INSIDE THE LINUX SCHEDULER

M. TIM JONES

The Linux kernel continues to evolve, incorporating new technologies and gaining in reliability, scalability, and performance. One of the most important features of the 2.6 kernel is a scheduler implemented by Ingo Molnar. This scheduler is dynamic, supports load-balancing, and operates in constant time – O(1). This article explores these attributes of the Linux 2.6 scheduler, and more.

1. What is a scheduler?

An operating system, in a general sense, mediates between applications and available resources. Some typical resources are memory and physical devices. But a CPU can also be considered a resource to which a scheduler can temporarily allocate a task (in quantities called slices of time). The scheduler makes it possible to execute multiple programs at the same time, thus sharing the CPU with users of varying needs.

An important goal of a scheduler is to allocate CPU time slices efficiently while providing a responsive user experience. The scheduler can also be faced with such conflicting goals as minimizing response times for critical real-time tasks while maximizing overall CPU utilization. Let's see how the Linux 2.6 scheduler accomplishes these goals, compared to earlier schedulers.

2. Problems with earlier Linux schedulers

Before the 2.6 kernel, the scheduler had a significant limitation when many tasks were active. This was due to the scheduler being implemented using an algorithm with O(n) complexity. In this type of scheduler, the time it takes to schedule a task is a function of the number of tasks in the system. In other words, the more tasks (n) are active, the longer it takes to schedule a task. At very high loads, the processor can be consumed with scheduling and devote little time to the tasks themselves. Thus, the algorithm lacked scalability.

The pre-2.6 scheduler also used a single runqueue for all processors in a symmetric multiprocessing system (SMP). This meant a task could be scheduled on any processor – which can be good for load balancing but bad for memory caches. For example, suppose a task executed on CPU-1, and its data was in that processor's cache. If the task got rescheduled to CPU-2, its data would need to be invalidated in CPU-1 and brought into CPU-2.

Date: 30 June, 2006.

The prior scheduler also used a single runqueue lock; so, in an SMP system, the act of choosing a task to execute locked out any other processors from manipulating the runqueues. The result was idle processors awaiting release of the runqueue lock and decreased efficiency.

Finally, preemption wasn't possible in the earlier scheduler; this meant that a lower priority task could execute while a higher priority task waited for it to complete.

3. Introducing the Linux 2.6 scheduler

The 2.6 scheduler was designed and implemented by Ingo Molnar. Ingo has been involved in Linux kernel development since 1995. His motivation in working on the new scheduler was to create a completely O(1) scheduler for wakeup, context-switch, and timer interrupt overhead. One of the issues that triggered the need for a new scheduler was the use of Java virtual machines (JVMs). The Java programming model uses many threads of execution, which results in lots of overhead for scheduling in an O(n) scheduler. An O(1) scheduler doesn't suffer under high loads, so JVMs execute efficiently.

The 2.6 scheduler resolves the primary three issues found in the earlier scheduler (O(n) and SMP scalability issues), as well as other problems. Now we'll explore the basic design of the 2.6 scheduler.

3.1. **Major scheduling structures:** Let's start with a review of the 2.6 scheduler structures. Each CPU has a runqueue made up of 140 priority lists that are serviced in FIFO order. Tasks that are scheduled to execute are added to the end of their respective runqueue's priority list. Each task has a time slice that determines how much time it's permitted to execute. The first 100 priority lists of the runqueue are reserved for real-time tasks, and the last 40 are used for user tasks (see Figure 1). You'll see later why this distinction is important.

In addition to the CPU's runqueue, which is called the active runqueue, there's also an expired runqueue. When a task on the active runqueue uses all of its time slice, it's moved to the expired runqueue. During the move, its time slice is recalculated (and so is its priority; more on this later). If no tasks exist on the active runqueue for a given priority, the pointers for the active and expired runqueues are swapped, thus making the expired priority list the active one.

The job of the scheduler is simple: choose the task on the highest priority list to execute. To make this process more efficient, a bitmap is used to define when tasks are on a given priority list. Therefore, on most architectures, a find-first-bit-set instruction is used to find the highest priority bit set in one of five 32-bit words (for the 140 priorities). The time it takes to find a task to execute depends not on the number of active tasks but instead on the number of priorities. This makes the 2.6 scheduler an O(1) process because the time to schedule is both fixed and deterministic regardless of the number of active tasks.

3.2. **Better support for SMP systems:** So, what is SMP? It's an architecture in which multiple CPUs are available to execute individual tasks simultaneously, and it

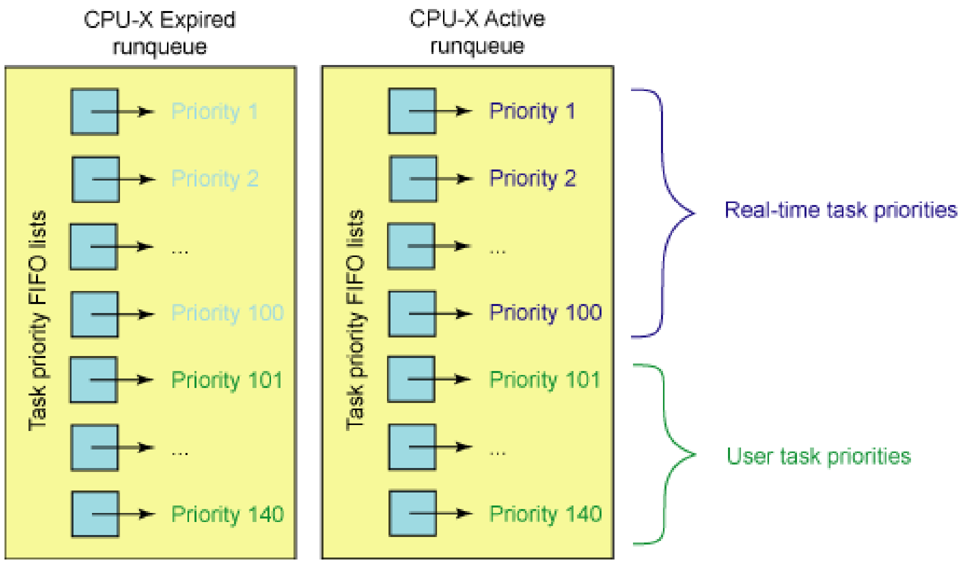

FIGURE 1. The Linux-2.6 scheduler runqueue structure

differs from traditional asymmetrical processing in which a single CPU executes all tasks. The SMP architecture can be beneficial for multithreaded applications.

Even though the prior scheduler worked in SMP systems, its big-lock architecture meant that while a CPU was choosing a task to dispatch, the runqueue was locked by the CPU, and others had to wait. The 2.6 scheduler doesn't use a single lock for scheduling; instead, it has a lock on each runqueue. This allows all CPUs to schedule tasks without contention from other CPUs.

In addition, with a runqueue per processor, a task generally shares affinity with a CPU and can better utilize the CPU's hot cache.

3.3. **Task preemption:** Another advantage of the Linux 2.6 scheduler is that it allows preemption. This means a lower-priority task won't execute while a higher-priority task is ready to run. The scheduler preempts the lower-priority process, places the process back on its priority list, and then reschedules.

4. But wait, there's more!

As if the O(1) nature of the 2.6 scheduler and preemption weren't enough, the scheduler also offers dynamic task prioritization and SMP load balancing. Let's discuss what these are and the benefits they provide.

4.1. **Dynamic task prioritization:** To prevent tasks from hogging the CPU and thus starving other tasks that need CPU access, the Linux 2.6 scheduler can dynamically alter a task's priority. It does so by penalizing tasks that are bound to a CPU and rewarding tasks that are I/O bound. I/O-bound tasks commonly use the CPU to set up an I/O and then sleep awaiting the completion of the I/O. This type of behavior gives other tasks access to the CPU.

Because I/O-bound tasks are viewed as altruistic for CPU access, their priority is decreased (a reward) by a maximum of five priority levels. CPU-bound tasks are punished by having their priority increased by up to five levels.

Tasks are determined to be I/O-bound or CPU-bound based on an interactivity heuristic. A task's interactiveness metric is calculated based on how much time the task executes compared to how much time it sleeps. Note that because I/O tasks schedule I/O and then wait, an I/O-bound task spends more time sleeping and waiting for I/O completion. This increases its interactive metric.

It's important to note that priority adjustments are performed only on user tasks, not on real-time tasks.

4.2. **SMP load balancing:** When tasks are created in an SMP system, they're placed on a given CPU's runqueue. In the general case, you can't know when a task will be short-lived or when it will run for a long time. Therefore, the initial allocation of tasks to CPUs is likely suboptimal.

To maintain a balanced workload across CPUs, work can be redistributed, taking work from an overloaded CPU and giving it to an underloaded one. The Linux 2.6 scheduler provides this functionality by using load balancing. Every 200ms, a processor checks to see whether the CPU loads are unbalanced; if they are, the processor performs a cross-CPU balancing of tasks.

A negative aspect of this process is that the new CPU's cache is cold for a migrated task (needing to pull its data into the cache).

Remember that a CPU's cache is local (on-chip) memory that offers fast access over the system's memory. If a task is executed on a CPU, and data associated with the task is brought into the CPU's local cache, it's considered hot. If no data for a task is located in the CPU's local cache, then for this task, the cache is considered cold.

It's unfortunate, but keeping the CPUs busy makes up for the problem of a CPU cache being cold for a migrated task.

5. Dig for more gold

The source for the 2.6 scheduler is well encapsulated in the file `/usr/src/linux/kernel/sched.c`. Some of the more interesting functions that can be found in this file are listed in Table 1.

The runqueue structure can also be found in the `/usr/src/linux/kernel/sched.c` file. The 2.6 scheduler can optionally provide statistics (via `CONFIG_SCHEDSTATS`). The statistics are available from the `/proc` filesystem at `/proc/schedstat` and present a variety of data for each CPU in the system, including load-balancing and process-migration statistics.

Table 1. Linux 2.6 scheduler functions

Function	Function description
`schedule`	The main scheduler function. Schedules the highest priority task for execution.
`load_balance`	Checks the CPU to see whether an imbalance exists, and attempts to move tasks if not balanced.
`effective_prio`	Returns the effective priority of a task (based on the static priority, but includes any rewards or penalties).
`recalc_task_prio`	Determines a task's bonus or penalty based on its idle time.
`source_load`	Conservatively calculates the load of the source CPU (from which a task could be migrated).
`target_load`	Liberally calculates the load of a target CPU (where a task has the potential to be migrated).
`migration_thread`	High-priority system thread that migrates tasks between CPUs.

6. Looking ahead

The Linux 2.6 scheduler has come a long way from the earlier Linux schedulers. It has made great strides in maximizing CPU utilization while providing a responsive experience for the user. Preemption and better support for multiprocessor architectures move it closer to an operating system that's useful both on the desktop and on the real-time system. The Linux 2.8 kernel is far away, but from looking at the changes in 2.6, I see good things ahead.