Scheduling is one of the most complex—and interesting—aspects of the Linux kernel. Developing schedulers that provide suitable behavior for single-core machines to quad-core servers can be difficult. Luckily, the Linux Scheduler Simulator (LinSched) hosts your Linux scheduler in user space (for scheduler prototyping) while modeling arbitrary hardware targets to validate your scheduler across a spectrum of topologies. Learn about LinSched and how to experiment with your scheduler for Linux.

Scheduling tasks in Linux is a complex task. Linux operates in a variety of use models (such as enterprise server, client desktop, and even embedded devices) over a huge spectrum of processor topologies (single core, multi-core, multi-core/multi-thread, and so on). It's amazing that the small number of scheduling policies in Linux work at all.

To make matters worse, measuring the efficacy of scheduling policies in Linux is difficult, because the scheduler resides deep in the kernel. Adding introspection such as tracing can actually change the behavior of the scheduler and hide defects or inefficiencies. Even more, set up scheduling scenarios to validate a given workload across the variety of processor topologies, and you're ready to pull your hair out.

Luckily, projects like LinSched (the User-space Linux Scheduler Simulator) can help solve this problem.

Introducing LinSched

LinSched is a Linux scheduler simulator that resides in user space. It isolates the Linux scheduler subsystem and builds enough of the kernel environment around it that it can be executed within user space. It also builds a set of application programming interfaces (APIs) around it to provide the necessary stimuli to validate scheduling workloads and collect enough relevant data to understand its behavior (I explore its overall architecture shortly).

Pulling the scheduler into user space makes it easy to execute the scheduler (because a crash of the scheduler won't bring the entire machine down). It also makes it simple to collect data about
the scheduler, because it can easily and efficiently gather data and store it within the local file system.

Alternatives include the use of virtualization, but LinSched offers the advantage of simplicity and efficiency (because booting another kernel is not necessary). As a user-space process, it's also easy to attach a debugger such as the GNU Debugger (GDB) to provide even greater insight into scheduler operation.

The origins of LinSched
LinSched was originally developed at the University of North Carolina (with grants from IBM and Intel) but was recently revived by Google to validate scheduling policies. This is an interesting move and may represent a trend in the future. Although Linux supports a number of scheduling classes and does a reasonable job of scheduling across a number of workloads, it may not be ideal for a specific workload on a given hardware topology. Because Google invests in a large cluster of identical machines that perform a specific task (for example, MapReduce), it would make sense that the schedule would be tuned for that task and environment. In its use of LinSched, Google has also contributed enhancements back to Linux for the Completely Fair Scheduler (CFS), including bandwidth provisioning and load balancing when a large weight differential exists between tasks.

LinSched architecture
LinSched is more than just a scheduler simulator that executes in user space: It's a migration of the Linux scheduler into user space with a thin emulation of the necessary components so that the scheduler is able to execute outside of the kernel. The LinSched source is actually a Linux distribution (currently 2.6.35) with a new subdirectory called /linsched for the wrappers and simulation engine. Because LinSched uses the Linux scheduler subsystem within Linux for its simulation, it's much simpler to make changes, and then integrate them back into the kernel.

The overall architecture of LinSched is shown in Figure 1. At the bottom is the host operating system. LinSched is a user-space application built from a number of components (including portions of the Linux kernel itself).

Figure 1. Architecture of the LinSched Scheduler Simulator
The environment module provides the abstraction of the Linux kernel (with flags to support compilation in the simulation mode). On the environment module is the simulation engine, which extends an API for use in configuring the environment and performing the simulation. The simulation engine API provides an initialization function that also defines the processor topology as well as functions for task creation, callbacks (for example, when a task is scheduled), and a run function to perform the simulation for some number of ticks (which in turn calls the internal Linux schedule() function to make scheduling decisions).

Above the simulation engine is the script environment (the code for the actual simulation). It can be a single script, but as the upcoming example shows, the simplest use is to amend the existing script environment to add test scenarios.

The basic architecture of LinSched is quite simple. Let's first explore how to install LinSched, and then discuss some of the APIs extended and how to use them in a simulation.

### Installing LinSched

LinSched was recently updated to support the 2.6.35 kernel. You can grab the current LinSched package in a couple of ways—either through git or through the project’s Web page (see Related topics). To grab the distribution with git, clone the repository as:

```
$ git clone git://google3-2.osuosl.org/linsched/2.6.35.git
```

In either case, you'll get a subdirectory called `2.6.35/` that contains the distribution. In the LinSched subdirectory, you can execute the following command line for a quick test:

```
$ make run_all_tests
```

This command line performs a number of scheduler tests over a variety of different processor topologies and emits the results of the tests.

### LinSched APIs

To perform a LinSched simulation, you create a script and perform a set of initializations to set up the desired scenario. This initialization is performed through a set of calls to the LinSched API. This section explores some of the important API functions (and later demonstrates their use). Note that the entire API is not disclosed below, but source files in `./linsched` provide a more detailed list.

Before running a simulation, you must initialize the simulator. You do so through a call to `linsched_init`. This function accepts as its argument the simulated processor topology and internally configures the simulator environment and starts the user-space Linux kernel.

```c
void linsched_init( struct linsched_topology *topo );
```

The topology structure, defined in `linux_linsched.h`, defines the number of processors and how they relate to one another (mapping to a physical package and node distance map). You can specify a single-processor package with `TOPO_UNIPROCESSOR` or a four-socket quad-core processor topology with `TOPO_QUAD_CPU QUAD_SOCKET`. 

---

Linux Scheduler simulation
Tasks within LinSched are simulated entities, although LinSched does give you control to create a large number of tasking scenarios. One of the most useful functions creates a number of tasks that follow a busy/sleep configuration. The caller provides the number of tasks to create (count), the mask of processors for which these tasks can execute (mask), and then the sleep/busy tick counts (identified as sleep and run).

```c
void create_tasks( unsigned int count, unsigned long mask, int sleep, int run );
```

To create a more specific tasking scenario, a number of additional task-creation API functions are available (see Listing 1). These functions permit the creation of normal, batch, or real-time tasks (first in, first out [FIFO] or round-robin), with a specific priority or nice level. Creating a normal task sets the scheduling policy to `SCHED_NORMAL`, where `batch`, `RTfifo`, and `RTrr` are set to `SCHED_BATCH`, `SCHED_FIFO`, and `SCHED_RR`, respectively. The user can provide a task structure to control how the particular task behaves or use the `linsched_create_sleep_run` API function to create a generic sleep/busy task. You can find each of these tasks in `linux_linsched.c`.

**Listing 1. LinSched task-creation API functions**

```c
int linsched_create_normal_task( struct task_data* td, int niceval );
void linsched_create_batch_task( struct task_data* td, int niceval );
void linsched_create_RTfifo_task( struct task_data* td, int prio );
void linsched_create_RTrr_task( struct task_data* td, int prio );
struct task_data *linsched_create_sleep_run( int sleep, int busy );
```

Through the API, it's also possible to create task groups, attach tasks to those groups, and add processor shares to the groups for task scheduling. Listing 2 provides a list of those functions (which can be found in `linux_linsched.c`).

**Listing 2. LinSched task group and group shares API functions**

```c
int linsched_create_task_group( int parent_group_id );
void linsched_set_task_group_shares( int group_id, unsigned long shares );
int linsched_add_task_to_group( int task_id, int group_id );
```

To begin a simulation, you make a call to `linsched_run_sim`. This call accepts as its only argument the number of ticks to run. At each tick, the potential for a scheduling decision is made. When the simulation is complete, the function returns (this function can be found in `hrtimer.c`).

```c
void linsched_run_sim( int sim_ticks );
```

Finally, you can view the results of a simulation through the calls shown in Listing 3. These calls emit the individual task statistics and group statistics, respectively.

**Listing 3. LinSched functions for emitting simulation statistics**

```c
void linsched_print_task_stats( void );
void linsched_print_group_stats( void );
```

For more detailed analysis, you can also grab a task based on its processor ID using the following function:
struct task_struct *linsched_get_task( int task_id );

With the task structure, you can find the total execution time for the task (with `task_exec_time(task)`), time spent not running (`task->sched_info.run_delay`), and the number of times the scheduler invoked the task (`task->sched_info.pcount`).

From this short list, you can see that LinSched provides a useful API for constructing scheduling scenarios. It's also possible to perform these functions iteratively, such that once a scenario has started and finished, new tasks can be added and the simulation continued through another call to `linsched_run_sim`. This functionality permits the construction of dynamic scheduling scenarios that are also repeatable.

**Experimenting with LinSched**

Now that you've explored the LinSched simulation engine API, let's have a look at it in action. For this example (shown in Listing 4), I've used the `basic_test` framework of LinSched to add a new scenario. In this framework, the second argument passed is the topology to use. There exists a `topo_db` to provide the various topologies supported (which the framework runs through all available options). With the topology defined, you call `linsched_init` to initialize the environment for this processor topology. You then create 11 tasks in three categories. The first five tasks are created using the simple `create_tasks` function, which emulates processor-bound tasks (busy 90% of the time, sleep 10% of the time). You then create five input/output (I/O)-bound tasks (busy 10% of the time, sleep 90% of the time). Finally, you create a real-time task (with `linsched_create_RTrTrr_task`) that is busy 100% of the time and has a priority of 90. You then start the simulator with a call to `linsched_run_sim` and emit results with `linsched_print_task_stats`.

**Note:** This example uses the standard CFS (as invoked through `schedule()` via `linsched_run_sim`).

**Listing 4. Sample LinSched script**

```c
void new_test(int argc, char **argv)
{
  int count, mask;
  struct linsched_topology topo;
  int type = parse_topology(argv[2]);
  topo = topo_db[type];
  // Allow all tasks to use any processor.
  mask = (1 << count) - 1;
  // Initialize the topology as provided by the script interpreter
  linsched_init(&topo);
  // Create five processor-bound tasks (sleep 10, busy 90)
  create_tasks(5, mask, 10, 90);
  // Create five I/O-bound tasks (sleep 90, busy 10)
  create_tasks(5, mask, 90, 10);
  // Create a busy real-time round-robin task with a priority of 90
  linsched_create_RTrTrr_task( linsched_create_sleep_run(0,100), 90 );
  // Run the simulation
```
Running this scenario results in a variety of output for each task in a given run. For example, in **Listing 5**, you can see the output for a run of the scenario (from **Listing 4**) on a uniprocessor topology. This output shows the list of tasks (numbered by task ID), their total execution time (in ticks), the amount of time they waited to execute, and finally the number of times they were invoked. Note that these tasks never exit, although it's possible to end a task through the API.

Per this scenario, the first five tasks are busy tasks and sleep 10% of the time. The second five tasks sleep most of the time and are busy 10% of the time. The last task is a real-time task and is busy 100% of the time. As shown, the real-time task receives the lion's share of the single processor and is invoked only 61 times. Compare this to the busy and non-busy tasks that were invoked roughly three times as often for significantly less of the processor. Also note that the scheduler is fair across the busy and non-busy tasks, giving them roughly the same amount of processor access.

**Listing 5. Scheduling a test on a uniprocessor topology**

```plaintext
Task id =  1, exec_time =   305000000, run_delay = 59659000000, pcount = 156
Task id =  2, exec_time =   302000000, run_delay = 58680000000, pcount = 154
Task id =  3, exec_time =   304000000, run_delay = 58708000000, pcount = 155
Task id =  4, exec_time =   304000000, run_delay = 58708000000, pcount = 155
Task id =  5, exec_time =   304000000, run_delay = 58708000000, pcount = 155
Task id =  6, exec_time =   296000000, run_delay = 56118000000, pcount = 177
Task id =  7, exec_time =   296000000, run_delay = 56118000000, pcount = 177
Task id =  8, exec_time =   296000000, run_delay = 56118000000, pcount = 177
Task id =  9, exec_time =   296000000, run_delay = 56118000000, pcount = 177
Task id = 10, exec_time = 57001000000, run_delay =  2998000000, pcount = 61
```

Now, let's look at the same test from **Listing 4** but this time on a quad-socket quad-core topology (16 logical processors). Because each task can have its own logical processor, it receives considerably more processor time for the same given duration of the test. Although the busy and non-busy tasks are invoked an equal number of times, the non-busy tasks receive 10% of the execution time compared to the busy tasks. This difference is the result of the sleep/busy configuration (busy tasks execute for 90 ticks, while non-busy tasks execute for 10 ticks). Also interesting to note is that your real-time task is invoked once and executes for the duration of the test (because it never sleeps, it was never rescheduled on its processor). **Listing 6** shows the test.
Listing 6. Scheduling a test on a quad-core quad socket

<table>
<thead>
<tr>
<th>Task id</th>
<th>exec_time</th>
<th>run_delay</th>
<th>pcount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>540000000000</td>
<td>0</td>
<td>601</td>
</tr>
<tr>
<td>2</td>
<td>54000156250</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>3</td>
<td>54000281250</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>4</td>
<td>54000406250</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>5</td>
<td>54000031250</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>6</td>
<td>6000187500</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>7</td>
<td>6000312500</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>8</td>
<td>6000437500</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>9</td>
<td>600062500</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>10</td>
<td>6000218750</td>
<td>0</td>
<td>600</td>
</tr>
<tr>
<td>11</td>
<td>59999343750</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

You can also visualize the data emitted from LinSched. Let's look at one more example with a graphical visualization of the output. In the example shown in Listing 7, you create 40 tasks with nice values ranging from -20 to 19. Recall that task nice values change the priority of a task (where -20 is the highest priority and 20 is the lowest). For normal task scheduling, tasks will receive a portion of the processor proportional to their nice value.

Listing 7. Demonstrating the effect of nice values on normal task scheduling

```c
void new_test(int argc, char **argv)
{
    int count, mask, i;
    struct linsched_topology topo;
    int type = parse_topology(argv[2]);

    topo = topo_db[type];

    // Allow all tasks to use any processor.
    mask = (1 << count) - 1;

    // Initialize the topology as provided by the script interpreter
    linsched_init(&topo);

    for (i = 0 ; i < 40 ; i++) {
        linsched_create_normal_task( linsched_create_sleep_run(0,100), i-20 );
    }

    // Run the simulation
    linsched_run_sim(TEST_TICKS*10);

    // Emit the task statistics
    linsched_print_task_stats();

    return;
}
```

The result of this application is plotted in Figure 2 (for a uniprocessor topology). As shown, high-priority tasks receive the vast majority of the processor, while low-priority tasks receive considerably less in an exponential curve. The highest-priority task (nice of -20) received ~120 billion ticks, while the lowest-priority task (nice of 19) received 21 million ticks.
Figure 2. Plot of execution times for each task from Listing 7

![Plot of execution times for each task from Listing 7](image)

Other options for scheduling visualization

Although LinSched is unique in its capability for user-space scheduler simulation, other tools exist for visualizing kernel scheduling and other activities. The Linux Trace Toolkit (LTT) catalogs system events in fine detail to provide visibility of the internal operation of the Linux kernel. This project has advanced to LTT next generation (LTTng). LTTng provides a user-space tool for graphically (or in text) visualizing the operation of the kernel. You can also use this tool for user-space tracing in addition to combined tracing, which aggregates tracing from the kernel and user space. Check out the Related topics section for more information.

Going further

From this short article, you can see the value in simulating the Linux scheduler in user space. Work with LinSched has found that user-space simulation and actual kernel-based scheduling are fairly well correlated, so this approach is useful for predicting the behavior of a scheduler in production. Also interesting is the approach that LinSched uses. Rather than build a new framework through which scheduling could be accomplished in user space, LinSched uses the Linux kernel code itself (with wrappers for platform emulation). This means that once a scheduler is validated in user space, it's quite simple then to migrate this scheduler into the kernel for use.

Check out Related topics for more information on LinSched in addition to other tools and technologies useful in scheduler simulation and visualization.
Related topics

• One of the best ways to learn more about the details of LinSched is the short white paper written about the tool titled LinSched: The Linux Scheduler Simulator. This paper provides a great overview of LinSched in addition to a sample usage and workload simulation.

• Early in the development of the 2.6 Linux kernel, a new scheduler was introduced to take the place of the existing (and complex) scheduler. It was called the \(O(1)\) scheduler, because it operated in constant time, regardless of the number of tasks to schedule. You can learn more about this scheduler in Inside the Linux Scheduler (developerWorks, June 2006).

• Following the \(O(1)\) scheduler, the CFS was introduced and adopted by the Linux kernel. CFS is based on the idea that all processes should be given a fair amount of the available processors. You can see that fairness in the example provided in this article, where all processes are given access to the processor—even the lowest priority. You can learn more about the CFS in Inside the Linux 2.6 Completely Fair Scheduler (developerWorks, December 2009).

• The Linux Trace Toolkit next generation (LTTng) provides a user-space visualization facility for tracing kernel and user-space activities. This tool is useful for understanding system behavior and can help debug software issues as well as performance issues. Another useful site is the Distributed Multi-Core Tracing Research Project, which develops techniques and algorithms for multi-core system tracing with minimal overhead.

• To learn more about scheduling, including those algorithms implemented in the Linux kernel, check out the Scheduling (computing) page at Wikipedia. This piece covers FIFO and round-robin scheduling in addition to providing a short summary of the schedulers in each major operating system.

• In the developerWorks Linux zone, find hundreds of how-to articles and tutorials, as well as downloads, discussion forums, and a wealth of other resources for Linux developers and administrators.

• Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment.

• Follow developerWorks on Twitter, or subscribe to a feed of Linux tweets on developerWorks.