In keeping with other kernel software devices like `/dev/kmem`, the major number for `/dev/sched` was set to 1. The minor was chosen as the next number available in the sequence, 10. The following files define the behavior of `/dev/sched`:

- mem.c: initialize the file operations table for major 1, minor 10 to be sched_ops.
- alt_policy.c (new): initialize sched_ops to NULLS and define a method for changing the ops for read, ioctl, alt_policy_name, and the global CPU policy array.

```
extern int register_sched( char *name, struct sched_policy **,
                           struct file_operations *);
extern int unregister_sched( char *name);
```

The register routine checks to make sure there isn't already a global policy defined, and returns EBUSY if there is. On success, it sets the global CPU policy array, changes the active scheduler name, and prints a console message, returning 0.

The unregister routine checks to make sure you are removing the module you think you are. If permitted, the global variables are reset to the default, all processes and CPUs are returned to the standard scheduler, and an unload message is printed to the console.

The `read_sched` routine tells you which scheduler is currently configured, and provides a default operation if the alternate scheduler has none. When no alternate schedulers are loaded, a `cat /dev/sched` command will produce the the output.

```
scheduler  : linux
```

4.5 Easy Configuration

By default, the `allow alternate schedulers n/y` prompt is set to NO because the designers did not want to impede or confuse the common user.

As an aside, the QoS scheduler prompt was changed to clarify that the feature is a networking feature, not a scheduler feature.

5. Installation Notes

Testing for this procedure used Linux version 2.2.14.5.0 (Redhat 6.2), Linux 2.2.16, and Linux 2.4 beta-test version 7 The kernel hooks to allow loadable scheduler modules may be incorporated into your system using these steps:

1. Download *ourpatchfile*
2. Start with a clean kernel tree (ksyms versioned variable names are *extremely* sensitive and don't always regenerate properly on a used tree).
3. cd new linux source directory
4. patch -Np1 < *ourpatchfile*
5. make config or xconfig
    - Select Alternate Scheduler Support in xconfig, and answer yes to allow alternate scheduler support
    - Make any other selections from the config menu required for your system
6. make dep bzImage module modules_install
7. Install new kernel through `make bzlilo` or for the safety minded
    - cp linux/arch/i386/boot/bzImage to /boot/vmlinux.altsched
    - Edit /etc/lilo.conf to add vmlinux.altsched
    - /sbin/lilo
8. Create the /dev/sched device
    - makedev /dev/sched c 1 10
    - chmod 666 /dev/sched
9. Reboot to new kernel
    - sync
    - reboot
    - Select new kernel name at boot prompt
10. /sbin/insmod pset

The utilities bundles can be installed by:

1. gunzip *bundlename.Z*
2. tar oxvf *bundlename*
3. cd pset
4. make install (as root)

6. Example Loadable Scheduler: Processor Sets

The scheduler is intended for multiprocessor platforms as a manageability aid. Processor sets allow tasks to be associated with a collection of processors rather than running on any available CPU on your machine. Think of a processor set as an isolated virtual partition of the CPUs on your machine. Scheduling in each processor set is done independently.

Why is this helpful? Processor sets enforce a resource prioritization on your system as well as a degree of isolation using the `psrset` user space utility or the C library interface. There is a class applications that need to process a constant stream of data without interuption, and need protection from usage spikes from other applications on the machine (tape backup, voice recognition, video streaming). Other types of applications may need special protection because they are high priority or high security (payroll, patient database, purchasing webpage). At the other end of the spectrum, there are programs that can grow without bound given sufficient resources (parallel compiles, web browsers, ray tracers). Each of these categories would be beneficial to isolate from the common interactive user.

Take the example where a 4 CPU web server has two special applications to isolate: credit_card (for security/reliability) and free_email (to limit how much processing is spent). In this case, you might create processor sets 1 and 2 for the protected programs using `psrset -c`. You can give each a dedicated CPU using `psrset -a`, leaving the rest (default processor set 0) for the interactive users, java programs, and so on. Individual processes may be moved over using `psrset -b` or executed in the proper set by `psrset -e`. The set membership of any process is available via `psrset -q`.

```
CPU             0             1        |       2      |       3
PSET            0             0        |       1      |       2
PURPOSE         all the rest           | credit_card  |  free_email
```

The default processor set always exists and may not be destroyed. All tasks and processors start out in

the system default group, including new logins and daemons like the swapper. For this reason, this implementation does not allow CPU 0 to be removed from the default group. Later versions might soften this restriction, but one CPU must always remain for the OS to function properly. Hence, this feature is of little value on a single CPU system.

A task or a CPU belongs to exactly one processor set at a time. However, processor sets with no CPUs can 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 which has no CPUs assigned, a task is effectively frozen, making no forward progress until such assignment is made. Compute intensive jobs not requiring immediate turn-around can be placed in such empty processor sets. After hours, when interactive users go home, one 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 at 8 AM and the CPU re-dedicated to interactive support.

When you destroy a processor set via `psrset -d`, all of its tasks and CPUs revert to the default system group unless you configure its attributes otherwise with `psrset -t`. If a processor set, such as the one containing the credit_card, is extremely important, you can set the NONEMPTY attribute to FAILBUSY. This means that any attempt to destroy this processor set when there is anything still running in it will result in an error. For transient or limited time processor sets, such as batch jobs, you may wish to simply kill off all the processes in the set by choosing the KILL attribute.

To list the current processor sets and their attributes, run `psrset -i`.

6.1 Implementation Details

For the sake of simplicity and security, only root or someone with system administrator capabilities can create, destroy, or change assignments for processor sets. The maximum number of processor sets PS_MAXPSET was limited to NR_TASKS because by that point every runnable job on the system would have its own set. Any additional sets would therefore be completely empty and pointless.

If a process is bound to a processor set that has no CPUs, it is effectively frozen. Even if you send it a kill signal, the process cannot respond to the signal and die, because this requires CPU usage. However, the moment you unbind such a signalled process, it will react appropriately.

When you `cat /dev/sched` you should see:

```
scheduler    : pset version
```

The private data structure, visible only from inside pset.c, looks like:

```
typedef struct processor_set_priv {
       psetid_t        ps_id;           /* created entity unique id */
       int             ps_spu_count;    /* number of cpus in this set */
       pset_attrval_t  ps_non_empty_op; /* behavior on delete of set in use */
       struct sched_policy* ps_next;    /* next pset in sorted list */
       struct sched_policy* ps_prev;    /* previous pset in sorted list */
} processor_set_priv_t;
```

Most of the other details can be learned from reading the man pages or pset.c.

unnecessary amounts of internal kernel implementation detail and variables.

```
+#ifdef CONFIG_ALTSCHED
+void
+resched_cpu( int cpu )
+{
+       /* entry point for loadable modules when policy changes for a CPU */
+#ifdef __SMP__
+       if ((cpu >= 0) && (cpu < NR_CPUS) && cpu_curr(cpu)){
+               cpu_curr(cpu)->need_resched = 1;
+               smp_send_reschedule(cpu);
+       }
+#else
+       current->need_resched = 1;
+#endif
+}
+#endif
```

This callout methodology means that when you run the default scheduler, even with the ability for alternate schedulers configured, you should suffer no measurable performance penalty.

4.3 Exported Symbols

The new ksyms.c file now includes `altpolicy.h` in order to expose the defined interfaces to loadable modules. It also exports the following scheduler variables so that they are readable from the new scheduler modules:

- When SMP is defined
  - smp_num_cpus (originally in arch dep ksyms.c, basic for any MP code)
  - cpu_number_map (translates physical CPU number to logical)
- When CONFIG_ALTSCHED is defined
  - register_sched (new)
  - unregister_sched (new)
  - resched_cpu (new)
  - nr_running (the current length of the run queue for default group initialization)
  - policy_of_cpu (to load/unload global policy)
  - pidhash (for task list traversals macro)
- When both SMP and CONFIG_ALTSCHED are defined
  - tasklist_lock (prevents concurrent update of list)
  - runqueue_lock (only one CPU can pick its next runner at a time)
  - task (to find the idle task for a given CPU)

Two variables from the Redhat 6.2 i386 symbol file were pulled up to the main ksyms.c file to make drivers multiprocessor capable: cpu_number_map, and smp_num_cpus. This file was not rebuilding properly at dependency time in the test environment, and loadable modules were not able to use the variables. There were no true architecture dependencies in the algorithms. A more experienced Linux kernel developer may be able to rectify this problem in a better way. These same dependencies regenerated properly for Linux 2.4, and no changes to any architecture dependent files were required.

4.4 The /dev/sched Interface

```
+#ifdef CONFIG_ALTSCHED
+       if (p->alt_policy)
+               p->alt_policy->sp_runnable--;
+#endif
        nr_running--;
        next->prev_run = prev;
        prev->next_run = next;
```

6. sched.c: declare a global per-CPU array

```
struct sched_policy ** policy_of_cpu = NULL;
```

7. sched.c: inside repeat_schedule label of schedule routine

```
         * this is the scheduler proper:
         */

+#ifdef CONFIG_ALTSCHED
+       if (policy_of_cpu && policy_of_cpu[this_cpu]->sp_choose_task){
+               next = (*policy_of_cpu[this_cpu]->sp_choose_task)(prev, this_cpu
);
+               goto choice_made;
+       }
+#endif
+
        p = init_task.next_run;
        /* Default process to select.. */
        next = idle_task(this_cpu);
```

8. sched.c: in the routine preemption_goodness, add the lines

```
 static inline int preemption_goodness (struct task_struct * prev,
                                struct task_struct * p, int cpu)
 {
+#ifdef CONFIG_ALTSCHED
+       if (p->alt_policy)
+               return (*p->alt_policy->sp_preemptability)(prev, p, cpu);
+#endif
        return goodness(prev, p, cpu) - goodness(prev, prev, cpu);
 }
```

9. sched.c (timing.c in 2.4): in update_one_process add the conditional callout

```
        p->per_cpu_utime[cpu] += user;
        p->per_cpu_stime[cpu] += system;
        do_process_times(p, user, system);
+#ifdef CONFIG_ALTSCHED
+       if (p->alt_policy && p->alt_policy->sp_handle_ticks)
+               (*p->alt_policy->sp_handle_ticks)(p, ticks);
+#endif
        do_it_virt(p, user);
        do_it_prof(p, ticks);
```

10. sched.c: bottom of file, add a new function `resched_cpu` to send a reschedule signal to a given CPU in a machine-independent fashion. Whenever a CPU changes policies, the currently running process should be purged. Defining this module-visible function was cleaner than exposing the

6.2 Kernel Patch

The kernel patch includes an example loadable scheduler policy, processor sets, encapsulated in the files `drivers/sched/pset.c` and `pset.h`. The makefile in this directory may be expanded to include other schedulers developed by the user or downloaded off the Internet. If any pset.o variable is not found during the insmod, you probably didn't start with a clean tree (the ksyms.o references weren't properly re-generated) or there was a mismatch between uniprocessor and SMP builds on the same box.

Module items called by the Linux kernel include:

- pset_ioctl: the control/monitoring interface for the user
- pset_choose_task: picks the next task to run
- pset_preemptability: decides if the candidate should kick off the task currently running on this CPU
- init_module: calls pset_init at module load time
- cleanup_module: unregisters pset scheduler at module remove time

6.3 Utilities Bundle

In order to take full advantage of the new module, two user interfaces have been provided : the pset library for C coding, and the command line utility `psrset`. The `psetps` command is a more elaborate version of `psrset -q` that takes all the same arguments as `ps` and prepends the current processor set of each process. Documentation for user interfaces is included in the utility bundle and current links appear below.

- psrset utility (1M)
- psetps utility (1)
- pset_create (2)
- pset_destroy (2)
- pset_assign (2)
- pset_bind (2)
- pset_getattr(2) & pset_setattr(2)
- pset_ctl (2)

A miniature test suite `pset_test` is included to give administrators confidence in the feature, demonstrate typical usage, and to provide regression tests for future changes to the code. It is built by the same makefile that compiles and installs the utilities. Do not run it while there are active pset users on the system because it starts by destroying all existing psets.

7. Making Your Own Scheduler

The private data field is purposely opaque, visible only from within the individual implementation. With the tools provided, a host of other policies could be implemented (batch, gang, NEC flock manager, duration-based queues, and the HP Fair Share Scheduler, just to name a few). Below are some examples of the private data that might be contained in various policy flavors. They are provided primarily for the potential they represent, not as hard recommendations.

- A variation of the pset feature provided would be the NT jobs abstraction. One can see how the template could be easily adapted.

```
struct NT_PROCESSOR_SETS {
        psetid_t        ps_id;          /* created entity unique id */
        int      num_cpus;              /* number of CPUs in this set */
        int      security;              /* who has access/admin authority */
        int      inheritance_model;     /* are children in same set? */
}
```

- Yet another Job Queue implementation could be prioritized and have properties like duration, maximum file size, and total memory consumption among others.

```
struct PRIO_JOB_Q {
        int      pq_id;                 /* created entity unique id */
        int      num_cpus;              /* number of CPUs in this set */
        int      priority;              /* relative importance */
        int      duration;              /* max duration for members on q */
        int      expiration_action;     /* what to do when time is up */
        int      max_file_size;
        int      exceed_action;
}
```

- The user-based fair scheduler could be described in these terms as well. Note that each user only has a pointer to one of these structures. The CPU-based structure can point to a root structure with an empty private section.

```
struct USER_FAIRNESS {
        uid_t    uid;                   /* user id represented */
        int      ref_count;             /* when 0, can be freed ? */
        int      shares_left;           /* value of per-user counter */
        int      user_nice;             /* relative priority of the user */
}
```

- A more general sharing scheme that can assign users or applications to abstract groups (like ShareII, QLinux, or the HP Fair Share Scheduler).

```
struct FAIR_GROUPS {
        int              fg_id;         /* created entity unique id */
        int              fg_ticks;      /* total run time accumulated */
        int              fg_pending;    /* number of tasks actually on CPUs */
        int              fg_shares;     /* user input shares */
        struct implementation details;  /* data structure links, etc */
}
```

One can see how a wide range of policies can be represented by these mechanisms. In psuedo-code a general skeleton of a loadable scheduler module might look like this:

```
my_module.c:
        preemptability_func
        choose_task_func
        handle_ticks_func (if any)

        ioctl_interface (if any)
        read_based_/dev/sched_interface (if any)

        move_task_to_group
                                                int ticks);
        char            sp_private[0]; /* per policy private data. MUST be last*/
} sched_policy_t;
```

Note that the run queue lock must be held during sp_choose_task and continue to be held upon return.

The function `sp_handle_ticks` is passed a task_struct instead of just the sched_policy because the process in question may need to be moved to another policy if it exceeds a certain time limit. Because not every scheduler makes decisions based on per-group usage, the sp_handle_ticks function pointer may be NULL, in which case the data collection action is skipped.

Many other per policy functions and variables are possible, but keeping it simple, the bare minimum was exposed; anything else can be implemented inside the per-policy private data area `sp_private`. Advanced versions envisioned for gang scheduling may need to add call outs for:

- fork
- exit
- suspend

The `altpolicy` header file also contains declarations for common functions and macros required by most schedulers :

- unregister_sched
- register_sched
- resched_cpu
- struct file_operations sched_fops
- idle_task
- goodness

The following kernel hooks are all surrounded by ifdef CONFIG_ALTSCHED:

1. sched.h: declaration and initialization of the task structure field sched_policy_t *alt_policy
2. fork.c: one line inheritance of alt_policy field of the task structure

```
        p->counter = current->counter;
+#ifdef CONFIG_ALTSCHED
+       p->alt_policy = current->alt_policy;
+#endif
```

3. sched.c: changed the hardcoded -1000 to the define IDLE_WEIGHT for consistency between schedulers.
4. sched.c: when adding to run queue

```
        p->next_run = next;
        next->prev_run = p;
+#ifdef CONFIG_ALTSCHED
+       if (p->alt_policy)
+               p->alt_policy->sp_runnable++;
+#endif
        nr_running++;
```

5. sched.c: when deleting from run queue

- linux/kernel/alt_policy.c (new)

3. Changes required for easy configuration of new feature
   - linux/Documentation/Configure.help (7)
   - linux/Makefile (1)
   - linux/drivers/Makefile (5)
   - linux/drivers/char/Config.in (4)
   - linux/kernel/Makefile (4)
   - linux/net/Config.in (2)
4. Sample loadable scheduler module: processor sets
   - linux/drivers/sched/Makefile (new)
   - linux/drivers/sched/pset.c (new)
   - linux/drivers/sched/pset.h (new)

4.2 Core OS

The heart of the generalized scheduling policy changes obey the following rule:

If a policy has been defined for a particular CPU or process, the desired function is invoked through indirection; otherwise, the logic falls through to the default Linux scheduling behavior.

This directive implies a new per-process property, as well as a per-CPU property for the current policy. For generality and symmetry, the two policy labels will share the same structure. By convention, a process inherits the policy of its parent and remains there until changed.

To decide what the minimum abstraction should be, designers need to examine the key decision points that define a scheduler policy. In no particular order:

- Rules for deciding which process runs next
- Recalculation of timeslice remaining
- Rules for deciding whether one process preempts another
- Bookkeeping done each clock interrupt
- Rules for inserting to/deleting from the run queue

The recalculation is normally done when no runnable candidates can be found, and may be integrated with the choice function. Overhead for new routines for run queue add and delete was considered too costly for this proof of concept. As a compromise, the number of jobs runnable in any given category is an interesting scheduler-visible statistic available at a low cost. According to the white paper on their web page, the Qlinux implementation requires this per-policy value, and others can use it for optimization.

The scheduling policy structure is therefore defined in alt_policy.h as:

```
typedef struct sched_policy {
        int             sp_runnable;    /* number of members on the run q */
        struct task_struct * (*sp_choose_task)(struct task_struct *current,
                                               int cpu);
        void                (*sp_preemptability)(struct task_struct *current,
                                               struct task_struct *thief,
                                               int cpu);
        void                (*sp_handle_ticks)(struct task_struct *current,
        move_cpu_to_group

        create_instance
                malloc
                initialize function pointers and private region
                increment MODULE users to ensure EBUSY when rmmod

        destroy_instance
                free
                decrement MODULE users

        init_module
                intialize policy_of_cpu array [NR_CPUS] to default policy
                set up a file operations structure
                register your alternate scheduler
                set every running process to an appropriate policy
                turn on feature variables

        cleanup_module/unregister
                turn off feature variables
                unregister
```

Recommended conventions include a macro to hide the implementation of the private region:

```
#define PSET_PRIV(x) ((processor_set_priv_t *)(&(x)->sp_private))

/* example usage */
my_id = PSET_PRIV(curr_pset)->ps_id;
```

The space for the sched_policy structure and the private structure should be allocated at the same time for simplicity. This is an example initialization:

```
new = kmalloc(sizeof(struct sched_policy)+sizeof(processor_set_priv_t),
              GFP_KERNEL);
if (!new)
        return -ENOMEM;

new->sp_choose_task = default_pset->sp_choose_task;
new->sp_preemptability = default_pset->sp_preemptability;
new->sp_handle_ticks = default_pset->sp_handle_ticks;
PSET_PRIV(new)->ps_spu_count = 0;
PSET_PRIV(new)->ps_non_empty_op = PSET_ATTRVAL_DFLTPSET;
...
```

The only other recommendation would be to test your new scheduler *extensively* before deploying it. While giving developers more power, this drop-in scheduler feature demands more responsibility when altering sensitive code.

8. Performance Measurement

Since performance was number two on the requirements list, time should be spent comparing the performance characteristics of

- Baseline Linux 2.2.14.50 (Redhat 6.2)
- Baseline Linux 2.4 beta-test version 7
- Our patch with scheduler hooks only, nothing loaded

- Pset module installed with everything in the default set
- Both uniprocessor and SMP kernel builds

To measure the difference in scheduling latency, the designers began with a benchmark by Larry McVoy called mhz which performs a computation of known duration a few million times to determine the approximate megahertz rate of a CPU. The computation was then increased by a factor of 100 so that it would be guaranteed to cross scheduling quanta. In theory, any added time should be due solely to scheduling overhead. To force frequent context switching, and account for runqueue traversal overhead, the main program spawns N children and wait for all of them to complete. The difference between 1 copy and 100 should be due to the initial fork overhead, plus the increased traversal time per scheduling decision.

Ideal time for a single process with no fork or scheduler overhead would be about 10000000000/clock_speed (22222222 for 450 MHz, 18181818 for 550 MHz). Ideal scaling would be single_process * ceiling(copies/cpus). For example, a 4 cpu box spawning 40 copies would expect to take at least 10 times longer than a single benchmark copy.

Times were recorded in microseconds and the average of three runs was taken. The tests were repeated for both single CPU and SMP versions of the kernels. The single CPU platform was a 450 MHz Pentium II with 512 KB cache, and about 44 system daemons running in the background. The four CPU platform was a 550 MHz Pentium III with 512 KB cache, and about 44 system daemons running in the background.

8.1 Redhat 6_2 Results

```
SINGLE CPU RAW DATA  (microseconds run time)

copies      1         10         100         1000

baseline    22364744  223642189  2237091802  22430762387
hooks only  22364907  223647447  2237107329  22429569811
one pset    22364406  223641866  2237107597  22430279717


FOUR CPU SMP RAW DATA (microseconds run time)

copies      1         4         40         400         1000

baseline    18198557  18223651  182160908  1821823149  4558992539
hooks only  18199491  18235188  182154058  1821863637  4558911971
one pset    18199051  18222965  182126227  1821866246  4558357402
```

The raw data alone doesn't tell us much other than

- Larger numbers of processes means more scheduling overhead.
- With fewer than 100 processes, the statistical noise may be greater than any differences. Therefore, it takes a long-running scheduler-intensive load before differences stand out clearly and reliably.

First, the scaling properties of the different methods were examined. A kernel was compared against itself for varying numbers of processes. The total completion time for N processes was compared against the mean time for a single process scaled up by a factor of N (perfect scaling). The difference between

of the kernel in any way. When the feature is configured but no module is loaded, there should be no measurable increase in scheduling decision times.
- Flexibility demands that the designers try to define interface structures and methods that can be used to implement virtually any scheduler. Any tools that may be needed to translate the functionality of existing scheduler feature patches should be provided. Although not an exhaustive list, this package defines a group of generic callout operations. Like a filesystem implementation, these callouts must be supplied by the new module and are invoked by the kernel as needed.
- Extensibility means that the designers should not define a static structure for these policies in the kernel. The new structure should be opaque to the existing Linux scheduler. Adding system calls for each new version could also be onerous. Therefore, the prototype uses the existing ioctl interface for the software device. In this way, the module loaded defines its own interface dynamically.
- Usability asks that some sort of `/proc` informational interface be provided, as well as user-level utilities and documentation. To provide this capability, the device name `/dev/sched` was chosen as the standard interface to all alternate schedulers. The supported operations on this device will be read and ioctl. A sample implementation of each appears in the example pset scheduler found in the source directory `linux/drivers/sched`. Any future loadable schedulers should be found in this directory as well.
- Finally, simplicity says that the designers should:
  - Implement the bare minimum for version 1
  - Only support the loading one such alternate scheduler into the kernel at any given time. Later versions might allow multiple policies or nesting, but interactions make for too many complications for a first release.
  - Try to encapsulate our changes into one kernel and one header file
  - Provide generic code likely to be common to all schedulers
  - Allow easy configuration from the `xconfig` menu

4. Modifications to Linux

This section begins with a categorized file list and then explains the important changes in detail.

4.1 File List

In total there are 5 new and 12 modified files in this patch. The number of lines added or changed for each file appears beside it below. The counts do not include blank or comment lines, but do include the ifdefs required to isolate the changes. All 17 files can be categorized as follows:

1. Core OS changes
   - linux/include/linux/sched.h (4)
   - linux/kernel/fork.c (3)
   - linux/kernel/sched.c (40 added, 40 moved to altpolicy.h)
   - Exposing more kernel variables to external schedulers
     - linux/arch/i386/kernel/i386_ksyms.c *Redhat 62 only*(2)
     - linux/kernel/ksyms.c (19)
2. Implementing the /dev/sched interface
   - linux/drivers/char/mem.c (4)
   - linux/include/linux/altpolicy.h (new)

does not pretend to be the definitive treatise on any of the above; rather, the information in this document is meant to facilitate sharing and discussions in the Open Source community. Address questions or comments on this document to :

`loadable scheduler feedback (prmfeedback@rsn.hp.com)`

2. Motivation

Why would anyone want a different scheduler? As more computer users move to Linux alternatives, there is an increasing demand for features they liked on proprietary operating systems. With the advent of the IA-64 architecture, several different operating systems will be able to run on a single machine. Compatibility between operating system feature sets is therefore desirable. For this reason, among others, it is increasingly difficult to satisfy all customers with a single scheduler policy.

The accepted Linux method of providing such features is through loadable kernel modules. Not only does this provide a way for people who like the default Linux scheduler to prevent unnecessary bloating of their OS image, but it gives the user a way to disable, fix, and re-enable features without a reboot.

Already, there is a profusion of proprietary schedulers for Linux:

- Qlinux Fair Queuing
- SGI CPU sets
- Real-time extensions
- User-based fair scheduler by Rik van Riel
- Others under development like ShareII

None of these are loadable modules. Rather, these are all somewhat mutually-exclusive, kernel-intrusive, and have not made it into the main Linux code stream. Someone needs to provide the Linux community with something open source, general purpose, and useful with low performance impact that can become the *standard* for implementing alternate scheduling policies.

There are many other scheduler features that might be developed for Linux if such a standard existed: batch, gang thread scheduling, NEC flock manager, Microsoft Job Objects, and the HP Fair Share Scheduler, just to name a few. Many of these were developed to improve the manageability of a multiprocessor technical computing environment. Several third party vendors take advantage of these features and could port to Linux if the proper foundation existed.

3. Requirements

For the purposes of this exercise, consider a scheduler to be simply a device driver for the CPU. Yet every device driver has constraints. Below are listed the core requirements considered when beginning this project:

- It is essential that our changes in no way affect the current POSIX scheduling class prioritization between realtime round-robin, realtime FIFO, and OTHER. Our alternative schedulers are treated as subclasses within the OTHER class, and need not interfere with the existing realtime classes.
- The second biggest consideration is probably performance. If the appropriate configuration flag is not set, no part of this scheduler feature will be built, and it will not affect the size or performance

the two is our added overhead for the method. To put it all in a metric which could be analyzed easily, the overhead difference was initially translated into a percentage of the real measurement. Unfortunately, percentages were too coarse. If a scheduler impacts your performance by a percent, you should worry. Therefore, the result was shifted by 6 decimal digits to indicate how many microseconds difference a given scheduler would make in a second of computation.

```
ADDED SCALING OVERHEAD (microseconds per second)

             UNIPROCESSOR         4-CPU
copies       100   1000           400    1000

baseline     276   2930           1080   2052
hooks only   271   2870           1051   1983
one pset     298   2937           1076   1885
```

As expected, the overhead increases almost linearly with the number of processes. The measurements for all OS versions are almost identical to the baseline within the bounds of experimental error. Most of the variation on this table is due to the differences in the widely fluctuating single-process times. However, according to our experiments, the apparent improvement for multi-CPU processor sets is real, and due to a single optimization in the recalculation of counter values.

The overhead difference is less on the multiprocessor systems primarily because of the faster clock speed, and because some overhead may no longer be serialized. Our sample scheduler performed better on a multiprocessor system because that is the intended usage, and thus where improvement efforts were concentrated.

During the tuning of this patch it became evident how sensitive this critical path code can be to changes. One unnecessary computation in the inner loop of the choose_task function, for example, has been known to add an extra two to three milliseconds to the overhead in the multi-CPU intermediate process count range. It is vital that anyone making changes or creating their own scheduler have benchmarks to gauge efficiency. For this reason, the benchmark source `sched_bench.c` was included in the distribution.

Next the different variants were compared against the baseline using the simple computation:

```
(variant - baseline)
```

Thus, positive scores mean that the variation takes longer, while negative numbers show it to be faster than the baseline.

```
OS COMPARISON  (microseconds per second)

                UNIPROCESSOR            4-CPU
                10   100   1000        40   400  1000

hooks - base =   24     7    -53       -38    22   -18
pset  - base =    1     7    -22      -190    24  -139
```

As before, there is little or no difference between the OS versions. In conclusion, the hooks for drop-in policies have no negative affect on scheduler performance. The comparison times are at least as good as the baseline. Although the time to execute a new policy can be a little more than the default, the impact can be limited by careful optimization strategies.

8.1 Linux 2_4 Results

Due to the significant changes in scheduler code and cacheline layout, the measurements were made again for the port to Linux 2.4 beta-version 7. The only difference for the benchmark was an improvement designed to eliminate forking overhead from the equation, a shared memory barrier that prevents any children from beginning until all children have been forked. The OS comparison adds single-cpu SMP because SMP is not the default configuration.

```
SINGLE CPU SMP RAW DATA (microseconds run time)

copies      10         100         1000

baseline    223669684  2237813461  22476639595
hooks only  223640394  2237517890  22473158913
one pset    223635808  2237276927  22444151229


FOUR CPU SMP RAW DATA (microseconds run time)

copies      40         400         1000

baseline    182360435  1824685492  4573495161
hooks only  182370057  1824820054  4573574065
one pset    182355817  1824386349  4571736520

ADDED SCALING OVERHEAD (microseconds per second)

            1-CPU SMP           4-CPU SMP
copies      100   1000          400    1000

baseline    499   4879          593    3167
hooks only  497   4855          613    3132
one pset    411   3590          454    2808

OS COMPARISON  (microseconds per second)

                1-CPU SMP              4-CPU SMP
                10   100   1000        40    400  1000

hooks - base =  -131  -132  -154       53      74    17
pset  - base =  -151  -240  -xxx      -25    -164  -385
```

As demonstrated with the Redhat 6.2 version, the new code adds no appreciable overhead. In the 1000 process case for the processor set example, the overhead even seems to be reduced.

Loadable Scheduler Modules on Linux (beta)

Hewlett-Packard
Management Solutions Lab (MSL)

9/7/00

Scott Rhine

Table of Contents

---

1. Introduction

This patch enables you to write and use loadable kernel modules to change your Linux machine's scheduler policies. Linux already employs loadable modules to pull in optional device drivers into your kernel without rebooting your system. This new feature extends the technology to the scheduler section of the kernel.

This document first discusses why someone would want to change scheduling policies. Then it explains the few changes necessary for the implementation, new structures, and recommended conventions. Finally, it provides evidence that these changes are harmless to the code base.

An example scheduler is provided, an open-source implementation of the *processor set* functionality found in various flavors on UNIX and NT platforms. Processor sets allow tasks to be associated with a collection of processors rather than running on any available CPU on your machine. Think of a processor set as an isolated virtual partition of the CPUs on your machine, improving the manageability of a multiprocessor environment. Scheduling in each processor set is done independently. A bundle of utilities, tests, and man pages accompany the example for improved usability.

The source bases used for operating system changes were Linux 2.2.14-5.0 (the popular Redhat 6.2 distribution), Linux 2.2.16, and the Linux 2.4 beta-test version 7. The test platforms were one and four-CPU Pentium boxes.

Disclaimer: This document assumes a basic knowledge of Linux, schedulers, and loadable modules. It