The Joy of Scheduling

Jeff Schaffer, Steve Reid
QNX Software Systems
jspchafer@qnx.com, stever@qnx.com

Abstract

The scheduler is at the heart of the operating system: it governs when everything — system services, applications, and so on — runs. Scheduling is especially important in realtime systems, where tasks are expected to run in a timely and deterministic manner. In these systems, if the designer doesn’t have complete control of scheduling, unpredictable and unwanted system behavior can and will occur.

Software developers and system designers should understand how a particular OS scheduler works in order to be able to understand why something doesn’t execute in a timely manner. This paper describes some of the more commonly used scheduling algorithms and how scheduling works. This knowledge can help developers debug and correct scheduling problems and create more efficient systems.

What Does a Scheduler Do?

Wikipedia provides a succinct and accurate list of what a scheduler does:

- **CPU utilization**, keeping the CPU as busy as possible
- **throughput**, the number of tasks that complete their execution per time unit
- **turnaround**, the total time between submission of a task and its completion
- **waiting time**, the amount of time a task waits in the ready queue
- **response time**, the amount of time it takes from when a request was submitted until the first response is produced
- **fairness**, allocating equal CPU time to each task

In realtime operating systems, the scheduler must also make sure that the tasks meet their deadlines. A “task” can be a few lines of code in a program, a process, or a thread of execution within a multithreaded process.

Common scheduling algorithms

A very simple system that repeatedly monitors and processes a few pieces of data doesn’t need an elaborate scheduler; in contrast, the scheduler in a complex system must be able to handle many competing demands for the processor. Because the needs of different computer systems vary, there are different scheduling algorithms.

**Cyclic executive / run to completion**

In a cyclic executive, the system loops through a set of tasks. Each task runs until it’s finished, at which point the next one is run. Interrupts are not permitted in this type of system, requiring hardware devices to be polled in order to determine if an operation has been completed and (if applicable) data is available for processing. This algorithm is typically used in homegrown OSs and legacy executives.

The problem with this type of scheduling is timing. While the original system as designed might meet all the deadline requirements (even if the requirement is as simple as just delivering “reasonable” performance without any hard deadline), modification and expansion of the system becomes difficult. Any addition to the system affects the ability to meet deadlines, since the entire timing of system changes.

![Diagram](https://via.placeholder.com/150)

*Figure 1. A cyclic executive iterates through a given set of tasks.*

**Cooperative multitasking**

In cooperative multitasking, applications run until they explicitly “yield” the CPU, so that another application can
run. Examples of where this algorithm is used are DOS and some versions of Microsoft Windows.

The problem with this type of scheduling is that it is dependent on all tasks’ behaving “properly”; should an uncooperative application refuse to yield in a timely manner, deadlines will not be met. It may also result in the system becoming unresponsive or “locking up” due to the failure of a task to yield, often necessitating a system reboot.

![Figure 2. Applications run until they yield in a cooperative multitasking system.](image)

**Round-robin scheduling**

In round-robin scheduling, a task runs until it consumes a specified amount of time (a number of OS timer ticks called a timeslice), or until it gets blocked. It then goes to the tail of the READY queue.

![Figure 3. Round-robin tasks run until they've consumed their timeslice, or until they're blocked.](image)

Round-robin scheduling is often used as the poor man’s version of the UNIX time-sharing scheme, but note that time sharing isn’t fair sharing:

- There is no dynamic system manipulation of priorities.
- Tasks do not always share the CPU equally. As shown in Figure 3, if a task blocks (such as when waiting on data to become available from an I/O operation), it loses part of its timeslice. Since this occurs sometime between system timer ticks, the next task scheduled will receive extra computation time because it can receive its full timeslice plus the unused time to the next timer tick from the previously scheduled task.

**Preemptive priority-based multitasking**

With this algorithm, a system designer assigns priorities to tasks (more on this later), and the scheduler uses these priorities to determine which task should run next. A running task is immediately preempted by a higher-priority task. This algorithm is implemented in most realtime operating systems (RTOSs).

![Figure 4. Running threads are preempted by threads with higher priorities.](image)

This algorithm is not designed to be fair; it is designed to be deterministic. The highest-priority task can never be preempted if it doesn’t block, which will cause other, lower-priority tasks, to be starved of CPU time. If the operating system permits multiple tasks to have the same priority, then another scheduling policy (for example, FIFO/run to completion or round-robin) determines the scheduling of tasks with the same priority.

**Time partition scheduling**

Time partition scheduling guarantees that threads or groups of threads get a predetermined percentage of the CPU. This can prevent one part of the system from starving other processes. It can also prevent malicious events (such as excessive computation incurred from something like a Denial of Service attack) from consuming all the processing time.

![Figure 5. Partitioning guarantees predetermined amounts of resource to competing tasks.](image)

**Deadline scheduling**

In this algorithm, which is available in specialized OSs, the schedule is dynamically computed such that the applications meet previously specified deadlines. While this
is an exciting area of research, it is not a common method of scheduling and beyond the scope of this paper.

![Diagram of A and B]

**Deadlines**

Figure 6. Applications scheduled so that they meet their deadlines.

**Be careful of reality**

Note that, because system requirements often change over time, what originally appears to be a “simple” system may grow in complexity very quickly. It is important to take great care in choosing the scheduling algorithm for a system, since what initially appears to be sufficient for the device may prove inadequate if (and usually when) system requirements change.

**Interrupts and scheduling**

An interrupt is exactly what it sounds like: an interruption of whatever is currently going on, and a diversion to another task. For example, if a hardware device has some data, it raises an interrupt request or IRQ. The processor runs an Interrupt Service Routine to handle the interrupt.

Traditional interrupt handlers aren’t controlled by the scheduler, and they effectively have a priority above that of any thread. Depending on the hardware, interrupts may also be nested.

![Diagram of IRQx and IRQy]

**Figure 7. Interrupts preempting running threads.**

Interrupts can cause threads to be rescheduled, as illustrated in Figure 8. In this case, interrupt handler y caused Thread B to be made ready to run. When interrupt processing has completed and normal thread scheduling occurs, the priority-driven scheduler chooses the higher-priority Thread B to be executed rather than Thread A (which was executing at the time the interrupt occurred). This is often missed by developers using a priority-driven scheduler; they assume that the thread that was running when the interrupt occurred will continue to run after the interrupt handling is complete.

![Diagram of Thread A and B]

**Figure 8. Interrupts causing a thread to be rescheduled.**

The interrupt processing architecture of the system can also affect deadlines. For example, it is common to break interrupt handling into two sections:

- activities that must be handled at interrupt time (such as emptying buffers that will be quickly overwritten)
- activities that can be delayed and handled during normal, priority-driven, thread scheduling (such as processing data for use by other tasks)

These activities are typically referred to as the **Interrupt Service Routine or ISR (the interrupt-handling part)** and the **Interrupt Service Thread or IST (thread-scheduled part of the handler).**

![Diagram of IRQx and IRQy]

**Figure 9. Requests for Interrupt Service Threads are queued individually, permitting deterministic interrupt handling.**

The architecture that implements the communication between these two parts is critical. Realtime systems will use an architecture that directly passes the request from the ISR to the queue for that particular IST, as shown in Figure 9. Each thread is then scheduled according to its individual priority; for example, the IST responsible for a processing data from a high-speed data network will typically run at higher priority than a serial port’s IST (which has a much
slower data rate). With this architecture, interrupt handling is deterministic.

Non-realtime systems, however, will often use a communication architecture, like the one shown in Figure 10. While the ISRs and ISTs may be prioritized, the architecture that connects them is not; it is a shared queue. This means that high-priority events can be delayed by lower-priority events. Interrupt handling is not deterministic, because it now depends on the order of the interrupts’ arrival into the system, rather than on the priority of the interrupt.

![Figure 10. Requests for Interrupt Service Threads are queued through a shared queue for execution, permitting requests for lower-priority interrupts to be executed before higher-priority interrupts.](image)

Priorities

Many scheduling algorithms use priorities to determine the order in which tasks should run. The priorities and how they’re manipulated can (not surprisingly) greatly affect the behavior of the entire computer system.

Changing priorities dynamically

Changing a task’s priority during runtime can be useful. For example, a system may have different normal states of operation that it switches between, each requiring a different operational “blend” of service. The operating system itself may change a task’s priority in order to meet a deadline or to optimize performance. The key points to consider are: a) can the priority be dynamically changed? and b) why is the change occurring?

The first point is the simpler — either the operating system permits dynamic task priority changes or it does not. Most operating systems (including the QNX® Neutrino® RTOS) permit this. Some operating systems, however, do not; for them priority is fixed at system implementation, so if this capability is an important design feature, it is worthwhile checking if it is supported.

The second point — why? — is the more interesting and difficult issue, since it can have a great effect on system performance. If the developers are making the change, it can be assumed that they understand the reasons for the change, as well as its consequences. If a malicious task is making the change, then a security problem exists which is well beyond just a scheduling problem.

The real issue is when the operating system itself changes the priority of the task. While there can be excellent reasons for the OS to do change a task’s priority, such as preventing priority inversion (as discussed in the next section) or a deadline scheduler modifying priorities to guarantee that deadlines are met, there is another reason the OS may change the priority: balancing performance and system responsiveness. The goal here is usually to prevent computationally intensive tasks from consuming all the CPU time. This type of scheduling, known as priority decay scheduling, permits the OS to temporarily lower the priority of a task because it has received too much CPU time, then raise it again when it has become starved for CPU time.

While priority decay scheduling may make perfect sense on non-realtime systems, it can wreak havoc on a realtime system with deadlines to meet. While it would seem reasonable to say that no RTOS would implement such scheduling, in reality it is quite possible and it has been done. Nor is it necessarily wrong, since realtime systems often have non-realtime components where such scheduling is appropriate. This does not mean that priority priority-decay scheduling doesn’t cause problems for developers, however, simply because they do not expect it.

For example, one RTOS made priority decay scheduling its default scheduling algorithm when processes were scheduled at the same priority level. This strategy was perfectly reasonable since the RTOS: a) was often used for...

---

2 This is usually better solved with other mechanisms and different system design; however, some designers prefer to tackle the problem by changing task priorities.

3 To be completely accurate, this type of scheduling was not the “default” for the system. The task inherited its priority from its parent task; however this was typically a command shell which, unless explicitly changed, ran with this scheduling algorithm.
computationally intensive, non-realtime tasks where priority
decay scheduling was appropriate, b) offered other types of
scheduling algorithms, and c) clearly marked in its
documentation that realtime tasks should change their
scheduling algorithm (and priority, if appropriate) to
something more suitable when needed.

A problem occurred, though, when developers incorrectly
assumed which type of scheduling algorithm was actually
being used; most assumed it was a round-robin algorithm.
These assumptions led to “interesting” failures in customers’
systems, such as a motor control mechanism that worked for
several minutes (when the control task was assigned its
normal priority), then did not work at all for several minutes
(when the scheduler lowered the priority of the task,
preventing it from getting any service since the remaining
tasks were of higher priority and consumed all of the CPU
time). In the next version of the operating system, this design
was removed — not because it was wrong, but because it
caus ed confusion for developers.

It is important to remember, however, that operating systems
do have legitimate reasons to dynamically adjust priority.
The key point is that the actual operation of an operating
system, whether it is dynamic priority adjustment or some
other service, may not be what the developers think it is, or
what they want. Most realtime operating systems provide
alternative mechanisms, but developers need to be aware
that these alternative mechanisms need to be explicitly
selected.

Priority inversion

Priority inversion is a situation that prevents the proper
scheduling of threads based strictly on priority. It typically
occurs when a shared resource is held by a lower-priority
thread, thus preventing the higher-priority thread from
executing as it should. Other types of priority inversion also
exist, such as having low-priority threads requesting a high-
priority thread do work on its behalf (such as reading from a
disk drive), thus “artificially” inheriting the higher priority of
the other thread for the work being executed and
preempting all threads whose priority lies between the two.
The opposite situation of having a high-priority thread being
delayed by sending requests to a low-priority thread for work
is also possible, though this is usually just indicative of poor
system design.

Priority inversion is a serious issue in realtime systems. It
leads to missed deadlines and unexpected actions by the
system. One notorious example of a priority inversion
occurred on the Mars Pathfinder lander: the inversion
resulted in the delayed execution of a task, preventing it from
completing its work in a timely and expected manner. This
error caused another task to determine that a severe error
had occurred and reboot the system.  

![Figure 11. An example of priority inversion](image)

**Figure 11. An example of priority inversion:**

Figure 11 shows an example of a simplified yet typical
priority inversion. The sequence of events that led to this
particular inversion is:

1. Task 3 (a low-priority task) is running and acquires a
   mutex to a resource shared with Task 1.

2. Task 1 (a high-priority task) becomes ready to run. It
   preempts Task 3; however, it cannot acquire the mutex
   to the shared resource being used by Task 3. Task 1
   blocks awaiting the release of the mutex, and execution
   of Task 3 is resumed. In normal operation, Task 3 will
   quickly complete its use of the shared resource and
   release the mutex, permitting Task 1 to acquire it and
   still meet its critical deadline.

3. In this case, however, an unanticipated event occurs:
   Task 2 (a medium-priority task) becomes ready to run.
   Since it has a higher priority than Task 3, it preempts
   the lower-priority task. Task 3 now cannot release the
   mutex in a timely manner.

4. Since Task 1 is delayed from execution until it can
   acquire the mutex, it misses its critical deadline due to
   the priority inversion.

---

4 Reeves, Glenn, *What Really Happened on Mars?*,
In this case, the priority inversion could have been prevented had the operating system provided mechanisms such as priority inheritance or priority ceiling emulation.

**Assigning task priority**

This question of what priority to assign to a task is one of the key issues developers face in a priority-driven scheduling system: what priority should they assign to each task? Typically, software developers just make an intuitive guess at the priorities for the tasks, then adjust them if the system’s behavior isn’t acceptable. Formal analysis of the system will give the best results and guarantee that deadlines are met, at the cost, however, of a significant amount of developer effort in order to perform analysis that may not really be necessary.

Rate Monotonic Analysis is the most common method used to analyze and assign priorities for systems with periodic deadlines. The steps in this analysis are as follows:

1. Determine the periods (deadlines) for all tasks.
2. Assign the highest priority to the tasks with the smallest periods.

The following table gives an example of the periods and priorities that might be assigned to some tasks.

<table>
<thead>
<tr>
<th>Period</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 msec</td>
<td>60</td>
</tr>
<tr>
<td>8 msec</td>
<td>50</td>
</tr>
<tr>
<td>22 msec</td>
<td>40</td>
</tr>
<tr>
<td>100 msec</td>
<td>20</td>
</tr>
</tbody>
</table>

This algorithm is robust and, assuming that the deadline calculations are correct, it is guaranteed to produce a workable schedule if one exists. It’s an excellent algorithm, but it is not a panacea. Limitations exist, primarily because the algorithm always assumes worst-case conditions in order to guarantee that deadlines are met:

- **RMA** is primarily used where systems are periodic in nature; however, many systems are asynchronous or event-driven (asynchronous HMI input, network traffic, etc.). It is not that an RMA-derived schedule will not work with this type of system, it is simply that it is difficult to calculate deadlines based on the scheduling of the random events.

- **Maximum CPU utilization** may not be achieved. RMA-derived schedules must account for worst-case scheduling based on the number of tasks being scheduled, eventually reaching a low of approximately 69% CPU utilization. It may be possible to achieve better CPU utilization in certain types of systems using other scheduling techniques; however, these other scheduling techniques may not be able to guarantee that deadlines will always be met.

- **Interrupts** need to be limited, or polling needs to be used. Barr\(^5\) has correctly pointed out the common misconception that RMA-derived schedules may not include interrupt processing, assuming that polling is the only way to ensure that deadlines will be met.

The problem is not that interrupts may not be used. It is that the interrupt-processing time must be accounted for in the deadline analysis. Unfortunately, given that any interrupt processing takes precedence over normal task scheduling, interrupts that may not be involved in meeting a particular deadline will still be executed in advance of regularly scheduled tasks that must be executed to meet the deadline. Additionally, should the rate of interrupts (whether involved in meeting the deadline or not) exceed the number used for deadline computation, then the RMA-derived schedule will not be valid. If it is possible, then polling may be preferable, because the processing time is bound and the order of service easily prioritized to ensure that deadlines are met (assuming that polling is done frequently enough so that no information is lost).

- **DMA** (Direct Memory Access) isn’t recommended. DMA transfers can prevent the task executing on the CPU from accessing memory for a short period during the transfer, although it is possible that another, lower priority task will be able to execute during this time period. The end of a DMA transfer also causes an interrupt to be generated, the problems of which are noted above. While it is not impossible to account for the additional time required, it is difficult to account for this time and it can lead to a deadline miss. Given this, the use of DMA transfers is not encouraged with RMA derived schedules.

- **RMA** does not permit interaction between the tasks. Only access to shared resources is permitted under the model.

Note that several of the issues presented here are not unique to RMA-derived schedules. Items such as an excessive number of interrupts affect all types of systems that must meet deadlines, and so must be accounted for regardless of the type of scheduling algorithm involved. The difficulty of dealing with some of these issues, however, may vary greatly depending on mechanisms provided by the operating system. The QNX Neutrino RTOS, for example, provides a mechanism where no ISR exists and all interrupt processing for a device (except for some minimal and bound time spent in the kernel) is pushed into the IST. This design permits interrupts to be processed in a more controlled and deterministic manner, since they are primarily processed under normal scheduled operation.

The good and the bad of priority scheduling

Priority-based, preemptive scheduling helps ensure that urgent tasks execute in a real-time system. It has many advantages:

- It provides deterministic real-time scheduling, ensuring that the “most urgent” software runs when it needs to.
- Multithreaded systems are common today, and the concept of priority-driven multithreaded systems is well known and understood by the design community.

Priority-based scheduling isn’t “fair share” scheduling:

- It does not guarantee that a thread will execute in a timely manner when higher-priority threads are ready to run.
- Without guaranteed CPU time, low-priority threads can become starved, causing system degradation or even system failure.
- Priority-based scheduling is vulnerable to problems that can monopolize the CPU, such as a Denial of Service attack or a high-priority thread’s running in an endless, non-blocking loop.
- It can be difficult to determine the proper priority for each thread.

Partitioning

Partitioning provides a way for an OS to share resources among competing processes, or even to protect (or secure) computer resources from malicious software.

Time partitioning is a mechanism that distributes CPU time within a defined time period (a periodic cycle) into logical divisions known as partitions. Partitions are made up of one or more threads, though the threads do not need to be from the same process. In fact, multithreaded processes may have each thread assigned to a different partition if the design requires it. Each partition is guaranteed a predetermined amount of CPU time; though partitions need not be assigned equal amounts of CPU time, any logical division works as long as it does not exceed 100% of the available CPU time.\(^6\)

![Partition scheduler](image)

Figure 12. Partitions govern the division of CPU time between different sets of threads.

When a partition is allocated CPU time, the question arises of which thread assigned to that partition should be scheduled. Although any scheduling algorithm, such as round-robin, could be used, virtually all systems use priority-based scheduling. The division between partitioning and priority scheduling guarantees that each major division of work (the partition) is guaranteed some amount of CPU time within the periodic cycle. However, the actual work to be done by the threads within that division is based on each thread’s overall importance within the system.

For example, one partition might include threads from processes that provide the user interface, another partition might contain threads from processes that control multimedia (to ensure uninterrupted music or video playback), and a third might control other data-processing functions necessary for a satisfying user experience. If this

\(^6\) Note that available CPU time does not include overhead for typical operating system functions such as scheduling. Processing time for hardware drivers and management software (such as a TCP/IP stack) may or may not be included, depending on the design of the operating system and the overall system design. In the QNX Neutrino RTOS, for example, most or all of this functionality can be controlled by the time partition scheduling mechanism.
were a factory automation system, the divisions might be a user interface, motor control, sensor processing, and communication with central factory-control computers. The combinations are limited only by the number of partitions that the operating system permits, the overall system requirements, and the imagination of the system architect. Regardless of the actual design chosen, putting different parts of the system into different partitions prevents one part of the system from using all the computer’s resources to the detriment of the other parts.

Time partitioning is in fact a general term that encompasses a mechanism that can be implemented using one of the following methods: static time partitioning or dynamic time partitioning. Each method will be described in detail below, though each is fairly obvious by definition: static time partitioning describes a system where CPU time assignments are defined in advance of runtime and never vary during the periodic cycle, while dynamic time partitioning defines a system where the operating system may adjust the actual CPU time assigned to a partition during the periodic cycle.

Note that some operating systems also support the concept of space partitioning, which provides a guaranteed amount of memory for each partition. Memory is reserved for use by processes within the partition, and the system architect can explicitly cap the memory consumption of each partition. Since this paper is about scheduling, however, time partitioning is what we’ll focus on here.

**Static time partitioning**

As defined above, static time partitioning is a scheduling mechanism where each partition is pre-assigned a specific amount of CPU time within a periodic cycle; these assignments cannot be changed at run time. Further, threads assigned to each partition are assigned at system design time and cannot be changed. It is typically found in avionics systems and is specified in the ARINC 653 standard.

There are some disadvantages to static time partitioning, however. These include:

- **Unused time is wasted time.** If a partition fails to use the CPU time allocated to it, the scheduler cannot assign it to another partition. The CPU remains idle during the remainder of the partition’s time allocation until the next partition can be scheduled.

**Dynamic time partitioning**

As noted above, dynamic time partitioning is a scheduling algorithm where the operating system can vary the amount of time each partition will receive during a processing cycle. The goal is to maximize CPU utilization; if the threads in a partition do not utilize their entire CPU allocation, the remaining time can be allocated to other tasks. Some refer to this method of scheduling as slack stealing, a term that is not completely accurate. Slack-stealing schedulers are only one type of dynamic time partition schedulers; other types exist.7

For this paper, we will look at the type of dynamic time partitioning implemented by the QNX Neutrino RTOS. Known as *adaptive partitioning*, it reallocates CPU time not used by a partition and makes it available to other partitions, maximizing performance in resource-constrained devices.8 Adaptive partitioning is very good at handling bursty CPU demands. If the system is lightly loaded, the scheduling is similar to that in a priority-driven system; if the system is heavily loaded, the scheduling is similar to that in static time partitioning. Further, unlike time partitioning in many systems, adaptive partitioning permits programmers to reassign threads to another partition, and dynamically adjust partition sizes (CPU allocated to the partition during the periodic cycle).

Some safety-critical systems require static time partitioning. Note, however, that an adaptive time partitioned system can easily become a static time partitioned system. By placing a thread containing a simple endless loop at the lowest priority in each partition, it forces the partition to use all of its allocated time and renders it into a static time partitioned system. A static time partitioned system cannot be made into an adaptive time partitioned system, however.

---

7 While slack-stealing schedulers do reallocate unused CPU time, most (if not all) slack-stealing schedulers use this time to schedule the processing of aperiodic events not normally covered during the periodic cycle.

8 Adaptive partitioning is more complex than our description here. We have limited our discussion to essential characteristics relevant to scheduling. For more information, see other QNX papers on adaptive partitioning, or the QNX Neutrino RTOS product documentation.
Example 1: Global Positioning System
Let’s compare how a GPS might work under static and adaptive time partitioning. Let’s assume that there are three partitions. We’ll see the colors for idle time and each partition in the charts that follow, and we’ll look at the usage of CPU time in a couple of scenarios.

<table>
<thead>
<tr>
<th>Name</th>
<th>Time allocation</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>User interface</td>
<td>30%</td>
<td></td>
</tr>
<tr>
<td>Route calculation</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>Diagnostics and data acquisition</td>
<td>30%</td>
<td></td>
</tr>
<tr>
<td>Idle</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

**Rerouting**
When rerouting, the GPS needs to determine a new route and display it, so the first two partitions might exceed their time allocations. In static partitioning, any time that the third partition doesn’t use becomes idle time, as shown in Figure 13.

Powering up
The effect of adaptive partitioning is even more dramatic when you power up the GPS. In this case, the partition for route calculation is idle, and the partition for the UI isn’t very busy. Under static time partitioning, the CPU usage might be as shown in Figure 15.

![Figure 13. CPU consumption while rerouting on a GPS using static time partitions.](image)

With adaptive time partitioning, the partition for diagnostics and data acquisition is allocated the time that the other partitions aren’t using, as shown in Figure 16.

![Figure 16. CPU consumption while powering up a GPS improves dramatically if adaptive time partitions are used.](image)

Example 2: Simple automation system
As another example, let’s consider a simple automation system with several components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Thread priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote web-monitoring agent</td>
<td>Low</td>
</tr>
<tr>
<td>Local Human Machine Interface (HMI)</td>
<td>Medium and low</td>
</tr>
<tr>
<td>Sensor scanning &amp; data acquisition</td>
<td>Medium and low</td>
</tr>
<tr>
<td>Motor control</td>
<td>High</td>
</tr>
</tbody>
</table>

If we rely on thread priorities alone in this system, then during the integration phase, we might discover that the remote monitoring agent works well until the operator uses
the local HMI, at which point the remote agent freezes and ceases to display updates. Troubleshooting might reveal that when the HMI commands result in a high level of motor control, the remote agent stops getting any CPU time.

![Remote web monitor](remote_web_monitor.png) ![Local HMI](local_hmi.png)

**Figure 17.** The remote web monitor is starved for CPU time in a priority-based system when the local HMI is busy.

**Trying to solve the problem by adjusting priorities**
In an attempt to fix the system, we might assign to the local HMI a lower priority than to the remote monitoring agent, but this could lead to unacceptable HMI performance. Setting the remote agent, data acquisition, and HMI to medium priority might not work either, because this compromises the performance of the data acquisition. Because priority reassignment doesn’t resolve the issue, we must attempt to change thread behavior and monitor the HMI’s CPU consumption, and make sure that it yields occasionally to let the remote agent run — a costly solution at the integration stage.

**Designing with partitions**
Time partitioning provides a better solution. We allocate CPU budget to subsystems and each one of our development teams, eliminating the need for system-wide priority schemes. Each design team can then develop its own priority schemes within each partition. The RTOS enforces the partitions’ budgets, then uses priority-based scheduling within each partition. The partitions can be static or adaptive, as required.

**Is time partitioning always the best solution?**
There are situations where time partitioning will have problems; for example, when someone presses the power button. In this case:
- cyclic schedulers need time to recognize the event
- the deadline must still be met
- handling it in interrupt processing can have issues

This type of situation affects both static and adaptive partition schedulers, but adaptive partition schedulers may be less affected. In the QNX Neutrino RTOS, for example, the concept of a critical thread exists; when this thread is activated (such as when someone presses an emergency power off button), the scheduler stops all partition scheduling and immediately processes that thread.

**Multicore scheduling**
On a single-core system, the RTOS scheduler ensures that the highest-priority thread runs. In this example, processes P1 and P2, and threads T1 and T2 share the same CPU, and execution is serialized on the CPU.

![Single-core processor](single_core_processor.png)

**Figure 18.** Several processes and threads run serially on a single-core processor.

On a Symmetric Multiprocessing (SMP) system, the scheduler ensures that the highest-priority threads run on all available cores.
When feasible, the SMP scheduler on a QNX Neutrino RTOS system runs a thread on the same core that it was last run in order to reduce thread migration, because excessive thread migration can result in decreased system performance due to the increased number of cache reloads.

**CPU affinity**

Some code that was written for a single-processor system might not run correctly on a multicore system. For example, if the developers assumed that threads would run sequentially (such as in a run-to-completion scheduling algorithm), and he relied on this assumption to control access to a shared resource, he would quickly discover that the system will fail when the threads are run in parallel by a multicore-enabled scheduler.

Rewriting the software could be expensive, or might not even be possible, so some OSs have implemented CPU affinity to associate processes with specific CPUs. The QNX Neutrino RTOS uses a type of CPU affinity (known as Bound Multiprocessing or BMP) that lets developers associate threads with one or more specific cores.

For example, in Figure 20, threads T1 and T2 have been bound to Core 4, while the remaining processes are free to run on any core. We could also bind processes P1 and P2 to cores 1, 2, and 3. This would reserve the remaining core for threads T1 and T2 alone, if this were necessary to make process P3 behave correctly.

This technique can be useful when we need to support a legacy code base and multicore-capable processes on the same system. Binding a process to a single core makes the system appear as a single-core processor to that process, and preserves any single-processor assumptions or errors in the software.

CPU affinity lets the system architect design a system where any thread can be run on a dedicated core, a set of cores \( x \) (which can be all cores in the system), or any core except a set of cores \( y \). This flexibility lets the architects optimize the scheduling in a manner that is best for the system they are designing.

For a more detailed discussion of CPU affinity, see the QNX whitepaper Processor Affinity or Bound Multiprocessing? Easing the Migration to Embedded Multicore Processing.

**Scheduling Problems**

Regardless of how good our system design is, there will always be problems that show up after the system is implemented. To put it bluntly, we are human and we will make mistakes or fail to comprehend all aspects of system behavior. Threads will not be scheduled in the order we expect, they will be pre-empted, etc.

In order to solve these issues, it is essential to have good analysis tools that look at what is actually occurring in the system. Analysing the workflow on a multicore system requires even more sophisticated tools to show us which core the thread is running on and the number of thread migrations from one core to another.

Tools such as the Systems Analysis perspective in the QNX Momentics\textsuperscript{®} Integrated Development Environment (IDE), as shown in Figure 21, show exactly what the system is scheduling/processing during a selected portion of
execution. The Systems Analysis perspective also helps us examine the workflow on multicore systems, letting us determine if a thread should be locked to a particular core for performance reasons.

Figure 21. The QNX Momentics Integrated Development Environment (IDE) is an ideal tool for analysing the scheduling on a multicore system.

**Conclusion**

It is important for software developers and system designers to understand how OS schedulers work, in order to make sure that all their systems’ processes and applications are scheduled so as to produce a system that runs correctly and as efficiently as possible. Too often developers assume they “know” how the scheduling works — often based on previous experience with another operating system — and they don’t take the time to find out how scheduling and items that affect scheduling are implemented on the operating system they are actually working with or are considering purchasing.

Scheduling is especially important in real-time systems, where tasks are expected to run in a timely and deterministic manner. Understanding how the operating system we are working with schedules tasks, and then choosing the proper scheduling technique makes the difference between a real-time system that works well and one that is inefficient or even fails.

---

**About QNX Software Systems**

QNX Software Systems is the leading global provider of innovative embedded technologies, including middleware, development tools, and operating systems. The component-based architectures of the QNX® Neutrino® RTOS, QNX Momentics® Tool Suite, and QNX Aviage® middleware family together provide the industry’s most reliable and scalable framework for building high-performance embedded systems. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle telematics and infotainment systems, industrial robotics, network routers, medical instruments, security and defense systems, and other mission- or life-critical applications. The company is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

[www.qnx.com](http://www.qnx.com)

© 2011 QNX Software Systems GmbH & Co. KG, a subsidiary of Research In Motion Ltd. All rights reserved. QNX, Momentics, Neutrino, Aviage, Photon and Photon microGUI are trademarks of QNX Software Systems GmbH & Co. KG, which are registered trademarks and/or used in certain jurisdictions, and are used under license by QNX Software Systems Co. All other trademarks belong to their respective owners. 302203 MC411.90