# **Real-Time Linux Kernel Scheduler**

### **Ankita Garg**

**Abstract**

The -rt patchset, or just -rt, adds real-time scheduling capabilities to the Linux kernel.

Many market sectors, such as financial trading, defense, industry automation and gaming, long have had a need for low latencies and deterministic response time. Traditionally, custom-built hardware and software were used to meet these real-time requirements. However, for some soft real-time requirements, where predictability of response times is advantageous and not mandatory, this is an expensive solution. With the advent of the PREEMPT\_RT patchset, referred to as -rt henceforth, led by Ingo Molnar, Linux has made great progress in the world of real-time operating systems for “enterprise real-time” applications. A number of modifications were made to the general-purpose Linux kernel to make Linux a viable choice for real time, such as the scheduler, interrupt handling, locking mechanism and so on.

A real-time system is one that provides guaranteed system response times for events and transactions—that is, every operation is expected to be completed within a certain rigid time period. A system is classified as hard real-time if missed deadlines cause system failure and soft real-time if the system can tolerate some missed time constraints.

## **Design Goal**

Real-time systems require that tasks be executed in a strict priority order. This necessitates that only the N highest-priority tasks be running at any given point in time, where N is the number of CPUs. A variation to this requirement could be strict priority-ordered scheduling in a given subset of CPUs or scheduling domains (explained later in this article). In both cases, when a task is runnable, the scheduler must ensure that it be put on a runqueue on which it can be run immediately—that is, the real-time scheduler has to ensure system-wide strict real-time priority scheduling (SWSRPS). *Unlike non-real-time systems where the scheduler needs to look only at its runqueue of tasks to make scheduling decisions, a real-time scheduler makes global scheduling decisions, taking into account all the tasks in the system at any given point.* Real-time task balancing also has to be performed frequently. Task balancing can introduce cache thrashing and contention for global data (such as runqueue locks) and can degrade throughput and scalability. A real-time task scheduler would trade off throughput in favor of correctness, but at the same time, it must ensure minimal task ping-ponging.

The standard Linux kernel provides two real-time scheduling policies, SCHED\_FIFO and SCHED\_RR. The main real-time policy is SCHED\_FIFO. It implements a first-in, first-out scheduling algorithm. When a SCHED\_FIFO task starts running, it continues to run until it voluntarily yields the processor, blocks or is preempted by a higher-priority real-time task. It has no timeslices. All other tasks of lower priority will not be scheduled until it relinquishes the CPU. Two equal-priority SCHED\_FIFO tasks do not preempt each other. SCHED\_RR is similar to SCHED\_FIFO, except that such tasks are allotted timeslices based on their priority and run until they exhaust their timeslice. Non-real-time tasks use the SCHED\_NORMAL scheduling policy (older kernels had a policy named SCHED\_OTHER).

In the standard Linux kernel, real-time priorities range from zero to (MAX\_RT\_PRIO-1), inclusive. By default, MAX\_RT\_PRIO is 100. Non-real-time tasks have priorities in the range of MAX\_RT\_PRIO to (MAX\_RT\_PRIO + 40). This constitutes the nice values of SCHED\_NORMAL tasks. By default, the –20 to 19 nice range maps directly onto the priority range of 100 to 139.

This article assumes that readers are aware of the basics of a task scheduler. See Resources for more information about the Linux Completely Fair Scheduler (CFS).

## **Overview of the -rt Patchset Scheduling Algorithm**

The real-time scheduler of the -rt patchset adopts an active push-pull strategy developed by Steven Rostedt and Gregory Haskins for balancing tasks across CPUs. The scheduler has to address several scenarios:

1. Where to place a task optimally on wakeup (that is, pre-balance).
2. What to do with a lower-priority task when it wakes up but is on a runqueue running a task of higher priority.
3. What to do with a low-priority task when a higher-priority task on the same runqueue wakes up and preempts it.
4. What to do when a task lowers its priority and thereby causes a previously lower-priority task to have the higher priority.

A push operation is initiated in cases 2 and 3 above. The push algorithm considers all the runqueues within its root domain (discussed later) to find the one that is of a lower priority than the task being pushed.

A pull operation is performed for case 4 above. Whenever a runqueue is about to schedule a task that is lower in priority than the previous one, it checks to see whether it can pull tasks of higher priority from other runqueues.

*Real-time tasks are affected only by the push and pull operations.* The CFS load-balancing algorithm does not handle real-time tasks at all, as it has been observed that the load balancing pulls real-time tasks away from runqueues to which they were correctly assigned, inducing unnecessary latencies.

## **Important -rt Patchset Scheduler Data Structures and Concepts**

The main per-CPU runqueue data structure struct rq, holds a structure struct rt\_rq that encapsulates information about the real-time tasks placed on the per-CPU runqueue, as shown in Listing 1.

**Listing 1. struct rt\_rq*10165l1.qrk***

struct rt\_rq { struct rt\_prio\_array active; ... unsigned long rt\_nr\_running; unsigned long rt\_nr\_migratory; unsigned long rt\_nr\_uninterruptible; int highest\_prio; int overloaded; };

Real-time tasks have a priority in the range of 0–99. These tasks are organized on a runqueue in a priority-indexed array active, of type struct rt\_prio\_array. An rt\_prio\_array consists of an array of subqueues. There is one subqueue per priority level. Each subqueue contains the runnable real-time tasks at the corresponding priority level. There is also a bitmask corresponding to the array that is used to determine effectively the highest-priority task on the runqueue.

rt\_nr\_running and rt\_nr\_uninterruptible are counts of the number of runnable real-time tasks and the number of tasks in the TASK\_UNINTERRUPTIBLE state, respectively.

rt\_nr\_migratory indicates the number of tasks on the runqueue that can be migrated to other runqueues. Some real-time tasks are bound to a specific CPU, such as the kernel thread softirq-timer. It is quite possible that a number of such affined threads wake up on a CPU at the same time. For example, the softirq-timer thread might cause the softirq-sched kernel thread to become active, resulting in two real-time tasks becoming runnable. This causes the runqueue to be overloaded with real-time tasks. When overloaded, the real-time scheduler normally will cause other CPUs to pull tasks. These tasks, however, cannot be pulled by another CPU because of their CPU affinity. The other CPUs cannot determine this without the overhead of locking several data structures. This can be avoided by maintaining a count of the number of tasks on the runqueue that can be migrated to other CPUs. When a task is added to a runqueue, the hamming weight of the task->cpus\_allowed mask is looked at (cached in task->rt.nr\_cpus\_allowed). If the value is greater than one, the rt\_nr\_migratory field of the runqueue is incremented by one. The overloaded field is set when a runqueue contains more than one real-time task and at least one of them can be migrated to another runqueue. It is updated whenever a real-time task is enqueued on a runqueue.

The highest\_prio field indicates the priority of the highest-priority task queued on the runqueue. This may or may not be the priority of the task currently executing on the runqueue (the highest-priority task could have just been enqueued on the runqueue and is pending a schedule). This variable is updated whenever a task is enqueued on a runqueue. The value of the highest\_prio is used when scanning every runqueue to find the lowest-priority runqueue for pushing a task. If the highest\_prio of the target runqueue is smaller than the task to be pushed, the task is pushed to that runqueue.

Figure 1 shows the values of the above data structures in an example scenario.



Figure 1. Example Runqueues

## **Root Domain**

As mentioned before, because the real-time scheduler requires several global, or system-wide, resources for making scheduling decisions, scalability bottlenecks appear as the number of CPUs increase (due to increased contention for the locks protecting these resources). For instance, in order to find out if the system is overloaded with real-time tasks—that is, has more runnable real-time tasks than the number of CPUs—it needs to look at the state of all the runqueues. In earlier versions, a global rt\_overload variable was used to track the status of all the runqueues on a system. This variable would then be used by the scheduler on every call to the schedule() routine, thus leading to huge contention.

*Recently, several enhancements were made to the scheduler to reduce the contention for such variables to improve scalability.* The concept of root domains was introduced by Gregory Haskins for this purpose. cpusets provide a mechanism to partition CPUs into a subset that is used by a process or a group of processes. Several cpusets could overlap. A cpuset is called exclusive if no other cpuset contains overlapping CPUs. Each exclusive cpuset defines an isolated domain (called a root domain) of CPUs partitioned from other cpusets or CPUs. Information pertaining to every root domain is stored in struct root\_domain, as shown in Listing 2. These root domains are used to narrow the scope of the global variables to per-domain variables. Whenever an exclusive cpuset is created, a new root domain object is created with information from the member CPUs. By default, a single high-level root domain is created with all CPUs as members. With the rescoping of the rt\_overload variable, the cache-line bouncing would affect only the members of a particular domain and not the entire system. All real-time scheduling decisions are made only within the scope of a root domain. *Garrick, shrink listing 2.*

**Listing 2. struct root\_domain*10165l2.qrk***

struct root\_domain { atomic\_t refcount; /\* reference count for the domain \*/ cpumask\_t span; /\* span of member cpus of the domain\*/ cpumask\_t online; /\* number of online cpus in the domain\*/ cpumask\_t rto\_mask; /\* mask of overloaded cpus in the domain\*/ atomic\_t rto\_count; /\* number of overloaded cpus \*/ .... };

## **CPU Priority Management**

CPU Priority Management is an infrastructure also introduced by Gregory Haskins to make task migration decisions efficient. This code tracks the priority of every CPU in the system. Every CPU can be in any one of the following states: INVALID, IDLE, NORMAL, RT1, ... RT99.

CPUs in the INVALID state are not eligible for task routing. The system maintains this state with a two-dimensional bitmap: one dimension for the different priority levels and the second for the CPUs in that priority level (priority of a CPU is equivalent to the rq->rt.highest\_prio). This is implemented using three arrays, as shown in Listing 3. *Garrick, shrink listing 3.*

**Listing 3. struct cpupri*10165l3.qrk***

struct cpupri { struct cpupri\_vec pri\_to\_cpu[CPUPRI\_NR\_PRIORITIES]; long pri\_active[CPUPRI\_NR\_PRI\_WORDS]; int cpu\_to\_pri[NR\_CPUS]; };

The pri\_active bitmap tracks those priority levels that contain one or more CPUs. For example, if there is a CPU at priority 49, pri\_active[49+2]=1 (real-time task priorities are mapped to 2–102 internally in order to account for priorities INVALID and IDLE), finding the first set bit of this array would yield the lowest priority that any of the CPUs in a given cpuset is in.

The field cpu\_to\_pri indicates the priority of a CPU.

The field pri\_to\_cpu yields information about all the CPUs of a cpuset that are in a particular priority level. This is encapsulated in struct cpupri\_vec, as shown in Listing 4.

Like rt\_overload, cpupri also is scoped at the root domain level. Every exclusive cpuset that comprises a root domain consists of a cpupri data value. *Garrick, shrink listing 4.*

**Listing 4. struct cpupri\_vec*10165l4.qrk***

struct cpupri\_vec { raw\_spinlock\_t lock; int count; /\* number of cpus at a priority level \*/ cpumask\_t mask; /\* mask of cpus at a priority level \*/ };

The CPU Priority Management infrastructure is used to find a CPU to which to push a task, as shown in Listing 5. It should be noted that no locks are taken when the search is performed. *Garrick, shrink listing 5.*

**Listing 5. Finding a CPU to Which to Push a Task*10165l5.qrk***

int cpupri\_find(struct cpupri \*cp, struct task\_struct \*p, cpumask\_t \*lowest\_mask) { ... for\_each\_cpupri\_active(cp->pri\_active, idx) { struct cpupri\_vec \*vec = &cp->pri\_to\_cpu[idx]; cpumask\_t mask; if (idx >= task\_pri) break; cpus\_and(mask, p->cpus\_allowed, vec->mask); if (cpus\_empty(mask)) continue; \*lowest\_mask = mask; return 1; } return 0; }

If a priority level is non-empty and lower than the priority of the task being pushed, the lowest\_mask is set to the mask corresponding to the priority level selected. This mask is then used by the push algorithm to compute the best CPU to which to push the task, based on affinity, topology and cache characteristics.

## **Details of the Push Scheduling Algorithm**

As discussed before, in order to ensure SWSRPS, when a low-priority real-time task gets preempted by a higher one or when a task is woken up on a runqueue that already has a higher-priority task running on it, the scheduler needs to search for a suitable target runqueue for the task. This operation of searching a runqueue and transferring one of its tasks to another runqueue is called pushing a task.

The push\_rt\_task() algorithm looks at the highest-priority non-running runnable real-time task on the runqueue and considers all the runqueues to find a CPU where it can run. It searches for a runqueue that is of lower priority—that is, one where the currently running task can be preempted by the task that is being pushed. As explained previously, the CPU Priority Management infrastructure is used to find a mask of CPUs that have the lowest-priority runqueues. It is important to select only the best CPU from among all the candidates. The algorithm gives the highest priority to the CPU on which the task last executed, as it is likely to be cache-hot in that location. If that is not possible, the sched\_domain map is considered to find a CPU that is logically closest to last\_cpu. If this too fails, a CPU is selected at random from the mask.

The push operation is performed until a real-time task fails to be migrated or there are no more tasks to be pushed. Because the algorithm always selects the highest non-running task for pushing, the assumption is that, if it cannot migrate it, then most likely the lower real-time tasks cannot be migrated either and the search is aborted. No lock is taken when scanning for the lowest-priority runqueue. When the target runqueue is found, only the lock of that runqueue is taken, after which a check is made to verify whether it is still a candidate to which to push the task (as the target runqueue might have been modified by a parallel scheduling operation on another CPU). If not, the search is repeated for a maximum of three tries, after which it is aborted.

## **Details of the Pull Scheduling Algorithm**

The pull\_rt\_task() algorithm looks at all the overloaded runqueues in a root domain and checks whether they have a real-time task that can run on the target runqueue (that is, the target CPU is in the task->cpus\_allowed\_mask) and is of a priority higher than the task the target runqueue is about to schedule. If so, the task is queued on the target runqueue. This search aborts only after scanning all the overloaded runqueues in the root domain. Thus, the pull operation may pull more than one task to the target runqueue. If the algorithm only selects a candidate task to be pulled in the first pass and then performs the actual pull in the second pass, there is a possibility that the selected highest-priority task is no longer a candidate (due to another parallel scheduling operation on another CPU). To avoid this race between finding the highest-priority runqueue and having that still be the highest-priority task on the runqueue when the actual pull is performed, the pull operation continues to pull tasks. In the worst case, this might lead to a number of tasks being pulled to the target runqueue, which would later get pushed off to other CPUs, leading to task bouncing. Task bouncing is known to be a rare occurrence.

## **Scheduling Example**

Consider the scenario shown in Figure 2. Task T2 is being preempted by task T3 being woken on runqueue 0. Similarly, task T7 is voluntarily yielding CPU 3 to task T6 on runqueue 3. We first consider the scheduling action on CPU 0 followed by CPU 3. Also, assume all the CPUs are in the same root domain. The pri\_active bitmap for this system of CPUs will look like Figure 3. The numbers in the brackets indicate the actual priority that is offset by two (as explained earlier).



Figure 2. Runqueues Showing Currently Running Tasks and the Next Tasks to Be Run Just before the Push Operation



Figure 3. Per-sched Domain cpupri.pri\_active Array before the Push Operation

On CPU 0, the post-schedule algorithm would find the runqueue under real-time overload. It then would initiate a push operation. The first set bit of pri\_active yields runqueue of CPU 1 as the lowest-priority runqueue suitable for task T2 to be pushed to (assuming all the tasks being considered are not affined to a subset of CPUs). Once T2 is pushed over, the push algorithm then would try to push T1, because after pushing T2, the runqueue still would be under RT overload. The pri\_active after the first operation would be as shown in Figure 4. Because the lowest-priority runqueue has a priority greater than the task to be pushed (T1 of priority 85), the push aborts.



Figure 4. Per-sched Domain cpupri.pri\_active Array after the Push Operation

Now, consider scheduling at CPU 3, where the current task of priority 92 is voluntarily giving up the CPU. The next task in the queue is T6. The pre-schedule routine would determine that the priority of the runqueue is being lowered, triggering the pull algorithm. Only runqueues 0 and 1 being under real-time overload would be considered by the pull routine. From runqueue 0, the next highest-priority task T1 is of priority greater than the task to be scheduled—T6, and because T1 < T3 and T6 < T3, T1 is pulled over to runqueue 3. The pull does not abort here, as runqueue 1 is still under overload, and there are chances of a higher-priority task being pulled over. The next highest task, T4 on runqueue 1, also can be pulled over, as its priority is higher than the highest priority on runqueue 3. The pull now aborts, as there are no more overloaded runqueues. The final status of all the runqueues is as shown in Figure 5, which is in accordance with scheduling requirements on real-time systems.



Figure 5. Runqueues after the Push and Pull Operations

Although strict priority scheduling has been achieved, runqueue 3 is in an overloaded state due to the pull operation. This scenario is very rare; however, the community is working on a solution.

A number of locking-related decisions have to be made by the scheduler. The state of the runqueues would vary from the above example, depending on when the scheduling operation is performed on the runqueues. The above example has been simplified for this explanation.

## **Summary**

The most important goal of a real-time kernel scheduler is to ensure SWSRPS. The scheduler in the CONFIG\_PREEMPT\_RT kernel uses push and pull algorithms to balance and correctly distribute real-time tasks across the system. Both the push and pull operations try to ensure that a real-time task gets an opportunity to run as soon as possible. Also, in order to reduce the performance and scalability impact that might result from increased contention of global variables, the scheduler uses the concept of root domains and CPU priority management. The scope of the global variables is reduced to a subset of CPUs as opposed to the entire system, resulting in significant reduction of cache penalties and performance improvement.

## **Legal Statement**

This work represents the views of the author and does not necessarily represent the view of IBM. Linux is a copyright of Linus Torvalds. Other company, product and service names may be trademarks or service marks of others.

**Resources*10165s1.qrk***

Index of /pub/linux/kernel/projects/rt (Ingo Molnar):<http://www.kernel.org/pub/linux/kernel/projects/rt>

[patch] Modular Scheduler Core and Completely Fair Scheduler [CFS] (Ingo Molnar):<http://lwn.net/Articles/230501>

Multiprocessing with the Completely Fair Scheduler, Introducing the CFS for Linux:<http://www.ibm.com/developerworks/linux/library/l-cfs/index.html>

RT Wiki:<http://rt.wiki.kernel.org>