the design, implementation, and possible applications of Real-Time Linux — a hard real-time version of the Linux operating system. In this system, a standard time-sharing OS and a real-time executive run on the same computer. Interrupt controller emulation is used to guarantee a low maximum interrupt latency independently of the base system. The use of a one-shot timer makes it possible to achieve a low task release jitter without compromising throughput. Lock-free FIFO buffers are employed for communication between real-time tasks and Linux processes. User-defined schedulers are allowed as are run-time changes in the scheduling policy.

The system is in active use for real-time data acquisition, control, and communications.

The software is free and can be obtained by FTP at `luz.cs.nmt.edu:/pub/rtlinux` or via the WWW at `http://luz.cs.nmt.edu/~rtlinux`.

Acknowledgements

I would like to express my deep appreciation to my advisor, Victor Yodaiken, for his support, encouragement, and friendship. His original ideas, comments and suggestions were invaluable. His help was essential in bringing this thesis to completion. I also wish to thank Victor for his patience in answering my numerous questions.

I thank other members of my committee: Dr. Lassez and Dr. Mazumdar, for their time and effort in reviewing my thesis. I found Dr. Lassez's comments on my presentation style to be immensely helpful. I am grateful to Dr. Mazumdar for encouraging me to perform more substantial testing of the system.

Thanks to Yuri G. Karpov, my academic advisor in Russia, for giving me an opportunity to study in the US and helping me deal with many important matters.

Last, but not least, I would like to thank Olga Tomina for her patience and understanding, and my parents, Alexander and Irina, for everything good they have done for me.

Table of Contents

List of Figures

List of Abbreviations

API Application Program Interface

CPU Central Processor Unit

DMA Direct Memory Access

EDF Earliest Deadline First

ELF Executable and Linkable Format

FIFO First-In-First-Out

FP Floating Point

GUI Graphical User Interface

I/O Input/Output

IPC Interprocess Communication

OS Operating System

PC Personal Computer

RAM Random-Access Memory

ROM Read-Only Memory

RT Real Time

TLB Translation Lookaside Buffer

Chapter 1

Introduction

1.1 Real-Time Systems

A real-time computer system can be defined as a system that performs its functions and responds to external, asynchronous events within a specified amount of time [4]. Most control and data acquisition applications, for example, fall into this category. A real-time operating system is an operating system capable of guaranteeing timing requirements of the processes under its control.

While time-sharing OS like UNIX strive to provide good *average* performance, for a real-time OS correct timing is the key feature. Throughput is of secondary concern.

There are *hard* and *soft* real-time systems [4]. Soft real-time systems are those in which timing requirements are statistically defined. An example can be a video conferencing system: it is desirable that frames are not skipped, but it is acceptable if a frame or two is occasionally missed. In a hard real-time system, the deadlines must be guaranteed. For example, if during a rocket engine test this engine begins to overheat, the shutdown procedure *must* be completed in time.

While it is possible (and even sometimes preferable) to implement real-time systems without any kind of operating system, doing so is usually not very convenient, nor is it easy. In this work we will only discuss real-time with respect to operating systems, focusing on UNIX-like OSs.

1.2 Problem Statement and Motivation

There are numerous real-time operating systems available today. What was missing, however, is an open, standard, supported, efficient, and inexpensive multitasking system with hard real-time capabilities. Many UNIX systems meet the first three requirements. Linux [1], a relatively recent free UNIX-like OS, originated by Linus Torvalds, features excellent stability, efficiency, source code availability, not restrictive license, and substantial user base. Source code availability is essential for verification of the system correctness, adaptation to specific problems, and mere bug fixing.

Linux can run on the most widely available computers: IBM PCs and compatible machines (most PC hardware is supported). Since many computer and engineering labs have computers of this type, it is possible for them to use Linux without any further investment.

Linux has all features of a modern UNIX system: several X Window System implementations, graphical user interface toolkits, networking, databases, programming languages, debuggers, and a variety of applications. It is embeddable and is able to perform efficiently using relatively small amounts of RAM and other computer resources (the Sunrayce project software that I will describe in Chapter 4 utilizes these properties of Linux). In short, Linux has the potential to make an excellent development platform for a wide variety of applications, including those involving real-time processing.

However, Linux has several problems preventing it from being used as a hard real-time OS, most notably the fact that interrupts are often disabled during the course of execution of the kernel. Other problems include time-sharing scheduling, virtual memory system timing unpredictability, and lack of high-granularity timers.

It turns out that using software interrupts [23], together with several other techniques, it is nevertheless possible to modify Linux so as to overcome these problems. The idea to use software interrupts so that a general-purpose operating system could coexist with a hard

[1] `http://www.linux.org/`

real-time system is due to Victor Yodaiken (personal communications). This thesis details how this idea and several others were applied to build a hard real-time version of Linux.

1.3 Organization

The thesis is structured as follows. Chapter 2 describes related research in the area of real-time UNIX systems. Chapter 3 details the design and implementation of Real-Time Linux. Chapter 4 contains a discussion of the application model of Real-Time Linux. I also describe several existing applications. The measurements results of Real-Time Linux performance can be found in Chapter 5. Finally, Chapter 6 summarizes the results of this work and outlines possible directions of future research.

Chapter 2

Related Work

Despite the recent proliferation of various operating systems, UNIX and compatibles are still standard in the industry and academia. Some non-UNIX systems, for example Windows NT, have limited compliance with POSIX.1003 [10], a standard clearly based on UNIX. The success of this system is partly due to the fact that it is open, mature, and standard. With the advent of POSIX-1003.1b real-time extensions, UNIX has a chance to become the most widespread platform for real-time processing.

Because of these reasons, in this chapter I will focus on real-time systems that have some relation to UNIX. I will describe problems with real-time computing on UNIX, and how these problems have been addressed in several systems.

2.1 Time-Sharing Nature of UNIX

UNIX was originally designed as a time-sharing system [20]. Most modern implementations retain this nature. They strive to optimize *average* performance. This goal is often at odds with low latency and high predictability requirements of real-time systems. To illustrate this let us consider a program to play a note through the speaker (Figure 2.1).

The speaker driver is assumed to have only two states: on and off. At first sight this program might seem like a legitimate way to make the speaker reproduce the sound

```
#define DELAY 10000
main()
{
        int i;
        while (1) {
                for (i = 0; i < DELAY; i++);
                speaker_on();
                for (i = 0; i < DELAY; i++);
                speaker_off();
        }
}
```

Figure 2.1: A Naive Sound-Generating Program

corresponding to a square wave with a given period. However, when run as a standard UNIX process, it will not work properly.

I ran this program under the Linux operating system on a PC with a 120 MHz Pentium processor. When there was no other noticeable activity in the system, the speaker was producing a somewhat steady tone. The tone was not completely steady, however. Every now and again clicks could be heard. Each key press or a mouse move caused the tone to momentarily change. In the presence of disk accesses or CPU-intensive processes the sound was badly distorted. Finally, start of a program with a large binary image, such as an X Window System server, caused intervals of silence lasting for up to half a second. If this program was to control a stepper motor instead of a speaker, it would not be able to make it run steadily.

Several design and implementation elements of Linux, and UNIX in general, contribute to this behavior. The main factors are time-sharing scheduling, low timer resolution, kernel non-preemptability, disabling interrupts, and virtual memory. Let us consider these factors in detail.

Scheduling is a set of policies and mechanisms built into the operating system that govern the order in which the work to be done by a computer system is completed [19].

Most UNIX operating systems, Linux in particular, have schedulers that are intended to balance response time and throughput and to ensure fair distribution of CPU time among processes [15]. Each process has a varying priority that depends on the amount of CPU time the process has spent, input/output intensity, and other factors.

UNIX systems typically schedule CPU time using fixed *time slices*. Initially a process is given a high priority. If during its time slice a process gives up the CPU, the priority of this process remains the same, or becomes higher. On the other hand, if a process uses up its time slice completely, its priority is lowered. This policy favors interactive programs, e. g. editors, since these programs spend most of their execution time waiting for I/O to complete. While convenient for a user at a terminal, such scheduling is next to useless for real-time processing because the execution of any process depends in a complex and unpredictable fashion on system load and the behavior of other processes.

Another problem is the timer resolution. Historically only 1 second resolution alarm signals and the `sleep()` system call were provided to user processes. Such crude timing is not sufficient for most kinds of real-time processing. Modern versions provide means to specify intervals with higher precision, however, internal clock implementations often limit the actual timing accuracy. I will talk more about the issue of timer resolution in Chapter 3.

In most UNIX systems processes running in kernel mode can not be preempted [15]. In other words, once a process has entered the kernel it will run until the system call is complete or until it blocks. If a higher-priority real-time process becomes ready in the meanwhile, it will have to wait. This design simplifies the task of kernel developers because for the most part the kernel does not have to be reentrant. However, a system call can take a long time to complete, and for a real-time process long delays can be unacceptable.

Closely related to the kernel non-preemptability is the problem of synchronization. To protect data that may be accessed asynchronously, for example in an interrupt handler, system designers often choose to disable interrupts during critical sections. This is a simple technique and is often more efficient than using semaphores or spinlocks. However, disabling interrupts compromises the system's ability to promptly respond to external events. This method also does not solve the synchronization problem on multiprocessor architectures.

Most UNIX systems use a virtual memory system with paging [15] [1]. Virtual memory makes it possible to simultaneously run programs with total size exceeding the available RAM by keeping only the working set of each program in memory. This model works well for time-sharing systems. For real-time systems, however, virtual memory introduces an intolerable level of unpredictability.

Having considered these factors it is clear that traditional UNIX systems are hardly capable of real-time processing. Some radical changes are required.

2.2 Existing Real-Time Operating Systems

Let us consider how designers of several operating systems have addressed (or not addressed) the problems described in the previous section.

The simplest solutions stop on changing the standard time-sharing scheduler. An example of such system is presented in [26]. The MINIX OS round-robin scheduler was replaced with a priority-based scheduler. Since neither paging nor swapping is used in MINIX, if the response time requirements are not very demanding, this approach can be acceptable.

Several systems adopt POSIX.1b-1993 standard for real-time features in UNIX [5] [10]. The standard defines prioritized scheduling, locking of user memory pages in memory, real-time signals, improved IPC and timers, and a number of other features. Compliance with this standard makes UNIX systems much more appropriate for real-time applications.

Linux partially supports the POSIX.1b standard [14]. As of May 1, 1997, functions for the control of the scheduler and memory locking are fully implemented in Linux, and timers are partially implemented. The problems of kernel non-preemptability, low timer resolution, and high interrupt latency remain unresolved. Thus, POSIX.1b compatibility only permits certain kinds of soft real-time processing in Linux.

Another example of a POSIX.1b-conforming system is QNX [8]. The QNX architecture is that of a microkernel [24]. The kernel only implements four services: process scheduling, interprocess communication, low-level network communication, and interrupt dispatching. All other services, such as device drivers and filesystems, are implemented as cooperating user processes. As a result the kernel is very small (about 7 kilobytes of code) and fast.

QNX also complies with POSIX 1003.1 (program interface) and POSIX 1003.2 (shell and utilities) [11] standards. This allows developers familiar with UNIX to be comfortable with the system [9]. QNX provides standard UNIX features: compilers, debuggers, X-Windows, and TCP/IP.

The microkernel approach has a number of advantages over traditional monolithic design. Debugging user processes is easier than debugging kernel components. If user processes are executed in separate address spaces (as in QNX), memory management errors in different modules are isolated. Drivers can easily take advantage of multithreading if it is provided by the underlying microkernel. Another advantage is scalability. For example, a QNX system can be scaled down to 100K to fit in the ROM, or expanded to a full-featured multi-machine development environment. Porting and maintenance of a microkernel-based system is also easier. In short, many of the problems of monolithic kernels are much less severe in microkernel systems.

For real-time processing microkernels offer light-weight processes, fast context switches, and IPC. A real-time user process can preempt a device driver, which is not the case in monolithic kernels. Another advantage is that since microkernels are very small, it is easier to calculate worst-case timing parameters, such as interrupt latency.

A weak point of many microkernels is performance. Microkernel architecture places heavy load on interprocess communication and context switching. Microkernels only provide simple services directly. Therefore, more system calls have to be performed in a microkernel system than in a monolithic one to accomplish the same task. Although some researchers argue that protection level changes, context switches, and message passing can be implemented very efficiently [16], it is mostly for performance reasons that monolithic kernels are still prospering.

One example of a monolithic system is VxWorks [27]. VxWorks is a proprietary real-time operating system geared towards host/target approach. A UNIX host is used for software development and for running non-real-time parts of an applications. The VxWorks kernel called *wind* runs real-time tasks on the target computer. The machines communicate using TCP/IP networking.

While VxWorks is not compatible with UNIX, it provides some of the POSIX interface functions, most notably those from POSIX.1b real-time extensions. Most of the VxWorks API is, however, proprietary.

In VxWorks, the kernel and tasks run in one address space. This allows task switching to be very fast and eliminates the need for system call traps. A run-time linker allows dynamic loading of both tasks and system modules. This feature makes for scalability. An interactive shell with C-like syntax can be used to examine and modify variables, evaluate expressions, call functions, and perform simple debugging. These features encourage experimentation and make development somewhat easier. They also make the system more fragile as errors in one module can easily affect others.

The REAL/IX operating system from MODCOMP [4] is monolithic. This is a full-featured UNIX system originated from UNIX System V and made capable of real-time processing. The kernel is fully preemptive. This is achieved by using kernel semaphores to provide exclusive access to resources rather than using traditional explicit `sleep/wakeup` functions and disabling interrupts. The use of semaphores instead of disabling interrupts lowers the interrupt latency and makes porting to multiprocessor machines simpler.

REAL/IX is a POSIX.1003-conforming OS. This feature makes for easy porting of UNIX applications. Besides real-time scheduling, real-time capabilities of REAL/IX include pre-allocating memory and file space, synchronous and asynchronous I/O, enhanced IPC and timers, and connected interrupts. The last feature allows user processes to handle interrupts.

There is currently a trend to use Windows NT for real-time processing. The main reason is compatibility with previous Windows versions, and therefore, a multitude of available and popular applications. There is also a desire to use one OS for everything: office work, serving, and real-time control. The use of one OS reduces costs of personnel training. Programmers for the Win32 API are widely available. Microsoft's marketing may also be a factor.

As pointed out in [25], the stock Windows NT kernel is incapable of hard real-time processing: the Win32 API is not designed for real-time, interrupt notification can be delayed for an unpredictable amount of time, memory preallocation is problematic, and for embedded systems the considerable memory footprint of Windows NT can be a problem.

Several companies provide solutions for some of these problems. The system provided by LP Elektronik GmbH[1] uses non-maskable interrupts (NMI) generated by a special hardware device that contains an interrupt controller and a timer. This approach guarantees timing independence of Windows since NMI interrupt requests are never disabled. This is used to run a VxWorks port (LP-VxWin) on the same computer. The two systems communicate using TCP/IP.

The INtime product of RadiSys[2] modifies NT's Hardware Abstraction Level to trap Windows' attempts to disable interrupts or reset the clock. This is an approach similar to that taken in Real-Time Linux. At the time of this writing the system is in beta state.

The designers of QNX chose to implement the Win32 API on top of their POSIX-compliant OS [7]. This method has an advantage of using one API for both real-time and non-real-time applications. The memory usage is smaller, which is important for embedded systems. However, there is no binary compatibility.

[1] `http://www.lp-elektronik.com/`
[2] `http://www.radisys.com/products/rtos/nt_prod.html`

Chapter 3

Design and Implementation

3.1 Interrupt Emulation

One of the problems with doing hard real-time on a standard Linux system is the fact that the kernel uses disabling interrupts as a means of synchronization. Promiscuous use of disabling and enabling interrupts (`cli` and `sti` machine instructions in i486 processors) inflicts unpredictable interrupt dispatch latency. The Linux kernel is monolithic and large. There are no protecting boundaries between parts that provide different services. Many people all over the world are involved in writing the Linux kernel. This makes it very hard to bound the time spent with interrupts disabled. Moreover, even if such bound is once found, it is possible that it will become incorrect when the new version of the kernel comes out. And even if we could cope with that, the bound could be just too high for our needs.

In Real-Time Linux, this problem is solved by putting a layer of emulation software between the Linux kernel and the interrupt controller hardware, a technique similar to that described in [23], but used for a different purpose. In the Linux source code all occurrences of `cli`, `sti`, and `iret` instructions (`iret` means return from interrupt) are replaced with emulating macros: `S_CLI`, `S_STI` and `S_IRET`. All hardware interrupts are caught by the emulator. The idea of emulating `cli` and `sti` to achieve real-time performance is due to Victor Yodaiken.

```
S_CLI:  movl $0, SFIF

S_STI:  sti
        pushfl
        pushl $KERNEL_CS
        pushl $1f
        S_IRET
1:
```

Figure 3.1: Soft CLI and STI

Where disabling interrupts would normally take place, a variable in the emulator is reset instead. Whenever an interrupt happens, the emulator checks that variable. If it is set (Linux has interrupts enabled), the Linux interrupt handler is invoked immediately. If, on the other hand, Linux interrupts are disabled, the handler is not invoked. Instead, a bit is set in the variable that holds the information about all pending interrupts. When Linux re-enables interrupts, the handlers of all pending interrupts are executed. We will call such simulated interrupts *soft interrupts*.

Since Linux has no direct control over the interrupt controller, it does not influence processing of real-time interrupts that do not pass through the emulator.

The macros for `S_CLI` and `S_STI` are shown in Figure 3.1. The code uses the GNU assembler conventions. The `S_CLI` macro simply resets the variable that holds the Linux interrupt state. The `S_STI` macro sets up the stack as if an interrupt is being handled, and then uses `S_IRET` macro to emulate the return. This works because `S_IRET` enables soft interrupts just as the hardware `iret` enables real ones.

The `S_IRET` macro (Figure 3.2) is the most interesting of the three. It starts with saving some scratch registers and initializing the data segment register to point to the kernel. The latter is necessary to access global variables. Then the bitmask representing all unmasked pending interrupts is scanned for a set bit. If no pending interrupt was found, the interrupt state variable is set, and a hard return from interrupt is performed. If an interrupt was found, a jump is made to the Linux handler. The handler's `S_IRET`, in turn, will jump to the next pending interrupt handler, and so on, until no interrupts are pending.

```
S_IRET: push %ds
        pushl %eax
        pushl %edx
        movl $KERNEL_DS ,%edx
        mov %dx,%ds
        cli
        movl SFREQ, %edx
        andl SFMASK, %edx
        bsfl %edx, %eax
        jz 1f
        S_CLI
        sti
        jmp  SFIDT(,%eax,4)
1:      movl $1, SFIF
        popl %edx
        popl %eax
        pop %ds
        iret
```

Figure 3.2: Soft IRET

Scanning and decision taking are done atomically — otherwise, if a new interrupt occurs between them, and the scan has not found any pending interrupts, the invocation of the new interrupt handler will be delayed until the next `S_STI` or `S_IRET`.

I used chained jumps instead of invoking Linux handlers using subroutine calls because the latter method would not fully emulate direct interrupt handling. Linux handlers examine the stack to find out whether it was the user or the kernel code that was interrupted, and make decisions based on it. Therefore, it is important to preserve the stack state.

3.2 Real-Time Tasks

Real-time tasks are user-defined programs that execute according to a specified schedule under the control of the kernel.

The initial design was to give each real-time task its own address spaces to provide memory protection. This was done by utilizing Intel 80x86 processors' built-in paging mechanism [13]. On each context switch the page directory base register was changed to point to the page directory of the new task. For security reasons, tasks were executed with the lowest priority level. Light-weight system calls were used for communication between the real-time kernel and processes.

Each real-time task was presented to the system as an object module in the Executable and Linkable Format (ELF) via a special Linux system call. A primitive loader brought the code and data into the memory, allocated and initialized a task structure, and started the `main()` function of the task. The task informed the kernel about its starting time and period, and relinquished the CPU. From the starting time on the task was scheduled with the specified period.

The above scheme works, but the system performance is not optimal. One reason for performance problems is that caches on 486 CPUs are virtual. Whenever the page directory base register is changed, the translation lookaside buffer (TLB) has to be invalidated. Since real-time context switches are frequent, TLB invalidations inflict a severe performance decrease.

Another source of overhead is in system calls. Protection level changes on i486 CPUs are expensive. A trap to a more privileged level, for example, takes as long as 71 cycles to execute, while most other instructions take less than 10 cycles.

One way to improve performance is to run all RT-tasks in one address space. By using the kernel address space, we also eliminate the overhead of protection level changes. Linux has a useful feature in this regard: loadable kernel modules. Kernel modules are object files that can be dynamically loaded into the kernel address space and linked with the kernel code. Each module defines two routines: `init_module()` and `cleanup_module()`. The former is called when the module is loaded in the kernel, the latter — when the module is removed. This provides an easy means to manipulate available drivers and filesystems in Linux.

Loadable kernel modules are used in the current version of Real-Time Linux to dynamically create real-time tasks. This approach is clearly more fragile: a bug in a real-time task can wipe out the whole system. The use of the C language aggravates this problem.

Equivalence of arrays and pointers, type casts make it all too easy to write programs with memory referencing bugs. On the other hand, since real-time tasks often control expensive peripheral devices, it is reasonable to use the same level of caution as when programming an OS kernel.

Running tasks in the kernel address space has several advantages. Besides eliminating frequent TLB invalidation and protection level changes mentioned above, the approach allows us to refer to functions and objects by names rather than descriptors. For example, real-time tasks are represented as C structs. Each task can be given an arbitrary C identifier that can be used in other tasks. Dynamic linking performed during module loading resolves symbols to addresses, so the access is very efficient.

Task switching is also easier if all tasks run in one address space. Real-Time Linux performs task-switching in software because hardware switches are slow on i486 CPUs. A context switch consists of pushing all integer registers on the stack and changing the stack pointer to point to the new task. Tasks with floating point context are also supported.

The programming interface for real-time tasks can be found in Appendix A.

3.3 Scheduling

The main task of a real-time scheduler is to satisfy timing requirements of tasks. There are many ways to express timing constraints and many scheduling policies [2]. No single policy is appropriate for all applications.

In most real-time systems, the scheduler is a large, complex piece of code that can not be extended in any way. The user can only modify the behavior of the scheduler by adjusting parameters, which may not be enough. The generic scheduler code is often slow.

In contrast, Real-Time Linux allows users to write their own schedulers. They can be implemented as loadable kernel modules. This makes it possible to easily experiment with different scheduling policies and algorithms and find the ones that best suit the specific application. Schedulers can use the interval timer facility described in the next section.

Two schedulers have been implemented so far. One of them is a priority-based preemptive scheduler. The scheduling policy is as follows. Each task is assigned a unique priority. If there are several tasks that are ready to execute, the task with the highest priority is executed. Whenever a task becomes ready it will immediately preempt the currently executing task if the current task has a lower priority. Each task is supposed to relinquish the CPU voluntarily.

The scheduler directly supports periodic tasks. The period and the offset (the starting time) is specified for each of them. An interrupt-driven (sporadic) task can be implemented by defining an interrupt handler that wakes up the needed task.

For periodic tasks with deadlines equal to periods a natural way to assign priorities is given by the *rate monotonic scheduling algorithm* [17]. According to this algorithm, tasks with shorter periods get higher priorities. A set of n independent periodic tasks scheduled by the rate monotonic algorithm is guaranteed to meet all deadlines if

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

where C_i is the worst-case execution time of task i, and T_i is the period of task i. Sporadic tasks can often be treated as periodic ones for priority assignment [22].

The scheduler treats Linux as the lowest priority real-time task. Thus, Linux only runs when the real-time system has nothing to do. To this end, on switching from Linux to a real-time task, soft interrupt state (see Section 3.1) is remembered, and soft interrupts are disabled. When switching back, soft interrupt state is restored.

The other scheduler was implemented by Ismael Ripoll[1]. It uses the Earliest Deadline First (EDF) algorithm [17]. In this algorithm tasks do not have static priorities. Rather, the task with closest deadline is always chosen to execute.

[1]`http://bernia.disca.upv.es/~iripoll`

3.4 Timing

Precise timing is necessary for the correct operation of the scheduler. Execution schedules often require task switching at specific moments of time. Timing inaccuracies cause deviations from the planned schedule, resulting in so-called *task release jitter* [22]. In most applications task release jitter has an undesirable effect. It is important to minimize it.

One reason for low timer resolution typically found in operating systems is the use of periodic clock interrupts. System designers have to trade off the amount of time spent in handling clock interrupts with timer resolution [15]. Real-time systems sometimes require timer precision that is impossible to get with any reasonable performance using periodic clocks.

Linux is no exception to this rule. On IBM PC compatibles it programs the hardware timer to interrupt at a rate of about 100 Hz. Thus, tasks can be released with only 10 milliseconds precision. Most commercial real-time OS such as VxWorks and REAL/IX also use periodic clock interrupts, although some of them allow the user to change interrupt frequency [27].

In Real-Time Linux, I avoid this tradeoff by using a programmable interval timer to interrupt the CPU only when needed. Specifically, I put the Intel 8354 timer chip present in some form in all IBM PC compatible computers into the interrupt-on-terminal-count mode. Using this mode, an interrupt can be scheduled with approximately 1 microsecond precision. In this scheme the overhead of interrupt processing is minimal while the timer resolution is high.

To keep track of the global time, all intervals between interrupts are summed up together. Although on i486-based machines I had to use some clever tricks for this to work correctly, most other modern hardware platforms provide a software-readable global time counter.

The timer interface allows the scheduler to obtain the current time and to register functions to be called at particular moments.

Periodic interrupts are simulated for Linux. With soft interrupts it is particularly easy:

to imitate an interrupt request, a bit in the pending interrupts mask is set. On the next soft return from interrupt, or soft `sti`, the handler will be invoked.

The interval timer use in Real-Time Linux has its share of problems. Reprogramming the 8354 timer on PCs takes a long time because the timer is not on the processor chip. Fortunately, most modern CPUs, e. g., Pentiums [12], have timers on-chip in addition to the 8354.

3.5 Interprocess Communication

Since the Linux kernel can be preempted by a real-time task at any moment, no Linux routine can safely be called from real-time tasks. However, some communication mechanism must be present. Simple FIFO buffers are used in RT-Linux for moving information between Linux processes or the Linux kernel and real-time processes. We will call this buffers real-time FIFOs to distinguish them from the UNIX IPC facility by the same name.

RT-FIFO buffers are allocated in the kernel address space. They are referred to by integer numbers. There is a static limit on the number of RT-FIFOs that can be changed during kernel recompilation.

The real-time task interface to RT-FIFOs includes creation, destruction, reading and writing functions (Appendix A). Reads and writes are atomic and do not block. Non-blocking avoids the *priority inversion problem* [21].

Linux user processes, on the other hand, see RT-FIFOs as ordinary character devices. Unlike the special system call interface used in a previous design, the character device interface gives the users full power of UNIX API for communication with real-time tasks.

Chapter 4

Applications

4.1 Application Structure

One of the assumptions made during the design of Real-Time Linux is that each application should be split into real-time and non-real-time parts. I will call the latter the user part since it executes in the user space. The real-time part is as simple as possible. It only includes the code that is directly time-critical. Low-level communication with hardware often belongs in the real-time part since most hardware imposes timing constraints on the program. The user part, on the other hand, implements most of the data processing, including distributing and archiving of data and user interfaces. The two parts communicate using data buffers.

Figure 4.1 shows the data flow in a typical real-time application according to this model.

4.2 PC Speaker Driver

Let us consider an improved version of the sound-generating program from Chapter 2. This version will implement a primitive `/dev/audio`-type interface to the internal PC speaker rather than simply playing one note.

Note that there already exists a Linux driver for the PC speaker[1]. It has much more features than the one described here. However, it suffers from the high interrupt latency

[1]`ftp://ftp.informatik.hu-berlin.de/pub/os/linux/hu-sound/`

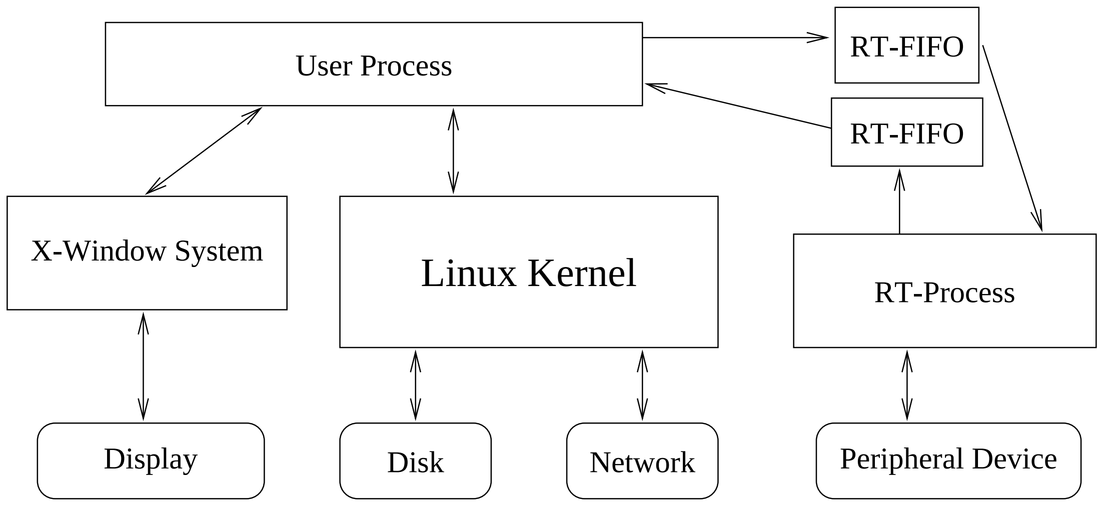

Figure 4.1: Data Flow in an Application

problem described in Chapter 2. The driver author notes, for instance, that the driver only plays well when there are no disk accesses. Our driver is free of this problem.

The `/dev/audio` device is a character one. In its simplest form it accepts a stream of logarithmically encoded (*ulaw*-encoded) sound samples, and plays the sound. The samples are usually one byte long, and the most common sampling rate is 8000 Hz. On some machines the hardware buffers the data sent to the speaker. This allows non-real-time systems to play the sound smoothly. On a PC, however, the internal speaker can only be controlled by toggling a bit in one of the I/O ports. This imposes tight timing constraints on the program.

The speaker driver code can be found in Appendix B. In the initialization routine the driver creates an RT-FIFO, and installs a handler for interrupts from the CMOS clock. The clock is then programmed to generate interrupts at 8192 Hz rate, which is close enough to 8 kHz. In the handler, an attempt is made to read a sound sample from the RT-FIFO. If successful, the 8-bit ulaw-encoded sample is reduced to 1 bit, and the speaker is turned on or off accordingly.

The driver can be used from the shell command line as follows:

```
cat linux.au >/dev/rtf0
```

where `/dev/rtf0` refers to the real-time FIFO created by the driver.

The driver does not use a real-time scheduler — mainly for efficiency reasons. Given the high frequency of the task, scheduling and reprogramming of the timer would incur overheads unacceptable on slow machines.

4.3 Real World Applications

In this section I will describe several practical applications of Real-Time Linux.

The first one is that of Harald Stauss[2] from the department of physiology at the Humboldt University in Berlin, Germany. He has implemented a system for the recording and display of hemodynamic measurements in rats. It uses an analog-to-digital converter card to acquire the signals from sensors. The card based on the MAX192BCPP chip has eight 12-bit channels that are multiplexed to one serial line connected to the serial port of the computer.

The system runs under Real-Time Linux and consists of a real-time task and a user process. The real-time task polls the card and passes the received data through an RT-FIFO to the user process. The process records the data to a file and at the same time displays it graphically in a Motif drawing widget. Mr. Stauss reports that the system is able to reliably acquire data with the total sampling rate of 3000 Hz on a i486/33MHz-based machine. On the same computer, Labtech Notebook for DOS, a commercial data acquisition program, could only provide sampling rates of less than 400 Hz.

Bill Crum at New Mexico Tech has developed an embedded control and monitoring software for the Tech Sunrayce car [3]. The system specification required data collection from 70 sensors with total sampling rate of 80 Hz. The needed response times ranged from 200 to 300 milliseconds. These requirements were satisfied by using one real-time task executing with 0.0125 seconds period on a 20 MHz i386–based computer running Real-Time Linux.

To facilitate the debugging of the system, Bill Crum has written a set of programs

[2] `h0651bkr@rz.hu-berlin.de`

that he describes as an "engineering workbench". The programs simulate tools commonly found in electrical engineering laboratories: an oscilloscope, a logic analyzer, and a signal generator. The complex uses a data acquisition card that provides analog and digital I/O. Real-time tasks are used to communicate with the card, and Linux processes implement graphical interfaces using the Qt widget set.

There are several other applications. A task running under Real-Time Linux is used for real-time communication with the PHANToM, a force feedback device[3]. This is a part of a system that creates virtual worlds. The system allows users to navigate through these worlds, to feel and manipulate objects. Dan Samber[4] from Mount Sinai Medical School in New York City uses Real-Time Linux to reliably communicate with a patient monitor over a serial line for recording and display of physiological parameters.

[3] `http://tesla.braintools.org/phantom-dev.html`

[4] `dan@camelot.mssm.edu`

Chapter 5

Experimental Results

In order to measure the performance of Real-Time Linux, I have conducted several experiments. The experiments were performed on two IBM PC compatible computers running Linux version 2.0.29 and Real-Time Linux version 0.5a. The results are summarized in Table 5.1.

To measure the maximum interrupt latency, an additional machine running Real-Time Linux (Machine 1) was used to send interrupt requests to the machine being tested (Machine 2) and to measure the response time of the latter (Figure 5.1).

The `D0` output of the parallel port of Machine 1 is connected to the `ACK` parallel port input of Machine 2. When the `ACK` input goes from a logic one to logic zero, an interrupt request is sent to the processor. To provide feedback, the `D0` output of Machine 2 is connected to a parallel port input (`PE`) on Machine 1.

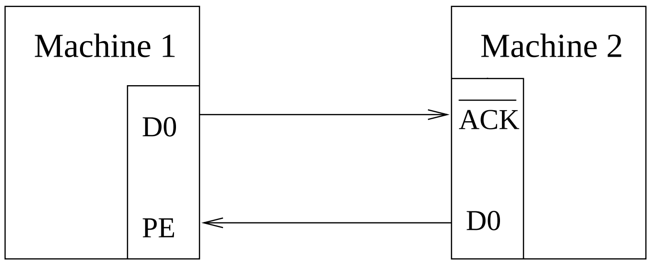

Figure 5.1: Measuring Interrupt Latency

System	Interrupt Latency (μs)	Scheduling precision (μs)
Machine A, Linux	220	—
Machine A, RT-Linux	84	155
Machine B, Linux	2494	—
Machine B, RT-Linux	34	64

Machine A: Intel 80486 33 MHz, ISA bus, 16 MB of RAM, Western Digital Caviar 2340 IDE hard drive, 3C509 Ethernet card.
Machine B: Intel Pentium 120 MHz, PCI bus, 32 MB of RAM, EATA-DMA DPT SCSI adapter, Conner CP30200 SCSI hard drive, NE2000 Ethernet card (in an ISA slot).

Table 5.1: Performance Measurements Results

The measurement is performed as follows. A real-time process on Machine 1 records the current time, sends a pulse to the `D0` output, and enters a busy loop waiting for the `PE` input to change. An interrupt request is sent to the CPU of Machine 2. From an interrupt handler (both real-time and ordinary Linux handlers were used) the output `D0` is toggled. The real-time process on Machine 1 exits the busy loop, obtains the current time, and computes how long it took Machine 2 to respond. This sequence is performed periodically over a substantial interval of time. The maximum response time encountered is taken to be an estimate of the worst-case interrupt latency.

As seen from Table 5.1, the interrupt handling latency in plain Linux is substantially higher than in Real-Time Linux. Moreover, it is quite possible that some device drivers that I have not used disable interrupts for longer periods, further increasing Linux interrupt processing latency.

To measure scheduling precision a periodic real-time task was run. On each wake-up the current time was obtained and compared to the estimate. Maximum deviations were recorded.

I found it impossible to reliably run periodic tasks as standard Linux processes, partly because Linux does not provide a periodic timers facility, and also because of other problems described in Chapter 2.

During all tests the system was heavily loaded with disk and network I/O operations. Although device drivers in Real-Time Linux do not disable hardware interrupts, heavy I/O does increase interrupt latency. This fact can be attributed to the DMA cycle stealing.

Several stress tests have also been performed. In one of them the system has successfully performed a backup of a local filesystem over the local network, scheduling two periodic real-time tasks each with 1 millisecond period at the same time. This experiment demonstrates that the presence of a real-time system has no adverse effect on the functioning of the Linux kernel.

Overall, the results show that Real-Time Linux is a viable platform for hard real-time processing.

Chapter 6

Future Directions

This thesis has described a way to transform a time-sharing operating system into a hard real-time one. One case study was described in detail, Real-Time Linux, a hard real-time operating system based on Linux. It has been shown that modifications made to the host operating system can be kept to minimum, while allowing to achieve good results with respect to interrupt latency and predictability.

There are several directions in which the future work can go. One of them is to explore how the ideas of Real-Time Linux can be applied to the structuring of operating system kernels.

Most device drivers have real-time constraints. These constrains are imposed by computer peripherals. A good example is serial communication hardware. Characters have to be transmitted or received at a given rate. If the rate is, say, 38400 baud (a fairly typical value), using 8 bits of data for each character plus 3 bits of control information gives 3490 characters per second. This leaves the operating system less than 300 microseconds to process each character. Although not critical during transmission, this timing requirement has to be satisfied while receiving characters to avoid losing them. Even though modern serial communication hardware has buffers, it does not completely eliminate these constraints.

However, in most modern UNIX systems, and in Linux in particular, real-time hardware requirements are not dealt with explicitly. Rather, satisfaction of timing constraints rests on the global knowledge about the maximum interrupt latency and the skill of kernel programmers to decide when it is needed and/or safe to disable or enable interrupts. As a result, a timing error made by a programmer in a hard disk driver can cause the serial port hardware to miss characters.

Real-Time Linux provides a way to solve this problem. Real-time requirements of the drivers should be made explicit and moved into the RT-kernel. This would allow them to utilize a low latency of Real-Time Linux, and to be independent of the non-real-time parts of the kernel. Moreover, this would solve a potential problem of undesirable interaction between the real-time kernel and the Linux kernel. Linux device drivers were written with an assumption that they will never be interrupted once interrupts have been disabled. This assumption is incorrect under Real-Time Linux. Although not encountered during experiments, it is conceivable that a driver can be interrupted at the wrong moment with a hardware error as a result.

Another possible direction of the future work concerns communications between real-time tasks with the rest of the system. Real-time FIFO buffers proved to be one satisfactory solution. There is a problem, however: any buffer can overflow or underflow. The former may result in a loss of data, while the latter may lead to a situation in which all real-time tasks are scheduled correctly, but deadlines are missed due to a lack of data. *Fine-grain scheduling* [18] is one way to solve this problem. Fine-grain scheduling gives higher priorities to processes that have more data than the others in their input or output buffers.

Lock-free synchronization methods [6] [18] can be used to provide various data exchange methods in addition to real-time FIFO buffers. Priority queues are also desirable in some applications.

Finally, new features can be added to the real-time kernel. Different scheduling policies come to mind first, followed by static schedulability analysis tools. Such tools can potentially solve the problem of locking the Linux kernel out of the CPU. The use of rate-monotonic analysis [17] can make it possible to guarantee some fixed fraction of the CPU time to Linux.

Bibliography

[1] Michael Beck, Harald Böhme, et al. *LINUX Kernel Internals.* Addison-Wesley, 1996.

[2] Alan Burns. Scheduling hard real-time systems: A review. *Software Engineering Journal*, 6(3):116–128, 1991.

[3] Bill Crum. Data acquisition and control using Real-Time Linux. Master's thesis, New Mexico Institute of Mining and Technology, 1997.

[4] Borko Furht, Dan Grostick, et al. *Real-time UNIX systems: design and application guide.* Kluwer Academic Publishers Group, Norwell, MA, USA, 1991.

[5] Bill O. Gallmeister. *POSIX.4 – Programming for the Real World.* O'Reilly & Associates, 1995.

[6] P. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), January 1991.

[7] Dan Hildebrand. Implementing the Win32 API over a POSIX realtime OS. Available from `http://www.qnx.com/whitepaper/qnxwin32.html`.

[8] Dan Hildebrand. An architectural overview of QNX. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 1992. USENIX.

[9] Dan Hildebrand. A microkernel POSIX OS for realtime embedded systems. Technical report, QNX Software Systems Ltd, 1993.

[10] IEEE. *Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application: Program Interface. IEEE/ANSI Std 1003.1, 1996 Edition.* Order Number SH94352-NYF.

[11] IEEE. *Information Technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities. IEEE/ANSI Std 1003.2-1992 & IEEE/ANSI 1003.2a-1992.* Order Number SH17129-NYF.

[12] Intel Corporation. *Pentium Processor Family Developer's Manual.* Order Number 241430-004.

[13] Intel Corporation. *Intel486 (TM) Processor Family. Programmer's Reference Manual*, 1995. Order Number 240486-003.

[14] Markus Kuhn. A vision for Linux 2.2 — POSIX.1b compatibility and real-time support. Available from `ftp://ftp.informatik.uni-erlangen.de/local/cip/mskuhn/misc/linux-posix.1b`.

[15] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, MA, USA, 1989.

[16] Jochen Liedtke. On micro-kernel construction. In *The Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[17] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):44–61, January 1973.

[18] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, 1992.

[19] Milan Milenković. *Operating Systems: Concept and Design.* McGraw-Hill, 1992.

[20] Dennis W. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the Association for Computing Machinery*, 17(7):365–375, July 1974.

[21] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[22] Sang H. Son, editor. *Advances In Real-Time Systems*, chapter 10, pages 225–248. Prentice Hall, 1984.

[23] Daniel Stodolsky, J. Bradley Chen, and Brian Bershad. Fast interrupt priority management in operating system kernels. In *The Proceedings of the 2nd USENIX Symposium on Microkernels and Other Kernel Architectures*. USENIX, September 1993.

[24] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[25] Martin Timmerman and Jean-Christophe Monfret. Windows NT as real-time OS? *Real-Time Magazine*, 1997. Available from `http://www.realtime-info.be/encyc/magazine/articles/winnt/winnt.htm`.

[26] Gabriel A. Wainer. Implementing real-time services in MINIX. *Operating Systems Review*, 29(3):75–84, July 1995.

[27] Wind River Systems, Inc., 1010 Atlantic Avenue, Alameda, CA 94501-1147, USA. *VxWorks Programmer's Guide 5.1*, December 1993.

Appendix A

Real-Time Task Interface

The material of this appendix is valid for Real-Time Linux version 0.5.

A.1 Task Control

`typedef struct rt_task_struct RT_TASK;`

Each real-time task is represented by a `rt_task_struct` structure. It contains task state, priority, and other parameters. The structure is opaque.

`int rt_task_init(RT_TASK *task, void (*fn)(int data), int data, int stack_size, int priority);`

Initializes the task structure pointed to by `task`. `fn` is the code for the new task; `data` is the value to pass to `fn` on startup; `stack_size` is the stack size to be allocated for this task. `priority` is task priority. It ranges from `1` (the highest) to `RT_LOWEST_PRIORITY`.

`int rt_task_make_periodic(RT_TASK *task, RTIME start_time, RTIME period);`

Marks this task as periodic; if the task is to start execution immediately, the return value of `rt_get_time()` can be used as `start_time`.

`int rt_task_delete(RT_TASK *task);`

Deactivates the task.

`int rt_task_wait(void);`

Suspends execution of the calling task until the beginning of the next period (for periodic tasks only).

`int rt_task_suspend(RT_TASK *task);`

Suspends the task.

`int rt_task_wakeup(RT_TASK *task);`

Resumes execution of the task.

`void rt_use_fp(int flag);`

Signals to the scheduler if floating point context needs to be preserved for this task. A non-zero value of `flag` makes the FP context to be saved on each context switch. The `flag` value of zero makes the opposite.

`typedef long long RTIME;`

Time is measured in clock ticks that are system-dependent.

`long long RT_TICKS_PER_SEC;`

This constant is equal to the amount of clock ticks in one second.

`RTIME rt_get_time(void);`

Returns the amount of time passed from the boot-up until the present moment. This time does not necessarily correlate with Linux time.

A.2 Real-Time FIFOs

`int rtf_create(unsigned int fifo, int size);`

Creates the FIFO buffer number `fifo`. `size` is the initial size of the buffer.

`int rtf_destroy(unsigned int fifo);`

Deactivates the FIFO. The buffer memory is freed.

`int rtf_resize(unsigned int fifo, int new_size);`

Resizes the FIFO.

`int rt_fifo_put(unsigned int fifo, char *buf, int count);`

Attempts to write `count` bytes to the FIFO identified by `fifo` from the buffer starting at `buf`. Returns -1 if there is not enough space in the FIFO; otherwise returns `count`.

`int rt_fifo_get(unsigned int fifo, char *buf, int count);`

Attempts to read `count` bytes from from the FIFO identified by `fifo` to the buffer starting at `buf`. Returns -1 if there is not enough data in the FIFO; otherwise returns `count`.

`int rtf_create_handler(unsigned int fifo, int (*handler)(unsigned int fifo));`

Attaches a handler to the FIFO.

The function `handler` will be called whenever a Linux process reads or writes to the FIFO. When the handler is called it is passed the FIFO number as the argument.

A.3 Interrupts

`int request_RTirq(unsigned int irq, void (*handler)(void));`

Installs the interrupt handler for interrupt number `irq`.

`void free_RTirq(unsigned int irq);`

Uninstalls the interrupt handler for interrupt `irq`.

Appendix B

The PC Speaker Driver Code

```
#define MODULE
#include <linux/module.h>

#include <linux/rtf.h>
#include <asm/rt_irq.h>
#include <linux/mc146818rtc.h>

#define CMOS_IRQ 8


static int filter(int x)
{
        static int oldx;
        int ret;

        if (x & 0x80) {
                x = 382 - x;
        }
        ret = x > oldx;
        oldx = x;
        return ret;

}


void intr_handler(void) {
        char data;
        char temp;
        (void) CMOS_READ(RTC_REG_C);        /* clear IRQ */
        if (rtf_get(0, &data, 1) > 0) {
                data = filter(data);
                temp = inb(0x61);
                temp &= 0xfd;
                temp |= (data & 1) << 1;
                outb(temp,0x61);  /* out to the speaker */
        }
}

char save_cmos_A;
char save_cmos_B;


int init_module(void)
{
        char ctemp;
        rtf_create(0, 4000);

        /* turn the speaker on:  */
        /* set the output of 8354's channel 2 to 1 */
        outb_p(inb_p(0x61)|3, 0x61);
        outb_p(0xb0, 0x43);
        outb_p(3, 0x42);
        outb_p(0, 0x42);

        request_RTirq(CMOS_IRQ, intr_handler);

        save_cmos_A = CMOS_READ(RTC_REG_A);
        save_cmos_B = CMOS_READ(RTC_REG_B);

        /* 32 kHz base; interrupt at 8192 Hz */
        CMOS_WRITE(0x23, RTC_REG_A);

        /* enable periodic interrupts */
        ctemp = CMOS_READ(RTC_REG_B);
        ctemp &= 0x8f;
        ctemp |= 0x40;
        CMOS_WRITE(ctemp, RTC_REG_B);

        (void) CMOS_READ(RTC_REG_C);    /* Clear the interrupt */

        return 0;
}


void cleanup_module(void)
{
        rtf_destroy(0);
        CMOS_WRITE(save_cmos_A, RTC_REG_A);
        CMOS_WRITE(save_cmos_B, RTC_REG_B);
        free_RTirq(8);
}
```