

The RT-Linux approach to hard real-time

(A short position paper)

Victor Yodaiken

Department of Computer Science

New Mexico Institute of Technology

Socorro, NM

yodaiken@cs.nmt.edu

<http://luz.nmt.edu/~rtlinux>

RT-Linux is an operating system in which a small real-time kernel coexists with the Posix-like Linux kernel. The intention is to make use of the sophisticated services and highly optimized average case behavior of a standard time-shared computer system while still permitting real-time functions to operate in a predictable and low-latency environment. At one time, real-time operating systems were primitive, simple executives that did little more than offer a library of routines. But real-time applications now routinely require access to TCP/IP, graphical display and windowing systems, file and data base systems, and other services that are neither primitive nor simple. One solution is to add these non-real-time services to the basic real-time kernel, as has been done for the venerable VXworks and, in a different way, for the QNX microkernel. A second solution is to modify a standard kernel to make it completely pre-emptable. This is the approach taken by the developers of RT-IX (Modcomp). RT-Linux is based on a third path in which a simple real-time executive runs a non-real-time kernel as its lowest priority task, using a virtual machine layer to make the standard kernel fully pre-emptable.

In RT-Linux, all interrupts are initially handled by the Real-Time kernel and are passed to the Linux task only when there are no real-time tasks to run. To minimize changes in the Linux kernel, it is provided with an emulation of the interrupt control hardware. Thus, when Linux has "disabled" interrupts, the emulation software will queue interrupts that have been passed on by the Real-Time kernel. Real-time and Linux user tasks communicate through lock-free queues and shared memory in the current system. From the application programmers point of view, the queues look very much like standard UNIX character devices, accessed via POSIX `read/write/open/ioctl` system calls. Shared memory is currently accessed via the POSIX `mmap` calls. RT-Linux relies on Linux for booting, most device drivers, networking, file-systems, Linux process control, and for the loadable kernel modules which are used to make the real-time system extensible and easily modifiable. Real-time applications consist of real-time tasks that are incorporated in loadable kernel modules and Linux/UNIX processes that take care of data-logging, display, network access, and any other functions that are not constrained by worst case behavior.

In practice, the RT-Linux approach has proven to be very successful. Worst case interrupt latency on a 486/33Mhz PC measures well under 30microseconds, close to the hardware limit. Many applications appear to benefit from a synergy between the real-time system and the average case optimized standard operating system. For example, data-acquisition applications are usually composed a simple polling or interrupt driven real-time task that pipes data through a queue to a Linux process that takes care of logging and display. In such cases, the I/O buffering and aggregation performed by Linux provides a high level of average case performance while the real-time task meets strict worst-case limited deadlines.

RT-Linux is both spartan and extensible in accord with two, somewhat contradictory design premises. The first design premise is that the truly time constrained components of a real-time application are not

compatible with dynamic resource allocation, complex synchronization, or anything else that introduces either hard to bound delays or significant overhead. The most widely used configuration of RT-Linux offers primitive tasks with only statically allocated memory, no address space protection, a simple fixed priority scheduler with no protection against impossible schedules, hard interrupt disabling and shared memory as the only synchronization primitives between real-time tasks, and a limited range of operations on the FIFO queues connecting real-time tasks to Linux processes. The environment is not really as austere as all that, however, because the rich collection of services provided by the non-real-time kernel are easily accessed by Linux user tasks. Non-real-time components of applications migrate to Linux. One area where we hope to be able to make particular use of this paradigm is in QOS, where it seems reasonable to factor applications into hard real-time components that collect or distribute time sensitive data, and Linux processes or threads that monitor data rates, negotiate for process time, and adjust algorithms.

The second design premise is that little is known about how real-time systems should be organized and the operating system should allow for great flexibility in such things as the characteristics of real-time tasks, communication, and synchronization. The kernel has been designed with replaceable modules wherever practical and the spartan environment described in the previous paragraph is easily ``improved" (or ``cluttered", depending on one's point of view). There are alternative scheduling modules, some contributed by the user community, to allow for EDF and rate-monotonic scheduling of tasks. There is a ``semaphore module" and there is active development of a richer set of system services. Linux makes it possible for these services to be offered by loadable kernel modules so that the fundamental operation of the real-time kernel is run-time (although not real-time) reconfigurable. It is possible to develop a set of tasks under RT-Linux, test a system using a EDF schedule, unload the EDF scheduling module, load a rate monotonic scheduling module, and continue the test. It should eventually be possible to use a memory protected process model, to test different implementations of IPCs, and to otherwise tinker with the system until the right mix of services is found.

-
- [About this document ...](#)

Victor Yodaiken

Mon Oct 6 23:35:42 MDT 1997