What is Real Time and Why Do I Need It?

Authored by: **Steve Furr**
Updated by: **Thomas Fletcher**

Real time is an often misunderstood - and misapplied - property of operating systems. In this article, I will attempt to provide a summary of some of the critical elements of realtime computing and discuss a few design considerations and benefits.

What Is Real Time?

We can start with a basic definition of a realtime system, as defined in the FAQ for the **comp.realtime** newsgroup:

"A realtime 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."

Real time, then, is a property of systems where time is literally "of the essence." In a realtime system, the value of a computation depends on how timely the answer is. For example, a computation that is completed late has a diminishing value, or no value whatsoever, and a computation completed early is of no extra value. Real time is always a matter of degree, since even batch computing systems have a realtime aspect to them - nobody wants to get their payroll deposit two weeks late!

Problems arise when there is competition for resources in the system and resources are shared among many activities, which is where we begin to apply the realtime property to operating systems. In implementing any realtime system, a critical step in the process will be the determination of a schedule of activities such that all activities will be completed on time.

Any realtime system will comprise different types of activities - those that can be scheduled, those that cannot be scheduled, such as operating-system facilities and interrupt handlers, and non-realtime activities. If non-schedulable activities can execute in preference to schedulable activities, they will affect the ability of the system to handle time constraints.

What about hard real time?

Hard real time is a property of the timeliness of a computation in the system. A hard realtime constraint in the system is one for which there is no value to a computation if it is late, and the effects of a late computation may be catastrophic to the system. Simply put, a hard realtime system is one where all of the activities must be completed on time.

What is soft real time?

Soft real time is a property of the timeliness of a computation where the value diminishes according to its tardiness. A soft realtime system can tolerate some late answers to soft realtime computations, as long as the value hasn't diminished to zero. A soft realtime system will often carry meta requirements such as a stochastic model of acceptable frequency of late computations. Note that this is very different from conventional applications of the term, which don't account for how late a computation is completed or how frequently this may occur.

Soft real time is often improperly applied to operating systems that don't satisfy the necessary conditions for guaranteeing that computations can be completed on time. Such operating systems are best described as

quasi-realtime or pseudo-realtime in that they execute realtime activities in preference to others whenever necessary, but don't adequately account for non-schedulable activities in the system.

Who needs real time?

Traditionally, realtime operating systems have been used in "mission-critical" environments requiring hard realtime capability, where failure to perform activities in a timely manner can result in harm to persons or property.

Often overlooked, however, are situations where there is a need to meet quality of service guarantees, particularly when failure to do so could result in financial penalty. This covers obvious service scenarios, such as "thirty minutes or it's free," but it also includes intangible penalties, such as lost opportunities or loss of market share.

More and more, real time is being employed in consumer devices - complex systems that demand the utmost in reliability. For example, a non-realtime device aimed at presenting live video, such as MPEG movies, that depends on software for any part of the delivery of the content, may experience dropped frames at a rate that is perceived by the customer as unacceptable.

In designing systems, developers need to assess whether the performance benefits warrant the use of realtime technology. A decision made early on can have unforeseen consequences when overload of the deployed system leads to pathological behavior in which most or none of the activities complete on time, if at all.

Realtime technology can be applied to conventional systems in ways that have a positive impact on the user experience, either by improving the perceived response to certain events, or ensuring that important activities execute preferentially with respect to others in the system.

What is a realtime OS?

To the best of my knowledge, an acceptable definition of what constitutes a hard realtime operating system has never been put forward. I propose a modest definition based on realtime scheduling theory that is consistent with industry practice: A hard realtime operating system must guarantee that a feasible schedule can be executed given sufficient computational capacity if external factors are discounted. External factors, in this case, are devices that may generate interrupts, including network interfaces that generate interrupts in response to network traffic.

In other words, if a system designer controls the environment of the system, the operating system itself will not be the cause of any tardy computations. We can apply this term to conventional operating systems - which typically execute tasks according to their priority - by referring to scheduling theory and deriving a minimum set of conditions that must be met. Without getting into too much detail, scheduling theory demonstrates that a schedule can be translated into static priority assignments in a way that guarantees timeliness. It does so by dividing the time available into periodic divisions and assuming a certain proportion of each division is reserved for particular realtime activities.

In order to do so, the following basic requirements must be met:

1. Higher-priority tasks always execute in preference to lower-priority tasks.
2. Priority inversions, which may result when a higher-priority task needs a resource allocated to a lower-priority one, are bounded.
3. Non-schedulable activities, including both non-realtime activities and operating-system activities, don't exceed the remaining capacity in any particular division.

Because of condition 3, we must discount those activities outside of the control of the operating system, yielding the external factors provision above.

We can then derive the following operating system requirements (OSRs):

1. The OS must support fixed-priority preemptive scheduling for tasks (both threads and processes, as applicable).
2. The OS must provide priority inheritance or priority-ceiling emulation for synchronization primitives.
3. The OS kernel must be preemptible.
4. Interrupts must have a fixed upper bound on latency.
    - By extension, nested interrupt support is required.
5. Operating-system services must execute at a priority determined by the client of the service.
    - All services on which it is dependent must inherit that priority.
    - Priority inversion avoidance must be applied to all shared resources used by the service.

OSR 3 and OSR 4 impose a fixed upper bound on the latency imposed on the onset of any particular realtime activity. OSR 5 ensures that operating system services themselves - which are internal factors - don't introduce non-schedulable activities into the system that could violate basic requirement 3.

How does an RTOS differ from a conventional OS?

The key characteristic that separates an RTOS from a conventional OS is the predictability that is inherent in all of the requirements above. A conventional OS, such as Linux, attempts to use a "fairness" policy in scheduling threads and processes to the CPU. This gives all applications in the system a chance to make progress, but doesn't establish the supremacy of realtime threads in the system or preserve their relative priorities, as is required to guarantee that they will finish on time. Likewise, all priority information is usually lost when a system service, usually performed in a kernel call, is being performed on behalf of the client thread. This results in unpredictable delays preventing an activity from completing on time.

By contrast, the microkernel architecture used in the QNX RTOS is designed to deal directly with all of these requirements.

The microkernel itself simply manages processes (and threads) within the system, and allows them to communicate with each other. Scheduling is always performed at the thread level, and threads are always scheduled according to their fixed priority - or, in the case of priority inversion, by the priority, as adjusted by the microkernel to compensate for priority inversions. A high-priority thread that becomes ready to run can preempt a lower-priority thread.

Within this framework all device drivers and operating system services apart from basic scheduling and inter-process communication (IPC) exist as separate processes within the system. All services are accessed through a synchronous message-passing IPC mechanism that allows the receiver to inherit the priority of the client. This priority-inheritance scheme allows OSR 5 to be met by carrying the priority of the original realtime activity into all service requests and subsequent device-driver requests.

There is an attendant flexibility available as well. Since OSR 1 and OSR 5 stress that device-driver requests need to operate in priority order, at the priority of the client, throughput for normal operations can be substantially reduced. Using this model, an operating service or device driver can be swapped out in favor of a realtime version that satisfies these requirements. Complex systems will generally partition such resources into realtime and non-realtime with different service and device-driver implementations for each resource.

Because of the above, all activities in the system are performed at a priority determined by the thread on whose behalf they are operating.

What is a soft realtime OS?

A soft realtime OS must be capable of doing effectively everything that a hard realtime OS must do. In addition, a soft realtime OS must be capable of providing monitoring capabilities with accurate cost accounting on the tasks in the system. It must determine when activities have failed to complete on time or when they have exceeded their allocated CPU capacity, and trigger the appropriate response.

How does all of this affect my application?

If you are writing an application or system for deployment on a realtime OS, it is important to consider the effect that the RTOS characteristics have on the execution of the application, and to understand how it can be used to your benefit. For example, with an RTOS you can increase responsiveness of certain operations initiated by the user.

Most applications will normally run at the default user priority within the system. This means that applications normally run in a round robin execution, competing with each other for a proportion of the CPU capacity. Without the type of realtime schedule mentioned above, you can manipulate the priorities of the processes in the system to have certain activities run preferentially to others in the system. Manipulation of the priorities is a double-edged sword. Used judiciously, it can dramatically improve response in areas that are important to the user. At the same time, it is possible to "starve" other processes in the system in a way that typically doesn't happen on a conventional desktop system.

The key to ensuring that higher-priority processes and threads don't starve out other processes in the system is to be certain of the limits imposed on their execution. By pacing the execution, or by throttling it in response to load, you can limit the proportion of CPU consumed by these activities so user processes get their share of the CPU.

Media players, such as audio players (MP3, .wav, etc.) and video (MPEG-2), are a good example of applications that can benefit from priority manipulation. The operation of a media player can be tied to the media rate that is required for proper playback (i.e. 44kHz audio, 30 fps video). So within this constraint, a reader thread that buffers data and a rendering or playback thread can both be designed to awaken on a programmable timer, buffer or render a single frame, and then go to sleep awaiting the next timer trigger. This provides the necessary pacing, so that the priority can be assigned above normal user activities, but below more critical system functions.

By choosing appropriate priorities, playback will occur consistently at the given media rate. A well-written media player will also take into account quality of service, so that if it doesn't receive adequate CPU time, it can reduce its requirements by selectively dropping samples or follow an appropriate fall-back strategy. This will then prevent it from starving other processes as well.

You may also wish to treat certain user events preferentially within the system. This works well when you increase the concurrency within an application, and when the event can always be handled in a predictable, small amount of time. The key concern here is the frequency at which these events can be generated. If they can't occur too frequently, it is safe to raise the priority of the thread responding to them. If they can occur too frequently, other activities will be starved under overload conditions.

The simplest solution is to divide responsibility for events into different handling threads with different priorities and queue requests or deliver them with messages. You can tie the handler's execution to a timer, so that the execution of the thread is throttled by the timer, handling a fixed number of requests within a given interval. This stresses the importance of factoring areas of application responsibility, giving a flexible

design with opportunities for effective use of concurrency and preferential response, all of which lead to a greater feel of "responsiveness."