

## Linux for Embedded Systems?

Linux is designed for desktop-like systems; using it in embedded systems carries risks.

---

David N. Kleidermacher  
Green Hills Software

---

In recent years, the Linux operating system has achieved success in the desktop and server computing arena that has traditionally been the mainstay of proprietary operating systems from Microsoft (Windows), Sun Microsystems (SunOS/Solaris) and others. The move to Linux can be attributed, at least in part, to lower cost and higher performance. Today's top-end Pentium class processors run Linux and far outperform similarly priced SPARC-based systems running Solaris. Another major reason for success is the open source nature of Linux (unlike Solaris or Microsoft Windows): users benefit from the collective enhancements of the worldwide Linux community.

Interestingly, vendors of Linux for use in desktop and server computing have mostly defocused from the embedded systems industry. Other vendors, such as Lineo and MontaVista, who have focused on embedded systems, have yet to achieve sustained profitability. What is it about the embedded systems industry that has prevented Linux from attaining the success it has reached on the desktop? The reason lies in the difference in requirements between desktop and embedded systems.

These differences are many and they are significant. We will examine a few of them, not to be exhaustive, but to show examples of challenges faced by embedded systems that desktops do not encounter. Embedded systems requirements differ in the areas of:

- Interrupt latency
- Thread response time
- Scheduling
- Device drivers
- Memory footprint
- Reliability and security

Furthermore, there are a variety of important non-technical considerations that

interrupts while it performs such critical sequences of instructions. The major component of worst-case interrupt latency is the number and length (in terms of time to execute those instructions) of these sequences.

If an interrupt occurs during a period of time in which the operating system has disabled interrupts, the interrupt will remain pending until software re-enables interrupts (Figure 1). The length of time for which interrupts are disabled provides a worst-case upper bound for interrupt latency.

The importance of understanding the worst-case interrupt disabling sequence must not be understated. A real-time system utterly depends upon guaranteeing that the critical events in the system are handled within the required time frame. The reality, however, is that operating system vendors generally publish an *average, typical* or *best-case* interrupt latency, measured in a lab environment. Is it possible to statically compute the worst-case disabling region? A team of Swedish researchers (see <http://citeseer.nj.nec.com/carlsson02worstcase.html>) recently attempted to discover the answer to this question for one commercial real-time operating system in use today.

The case study employed some advanced methods of program flow analysis in an attempt to determine the location and structure of all the interrupt disabling regions. The researchers used cycle accurate models to determine execution counts of the selected regions. The case study took five months to complete.

The results are not encouraging. Because the microprocessor instruction used to disable and enable interrupts uses a variable register value as its source, it was often impossible to statically determine whether a given instruction enabled or disabled interrupts. Other problems of program flow, such as nested disabling/enabling sequences, hampered the study.

In the end, after five months, the researchers estimate that only approximately *half* of the disabling regions were identified. How many is that? Six hundred and twelve regions. In other words, another six hundred regions lurk in the system with an unknown impact on worst-case response time.

Finally, the researchers estimated the execution time of regions they could identify. Some of these regions had calls to out of line functions. A few regions even had triply nested loops. And some loops found in critical regions were of variable bound. The cycle count estimate for one of these nested loop regions was 26,729. On a 100 MHz microprocessor, that would translate into approximately 250 microseconds just for that one region. Rest assured that no real-time operating system vendor would claim an interrupt latency measurement of this magnitude. In contrast to embedded systems, desktop applications—where real-time reaction is not required—worst-case interrupt latency is irrelevant. Linux was designed for desktop use, not for real-time performance, so we can expect the results of a similar experiment on it to be even worse.

### Thread Response Time

Thread response time is defined as the elapsed time from interrupt to execution of the first instruction in a thread awakened to service the interrupt (Figure 2). This is also an important measurement in real-time systems since designers would prefer to place device manipulation code in threads where it is often easier to debug and to reduce the amount of code executing with interrupts disabled.

A significant problem involves the interaction between the thread responding to the high priority interrupt and other lower priority interrupts. Since interrupts are enabled while the high priority thread is executing, an unbounded number of low priority interrupts can occur, increasing the thread response

## The Softer Side

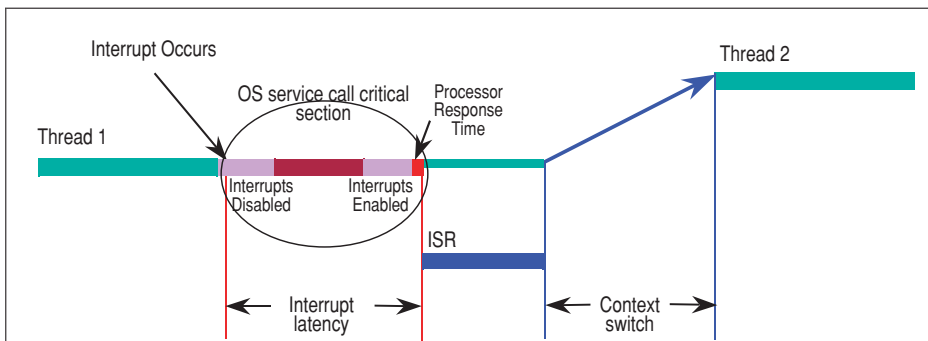


Figure 1

Interrupt latency is a critical factor in embedded systems, but it *must* take into account any OS decision to disable interrupts.

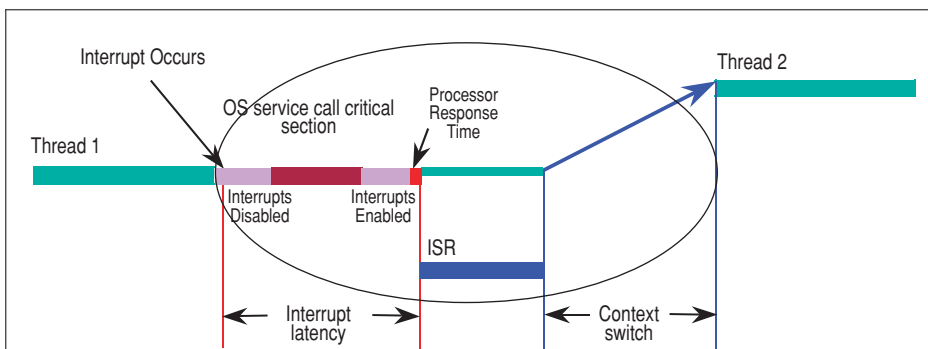


Figure 2

Thread response time takes into account the time from an interrupt to execution of the first instruction in an awakened thread that will service the interrupt.

but the real-time behavior is still not there. According to Red Hat (<http://people.redhat.com/drepper/nptl-design.pdf>), “Real time support is mostly missing from the library implementation. The system calls to select scheduling parameters are available but they have no effects. The reason for this is that large parts of the kernel do not follow the rules for real-time scheduling...and there are additional places where the kernel misses appropriate real-time support.”

In addition to long interrupt latency, Linux disables preemption or dispatching for very long periods of time (even using the “preemption patch” that is now standard in the 2.6 kernel). A recent article (<http://www.eet.com/sys/news/OEG20030908S0064>) notes a 2.6 kernel response time of 1.2 milliseconds, whereas a real-time embedded operating system can often achieve worst-case times in the sub-microsecond range (obviously all these numbers depend on the particular hardware configuration). Furthermore, the measured times for Linux do not reflect the theoretical worst-case times. No one truly knows what the worst-case response times are for Linux—that is unacceptable for real-time systems.

## Device Drivers

Device driver code adds less risk to the system if it runs in its own protected address space. An operating system architecture designed to facilitate *virtual device drivers* (Figure 3) is preferred over the traditional method of requiring device drivers to run in physical memory along with the kernel. This requires a flexible yet powerful and efficient API for providing the virtual device driver with secure access to the physical device resources it requires.

Linux lacks this architecture. In fact, despite offering protected virtual memory for applications, Linux promotes the addition of complicated device drivers into the physical kernel address space where it adds the most risk. The traditional method of adding a new Linux device driver is to compile the driver code into object files and then either link them into the kernel or load them dynamically via the *insmod* kernel module loader. These drivers have unfettered access to physical memory and are difficult to debug. Kernel-resident drivers are also the source of security attacks against monolithic kernels like Linux and Windows: a faulty device driver can overflow its run-time stack, polluting the kernel with

time as each interrupt service routine is executed. This is an example of “priority inversion,” since an essentially unbounded amount of low priority execution can delay the execution of higher priority work. In desktop systems, such inversions are unimportant; delays are usually not noticeable by the human user, and in a worst-case lock-up, one can simply re-boot.

A real-time operating system, though, should provide a method of preventing this kind of priority inversion. One solution is to enable device driver designers to prioritize interrupts below critical interrupt handling threads. When a thread is scheduled, the kernel inhibits the interrupts that are assigned a lower priority than the thread. When there is no higher priority thread to run, the lower priority interrupts are re-enabled; a simple yet effective solution.

## Scheduling

Linux has no such provision for prioritizing threads relative to interrupts; howev-

er, the actual situation is worse. Real-time operating systems provide priority-based scheduling because it must be possible to guarantee that the most critical threads in the system can run immediately in response to an event. It is forbidden to use heuristics or any other constructs in the kernel that might make this response nondeterministic.

The standard Linux scheduler is a *fairness*-based heuristic scheduler. This comes from Linux’s UNIX heritage as a time-sharing, interactive operating system. Thus, it is not possible for the designer to specify an absolute “highest” priority thread. When an interrupt handler makes a thread ready to run in order to process the event, the Linux scheduler is quite likely to choose some other thread to run first. It simply isn’t possible to determine the worst case thread response time.

The 2.6 version Linux kernel does have some scheduling performance improvements over earlier 2.4 vintage kernels,

foreign data that can then be executed as code to take control over the system.

The monolithic design of Linux also has repercussions in terms of system robustness, reliability and security (more on this later). A better architecture, found in a few embedded operating systems, is to employ a microkernel and push device drivers, networking stacks, file systems and other complicated software into application space where they cannot harm the kernel or other applications.

### Perceived to be Free

The hidden “royalty” due to Linux’s large footprint is just one example of the misconception that Linux is “free”. MontaVista, purveyors of the “Hard Hat” flavor of Linux, charges \$15,000 per developer per year to support its version of Linux. Over five years, the investment for a team of only five developers would be \$375,000. Many proprietary embedded operating system solutions are priced much lower than this. The total cost of using a commercial Linux distribution is often *more* expensive than a proprietary solution.

Many customers have avoided these costly Linux vendors, instead, choosing to download Linux from the Internet and get their own Linux “gurus” to handle all of the Linux support issues. Unfortunately, these organizations often underestimate the cost of supporting Linux internally. Some companies boast that they only need one full-time Linux “hacker” on staff. At a conservative estimate of \$150,000 per year for a fully burdened engineer, the five year cost of this approach is \$750,000, dwarfing the cost of most purchased proprietary solutions. And although the local guru may be able to handle myriad Linux support issues, clearly there is a limit to what can be accomplished without being the original author of the product.

Because of the high cost of supporting Linux relative to many proprietary solutions, many customers—who have attempted to use Linux only to discover this reality—have come to use the term “perceived to be free”. They have learned from difficult experience that perception sometimes falls short of reality.

### Long Term Support

Most embedded products have a shelf life of several years. An aircraft avionics system may be in development for as long as a

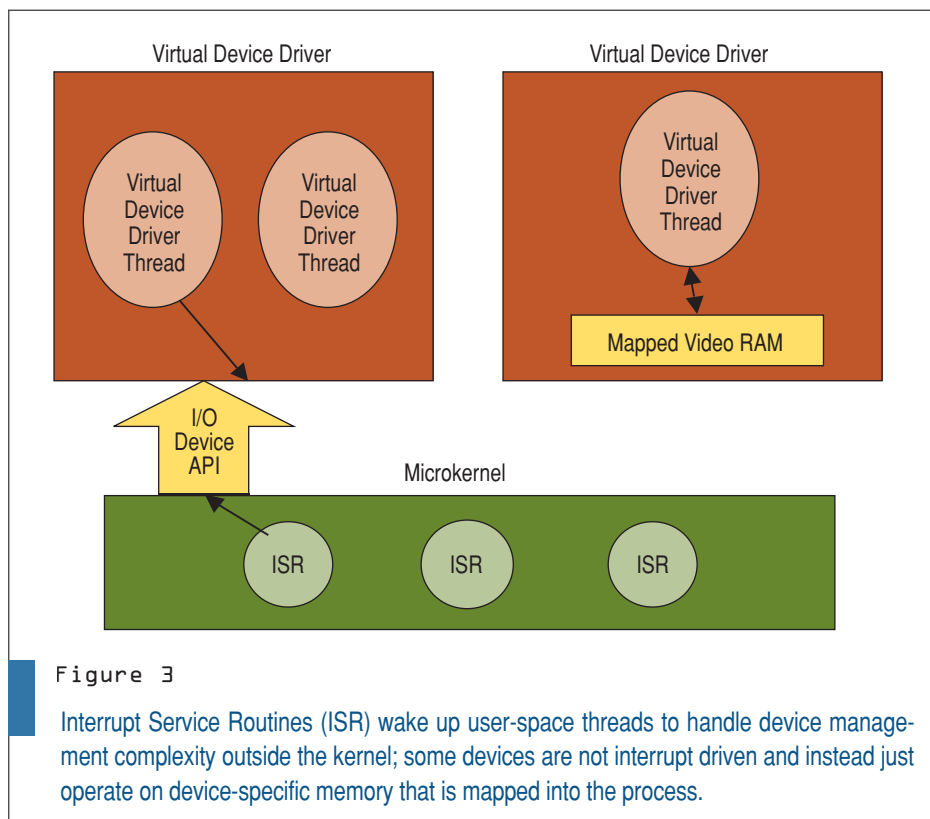


Figure 3

Interrupt Service Routines (ISR) wake up user-space threads to handle device management complexity outside the kernel; some devices are not interrupt driven and instead just operate on device-specific memory that is mapped into the process.

must be taken into account when thinking about using Linux in an embedded device.

### Interrupt Latency

By definition, a real-time system is one in which an event (for example, an interrupt from an important peripheral device) must be handled within a bounded (and typically short) amount of time; failure to respond causes a failure in the embedded system. Many embedded systems have real-time requirements; desktop systems do not. Along with context switch time, interrupt latency is the most often analyzed and benchmarked measurement for embedded real-time systems. Operating system architecture is the most significant factor for determining interrupt latency and thread response times in an embedded system.

Interrupt latency is defined as the elapsed time between an interrupt and the execution of the first instruction in the corresponding interrupt service routine. Interrupts are typically prioritized (by hardware) and nest; therefore, the latency of the highest priority interrupt is usually examined. How can software increase interrupt latency? By actually deferring interrupt processing during certain “critical” operating system operations. The operating system does this by “disabling”

decade before it is fielded, and then it might remain in service for much longer than that. Even the life of a handheld PDA may span multiple years from the start of development until obsolescence.

Embedded systems developers put a high value on a vendor that has a stable, successful, proven business model, because they must rely on that vendor to support their electronic products for such an extended period of time and to be around for the next generation of product development as well.

Many embedded Linux vendors have come to the realization that their support and services business models do not lend themselves to the long-term growth that investors demand. The reason is simple: support and services are manpower-intensive. The only way to increase revenues is to proportionally increase the engineering support staff. Most companies make money by engineering a new product and using relatively low overhead to sell that product over and over again.

Some of these vendors have realized this predicament and are now dabbling in proprietary extensions and other software “add-ons” that they can sell. Alas, this is a Catch-22 situation: by adding proprietary extensions, a Linux vendor positions itself as

## The Softer Side

just another proprietary solutions provider. The Linux community will not accept nor enhance proprietary extensions.

### The GPL

Linux is licensed under the GNU Public License (GPL), which governs how companies may incorporate the licensed software into the products that they sell to their customers. Much has been written about the pitfalls of incorporating GPL software into a product. An often overlooked consideration, however, are the costs of having to even *worry* about these licensing issues. The obvious concern is whether a proprietary product will be subject to the GPL because of its use of a GPL product.

The operating system is an important portion of the overall embedded system. Every single unit of the final electronic product will be shipped with a copy of the operating system under the hood. From the GPL Section 2b (emphasis added): “You must cause any work that you distribute or publish, that in whole or in part contains or is derived from [Linux] or any part thereof, **to be licensed as a whole at no charge to all third parties** under the terms of this License.” Companies cannot undertake the risk of incorporating a GPL product without the knowledge that their own intellectual property will not be forced into the public domain because of the use of software that follows the GPL.

Companies spend a lot of money researching the implications of using this GPL software. Some customers have legal teams dedicated to these software licensing issues alone. The legal costs of dealing with standard third-party licensing is comparatively low.

What some of these legal teams have discovered is that there is a lot of gray area with respect to the interpretation of the GPL. There are few cases to use for precedence. And influential people in the Linux community often disagree regarding the application of the GPL. A notable example is the Linux “loadable module”. As described earlier, Linux users can dynamically load modules directly into the kernel.

Some well-known members of the Linux community are adamant that loadable modules should be subjected to the GPL license. Yet Linus Torvalds, inventor of Linux,

has taken the opposite view, stating that loadable modules should not be subject to the GPL. Still others state that the applicability of the GPL depends on the manner in which the loadable module interacts with the kernel (for example, whether any data structures are shared). Most legal teams are not happy when there is no set rule for something like this: gray area implies risk.

A separate area of the GPL that poses challenges to companies attempting to incorporate open source software into their products is patent infringement and copyright. The GPL does not guarantee users that licensed code is free from copyright infringement, and it does not guarantee that code does not violate existing patents. In fact, it explicitly states that it makes no such assertions, leaving users on their own to face potential patent and copyright infringement penalties down the road. If this happens after thousands of units have been produced, sold and shipped, the manufacturer might face an incredibly complex recall and/or royalty fee demand from the copyright/patent holder.

### Open Source vs. Open Standard

Some people are under the false impression that Linux is an open standard. Open source software is freely available and modifiable. An open standard is a specification that is freely available for all to read and implement with the goal of increasing portability and avoiding getting locked into a particular vendor. There is no specification for Linux, and the fractured state of modified Linux distributions and vendors causes non-portability in some areas.

POSIX (Portable Operating System Interface) is an example of an open standard. In fact, it is a standard that describes application interfaces to operating systems. There are several proprietary operating system vendors that provide a POSIX-conformant interface for their products. This allows developers to use the proprietary operating system without getting locked into it; developers can take the same code and easily move it to another POSIX-conformant operating system.

For 30 years, UNIX operating systems have defined a common set of interfaces that are the basis for today’s POSIX standard (the latest standard, POSIX.1-2001, was recently updated and released in 2003). One would think that with its UNIX heritage, Linux too would be POSIX conformant. Not so.

Although Linux has a POSIX-like interface, it is not conformant in several key areas (such as scheduling, described earlier, signals, and thread implementations). Even the latest release of the Linux threads library (called the NPTL, Native POSIX Thread Library for Linux) is not POSIX-conformant. Developers writing portable POSIX applications may find that their code does not work on Linux and vice-versa.

These differences may be subtle or catastrophic. One known problem is in the implementation of the *setuid* system call. POSIX requires that a *setuid* invocation apply to all threads in the process, but on Linux, only the calling thread is affected. POSIX programs are sometimes created (for example, by an authorized system administrator) with “super-user” privileges, which give them complete system access, including the ability to delete any user’s files on the computer’s hard drive. Sometimes these programs will programmatically (using the *setuid* call) lower their own privilege for security reasons after executing an initial trusted super-user activity.

A POSIX-conformant program running on Linux will find that any threads created prior to the *setuid* call still have super-user privilege after the call. An erroneous (or worse, subverted) program that is unable to do any harm on a POSIX-conformant operating system now has the ability to do very bad things on a Linux system.

One might also think that the developers of an open source operating system might also be in favor of open operating system standards. Linus Torvalds has in fact spoken out against POSIX (<http://www.uwsg.iu.edu/hypermil/linux/kernel/0208.0/0427.html>), saying “POSIX is a hobbled standard, and does not matter. We’re not making a POSIX-compliant OS.” Will Linux ever be POSIX-conformant? According to Linus, “The fact is, there is...zero... reason to change existing functionality.” ■■

Green Hills Software  
Santa Barbara, CA.  
(805) 965-6044.  
[www.ghs.com].