

Running Linux Applications in an Embedded, Real-Time Environment

Following its success in the server market, Linux is generating a lot of interest as the new OS of choice in embedded and real-time devices. However, because the Linux kernel was not originally designed for these types of applications, it faces a number of interesting technical challenges that must be overcome if it is to gain widespread acceptance in this market. This paper starts with a discussion of the architectural features of the Linux kernel, with particular emphasis on showing where there is a good match with the requirements of the embedded market and where problems exist. The paper then goes on the present a comparison of the different approaches that have been used to adapt Linux for real-time and embedded systems.

INTRODUCTION

Linux is the rising star among operating systems. Mergers, IPOs and surging growth have catapulted Linux from the comfortable realms of Universities and research labs into the bright lights of the business world at breakneck speed. Windows assailant, Unix saviour, the reasons for Linux success in the server market are as many as they are diverse. From a purely technical point of view, Linux shares many of the strengths of Unix, where it has its roots - industry standard application programming interfaces (APIs), a rich feature set, robustness and stability through its use of memory protection, fully integrated networking and a wealth of application developers. In addition, the open source movement has allowed Linux to avoid the diversification that was the bane of Unix. More recently, the (usually staid) real-time, embedded systems community has started to buzz with the same excitement. On reflection, this is no big surprise. The benefits of Open Systems are well known to embedded developers. Many attempts have been made during the past several decades to adapt general purpose operating systems such as Unix and Windows to the special needs of this market. But none ever really managed to attain the critical mass needed to prevail over the fragmentation that has dominated the market for embedded real-time operating systems (RTOS). Can Linux succeed where others have failed? It does have one, additional advantage that makes it extremely attractive to the developers of embedded systems that are going into tomorrow's high-volume, consumer products such as handheld devices, internet access devices or automobiles. And that is its very low run-time cost.

EMBEDDED SYSTEMS

An embedded system is a computer system that is integrated into a larger system which is not, itself, a general purpose computer. So, generally, the embedded computer is not apparent to the user. Our lives are run by embedded systems, we encounter and use them by the hundreds every day. They are in car, the portable phone, the washing machine, the office photocopier, the aircraft's navigation system, the cash dis-

penser, the list goes on endlessly. One thing that is immediately obvious is that no two embedded systems look alike. How much more different can you get than a personal organizer and a cruise missile? This is in stark contrast to the desktop computer market where hardware and software compatibility is the name of the game. The diverse nature of embedded systems is, in large part, responsible for the plethora of RTOS's that are used to run them.

Linux Features/Benefits for Embedded Market

- Open Systems Approach
 - Software reuse
 - Lower training and maintenance cost
- Royalty Free
 - Lower production cost
 - Suppliers focus on service
- Reliability
 - Fully integrated memory protection
- Broad Support
 - 3rd Party Applications
 - Device Drivers
- Open Source
 - Can be customised

Note 1. Linux Features/Benefits for Embedded Market

REAL-TIME

Many embedded systems also have a need for real-time. Telephone switches, avionics systems, medical instruments, robot control, flight simulators are a few examples. What differentiates these systems is the need to be able respond to timed or external events within a fast, predictable period of time, every time. Failure to do so may lead to inconvenience for the user (you can't get the cash from the ATM), economic consequences (the factory production line goes down), or, in the most critical cases, even loss of human life (an

aircraft crash). These real-time requirements are usually characterized as scaling from "soft" (less critical) to "hard" (most critical). What makes or breaks a real-time system is the underlying operating system. It is the OS that is responsible for handling hardware interrupts and for deciding how and when the various tasks in the system execute. So, given the nature of the embedded market, its extreme diversity and its need for real-time, the challenge facing Linux is how an operating system designed for general purpose, desktop computing can be made to fit these totally different requirements.

LINUX

The Linux Operating System consists of many parts. There are a large number of utilities that are included as part of Linux. These utilities provide Linux with its user interface, networking, system administration, software development, and system monitoring. Despite the label of monolithic, which is sometimes given to the Linux kernel, the kernel is but one small part of the full Linux environment. It is, however, the kernel that binds all the other parts of Linux together; it is interface between all programs and the hardware, and it provides the low level multitasking, the interrupt processing and the scheduling, which determines the order in which things will be done.

Scheduling

The way in which these responsibilities are implemented in the standard Linux kernel make it unsuitable for many embedded, real-time systems. Firstly, the scheduler, which decides when each task will run, and for how long. The Linux scheduler was not designed to deal with tasks that have hard timing deadlines. It was designed for multi-user, time-sharing work and endeavors to be fair with its task scheduling while striving for maximum total system throughput. Although Linux does provide a class of "real-time" processes, which are always scheduled before normal processes, the mechanism can at best achieve only soft real-time. Other activities performed by the kernel, namely some task queues and the so-called bottom half handlers, are always scheduled before even the highest priority real-time user process. So, even if, from a particular application's point of view, a bottom half handler is not performing a critical activity, it will nevertheless be scheduled before any real-time user processes.

Preemption

Related to scheduling is the concept of preemption. To provide multitasking, the scheduler must be able to interrupt (pre-empt) the currently executing task and run another task it considers to be of higher priority. Under Linux, an application task can always be preempted while it is running user code. But within the kernel, there are large stretches of non-preemptible code. Once a user process makes a system call and enters one of these regions, it can not be preempted until it exits the region. Meanwhile, another, higher priority process could be blocked, waiting to run. This could be disastrous for a task with critical timing constraints.

A popular approach to improving the real-time perfor-

mance of non-preemptive kernels is the addition of preemption points. It involves the least amount of change to the kernel and can be sufficient for some (softer) real-time applications. When reaching a preemption point, the kernel checks to see if a higher priority task is waiting to run. If there is, it switches execution to that task immediately. The worst case preemption latency is now the longest time between two preemption points. To obtain a reasonable latency may require the addition of a large number of preemption points, which has the undesirable side-effect of adding overhead to the kernel's execution. Even with this technique, however, the worst case preemption time may still be unbounded because of the way certain kernel data structures are implemented. When the kernel manipulates one of its internal data structures, (for example, the process table), the action must be atomic, that is to say, preemption is not possible part way through. Otherwise, the data structure could end up in some incoherent state. If the data structure is designed as some type of list, access time, and hence preemption latency, would vary as a function of the number of items (for example, tasks) on the list. The only really effective solution for real-time is to design a kernel that is fully preemptible. This means that by default the kernel is always preemptible. Critical code regions, which access shared data structures use a suitable synchronization mechanism such as semaphores or disabling preemption. The data structures themselves must also be designed for fast and deterministic access. For example, the scheduler should be implemented in such a way that the time to select a task to run and to perform a context switch is constant, independent of the number of tasks in the system.

Problems with Linux Kernel

For real-time embedded systems :

- Real-Time Performance
- Scalability
- Real-Time APIs
- CPU support

Note 2. Problems with Linux Kernel

Interrupt handling

The way in which a kernel handles hardware interrupts is probably the most critical factor contributing to the determinism and overall responsiveness of a real-time system. With the evolution of embedded systems to integrate additional I/O devices such as networking and graphical user interfaces, this fact is ever increasingly true. To the point that even some traditional RTOS architectures may no longer be appropriate.

The Linux kernel uses a combination of device driver interrupt handlers and bottom half handlers to perform interrupt processing. The concept of off-loading interrupt processing from the interrupt handler to a separately scheduled task (the bottom half handler) is not

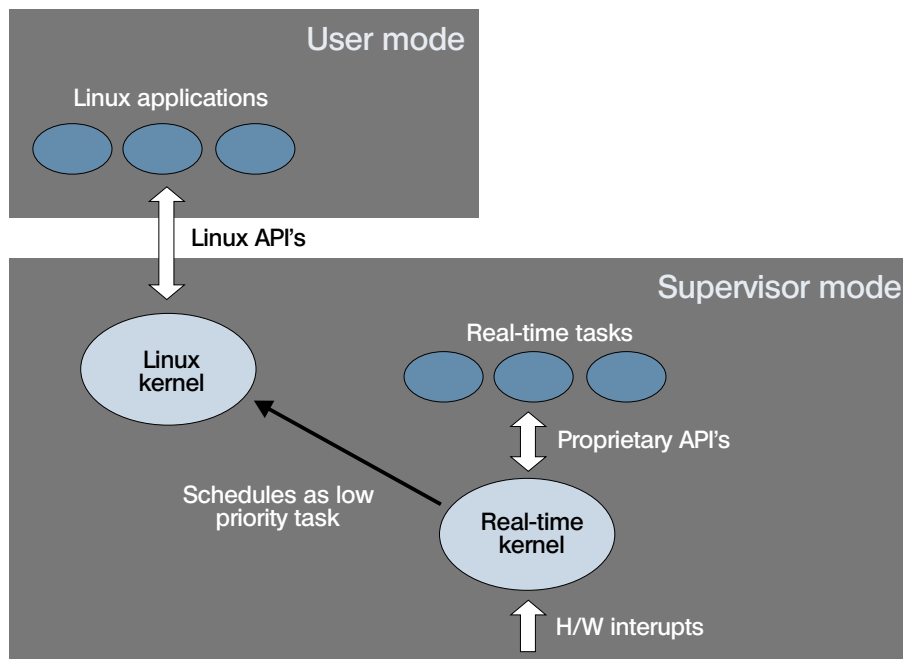


Figure 1. Kernel co-habitation architecture

unique to Linux and can be found in many general purpose and real-time OS's today. By shortening the interrupt handlers, the system is able to respond to subsequent interrupts sooner, thus reducing the worst case interrupt latency. However, the Linux bottom half handlers are scheduled before all user tasks. So, all interrupt processing under a Linux kernel effectively runs at a higher priority than all user tasks, including real-time processes. For the time-critical user application that is blocked from running, there is little benefit in the fact that the work is being done in a bottom half handler rather than the interrupt handler. And if the interrupt is associated with a network running TCP/IP or a serial terminal, for example, the application could find itself waiting for a significant period of time. To make matters worse, the number of interrupts generated by any single device is not bounded. Applications could, in certain scenarios, be delayed for extremely long, unpredictable periods of time while these interrupts are being serviced.

RTLinux

One approach to adapting Linux for embedded real-time systems is represented by RTLinux. Rather than address the daunting task of making the Linux kernel real-time, the RTLinux approach is one of kernel co-existence, with a small real-time executive running alongside the Linux kernel. In fact, Linux is executed as a low-priority task by the real-time executive. The real-time executive is responsible for handling hardware interrupts and for scheduling real-time tasks. DIAPM-RTAI is a derivative of RTLinux but uses essentially the same architecture. While this technique can obtain very impressive real-time performance for the time-critical tasks, it does so at the expense of some of the major strengths of Linux. Real-time tasks that run under

the executive are executed in supervisor mode. This means that they do not benefit from the Linux memory protection model. A bug in the user's code could corrupt another part of the system or even bring the whole system down. For the development of real-time applications, the executive provides a proprietary API along with a small subset of the standard Linux APIs. Compared to the rich set of services offered by the Linux kernel, the real-time executive is fairly limited in functionality and because of its proprietary nature, code must be ported.

Despite these drawbacks, RTLinux is well suited to a certain class of system. These are systems that have the more traditional characteristics of a real-time embedded system - relatively small, simple and with a clear distinction between the real-time and non-real-time functions. However, the trends taking place in the embedded, real-time market mean that this traditional model of a real-time system are no longer appropriate for an increasingly large number of applications. Product value and differentiation is increasingly being provided through software. Each new product generation adds more features and services. All the while, software development cycles are being pushed down, making the ability to integrate third party off-the-shelf software highly valuable. Portability and reuse of code are a necessity. As systems become more complex and feature rich, the system reliability provided through memory protection becomes another key element. The distinction between real-time and non-real-time functions in a system is often far from obvious. Take, for example, network connectivity, which is becoming almost a standard feature in embedded systems. This service will most likely be required by both the real-time and the non real-time parts of the application, which does not fit well with the kernel co-existence

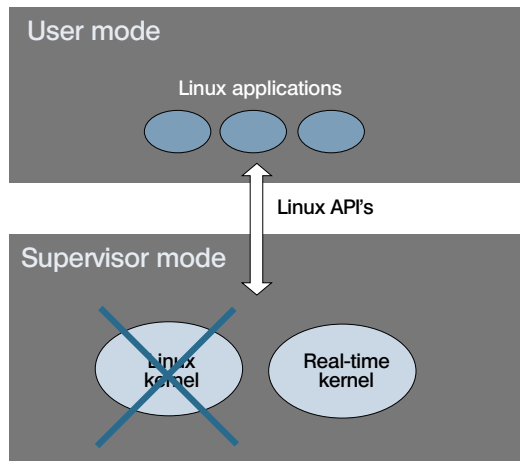


Figure 1. Kernel replacement

model where something is either real-time or non real-time, but not both. The software, including the operating system, must be flexible enough to handle these dynamic requirements. In short, the key benefits of Linux are becoming more and more essential for the success of many embedded real-time systems and can not be sacrificed.

LynxOS

To address these market needs, the approach adopted by Lynx Real-Time Systems Inc. is to replace the Linux kernel with a completely different kernel that meets the needs of real-time applications but can also run all the applications, tools, and interfaces that normally run as part of Linux, including the wealth of third party software. The kernel that was designed for this purpose is the LynxOS kernel. The goals were to design a real-time operating system from the ground up to support hard real-time applications but give it an industry standard interface, namely Unix (including Linux) and POSIX. Other goals included expanding the Unix I/O system, portability across CPU architectures, robustness through memory protection, ROMability and scalability.

The model for most real-time applications is that of multiple tasks, each with its own response needs. LynxOS supports these applications by providing fixed-priority preemptive scheduling, allowing task preemption even in the kernel. LynxOS goes even further by executing extended asynchronous interrupt processing at user task priority levels. The worst case preemption delay and blocking times are known for the kernel and can be used in conjunction with task execution times to ensure that independent tasks will always meet their timing deadlines.

The Linux kernel contains many functions. It provides for a hierarchical file system, process creation, program loading, task control via signals, networking, etc. All of these functions were given to the LynxOS kernel. By providing APIs that are POSIX and Linux compatible, applications are completely portable between the Linux and LynxOS environments. There is no need to develop real-time tasks using a proprietary API. Third-party software can easily be integrated into an appli-

cation and will transparently benefit from the real-time characteristics of LynxOS.

But LynxOS implements the Linux APIs using very different underlying mechanisms due to the fact LynxOS was independently designed with real-time constraints being the driving force. The LynxOS kernel was designed to be fully preemptive without adding long blocking regions. A number of techniques are available for protecting shared data structures used in the kernel - semaphores, disabling preemption and disabling interrupts. To ensure that preemption and blocking delays are minimized, LynxOS data structures were built for very fast, deterministic access. To provide a highly reliable execution environment, LynxOS makes full use of the memory management hardware. All application tasks are protected from each other and the kernel itself is protected user code. An errant task can not corrupt or crash the system. It will simply generate an exception, which can be handled in a clean manner while the rest of the system continues to operate.

LynxOS has a unique interrupt handling architecture to address the problem of interrupt processing discussed earlier. LynxOS executes the bulk of interrupt servicing at a user task priority level through the use of dedicated kernel threads. A kernel thread's priority is set based on the priority of the user task that is using the I/O device. If the user task changes its priority, then the kernel thread's priority is adjusted correspondingly. This dynamic "priority tracking" is very different from the static priority model of Linux bottom half handlers and is key to building a predictable system, even in the presence of the multitude of interrupt sources typical of today's embedded system architectures.

Another feature of the LynxOS kernel that is important for embedded systems is its scalability. Many embedded system have limited memory resources and require booting and execution from Flash. Although Linux is to some extent configurable, its memory footprint is just too large for many applications. Through its highly modular design, LynxOS can be configured to contain just those functions required by an application, everything else being removed. Other memory saving techniques provided by LynxOS include shared libraries, shared text and the ability to run the kernel, applications and a filesystem directly out of Flash. So, LynxOS can scale down to address the needs of very small embedded system but at the same time, thanks to the deterministic design of its internal data structures and the use of memory protection, it scales up very well, without loss of performance, to very large, complex systems. And Linux applications can run directly on all of these configurations.

CONCLUSION

The real problem of the fragmented RTOS market is application portability. Given a standardised API, what does it really matter to applications what the underlying RTOS is? Performance, of course. But in this respect, most RTOS's are not that different from one another. In terms of APIs, they are worlds apart. Given the heterogeneous nature of embedded systems, it is

unlikely that only a few RTOS's will come to dominate the market. Would this even be desirable, since no one size fits all? So trying to adapt the Linux kernel itself to meet all these diverse needs is probably not a worthwhile objective exercise. However, if you remove the Linux kernel, what remains are the APIs. Application portability is an extremely valuable and probably attainable goal. It is the Linux APIs, rather than the Linux kernel itself, that holds great promise for the embedded real-time market. One strategy to achieve this goal is to replace the Linux kernel with a functionally equivalent real-time kernel such as LynxOS ■

Chris Clark is Regional Systems Manager for Europe, Middle East and Africa at Lynx Real-Time Systems. He has been working in the embedded RTOS market for 15 years, 8 of those with LynuxWorks (formerly Lynx Real-Time Systems), in a variety of support, marketing and sales positions.

AD MPL