Anatomy of real-time Linux architectures

From soft to hard real-time

Skill Level: Intermediate

M. Tim Jones (mtj@mtjones.com)
Consultant Engineer
Emulex Corp.

15 Apr 2008

It's not that Linux® isn't fast or efficient, but in some cases fast just isn't good enough. What's needed instead is the ability to deterministically meet scheduling deadlines with specific tolerances. Discover the various real-time Linux alternatives and how they achieve real time—from the early architectures that mimic virtualization solutions to the options available today in the standard 2.6 kernel.

This article explores some of the Linux architectures that support real-time characteristics and discusses what it really means to be a *real-time architecture.* Several solutions endow Linux with real-time capabilities, and in this article I examine the thin-kernel (or micro-kernel) approach, the nano-kernel approach, and the resource-kernel approach. Finally, I describe the real-time capabilities in the standard 2.6 kernel and show you how to enable and use them.

Defining real time and its requirements

The following definition of *real time* sets the stage for discussing real-time Linux architectures. This definition comes from Donald Gillies in the Realtime Computing FAQ (see Resources for a link):

> A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred.

More in Tim's Anatomy of... series on developerWorks

- Anatomy of the Linux SCSI subsystem
- Anatomy of the Linux file system
- Anatomy of the Linux networking stack
- Anatomy of the Linux kernel
- Anatomy of the Linux slab allocator
- Anatomy of Linux synchronization methods
- All of Tim's *Anatomy of...* articles
- All of Tim's articles on developerWorks

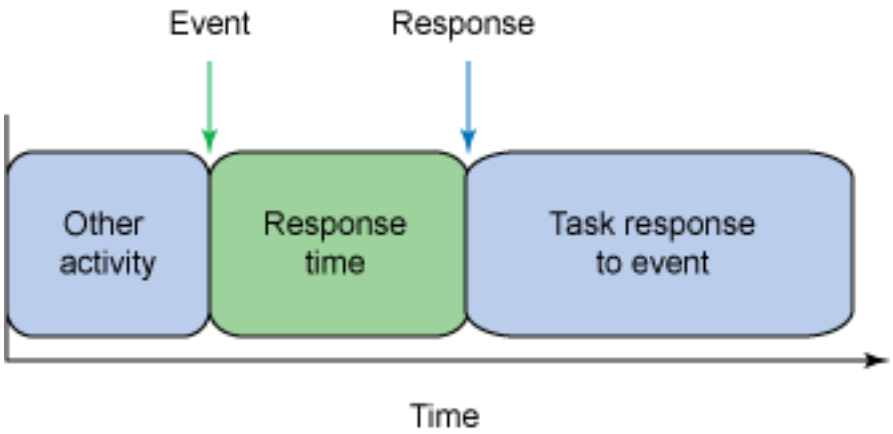
In other words, the system must be deterministic to guarantee timing behavior in the face of varying loads (from minimal to worst case). Note that the above definition says nothing about performance, because real time isn't about speed: It's about predictability. For example, using a fast modern processor, Linux can provide a typical interrupt response of 20µs, but occasionally the response can be much longer. This is the fundamental problem: It's not that Linux isn't fast or efficient, it's just that it isn't deterministic.

Some examples will demonstrate what all this means. Figure 1 shows the measure of interrupt latency. When an interrupt arrives (the *event*), the CPU is interrupted and enters interrupt processing. Some amount of work is done to determine what event occurred and, after a small amount of work, the required task is dispatched to deal with the event (a *context switch*). The time between the arrival of the interrupt and dispatching of the required task (assuming it's the highest-priority task to dispatch) is called the *response time.* For real time, the response time should be deterministic and operate within a known worst-case time.

Context switching

Implicit in the process of dispatching a new task based on an interrupt is a *context switch.* This is the process of storing the current state of a CPU at interrupt time, and then restoring the state of a given task. What constitutes a context switch is a function of both the operating system and the underlying processor architecture.

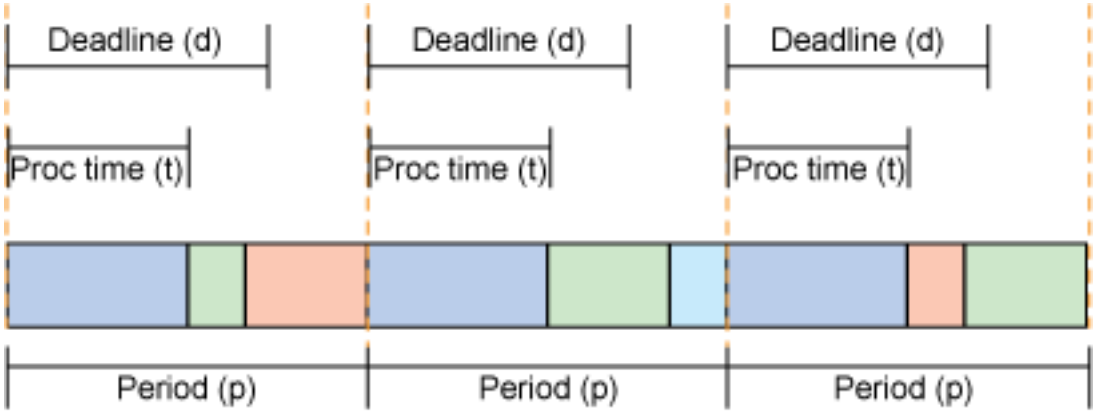
Figure 1. Interrupt latency and response time



One useful example of this process is the standard airbag in vehicles today. When the sensor that reports a vehicle collision interrupts the CPU, the operating system should quickly dispatch the task that deploys the airbag rather than allow other, non-real-time processing to interfere. An airbag that deploys a second later than it should is worse than no airbag at all.

In addition to bringing determinism to interrupt processing, task scheduling that supports periodic intervals is also needed for real-time processing. Consider Figure 2. This figure demonstrates periodic task scheduling. Many control systems require periodic sampling and processing. At a defined period (*p*), a particular task must execute for system stability. Consider a vehicle anti-lock braking system (ABS). This control system samples the speed of each wheel on a vehicle and controls each brake's pressure (to stop it from locking up) at up to 20 times per second. For the control system to work, sensor sampling and control must be performed at periodic intervals. This means that other processing must be preempted to allow the ABS task to execute at the desired periods.

Figure 2. Periodic task scheduling



Hard real-time vs. soft real-time systems

An operating system that can support the desired deadlines of the real-time tasks (even under worst-case processing loads) is called a *hard real-time* system. But

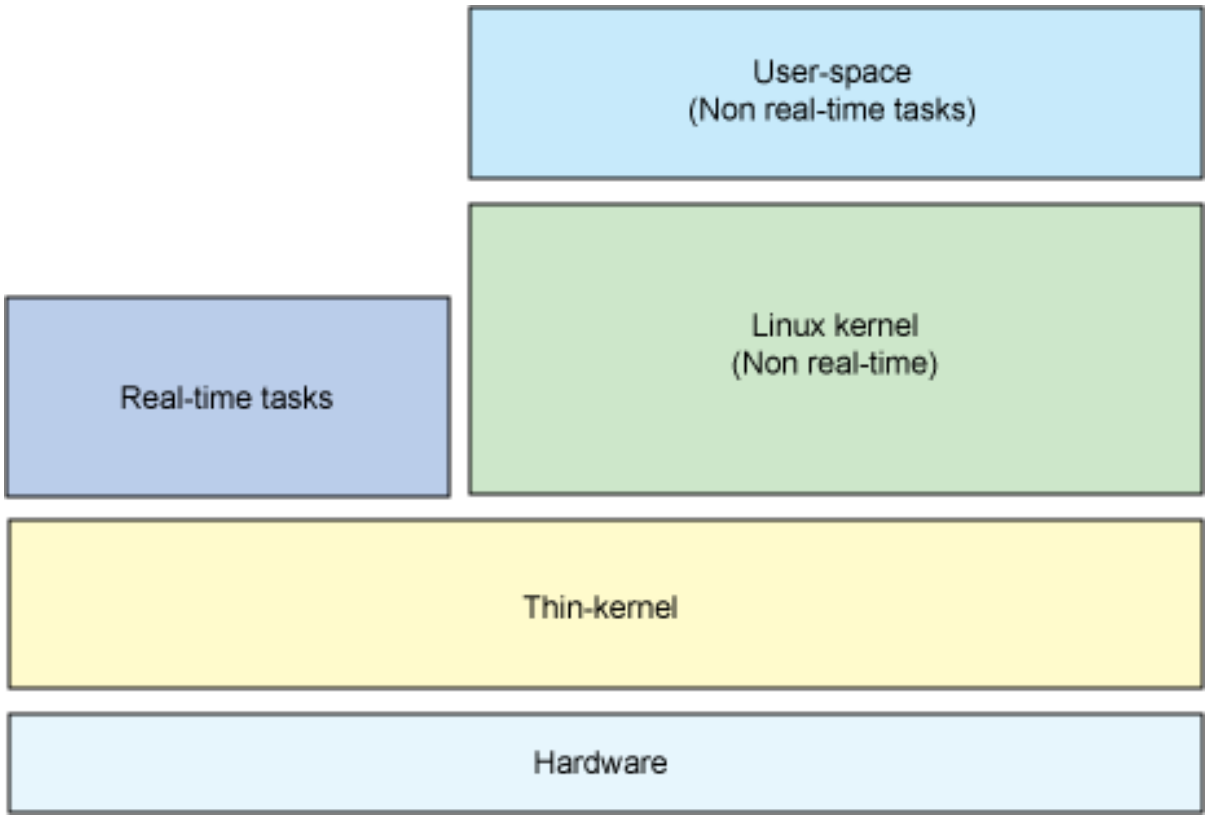
hard real-time support isn't necessary in all cases. If an operating system can support the deadlines on average, it's called a *soft real-time* system. Hard real-time systems are those in which missing a deadline can have a catastrophic result (such as deploying an airbag too late or allowing brake pressure to slip for too long). Soft real-time systems can miss deadlines without the overall system failing (such as losing a frame of video).

Now that you have some insight into real-time requirements, let's look at some of the Linux real-time architectures to see what level of real time they support and how.

Thin-kernel approach

The thin-kernel (or micro-kernel) approach uses a second kernel as an abstraction interface between the hardware and the Linux kernel (see Figure 3). The non-real-time Linux kernel runs in the background as a lower-priority task of the thin kernel and hosts all non-real-time tasks. Real-time tasks run directly on the thin kernel.

Figure 3. The thin-kernel approach to hard real time



The primary use of the thin kernel (other than hosting the real-time tasks) is interrupt management. The thin kernel intercepts interrupts to ensure that the non-real-time kernel cannot preempt the operation of the thin kernel. This allows the thin kernel to

provide hard real-time support.

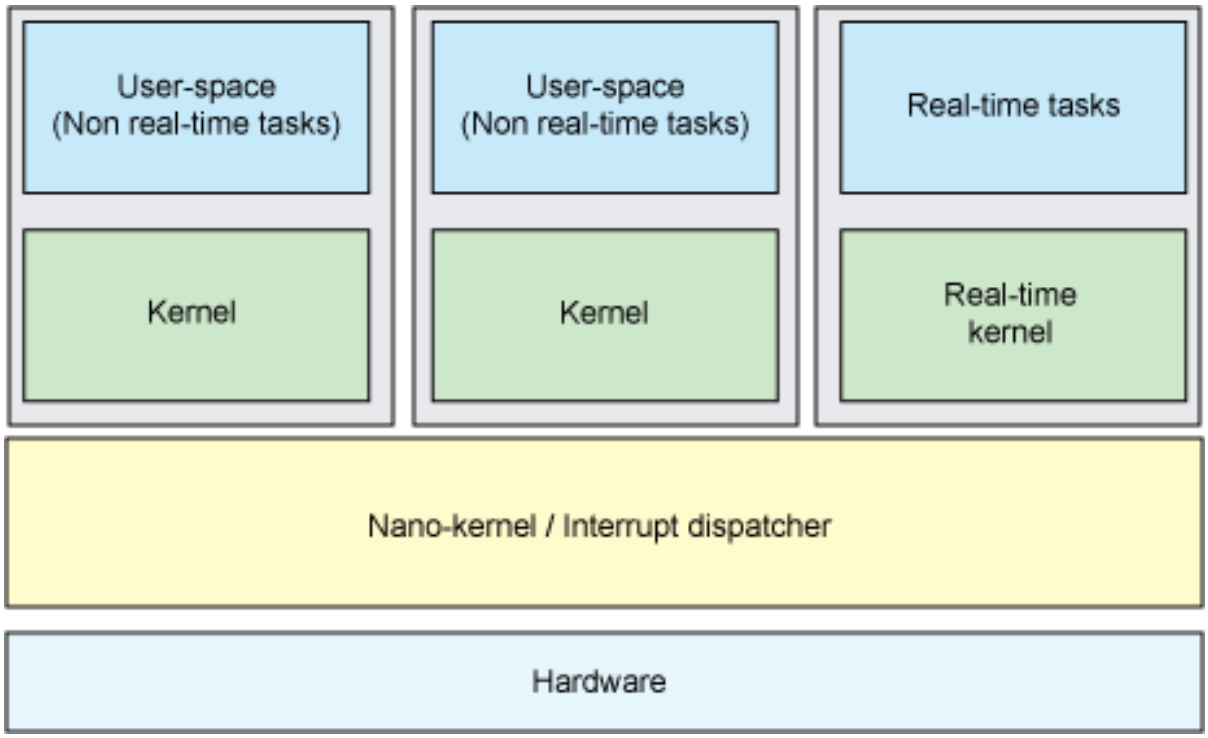
Although the thin kernel approach has its advantages (hard real-time support coexisting with a standard Linux kernel), the approach does have drawbacks. The real-time and non-real-time tasks are independent, which can make debugging more difficult. Also, non-real-time tasks do not have full Linux platform support (the thin kernel execution is called *thin* for a reason).

Examples of this approach include RTLinux (now proprietary and owned by Wind River Systems), Real-Time Application Interface (RTAI), and Xenomai.

Nano-kernel approach

Where the thin kernel approach relies on a minimized kernel that includes task management, the nano-kernel approach goes a step further by minimizing the kernel even more. In this way, it's less a kernel and more a hardware abstraction layer (HAL). The nano-kernel provides for hardware resource sharing for multiple operating systems operating at a higher layer (see Figure 4). Because the nano-kernel abstracts the hardware, it can provide prioritization for higher-layer operating systems and, therefore, support hard real time.

Figure 4. The nano-kernel approach to hardware abstraction



Note the similarities between this approach and the virtualization approach for running multiple operating systems. In this case, the nano-kernel abstracts the

hardware from the real-time and non-real-time kernels. This is similar to the way that hypervisors abstract the bare hardware from the guest operating systems. See Resources for more information.

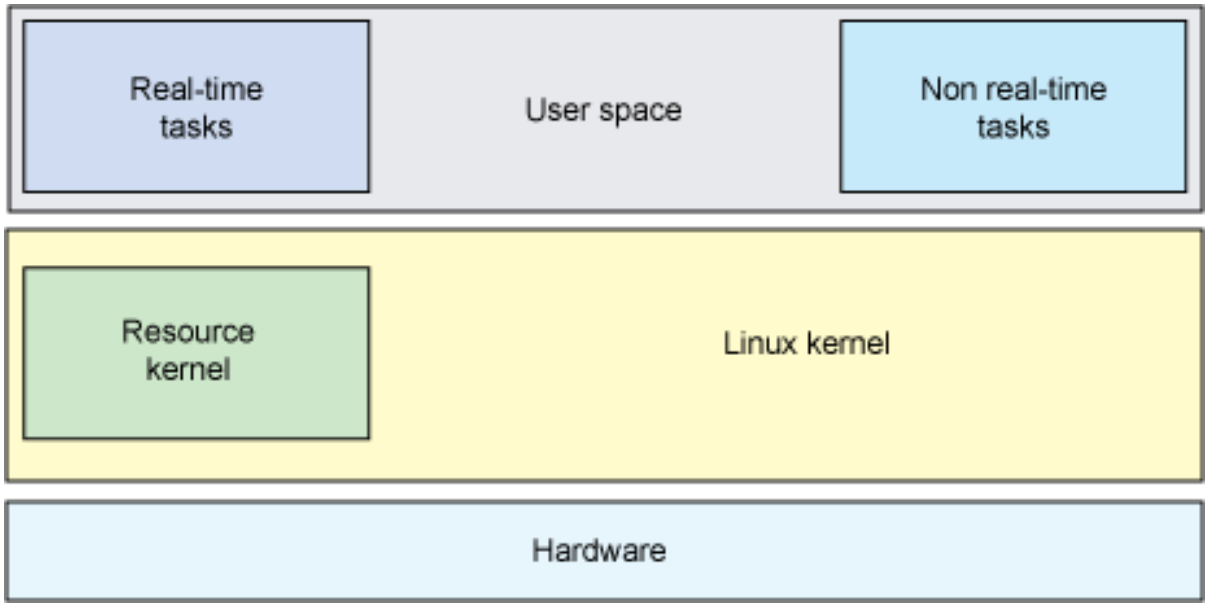
An example of the nano-kernel approach is the Adaptive Domain Environment for Operating Systems (ADEOS). ADEOS supports multiple concurrent operating systems running simultaneously. When hardware events occur, ADEOS queries each operating system in a chain to see which will handle the event.

Resource-kernel approach

Another architecture for real time is the resource-kernel approach. This approach adds a module to a kernel to provide reservations for various types of resources. The reservations guarantee access to time-multiplexed system resources (CPU, network, or disk bandwidth). These resources have several reserve parameters, such as the period of recurrence, the required processing time (that is, the amount of time needed for processing), and the deadline.

The resource kernel provides a set of application program interfaces (APIs) to allow tasks to request these reservations (see Figure 5). The resource kernel can then merge the requests to define a schedule to provide guaranteed access using the task-defined constraints (or return an error if they cannot be guaranteed). Using a scheduling algorithm such as Earliest-Deadline-First (EDF), the kernel can then be used to handle the dynamic scheduling workload.

Figure 5. The resource kernel approach to resource reservation



One example of a resource kernel implementation is CMU's Linux/RK, which integrates a portable resource kernel into Linux as a loadable module. This

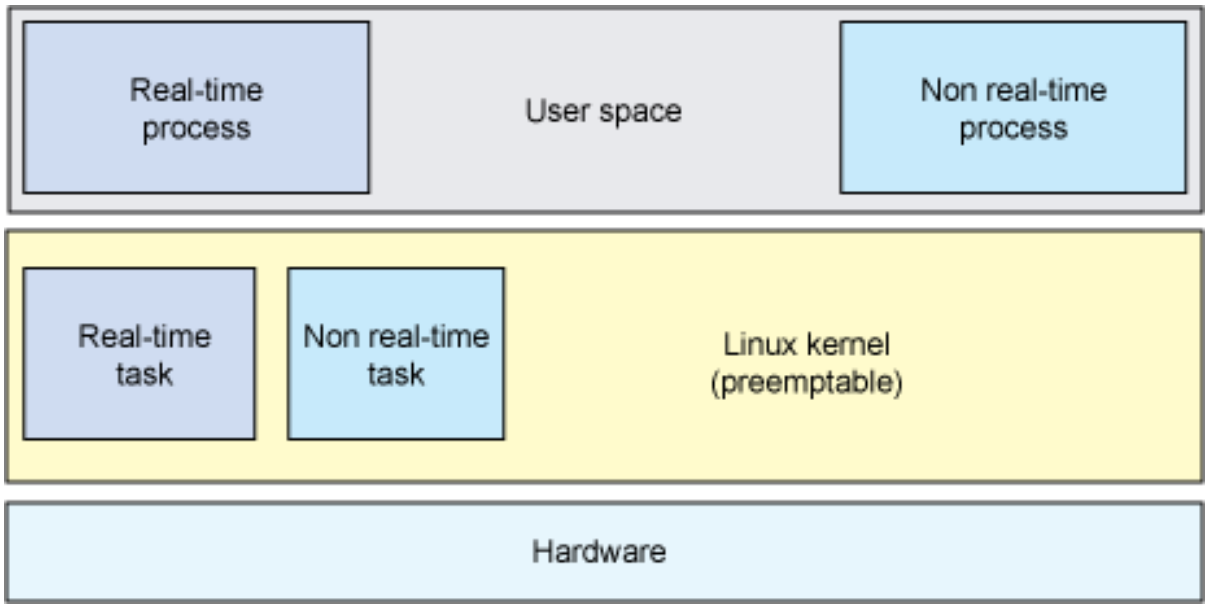
implementation evolved into the commercial TimeSys Linux/RT offering.

Real time in the standard 2.6 kernel

While all the approaches discussed so far are architecturally interesting, they all operate around the periphery of the kernel. Instead, what if the standard Linux kernel incorporated the necessary changes to support real time?

Today, in the 2.6 kernel, you can get soft real-time performance through a simple kernel configuration to make the kernel fully preemptable (see Figure 6). In the standard 2.6 Linux kernel, when a user space process makes a call into the kernel (through a system call), it cannot be preempted. This means that if a low-priority process makes a system call, a high-priority process must wait until that call is complete before it can gain access to the CPU. The new configuration option `CONFIG_PREEMPT` changes this behavior of the kernel by allowing processes to be preempted if high-priority work is available to do (even if the process is in the middle of a system call).

Figure 6. Standard 2.6 Linux kernel with preemption



But this configuration option has a trade-off. Although the option enables soft real-time performance and even under load makes the operating system execute more smoothly, it does so at a cost. That cost is slightly lower throughput and a small reduction in kernel performance because of the added overhead of the `CONFIG_PREEMPT` option. This option is useful for desktop and embedded systems, but it may not be right in all scenarios (for example, servers).

The new O(1) scheduler
The new O(1) scheduler in the 2.6 kernel is a great benefit to

performance, even when a large number of tasks exists. This new scheduler can operate in bounded time regardless of the number of tasks to execute. You can read more about this schedule and how it works in the Resources section.

Another useful configuration option in the 2.6 kernel is for high-resolution timers. This new option allows timers to operate down to 1µs resolution (if supported by the underlying hardware) and also implements timer management with a red-black tree for efficiency. Using the red-black tree, large numbers of timers can be active without affecting the performance of the timer subsystem (*O(log n)*).

With a little extra work, you can get hard real-time support with the PREEMPT_RT patch. The PREEMPT_RT patch provides several modifications to yield hard real-time support. Some of the changes include reimplementing some of the kernel locking primitives to be fully preemptable, implementing priority inheritance for in-kernel mutexes, and converting interrupt handlers into kernel threads so that they are fully preemptable.

Summary

Linux is not only a perfect platform for experimentation and characterization of real-time algorithms, you can also find real time in Linux today in the standard off-the-shelf 2.6 kernel. You can get soft real-time performance from the standard kernel or, with a little more work (kernel patch), you can build hard real-time applications.

This article gave a brief overview of some of the techniques used to bring real-time computing to the Linux kernel. Numerous early attempts used a thin-kernel approach to segregate the real-time tasks from the standard kernel. Later, nano-kernel approaches came on the scene that appear very much like the hypervisors used in virtualization solutions today. Finally, the Linux kernel provides its own means for real time, both soft and hard.

Although this article has skimmed the top of the real-time methods for Linux, the Resources section provides more information on where to get additional information and other useful real-time techniques.

Resources

Learn

- In Tim's article "Virtual Linux" (developerWorks, December 2006), learn how the thin-kernel and nano-kernel approaches to real time borrow from the virtualization architectures so popular and useful today.
- In Tim's article "Inside the Linux scheduler" (developerWorks, June 2006), read more about the O(1) scheduler in the 2.6 kernel.
- Read all of Tim's *Anatomy of...* articles on developerWorks.
- Read all of Tim's Linux articles on developerWorks.
- The Realtime Computing FAQ is a great place to start learning about real-time computing. It covers the basics of Portable Operating System Interface (POSIX), real-time operating systems, and pointers for real-time analysis.
- "Real-time Control Systems" (PDF) is an interesting tutorial by A. Gambier that introduces the design of real-time control systems and discusses various scheduling algorithms.
- The proceedings of the 6th Real-time Linux Workshop (held November 2007) include papers and resources to help you explore the cutting edge of real-time Linux.
- The early RTLinux, the RTAI, and Xenomai have all applied the thin-kernel approach. Though the application was slightly different for each, the overall approach has proven valuable.
- ADEOS is a hardware abstraction layer (HAL), or nano-kernel, that provides real-time capabilities in the Linux kernel. It has also been used for symmetric multiprocessor (SMP) clustering and kernel debugging.
- "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior" describes a component that provides guaranteed access to time-multiplexed resources (such as the CPU, network, or disk) through a reservation API.
- Red-black tree is a self-balancing binary search tree. Although the algorithm itself is complex, it is extremely efficient in practice and operates in O(log n) time.
- The PREEMPT_RT patch provides hard real-time capabilities in a standard 2.6 Linux kernel. These pages provide details on installing and using the RT_PREEMPT configuration as well as a useful FAQ.
- In this priority inversion scenario, a lower-priority task holds a resource that a higher-priority task requires. (This problem occurred on Mars in the Pathfinder

probe using Wind River's VxWorks.) The solution to this problem is priority inheritance, which increases the priority of a lower-priority task to allow it to run and release the resource.

- Novell recently announced their SUSE Linux Enterprise Real-Time (SLERT) version 1.0. This distribution contains kernel updates and POSIX real-time support for real-time applications. Two advancements include CPU shielding and on-the-fly priority assignment.
- Xenomai/SOLO announced migration away from the co-kernel approach to a tighter integration with the Linux kernel layer. This approach supports traditional POSIX (user-space) programming.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the developerWorks community through blogs, forums, podcasts, and community topics in our new developerWorks spaces.

About the author

M. Tim Jones
M. Tim Jones is an embedded software engineer and the author of *Artificial Intelligence: A Systems Approach, GNU/Linux Application Programming* (now in its second edition), *AI Application Programming* (in its second edition), and *BSD Sockets Programming from a Multilanguage Perspective.* His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Consultant Engineer for Emulex Corp. in Longmont, Colorado.

Trademarks