Plug-In Scheduler Policies for Linux

Author:	Scott Rhine
e-mail:	scott_rhine@hp.com
Company:	Hewlett-Packard Manageability Solutions Lab (MSL)
	Mail stop d1w45
	3000 Waterview Parkway
	Richardson, TX 75080
Phone:	972-497-4792
Fax:	972-497-3123

1. Introduction

Loadable kernel modules are commonly used in Linux for a variety of purposes [1,2]. Often, these modules are invoked as drivers for specific devices that may not be common to all Linux systems. Modules may be installed as needed while a system is running, without the need for a reboot. With the installation of a new patch [3] the kernel CPU scheduler may be treated like a software device, allowing the system administrator to change the scheduling policy to meet business objectives.

In section 2, instructions are given for system administrators who wish to take advantage of alternate scheduling policies. Examples of two policies are offered to clarify the concept and to give an idea of what can be done with the open framework provided. In section 3, the framework itself is analyzed to provide a better understanding of the mechanisms involved. Finally, section 4 offers helpful hints for kernel hackers wishing to implement a customized scheduler.

As a matter of formatting, C code, commands, file names, and function names will appear in the `Courier New font`.

2. System Administration

2.1 Why Plug-In Scheduler Policies?

The default Linux scheduler is a general-purpose algorithm that performs very well for a wide range of work loads and benchmarks [4]. However, there are times when another scheduler might be advantageous to manage a specific workload with different job selection preferences, CPU distributions, expected latencies, business priorities, or other performance considerations. For example, take a machine with a mixed load of ray-tracers, compiles, and animations attempting to display. If the other (non-animation) consumers take up too much CPU time, the animations will appear slow and bursty. An alternate policy could smooth the animation behavior and give the application more priority. This same technique could be used to provide essential resource minimums for video streaming, databases, on-line shopping transactions, nightly back-ups, or any other set of mission-critical applications or users. These mechanisms could also be employed to help limit consumption by greedy applications or denial of service attacks.

Essentially, any scheduler feature available on other operating systems may now be ported more easily to Linux: Processor/CPU Sets, batch queuing, gang thread scheduling, Job Objects, and Fair Share scheduling just to name a few. A few of these methods of making multi-CPU machines more manageable have already been implemented.

2.2 The Test Drive

The best way to demonstrate the usefulness and effectiveness of this feature is to see it first hand. How can a system administrator try this capability out? For Linux versions 2.2.14.50, 2.2.16, and several flavors of 2.4.0, a patch and utilities are available for download [3]. Follow the instructions for building and configuring a kernel with the new

feature enabled. The changes made are relatively minor and have been submitted to the main OS tree as part of the HP IA64 project. However, there is no guarantee of when they might become part of a standard Linux release.

One of the last steps in the patch procedure is to create the new `/dev/sched` device via the `mknod` command. Once the new kernel is running, any read of the device will display the scheduling policy currently being enforced.

```
# cat /dev/sched
scheduler   : linux
```

At the lowest level, modifications to a particular scheduling policy such as Processor Sets may be made via `ioctl` calls to this device; however, utility (`psrset`, `psetps`) and library (`libpset.so`) interfaces are provided for usability.

The best demonstration for load behavior is a visual one. Several interesting graphical programs are available free of charge as part of the OpenGL graphics library [5]. The bouncing ball program called `bounce` is a particular favorite. To build the package, type:

```
# make linux
```

Start up one or several of these animations, up to one per CPU. Get a feel for how fast they are displaying. The OpenGL programs often print rates of frames per second for those who prefer numeric results. Note that displays will tend to be more responsive on a system console than over the network.

Choose a good artificial load generator. Two or three consumers per CPU should be sufficient. The simplest CPU consumer is a tight while loop, easily programmable in a wide array of languages. The utilities bundle also has a convenient `sched_bench` program capable of spawning as many consumers as desired (assuming your system tunable parameters permit). Watch as the animation performance degrades.

Constant Time Round Robin Scheduler

Now the stage is set to experiment. While all this is running, install an alternate scheduler by typing:

```
# /sbin/insmod const_sched
```

Watch the performance improve slightly. Confirm that the new scheduler is really running by typing:

```
# cat /dev/sched
```

The Constant Time Round Robin scheduler is the simplest example available, reducing OS overhead. The default scheduler looks over every runnable process and decides which one goes next based on a variety of factors such as tick counters, memory region affinity, threads, scheduling class, and nice levels. Because of this, the more processes it finds on a system, the longer each decision takes. Scaling becomes a problem [6,7,8,9]. By contrast, the Constant Time scheduler treats all jobs the same. When it is time to pick the next runner, it takes the next job on the queue and runs it for a slice. When finished, the job gets placed on the end of the queue, and the next one in line gets its turn. Whether there are thousands of processes, or just one, the overhead remains constant.

This scheduler provides an interesting academic lower bound for complexity while demonstrating the effectiveness of the plug-in scheduler concept.

When satisfied, return to the default scheduler by removing the module:

```
# /sbin/rmmod const_sched
```

The behavior will then return to normal. It's that simple.

Processor Sets

For a more advanced experiment, install the processor sets module under the same load:

```
# /sbin/insmod pset
```

Nothing will change at first (except the label in `/dev/sched`), because everything on the system still belongs to the default processor set (0). This may be verified using `psetps` with any options normally used for the `ps` command.

Scheduling in each processor set is done independently, and each emulates the default Linux scheduler. Processor sets allow tasks to be associated with a collection of processors rather than running on any available CPU on the machine. Think of a processor set as an isolated virtual partition of the CPUs on a machine. A task or a CPU belongs to exactly one processor set at a time, but may be reassigned by root. The default processor set always exists and may not be destroyed. One CPU must always remain in it for the OS to function properly. Hence, this feature is of little value on a single CPU system.

Take the example of a 4-CPU machine with CPUs numbered 0 through 3. The exact numbering on a particular system can be obtained via `/proc/cpuinfo` or through `psrset`. To re-partition a multi-CPU machine, create a new pset containing CPU 3.

```
# psrset –c 3
```

This should generate processor set number 1. Verify with the following command:

```
# psrset (no args)
PSET ID        TYPE  CPUs   CPULIST
      0     FAILBUSY   3    0, 1, 2
      1     MOVEDFLT   1    3
```

Next, move the load generating spinners into the new processor set based on their process identifiers provided by the `ps` command.

```
# psrset –b 1 <pid list>
```

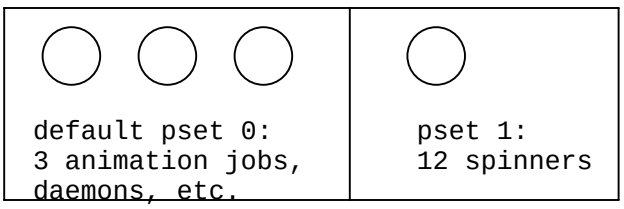

Figure 1: Segregation of CPUs and Jobs into Processor Sets

When this command has completely, watch the performance of the animation improve dramatically. As far as the animations are concerned, they are running on a dedicated 3-CPU machine, while the spinners fight each other over the remaining one.

Note the difference in CPU accumulation rates visible via `ps` or

```
# psetps –ef
```

Try moving the CPUs around using

```
# psetps –a PSET_id  CPU_id
```

Processor sets with no CPUs can also exist. Such a situation may be due to temporary resource needs elsewhere in the system, or a crude implementation of batch queues. When placed in a processor set that has no CPUs assigned, a task is effectively frozen, making no forward progress until such assignment is made. Local, compute-intensive jobs not requiring immediate turn-around can be placed in such empty processor sets. After hours, when interactive users go home, some of the interactive CPUs can be loaned to the batch job. If the batch jobs do not all complete by morning, they can be re-frozen and the CPUs re-dedicated to interactive support.

While there are still two processor sets, try to remove the module. Notice that the OS will not allow this because the module is busy. In order to cleanly shut down the processor set scheduler, first destroy all active sets, and then remove the module as normal.

```
# psrset –d all
# /sbin/rmmod pset
```

More details are available from the man pages provided with the utilities bundle [3]. More plug-in schedulers will be posted to the web site as they become available.

Lastly, now that the default scheduler has been reinstated, the demonstration programs may be cleaned up.

3. Flexible Scheduler Framework

What makes these plug-in scheduler modules work? The abstraction can be divided up into three components: `sched_policy`, `/dev/sched` support, and the loadable modules themselves. For those interested in details, read the raw patch file `diff` or the original design white paper [3]. All changes have been protected with `ifdefs` and will not become active unless alternate schedulers are enabled from a configuration menu.

3.1 Scheduling Policies

There are only a few key changes in `sched.c`. In a nutshell, the technique inserts code in front of about five major decision points in the scheduler. If the kernel has an alternate scheduler installed and that alternate scheduler has a preferred (non-NULL) method for this decision, it will call out to that method in the module; otherwise, it falls through to the default Linux behavior.

Pointers to the `sched_policy` structure (figure 2) appear in both per-process and global per-CPU structures.

```
CLASS: sched_policy
Variables:
      sp_runnable
      sp_private
Methods:
      sp_choose_task
      sp_preemptability
      sp_choose_cpu
      sp_handle_ticks
```

When processes enter or leave the global run queue, they increment or decrement the per-policy member count `sp_runnable`.

Many other per-policy functions and variables are possible, but core Linux doesn't need to see them. Anything else can be implemented inside the opaque, per-policy, private data area `sp_private`.

Figure 2

`sp_choose_task` is the most basic and necessary of the methods. It chooses the next process that will be run on this CPU. Note that the run queue lock must be held during the invocation and continue to be held upon return.

`sp_preemptability` decides for a specific CPU how likely another job is to preempt the currently running job according to the rules of this policy. It is generally called in a loop for every CPU.

`sp_choose_cpu` contains the same core logic as `sp_preemptability`, but decides the best CPU to preempt. This became necessary for Linux 2.4.0+ kernels because `sp_preemptability` is no longer invoked for idle CPUs. If, as in the case of processor sets, a task may not run on just any idle CPU, a higher level abstraction is required. At most one of the pair of methods will ever get invoked.

`sp_handle_ticks` may be used to accumulate information every clock tick for the purpose of debugging, monitoring, or control decisions. The Fair Share Schedulers use this method to track how much usage groups get with respect to each other [10,11,12].

3.2 /dev/sched Kernel Support

The new file `altpolicy.c` provides support for the `/dev/sched` interface.

```
CLASS: scheduler device
Variables:
      array policy_of_cpu
      scheduler name
      file ops structure
Methods:
      default_read_method
      register_scheduler
      unregister_scheduler
```

This file holds the globals we rely on for the implementation, especially the file operations for this device. If there are no alternate schedulers loaded, a read operation invokes the default method, and an `ioctl` has no effect. However, when a module `registers` itself, it may provide replacement methods for `read`, `ioctl`, as well as a new name and policy array. At most one alternate policy may be registered at a time.

Figure 3

The `unregister` operation resets all globals and process pointers to their default values.

3.3 Loadable Scheduler Modules

The source code for the plug-ins themselves resides in the directory `linux/drivers/sched`. The convention is to provide a single C file, a single header file, and a single line in the `Makefile` for every new alternate scheduler.

```
CLASS: loadable sched
       module
Variables:
      busy semaphore
      module name
Methods:
      init_module
      cleanup_module
      choose_task
      preemptability*
      choose_cpu*
      handle_ticks*
      ioctl_func*
*optional
```

A unique `module name`, including version number is necessary when registering or unregistering the module.

The `busy semaphore` prevents unloading of the module while structures inside the kernel may still reference addresses inside the module, or possibly while in the middle of a critical operation.

`init_module` is invoked upon install (`insmod`) of the module. It should first attempt to allocate space for internal data structures and `register` itself with the kernel. If either operation fails, the install should fail.

`cleanup_module` should likewise invoke `unregister`.

Figure 4

The local functions for `choose_task, preemptability, choose_cpu`, and `handle_ticks` are assigned to the corresponding `sched_policy` functions. Any optional function with no local preference may be left `NULL`, and Linux default behavior will be used.

`ioctl_function` is the interface between the user world and the private functions of this particular policy module. Any module expecting to control or modify the alternate scheduler behavior must currently receive instructions in this manner.

Constant Time Scheduler

The Constant Time scheduler requires no interaction, complicated CPU selection method, or statistic collection. Thus, most of its exported methods are `NULL`, and no `busy semaphore` is needed. Even the `sp_preemptibility` function has been simplified to the difference between two counter fields. The Constant Time scheduler, like the default Linux scheduler is unified or universal, meaning that every process and CPU on the machine point to a single `sched_policy` structure. There is also no private `sched_policy` data. At 174 commented lines, this module is the shortest useful example available.

Processor Set Scheduler

By contrast, the Processor Set scheduler can have potentially unlimited instances of the same template, one for each set. Each copy has it own method pointers (same) and its own private data region (distinct). This replication requires additional methods for managing instances, which are accessed via `ioctl`. When defining the `ioctl` constants in

the header file, a unique offset must be used for each potential scheduler on the machine. Why? Because a user application is not guaranteed from one system call to the next which scheduler is actually running. If each scheduler starts numbering at 1, applications attempting to control one scheduler flavor may get a successful return code, but totally incorrect results. Any number outside the reserved range for a scheduler module should return `EINVAL`.

```
Private variables:
      id
      spu_count
      non_empty_op
module methods:
create_group
destroy_group
move_task_to_group
move_cpu_to_group
set_group_attributes
get_group_attributes
query_group_info
```

Figure 5

The meaning of the private variables as well as the extra functions are described in detail in the white paper [3] and man pages.

Apart from the `attribute` based functions, which define individualized behavior in certain boundary conditions (like deleting a non-empty set), all of the methods mentioned would be a good base pattern for any multi-group scheduler. This includes the concept of a default group (0) where all processes and CPUs begin. The `busy lock` should remain on until only 0 remains.

All `sched_policy` methods are defined except for `sp_handle_ticks`. This is because the tick information could be gleaned indirectly from per-CPU statistics.

4. Experiences Building New Schedulers

4.1 What I Learned

This project was just the first iteration of feature development. I hope that it evolves into a standard for plug-in scheduler policies. Already, other developers in the workload management and IA64 spaces are attempting to leverage this work to save effort.

What did I learn personally from this project? Linux as a development tool is tremendous. The fix and retry cycle on a loadable module is incredibly fast, much better than having to reboot every time. Having booted the same code both built-in and as a module, it is much simpler to write for a machine that is already up. Special cases for the idle/boot process can eat you alive and good luck finding out why it crashed. By contrast, debugging a module is easy. There are only one or two critical routines to go wrong. Most errors are non-fatal. If it core dumped, there was generally an obvious bug in my initialization of the private module data structures. A silent hang was inevitably due to a missing unlock, membership count being off, or fumbling the `SCHED_YIELD` flag.

The `sched_bench` benchmark developed for this project may not be the most sophisticated, but it was great to show me the performance impact of changing one line of code. I was able to use it to tune the prototypes to speeds indistinguishable from or better than the default (see Figure 6).

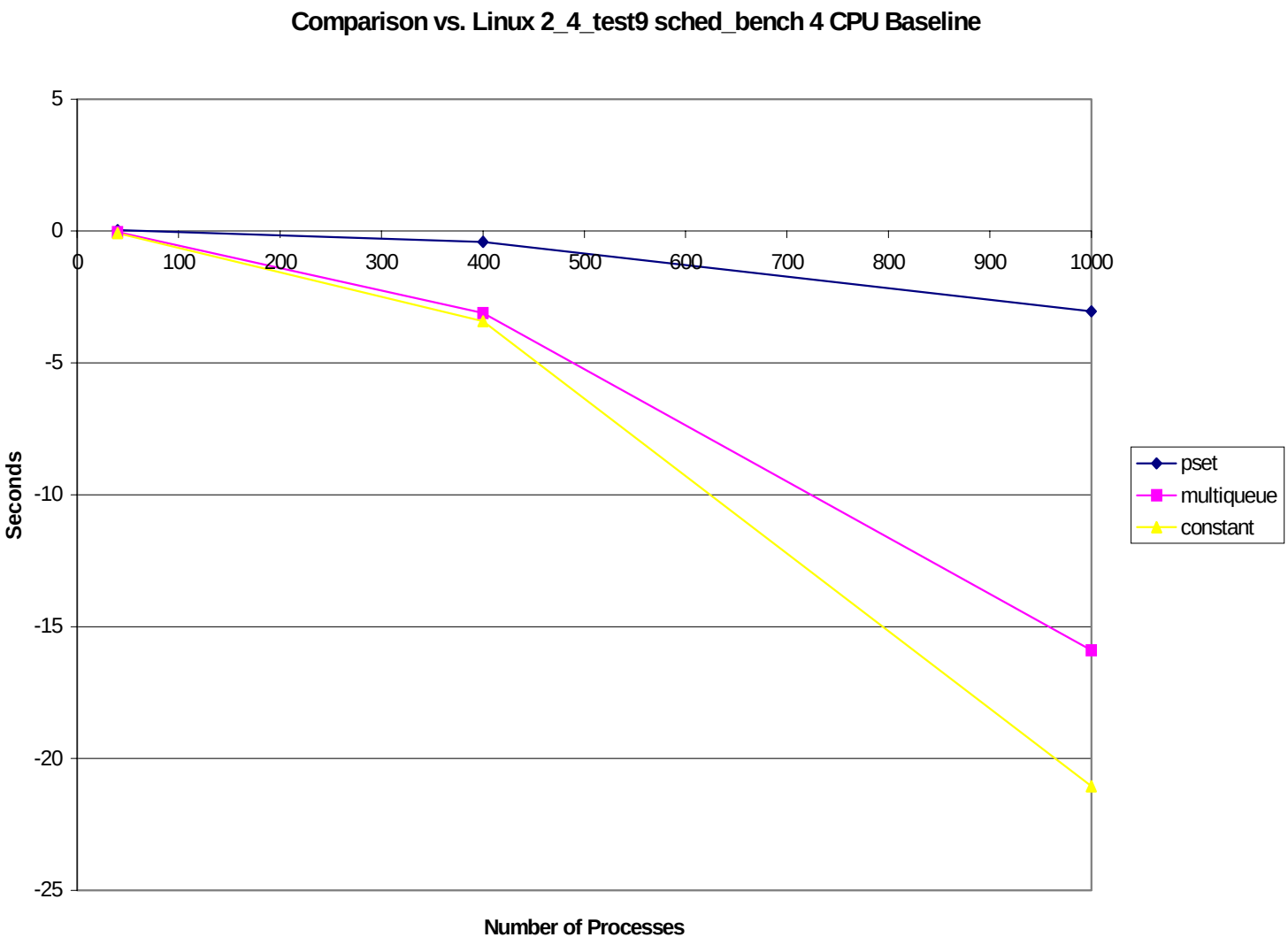

Figure 6: sched_bench results

One important side note relates to the `sched_policy` function call out pointers. If your scheduler changes states, it may be more efficient to write separate call out functions and switch them atomically at run time rather than stuff spaghetti logic into a single all-purpose function. A single extra line in the inner loop of the scheduler can have a huge impact on performance.

The entire development experience was so positive that I now use Linux to prototype new features for our product before running on other operating systems.

4.2 Recommendations

I highly recommend that people who have written their own schedulers in the past, rewrite with this modular scheme. Even people in university OS courses can play with the internals with relatively low risk. In the development process, however, I do have some suggestions.

- Don’t reinvent the wheel. Start from a working pattern, and you can be testing in hours instead of days. Due to the sheer amount of plumbing that has to work together, if the module builds, installs, and runs on an idle system, the job is half finished.
- Run a simple `ls` command. If the shell prompt comes back, congratulations.
- Write an exhaustive test for your `ioctl` interface. Remember, the module is root.
- Write an automated basic functionality test to make sure that the scheduler behaves as expected. This is especially important to make multi-platform and multi-OS-version testing consistent and painless.
- Run `sched_bench` to see if it stays up under 1000 hungry processes.
- Compare results against the default scheduler and check for scalability issues.
- Run the bounce demo under loadas described in section 2. It’s a great combinatorial/IO test.

References

[1] "Standalone Device Drivers in Linux", Theodore Ts'o, Proceedings of the1999 USENIX Annual Technical Conference, June 1999
http://www.usenix.org/publications/library/proceedings

[2] "Red Hat Documentation – Kernel Modules"
http://www.europe.redhat.com/documentation/HOWTO/Kernel-HOWTO-10.php3

[3] "Plug-In Scheduler Policies Web Page"
http://resourcemanagement.unixsolutions.hp.com/WaRM/schedpolicy.html

[4] "Linux SMP Scheduler", Moshe Bar, Byte Magazine, November 29, 1999
http://www.byte.com

[5] "Open GL Mesa Library and Demos Download page"
http://sourceforge.net/projects/mesa3d

[6] "The Linux Edge", Linus Torvalds, 1999, in Open Sources, Eds. DiBona, Ockman, & Stone, O'Reilly, Sebastopol, Ca., page 110.

[7] "Scalable Linux Scheduling", Molloy, Honeyman, & Bryant
http://www.research.ibm.com/acas/projects2000/ScalableLinuxScheduling.htm

[8] "Java Technology, Threads, and Scheduling in Linux", Bryant & Hartner
http://www.ibm.com/developerworks/library/java2

[9] "Linux Scalability Effort Project Page"
http://sourceforge.net/projects/lse

[10] "Process Resource Manager Project Page"
http://www.hp.com/go/PRM

[11] "A Hierarchical CPU Scheduler for Multimedia Operating Systems", P. Goyal, X. Guo, & H.M. Vin, Proceedings of 2nd Symposium on Operating System Design and Implementation (OSDI'96), Seattle, WA, pages 107-122, October 1996
http://www.cs.umass.edu/~lass/software/qlinux

[12] "Rik van Riel's Scheduler Page"
http://linux-patches.rock-projects.com/v2.2-c/schedule-bigpatch.html