A Real-Time Linux

Victor Yodaiken and Michael Barabanov
New Mexico Institute of Technology [1]

Abstract

The paper describes the design, implementation, experimental results, and possible applications of a real-time version of the Linux operating system. We have used the idea of virtual machines for running a standard time-sharing OS and a real-time executive on the same computer. Services provided by the real-time kernel are described. The comparison of our solution with other work on real-time UNIXes is also presented.

1 Introduction

We have developed a small, real-time executive that runs the Linux operating system as a completely preemptable task. The executive schedules and runs real-time tasks at a relatively high level of time precision and with low latency and overhead. The Linux task supports network services, GUI, development tools and a standard programming environment. Our system has been designed to require little modification of Linux itself — essentially changes are limited to the low level interrupt "wrappers" and the routines to disable and enable interrupts. As a result, we are able to take advantage of the rapid pace of development of Linux and Linux tools. On the other hand, our system has been designed to allow real-time programmers to make nearly full use of the available hardware and processing power, without paying the price normally associated with more sophisticated operating systems. While our immediate interests are in the control of scientific instruments, we believe this method to be generalizable to other operating systems and to other real-time problems. We also believe that this method offers an alternative avenue to modularity that may be of interest in general operating system design.

Real-time operating systems should be small, fast, and predictable. Because of the great variety of demands on real-time scheduling, a real-time operating system should also include a flexible and reprogrammable task scheduling discipline. These requirements are not easy to satisfy, but it has been increasingly clear over the last several years that real-time operating systems also need to satisfy user requirements for sophisticated development tools, graphical user interfaces, and networking support. We have attacked this problem of apparently contradictory requirements using a simple version of the well known "virtual machine" technique [12]. Linux interacts with a software emulation of the interrupt control hardware. The emulation supports the synchronization requirements of the Linux kernel while preventing Linux from disabling interrupts. Interrupts that are handled by Linux are passed through to the emulation software after any needed real-time processing completes. If Linux has requested that interrupts be disabled, the emulation software simply marks the interrupts as pending. When Linux requests that interrupts be enabled, the emulation

[1] Email address: `yodaiken@nmt.edu`.
The research described here was partially funded under NSF grant CCR-9409454.

software causes control to switch to the Linux handler for the highest priority pending interrupt. Linux is then able to provide sophisticated services to the real-time system without increasing interrupt latency.

A virtual machine layer has been advanced as a technique for making UNIX real-time as far back as 1978 [8], but our use of the technique differs from previous efforts in both scope and purpose. Our virtual machine "layer" emulates only a specific hardware component — interrupt control. Linux is able to otherwise directly control the hardware both for run-time efficiency and in order to minimize the need for modifications to the Linux kernel. The real-time executive which acts as the 0-level operating system does not provide any basic services that can be provided by Linux. Instead the real-time executive is intended to provide services that Linux cannot provide. Thus, "primitives" for process creation and switching or memory management are not provided by the real-time executive. Only real-time services are provided.

The remainder of this paper is in four parts. Section 2 describes the applications we have in mind, the constraints we have, and the experimental results we have obtained. Section 3 details the services provided by the real-time executive. Section 4 describes our virtual machine implementation on the x86 architecture. The conclusion compares this approach to other work on real-time OS design and points out the directions for future work.

2 Goals, barriers, and measured results

Our immediate goal was to develop a Linux kernel that would support real-time control of scientific instruments. The limitations of standard time-shared operating system for this purpose are obvious, but we should mention both unpredictability of execution and high interrupt latency as critical problems. General purpose time-shared operating systems have schedulers that are intended to balance response time and throughput. As a result the execution of any process depends in a complex and unpredictable fashion on system load and the behavior of other processes. These problems are compounded in Linux and most other UNIX derivatives, because kernel mode execution is non-preemptable [6] and because disabling interrupts is used as the primary means of synchronization.

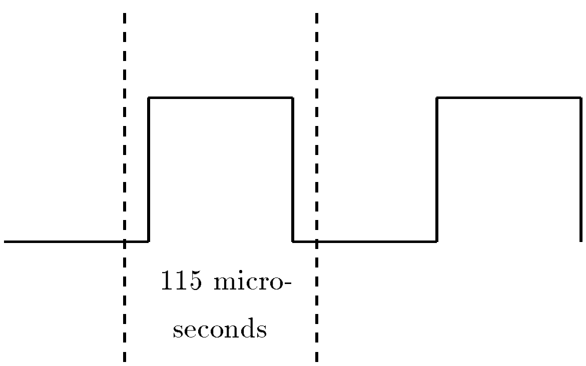
By locking process pages into memory and requiring use of a round-robin scheduler as in the POSIX.1b standard [2] one can gain a certain degree of predictability, but still not meet the requirements of even moderately demanding hard real-time systems [14]. Low interrupt handling latency is critical for any real-time operating system. But interrupt latency is high in in Linux. On a 120MHz Pentium based PC, we measure up to 400 μsec latency in handling of "fast" Linux interrupts. It has been reported that the Linux console driver disables interrupts for as long as several milliseconds when switching virtual consoles. Clearly, a frame-buffer that must be emptied every millisecond is then beyond the capabilities of the system and this timing requirement is one of the least demanding that we can expect to see.

The fundamental limits for real-time processing are determined by the hardware. For example, on our test system we measure a time of approximately 3.2μsec for setting a bit on the parallel port. Obviously we cannot then support a requirement for a data rate of over

280 KHZ no matter what we do with the operating system. Similarly, the minimal interrupt latency is bounded by the hardware interrupt processing time. On a Pentium processor, at least 61 cycles is needed to enter and exit the interrupt, and some time is also needed for the interaction with the interrupt controller. Devices that need more rapid response or more precise timing call for dedicated, or at least different, hardware. But modern PC hardware is capable of handling the real-time requirements of a wide range of devices.

The current version of RT-Linux is a modification of Linux 1.3.32. Efforts are currently underway to move to a 2.0 Linux kernel and we hope to port the system to a PowerPC box in the next month or so. Our test system has a 120MHZ Pentium processor, a 512KB secondary cache and 32MB of main memory. All I/O devices, other than the video display and keyboard are DMA devices. Non-DMA controllers for mass storage devices are difficult to integrate into a real-time control system. The

To validate the performance of our real-time Linux, we have concentrated on periodic tasks such as those needed to control a stepper motor or to sample sensors. The most demanding test of real-time scheduling was a task with a 100μsec period that did nothing but toggle a bit on the parallel port. We attached the output pin to a digital storage oscilloscope to measure timing and latency. Our measurements show a square wave with a maximum variation of 15μsec even during very heavy system load. This 15μsec includes the time to process an interrupt from the clock, detect that the toggle task needed to run, and start the toggle task.



A second experiment used two periodic tasks with 100μsec periods where one set and one cleared the same bit on the parallel port. This experiment also showed a maximum variation of 15 μsec. Both experiments measured timings under high system demand: a recursive disk copy was running under Linux, a network connection was driving a remote X-windows display, and the Netscape browser program was both started and used to display graphics. The changes in Linux work-load had no effect on the timing of the real-time tasks. At periods of significantly less than 50μsec Linux performance became too poor to be of much use. A single real-time task with a scheduled period of 40μsec essentially prevents Linux from progressing. This performance characteristic is the desired one: the non-real-time system gets whatever processing time is not needed by the real-time system. The perils of measuring timing on a system with a significant cache and complex pipeline are not to be discounted, but these experiments do indicate that the real-time OS makes it possible to control devices at quite a precise level of timing.

3 Interface

Real-time processes are light-weight threads executing each in its own address space. The context of a real-time process consist only of integer registers. This ensures fast context switch, which, together with the overhead of a system call to the RT-kernel takes 224 processor cycles[2].

Currently only periodic real-time tasks are supported. Essentially this this means that we dispatch the timer interrupt. Dispatching other interrupts can be added trivially.

RT-Linux provides the following system calls for the process control.

- `int RTload(const char *file)` loads a RT-program "`FILE`", creates a process and returns its pid. The process is suspended until `RTrun` is called.
- `int RTrun(int pid)` starts execution of the RT-process at a low priority level; the process must be first loaded with `RTload` routine
- `int RTkill(int pid)` kills a RT-process
- `int RTget_time(RTime *t)` returns current time; time is a 64 bit integer (long long int), containing number of clock ticks passed since system booted The constant `RT_TICKS_PER_SEC` contains the number of clocks per second.
- `int RTset_params(RTime * start, RTime * period, int priority)` changes the scheduling parameters of the process
- `int RTwait_start(RTime * start, RTime * period, int priority)` suspends the process until its start time; when start time comes, set the priority of the process to the requested value.
- `int RTwait_period()` suspends the execution of the process until the beginning of the next period.

Real-time Linux does not use the hardware context switch mechanism that Intel x86 processors provide - it saves too much state and so is not fast enough. Instead we save the context on the stack and then switch stacks.

A simple priority-based preemptive scheduler is currently used in real-time Linux. It is implemented as a routine which chooses among the ready process the highest-priority one and marks it as a next process to execute. Tasks give up the processor voluntarily, or are preempted by a higher-priority task when its time to execute comes.

Typically there is a tradeoff between the clock interrupt rate and the *task release jitter* [13]. In most systems tasks are resumed in the periodic clock interrupt handler. High clock interrupt rate ensures low jitter, but at the same time incurs much overhead. Low interrupt rate causes tasks to be resumed either too early or too late. In RT-Linux this tradeoff is resolved by using a one-shot timer instead of periodic clock. Tasks are resumed in the timer interrupt handler precisely when needed.

[2]These were calculated for the Intel 486 Processor; no cache and TLB misses was assumed

Note that all task resources are statically defined. In particular there is no support for dynamic memory allocation. Our basic approach here is that any sophisticated services that require dynamic memory allocation should be moved into Linux processes. In keeping with this approach the Real-time kernel itself is not preemptable.

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 FIFOs are used in RT-Linux for moving information between Linux processes or the Linux kernel and real-time processes. In a data-collecting application, for example, a real-time process would poll a device, and put the data into a FIFO. Linux process can then be used for reading the data from the FIFO and storing it in the file, or displaying it on the screen. Currently, interrupts are disabled when a RT-FIFO is accessed. Since data are transmitted in small chunks, this does not compromise a low response time. Other approaches, notably using lock-free data structures [3], [9] are also possible and are being considered.

The following are the system calls related to RT-FIFOs.

- `int RTfifo_create(unsigned int fifo, int size)` creates a RT-FIFO "`FIFO`" of size "`SIZE`" bytes. FIFOs' numbers are global; FIFOs are numbered from 0 to `RT_MAX_FIFO-1`. Applications must agree on the use of the FIFOs available.

- `int RTfifo_destroy(unsigned int fifo)` destroys a FIFO.

- `int RTfifo_get(unsigned int fifo, char * buf, int count)` reads "`COUNT`" bytes from "`FIFO`" to "`BUF`". return -1 if there is not enough data in the FIFO; otherwise return "`COUNT`".

- `int RTfifo_put(unsigned int fifo, char * buf, int count)` writes "`COUNT`" bytes from "`BUF`" to "`FIFO`"; return -1 if there is not enough space in the FIFO; otherwise return "`COUNT`".

Our next version of RT-Linux will abandon the separate address space for real-time tasks and will place real-time tasks within the Linux Kernel address space. The original design was motivated by a desire to have relatively user-safe real-time facilities. Real-time tasks loaded into their own address space cannot overwrite Linux data structures although they are capable of crashing the system in other ways. Our next system will sacrifice some of this safety for speed and the convenience of using Linux modules to allow dynamic loading and replacement of real-time tasks and the real-time scheduler.

4 The Virtual Machine

The RT-executive has been implemented on the x86/PC architecture [5] [10].

4.1 Interrupt handling

Modifications to the Linux kernel are primarily in three places:

- The `cli` routine to *disable* interrupts is modified to simply clear a global variable controlling *soft interrupt enable*.
- The `sti` routine to *enable* interrupts is modified to generate emulated interrupts for any pending soft interrupts.
- The low-level "wrapper" routines which save and restore state around calls to handlers have been changed to use soft *return from interrupt* code instead of using the machine instruction.

When an interrupt occurs, control switches to a real-time handler. The handler does whatever needs to be done in the real-time executive and then may pass the interrupt on to Linux. If the soft interrupt enable flag is set, then the stack is adjusted to fit the needs of the Linux handler and control is passed, via a soft interrupt table, to the appropriate Linux "wrapper". The "wrapper" saves additional state and calls the Linux handler — a program usually written in C. When the handler returns control to the "wrapper" a *soft return from interrupt* is executed. Soft return from interrupt restores state and then checks to see if any other soft interrupts are pending. If not, a hard return from interrupt is executed. If there are interrupts pending, then the highest priority one is processed.

Linux is reasonably easy to modify because, for the most part, the kernel code controls interrupt hardware through the routines `cli`() and `sti`(). In standard x86 Linux, these routines are actually assembly language macros that generate the x86 `cli` (clear interrupt bit) and `sti` (set interrupt bit) instructions for changing the processor control word. Because interrupts can be disabled and enabled individually in the interrupt controller, and because some Linux drivers directly access the interrupt controllers and the hardware timer, we also had to modify some driver code. All in all, our changes required under 2000 lines of new code, and modification of a few hundred lines of the Linux code.

Figure 1 shows the code for three macros.

Interrupt handlers in the RT-executive perform whatever function is necessary for the RT system and then may pass interrupts on to Linux. Since the real-time system is not involved in most I/O, most of the RT device interrupt handlers simply notify Linux. On the other hand, the timer interrupt increments timer variables, determines whether a RT task needs to run, and passes interrupts to Linux only at appropriate intervals.

If software interrupts are disabled (`SFIF == 0`), control simply returns through `iret`. Otherwise, control is passed to `S_IRET`. This macro invokes the software handler corresponding to the interrupt that has the highest priority among pending and not masked ones.

The `S_IRET` code begins by saving minimal state and making sure that the kernel data address space is accessible. In the critical section surrounded by the actual `cli` and `sti` we apply the software interrupt mask to the variable containing pending interrupts, and then look for the highest-priority pending interrupt. If there are no software interrupts to be processed, we re-enable software interrupts, restore the registers, and return from the interrupt. If we find an interrupt to process, we pass control to its Linux "wrapper".

Each Linux "wrapper" has been modified to fix the stack so that it looks as if control has been passed directly from the hardware interrupt. This step is essential because Linux actually looks in the stack to see if the system was in user or kernel mode when the interrupt

Figure 1: "Soft" `cli`, `sti` and `iret`

```
/* These are macros */

S_CLI:  movl $0, SFIF

S_IRET: push %ds
        pushl %eax
        pushl %edx
        movl $KERNEL_DS, %edx
        mov %dx,%ds
        cli
        movl SFREQ,%edx
        andl SFMASK,%edx
        bsrl %edx,%eax
        jz not_found
        movl $0,SFIF
        sti
        jmp  SFIDT (,%eax,4)
not_found:
        movl $1,SFIF
        sti
        popl %edx
        popl %eax
        pop %ds
        iret

S_STI:  pushfl
        pushl $KERNEL_CS
        pushl $done_STI
        S_IRET
done_STI:
```

occurred. If Linux believes that the interrupt occurred in kernel mode, it will not call its own scheduler. The body of the wrapper has not been modified, but instead of terminating with an `iret` operation, the modified wrapper invokes `S_IRET`. Thus, wrappers essentially invoke each other until there are no pending interrupts left.

On re-enabling software interrupts, all pending ones, of course, should be processed. The code simulates a hardware interrupt. We push the flags and the return address onto the stack, and use `S_IRET` (see Figure 1).

Individual disabling/enabling of interrupts is handled similarly.

5 Conclusion

Our approach to building a real-time operating system can be contrasted two two more well-known methods. One method is add real-time support to a general purpose operating system. The Real-Time Unix of [1] is a good example of this approach and illustrates the effort needed to make a Unix kernel fully preemptive. Other examples include VAX VMS [?], the POSIX 1.b standard (and the similar work in [15]) and the Maruti real-time OS [7]. The second approach is to design an operating system specifically to support real-time. VX-Works[16] is a particularly successful example of such a system. Other examples include the QNX microkernel [4] and OS9[11].

We have chosen a third path. Real-time POSIX standards alone are not "hard" enough for our purposes. To make Linux fully pre-emptable was too time consuming and would cut us off from the mainstream of Linux development. We are interested in real-time operating system design and want very much to leave TCP/IP, NFS, GUIs, and other important general purpose operating system components to others. But the special purpose operating systems have the same problem. Vendors are rapidly adding support for general purpose operating system utilities. In fact, several vendors are now advertising "POSIX compatibility". Grafting POSIX to a real-time operating system seems to us to be no less complicated and time consuming than grafting real-time on to an existing general purpose operating system. With both approaches, we were concerned that the interaction between the real-time and non-real-time subsystems would cause problems that our approach avoids through its clear separation between real-time and general purpose components.

Finally, although we have made an effort to modify Linux as little as possible, the real-time executive approach might be used as a basis for a significant redesign of Linux and similar operating systems. For example, device drivers often have real-time constraints. If the real-time requirements of the drivers were made explicit and moved into the RT-kernel, then configuration programs could attempt to find a feasible schedule rather than allowing users to find out by experiment whether device timing constraints are feasible. It may also be possible to simplifiy design of the general purpose kernel by giving the emulation a cleaner semantics than the actual hardware.

References

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

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

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

[4] Dan Hildebrand. An architectural overview of QNX. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, April 27-28 1992. USENIX.

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

[6] 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.

[7] S.-T. Levi, S. K. Tripathi, S. D. Carson, and A. K. Agrawala. The MARUTI hard real-time operating system. *ACM Operating Systems Review, SIGOPS*, 23(3):90–105, July 1989.

[8] H. Lycklama and D. L. Bayer. Unix time-sharing system: The MERT operating system. *Bell System Technical Journal*, 57(6):2049–2086, 1978.

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

[10] Muhammad Ali Mazidi and Janice Gillespie Mazidi. *Design and Interfacing of the IBM PC, PS, and Compatibles.* Prentice Hall, 1995.

[11] OS9 Real-Time Operating System. http://www.gespac.com/html/os9_arch_diagram.html.

[12] L. H. Seawright and Mackinnon R. A. VM/370 — A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 18:4–17, 1978.

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

[14] J. A. Stankovic. Misconceptions about real-time computing - A serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19, October 1988.

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

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