Energy-aware virtual CPU scheduler

April 9, 2015

Contents

Chapter 1

Introduction

Written by Andrii Berezovskyi & Armand Zangue

Scope

This is a documentation for the scheduler developed as part of the project on Operating Systems at KBS. The main purpose of the scheduler is to save power on a server mainly used to run virtual machines. The VM hypervisor for this project is QEMU[1].

The remaining part of the document is structured as follows: first, the scheduler design is described. Then the detailed design of the scheduler subsystem is given, followed by the documentation of the power-saving subsystem. Finally, instructions on how to build, install and debug as well as evualuate the developed project are provided.

Overview

To achieve the project purpose, the CPU is allocated to virtual machine CPUs (VCPUs) in superperiods of a fixed duration of 200ms, so that each VCPU has a certain time to run proportionally to its predefined share of the CPU. Before the end of each superperiod, if there is any time left, it can be used to actually save energy.

[1] Which, in turn, relies on KVM for acceleration and SMP support using options `-enable-kvm` and `-smp <n>` respectively

Linux has a modular scheduler architecture that allows different scheduling policies to co-exist. These policies are wrapped up in so-called scheduler classes. Thus, implementing a new scheduling policy for Linux mainly consists of writing a new scheduler class.

Originally, there are 2 main scheduling classes: Real-Time and Completely Fair Scheduler (CFS). The Real-Time scheduling class is responsible for the following policies:

- `SCHED_FIFO`
- `SCHED_RR`

The CFS scheduler, on another hand is providing other 2 policies:

- `SCHED_NORMAL`
- `SCHED_BATCH`

By default, newly created processes are assigned the *normal* scheduling policy and all forked subprocesses will inherit it[2].

Existing schedulers serve their purpose fairly well, robustly scheduling the tasks while meeting the deadlines and distributing the machine resources. However, when the system is frequently remaining idle, the power is not saved or the saving is enable only via the **ondemand in-kernel governor** [3] or through the **dynamic frequency scaling** techniques, such as *Intel SpeedStep* and *Intel Turbo Boost*.

This way, when there are no tasks in the system ready for the execution, the control is relinquished to the *idle* task.

The idle task basically halts the processor (for *x86* architecture):

```
52 static inline void native_halt(void)
53 {
54   asm volatile("hlt": : :"memory");
55 }
```

Listing 1.1: arch/x86/include/asm/irqflags.h

This does not yield significant power savings alone, even when combined with the aforementioned techniques.

NOTICE

The line numbering corresponds to the codebase state as of commit 9125134e. In case of discrepancies, check out the repository at the given commit hash.

[2] Both policy and inheritance can be changed via the `sched_setscheduler(2)` call.
[3] See https://www.kernel.org/doc/ols/2006/ols2006v2-pages-223-238.pdf for details

Chapter 2

Scheduler

Written by Andrii Berezovskyi, Armand Zangue & Tamilselvan Shanmugam

Integration into the kernel

The scheduler was integrated into the kernel scheduling subsystem by making changes in the following places.

Data structures

The OSPJ scheduler has introduced several changes to the data structures, mainly located in two `sched.h` files[1].

First, the `'struct ospj_rq'` was defined:

```
365 struct ospj_rq {
366     struct list_head ospj_list_head;
367     struct list_head *ptr_eligible_q;
368     struct list_head *ptr_waiting_q;
369     struct list_head eligible_q;
370     struct list_head waiting_q;
371     struct rq *rq;
372     struct task_struct *idle;
373     unsigned int nr_running;
374     unsigned int idle_request;
375     unsigned int period_ticks;
376 };
```

Listing 2.1: kernel/sched/sched.h

[1] `kernel/sched/sched.h` and `include/linux/sched.h`

Main point of interest is the existence of two queues, `eligible_q` and `waiting_q`. The meaning of both queues will be explained below. Another interesting components are the queue pointers also defined in the structure. They are specifically defined for easy process of swapping the queues (also explained later). The runqueue is designed to hold all the tasks assigned to the responsible scheduling class on one processor. Each CPU has its own runqueue, which holds the runqueue stuctures for each scheduling class:

```
413 /*
414  * This is the main, per-CPU runqueue data structure.
415  * ...
416  */
417 struct rq {
418     ...
419     struct cfs_rq cfs;
420     struct rt_rq rt;
421 #ifdef CONFIG_SCHED_VMS
422     struct ospj_rq ospj;
423 #endif /* CONFIG_SCHED_VMS */
424     ...
425 }
```

Listing 2.2: kernel/sched/sched.h

Finally, the scheduling class is defined on a per-task basis:

```
1055 struct task_struct {
1056     ...
1057     const struct sched_class *sched_class;
1058     ...
1059 #ifdef CONFIG_SCHED_VMS
1060     struct sched_vms_entity vms;
1061     struct list_head ospj_list_node; /* list item to insert task into Q
         */
1062     unsigned int share;
1063     unsigned int ospj_time_slice;
1064     unsigned int ospj_assigned_time_slice;
1065 #endif /* CONFIG_SCHED_VMS */
1066     ...
1067     unsigned int policy;
```

Listing 2.3: include/linux/sched.h

In order to understand how the `sched_class` is set, we need to look into the scheduler hierarchy.

SCHEDULER HIERARCHY

In order to register a scheduler in a kernel, one must provide a structure initialized with the pointers to the scheduler functions. Below you can find the excerpt of the structure defined for our scheduler:

```
556 const struct sched_class ospj_sched_class = {
557     .next               = &idle_sched_class,
558     .enqueue_task       = enqueue_task_ospj,
559     .dequeue_task       = dequeue_task_ospj,
560     .yield_task         = yield_task_ospj,
561
562     .check_preempt_curr = check_preempt_curr_ospj,
563
564     .pick_next_task     = pick_next_task_ospj,
565
566 ...
567
568 };
```

Listing 2.4: ospj.c

This allows the scheduler subsystem (implemented mainly in `core.c`) to perform generic calls to every scheduler, e.g. the following snippet from the `pick_next_task` function:

```
2949 for_each_class(class) {
2950     p = class->pick_next_task(rq);
2951     if (p)
2952         return p;
2953 }
```

Listing 2.5: core.c

However, before the function of our new scheduling class is going to be called in this manner, we have to put a pointer to our `struct sched_class` structure into the linked list that is being iterated in this `for` loop. The head of this list is defined in `sched.h`[2]:

```
1041 #define sched_class_highest (&stop_sched_class)
```

Listing 2.6: kernel/sched/sched.h

Then, each next node of the list is defined in the `next` field of the `struct sched_class` structure:

```
105 const struct sched_class stop_sched_class = {
106     .next           = &rt_sched_class,
107     ...
108 }
```

Listing 2.7: Designated initializer in kernel/sched/sched.h

The original hierarchy is ordered as follows:

[2] there is also defined another header under `include/linux/sched.h`

1. Stop scheduler class.
2. RT scheduler class.
3. Fair scheduler class.
4. Idle scheduler class.

Our implementation inserts a new scheduler class after the fair scheduler right before the idle class:

```
6167 const struct sched_class fair_sched_class = {
6168 #ifdef CONFIG_SCHED_VMS
6169     .next           = &ospj_sched_class,
6170 #else
6171     .next           = &idle_sched_class,
6172 #endif /* CONFIG_SCHED_VMS */
6173 }
```

Listing 2.8: fair.c

CONFIGURATION FLAGS

As you can see, the "insertion" is done conditionally, depending on whether the kernel was configured with `CONFIG_SCHED_VMS` flag.

The kernel flag itself is defined in the `Kconfig` file:

```
2334 menu "VMS scheduler (OSPJ)"
2335
2336 config SCHED_VMS
2337     bool "Virtual Machines Scheduling (VMS) policy"
2338     default y
2339 endmenu
2340
2341 }
```

Listing 2.9: /Kernel/arch/x86/Kconfig

This flag is also used to link project-related object files at the later build stage:

```
20 obj-$(CONFIG_SCHED_VMS) += ospj.o
21 obj-$(CONFIG_SCHED_VMS) += ospj_green_peace.o
```

Listing 2.10: /Kernel/kernel/sched/Makefile

CHANGING SCHEDULING POLICY TO SCHED_VMS

The purpose of `sched_setscheduler` syscall is to change the scheduling policy of given `pid` to `SCHED_VMS`.

The OSPJ policy defines a new property of process *share* that has to be passed to the syscall. For this purpose, the `sched_param` structure was also modified:

```
 7 struct sched_param {
 8     int sched_priority;
 9
10 #ifdef CONFIG_SCHED_VMS
11     unsigned int share;
12 #endif /* CONFIG_SCHED_VMS */
13 };
```

Listing 2.11: include/linux/sched.h

This modified syscall verifies following parameters before changing the policy:

1. Policy should be valid or recognized by the scheduler.
2. Priority of the policy should be valid. In `SCHED_VMS` case, valid priority is `0`.
3. User should have appropriate permission to change the scheduling policy.
4. Valid share (1 to 100) must be given.

If all the conditions are satisfied, the scheduling policy of process `pid` can be successfully changed to `SCHED_VMS`.

Design and implementation

QUEUES

The scheduling class performs periodic scheduling using 2 FIFO queues. There is a need for 2 queues to be in place for implementing a no-return policy for the tasks that have finished their allotted time (or, depending on the scheduling mode, those that yielded their priority before running out of time).

This way, each task which is determined not be allowed to return to the main `eligible_q` queue, are put into the `waiting_q` instead. The queues are eventually swapped, as outlined later in this section under **Queue operations** subsection.

TIME ALLOCATION

Each VCPU has a time slice within a superperiod computed according to the share passed in the `sched_setscheduler` system call. The time slice[3] itself is then calculated by calling the `ospj_calc_time_slice` function:

[3] One time slice is equal to 10ms

```
3890 __setscheduler_vms(struct rq *rq, struct task_struct *p, int policy,
         int share)
3891 {
3892     if(policy == SCHED_VMS && share > 0 && share <= 100) {
3893         p->sched_class = &ospj_sched_class;
3894         p->policy = policy;
3895         p->share = share;
3896         p->prio = 0;
3897
3898         ospj_calc_time_slice(p, share);
3899     ...
```

Listing 2.12: core.c

On each call of the `task_tick_ospj`, the task time slice is decremented. When the time slice reaches 0, it is replenished immediately and the task is forcibly rescheduled:

```
357 if (--p->ospj_time_slice == 0) {
358         ...
359         p->ospj_time_slice = p->ospj_assigned_time_slice;
360         set_tsk_need_resched(p);
361         yield_task_ospj(rq);
362         return;
363     }
```

Listing 2.13: ospj.c

TASK RESCHEDULING

When the task is rescheduled, it is moved from the `eligible_q` to the `waiting_q`:

```
174 static void requeue_task_ospj(struct rq *rq, struct task_struct *p)
175 {
176     list_move_tail(&p->ospj_list_node, rq->ospj.ptr_waiting_q);
177     dprintk(KERN_DEBUG, "[OSPJ] requeue(): del %d, n = %d\n",
178             p->pid, rq->ospj.nr_running );
179 }
```

Listing 2.14: ospj.c

Next task is picked by the `pick_next_task_ospj` function. Generally, during the non-idle period, the next task is picked from the `eligible_q`:

```
283 next = list_entry(rq->ospj.ptr_eligible_q->next, struct task_struct,
        ospj_list_node);
284 next->se.exec_start = rq->clock_task;
```

Listing 2.15: ospj.c

Otherwise, the *idle* period lasts until the end of the superperiod.

IDLE PERIOD

After the *eligible* tasks have been scheduled in current superperiod, it is checked if there is any time left. In case of success, the ownership of the idle task is taken in the `pick_next_task_ospj` (see `ospj.c:244`).

During this period, the power code is supposed to be run instead of simply halting the processor. During this phase, the OSPJ scheduler is receiving task tick calls while the idle task is executing.

After the last tick, the scheduling class of the idle task is set back to `idle_sched_class`. This is done in two steps:

1. `rq->ospj.idle_request` is set to `0` in `task_tick_ospj`.
2. Subsequent call to `pick_next_task_ospj` results in the actual assignment (`ospj.c:265`).

SUPERPERIOD REPETITION

Finally, after the superperiod has ended (i.e. `ospj.period_ticks == 0`), the superperiod is replenished and starts over again with queue swapping.

QUEUE OPERATIONS

One of the main operations performed with the queues is *swapping*, which happens in the `pick_next_task_ospj` function:

```
242 if (is_idle_period(rq)) {
243     ...
244     /* Swap Qs */
245     tmp = rq->ospj.ptr_waiting_q;
246     rq->ospj.ptr_waiting_q = rq->ospj.ptr_eligible_q;
247     rq->ospj.ptr_eligible_q = tmp;
248     d("Queues swaped\n");
249 }
```

Listing 2.16: ospj.c

As you might see above, it is not the queues itself that are compared, but their pointers. These pointer fields are initialized in the `init_ospj_rq` call:

```
63 ospj_rq->ptr_eligible_q = &ospj_rq->eligible_q;
64 ospj_rq->ptr_waiting_q = &ospj_rq->waiting_q;
65
66 ospj_rq->idle_request = 0;
```

```
67
68 INIT_LIST_HEAD(&ospj_rq->ospj_list_head);
69 INIT_LIST_HEAD(ospj_rq->ptr_eligible_q);
70 INIT_LIST_HEAD(ospj_rq->ptr_waiting_q);
```

Listing 2.17: ospj.c

QEMU threading model

As our scheduler is primarily targeted at scheduling QEMU threads, it is worthwhile to discuss its internals that are relevant to the project.

When QEMU is started with 2 virtual CPUs, it creates 3 permanent threads:

- 2 threads correspond to 2 VCPUs
- 1 thread is responsible for handling IO operations

Other threads are temporary/permanent and correspond to any of the following categories:

- Configuration dependent threads
- Worker threads
- Dedicated threads
- Reusable threads

Configuration dependent threads: It depends on the virtual machine features. E.g.: VNC thread, Gluster/Ceph.

Worker threads: Sometimes long-running computations simply hog the CPU and are difficult to break up into callbacks. In these cases dedicated worker threads are used to carefully move these tasks out of core QEMU.

Examples of worker thread users are `posix-aio-compat.c`, `ui/vnc-jobs-async.c`. Worker threads perform specialized tasks and do not execute guest code or process events.

Dedicated threads: Dedicated to perform only one task if the task is active, e.g. audio processing.

Reusable threads: `thread-pool.c` generates reusable threads.

In general threads will have the same CPU affinity as the main loop/iothread. There are QMP APIs to query the `tid`'s of some threads.

Also, it is possible to interact with QEMU via the QMP interface. As this interaction was not viable from kernel-space, this option was abandoned. Sample code is provided in **Appendix A**. Also, links to useful resources are provided at the end of the chapter.

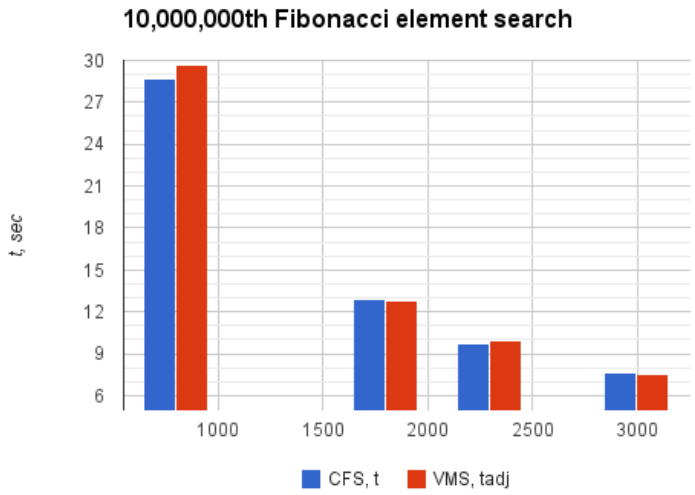
Scheduler evaluation

In order to test how well the scheduling subsystem performs, performance measurements were taken using the Fibonacci element calculation program.

The measurements were performed at each frequency step with the following configurations (5 times for each configuration to produce a statistically significant result):

- CFS scheduler
- VMS scheduler with 50% share

The results show that the calculation is done in very similar time, while VMS scheduler showing better results in 2 measurements and CFS - in 2 others (especially taking into account that the difference in terms of performance is comparable to the standard deviation within the series of a single measurement):



The table with the full measurement data is supplied separately.

Useful resources

SCHEDULER

- http://wiki.xen.org/wiki/Credit_Scheduler

- http://www.embedded.com/design/operating-systems/4204929/Real-Time-Linux-Scheduling-Part-1

QEMU

- http://wiki.qemu.org/QMP
- http://dachary.org/?p=1474
- https://rubygems.org/gems/qemu
- https://rubygems.org/gems/qemu-toolkit
- https://pypi.python.org/pypi/PyQemu/0.1
- http://blog.vmsplice.net/2011/03/qemu-internals-overall-architecture-and.html
- http://comments.gmane.org/gmane.comp.emulators.qemu/135139

Chapter 3

Power saving subsystem

Written by Lukas Braband & Marat Timergaliev

What was done by the energy subgroup during the project

Floor… In one of the first meetings it was pointed out that the project task requires some kind of “energy saving idle process”. This “energy saving idle process” would run for the amount of time where the current hyperperiod is already served completely. So during this timespan no resources (here CPU), agreed on in an SLA, are needed and therefore should be switched off “as deeply as possible”.

Power management. CPU sleeping states access

Power management implies dynamic frequency or voltage scaling on the certain devices based on performance needs. In the scope of power management in the project, the target goal was saving energy within the usage of CPU during the system idle time.

The finding of an appropriate way of “switching off” the CPU(s) turned out to be a more difficult and time consuming task than initially expected. Several ideas and approaches had come up. In this part of the documentation the investigated approaches are depicted.

SLEEPING STATES (C-STATES)

In order to save power, CPU has been enabled to support low-power mode states, or in other words c-states (sleeping state).

The basic idea of these modes is to cut the clock signal and power from idle units inside the CPU. The more units you stop (by cutting the clock), reduce the voltage or even completely shut down, more energy you save, but more time is required for the CPU to "wake up" and be again 100% operational.

The CPU power states C0–C3 are defined as follows:

- **C0** is the operating state.
- **C1** (often known as Halt) is a state where the processor is not executing instructions, but can return to an executing state essentially instantaneously. All ACPI-conformant processors must support this power state. Some processors, such as the Pentium 4, also support an `Enhanced C1 state` (`C1E` or Enhanced Halt State) for lower power consumption.
- **C2** (often known as Stop-Clock) is a state where the processor maintains all software-visible state, but may take longer to wake up. This processor state is optional.
- **C3** (often known as Sleep) is a state where the processor does not need to keep its cache coherent, but maintains other state. Some processors have variations on the C3 state (Deep Sleep, Deeper Sleep, etc.) that differ in how long it takes to wake the processor. This processor state is optional.

Additional states are defined by manufacturers for some processors. For example, Intel's Haswell platform has states up to C10, where it distinguishes core states and package states.

C-states are dynamically requested by software and are exposed through ACPI objects. C-states can be requested on a per-core basis. Software requests a C-state change in one of two ways, either by executing the HLT instruction or, for revision E, by reading from an IO specific address `CstateAddr`.

Since the access to c-states required installed ACPI module, here is the required configuration for linux in order to enable ACPI.

```
CONFIG_PM=y
CONFIG_ACPI=y
CONFIG_ACPI_AC=y
CONFIG_ACPI_BATTERY=y
CONFIG_ACPI_BUTTON=y
CONFIG_ACPI_FAN=y
CONFIG_ACPI_PROCESSOR=y
CONFIG_ACPI_THERMAL=y
CONFIG_ACPI_BLACKLIRG_YEAR=0
```

```
CONFIG_ACPI_EC=y
CONFIG_ACPI_POWER=y
CONFIG_ACPI_SYSTEM=y
```

We had written the basic class in order to read the `msr` register, specific for the certain processor. D18f4x118 recommended `msr` register to read, in order to send the CPU core to a c-state.

The result was not satisfactory, since the system's ACPI was still not enabled due to the absence of ACPI component on the motherboard.

We applied different techniques to resolve the absence of necessary component. Various reasons may exist of failing this task which are related to the different aspects such as lack of knowledge and proper documentation, CPU or motherboard related problems (may not support it), absence of necessary drivers for current CPU and etc.

Some work, that has been done in order to investigate possible solution to enable ACPI and CPU idle states was as follows:

- Update BIOS. It had to have a special configuration field to enable or disable ACPI, which was absent.
- Installation of `powertop` didn't show that the system has any cpu idle state.
- Research for possible reason of idle states absence
- CPU idle was not installed along with ACPI
- CPU idle driver missing. Attempt to find a proper driver for our processor failed.
- Research why ACPI doesn't work properly. Can be that in new kernel version acpi may require special actions, since it should work from `/proc/acpi` directory, which is deprecated in our kernel version, in favor of `/sys` directory. CPU idle configuration failed.

HLT execution cannot be verified since cpu idle states monitoring does not work.

CORE DISABLING

The first traditional approach of saving energy did not approve its efficiency, due of some reasons mentioned above, we decided to apply another way to manage the power consumption by the CPU.

An easy way, relatively from the first sight, to save energy is disabling cores. This technique can be applied on the unstressed systems, because of low demand of CPU, since the enabling a core takes a relatively long time (from 15ms to 45ms).

The observations from testing core disabling is provided in the table below. The benchmark duration - 10 minutes, and the load in each experiments - 100%.

Table. Benchmark on core disabling. (Duration for all experiments: 10 min).

Experiments	Cores	Frequency	Energy consumption	Response
1 (Baseline)	4	3 GHz	77194 Ws	0
2	3	3 Ghz	71693 Ws	15-45ms (unstable)
3	4	3 cores on 3GHz 1 core on 800MHz	73150 Ws	5 µs
4	4	4 cores on 800MHz	37966 Ws	5 µs

The discovery, from the conducted experiments, shows the minor difference in the indicators of energy saving, which plays lower priority role, rather than responsiveness of dynamic frequency scaling compared to core disabling. (See Table - Benchmark on core disabling).

As a result of the experiments, collaborative discussion and defining the priority, we decided to stick to dynamic frequency scaling.

Frequency Scaling (P-states)

P-states are operational performance states (states in which the processor is executing instructions, that is, running software) characterized by a unique frequency of operation for a CPU core. The P-state control interface supports dynamic P-state changes in up to 16 P-states called P-states 0 through 15 or P0 though P15. P0 is the highest power, highest performance P-state; each ascending P-state number represents a lower-power, lower-performance state.

In our project we use two levels of frequencies (800MHz and 3 GHz), which is justified by basic need and the easiness of measurements for energy saving. The other possible frequencies can be considered for the future development.

800 MHz frequency used during the idle time of system in order to save energy.

3 GHz frequency used for the purpose of high performance demand.

The switching to lower frequency happens, when the function of energy saving is called before starting the idle task, and lasts the duration of a given time.

When the time is over, the function of setting back the high frequency (3 GHz) is called.

Because scaling the frequency down to 800Mhz has almost the same effect (6) on energy consumption as disabling the core with the hlt command and the transition delay is strictly defined and very short ($<=5\mu s$ vs $\sim$40ms), we decided to concentrate on getting to work the frequency scaling. The progress in time contributed to that decision.

We discovered three kinds of ways on how the CPUs frequency can be manipulated. These are explained as follows:

SWITCHING FREQUENCY VIA SHELL

Starting with the topic, the following linux-command enables us to set the frequency of `cpu0` to 800Mhz:

```
cpupower -c 0 frequency-set -f 800000
```

SWITCHING FREQUENCY FROM C IN USER-SPACE

By looking how the upper command realizes this functionality we found the function

```
sysfs_set_frequency(unsigned int cpu, unsigned long target_frequency
    );
```

In the kernel-sourcecode in file `tools/power/cpupower/lib/sysfs.h` to change the frequency.

This function simply writes a value into `/sys/devices/system/cpu/cpu3/online` which then causes the frequency to change to the specified value.

Maximum transition frequency latency is $<=5\mu s$, which fits our use case very well.

SWITCHING FREQUENCY FROM C IN KERNEL-SPACE

Unfortunately the upper function could not be called from kernel-space (which is what we need when developing a new scheduler).

We ran several tests on getting to work the scaling of frequencies from the kernel-space.

One big step that had to be done was to fully separate the two developing issues into the `scheduling` development branch and the `power_only` development

branch. One clean Kernel had to be found where the frequency tests could be ran and continued to develope on. After Stephan Bauroth had adapted the build-system, it was possible to directly run the current developments of the `power_only`-branch from the git-repository on the deneb-server without merging it with the master branch (the current changes on the OSPJ-Scheduler itself).

By doing this, the bug sources for each development part could be embanked. Before that there were some OSPJ-Kernel-versions which ran very unstable (random freezes of the server within minutes or hours) - and nobody knew whose code was responsible for the unwanted behaviour.

After looking at the cpufreq-driver code the plan was to copy the current `cpufreq_policy` to two new ones (`policy_min` and `policy_max`). Under several other fields the cpufreq_policy-struct contains the integer variables `min`, `max` and `cur` which contain the minimum-, maximum and current frequency of that policy.

The new policies should look like this:

policy_min:

- min = 800Mhz
- max = 800Mhz
- cur = 800Mhz

and

policy_max:

- min = 3000Mhz
- max = 3000Mhz
- cur = 3000Mhz

Chapter 4

Setup

Written by Andrii Berezovskyi

> Just as a preface, I would like to ask you: do not try working on kernel under Windows, heaven forbid! Otherwise, you'll immediately run into an issue while cloning the kernel repository, namely with checking out the `aux.c` file[1].

Overview

The development of the project was organized as follows: the code was kept under Git, continuous integration was performed using Jenkins and custom scripts on the test server. Custom-built kernels were tested in QEMU, as well using serial terminal connections to the test server and `printk` statements that were output to the syslog.

Development

Git & Gitlab

The Git repository follows these guidelines: compiling and QEMU-passing code is committed to the `master` branch and Jenkins checks the master branch regularly and upon a build success it merges `master` branch to `deploy`. The deploy branch itself is marked as protected in Gitlab.

[1] And, hopefully, many others :)

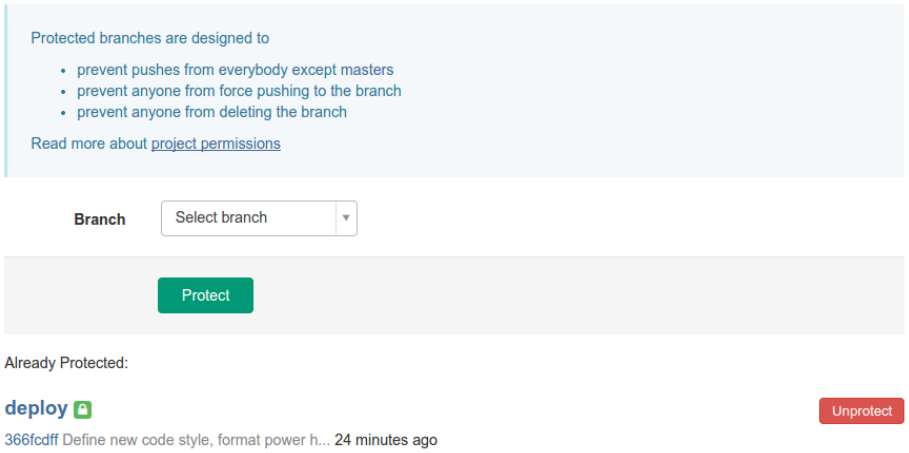


Figure 4.1: Screenshot of the branch protection interface

JENKINS CI

CI server runs on the local network on the server with hostname `zambezi`. The Jenkins installation runs on port `8080` so you can access it at http://zambezi:8080 when you're connected to the local network (or when you're logged in to the `poolgate` or `deneb`). If you're outside of the network, set up your connection as described in the forum post message #16.

How does the CI process happen and do I have to care about it?

The CI process it triggered automatically whenever you push anything to the master branch on Gitlab. Thus, it's better if you merge your changes to the master branch more often than if you merge 20 commits at the same time (I hope it would trigger 20 separate builds but we still need to see how it goes).

The Jenkins server would check out the recent copy of the source and try to build it. If the build succeeds, it's pushed to the `deploy` branch (for now the keys are attached to my account so you'll see them as my commits, but we'll resolve that later if we'll get some time allocated for that).

On `deneb`, there is a cron task (`/etc/crontab`) that pulls the `deploy` branch every 15 minutes. If the actual kernel release (`uname -r`) of the server doesn't match the latest change in the repository, then the new kernel is built and installed on `deneb`. If the installation was successful, all users with open terminal sessions are warned that server will reboot in 5 minutes and then the server reboots with the new kernel.

Okay, shall I care at all about that?

- If you expect to see some changes, but they're not working (or if you

see some changes you didn't expect :P), check the current kernel release (`uname -r`).
- If you plan to run long-running tasks in the background, beware that reboot may interrupt them.

Testing

QEMU

In order to run the QEMU emulator, you need to do the following steps:

1. Compile the `sched_test` in the `Userspace` folder
2. Run the `rebuild-initramfs.sh` script in QEMU folder
3. If you don't have KVM or you want a single-core machine, run `cmd-single.sh` script.
4. If you have KVM and you want a 4-core machine, run `cmd.sh` script (you might need `sudo` for it, though).

General advice is not to execute `rebuild-initramfs.sh` with `sudo` privileges as it will change ownership of the generated files.

Access to the test server

In order to access `deneb` from the global network (inlcuding eduroam[2]), you must connect to the gateway first:

```
ssh <$tubit_user>@poolgate.kbs.tu-berlin.de -p 20122
```

The SSH will prompt for a password, use your tubIT password. If this action fails, you shall write a letter to our administrator[3] and ask for an account on `poolgate`.

From there, you shall be able to log in to `deneb`:

```
ssh your_deneb_login@deneb
```

[2]Please note that at TEL-building the DNS & SSH connections from guest network are blocked.
[3]Matthias Druve <matthias.druve@tu-berlin.de>

Serial login and boot debugging

In case of kernel panic, the stacktrace is printed on the screen in most cases. Unfortunately, the stack trace can span multiple screens and vitally important information gets lost.

To overcome this issue, we set up the login to the server over the serial tty. We also set up the GRUB over serial later as well. This way, by connecting the serial-to-USB converter[4] (through the null modem

- **beware, it must be crossing!**[5]) to the Jenkins server USB port and opening a `minicom` inside `screen` on Jenkins server, we always had the boot and crash output available upon SSHing into the Jenkins server (`zambezi` in our case).

The login capability was enabled by uncommenting a line in the `/etc/inittab` file:

```
# SERIAL CONSOLES
s0:12345:respawn:/sbin/agetty -L 115200 ttyS0 vt100
```

The GRUB over serial output was enabled via adding the following lines to `/etc/default/grub`:

```
GRUB_TERMINAL_INPUT="console serial"
GRUB_TERMINAL_OUTPUT="console serial"
GRUB_SERIAL_COMMAND="serial --speed=115200 --unit=0 --word=8 --
    parity=no --stop=1"
```

You can read more on this and other debug-related questions in Arch Wiki.

Useful information

Commands and advanced SSH configuration

Speaking of logins, there are two interesting commands:

```
smarx721@bs200 ~ $ w
 19:43:01 up 14 days,  9:46, 15 users,  load average: 0.00, 0.01,
     0.05
USER     TTY        LOGIN`   IDLE   JCPU   PCPU WHAT
smarx721 pts/0     19:42    0.00s  0.00s  0.00s w
anton.e  pts/2     18:29   14:05   3.61s  3.61s ssh root@snb-ep
```

[4] We used this Logilink

[5] A gender changer similar to this one didn't work for us!

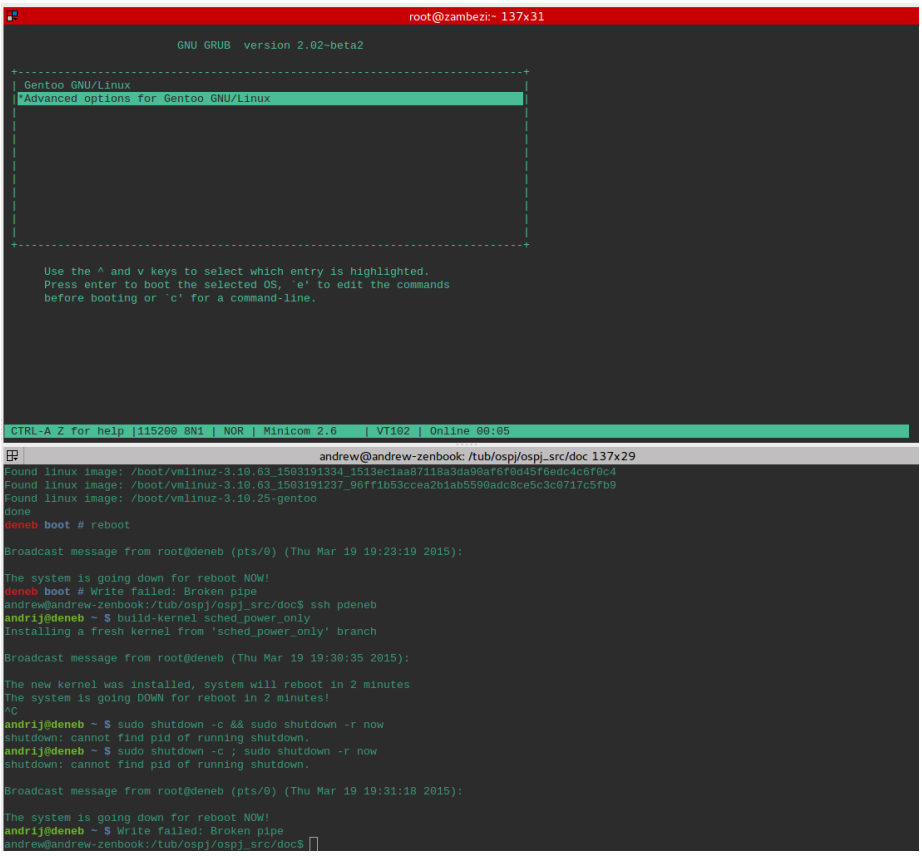


Figure 4.2: GRUB over serial

```
armandza pts/3     17:54    9:28   0.11s  0.09s ssh armand@deneb
...
```

and

```
andrij@deneb ~ $ write andrij

Message from andrij@deneb on pts/0 at 19:44 ...
```

As you have to log in regularly, it's sad to enter your password all the time. Instead of authentication with a password, you can do it with public/private keys.

In order to upload keys, use the command

```
ssh-copy-id <$tubit_user>@poolgate.kbs.tu-berlin.de
```

Good. Now we don't have to enter the password. But we still have to type this to log in:

```
ssh <$tubit_user>@poolgate.kbs.tu-berlin.de
```

There two good solutions: use "Ctrt + R" or use SSH config. Let's go for the second:

```
andrew@andrew-zenbook:~$ cat ~/.ssh/config
Host poolgate
    HostName poolgate.kbs.tu-berlin.de
    Port 20122
    User $<tubit_user>
```

Now you can login to the gateway with "`ssh poolgate`".

Cool, but not enough. We'd like to ssh into the deneb right away, huh?

```
andrew@andrew-zenbook:~$ cat ~/.ssh/config
Host poolgate
    HostName poolgate.kbs.tu-berlin.de
    Port 20122
    User <$tubit_user>

Host deneb
    User <$deneb_login>
    ProxyCommand ssh -q poolgate nc -q0 %h 22
```

Now it works like this:

```
andrew@andrew-zenbook:~$ hostname
andrew-zenbook
andrew@andrew-zenbook:~$ ssh deneb
```

```
andrij@deneb ~ $ hostname
deneb
```

Only catch is that on the poolgate there should be a private key such that it's public key is in your deneb's `authorized_keys` list. You can run `ssh-keygen` on `poolgate` and then do the same `ssh-copy-id` operation!

UNTRACKED FILES

`/etc/default/grub` contains important information, mainly the kernel boot /command and serial debugging support.

`/var/OSP/deploy/script/build.sh` is a script to install the latest kernel version from the repository (by default, from `deploy` branch). ** Must be executed by user gitlab only!**

`/usr/bin/build-kernel` is a script that invokes the script above (with a proper user). Email[6] is sent to Fleep on completion. Usage:

```
build-kernel <branch>

    e.g.

    build-kernel
    build-kernel deploy
    build-kernel sched_power
```

`/usr/bin/rm-kernel` removes the kernel that matches the given string. Intended to be used with a commit hash. Email is sent to Fleep on success.

GIT ALIASES

In order to maintain productive Git workflow, I suggest you to take a look at my Git aliases:

```
[push]
  default = current ; do not touch the other branches
[alias]
  p = pull
  s = status
  b = branch
  c = commit
  ca = commit -a
  add = add --ignore-removal
  all = add -A :/
  co = checkout
  sync = !git pull -r && git push && git push --tags
  ri = rebase -i
```

[6] Letters are sent out using the `nullmailer` program that is forwarding the letters via Mailgun.

```
14    pusha = "!git add -A ; git commit -a ; git pull -r; git push"
15    lg = log --graph --abbrev-commit --decorate --date=relative --format=
          format:'%C(bold blue)%h%C(reset) - %C(bold green)(%ar)%C(reset) %
          C(white)%s%C(reset) %C(dim white)- %an%C(reset)%C(bold yellow)%d%
          C(reset)' --all
16  [branch]
17    autosetuprebase = always
```

Listing 4.1: .gitconfig

Userspace tools

ospj-setsched

Build instructions

Run `make` to build the tool.

How to run it?

From the tool help output:

```
Usage: ospj-setsched OPTIONS

Option              Meaning
-h                  Show this message
-v                  Show program version
-d                  Verbose output
-s <#pid>           Set scheduling policy for the process #pid
-r <#pid>           Read scheduling policy for the process #pid
-p <#policy> or     Supply #policy (see sched.h) for the flag -s
-p SCHED_(OTHER|FIFO|RR|BATCH|IDLE|VMS)
-u <#per>           VMS usage share #per in range [1,100]
-o <#prio>          Scheduling priority prio in range [1,99]
                    (only for SCHED_FIFO and SCHED_RR)
```

How to debug it?

Before running the `ospj-setsched`, execute

```
ulimit -s unlimited
ulimit -c unlimited
```

Run the program and right after it finishes, execute the following if there was an error:

```
echo $?
```

To get the exit code. Then look up this code in the source.

If you got a segfault, run

```
gdb ospj-setsched core
```

In the gdb prompt, run `'bt'`. It'll give you something similar to the stacktrace you might have been used to from Java.

SAMPLE OUTPUT

```
andrew$ ./ospj-setsched -s 31792 -p SCHED_IDLE
Policy SCHED_IDLE was succesfully set for pid 31792
andrew$ ./ospj-setsched -s 31792 -p SCHED_VMS
ospj-setsched was built without SCHED_VMS support
Run 'ospj-setsched -v' to verify the program version

Usage: ospj-setsched OPTIONS
...
andrew$ ./ospj-setsched -s 31792 -p SCHED_FIFO
SCHED_FIFO and SCHED_RR require a valid priority

Usage: ospj-setsched OPTIONS
...
andrew$ ./ospj-setsched -s 31792 -p SCHED_FIFO -o 80
Failed to set policy SCHED_FIFO for pid 31792: you must be root to
    execute this action.
Aborted (core dumped)
```

sched_test

This program is intended to be a lightweight test for the scheduler, allowing easier debugging. In the most primitive setting, it can execute from a single thread or perform multiple forking.

In order to build it, run `'make all'` in the `sched_test` folder and run the `sched_test/demo/demo` executable.

You must build the demo program manually before you rebuild the QEMU appliance. Otherwise you risk getting compilation errors or having stale executable.

NB! If you want the tool to be compiled with the VMS scheduler enabled, you have to perform the build on the `deneb` server or on other machine that performed installation of the headers for the custom kernel.

Chapter 5

Measurement and Evaluation

Written by Stephan Bauroth

Measurement Scripts

The scripts to measure the energy consumption on `deneb` connect to the measurement device `Expert PDU 8001`.

The device supports measuring two channels, we have `deneb` fixed on channel one. To actually start a measurement, the server needs to be started on your machine and `energy`, a machine within the local network of `deneb`, needs to be reachable from your local area network.

First, the server script is started, then energy consumption up until now can be read with the client script. For convenience, `measure.sh` will measure the consumed energy over a given timespan (default 10 min.) automatically.

Server Script

The server script periodically queries the device for information via SNMP on already consumed power and `echo`s the return value into `/tmp/energy.channel`.

```
1 case $1 in
2     "1" | "2") channel=$1;;
3     *)
4         echo "usage: $0 <channel>"
```

```
5           exit
6   esac
7
8   interval=0.43
9   trap cleanup EXIT
10  cleanup()
11  {
12      echo "Quit."
13      rm -f /tmp/energy.$channel
14      exit
15  }
16
17  wh=`snmpget -c public -v 1 energy 1.3.6.1.4.1.28507.2.1.3.1.3.$channel
        | awk '{ print $4 }'`
18  wh_old=$wh
19  echo -n "Calibrating..."
20  while sleep $interval; do
21      wh=`snmpget -c public -v 1 energy 1.3.6.1.4.1.28507.2.1.3.1.3.
            $channel | awk '{ print $4 }'`
22      if [ $wh -ne $wh_old ]; then
23          wh_old=$wh
24          break
25      fi
26      wh_old=$wh
27  done
28  echo " done."
29  told=`date +"%s%N"`
30  sumwdiff=$(($wh*3600000000000))
31  xdiff=""
32  while sleep $interval; do
33      w=`snmpget -c public -v 1 energy 1.3.6.1.4.1.28507.2.1.3.1.4.
            $channel | awk '{ print $4 }'`
34      wh=`snmpget -c public -v 1 energy 1.3.6.1.4.1.28507.2.1.3.1.3.
            $channel | awk '{ print $4 }'`
35      t=`date +"%s%N"`
36      dt=$(($t-$told))
37      told=$t
38      sumwdiff=$(($sumwdiff + $w*$dt))
39      if [ $wh -ne $wh_old ]; then
40          xdiff=" DIFF=$((($sumwdiff-$wh*3600000000000)/1000000000))"
41          sumwdiff=$(($wh*3600000000000))
42          wh_old=$wh
43          echo "dt=$dt W=$w Ws=$(($sumwdiff/1000000000)) Ws_ref=$(($wh
                *3600))$xdiff"
44      fi
45      echo $(($sumwdiff/1000000000)) > /tmp/energy.$channel
46      xdiff=""
47  done
```

Listing 5.1: energie-mess-server.sh

Client Script

The cleint script queries the file `/tmp/energy.channel` and outputs the result to `STDOUT`.

```
case $1 in
    "1" | "2") channel=$1;;
    *)
        echo "usage: $0 <channel>"
        exit
esac
cat /tmp/energy.$channel
```

Listing 5.2: energie-mess-client.sh

Measurement Script

The measurement script by Lukas Braband reads the already consumed energy at starting time, waits for a given timespan, reads again, calculates the difference and outputs everything nicely to `STDOUT`. The server script is supposed to be already running when calling this.

```
timespan=600    #in seconds 600=10min
# ============================================
# created by Lukas Braband
echo Time after which measurement will start:
date

timeA=`./energie-mess-client.sh 1`
sleep $timespan
timeB=`./energie-mess-client.sh 1`

diff=$(($timeB-$timeA))

echo "starting point:   " $timeA
echo "end point:        " $timeB
echo Consumption in Ws within the last $timespan seconds: $diff Ws
```

Listing 5.3: measure.sh

Evaluation

For now, since the power subsystem is not stable, the only measureable adavtage of the scheduler is the strict shares. So, a CPU-intense task (`yes > /dev/null`) is measured while run in QEMU. Setup A runs QEMU scheduled by CFS, Setup B is scheduling the QEMU process with VMS and a share of

30%. As expected, the power consu,ption is reduced by the built-in routines of Linux because the processor is idle.

Table. Benchmark on limiting a tasks share (Duration for all experiments: 10 min).

Setup	Energy consumption
A	57051
B	47989

Chapter 6

Appendices

Appendix A. Interacting with QEMU via QMP

```
source "https://rubygems.org"

gem 'qemu', '~> 0.4'
```

Listing 6.1: Gemfile

```
#!/usr/bin/env ruby

require 'qemu'
include QEMU

d = Daemon.new

puts 'Daemon constructed'
```

Listing 6.2: main.rb

Appendix B. Untracked files

```
# Copyright 1999-2013 Gentoo Foundation
# Distributed under the terms of the GNU General Public License v2
# $Header: /var/cvsroot/gentoo-x86/sys-boot/grub/files/grub.default-2,v
    1.4 2013/09/21 18:10:55 floppym Exp $
#
```

```
5  # To populate all changes in this file you need to regenerate your
6  # grub configuration file afterwards:
7  #     'grub2-mkconfig -o /boot/grub/grub.cfg'
8  #
9  # See the grub info page for documentation on possible variables and
10 # their associated values.
11
12 GRUB_DISTRIBUTOR="Gentoo"
13
14 GRUB_DEFAULT=saved
15 GRUB_HIDDEN_TIMEOUT=0
16 GRUB_HIDDEN_TIMEOUT_QUIET=true
17 GRUB_TIMEOUT=5
18
19 # Append parameters to the linux kernel command line
20 # GRUB_CMDLINE_LINUX=""
21
22 # Append parameters to the linux kernel command line for non-recovery
       entries
23 #GRUB_CMDLINE_LINUX_DEFAULT="oops=panic panic=-1 CONFIG_DEBUG_KERNEL
       CONFIG_DEBUG_STACKOVERFLOW CONFIG_DEBUG_STACK_USAGE"
24 GRUB_CMDLINE_LINUX_DEFAULT="oops=panic CONFIG_DEBUG_KERNEL
       CONFIG_DEBUG_STACKOVERFLOW CONFIG_DEBUG_STACK_USAGE console=tty0
       console=ttyS0,115200n8 maxcpus=1"
25
26 # Uncomment to disable graphical terminal (grub-pc only)
27 #GRUB_TERMINAL=console
28 GRUB_TERMINAL_INPUT="console serial"
29 GRUB_TERMINAL_OUTPUT="console serial"
30 GRUB_SERIAL_COMMAND="serial --speed=115200 --unit=0 --word=8 --parity=
       no --stop=1"
31
32
33 # The resolution used on graphical terminal.
34 # Note that you can use only modes which your graphic card supports via
        VBE.
35 # You can see them in real GRUB with the command `vbeinfo'.
36 GRUB_GFXMODE=1280x1280x16
37
38 #GRUB_GFXPAYLOAD_LINUX=keep
39
40 # Path to theme spec txt file.
41 # The starfield is by default provided with use truetype.
42 # NOTE: when enabling custom theme, ensure you have required font/etc.
43 #GRUB_THEME="/boot/grub/themes/starfield/theme.txt"
44
45 # Background image used on graphical terminal.
46 # Can be in various bitmap formats.
47 #GRUB_BACKGROUND="/boot/grub/mybackground.png"
48
49 # Uncomment if you don't want GRUB to pass "root=UUID=xxx" parameter to
        kernel
50 #GRUB_DISABLE_LINUX_UUID=true
51
52 # Uncomment to disable generation of recovery mode menu entries
53 GRUB_DISABLE_RECOVERY=true
```

Listing 6.3: /etc/default/grub

```
#!/bin/bash
FOLDER=/var/OSP/deploy
#FOLDER=/var/OSP
LOG=$FOLDER/log/$(date +%y%m%d-%H%M).log

LOCKFILE=$FOLDER/build.lock

rmlock() {
    lockfile-remove -l $LOCKFILE
}

lockfile-create -r 0 -l $LOCKFILE || { echo "Lock is taken, skipping the build" ; exit 99;}

cd $FOLDER/operating-system-project
#cd $FOLDER/gitRepo
branch=${1-deploy}

git fetch --all > "$LOG" 2>&1 || { echo 'git fetch failed' ; rmlock; exit 1; }
git checkout $branch --force > "$LOG" 2>&1 || { echo 'git checkout failed' ; rmlock; exit 1; }
PRE=$(uname -r)
PRE=${PRE: -40}
PREV=${PRE:0:8}
git pull origin $branch >> "$LOG" 2>&1
NEXT=$(git rev-parse HEAD)
NEXT_CMP=${NEXT:0:8}
if [ "$PREV" == "$NEXT_CMP" ]
then
 echo "kernel already up-to-date (rev $PREV)" >> "$LOG" ; rmlock; exit 2;
fi

INSTALLED=$(ls /boot/vmlinuz* | grep -c "$NEXT")

if [ "$INSTALLED" -gt 0 ]
then
   echo -e "kernel $NEXT was already installed (and $NEXT_CMP not equal to $PREV)\nhttps://gitlab.tubit.tu-berlin.de/mirko/operating-system-project/commit/$NEXT_CMP\nhttps://gitlab.tubit.tu-berlin.de/mirko/operating-system-project/commit/$PREV" | tee -a  "$LOG" ; rmlock; exit 3;
fi


cd Kernel
make LOCALVERSION="_$(date +%y%m%d%H%M)_$NEXT" ARCH=x86_64 -j4 >> "$LOG" 2>&1 || { echo 'kernel compilation failed' | tee -a "$LOG"; rmlock; exit 4; }
echo "Kernel was built on $(date)" | tee -a "$LOG"

sudo make -j4 ARCH=x86_64 headers_install >> "$LOG" 2>&1 || { echo 'header installation failed' | tee -a "$LOG"; rmlock; exit 5; }
sudo make -j4 ARCH=x86_64 modules_install >> "$LOG" 2>&1 || { echo 'module installation failed' | tee -a "$LOG"; rmlock; exit 6; }
sudo make -j4 ARCH=x86_64 install >> "$LOG" 2>&1 || { echo 'kernel
```

```
    installation failed' | tee -a "$LOG"; rmlock; exit 7; }
46
47  sudo grub2-mkconfig -o /boot/grub/grub.cfg >> "$LOG" 2>&1
48
49  cd $FOLDER/operating-system-project/Userspace/ospj-setsched
50  if [ -e Makefile ]; then
51      make >> "$LOG" 2>&1 || { echo 'compiling ospj-setsched failed' |
            tee -a "$LOG"; rmlock; exit 8; }
52      sudo make install >> "$LOG" 2>&1 || { echo 'installing ospj-
            setsched failed' | tee -a "$LOG"; rmlock; exit 9; }
53  fi
54
55  echo "New kernel $NEXT was successfully installed on $(date)" | tee -a
        "$LOG"
56
57  rmlock
58
59  exit 0;
```

Listing 6.4: /var/OSP/deploy/script/build.sh

```
1   #!/bin/bash
2
3   mail_fleep() {
4   echo -e "$2"
5   echo -e "$2" | mail -s "$1" conv.36nrjbz27sfu41@fleep.io
6   }
7
8   branch=${1-deploy}
9
10  mail_fleep "Manual kernel build started" "Installing a fresh kernel
        from '$branch' branch"
11
12  build_output=$(sudo -u gitlab /var/OSP/deploy/script/build.sh $branch)
13  build_status=$?
14
15  mail_fleep "Manual kernel build finished" "The build has finished with
        status '$build_status':\n$build_output"
```

Listing 6.5: /usr/bin/build-kernel

```
1   #!/bin/bash
2
3   mail_fleep() {
4   echo -e "$2"
5   echo -e "$2" | mail -s "$1" conv.36nrjbz27sfu41@fleep.io
6   }
7
8   sudo rm -i /boot/*$1*
9
10  mail_fleep "Kernel removed" "Kernel build ${1:0:8} was removed from the
        server"
```

Listing 6.6: /usr/bin/rm-kernel

```
redefinition of the experiment by mohammad/group:
4 cores @3Ghz vs
3 cores @3Ghz, 1 core @800Mhz vs
3 cores @3Ghz, 1 core disabled

##################################################

Results:
4 cores @3Ghz: 79684 Ws
result is an average value. Executed experiment 3 times (79986 Ws,
    79567 Ws, 79500 Ws)
web-interface showed active power of: 133W

3 cores @3Ghz, 1 core @800Mhz: 73150 Ws
executed: 2x, (73149 Ws, 73150 Ws)
web-interface showed active power of: 121-122W

3 cores @3Ghz, 1 core disabled: 71693 Ws
executed: 2x, (71690 Ws, 71695 Ws)
web-interface showed active power of: 119-120W

(done in Task 2117)



############## older Task: 2049 ####################
Measurement results:
all 4 cores 800Mhz, 4*100% cpu-load:
energy-consumption-counter:
before: 283095371 Ws
after: 283133337 Ws
diff: 37966 Ws
benchmark-duration: 10min

all 4 cores 3Ghz, 4*100% cpu-load:
before: 283158544 Ws
after: 283235738 Ws
diff: 77194 Ws
benchmark-duration: 10min
```

Listing 6.7: energy-experiments.txt