RTOS versus GPOS:

What is best for embedded development?

By Paul N. Leroux

Do most embedded projects still need an RTOS? It is a good question, given the speed of today's high-performance processors and the availability of real-time patches for Linux, Windows, and other General Purpose Operating Systems (GPOSs).

Speed without the expense

The answer lies in the very nature of embedded systems, which are often manufactured by the thousands or even millions of units. Even a one dollar reduction in per-unit hardware costs can save the manufacturer a small fortune. Many of these systems are cost sensitive, where the use of multi-gigahertz processors or a large memory array is not possible. In the automotive telematics and infotainment market, for instance, the typical 32-bit processor runs at about 200 MHz – a far cry from the 2 GHz or faster processors now common in desktops and servers. In an environment like this, an RTOS designed to extract extremely fast (and predictable) response times from lower-end hardware offers a serious economic advantage.

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 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 must execute first (for example, an operation that protects user safety).

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 user's perspective, the device is unreliable. But with an RTOS, the developer can precisely control the order in which software processes execute, and thereby ensure that playback occurs at an appropriate and consistent media rate.

RTOSs are not fair

The need for *hard* real time – and for OSs that enable it – remains prevalent in the embedded industry. For evidence, consider recent developments in the Linux world. MontaVista, for example, has launched an open source project in an attempt to improve task preemption in the Linux kernel. Meanwhile, a recent study conducted by Venture Development Corporation suggests that lack of real-time performance is the biggest impediment to Linux adoption. The questions are:

- What does an RTOS have that a GPOS does not?
- How useful are the real-time extensions now available for some GPOSs?
- Can such extensions provide a reasonable facsimile of RTOS performance?

Task scheduling

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 offers no guarantees 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 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 will, 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 preempted by an even higher-priority thread. This approach, known as priority-based preemptive scheduling, allows high-priority threads to meet their deadlines consistently, no matter how many other threads are competing for CPU time.

Preemptible kernel

For most GPOSs, the OS kernel is not 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, executes 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. There are still windows of time in which preemption may not occur, but in a well-designed RTOS, those intervals are extremely brief, often on the order of hundreds of nanoseconds. Moreover, the RTOS will impose an upper bound on how long preemption is held off and interrupts disabled; this allows developers to ascertain worst-case latencies.

To achieve this goal, the RTOS kernel must be simple and as elegant as possible. Only services with a short execution path should be included in the kernel itself. Any operations that require significant work (for instance, process loading) must be assigned to external processes or threads. Such an approach helps ensure that there is an upper bound on the longest non-preemptible code path through the kernel.

In a few GPOSs, such as Linux 2.6, 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 incorporated into the kernel (for example, networking and file systems). Moreover, a preemptible kernel does not 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

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.* 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 fixed and bounded interval of time, an RTOS may provide a choice of mechanisms including priority inheritance and priority ceiling emulation. We could not possibly do justice to both mechanisms here, so let us focus on a simple example of priority inheritance.

To begin, we first must look at the blocking that occurs from synchronization in systems, and how priority inversion

"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."

can occur as a result. Let us say two jobs are running, 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 as a result of synchronization – waiting for a shared resource controlled by a lock or a semaphore – or as a result of requesting a service.

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 may vary, with a minimum, average, and maximum time. This interval is known as the blocking factor. If Job 1 is to meet any of its timeliness constraints, this factor cannot 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.

Now let us introduce a third job that has a higher priority than Job 2 but a lower priority than Job 1 (Figure 1). If Job 3 becomes ready to run while Job 2 is executing, it will preempt Job 2, and Job 2 will not be able to run again until Job 3 blocks or completes. This 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.

In fact, multiple jobs can preempt Job 2 in this way, resulting in an effect known as chain blocking. Under these circumstances, Job 2 might be preempted for an indefinite period of time, yielding an unbounded priority inversion, causing Job 1 to fail to meet any of its timeliness constraints. 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 will not be able to preempt Job 2, and the resulting priority inversion is avoided (Figure 2).

Dueling kernels

GPOSs – Linux, Windows, and various flavors of UNIX – typically lack the mechanisms we have just discussed. Nonetheless, vendors have developed a number of real-time extensions and patches in an attempt to fill the gap. There is, for example, the dual-kernel approach, in which the GPOS runs as a task on top of a dedicated real-time kernel (Figure 3).

Any tasks that require deterministic scheduling run in this kernel, but at a higher priority than the GPOS kernel. These tasks can thus preempt the GPOS whenever they need to execute and will yield the CPU to the GPOS only when their work is done.

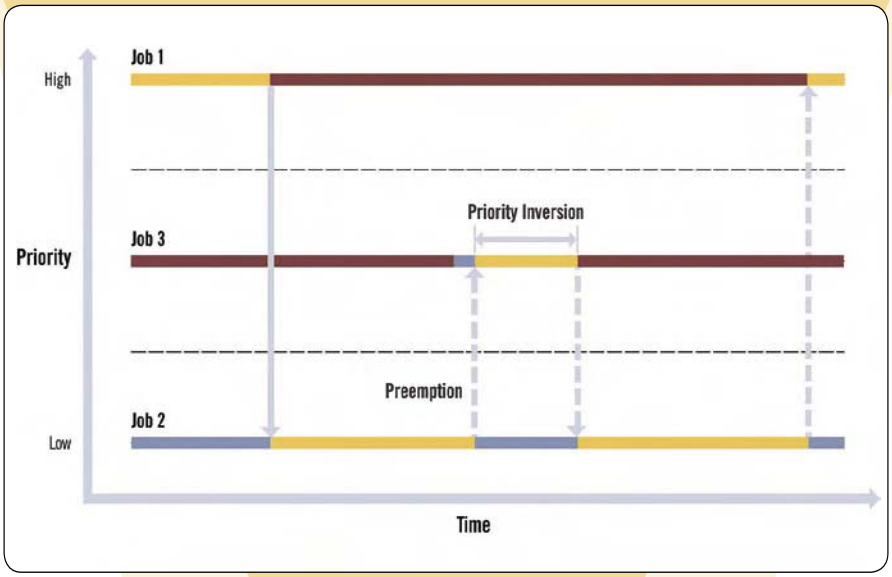

Figure 1

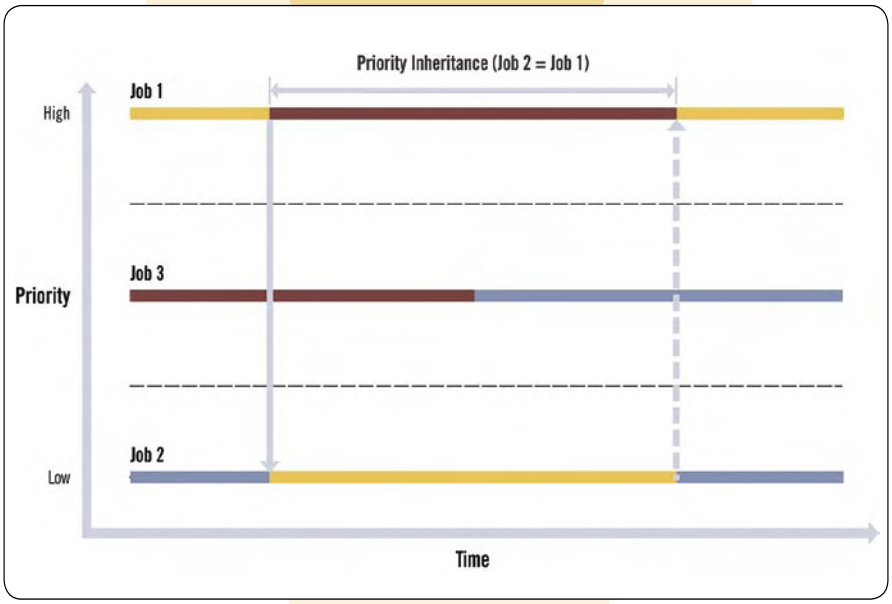

Figure 2

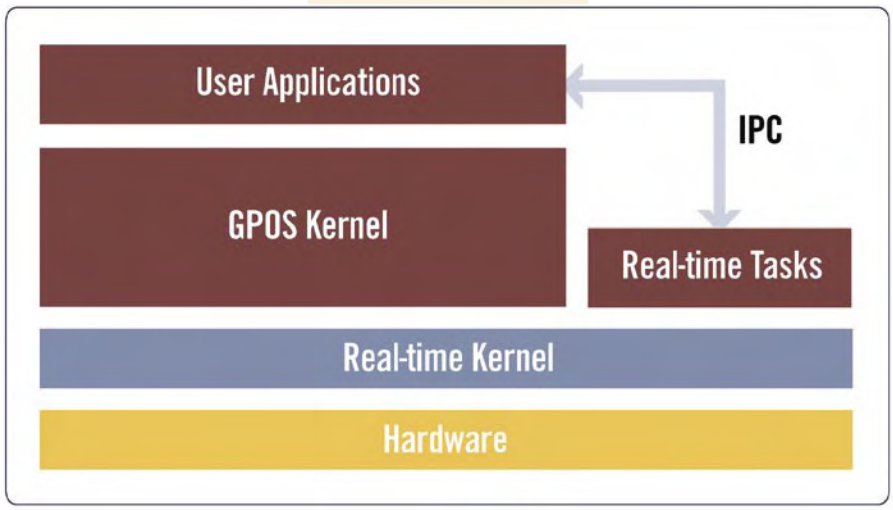

Figure 3

Unfortunately, tasks running in the real-time kernel can make only limited use of existing system services in the GPOS – file systems, networking, and so on. In fact, if a real-time task calls out to the GPOS for any service, it will be subject to the same preemption problems that prohibit GPOS processes from behaving deterministically. As a result, new drivers and system services must be created specifically for the real-time kernel – even when equivalent services already exist for the GPOS.

Also, tasks running in the real-time kernel do not benefit from the robust Memory Management Unit (MMU) protected environment that most GPOSs provide for regular, non-realtime processes. Instead, they run unprotected in kernel space. Consequently a real-time task that contains a common coding error, such as a corrupt C pointer, can easily cause a fatal kernel fault. To complicate matters, different implementations of the dual-kernel approach use different APIs. In most cases, services written for the GPOS cannot be easily ported to the real-time kernel, and tasks written for one vendor's real-time extensions may not run on another's real-time extensions.

Modified GPOSs

Rather than use a second kernel, other approaches modify the GPOS itself, such as by adding high-resolution timers or a modified process scheduler. Such approaches have merit, since they allow developers to use a standard kernel (albeit with proprietary patches) and programming model. Moreover, they help address the requirements of reactive, event-driven systems.

Unfortunately, such low-latency patches do not address the complexity of most real-time environments, where real-time tasks span larger time intervals and have more dependencies on system services and other processes than do tasks in a simple event-driven system. For instance, in systems where real-time tasks depend on services such as device drivers or file systems, the problem of priority inversion would have to be addressed.

In Linux, for example, the driver and Virtual File System (VFS) frameworks would effectively have to be rewritten along with any device drivers and file systems employing them. Without such modifications, real-time tasks could experience unpredictable delays when blocked on a service. As a further problem, most existing Linux drivers are not preemptible. To ensure predictability, programmers would also have to insert preemption points into every driver in the system.

All this points to the real difficulty, and immense scope, of modifying a GPOS so it is capable of supporting real-time behavior. However, this is not a matter of *RTOS good, GPOS bad*. GPOSs such as Linux, Windows XP, and UNIX all serve their intended purposes extremely well. They only fall short when they are forced into deterministic environments they were not designed for, such as those found in automotive telematics systems, medical instruments, and continuous media applications.

What about an RTOS?

Still, there are undoubted benefits to using a GPOS, such as 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 cannot afford to ignore these benefits. Extensive support for POSIX APIs – the same APIs used by Linux and UNIX – is an important first step. So is providing well-documented source 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. In a microkernel RTOS, only a small core of fundamental OS services (such as signals, timers, and scheduling) reside in the kernel itself. All other components (such as drivers, file systems, protocol stacks, and applications) run outside the kernel as separate, memory-protected processes (Figure 4).

As a result, developing custom drivers and other application-specific OS extensions does not require specialized kernel debuggers or kernel gurus. In fact, as user-space programs, such extensions become as easy to develop as regular 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,

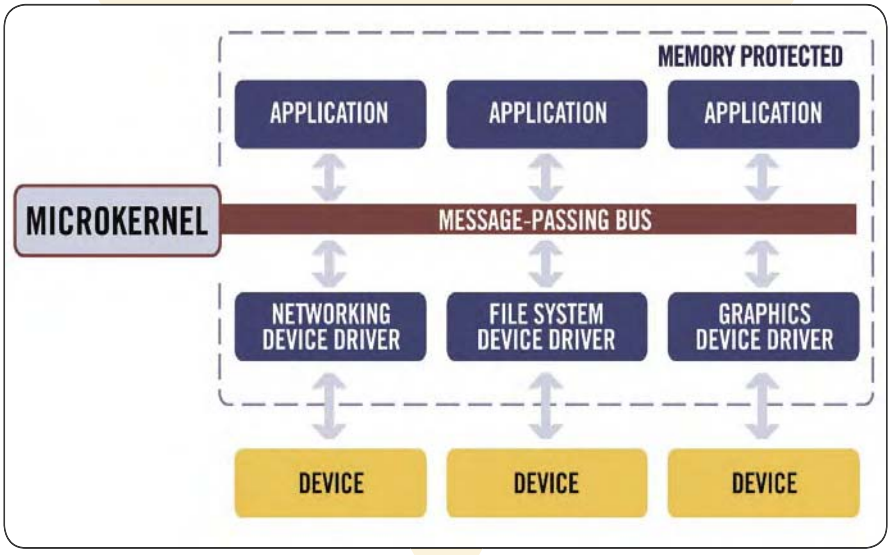

Figure 4

RTOS vs GPOS

along with diagnostic information such as the contents of data items and a history of function calls.

This architecture also provides superior fault isolation. 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. Likewise, drivers and other services can be stopped, started, or upgraded dynamically, again without a system shutdown.

A strategic decision

An RTOS can help make complex applications both predictable and reliable. In fact, the predictability made possible by an RTOS adds a form of reliability that cannot be achieved with a GPOS (if a system based on a GPOS does not 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.

Consider Internet support. Does the RTOS support an up-to-date suite of protocol stacks such as IPv4, IPv6, IPsec, SCTP, and IP filtering with NAT? And what about scalability? Does the RTOS support a limited number of processes, or does it allow hundreds or even thousands of processes to run concurrently? And does it provide support for distributed or symmetric multiprocessing?

GUI considerations

Graphical User Interfaces (GUIs) are becoming increasingly common in embedded systems, and those interfaces are becoming increasingly sophisticated. Consequently, does the RTOS support primitive graphics libraries, or does it provide an embeddable windowing system that supports 3D rendering, multi-layer interfaces, and other advanced graphics? Can you customize the GUI's look-and-feel? Can the GUI display and input multiple languages simultaneously? And does the GUI support an embeddable web browser? The browser should have a scalable footprint, and be capable of rendering web pages on very small screens. It should also support current standards such as HTML 4.01, XHTML 1.1, SSL 3.0, and WML 1.3.

Tool considerations

On the tools side, does the RTOS vendor offer diagnostic tools for tasks such as trace analysis, memory analysis, application profiling, and code coverage? And what of the development environment? Is it based on an open platform like Eclipse, which lets you readily plug in third-party tools for modeling, version control, and so on? Or is it based on proprietary technology?

On one point, there is no question. The RTOS can play a key role in determining how reliable your system will be, how well it will perform, and how easily it will support new or enhanced functionality. And it can support many of the rich services traditionally associated with GPOSs, but implemented in a way to address the severe processing and memory restraints of embedded systems. ECD

Paul Leroux *is a Technology Analyst at QNX Software Systems, where he has served in various roles since 1990. His areas of focus include OS architecture, high availability systems, and integrated development environments.*

For more information, contact Paul at:

QNX Software Systems Ltd.

175 Terence Matthews Crescent

Ottawa, Ontario, Canada K2M 1W8

Tel: 613-591-0931 • Fax: 613-591-3579

E-mail: paull@qnx.com

Website: www.qnx.com