Challenges Using Linux as a Real-Time Operating System

Michael M. Madden*
NASA Langley Research Center, Hampton, VA, 23681

Human-in-the-loop (HITL) simulation groups at NASA and the Air Force Research Lab have been using Linux as a real-time operating system (RTOS) for over a decade. More recently, SpaceX has revealed that it is using Linux as an RTOS for its Falcon launch vehicles and Dragon capsules. As Linux makes its way from ground facilities to flight critical systems, it is necessary to recognize that the real-time capabilities in Linux are cobbled onto a kernel architecture designed for general purpose computing. The Linux kernel contain numerous design decisions that favor throughput over determinism and latency. These decisions often require workarounds in the application or customization of the kernel to restore a high probability that Linux will achieve deadlines.

I. Introduction

The human-in-the-loop (HITL) simulation groups at the NASA Langley Research Center and the Air Force Research Lab have used Linux as a real-time operating system (RTOS) for over a decade [1, 2]. More recently, SpaceX has revealed that it is using Linux as an RTOS for its Falcon launch vehicles and Dragon capsules [3]. Reference 2 examined an early version of the Linux Kernel for real-time applications and, using black box testing of jitter, found it suitable when paired with an external hardware interval timer. Though the real-time capabilities of the Linux Kernel have since improved with the introduction of the CONIFG_PREEMPT_RT Patch [4], the kernel has also increased in complexity and has adopted a number of features that maximize throughput at the expense of latency and determinism. Therefore, additional OS configuration and application design principles are required to assure operation that approximates hard real-time. In fact, Ref [5] states:

> The real-time scheduling policies in Linux provide soft real-time behavior. Soft real-time refers to the notion that the kernel tries to schedule applications within timing deadlines, but the kernel does not promise to always be able to fulfill them. Conversely, hard real-time systems are guaranteed to meet any scheduling requirements within certain limits. Linux makes no guarantees on the ability to schedule real-time tasks. The Linux scheduling policy, however, does ensure real-time tasks are running whenever they are runnable. Despite not having a design that guarantees hard real-time behavior, the real-time scheduling performance in Linux is quite good. The 2.6 kernel is capable of meeting very stringent timing requirements.

This paper highlights aspects of the Linux Kernel that require special attention for real-time applications and covers more topics than addressed by the *Build an RT-application HOWTO* of the *Real-Time Linux Wiki* [4].

II. Characteristics of an Real-Time Operating System (RTOS)

To assess the real-time capabilities of the Linux kernel, it is necessary to establish a definition of a real-time operating system. The paper uses the following characteristics to assess Linux as a Real-Time Operating System (RTOS):

- Allows the real-time application to reserve resources, especially memory, during initialization. Resource allocation is often a non-deterministic operation and should therefore be avoided during real-time processing.
- Allows the real-time application to lock its machine code and data into physical RAM during initialization in order to avoid page faults during operation.
- Allows isolation of one or more CPUs for running one or more real-time applications.
- Enables real-time applications and asynchronous applications to coexist while assuring that real-time applications have access to the processor when needed.
- The scheduler allows application developers to establish the execution precedence of applications, both real-time and asynchronous, that assures real-time applications can complete execution prior to their deadlines.
- Limits interference by the operating system while real-time applications are running.

* Chief Engineer for Modeling and Simulation; Senior Member of AIAA.

III. Kernel Preemption: Mainline Kernel vs. CONFIG_PREEMPT_RT patch

To assure that user real-time tasks get access to the processor when needed, an RTOS either isolates the user real-time task from the kernel's own processing or permits the user real-time task to preempt most kernel operations. The latter is called a preemptible kernel and is the mechanism used to support real-time applications on the Linux kernel. The mainline kernel, which is used in most Linux distributions, has three preemption configurations [6]:

- CONFIG_PREEMPT_NONE is the traditional preemption model that is optimized for throughput. Kernel operations are non-preemptible. The typical latencies introduced by kernel operations, including scheduling, are typically less than 200 microseconds on modern ARM processors [7]. However, the kernel offers no guarantees and the maximum latency experienced by user applications over long periods can reach hundreds of milliseconds [4].
- CONFIG_PREEMPT_VOLUNTARY augments the traditional kernel with additional rescheduling points. In other words, the kernel remains non-preemptible, but voluntarily yields the processor during some operations. The added rescheduling points are designed to minimize scheduling latency.
- CONFIG_PREEMPT activates kernel modifications that make sections of the kernel preemptible. Latencies are an order of magnitude lower than the nonpreemptible kernel. This option, however, still makes some decisions that favor throughput over determinism which result in larger bounds on latency than may be acceptable for some hard real-time applications. The Real-Time Linux Wiki FAQ [4] states that this option drops worst-case latency to the order of milliseconds. Modern ARM processors show typical latencies are O(10 microseconds), and the worst-case maximum latency is in the neighborhood of 100 microseconds [7].

Linux distributions that target servers typically use the non-preemptible kernel (CONFIG_PREEMPT_NONE or CONFIG_PREEMPT_VOLUNTARY) in order to maximize throughput. Distributions targeting desktops use CONFIG_PREEMPT_VOLUNTARY or CONFIG_PREEMPT to provide greater responsiveness to the user.

The CONFIG_PREEMPT_RT project maintains a patch to the mainline Linux kernel that makes the kernel fully preemptible and that favors reduced latency and determinism over throughput. In fact, the CONFIG_PREEMPT option of the mainline kernel is a subset of the CONFIG_PREEMPT_RT patch. The CONFIG_PREEMPT_RT patch makes the following changes to the kernel [4]:

- Replaces locking-primitives in the kernel with rtmutexes to make the locks preemptible.
- Critical sections protected by spinlock_t and rwlock_t are now preemptible.
- Implements priority inheritance for in-kernel spinlocks and semaphores.
- Converts interrupt handlers into preemptible kernel threads.
- Divides timeouts from kernel timers and converts the kernel timers to high resolution. As a side effect, user-space POSIX timers become high resolution.

Under CONFIG_PREEMPT_RT, some resource management functions of the kernel, especially memory management, continue to have indeterminate maximum latencies. However, the CONFIG_PREEMPT_RT patch assumes that real-time applications will acquire their resources prior to entering real-time operation. ARM processors show that the maximum interrupt response time under the CONFIG_PREEMPT_RT patch is 37% to 72% lower than under the CONFIG_PREEMPT kernel option [7]; task switch time for CONFIG_PREEMPT_RT and CONFIG_PREEMPT where nearly equal. The Open Source Automation Development Lab (OSADL) [8] performs daily latency monitoring of a variety of stable kernels with the CONFIG_PREEMPT_RT patch over a large variety of process boards. On March 16th 2016, the maximum latency ranged from 40 microseconds on an Intel Allagash board using a 2 GHz Intel Xeon processor to 400 microseconds on an ASUSTek N3150M-E board using a 1.6 GHz Intel Celeron processor. However, the ASUSTek board has a System Management Mode enabled BIOS that may be a contributing cause of longer latencies as described in the next section.

IV. Hardware Selection and Real-Time

Though the CONFIG_PREEMPT_RT patch creates a fully preemptible kernel, real-time characteristics of the system still depend on the underlying hardware, the configuration of that hardware, and the actions taken by device drivers for the hardware. Many of these issues are generally applicable to all real-time operating systems (RTOS). For one, latencies are partly a function of the underlying speed of the hardware. The hardware can also provide configurable features that degrade determinism or increase latency. For example, power management features, such as CPU throttling, or virtualized cores, such as hyper-threading, add variability to runtime and responsiveness. Furthermore, if a hardware device occasionally monopolizes an on-board resource such as a PCI bus, then latencies may also increase for real-time applications. Firmware, such as the BIOS, may contain features that occasionally and unpredictably suspend the operating system to perform low-level functions. A common example of interrupting firmware is a BIOS that utilizes the System Management Mode of x86 processors. System Management Mode may

be used to perform such functions as power management, device control and housekeeping, device emulation, or software correction of device defects. SMM functions are often executed aperiodically and may take 100's μs to complete [4]. Lastly, the available drivers for devices on the system must continue to operate correctly when preempted and must not contain code that prevents preemption for actions that take long periods of time. Nearly all modern Linux device drivers are written in two halves where the top half is an interrupt service routine and the bottom half is executed in the context of a kernel thread. The top half performs a minimal amount of work; typically it moves data to a device specific buffer in the kernel, schedules the bottom half, and exits. The bottom half performs the remaining work of the device driver. The two-halves design is well suited for the preemptive kernel of the CONFIG_PREEMPT_RT patch so the majority of Linux device drivers should perform correctly and not interfere with real-time operation.

V. Linux Memory Management

A real-time application typically reserves the memory it needs during initialization and locks the memory into physical RAM. Linux does provide system calls to request memory and to lock memory. However, two Linux features cause memory requests and locking to work differently than other operating systems, thwarting simple attempts to pre-allocate memory. These features our its demand paging implementation and overcommit memory. Other Linux services are also not written with real-time applications in mind and may exercise memory management when called.

A. Demand Paging and Memory Locking

Demand paging is a technique used by many operating systems, in which the operating system defers loading each page of the application's virtual memory into physical memory until the application accesses the page. However, in many early and existing implementations of demand paging, the portion of the application's virtual memory not present in physical memory is mapped to a 'swap' file in the filesystem. Furthermore, some operating systems also map executable pages to their file images using the swap only for heap and stack pages. Linux takes demand paging a step further and initially maps stack and heap pages not to the filesystem but to the a single write-only page filled with zeros. The application's virtual memory pages are not mapped into physical memory or the swapfile until the application attempts to write to the page. This generates a page fault and Linux responds by remapping the virtual page from the zero-value page to a physical page. From that point forward, the page also becomes eligible to be paged out of physical memory into the swap file. In this way, an application does not immediately consume either physical memory or swap space when it requests large blocks of memory that it may or may not use later. This strategy works well for Linux's target use as a general-purpose operating system that manages large numbers of concurrently running applications. However, it thwarts the memory strategy commonly used by real-time applications. A real-time application attempts to allocate all the memory it will require during initialization to avoid requests for memory during operation since the execution time of memory requests is potentially unbounded.

In traditional, memory management implementations using demand paging, allocating memory upfront was still not enough since the operating system would initially map that memory in the swap file; also, only the portions of the application that were run during initialization may be present in physical RAM with the remainder mapped to the applications image on disk. An RTOS using demand paging would also need to provide a mechanism that forced the RTOS to place all of the application's virtual memory into physical RAM. Furthermore, a real-time application would need a guarantee from the operating system that it would not later swap portions of the application's virtual memory back to the filesystem. This mechanism is most commonly called a memory lock. Linux does provide memory locking using the mlockall() system call. A real-time application should call mlockall(), shortly after the start of the process or thread, with the options MCL_CURRENT | MCL_FUTURE. Those options assure that all current and future pages allocated to the process or thread are locked into physical RAM. This call is recommended even when the system is configured with no swap because Linux can still swap executable pages to their file images in the file system. Prior to kernel 2.6.9, only privileged processes (e.g., run by the root user) could lock memory. With later versions, unprivileged processes can lock up to RLIMIT_MEMLOCK bytes of memory. The default for RLIMIT_MEMLOCK is typically 64 kilobytes, so the limit may need to be raised where real-time processes are unprivileged. Nevertheless, the mlockall() call only locks the application's pages as mapped by Linux. Therefore, virtual memory pages that are mapped to the read-only zero page, due to delayed page allocation, remain mapped to that page and they will generate page faults on first write to the page during operation. Then, Linux will perform both the page allocation and the page lock causing added latency.

Linux provides no kernel options or parameters that disable the deferred page allocation behavior of its demand paging implementation. Therefore, real-time applications must take three extra steps to assure all the memory the application has requested is ready for use before entering operation. First, when the application allocates memory on the heap, the application must also write to every page of the requested allocation. Second, the application must include

a function that writes to the maximum stack space that the application will use during operation. Third, the application may need to fully populate user-space buffers managed by the kernel such as socket send and receive buffers.

B. Overcommit Memory

The Linux Kernel takes advantage of its demand paging implementation by allowing applications to request more virtual memory than the total virtual memory (physical plus swap) of the system. This feature is called overcommit memory. The operating system makes an assumption that all running applications will not utilize all the memory that each application requests and that those applications will not all run at their maximum memory utilization at the same time. Overcommit memory has undesirable side effects for real-time applications. It makes it more difficult to detect and debug over-allocation of memory. With overcommit enabled, memory requests [e.g., malloc(), new()] will nearly always return success as over-allocation first occurs. In the default configuration, Linux will allow the total requested virtual memory to exceed the system's virtual memory by an amount equal to the swap plus 150% of physical memory. Over-allocation may not be realized until a write-to-memory actually forces a page fault when system virtual memory has been exhausted. When this occurs, the kernel responds by killing a process, and that process may not be the one performing the write operation. Thus, the failure may not occur at its root cause and may occur in a different process. Furthermore, to detect this condition during verification and validation, one needs a test that will exercise all the memory allocated by all the applications. Additionally, overcommit memory causes underreporting of memory utilization. The reported memory utilization of an application counts only the actual page allocations and does not include the deferred page allocations. Again, worst-case memory utilization of an application becomes evident only under tests that exercise all the memory allocated by the application. Fortunately, Linux provides a kernel parameter, overcommit_memory, with a setting to disable memory overcommit. Disabling memory overcommit is recommended for real-time applications.

C. System Calls and Page Faults

Some system calls will perform memory allocation, which can therefore lead to a page fault. These calls are to be avoided or the application must take other actions to ensure that memory allocation does not occur during these calls. Unfortunately, there is no comprehensive list of Linux system calls that may generate a page fault. Further, the manual pages for system calls often do not explicitly state whether the call can cause a page fault. Instead, this information must be inferred from the interface design of the call, from the description of the actions taken by the system call, or from the description of the options for the system call. System calls for IP-based communication are among those that can cause a page fault because Linux dynamically sizes the send and receive buffers. Therefore, a real-time application either should avoid IP-based communications or should exercise communications during initialization in order to grow the send and receive buffers to the size required by the application.

VI. Timers

Real-time applications need high-accuracy timers to assure that periodic operations step through time correctly and to assure other time-critical operations. This section discusses the timers available in the Linux Kernel and a Linux feature called timer slack that can affect timing in processes without a real-time scheduling policy.

A. The Software Clock and the High Resolution Timer

Linux defines a software clock for handling temporal events managed by the kernel like scheduling, sleeping, timeouts, and measuring CPU time. The software clock counts time in units of jiffies. The length of a jiffy differs with kernel versions and hardware platform but since 2.6.20, the Linux kernel supports jiffy lengths that correspond to clock frequencies of 100 Hz (i.e., 1 jiffy = 10 milliseconds), 250 Hz, 300 Hz, or 1000 Hz. On modern x86 and PowerPC hardware using the 2.6 or later kernel, a jiffy is typically 1 millisecond. Thus, any system call that relies on the software clock for timing, cannot time its activity with a precision better than one jiffy.

However, starting with the 2.6.21 kernel, Linux added support for high resolution timers. The interface for the high resolution timer allows for nanosecond precision but the actual precision of the timer will depend on the hardware. When the high resolution timer is enabled (via the CONFIG_HIGH_RES_TIMERS kernel configuration parameter), the kernel uses the high resolution clock instead of the software clock to manage some temporal events. Moreover, select system calls have been modified to use the high resolution timer including but not limited to select, poll, epoll_wait, nanosleep, and futex. Thus, the timeout of a select() call can achieve an actual precision better than a millisecond. For more information, see the time(7) man page.

B. Timer Slack

Adding the high resolution timer makes it possible to wake-up the CPU more frequently. This can both reduce efficient use of the CPU and increase power consumption. To combat this, a feature known as timer slack was added to the kernel beginning with 2.6.22. Timer slack represents a length of time, within which closely spaced timer events will be consolidated. The default time length is 50 microseconds but is configurable for each thread starting with the 2.6.28 kernel. When a thread causes the setting of a new timer event, the kernel will examine whether a timer event already exists within the range of the requested time plus the slack. If yes, then the kernel will assign the new request to the existing request and the request will be serviced at a time later than requested. Thus, timer slack introduces additional jitter to requested timer events. Per the prctl(2) man page, timer slack is not applied to threads with real-time scheduling policy, and, therefore, only affects threads with normal policy (see section VII.A for descriptions of scheduling policies). Though timer slack will not affect real-time threads in an application, it can affect asynchronous threads with normal policy. If the system relies on soft deadlines for those asynchronous threads, then it may be necessary to schedule those threads with a real-time policy to avoid the additional jitter that timer slack can introduce.

VII. Process Scheduling and CPU Isolation

In this section, the term "task" is used to refer to either a process or a thread. The Linux scheduler treats both entities equally. A real-time task executes most quickly and can better meet deadlines when it is the only task on a processor and is not interrupted by the operating system. However, modern real-time systems may execute multiple real-time and asynchronous tasks over multiple processor cores. This section explores the Linux features that permit the developer to prioritize and schedule a mixture of real-time and asynchronous tasks and to isolate processors for one or more tasks. This section also explores the circumstances under which the operating system interrupts tasks and how those interruptions can be reduced.

A. Scheduling Policy and Priority

The scheduling policies in the Linux Kernel are divided into real-time policies and normal policies. All long-term Linux kernels provide two real-time policies: SCHED_FIFO and SCHED_RR. Linux kernel 3.14 introduces a third real-time policy SCHED_DEADLINE. SCHED_RR and SCHED_FIFO use a static priority in the range 1 (lowest) to 99 (highest) and are always higher priority than normal tasks which have static real-time priority of zero.[†] The SCHED_FIFO policy runs a task until it is preempted by a higher priority task, the task blocks, the task yields the processor, or the task terminates. In other words, SCHED_FIFO uses no time slices, and a task can come close to monopolizing the processor. (Linux has mechanisms that prevent complete monopolization of the processor by a task and these are discussed later.) Even the next task of equal priority under SCHED_FIFO is granted the processor only when the current task blocks, sleeps, yields, or terminates. SCHED_RR behaves as SCHED_FIFO with the addition of time slices. Each task can monopolize the processor only for the length of its time slice; it is then preempted if other tasks of the same priority are on the run queue. Before the 3.9 kernel, the default time slice was fixed at 100 milliseconds; afterwards, it can be set using the sched_rr_timeslice_ms kernel parameter. However, Linux will also adjust the time slice for an RT task if it is assigned a nice value other than zero using setpriority(); the algorithm will linearly assign a time slice between 10 milliseconds when nice is +19 to 200 milliseconds when nice is -20. The SCHED_DEADLINE policy schedules periodic tasks with an enforced relative deadline within the period; in order to schedule multiple SCHED_DEADLINE tasks, the kernel also requires an estimate of the maximum allowed runtime of the task within the period. The schedule then runs each task based on an earliest deadline first algorithm. The scheduler does performs schedulability checks when a new task is added with the SCHED_DEADLINE policy to assure deadlines for all SCHED_DEADLINE tasks. All tasks with a SCHED_DEADLINE policy are treated equally and, as a group, have higher priority than tasks under any other policy in order to, again, assure that the deadlines can be met. This policy was first introduced in the 3.14 kernel. Prior to Linux Kernel 2.6.12, only a privileged task could execute with a real-time scheduling policy. In later kernels, unprivileged tasks can execute with a SCHED_RR or SCHED_FIFO policy but the priority of those tasks cannot exceed the RLIMIT_RTPRIO resource limit; the SCHED_DEADLINE policy remains limited to privileged tasks.

There are three normal scheduling policies: SCHED_OTHER, SCHED_BATCH, and SCHED_IDLE. The SCHED_IDLE policy runs a task only when a processor would otherwise be idle. All tasks under SCHED_IDLE are

[†] The Linux kernel is not consistent in how it represents priorities between user interfaces and kernel structures. In some contexts, real-time priorities are presented in reverse order in which 1 is the highest priority and 99 is the lowest, then normal priorities have a range of 100 to 139 that correspond to the nice values of -20 (100) to 19 (139).

treated equally and executed round-robin; priorities are ignored. SCHED_OTHER and SCHED_BATCH both utilize an additional dynamic priority. The scheduling algorithm sets and updates the dynamic priority for each task based on a set of objectives such as responsiveness or fairness. This dynamic priority is influenced by the setting of a *nice* value in the range -20 (highest) to +19 (lowest). The difference between SCHED_OTHER and SCHED_BATCH is that tasks in SCHED_BATCH are automatically treated by the scheduling algorithm as if they are non-interactive (i.e., processor bound). Under most scheduling algorithms, tasks in SCHED_BATCH are slightly disfavored in scheduling decisions relative to those in SCHED_OTHER because most scheduling algorithms are designed to favor interactivity (i.e., I/O bound tasks).

Real-time systems should use the real-time scheduling policies for their periodic and time-critical functions. However, in these systems, understanding how Linux performs scheduling for normal operations is also important since the system may have asynchronous activities to perform and because some kernel tasks are scheduled as normal tasks by default.

1. *The Completely Fair Scheduler (CFS)*

Since Linux 2.6.23, the default scheduler for normal priority tasks is the Completely Fair Scheduler (CFS). The goal of the CFS is to model an "ideal, precise multi-tasking CPU" [9]. In the simple case where all the tasks have the same nice value, the CFS would attempt to divide processor time equally among the tasks running on a processor. The CFS does this division on an epoch described as the CFS scheduler latency, because it would represent the longest time period that a task will wait to run. The default differs with different kernels as CFS has undergone a number of revisions. In the 3.2 and 3.14 kernels, the default is a function of the number of processors: 6 milliseconds * [1 + $\log_2$(#CPUs)]. However, to avoid rapid and inefficient context switching under heavy loads, the CFS also defines a minimum runtime (a.k.a. granularity) for a task before it can be preempted. Again, in the 3.2 and 3.14 kernels, this default is a function of the number of CPUs, 0.75 milliseconds * [1 + $\log_2$(#CPUs)]. Thus, with the default values, once the load becomes larger than eight tasks (of equal nice value), the epoch on a processor will effectively increase to the granularity multiplied by the number of tasks. To give each task its fair share of the processor, CFS tracks the planned runtime of each task and orders tasks in a red-black tree (rbtree) by their planned runtime. To apply nice values and maintain fairness in the face of real-world changes to the rbtree of runnable tasks, CFS defines a virtual runtime rather than using the actual runtime. For example, nice values are converted to a weight factor on the increment of virtual runtime for a task. Tasks with a lower nice value (and thus higher priority) will see their virtual runtime increment more slowly than actual runtime, and those tasks with a higher nice value will see their virtual runtime increment more quickly than actual runtime. Since the CFS is designed to give tasks equal access to the processors, CFS implicitly boosts the priority of tasks that frequently block or sleep since the CFS algorithms effectively credits their unused time sliced when computing a new virtual runtime. This makes it more likely an I/O bound task will preempt a CPU-bound task when the I/O bound task wakes up.

2. *Should asynchronous tasks have normal or real-time policy?*

Here, a task is considered asynchronous if its start and completion time is permitted to vary. Such tasks may still be subject to an average execution rate with soft deadlines or a hard deadline that spans many execution frames but is many times larger than the runtime of the task. Such a task may run without problems using the CFS scheduler. However, there are some advantages to running these tasks under a real-time policy. These include:

- The task will have priority over "normal policy" kernel tasks, increasing the repeatability of the task's execution.
- The real-time policy takes less computation than CFS when scheduling, O(1) vs. O(log N).
- The real-time policy does not attempt to allocate the processor fairly. Thus, the real-time policy tends to favor CPU-bound tasks over I/O bound tasks.
- The real-time policy allows the developer to select between no timeslices (SCHED_FIFO) or fixed timeslices (SCHED_RR). CFS uses timeslices that vary during runtime based on number of tasks in the run queue and their nice values.
- Allows for finer control over the order in which tasks execute.
- Real-time policy threads are not affected by timer-slack (see section VI.B) and, therefore, experience less jitter in the expiration of timer events.

However, the application developer must also remain careful not to oversubscribe the processor with real-time tasks. This may adversely affect the synchronous tasks if real-time bandwidth enforcement remains enabled (see section VII.E). Furthermore, if one wants to limit the CPU bandwidth of the asynchronous tasks as a group, then one can use real-time group scheduling (see section VII.F).

B. Scheduling Events

In general, the Linux kernel executes the scheduler and, thus, interrupts the running task for the following events:

- Periodically, at the rate of the system clock, which is typically 1 millisecond on the PowerPC and Intel x86 platforms.
- At the expiration of a high resolution timer set the by the scheduler. If the scheduler determines that the running task will exhaust its time slice before the next system clock update, then it will set a high resolution timer to trigger when the time slice expires.
- When a task terminates, blocks, sleeps, or yields the processor. These events remove a task from the run queue.
- When a task wakes up (from a block or sleep) or when a new task is spawned. These events place a task onto the run queue.

Therefore, these are the only events at which tasks can be preempted or selected to run. These events also will interrupt the current task; thus, the scheduler will interrupt the running task at least every millisecond on Intel and PowerPC platforms. The kernel processing associated with the timer tick can consume up to 1% of the processor [10]. Linux, however, does provide a kernel option starting with the 3.10 kernel to disable the timer tick on processors running a single task.

1. Disabling Timer Tick (CONFIG_NO_HZ_FULL)

The kernel option to use adaptive clock ticks (which is sometimes referred to as NOHZ or NO_HZ) can reduce timer tick interruptions of a computationally intensive task or a real-time task. This can reduce jitter and improve the throughput of an isolated task on a processor. This option is activated by setting the kernel configuration option CONFIG_NO_HZ_FULL. The option must then be applied to a list of processors using the no_hz_full boot parameter or can be applied to all but the "boot" processor by setting the CONFIG_NO_HZ_FULL_ALL kernel option. However, disabling the timer tick does have some limitations and drawbacks. First, the timer tick cannot be disabled on the "boot" processor (usually CPU 0). The boot processor must retain the system clock interrupt to maintain time keeping for the benefit of the other processors that disable the timer tick. Second, the kernel disables the timer tick only when there are zero or one tasks in its run queue for the processor. Linux assigns a set of threads to each processor (see section E) though they spend much of the time on the wait queue. Nevertheless, when one or more of these threads wake, they may also temporarily re-enable the timer tick. When there are zero tasks on the run queue and the timer tick is disabled, then no activity may occur on the processor for a period of time and some processors may enter a power save mode unless explicitly disabled via firmware or other means. Thus a real-time task with processor margins equivalent to milliseconds could find the processor in a throttled state when the task is next ready to run and will incur latency as the processor returns to full speed. Third, the option does not completely disable the timer tick but actually sets it to 1 Hz; this is currently necessary to maintain correct operation of the scheduler and scheduler statistics. Lastly, the adaptive tick option incurs computational penalties for user/kernel transitions, clock reprogramming, and idle loop transitions. The kernel documentation (Documentation/timers/NO_HZ.txt) contains a full list of limitations. This document does warn about the impact of adaptive clock tick on POSIX CPU timers but this may not apply under the CONFIG_PREEMPT_RT patch whose timer changes are reported to replace the system clock with high resolution timers in the user-space POSIX timers (see section III). In any case, real-time applications should use high resolution timers and related functions (e.g., nanosleep) over user-space POSIX timers and other related functions that normally rely on the system clock.

C. Scheduling Latency

The scheduling latency discussed here is the latency between the waking of a real-time task with higher priority than the tasks in the run queue and when that task begins execution. For the Linux Kernel, scheduling latency is primarily determined by the hardware. Measurements on a modern ARM processor showed worst-case task switch latency of about 75 microseconds [7]. OSADL [8] performs daily monitoring of worst-case latency, which includes both interrupt and scheduling latency, on a variety of processor boards. OSADL results typically range from O(10 us) to O(100 us) though processor boards with the System Management Mode feature exhibit maximum latencies approaching (and sometimes exceeding) 1 millisecond (see section IV). Real-time developers should measure the scheduling latency on their target hardware and use the result in decisions on how to schedule tasks and apportion them to processor cores. Schedule latency also has a major influence on the decision to have real-time tasks sleep or busy wait when they are done with their current processing.

D. Sleep or Busy Wait?

When a task chooses to sleep, it is also yielding the processor and the scheduler can select another task to run. Thus, when the task wakes, it is subject to scheduling latency before it resumes operation (assuming it is the highest

priority real-time task). This latency is negligible when it is the only running task on the processor, but the kernel also has a number of sleeping tasks that occasionally may wake during the sleep period and inject typical schedule latencies. Therefore, a task might be better served performing a busy wait loop than executing a nanosleep() function when the sleep time is less than the bounds on schedule latency for the system. Furthermore, if the sleep time is longer but latency on wake must be minimized, then the task may have to use a combination of sleep and busy wait. The task subtracts the schedule latency from the sleep time and, once the task wakes up, it busy waits until the target time of its next operation. In this circumstance, the task should only execute a sleep if the time to next operation is greater than twice the schedule latency. Otherwise, any other task that might be scheduled during the sleep and, subject to its own schedule latency, may perform little or no work before being preempted. However, monopolizing the processor with by busy waiting may not be a viable solution either. Depending on the kernel version, hardware drivers, and other software that the real-time application interacts with, processor monopolization by busy waiting may lead to instability and failures. In fact, the kernel contains real-time bandwidth limits to prevent processor monopolization by default.

E. Real-Time Bandwidth Limits

By default, the Linux kernel limits the CPU utilization of real-time tasks as a group. The limit is 95% utilization and is enforced every second. The kernel parameters sched_rt_period_us and sched_rt_runtime_us configure real-time bandwidth limits. sched_rt_period_us sets the period of enforcement (default of 1 second), and sched_rt_runtime_us sets the utilization limit (default of 950,000 microseconds). These kernel parameters are system-wide limits that apply to all processors. Using the defaults, real-time processes monopolizing the processor could find themselves suspended for 50 milliseconds every second. Therefore, hard real-time systems should disable the limits by setting sched_rt_runtime_us is set to -1. However, if removing the limits leads to instability or failure, then developers should assure that the real-time processes yield the processor often enough and for enough length of time that their processor utilization along with real-time kernel tasks remains in the neighborhood of 95% each second.

F. Control groups and real-time group scheduling

Control groups are a kernel feature that allows the administrator to set resource utilization limits for a group of tasks. Real-time group scheduling is a real-time policy extension to control groups that allow a control group to contain tasks under a real-time policy. In fact, real-time bandwidth limits (see section VII.E) are, in effect, a global form of real-time group scheduling. Section VII.A.2 describes one situation, real-time scheduling of asynchronous threads, where real-time group scheduling can be used to assure that the group does not starve normal tasks or adversely impact critical real-time tasks by exhausting real-time bandwidth limits.

G. Softirq Processing in the Linux Kernel

The softirq is an important feature of the Linux kernel. It was primarily designed for execution of the bottom half of device drivers, i.e., the softirq performs the majority of the work that is required to handle a device interrupt and that can be performed in a context with the hard interrupts enabled. However, the Linux kernel also uses the softirq to defer other forms of high-priority work. Because the softirq was developed primarily for processing the bottom half of device drivers, a softirq action has the restriction that it must be performed on the same processor that raised the hardware interrupt. Additionally, a softirq task is restricted from sleeping.

The kernel maintains an array of softirqs with a hard limit of 32 entries though most kernel versions are compiled using ten or fewer. Valid softirq entries must be defined when the kernel is compiled, and they must be consecutive entries. The kernel does not permit activation of unused softirq entries at runtime. The softirq entry is itself a registered pair of an action (i.e., a function pointer) and a pointer to the data that the action operates on. Though the softirq entries are reserved at compile time, the registration of the action-data pair to a softirq entry occurs at runtime. Two of the softirq entries are normally reserved to operate on a queue of tasklets as their action. The tasklet provides a mechanism for third-party drivers to implement their bottom-half under a softirq without requiring a new softirq entry and, thus, a recompile of the kernel.

A softirq is inactive until it is 'raised'; then, it is placed in the 'pending' state. Pending softirqs are executed in the order of their position in the softirq array. In other words, a pending softirq at index 0 of the array always executes before the softirq at index 1. The kernel events that trigger softirq execution are:

- when an interrupt service routine (ISR) returns
- when some kernel subsystems, like the networking subsystem, make a call to perform bottom-half processing
- when the ksoftirqd thread is scheduled to run

The first two conditions are also the conditions under which a softirq is raised. In other words, actions that raise a softirq also trigger softirq execution. When triggered, all pending softirqs are executed, not just deferred work that

may be associated with the triggering event. For example, an ISR servicing a serial port interrupt may trigger, on return, both the bottom-half of the serial port driver and deferred work from the network stack if the associated softirq entry is pending. In fact, a user real-time application can be hijacked to run the pending softirqs in its context if the application calls kernel functions that may perform softirq processing such as send().

The softirq complicates efforts to predict when and for how long the kernel may interrupt a user real-time task. Thomas Gleixner, a maintainer of the RT Preempt Patch, made the following observation about softirqs [11]:

> First of all, it's a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them. It's extremely hard to get the parameters right for a RT system in general. Adding something which is obscure as soft interrupts to the system designers todo list is a bad idea. [sic]

The mainline, nonpreemptible kernel uses a bifurcated priority scheme for executing softirqs. As long as demand for softirq processing is light, the softirq can only be interrupted by a hard IRQ interrupt service routine (ISR). To assure that the softirq performs bottom half processing for each hard IRQ received, the kernel maintains a count for each IRQ. The softirq will decrement the count when it performs processing for the IRQ and will raise itself if the count remains positive. In this scenario, a heavy load of external events could cause the hard IRQ ISR and the softirq to monopolize the processor. To prevent this occurrence, the mainline kernel limits the number of consecutive softirq actions that may occur at heightened priority. The kernel offloads the remaining softirq actions onto a kernel thread named ksoftirqd, of which there is one thread per CPU since a softirq action must run on the same processor that raised the softirq action. This thread has the normal SCHED_OTHER policy with a nice value of 19. So, the softirq now shares the processor with other normal tasks and its nice value will cause it to have a lower fraction of the available CPU than other normal tasks.

How the preemptible kernel (CONFIG_PREEMPT_RT) handles softirq differs based on kernel version. The kernel 2.6.x versions of the CONFIG_PREEMPT_RT patch create a softirq thread for each softirq entry (on each processor). These threads have a SCHED_FIFO real-time policy. The priority assigned to the thread differed among patch versions but was most often 1, 24, or 50.[‡] However, this proliferation of threads was perceived as a performance issue and, in early 3.x patches, the ksoftirqd thread on each processor is used to consolidate and operate all of the softirq entries. These ksoftirqd threads are also have a SCHED_FIFO policy and are given a default real-time priority of 1, 24, or 50 dependent on patch version. However, starting with the patch to the 3.6 kernel (and back ported to some earlier versions), each softirq entry is executed in the context of the thread that last raised the softirq entry. Thus, a user real-time task that raises a softirq entry (e.g., as a result of calling the networking subsystem) will execute that softirq in its context until complete or the task is interrupted by another task (e.g., a hardware IRQ task) that raises the same softirq. As a result, softirq locking was modified to prevent priority inversion when threads of differing priority raise the same softirq entry.[§]

H. Kernel Task Policy and Priority

In the preemptible kernel, kernel tasks can be divided into two groups, critical tasks with a real-time policy and other services with a normal policy. Kernel tasks using a real-time policy use the SCHED_FIFO policy since this more closely mimics how the task runs under the nonpreemptible kernel. Among kernel tasks with real-time policy, IRQ threads have a static priority of 50 by default, some per-CPU threads have a static priority of 99, and the priority of softirq threads differs by kernel version but is often 1, 24, or 50. Kernel tasks using a normal policy use SCHED_OTHER and tend to have a nice value of either zero or nineteen by default.

Any user real-time task will share a processor with other kernel tasks even when isolating the processor (see section VII.J). Though the CONFIG_PREEMPT_RT patch makes the Linux kernel real-time capable, the Linux kernel was not designed for real-time. Additionally, device drivers may also not be written with real-time applications in mind. Therefore, the system configuration least likely to encounter problems generally maintains real-time kernel tasks at higher priority and any user real-time tasks given priority equal to or greater than the real-time kernel tasks should behave similarly to those kernel tasks. In other words, any user real-time task with a priority over 49 should have a

[‡] Use of a priority of 50 was the default for softirqs until the 2.6.29 kernel. Then the decision was made to always use a lower priority for softirq threads than the priority used for IRQ threads.

[§] There are conflicting accounts of the history of softirq handling in CONFIG_PREEMPT_RT. In alternate accounts, all 3.x patches allowed the softirq entry to be run in the context of the thread that raised it but there was a single lock for the softirq table to prevent priority inversion. The change that the 3.6 kernel patch made was to split the lock for each softirq entry.

short duration of execution and limit its rate of execution so that its total processor utilization is only a few percentage points.[**] These tasks would either be I/O bound or spend a majority of the time sleeping.

However, these guidelines are not always realistic for an application. Real-time applications for aerospace use tend to be CPU-bound, long duration tasks. These applications may have time-critical sections, which should ideally have priority over all other tasks including the kernel. In these cases, the application developer can protect those time-critical sections from preemption by giving them a priority boost using a mutex configured with a priority ceiling. However, any time-critical section placed at static priority of 99 should avoid or carefully engineer loops, recursive calls, or other logic constructs that can result in infinite execution in the presence of an error or upset since this can cause the time-critical section to block out other tasks that could be used to recover the processor.

Depending on the kernel version and CONFIG_PREEMPT_RT patch version in use, user tasks coexisting with softirq processing can also become problematic. It is 'safe' for a real-time application to have priority over softirq threads since, in the mainline kernel, heavy loads can lead to softirqs experiencing long latencies until the CPU is idle (see section G). However, 'safe' only means that the kernel and device drivers do not experience a logic failure. Such long latencies can still lead to loss of data (e.g., lost packets). Furthermore, if the application explicitly or implicitly relies on softirq completion for nominal application processing, then the application can cause an issue if it prevents the softirq from executing. Performing a synchronous exchange of a large data block over Ethernet is one example where a conflict between user application and softirq may arise. In this scenario, if the data block exceeds limits for in-context transfer to the NIC, then only a portion of the block will be transmitted in-context and the kernel will raise NET_TX_SOFTIRQ to transfer the remainder of the block later. As with all softirqs, the raised softirq is restricted to running on the processor that raised it. Therefore, the softirq and the software application are now both in contention for the same processor. Under the 2.6 and early 3.x kernels, if the application has a higher priority than the softirq thread, then the softirq will not execute until the application next blocks, sleeps, or makes another system call that can raise a softirq like socket send.[††] In fact, if the application has priority over softirqs, then pending softirq work can accumulate; should the application be the next task to perform softirq processing, the application becomes that ‘randomly chosen victim’ to execute all that pending work. That will increase the jitter of the application and may cause it to overrun a frame. Thus, to assure the synchronous transfer of data in this example, the softirq processing could be set at a higher priority than the application. However, in early 3.x CONFIG_PREEMPT_RT patches, the priority cannot be set specifically for NET_TX_SOFTIRQ, and all softirq processing, performed by the ksoftirqd task, would then have priority over the application. This makes the application vulnerable to unbounded delays should the system become heavily loaded with various interrupt events. In the case of more recent CONFIG_PREEMPT_RT patches to the 3.6 and later kernels, the fact that most softirq processing is done in the context of the thread that raised the softirq makes it more difficult to balance application priority with NET_TX_SOFTIRQ priority since it places the developer in the position of identifying all the threads that could raise the NET_TX_SOFTIRQ. Moreover, when the system experiences a heavy load of external interrupts, it may begin to migrate all softirqs from the IRQ threads to the ksoftirqd task. Thus, the application remains vulnerable to unbounded delays if it uses a lower priority than ksoftirqd in order to guarantee priority to NET_TX_SOFTIRQ. The application may need to use other strategies that permit the ksoftirqd thread temporary priority over the application. One simple work around is to set the application to sleep for a short period after performing the synchronous write operation. This would provide an opportunity for the kernel to run the ksoftirqd task and operate any pending work under NET_TX_SOFTIRQ (assuming no other softirqs are pending and no other tasks with higher priority than ksoftirqd are in or added to the run queue during the sleep period).

I. Priority Inversion Prevention

The CONFIG_PREEMPT_RT patch replaces many locking primitives in the kernel with locks that prevent priority inversion by using priority inheritance. Therefore, priority inversion is normally not an issue when an application interacts with the kernel. For user-space applications, the mainline kernel provides a POSIX thread mutex that support priority inheritance as an option. Applications can, therefore, utilize POSIX thread mutexes configured for priority

[**] Developers can, of course, ignore this guideline and, further, can disable or strip kernel tasks from a processor to provide better assurances of processor availability and isolation for the real-time application. However, this requires in-depth knowledge of the kernel and its operations to assure the customizations will not adversely affect the stability and performance of the operating system. Customizations also decrease the applicability of community testing and usage that apply some confidence of long duration fault-free operation of the Linux kernel.

[††] Under specific circumstances, network activity performed by other processors or an in-stack transmission that occurs in response to received packets (e.g., TCP processing) may also force progress on transmitting the remaining block. This example assumes that these are not present or cannot be deterministic drivers of the remaining data.

inheritance to improve assurances that contention for application resources or serialization of code sections do not cause priority inversion among the application's tasks.

J. CPU Isolation under Linux

Linux provides two features for CPU isolation. The first is the isolcpus boot parameter. This parameter essentially sets the default affinity mask for tasks to exclude the listed processors so that the scheduler and load balancer will not assign tasks to the identified CPUs. The second mechanism is cpuset. The cpuset mechanism can be used to give a control group (see section VII.F) or list of process IDs access to a set of CPUs. The cpuset can be created during boot or during runtime. The cpuset, however, has a few drawbacks. The biggest is that there are corner cases in which processes that exist on the CPU prior to creation of the CPU set will remain on the CPU. This includes softirqs that continue to raise themselves and any pre-existing timers. Furthermore, for cpusets created after boot, there will be a latency in the migration of other processes currently running on the excluded CPUs to other CPUs. However, there are also some advantages to cpusets. These include the ability to load balance within the cpuset and runtime changes to the list of processes included in the cpuset. The isolcpus boot parameter and the cpuset are not mutually exclusive. A cpuset can include isolated CPUs. In fact, a user-space task can only be assigned to an isolated CPU by either changing its affinity mask or by inclusion in a cpuset, either of which includes the isolated CPU. The one side of effect of combining isolcpus with cpusets is that once a process is assigned to an isolated CPU, it cannot be migrated to another CPU. Both mechanisms, however, only guarantee exclusivity of the CPU from other user-space processes. The kernel also honors these settings for some of its tasks. But, the kernel can and does ignore the isolcpus and cpuset for other tasks. Specifically, the kernel does not honor these settings for the timer tick, the scheduler, inter-processor interrupts (IPIs), global workqueues, IRQ threads, softirq, and per-CPU threads. Nevertheless, there is interest from both the real-time and high-performance computing communities in reducing kernel interruptions on isolated processors so there is continuing work in reducing interruptions or targeting interruptions (i.e., changing some 'global' interruptions into interruptions smart enough to target affected CPUs). Therefore, with newer versions of the kernel there have been some reductions in interruptions by the kernel or new kernel options that result in reductions. The sources of kernel interruption are discussed below:

1. *Timer Tick and Scheduler*

Timer tick and scheduler interruptions are detailed in section VII.B. To summarize, the kernel schedules a timer tick for every millisecond (on computers using Intel or PowerPC processors), and this is one of the events that runs the scheduler. Newer versions of the kernel have an option that limits the timer tick to running every second. When the timer tick has been limited, the processor is said to be running in dynamic tick mode because the scheduler may still set the high resolution timer to wake it at a future point such as the end of the time slice (if any) for the task selected to run. The scheduler will also run when the running task blocks, sleeps, or terminates and when an event wakes up a task. These interruptions cannot be entirely avoided but can be reduced by assigning only one task to the CPU and running that task under the SCHED_FIFO policy.

2. *Per-CPU Threads*

On multiprocessor systems, the kernel creates a number of per-CPU threads either to perform housekeeping tasks on each CPU or to enable the kernel to distribute or load balance some services across processors. The per-CPU threads can be real-time or normal tasks. The normal tasks will not disrupt applications with real-time policy. However, some per-CPU threads have the highest real-time priority (99) and the RT Wiki [4] cautions against setting real-time applications to the highest priority since these high-priority kernel tasks tend to be critical management tasks. Thus, occasional interruption by per-CPU threads may not be avoidable. Moreover, it may be necessary for stability of the kernel that the real-time application keep its processor utilization below a cap so that lower priority per-CPU threads have an opportunity to run.

3. *IRQ Threads and Softirqs*

Whether an isolated processor is eligible to handle an interrupt is dependent on the underlying hardware. One some hardware platforms, only CPU0 can handle interrupts. On modern Intel platforms, the Advanced Programmable Interrupt Controller (APIC) allows the kernel to program which CPU services an interrupt. Other platforms may exist somewhere in between, such as using a fixed assignment of interrupts to CPUs via firmware.

For Intel platforms with APIC, the Linux kernel uses the programmability of APIC to implement IRQ load balancing that is separate from the scheduler's load balancer. However, the Linux implementation of IRQ load balancing also allows the CPU affinity of each IRQ to be set. Therefore, IRQ load balancing can be configured to respect the isolation of CPUs. On other platforms where interrupts can be serviced on CPUs other than CPU 0, the board support package or on-board firmware may provide the ability to configure IRQ handling such that it respects the isolation of CPUs.

Often the IRQ thread will raise a softirq to perform the bulk of the work for servicing the interrupt. As a result, the softirq is restricted to run on the same CPU that raised the softirq. Section VII.G describes how softirqs are represented in the kernel, which differs, by kernel version. Section VII.H further describes the balancing of user tasks with the softirq and the potential side effects that setting relative priorities alone cannot solve.

4. Interprocessor Interrupts and Global Workqueues

The kernel globally executes interprocessor interrupts (IPIs) and workqueues to perform some housekeeping tasks, especially for memory, that maintain a consistency across the processors in a multiprocessor system. There is no direct configurability of this behavior, and real-time applications must currently accept it as part of the normal overhead of the operating system.

5. Reserve CPU 0 for the Operating System?

One common practice for multiprocessor real-time systems running on the Linux Kernel (with or without the RT_PREEMPT patch) has been to avoid assignment of user-space applications to CPU 0 (a.k.a. the boot processor) and reserve CPU 0 for the operating system. This recommendation may be a holdover from older computing devices in which CPU 0 handled all hardware interrupts. The recommendation may also be based in the myth that real-time applications will experience higher latency on CPU 0 than on other processors due to the kernel functions normally performed on CPU 0. However, latency tests of the CONFIG_PREEMPT_RT patch have demonstrated no differences of significance in latency when real-time applications are run CPU 0 [8]. However, a developer may still choose to direct all IRQs to CPU 0 and place user real-time tasks on other processors to minimize interruptions of those tasks. Another argument for reserving CPU 0 for the operating system was that developers would be free to treat the operating system as a black box. However, earlier topics should expose the folly of that line of reasoning. Even so, reserving CPU 0 for the kernel can still provide some protection against overlooked and unwanted side effects when real-time applications cause delays in lower priority kernel activities. For this reason, it may remain a good practice to avoid placing user-space, real-time tasks on CPU 0 unless the other cores have reached project targets for maximum CPU utilization. Even then, user-space, real-time tasks on CPU 0 should be limited to lower than 70% CPU utilization assuming that kernel processing (including interrupts) can consume up to 30% of CPU 0 under typical loads.

K. Inheritance of Processor Affinity, Scheduling Policy, and Scheduling Priority

Tasks created using fork() inherit the processor affinity, scheduling policy, and schedule priority of the calling task. The calling task cannot change this behavior; it becomes the responsibility of the child task to change these task attributes if needed. By default, POSIX threads in Linux also inherit processor affinity, scheduling policy, and scheduling priorities of the creating thread. However, the scheduling policies can be set by the creating thread using the attribute argument of pthread_create(). The child thread remains responsible for changing the processor affinity. Developers need to account for this behavior when they use a real-time task to spawn asynchronous tasks.

VIII. Conclusion

The Linux Kernel, even with the real-time patch, is not designed to guarantee deadlines; it retains compromises the favor throughput over latency. Though the Linux Kernel is not optimized for real-time, real-time applications will often achieve deadlines at high enough probabilities, especially on modern hardware, that black box testing of limited duration may fail to expose interactions with the potential to generate large latencies. Therefore, developers need to understand Linux features that act contrary to the goals of low latency and determinism. The developer can then optimize the operating system and the application to minimize the effects of those features. Though the kernel is not designed to guarantee deadlines, developer optimizations may be able to minimize those probabilities so that they are equal to or less than probabilities for other types of system failures. In those cases and in applications better categorized as soft real-time, Linux may be a suitable operating system option for the real-time system. However, users that demand very low latency and very high reliability of deadlines may be better served using a proprietary real-time operating system.

Developers of real-time applications should make the following optimizations to the kernel or application to improve real-time performance:

- Use the preemptible kernel, either the CONFIG_PREEMPT option of the mainline kernel or apply the CONFIG_PREEMPT_RT patch. The choice depends on the latency tolerance of the application.
- The application should call memlockall(MCL_CURRENT | MCL_FUTURE) to assure that all current and future memory pages are locked into physical RAM. The kernel resource limit RLIMIT_MEMLOCK may also need to be increased in order to run real-time applications without elevated privilege.

- To overcome Linux's delayed paging, the application should, during initialization, write to each page of heap memory that it requests, execute a function that writes to the maximum stack size that it will use, and exercise system calls that rely on kernel buffers.
- Disable overcommit memory so that the Linux kernel detects over-allocation of memory when memory is requested rather than when it is written to.
- Use a real-time policy for time sensitive tasks, including background tasks that rely on system timers. The kernel resource limit RLIMIT_RTPRIO may need to be increased in order to run tasks with real-time policy without elevated privileges.
- Developers must balance the schedule priority of user real-time tasks against any kernel tasks that share the processor including IRQs and softirqs, especially if the application relies on those kernel tasks.
- Disable real-time bandwidth limits by setting the kernel parameter sched_rt_runtime_us to -1.
- Real-time tasks idling until the next deadline should sleep for the majority of their idle time rather than busy wait for the remaining duration. To minimize jitter around a deadline, combine sleep with a shortened busy wait.
- The isolcpus boot parameter or a cpuset can isolate one or more processors for real-time tasks.
- On platforms that support it, configure IRQ load balancing either to exclude IRQ processing from processors isolated for real-time tasks or to selectively place IRQ on processors with the tasks that utilize the associated device.

Additionally, developers may benefit from disabling the timer tick on processors running a single real-time task to reduce interruptions by the scheduler. Developers should also pay attention to scheduling attributes inherited by child tasks. Lastly, developers need to become familiar with how their target kernel version implements the softirq mechanism and how that implementation may result in undesirable side effects when a user real-time task relies on devices or kernel services that utilize a softirq.

IX. References

[1] Cunningham, K., Shah, G. H., Hill, M. A., Pickering, B. P., Litt, J.S., and Norin. S.. "A Generic T-tail Transport Airplane Simulation for High-Angle-of-Attack Dynamics Modeling Investigations", AIAA-2018-1168, https://doi.org/10.2514/6.2018-1168.

[2] Nalepka, J., Danube, T., Williams, G., Bryant, R., and Dube, T., "Real-time simulation using Linux", AIAA-2001-4185, https://doi.org/10.2514/6.2001-4185.

[3] Edge, J. "ELC: SpaceX Lessons Learned", Eklektix, Inc., URL: https://lwn.net/Articles/540368/, March, 6, 2013.

[4] "Real-Time Linux Wiki", https://rt.wiki.kernel.org/index.php/Main_Page [retrieved May 17, 2018].

[5] Love, R., *Linux Kernel Development (3rd Edition)*, Addison-Wesley Professional, Upper Saddle River, NJ, 2010, ISBN-13: 978-0672329463, pg. 64.

[6] Simmonds, Chris, Mastering Embedded Linux Programming (Second Edition), Packt Publising, Birmingham, U.K., 2017, ISBN-13: 978-1787288850, pg. 420.

[7] Toyooka, H. "Evaluation of Real-time Property in Embedded Linux" LinuxCon, Japan, 2014, URL: https://events.linuxfoundation.org/sites/events/files/slides/toyooka_LCJ2014_v10.pdf, slides 32 – 34. [retrieved May 24, 2018]

[8] "Open Source Automation Development Lab," URL: https://www.osadl.org/QA-Farm-Realtime.qa-farm-about.0.html. [retrieved March 16, 2016]

[9] "CFS Scheduler," URL: https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt. [retrieved May 24th, 2018]

[10] Corbet, J., "(Nearly) full tickless operation in 3.10," , Eklektix, Inc., URL:http://lwn.net/Articles/549580/, May 8, 2013. [retrieved May 24, 2018]

[11] Corbet, J., "Software interrupts and realtime," Eklektix, Inc., URL: https://lwn.net/Articles/520076/, October 17, 2012. [retrieved May 24, 2018]