Exactly When Do You Need an RTOS?

Paul Leroux, Technology Analyst
QNX Software Systems
paull@qnx.com

Abstract

Together, the speed of today's high-performance processors and realtime patches for general-purpose OSs appear to have re-opened the question of whether embedded systems still need an RTOS. The answer hasn't changed: the guarantees only a true RTOS can offer on relatively low-end processors mean that these OSs are here to stay in embedded environments.

Introduction

Do most embedded projects still need an RTOS? It's a good question, given the speed of today's high-performance processors and the availability of realtime patches for Linux, Windows, and other general-purpose operating systems (GPOSs).

The answer lies in the very nature of embedded devices. Devices that are often manufactured in the thousands, or even millions, of units. Devices produced on a scale where even a $1 reduction in per-unit hardware costs can save the manufacturer a small fortune. Devices, in other words, that can't afford the cost (not to mention the heat dissipation) of multi-gigahertz processors.

In the automotive telematics market, for instance, the typical 32-bit processor runs at about 600 MHz — far slower than the processors common in desktops and servers. In an environment such as this, an RTOS designed to extract extremely fast, predictable response times from lower-end hardware offers a serious economic advantage.

Cost savings aside, the services provided by an RTOS make many computing problems easier to solve, particularly when multiple activities compete for a system's resources. Consider, for instance, a system where users expect (or need) immediate response to input. With an RTOS, a developer can guarantee that operations initiated by the user will execute in preference to other system activities, unless a more important activity (for instance, an operation that helps protect the user's safety) must execute first.

Consider also a system that must satisfy quality of service (QoS) requirements, such as a device that presents live video. If the device depends on software for any part of its content delivery, it can experience dropped frames at a rate that users perceive as unacceptable—from the users' perspective, the device is unreliable. With an RTOS, however, the developer can precisely control the order in which software processes execute and ensure that playback occurs at an appropriate and consistent media rate.

RTOSs Aren't "Fair"

The need for "hard" real time—and for the RTOSs that enable it—remains prevalent in the embedded industry. The question is: what does an RTOS have that a GPOS doesn't? And, how useful are the realtime extensions now available for some GPOSs? Can they provide a reasonable facsimile of RTOS performance?

Let's begin with task scheduling. In a GPOS, the scheduler typically uses a "fairness" policy to dispatch threads and processes onto the CPU. Such a policy enables the high overall throughput required by desktop and server applications, but it offers no assurances that high-priority, time-critical threads will execute in preference to lower-priority threads.

For instance, a GPOS may decay the priority assigned to a high-priority thread, or otherwise dynamically adjust the thread's priority in the interest of fairness to other threads in the system. A high-priority thread can, as a consequence, be preempted by threads of lower priority. In addition, most GPOSs have unbounded dispatch latencies: the more threads in the system, the longer it takes for the GPOS to schedule a thread for execution. Any one of these factors can cause a high-priority thread to miss its deadlines, even on a fast CPU.

In an RTOS, on the other hand, threads execute in order of their priority. If a high-priority thread becomes ready to run, it can, within a small and bounded time interval, take over the CPU from any

lower-priority thread that may be executing. Moreover, the high-priority thread can run uninterrupted until it has finished what it needs to do—unless, of course, it is pre-empted by an even higher-priority thread. This approach, known as priority-based preemptive scheduling, allows high-priority threads to meet their deadlines consistently, even when many other threads are competing for CPU time.

Preemptible Kernel

In most GPOSs, the OS kernel isn't preemptible. Consequently, a high-priority user thread can never preempt a kernel call, but must instead wait for the entire call to complete — even if the call was invoked by the lowest-priority process in the system. Moreover, all priority information is usually lost when a driver or other system service, usually performed in a kernel call, exe-cutes on behalf of a client thread. Such behavior causes unpredictable delays and prevents critical activities from completing on time.

In an RTOS, on the other hand, kernel operations are preemptible. As in a GPOS, there are time windows during which preemption may not occur, though in a well-designed RTOS, these windows are extremely brief, often in the order of hundreds of nanoseconds. Moreover, the RTOS imposes an upper bound on how long preemption is held off and interrupts disabled; this upper bound allows developers to ascertain worst-case latencies.

To realize this goal of consistent predictability and timely completion of critical activities, the RTOS kernel must be simple and elegant as possible. The best way to achieve this simplicity is to design a kernel that includes only services with a short execution path. By excluding work-intensive operations (such as process loading) from the kernel and assigning them to external processes or threads, the RTOS designer can help ensure that there is an upper bound on the longest non-preemptible code path through the kernel.

In a few GPOSs, some degree of preemptibility has been added to the kernel. However, the intervals during which preemption may not occur are still much longer than those in a typical RTOS; the length of any such preemption interval will depend on the longest critical section of any modules (for instance, networking) incorporated into the GPOS kernel. Moreover, a preemptible GPOS kernel doesn't address other conditions that can impose unbounded latencies, such as the loss of priority information that occurs when a client invokes a driver or other system service.

Avoiding Priority Inversion

In a GPOS, and even in an RTOS, a lower-priority thread can inadvertently prevent a higher-priority thread from accessing the CPU—a condition known as priority inversion. When an unbounded priority inversion occurs, critical deadlines can be missed, resulting in outcomes that range from unusual system behavior to outright failure. Unfortunately, priority inversion is often overlooked during system design. Many examples of priority inversion exist, including one that plagued the Mars Pathfinder project in July 1997.[1]

Generally speaking, priority inversion occurs when two tasks of differing priority share a resource, and the higher-priority task cannot obtain the resource from the lower-priority task. To prevent this condition from exceeding a bounded interval of time, an RTOS may provide a choice of mechanisms unavailable in a GPOS, including priority inheritance and priority ceiling emulation. We couldn't possibly do justice to both mechanisms here, so let's focus on an example of priority inheritance.

To begin, we must consider how task synchronization can result in blocking, and how this blocking can, in turn, cause priority inversion. Let's say two jobs are running, Job 1 and Job 2, and that Job 1 has the higher priority. If Job 1 is ready to execute, but must wait for Job 2 to complete an activity, we have blocking. This blocking may occur because of synchronization; for instance, Job 1 and Job 2 share a resource controlled by a lock or semaphore, and Job 1 is waiting for Job 2 to unlock the resource. Or, it may occur because Job 1 is requesting a service currently used by Job 2.

The blocking allows Job 2 to run until the condition that Job 1 is waiting for occurs (for instance, Job 2 unlocks the resource that both jobs share). At that point, Job 1 gets to execute. The total time that Job 1 must wait is known as the blocking factor. If Job 1 is to meet any of its timeliness constraints, this blocking factor can't vary according to any parameter, such as the number of threads or an input into the system. In other words, the blocking factor must be bounded.

[1] Barr, Michael. "Introduction to Priority Inversion," *Embedded Systems Programming,* Volume 15: Number 4, April 2002.

Now let's introduce a third job—Job 3—that has a higher priority than Job 2 but a lower priority than Job 1 (see Figure 1). If Job 3 becomes ready to run while Job 2 is executing, it will preempt Job 2, and Job 2 won't be able to run again until Job 3 blocks or completes. This new job will, of course, increase the blocking factor of Job 1; that is, it will further delay Job 1 from executing. The total delay introduced by the preemption is a priority inversion.

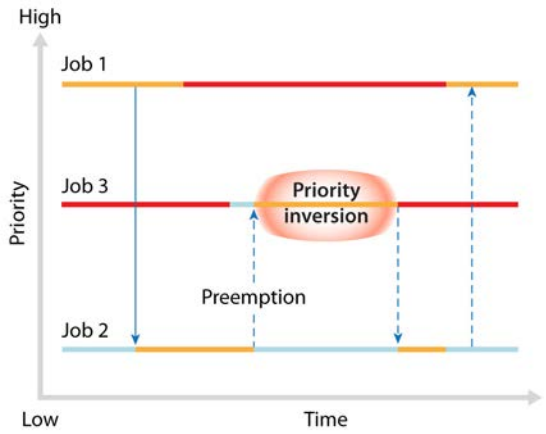

Figure 1. Job 1 is waiting for Job 2 to complete an activity, when Job 3 preempts Job 2. This new job further delays Job 1 from executing

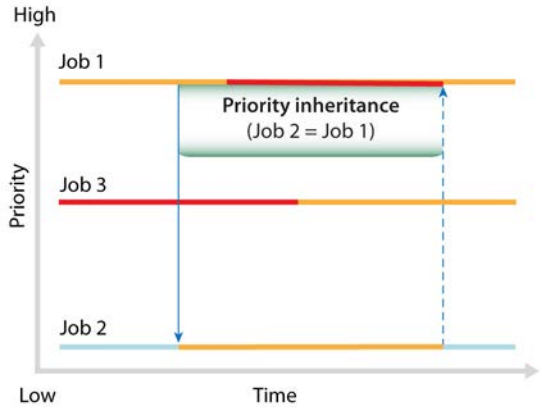

Figure 2. Job 2 inherits Job 1's higher priority, thereby preventing Job 3 from preempting Job 2. Job 3 no longer delays Job 1 from executing.

In fact, multiple jobs can preempt Job 2 in this way, resulting in an effect known as chain blocking. Under such circumstances, Job 2 might be preempted for an indefinite period of time, yielding an unbounded priority inversion and causing Job 1 to fail to meet any of its deadlines.

This is where priority inheritance comes in. If we return to our scenario and make Job 2 run at the priority of Job 1 during the synchronization period, then Job 3 won't be able to preempt Job 2, and the resulting priority inversion is avoided (see Figure 2).

Partitioning Schedulers

For many systems, guaranteeing resource availability is critical. If a key subsystem is deprived of, say, CPU cycles, the services provided by that subsystem becomes unavailable to users. In a denial-of-service (DoS) attack, for instance, a malicious user could bombard a system with requests that need to be handled by a high-priority process. This process could then overload the CPU and starve other processes of CPU cycles, making the system unavailable to users.

A security breach isn't the only cause of process starvation. In many cases, adding software functionality to a system can push it "over the brink" and starve existing applications of CPU time. Applications or services that were functioning in a timely manner no longer respond as expected or required. Historically, the only solution to this problem was to either retrofit hardware or to recode (or redesign) software—both undesirable alternatives.

To address these problems, systems designers need a partitioning scheme that enforces CPU budgets, either through hardware or software, to prevent processes or threads from monopolizing CPU cycles needed by other processes or threads. Since an RTOS already provides centralized access to the CPU, memory, and other computing resources, an RTOS is an excellent candidate to enforce CPU partition budgets.

Some RTOSs offer a fixed partition scheduler. Using this scheduler, the system designer can divide tasks into groups, or partitions, and allocate a percentage of CPU time to each partition. With this approach, no task in any given partition can consume more than the partition's statically defined percentage of CPU time. For instance, let's say a partition is allocated 30% of the CPU. If a process in that partition subsequently becomes the target of a denial of service attack, it will consume no more than 30% of CPU time. This allocated limit allows other partitions to maintain their availability; for instance, it can ensure that the user interface (e.g. a remote terminal) remains accessible. As a result, operators can access the system and resolve the problem—without having to hit the reset switch.

Nonetheless, there is a problem with this approach. Because the scheduling algorithm is fixed, a partition can never use CPU cycles allocated to other partitions, even if those partitions haven't used their allotted cycles. This approach squanders CPU cycles and prevents the system from handling peak demands. Systems designers must, as a result, use more-expensive processors, tolerate a slower system, or restrict the amount of functionality that the system can support.

Adaptive partitioning

Another partitioning scheme, called adaptive partitioning, addresses the drawbacks of static partitions by providing a more dynamic scheduling algorithm. Like static partitioning, adaptive partitioning allows the system designer to reserve CPU cycles for a process or group of processes. The designer can thus guarantee that the load on one subsystem or partition won't affect the availability of other subsystems. Unlike static approaches, however, adaptive partitioning can dynamically reassign CPU cycles from partitions that aren't busy to partitions that can benefit from extra processing time—partition budgets are enforced only when the CPU is fully loaded. As a result, the system can handle peak demands and achieve 100% utilization, while still enjoying the benefits of resource guarantees.

Just as importantly, adaptive partitioning can be overlaid on top of an existing system without code redesign or modifications. In the QNX® Neutrino® RTOS, for example, a system designer can simply launch existing POSIX-based applications in partitions, and the RTOS scheduler ensures that each partition receives its allocated budget. Within each partition, each task continues to be scheduled according to the rules of priority-based preemptive scheduling—applications don't have to change their scheduling behavior. Moreover, the designer can dynamically reconfigure the partitions to fine-tune the system for optimal performance.

"Dualing" Kernels

GPOSs—including Linux, Windows, and various flavors of Unix—typically lack the realtime mechanisms discussed thus far. In an attempt to fill the gap, GPOS vendors have developed a number of realtime extensions and patches. There is, for example, the dual-kernel approach, in which the GPOS runs as a task on top of a dedicated realtime kernel (see Figure 4). All tasks that require deterministic scheduling run in this kernel, but at a higher priority than the GPOS. These tasks can thus preempt the GPOS whenever they need to execute, and yield the CPU to the GPOS only when their work is done.

Unfortunately, tasks running in the realtime kernel can make only limited use of existing system services in the GPOS — file systems, networking, and so on. In fact, if a realtime task calls out to the GPOS for *any* service, this task is subject to the same preemption

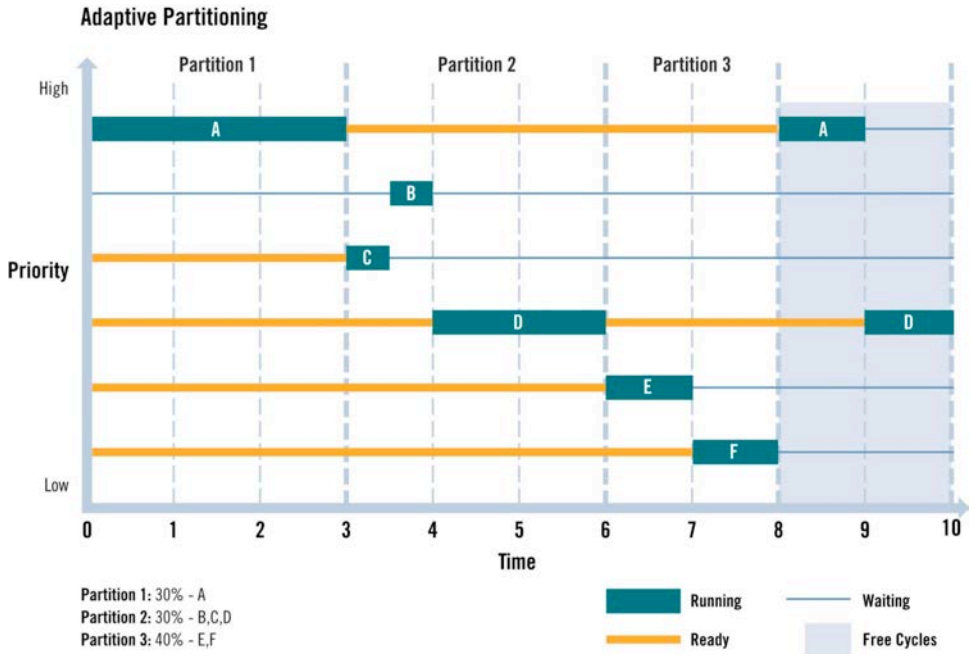

Figure 3. Adaptive partitioning prevents high-priority tasks from consuming more than their assigned CPU percentage, unless the system contains unused CPU cycles. For instance, tasks A and D can run in time allocated to Partition 3 because tasks E and F don't require the rest of their budgeted CPU cycles.

problems that prohibit GPOS processes from behaving deterministically. As a result, new drivers and system services must be created specifically for the realtime kernel, even when equivalent services already exist for the GPOS. Also, tasks running in the realtime kernel don't benefit from the robust MMU-protected environment that most GPOSs provide for regular, non-realtime processes. Instead, they run unprotected in kernel space. Consequently, a realtime task that contains a common coding error, such as a corrupt C pointer, can easily cause a fatal kernel fault. That's a problem, since most systems that need real time also demand a very high degree of reliability.

To complicate matters, different implementations of the dual-kernel approach use different APIs. In most cases, services written for the GPOS can't easily be ported to the realtime kernel, and tasks written for one vendor's realtime extensions may not run on another vendor's extensions.

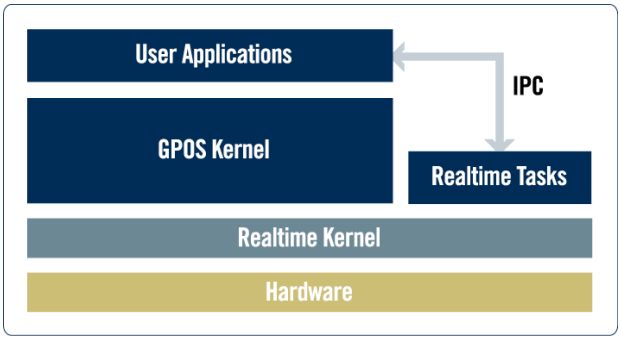

Figure 4. In a typical dual-kernel implementation, the GPOS runs as the lowest-priority task in a separate realtime kernel.

Such solutions point to the real difficulty, and immense scope, of making a GPOS capable of supporting realtime behavior. This isn't a matter of "RTOS good, GPOS bad," however. GPOSs such as Linux, Windows, and the various Unixes all function very well as desktop or server OSs. They fall short, however, when forced into deterministic environments that they weren't designed for—environments such as in-car telematics units, medical instruments, realtime control systems, and continuous media applications.

Extending the RTOS for Application-specific Requirements

Whatever their shortcomings in deterministic environments, there are, nonetheless, benefits to using GPOSs. These benefits include support for widely used APIs and, in the case of Linux, the open source model. With open source, a developer can customize OS components for application-specific demands and save considerable time troubleshooting. The RTOS vendor can't afford to ignore these benefits. Extensive support for POSIX APIs—the same APIs used by Linux and various flavors of Unix—is an important first step. So is providing well-documented source code and customization kits that address the specific needs and design challenges of embedded developers.

The architecture of the RTOS also comes into play. An RTOS based on a microkernel design, for instance, can make the job of OS customization fundamentally easier to achieve than with other architectures. In a microkernel RTOS, only a small core of fundamental OS services (for instance, signals, timers, scheduling) reside in the kernel itself. All other components—drivers, file systems, protocol stacks, applications—run outside the kernel as separate, memory-protected processes (see Figure 5). As a result, developing custom drivers and other application-specific OS extensions doesn't require specialized kernel debuggers or kernel gurus. In fact, as user-space programs, such extensions become as easy to develop as standard applications, since they can be debugged with standard source-level tools and techniques.

For instance, if a device driver attempts to access memory outside its process container, the OS can identify the process responsible, indicate the location of the fault, and create a process dump file viewable with source-level debugging tools. The dump file can include all the information the debugger needs to identify the source line that caused the problem, along with diagnostic information such as the contents of data items and a history of function calls.

Such an architecture also provides superior fault isolation and recovery: if a driver, protocol stack, or other system service fails, it can do so without corrupting other services, *or the OS kernel*. In fact, "software watchdogs" can continuously monitor for such events and restart the offending service dynamically, without resetting the entire system or involving the user in any way. Similarly, drivers and other services can be dynamically stopped, started, or upgraded, again without a system shutdown.

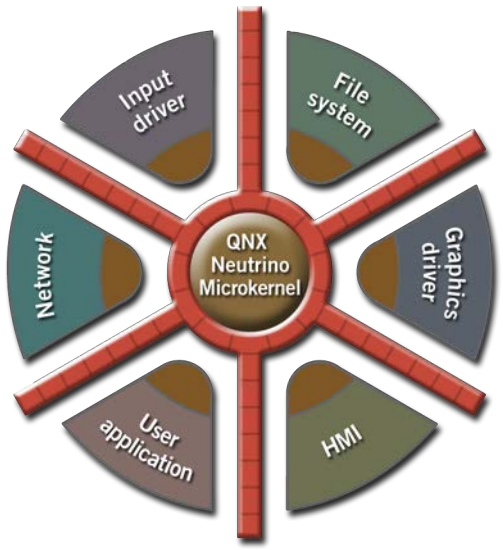

Figure 5. In a microkernel RTOS, system services run as standard, user-space processes, simplifying the task of OS customization.

These benefits shouldn't be taken lightly—the biggest disruption that can occur to realtime performance is an unscheduled system reboot! Even a scheduled reboot to incorporate software upgrades disrupts operation, though in a controlled manner. To ensure that deadlines are always met, developers must use an OS that can remain continuously available, even in the event of software faults or service upgrades.

A Strategic Decision

An RTOS can help make complex applications both predictable and reliable; in fact, the precise control over timing made possible by an RTOS adds a form of reliability that cannot
be achieved with a GPOS. (If a system based on a GPOS doesn't behave correctly due to incorrect timing behavior, then we can justifiably say that the system is unreliable.) Still, choosing the right RTOS can itself be a complex task. The underlying architecture of an RTOS is an important criterion, but so are other factors. These include:

- *Flexible choice of scheduling algorithms* — Does the RTOS support a choice of scheduling algorithms (FIFO, round robin, sporadic, etc.)? Can the developer assign algorithms on a per-thread basis, or does the RTOS force him into assigning one algorithm to all threads in the system?
- *Time partitioning* — Does the RTOS support time partitioning, which can provide processes with a guaranteed percentage of CPU cycles? Such guarantees simplify the job of integrating subsystems from multiple development teams or vendors. They can also ensure that critical tasks remain available and meet their deadlines, even when the system is subjected to denial of service (DoS) attacks and other malicious exploits.
- *Support for multi-core processors* — The ability to migrate to multi-core processors has become essential for a variety of high-performance designs. Does the RTOS support a choice of multi-processing models (symmetric multiprocessing, asymmetric multi-processing, bound multiprocessing) to help developers take best advantage of multi-core hardware? And is the RTOS supported by system-tracing tools that let developers diagnose and optimize the performance of a multi-core system? Without tools that can highlight resource contention, excessive thread migration, and other problems common to multi-core designs, optimizing a multi-core system can quickly become an onerous, time-consuming task.
- *Tools for remote diagnostics* — Because downtime is intolerable for many embedded systems, the RTOS vendor should provide diagnostics tools that can analyze a system's behavior without interrupting services that the system provides. Look for a vendor that offers runtime analysis tools for system profiling, application profiling, and memory analysis.
- *Open development platform* — Does the RTOS vendor provide a development environment based on an open platform like Eclipse, which permits developers to "plug in" their favorite third-party tools for modeling, version control, and so on? Or is the development environment based on proprietary technology?
- *Graphical user interfaces* — Does the RTOS use primitive graphics libraries or does it support multiple HMI technologies (HTML5, Qt, OpenGL ES, etc.) and provide advanced graphics capabilities such as multi-layer interfaces, multi-headed displays, accelerated 3D rendering, and a true windowing system? Can the look-and-feel of

GUIs be easily customized? Can the GUIs display and input multiple languages (Chinese, Korean, Japanese, English, Russian, etc.) simultaneously? Can 2D and 3D applications easily share the same screen?

- *Standard APIs* — Does the RTOS lock developers into a proprietary API, or does it provide certified support for standard APIs such as POSIX and OpenGL ES, which make it easier to port code to and from other environments? Also, does the RTOS offer comprehensive support for the API, or does it support only a small subset of the defined interfaces?
- *Middleware for digital media* — Flexible support for digital media is becoming a design requirement for an array of embedded systems, including car radios, medical devices, industrial control systems, media servers, and, of course, consumer electronics. A system may need to handle multiple media sources (device, streaming, etc.), understand multiple data formats, and support a variety of DRM schemes. By providing well-designed middleware for digital media, an RTOS vendor can eliminate the considerable software effort needed to connect to multiple media sources, organize the data, and initiate proper data-processing paths. Moreover, a well-designed middleware solution will have the flexibility to support new data sources, such as a next-generation iPod, without requiring modifications to the user interface or to other software components.

Choosing an RTOS is a strategic decision for any project team. Once an RTOS vendor has provided clear answers to the above questions, you'll be much closer to choosing the RTOS that's right for you now—and in the future.

About QNX Software Systems

QNX Software Systems Limited, a subsidiary of Research In Motion Limited (RIM) (NASDAQ:RIMM; TSX:RIM), is a leading vendor of operating systems, middleware, development tools, and professional services for the embedded systems market. Global leaders such as Audi, Cisco, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle telematics units, network routers, medical devices, industrial control systems, security and defense systems, and other mission- or life-critical applications. Founded in 1980, QNX Software Systems Limited is headquartered in Ottawa, Canada; its products are distributed in over 100 countries worldwide. Visit www.qnx.com, follow @QNX_News on Twitter, and visit facebook.com/QNXSoftwareSystems.

www.qnx.com