NTNU

Norwegian University of Science and Technology

A Multicore-aware Deadline-driven Real-Time Scheduler for the Linux Kernel

Henrik Austad

Master of Science in Engineering Cybernetics
Submission date: June 2009
Supervisor: Sverre Hendseth, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

Implement a multicore aware deadline driven real-time scheduler for the Linux kernel;
a) Determine type of algorithm that is best suited in a multi-core system (heterogeneous CPUs) with respect to CPU cache, race conditions and overall system utilization.
b) Understand and modify the kernel source, configuration and compilation system.
c) Discuss the proposed scheduling class with the kernel community to make sure it is a desired project.
d) Track the mailing list, filter relevant topics, submit patches and perform patch review when necessary. All of this to gain practical knowledge about kernel development and maintenance.
e) Identify relevant subsystems and create test-modules to verify behavior.
f) Create userland interface to the class via syscalls and sysfs entries.
g) Implement the scheduling class and connect it with the relevant subsystems.
h) Design and implement a small test for the new scheduling class.
i) Have the new scheduling class included in the kernel repository, either mainline Linux or the real-time preemption patch series.

Assignment given: 12. January 2009
Supervisor: Sverre Hendseth, ITK

Norwegian University of Science and Technology

Department of Engineering Cybernetics

TTK4900

Dedicated Computer Systems,

Master Thesis

Rate-based Real-Time Scheduler for the Linux kernel on multicore systems

Author:
Henrik Austad

Supervisor:
Sverre Hendseth

Preface

This paper is written as fulfillment of the Master of Engineering degree, completed during the 10th semester at the Norwegian University Science and Technology (NTNU), Department of Engineering Cybernetics. In this paper, a multi-core aware, rate-based scheduler for real-time tasks is implemented in the Linux kernel. Due to problems faced during the integration process, a fully functional scheduler has not been implemented.

I acknowledge the invaluable assistance offered by associate professor Sverre Hendseth at the Department of Engineering Cybernetics, Norwegian University of Science and Technology. I would also like to show my gratitude to Peter Zijlstra for helping me with technical issues. I also thank Bill Huey's patience for the countless hours spent listening to my ideas and opinions about scheduling in the kernel. Without his guidance, I would have been truly lost. Finally, I thank my brother Jon, for helping me realize the difference between a comma and a period.

Henrik Austad

Trondheim, June 8, 2009

Abstract

This paper describes the implementation of a new scheduling class in the Linux kernel. The required subsystems are covered and tested in detail.

The scheduling class will handle real-time tasks that cannot miss a deadline (hard real-time) and is planned to be placed on top of the RT Preemption Patch. The goal is to extend the Linux kernel with a deadline-driven scheduler.

The scheduling algorithm is the PD^2 algorithm developed by Baruah et. al [22]. Some optimizations have been performed, most notably the way the subjob values are being calculated. Furthermore, the release of a subjob has been updated to be an O(1) operation. For reasons to be discussed later, the implementation failed, but from the failure arose a better understanding. A modified MLLF [20] scheduler has been proposed, one which is designed specifically to counter many of the problems faced during this project.

Contents

Part I

Exordium

Chapter 1

Introduction and Motivation

"A computer terminal is not some clunky old television with a typewriter in front of it. It is an interface where the mind and body can connect with the universe and move bits of it about."
— Douglas Adams

This project is about implementing a multi-core, rate based scheduler for real-time tasks in the Linux kernel. The pfair algorithm is given particular attention. The theory is "state of the art", and at the time of this writing, no operating system officially supports a multi-core, rate based scheduler.

The major tasks in this project has been implementing the algorithm, including it into the kernel and utilizing the various subsystems. Furthermore, the theory must be mastered, as must the somewhat daunting task of interfacing with the kernel community via the Linux Kernel Mailing List (LKML).

The report is split into 4 distinct parts. Part I contains introduction and background material. The background covers the current scheduling system in the Linux kernel, scheduling theory and a brief look at related work. Part II describes design and implementation of the pfair scheduler. This is where most of the actual work is described. Then, part III evaluates the scheduler and the outcome. Part IV contains the Appendix with various elements not strictly related to the project, but are needed nevertheless.

This report is intended as a document to show what was done, why and how, but is by no means exhaustive. For a complete understanding of the kernel internals, the only way is to work with the kernel code. All relevant code is attached for the readers perusal.

This chapter introduces the project, and explains the primary motivation for writing a whole new scheduling class instead of "just" extending the current scheduler. Then it outlines the project itself, with all the practicalities, dead ends and planning. It also lists the required skills needed to successfully complete a task of this magnitude.

1.1 Motivation

The pfair scheduler is a near optimal rate based scheduler. An optimal scheduler is an instance of a given algorithm that performs at the peak of the theoretically possible. For instance, EDF is optimal because deadline driven scheduling can achieve 100% utilization on a single core machine. Rate Monotonic scheduling is also optimal, even though it is bounded to approximately 68%, because the static priority algorithm is bounded to this level.

The current real-time scheduler in the Linux kernel is a static priority, partitioned scheduler. It will never change the priority of a task, but it may move a task from one processor to another to balance the load. The idea is to give the application programmer complete control over the priorities. This works to some extent, but have several serious drawbacks:

- The application developer must decide the priorities for all tasks in order to successfully meet the deadlines. When the number of tasks grow, this can become an unmanageable task.
- Very dynamic systems, or systems composed of several applications, quickly becomes too complex for the most advanced tool to analyze offline.
- By using a partitioned scheduler, most of the global scheduler related information will be unreachable. To perform globally "sound" scheduling decisions, the scheduler must itself be global (or have some serious load balancing).
- A partitioned scheduler must use approximation when accepting jobs, and hence, by definition, cannot ever hope to perform as well as a scheduler than takes the entire system into consideration.
- In order to allow all tasks execute as quickly as possible, the load across the CPUs must be kept as even as possible. A partitioned approach has to resolve to load balancing, adding extra overhead and unpredictability. A global scheduler can avoid this issue all together.

- It has no notion of passed execution time and remaining time, and this results in "blindfolded" task-switches — the scheduler preempt tasks hoping it has done the right thing.

For some applications, this scheduler works fine. However, as the computers keep evolving, and more and more real-time tasks are being handled by the computers, the analysis grow in complexity. The current scheduler cannot scale to very large, complex schedules. To fully utilize the hardware, the scheduler must accept jobs until the processor can be kept busy at all time, while still ensuring that all deadlines are met. All in all, the kernel must provide greater *expressiveness* to the application programmer. The developer must be given the ability to express the needs of the application in as a precise manner as possible. By allowing the deadlines to be passed directly and used internally, much greater expressiveness can be achieved than by using simple priorities.

Chapter 3 discusses the pfair scheduler, a multi-core rate based scheduler that is near-optimal[1]. To justify the large effort required to implement a new scheduler in the kernel, it must help, or ease:

- Scenarios too complicated for application developers to deal with manually, i.e. if a system has a large number of tasks.
- Scenarios that are too dynamic to deal with either manually or by a single application, e.g. when more than one real-time application run on the computer.
- Accept new tasks more broadly than the current static priority, partitioned scheduler, and guarantee the overall system being deterministic and meet all deadlines.
- Induces less overhead and computational complexity than a partitioned scheduler with an advanced load balancer.
- Provide the application programmer with greater *expressiveness* when it comes to communicating the needs of the task to the scheduler.
- Give the scheduler a simple way of enforcing bandwidth control over tasks.

The PD^2 scheduler can do a lot of this, but not all. It is neither truly optimal, nor able to solve all the bin-packing problems. It strives to, but unless the timeslices shrinks to infinitesimal size, the scheduling decisions take zero time

[1]Almost optimal, but not quite due to some design choices that minimized computational overhead, but adds to the "error"

and the time needed to switch tasks is non-existent, it will never be truly optimal. Because of the global runqueue needed by the algorithm, it can never scale linearly to an ever increasing number of CPU cores. In Sec. 3.4, the scheduler is presented more formally. Although it is not optimal, it is very close in many areas. This makes it a better candidate than a lot of other partitioned algorithms which fail to do this. Another "neat" feature of pfair, is that it will reduce jitter dramatically, and it can act as "computational timers", by this we mean that not only will pfair meet the deadlines, but the jobs will also finish close to the actual deadline, allowing for "just in time computation".

An old "mantra" when it comes to real-time engineering, is to just add more and faster hardware if the system experienced deadline misses[2]. As long as Moore's Law applies, this works. But, as the hardware manufacturers are moving from improving single-core performance to "just" adding more cores, this rule of thumb breaks down. In addition, as we continue to see more and more embedded devices doing tasks previously performed by big systems, or humans, using *less* power becomes very important. If the scheduler is multi-core optimal, it means that the system can be *minimized* and still meet the deadlines and provide the required QoS.

In this project, we look at hard real-time. By doing that, we do not allow for any tasks to miss a deadline. This is next to impossible to test at run-time, and the added overhead for doing so, makes this non-feasible. Instead, *Resource Reservation* is used. This is an effective technique for ensuring that tasks do not use more resources than the system can provide, and still be able to utilize as much of the available system as possible. It works by reserving a certain amount of resources as an acceptance test. If the system can provide the requested amount, the task is accepted. Otherwise it is turned down. Once accepted, the system provide assurance to the task that it will be given the requested amount.

As a final note, by implementing a scheduling policy that handles time as the fundamental key to order the tasks, the system frequency[3] becomes less of an issue. If the scheduler can set a lower frequency limit, the system can scale down to this limit and still meet all deadlines. For small and embedded systems, this is a desirable goal. However, this is not part of this project; it is just a motivating factor.

[2]If tweaking the priorities proved unsuccessful for various reasons.

[3]CPU frequency, along with System Bus Frequency and Memory frequency, i.e. overall speed.

1.2 Problem definition

The kernel needs a scheduler to handle tasks with fixed deadlines without forcing the tasks to readjust their priorities dynamically at run-time. It is desirable to allow a task to reweigh[4] its parameters dynamically in order to facilitate non-fixed deadlines. All of this must be done in a way that preserves predictability, which is *the* most important property of any RTOS.

Now is the time to extend the Linux scheduler. Consumer hardware already has several cores[5]. More interestingly, the embedded market will see chips with multiple cores [13], thus revealing an even greater need for a multi-core aware scheduler. A scheduler like pfair can enable system engineers to utilize the entire system without having to verify all priorities to ensure that the deadlines are met. Furthermore, it is a *scalable* along several axes:

- The available processing power will scale almost linearly to the number of cores (up to a certain point).
- All deadlines can be met regardless of the number of tasks (given that the overall utilization does not exceed the available processing capacity).

A deadline driven scheduler allows the engineer to focus on the actual task, not on computing the priorities for all the tasks. The current Linux scheduler cannot easily be extended this way, thus the need for a new scheduling class.

1.3 Summary of work

Most of this project is practical work. A large portion of time was spent on understanding the kernel internals, build system and coding style, as well as submitting patches to the scheduler region of the kernel.

As the algorithm of choice is a *global* approach, care had to be taken when choosing data structures, which subsystems to use, and that these were employed as efficiently as possible. To describe this, the implementation part is divided into a series of chapters, each describing, as thoroughly as possible, the subsystems used. Chapter 10 ties everything together. Tasks too small to justify a dedicated chapter in part II are included here.

[4]Change the parameters, period, deadline, worst case execution time, and if it is periodic or sporadic.

[5]Intel and Advanced Micro Devices (AMD) both offer quad-core chips, Sun with its Niagara processor takes this to 8, each capable of running 4 threads, to name a few.

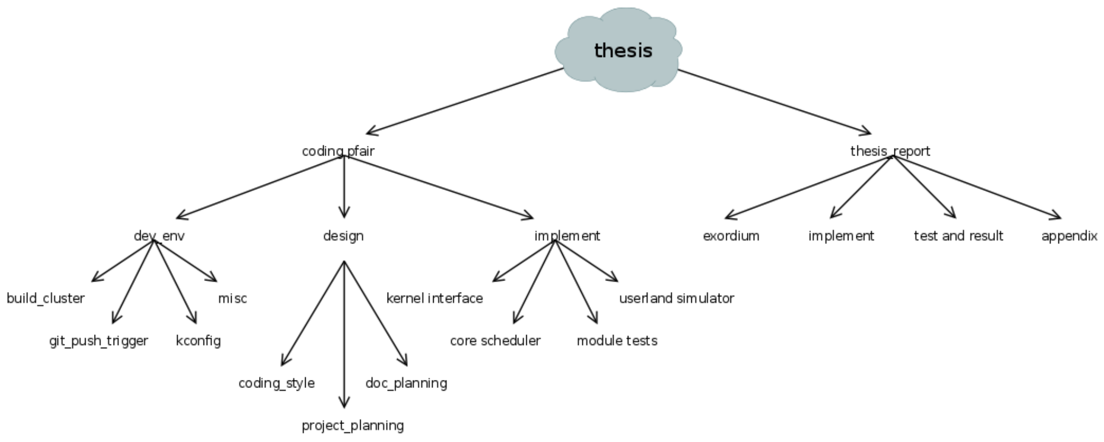


Figure 1.1: Current tree describing the different tasks and how they are represented in the TaskJuggler files

The rest of this section deals with tasks and challenges required by the project, but not directly relevant to the scheduler itself. Nevertheless, time and effort was spent on these topics, and they are relevant to the report.

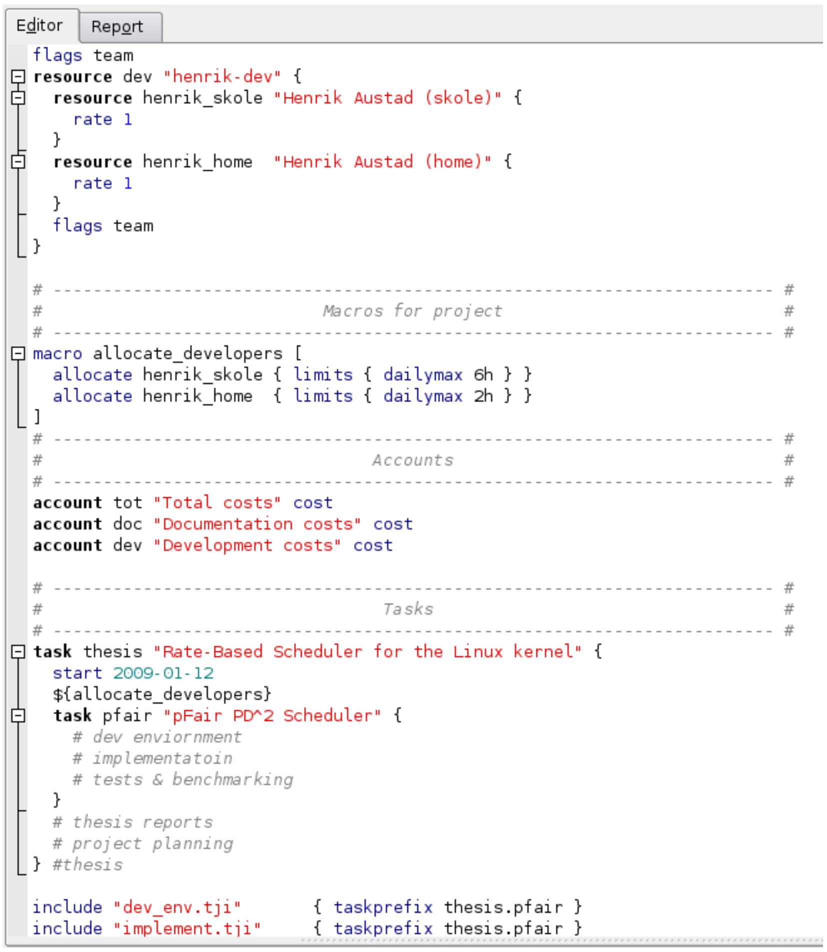
1.3.1 Bringing order to chaos

It was clear from the beginning that this was a complex task. A vast amount of details needs to be addressed in order to get a working scheduler. As the scheduler is a central part of the kernel, adding a new scheduling class is a rather complicated affair. Also, several smaller subtasks had to be identified and properly planned.

TaskJuggler was used in order to organize the project. The process of identifying all the subtasks worked quite well. Granted, it had to be refined and altered slightly, but by and large, the initial tree of tasks has been kept more or less unchanged. Fig. 1.1 shows how the subtasks are ordered in a tree-like hierarchy. Also, note that the right side of the tree is not complete. This was left out to make the diagram more illustrative.

In TaskJuggler, all tasks are listed in a text file format, and you then compile it into various charts and diagrams. From the list of tasks, TaskJuggler will order elements in the best way possible way and save you a lot of time moving boxes around. The project website[6] offers several excellent tutorials and a very

[6]`http://taskjuggler.org/`



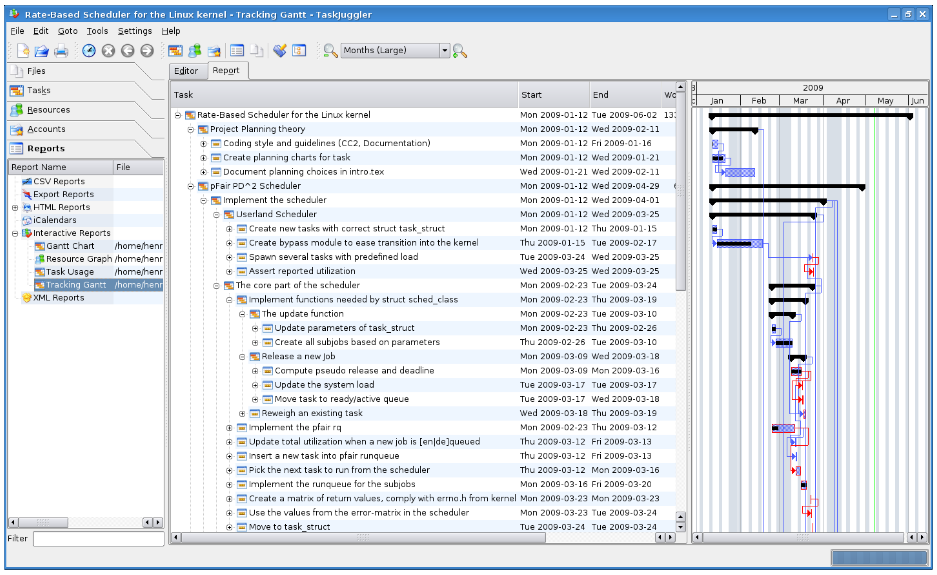
```
flags team
resource dev "henrik-dev" {
  resource henrik_skole "Henrik Austad (skole)" {
    rate 1
  }
  resource henrik_home  "Henrik Austad (home)" {
    rate 1
  }
  flags team
}

# ------------------------------------------------------------------------ #
#                            Macros for project                            #
# ------------------------------------------------------------------------ #
macro allocate_developers [
  allocate henrik_skole { limits { dailymax 6h } }
  allocate henrik_home  { limits { dailymax 2h } }
]
# ------------------------------------------------------------------------ #
#                                 Accounts                                 #
# ------------------------------------------------------------------------ #
account tot "Total costs" cost
account doc "Documentation costs" cost
account dev "Development costs" cost

# ------------------------------------------------------------------------ #
#                                  Tasks                                   #
# ------------------------------------------------------------------------ #
task thesis "Rate-Based Scheduler for the Linux kernel" {
  start 2009-01-12
  ${allocate_developers}
  task pfair "pFair PD^2 Scheduler" {
    # dev enviornment
    # implementatoin
    # tests & benchmarking
  }
  # thesis reports
  # project planning
} #thesis

include "dev_env.tji"      { taskprefix thesis.pfair }
include "implement.tji"    { taskprefix thesis.pfair }
```

Figure 1.2: The main editor window in TaskJuggler, showing the "Master" document where all other documents are included and the base environment is defined. Note how several developers can be added, each with different hourly effort.

thorough manual.

Even though a lot of the deadlines set in the original schedule proved to be either too optimistic or too pessimistic, TaskJuggler turned out to be a valuable tool for listing all tasks, dividing these into "supertasks" and subtasks. From the supertasks, a natural report-layout grew, and in part II, the chapters are more or less a reflection of these "supertasks" found in this process. The entire TaskJuggler schedule can be found in the attachment under `planning/` .

One of the most prominent features lacking from TaskJuggler, is a way to export a diagram to a picture. At the moment, you are more or less tied to


Rate-Based Scheduler for the Linux kernel - Tracking Gantt - TaskJuggler
File Edit Goto Tools Settings Help
Months (Large)
Files
Tasks
Resources
Accounts
Reports
Editor
Report
Filter
2009
Jan
Feb
Mar
Apr
May
Jun

Report Name	File
CSV Reports	
Export Reports	
HTML Reports	
iCalendars	
Interactive Reports	
Gantt Chart	/home/henr
Resource Graph	/home/henr
Task Usage	/home/henr
Tracking Gantt	/home/henr
XML Reports	

Task	Start	End	Wo
Rate-Based Scheduler for the Linux kernel	Mon 2009-01-12	Tue 2009-06-02	13
Project Planning theory	Mon 2009-01-12	Wed 2009-02-11	
Coding style and guidelines (CC2, Documentation)	Mon 2009-01-12	Fri 2009-01-16	
Create planning charts for task	Mon 2009-01-12	Wed 2009-01-21	
Document planning choices in intro.tex	Wed 2009-01-21	Wed 2009-02-11	
pFair PD^2 Scheduler	Mon 2009-01-12	Wed 2009-04-29	6
Implement the scheduler	Mon 2009-01-12	Wed 2009-04-01	
Userland Scheduler	Mon 2009-01-12	Wed 2009-03-25	
Create new tasks with correct struct task_struct	Mon 2009-01-12	Thu 2009-01-15	
Create bypass module to ease transition into the kernel	Thu 2009-01-15	Tue 2009-02-17	
Spawn several tasks with predefined load	Tue 2009-03-24	Wed 2009-03-25	
Assert reported utilization	Wed 2009-03-25	Wed 2009-03-25	
The core part of the scheduler	Mon 2009-02-23	Tue 2009-03-24	
Implement functions needed by struct sched_class	Mon 2009-02-23	Thu 2009-03-19	
The update function	Mon 2009-02-23	Tue 2009-03-10	
Update parameters of task_struct	Mon 2009-02-23	Thu 2009-02-26	
Create all subjobs based on parameters	Thu 2009-02-26	Tue 2009-03-10	
Release a new Job	Mon 2009-03-09	Wed 2009-03-18	
Compute pseudo release and deadline	Mon 2009-03-09	Mon 2009-03-16	
Update the system load	Tue 2009-03-17	Tue 2009-03-17	
Move task to ready/active queue	Tue 2009-03-17	Wed 2009-03-18	
Reweigh an existing task	Wed 2009-03-18	Thu 2009-03-19	
Implement the pfair rq	Mon 2009-02-23	Thu 2009-03-12	
Update total utilization when a new job is [en|de]queued	Thu 2009-03-12	Fri 2009-03-13	
Insert a new task into pfair runqueue	Thu 2009-03-12	Fri 2009-03-13	
Pick the next task to run from the scheduler	Thu 2009-03-12	Mon 2009-03-16	
Implement the runqueue for the subjobs	Mon 2009-03-16	Fri 2009-03-20	
Create a matrix of return values, comply with errno.h from kernel	Mon 2009-03-23	Mon 2009-03-23	
Use the values from the error-matrix in the scheduler	Mon 2009-03-23	Tue 2009-03-24	
Move to task_struct	Tue 2009-03-24	Tue 2009-03-24	

Figure 1.3: One of many available reports generated from the TaskJuggler-grammar composed in the editor.

using TaskJuggler alone, and that *requires* KDE. Thus, once you decided to use TaskJuggler, you are not only locked to a specific operation system, you are looked to a window-manager library as well!

1.3.2 Optimizing the build

This project was all about building kernels. From a clean start, building a complete kernel with the Debian default config (used as a form of benchmark), took on average 40 minutes on `shaky`, on `medea` it took about 30 minutes. In App. A both `distcc` and `ccache` are presented. When `distcc` and `ccache` were used, the time dropped to 7-8 minutes, 3 for a more optimized config, less than 1 minute when building just `vmlinux`. Conscious configuration of the kernel is therefore something it has been invested a fair amount of time in.

To make the compilation, linking and installation of new kernel images as simple as possible, a script found in [16] was modified and extended to suit my needs. The script can be found in Appendix C.1 and was a very useful tool as it allowed for the compilation to be started without much required attention, and when done, the computer could be rebooted, with no forgotten steps and no time wasted waiting for one of the sub-commands to finish.

Another *very* useful script, was the `trigger_script.sh` . This was called from a post-update hook on medea. Whenever a branch with the post-fix "`build_`" was received, medea would trigger a complete kernel build, and send the result of the compilation back via email. This allowed for thorough testing without disturbing the workflow *at all*. The post-update script was a modified version found in the Git community. The `trigger_script.sh` was written from scratch. It can be found in Appendix C.2. The latter, when combined with a properly configured `distcc` -cluster made adding changes to the kernel (in particular `sched.h` , which require a full kernel rebuild) a much more pleasant experience.

Building and linking the entire kernel is, however useful, not always necessary. Often, all you need to do, is to make sure that no syntactic errors have been introduced. The build system therefore has the option of just compiling a part of the hierarchy. `make M=kernel/` [7] will do this. Do note that you should have compiled the kernel once to allow for linking to other parts (in particular the `include/linux/` directory).

1.3.3 It's all about source control

Git[8] is the source control tool used by the kernel. It was written by Linus Torvalds after the license to use BitKeeper was withdrawn. As it is, Git is not for the feeble minded or computer illiterate. It was design to aid Torvalds doing his job, and is therefore a tool for developers. User-friendliness maps into expressiveness, and not ease-of-use. However there are ways of making Git a bit easier to use. To name but a few: `magit`, an Emacs mode, and `git-gui`.

One of the many features with Git has, is the possibility to track several remote repositories, and to push to several. Another useful element, is the concept of hooks. This allowed for `trigger_script.sh` . Of course, the *real* benefit with using Git, is the way Git handles content, merging of branches, that it has merge history and the ability to pick individual commits from other branches. Add the excellent email-support and you start to understand why Git has become so popular.

1.3.4 Required knowledge

The aforementioned elements are a bit beside what one would expect this task to require. They will always be handy, but during the past few months, it has become more and more apparent that they are not only handy — they are *vital*. Without Git, sending patches and keeping the work tidy and ordered would be

[7] Substitute 'kernel/' with the directory of your choice.
[8] Git is the name of the project, when referring to the command, `git` should be used.

impossible. Without distributed and cached compilation, the time required to build all the kernels (several hundreds, if not more, have been compiled) would have crippled all progress.

In order to attain an efficient scheduler, a few additional skills are required. These include good knowledge about computer architecture (cache awareness and memory layout especially), sound theoretical understanding of real-time scheduling and operating systems (preemption, locking, race conditions etc.) and of course: the C programming language. This reader is presumed to be familiar with these.

1.3.5 Related work

Patches submitted

While working on this project, I have submitted a few patches to the kernel list. Some of the minor ones where accepted almost immediately. Two of the larger patches, which dealt with moving priority calculation into two new scheduling class functions, were NACK'ed. The reason was to avoid further function call overhead. This was a bit disappointing, but it highlights 2 key points; *a)* patches are reviewed thoroughly, and no new change is accepted without testing and discussion and *b)* the scheduler code is optimized for speed, not readability or maintainability, hence new code *must* take this into consideration.

This might seem trivial, but it was one of the most intimidating tasks undertaken. Submitting patches to LKML is not easy, nor without hazards[9]. The patches should be formatted according to a very strict set of rules, and they better have a very good reason for being submitted in the first place. The kernel community is a very technocratic crowd, and the only real currency is knowledge.

Discussions on LKML

Strongly related to submitting patches, are the discussions on the mailing list. The volume is high, sometimes reaching almost 1,000 new messages pr. day. To filter out the relevant messages, reading the patches and understanding the impact has been a much larger job than expected. After some creative testing with procmail, a reasonable setup was found, where all emails were sorted by topic. This way, the volume was manageable.

[9]If you do a bad enough job, you will be given a *very* explicit message.

The discussions tend to revolve around patch review, new feature discussion or regression. Then there are the version releases which sparks off a discussion once in a while[10].

Discussions on IRC

Several of the kernel hackers frequent on various IRC channels. Some of the most fruitful discussions relevant to the project has been conducted here. Peter Zijlstra and Bill Huey both provided a lot of insight and invaluable comments on the planned scheduler. Most of the practical advice I have received regarding actual kernel development, come from these discussions. Some channels for the interested: `#linux-rt@freenode.net`, `#kernelnewbies@oftc.net`, `#sched@oftc.net`

1.4 Related projects

Real time research and development is an active area. Since Linux is one of the few truly open operating system kernels, real-time forks of the kernel is a natural thing. That Linux is supported on more platforms than any other OS, as well as having support for more hardware than most other operating systems, is yet another reason for adding real-time capability to Linux. The Linux Real-time preemption patch, which is the semi-official kernel project for gaining real-time capabilities in the kernel, is covered in the next chapter.

1.4.1 LitmusRT

Of all the other real-time project currently undertaken, LitmusRT[11] is probably the most advanced and well maintained. It is from this research group that pfair has its origins. They also have an implementation of pfair, but it lacks in several aspects and is difficult to integrate into the current kernel as it adds changes to large areas of the kernel. LitmusRT aims to do a lot more than "just" scheduling, recent activity has focused on locking and how to avoid deadlocks and large unpredictable latencies.

By being very dependent upon HZ, they cannot scale the timeslice lower than 1ms, nor can they move above 10ms. Furthermore, once the HZ is set, it cannot be changed without recompiling the kernel. It will also add a overhead to the kernel, even when no pfair tasks are running. Finally, they use a linked list for

[10]especially the discussion following Linux v2.6.29: `http://lkml.org/lkml/2009/3/23/449`, which turned into a massive thread discussing the ext3/ext4 filesystem performance

[11]`http://www.cs.unc.edu/~anderson/litmus-rt/`

storing the entire runqueue, making it very expensive to schedule a large set of tasks.

The complexity of merging the pfair-related code into the standard kernel was the reason why the LitmusRT pfair scheduler was not ported to the vanilla kernel, but instead written from scratch.

1.4.2 RT-Linux

RT-Linux is now a registered trademark of Wind River. Being one of the first real-time Linux systems, RT-Linux acted as a *hypervisor*, letting the kernel run as a standard process. Having complete control over all interrupts, being able to compartmentalize the kernel behavior and to react very quickly to external events, RT-Linux was able to achieve hard real-time status.

1.4.3 Bill Hueys generic deadline framework

Bill Huey is currently working on a generic deadline driven framework. Originally developed to be partitioned, it is now planned to gradually extend to global algorithms. The framework will handle all the basic requirements from connecting to the rest of the scheduler, setting timers and acting as a mediator for new tasks. An implementing class need only provide an acceptance function for new tasks and a way of returning the next task to run when the currently running should be rescheduled. Some internal logic for storing the task is needed as the framework will keep the task in an unsorted list.

Still in its infancy, the project shows great promise, and the at the time of this writing, it seems this project can be tied into the generic framework as a first proof-of-concept. The following paragraph describes the current framework which is partitioned. The global is still being discussed in various channels, involving several people, including me.

General idea:
The deadline scheduling is straight forward. Each task classified as either periodic or sporadic. The difference is that a sporadic task will migrate once preempted by a periodic task, a periodic task will keep to the given CPU as it is here it has reserved bandwidth. The framework keep track deadlines and will signal a task if it misses the deadline. Each task is given a bandwidth tied to WCET and deadline. To keep compatibility with the existing kernel, a runnable EDF task will be given the systems highest priority, and be classified as less runnable than

the idle-task when the budget is exhausted.

The priority inversion/inheritance solution is quite similar to the proxy execution protocol planned in this project, see Sec. 4.4, but due to the partitioned landscape, some extra concepts need to be introduced: task groupings and yield-nests. The former is a way of visualizing the relationship between tasks, the latter is the actual concept implemented. If task A requires a resource R currently held by B, you have a $A \rightarrow B$. Task A is *not* removed from the run-queue, and when it is scheduled, A's budget is decreased and B is scheduled to run. The `rt-mutex` keep track of the blocked tasks, and when more tasks request R, or some resource held by the tasks waiting for R, you now get a DAG. This DAG will gradually expire. By degrading B to sporadic status (which will trigger reallocation to other CPUs but without moving B from the original runqueue), the lock will be released as quickly as possible as B will execute on different CPUs on behalf of the tasks in the wait-list. Once B releases R, the `rt-mutex` will notify the next task. As tasks will gradually expire, they will be removed from the wait-list, thus forcing the `rt-mutex` to give access to the task in the list with the shortest deadline.

1.5 The rest of this report

The rest of this part is dedicated to general information, such as the Linux kernel, with particular focus on the task scheduler. The theory is introduced in Chapter 3. Most attention is given to multi core algorithms, but for completeness, a short section is given to single core scheduling.

Part II contains design and implementation. Ch. 4 outlines the overall design for the scheduler and moves on to the implementation. The implementation is split into several chapters, each describing a defined subsystem needed by the scheduler. Beginning with the overall design in Ch. 4, memory management, particularly the SLAB-allocator, is discussed in Ch. 5. The two major data structures, linked lists and red-black trees, are covered in Ch. 6.1 and 6.2 respectively. Ch. 7 follows with the SysFS infrastructure. The subsystem-chapters sequence is closed with a look at timekeeping in Ch. 8. Then, finally, the core scheduling algorithm and implementation is treated in Ch. 9.

Part III All the loose ends and a few various small topics are tied together and discussed in Ch. 10. Then the scheduler itself is discussed and the project evaluated in Ch. 11. The pfair-algorithm has been found inadequate despite the initial enthusiasm. To battle this, an adaptive version of Modified Least Laxity First[20] (global adaptive slack-based deadline driven scheduler) is presented, and to the authors knowledge, no such algorithm has been implemented in any operating system.

Part IV contains the appendices describing distributed and cached compilation of the kernel in App. A, the test-modules in App. B and some small scripts in C.

Chapter 2

The Linux Kernel

"Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with your using PREEMPT_RT*."*
— Linus Torvalds

This chapter gives an introduction to the Linux kernel. Starting out with background in 2.1. Sec. 2.2 then covers the current scheduler with classes, policies and entities. This is meant to provide the reader with the essentials of the scheduler internals. The RT Preemption Patch and why this moves Linux into the RTOS realm, is discussed in 2.3, while the kernel API is presented in 2.4. The latter is a pervasive topic, and naturally, only relevant sections of the API is discussed. The "non-stable kernel API" policy is also discussed in this section. Sec. 2.5 concludes the chapter.

2.1 Background

In 1991, Linus Torvalds released the first version of Linux, an open source clone of the UNIX operating system kernel[1]. Originally written specifically for the Intel 80383 microprocessor, Linux has since grown and it supports 22 different architectures, which is more than any other operating system today[2]. The reason for this tremendous success was the early adoption of the Gnu Public License (GPL). Torvalds later stated *"Making Linux GPL'd was definitely the best thing I ever did"*, describing how the shared effort across the world was made possible only through the public license.

[1]Linux is the kernel in a GNU/Linux operating system.
[2]`http://en.tldp.org/HOWTO/User-Group-HOWTO-1.html`

Today, Linux is a preemtible, fully POSIX-compliant operating system kernel. It has support for a wide range of hardware through its device driver system, it scales down to system as small as a watch, and runs on approximately 88% of the top500 [3] supercomputers.

2.2 Current status in the Linux scheduler

The Completely Fair Scheduler (CFS) was introduced in Linux v2.6.23. The old O(1) scheduler can still be found in the `sched_rt`.

. The scheduler in the kernel implements a set of *scheduling classes*. Each class is responsible for one or more *scheduling policies* Policies govern how tasks will be scheduled. A task can only belong to a single policy. As of this writing, the official kernel supports 5 policies, all of which are discussed below. As the scheduler actually is a collection of scheduling policies, the "main scheduler", is the logic in `sched.c` that drives the overall flow. It is from here that each policy and class is invoked. This can be a bit confusing at first glance.

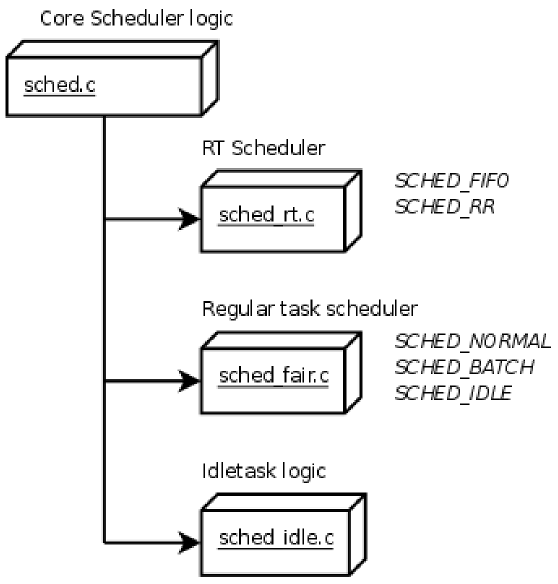


Figure 2.1: Where the different classes are implemented, and how the policies relate to this. Note that SCHED_IDLE and `sched_idle.c` is not the same thing. The idletask logic is invoked when *no* task is eligible to run on that CPU.

[3] `http://www.top500.org/stats/list/32/osfam`

2.2.1 The “normal” scheduling policy

The CFS handles all tasks with priorities in the $[-19, 20]$-range[4]. The class implements three scheduling policies SCHED_NORMAL, SCHED_BATCH and SCHED_IDLE.

The CFS schedules `sched_entities` , not tasks. Both the CFS and the RT-scheduler uses `sched_entities` as this allows for a more generic approach. This allows groups of tasks to be scheduled similarly to single tasks. These entities are then stored in a sorted red-black tree. The key used to keep the tree sorted is the delta run-time value for the entity (`se.vruntime`). The lower the value, the further left in the tree the entity will be inserted. Every once in a while the currently running task will be preempted and inserted into the runqueue. The entity's vruntime is updated, and the task (being treated as a sched-entity) will be inserted at the appropriate place. The scheduler then picks the leftmost leaf in the tree, i.e. the task with the gravest need for running. From this, it becomes clear that CFS does not use priorities directly to schedule tasks. The priorities are used to determine the base timeslice. The time a task is allowed to run will eat into this slice, and once it becomes exhausted (or relatively “more exhausted” than another tasks timeslice, it will be preempted and the other task allowed to run).

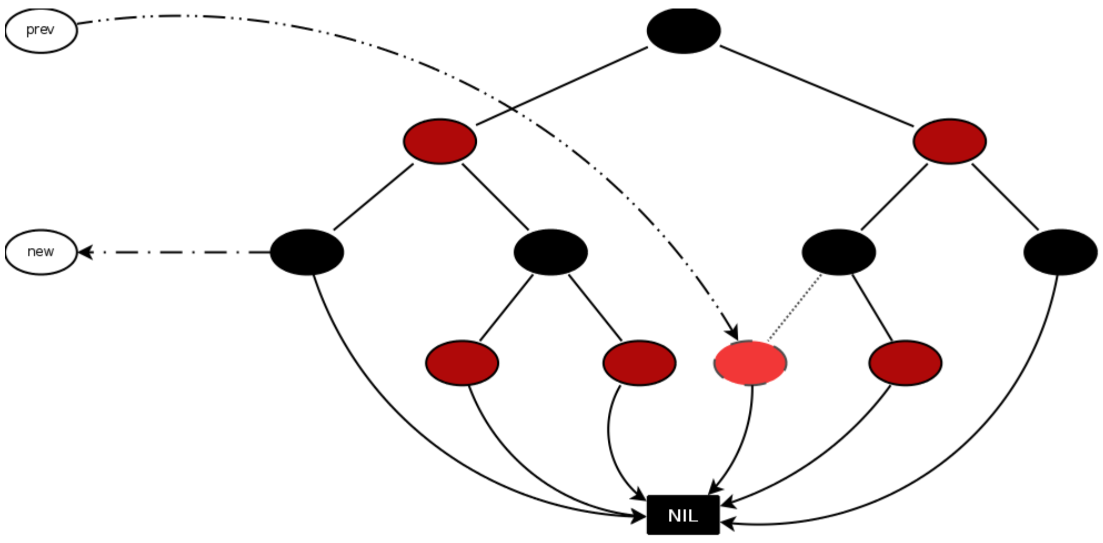


Figure 2.2: Figure showing briefly insertion and which node that will be returned when the next task is picked from the CFS red-black tree.

[4]or 0 to 39, depending on whether you look at priorities from the users perspective, or from the kernel

SCHED_NORMAL is the class for "normal applications". It will be given a default weight corresponding to the priority, and this will influence its place in the runqueue tree.

SCHED_BATCH are for tasks that does not require interactivity. This leads to less frequent preemption and they may execute for long period of time. This is ideal for background jobs having a lot of work to do, but has little or no dialog with a user or other jobs.

SCHED_IDLE - the last policy to be handled by CFS. This is a way of setting the priority of a task even lower than nice 19. To avoid conflict with the idletask, this policy has slightly higher priority, but lower than any other task from the other policies.

2.2.2 Real-time class and policies

The sched_rt.c contains the real-time scheduling class. This in turn, implements the rt-policies,SCHED_FIFO and SCHED_RR. It embodies the old O(1) scheduler, although the 140 levels are reduced to 100 (see [4] for more details around this). Each level in the runqueue contains a linked list of tasks eligible to run. Both policies uses the same runqueue, but SCHED_FIFO does not use timeslices.

In practice, this difference means that a SCHED_FIFO task will run indefinitely until it either blocks, suspends or a task of higher (rt-) priority enters STATE_RUNNING. SCHED_RR will let a task run until it has exhausted its timeslice before inserting the task at the tail of the priority level list. This allows other tasks on the same priority level to run, but it will block all tasks below it, and it will be blocked by any task with higher priority.

2.2.3 Challenges faced with the current scheduler

One of the major problems with any large project, is to handle the schedule. For normal tasks, this is seldom a problem: the scheduler does a fairly good job, and the tasks do not have very strict deadline requirements.

For real-time tasks, things are different. Very often, the priorities must be worked out to ensure that the deadlines will be met. It is therefore of vital importance that the operating system is *deterministic*. Nevertheless, as the rt-tasks in the kernel have static priorities, the kernel will not provide any help to the application programmer to determine the static schedule. If the kernel

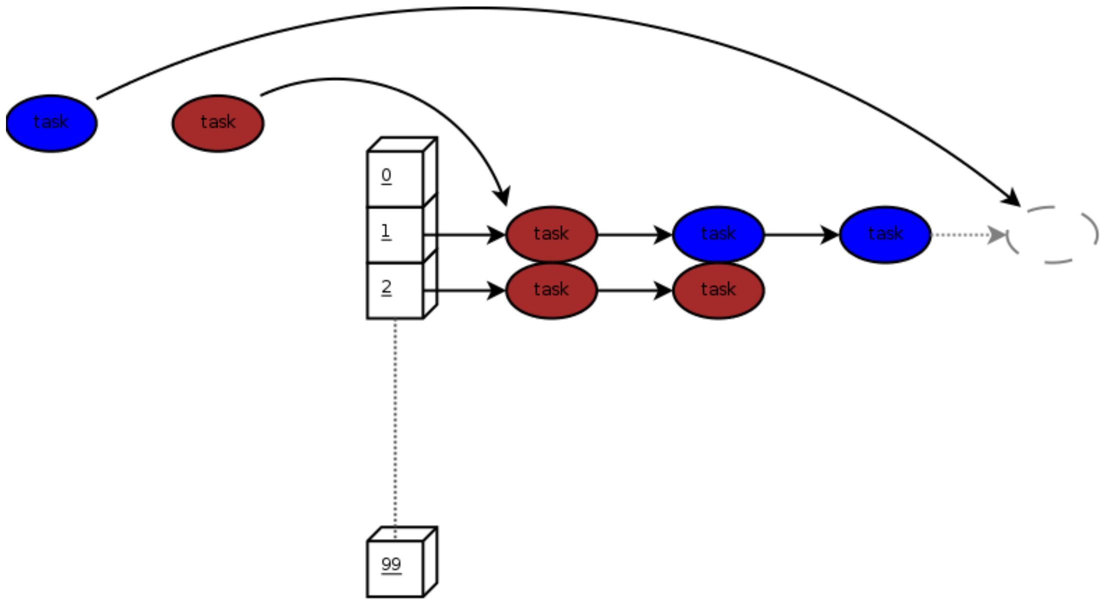


Figure 2.3: Insertion of a task into the real-time runqueue. Red indicates SCHED_FIFO, blue SCHED_RR. The SCHED_FIFO task immediately enters the head, SCHED_RR moves to the tail.

was deadline aware, this would be made much simpler, as the priorities often are inferred from a set of expressed deadlines.

2.3 Ingo Molnar's RT-preemption patch

The RT Preemption Patch (RT-Patch)[25], is a set of patches currently maintained by Thomas Gleixner (originally by Ingo Molnar). The RT-Patch includes full kernel preemption[5] Priority Inheritance, conversion of ISR into schedulable kernel threads, transforming spin-locks into adaptive spinlocks and mutexes as well as deadlock detection logic.

Several features from the RT patch series have found their way into the mainline kernel. Amongst those are generic IRQs, gettimeofday() architecture and hrtimers. The latter is discussed later in Ch. 8. Other features remain in the RT-patch. Not because they are not useful, but because the mainline kernel is a GPOS and some of the features in the patch-series is geared towards increased determinism often resulting in higher kernel overhead (and lower throughput).

[5]Except in a few important and short atomic sections.

The most important feature of any RTOS, is determinism. To obtain this, the OS must not necessarily minimize latencies, but it must make them predictable. The following subsections look at different causes for latencies, and how the RT-patch deals with these. After determinism, latency and throughput are important. The RT-patch improves latencies in the kernel, making the impact of interrupts and syscalls smaller on other applications running.

2.3.1 Fully preemptible kernel

This is perhaps the most "famous" part of the RT-patch. By making the entire kernel fully preemptible, the kernel latencies can be minimized. This started out by changing the Big kernel lock (BKL) from a recursive spinlock to a `rt-mutex`. This was one of the first, dramatic reductions of kernel latencies, and the first element from the RT-Patch to be included in the Linux kernel.

The `rt-mutex` was another improvement to this. Originally, Linux 2.6 did not have mutexes. All it had was semaphores (which, granted, where used as mutexes), and these semaphores where quite slow and did not have any idea about ownership. The mutex knows who the current owner is, and is one of the strict requirements for the Priority Inheritance protocol discussed later.

2.3.2 Threaded ISRs

One of the big contributors to kernel latencies are the interrupt handlers, or ISRs. By moving most of the time consuming work coupled with the interrupt handlers into schedulable threads, it is possible to let a high priority thread execute regardless of network activity, a misbehaving timer or disk activity. In effect, what actually happens is quite simple: the interrupt causes ISR to execute. This sets the IRQ_INPROGRESS flag and redirects the irq handler. If the handler is a "nodelay", it is executed immediately. Otherwise, the `do_irqd` thread is awoken, the interrupt masked and the interrupt exits. When `do_irqd` is executed, it will run the handler and clear the IRQ_INPROGRESS flag, finishing the interrupt handling.

This means that the priority inversion latency (the delay caused for the currently executing thread) will only be the time it takes to mask the interrupt and wake the `do_irqd` thread, a lot less than actually handling the entire interrupt. As a real-time task can be given higher priority than the interrupt thread, the interrupt impact can be negligible. Of course, a misbehaving real-time application can render the entire system useless if *no* interrupts are processed properly, but that is not really an issue. RTOS is all about control. If the application misbehaves, it is the fault of the application developer, not the kernel.

2.3.3 Spinlocks

Spinlocks should be fast. Unfortunately, some events can cause the spinning to take a very long time, ultimately delaying other parts of the kernel. As they avoid sleeping, when the time spent waiting is short, spinlocks are very a effective way of synchronizing resources. However, a spinlock may cause a thread running on another CPU to spin, so great care must be taken in order not to delay other parts of the kernel. The same goes for locks that are shared by interrupt service routines and the "normal" kernel. If one part of the kernel takes the lock without disabling interrupts, the system will eventually deadlock. It goes without saying that spinlocks are a delicate matter, and as such possess a hazard in an RTOS[6].

The RT-patch converts a large portion of the spinlocks in the kernel into mutexes. This allows the task to be preempted, removing a whole series of deadlock candidates. Unfortunately, this also increases the kernel lock overhead dramatically. Some of the locks however, must not be altered. Instead of converting these into mutexes, they are changed to `raw_spinlock`, helping the compiler to optimize the code in a safe manner.

Lately, a new type of spinlock has emerged, the adaptive spinlock. This is a spinlock that will try to spin in order to acquire the resource. If the busy wait has been unsuccessful for a predefined amount of time, it will sleep until the resource is released allowing another thread to execute, possibly releasing the lock.

All of this does not help predictability at all. In fact, it can introduce non-determinism. On the other hand, it helps avoiding deadlocks and starvation as other threads can run in place of a spinning thread.

2.3.4 Priority Inheritance (PI)

Priority Inheritance is a well known way of avoiding, or at least reducing, the effect of priority inversion. Priority Inversion is when a high priority task is blocked by a lower priority task waiting for a shared resource to be released. This is one of the major contributors to latencies, and must be handled carefully. Inversion cannot be completely avoided, but it is possible to avoid unbounded latencies.

[6]also in a GPOS, but the impact of a deadlock in a GPOS is less severe

There are several approaches to avoid Priority Inversion. As the kernel is too big and complex for most of these techniques (design or priority ceiling), the kernel implements the PI. When a high priority thread requests a resource held by a lower priority thread, the priority of the resource owner is boosted to equal the high priority task. This approach works well and scales to large projects.

2.4 Kernel API and conventions

2.4.1 Coding conventions

In order to implement a new scheduling class in the kernel, several of the subsystems must be used. The elements intended for general usage, is placed in the `include/` directory and can be referenced as normal C header files.

2.4.1.1 Return values

The kernel functions normally return negative numbers upon error, 0 on success, or a positive integer describing the amount the operation caused.

2.4.1.2 Coding style

In a project as large as Linux, with approximately 1000 active developers spread all over the world, the need for a well defined coding style is imperative. In fact, patches will be rejected if the style is too deviant from the norm. As a first meeting with the kernel code, this can be a bit frustrating, but after all, with well over 7 million lines of code[7] the rules *must* be strict.

Unlike many other projects however, the Linux kernel has very clearly stated policy about unstable kernel API. You should never, *ever* assume that the API stays the same. No interface is perfect, and instead of keeping a consistent API and ensuring backwards compatibility, the kernel hackers believe in a clean slate. If it is wrong — fix it. The only way of programming kernel code, is to know the API, and stay on top of it.

2.5 Summary

In this chapter we saw how the kernel handles all types of tasks, we looked at a few of the limitations with the current real-time scheduler, and how the real-time

[7] 7 210 957 lines of ANSI-C code found by `sloccount` for v2.6.30-rc5

preemption patch tries to solve some of these issues. Then, at the end, we looked briefly at the kernel API with special focus on coding conventions.

Chapter 3

Scheduling Theory

"The key is not to prioritize what's on your schedule, but to schedule your priorities."
— Stephen R. Covey

The previous chapter covered the current Linux scheduler and indirectly some of the theory. This chapter introduce theory for deadlines, global and multicore scheduling algorithms.

Section 3.1 looks very briefly at background and terminology, with particular attention given to common simplifications regarding scheduling algorithm design. Section 3.2 then presents single core scheduling before the main topic of this chapter, multi core scheduling algorithms are presented in Sec. 3.3. As this section outlines the general idea for multi core algorithms, the theory for pfair (the algorithm treated in this project) is presented in full in Sec. 3.4. The chapter is then concluded in Sec. 3.5.

3.1 Background and terminology

The "era" of real-time scheduling theory began with the famous paper by Liu and Layland ([18]) where they presented Rate Monotonic and Earliest Deadline First. They then showed how these would make it possible to schedule complex sets of real-time tasks in "multi-computers".

A real-time system consists of a set of tasks, τ, where an individual task i is identified by τ_i. Each task is characterized by its release-instances, called a job which will occur at least T_i time-units apart. This is called the period of the task. The kth job is denoted $\tau_{i,k}$. Each task has a relative release-time, R_i and each job has an absolute release-time $r_{i,k}$. The same can be found for deadlines, the time for when the job *must* have finished, as D_i and $d_{i,k}$ respectively. The time required to run a single job is called its execution time ($c_{i,k}$), and the worst case execution time is $C_i = \max(c_i, k)\ \forall\, k \in \mathbb{N}$. Finally, a task has a given *utilization*, $U_i = \frac{C_i}{T_i}$, $T_i = D_i$.

A (real-time) schedule is said to be feasible if a collection of (real-time) tasks can be ordered in such a way that all the deadlines for all the tasks are met. A set of simplifications are normally assumed when discussing real-time scheduling:

No dependencies between tasks This includes hardware resources, or software resources (locks, queues, other shared datastructures). By assuming this, deadlock- and starvation logic can be simplified. Furthermore, priority (deadline) inversion (PI) is simplified and can be handled easily or avoided all together.

Deterministic periods and deadlines Normally the deadline is set to the period, and the task is released *exactly* at the period. This makes several equations simpler. In some cases, the scheduler uses the shortest of the two as an approximated period and consider the larger period to be a special case when the task is released late.

Free context switches changing between tasks is essentially free, or, takes negligible time.

Isolated system By ignoring all other elements in the system, other tasks and the effect they play on the system, the real-time tasks can be analyzed separately.

By looking briefly at this list, we can see that in any normal setting, this does not hold. However, by assuming this, the *general* idea can be expressed more clearly and verified. Then, upon implementation and testing, extra logic can be added that will deal with these elements. This is a flaw most algorithms are being criticized for, but something it is very hard to do anything about. If you take every single unpredictable element into consideration when devising a new, clever algorithm — you will never get anywhere.

For a complete list of terminology, please see [4, Sec. 2.1]. The glossary (page 121) also contains some of the terms used. Where appropriate, the terms will be discussed in situ.

3.2 Single core scheduling algorithms

This section gives a very brief introduction to single core scheduling algorithms. Since the target system is multicore, this section is only included for completeness, and for helping understand the problems faced with multi core systems. For a better discussion about single core scheduling, see [4, Sec. 3.3].

3.2.1 Rate-Monotonic (RM) scheduling

In RM, a tasks priority is determined based on the period. The shorter the period, the higher the priority. In many cases, this assumption is correct. If a task has a very short period, it is natural to assume that the corresponding deadline is also short, and hence, the priority must be high.

The acceptance test for the RM scheduler is very simple: Keep accepting new tasks for as long as the overall utilization is below the following limit:

$$U = \sum_{i=1}^{n} u_i \leq n(\sqrt[n]{n} - 1) = ln(2) \approx 69.3\%$$

One of the major advantages of RM, is that it has low overhead. Furthermore, it is relatively simple to implement and works well with other priority based scheduling algorithms. As the “normal” way of implementing a scheduler, is by using priorities, this is a very important feature.

3.2.2 Earliest Deadline First (EDF) scheduling

EDF looks directly at the deadline of a task. Where RM assumes that the deadline is related to the period, EDF ignores the period and picks the task with the shortest deadline. The acceptance test:

$$U = \sum_{i=1}^{n} u_i \leq 1$$

EDF can utilize the processor fully, but it will have higher overhead than RM. In many cases (most notably in theoretical work), this is ignored. It has been shown that EDF actually results in. *fewer* task switches [8], so the overhead with respect to task switches is lower. However, since “all” operating systems use priorities as the base element for scheduling tasks, including an EDF scheduler will be difficult. The common approach is to map the deadlines to priorities, but for each new task arriving, the *entire* schedule must be recomputed and some, if not all, priorities changed. This lead to a very inefficient implementation.

3.3 Multi-Core scheduling algorithms

Extending single core algorithms to multicore systems is not trivial. In fact, in many cases, known assumptions and features fall apart. One example is EDF and the utilization bound. On a multicore system, EDF has utilization bound by $U \leq m(u_{max} - 1) - u_{max}$ where u_{max} is the highest utilization by a single task in the schedule[14]. EDF-US[1/2] can then increases this to $U \leq \frac{m+1}{2}$ [5]. As a starting point, this section will use EDF as an example to show that EDF can be implemented differently, and how that will affect the overall performance and utilization.

Multicore scheduling algorithms can be classified into two distinct groups, *global* and *partitioned*. Global algorithms use a single runqueue and centralized dispatch logic to handle the tasks. Partitioned algorithms use a per-CPU[1] runqueue and logic. A term used to describe something in between, is called a *clustered* scheduler. A global scheduler is a cluster of one domain. A partitioned scheduler is a cluster of n domains, where n is the number of CPUs.

3.3.1 Global

A global scheduler is a scheduling algorithm with a *single* runqueue for the entire system. By looking at the entire system at once, the algorithm can pick the task best suited to run at a given CPU/core at a given point in time. The benefits of doing this, is that it is much easier to achieve optimal scheduling decisions. You also avoid the bin-packing problem that the partitioned (clustered) algorithms strive to avoid. On top of this, since the algorithm is global, you will not have to worry about load balancing logic, as this will be handled by the scheduler.

[1]The recent advances in multicore systems lead to a lot of confusion between CPU and core. In this report we use the two to describe the same thing - a single entity capable of executing a task.

There are downsides:

- The moment you have several cores competing for a single shared resource (namely the runqueue), the scheduler code can, and will block, causing non-deterministic kernel latencies.
- The runqueue will introduce race-conditions (previous item), and the scheduler is *required* to introduce logic to handle this. One approach is *staggering*, but either way, it will introduce higher overhead and more complex code.
- Another downside, is the overhead. Since the scheduler now must consider *all* tasks running on *all* cores, every single scheduling decision becomes quite involved. This translates directly into *slow* and *not scalable*. The requirement for fast and efficient runqueue implementation is therefore paramount.
- Yet another problem, is memory bandwidth for the shared runqueue. Besides (dead)locking, the data structure must be read by all cores, forcing the data to be written out to memory (due to cache invalidation whenever something is updated in the runqueue) and read back in. This will cause high memory traffic, decreasing the performance dramatically.

In fact, one of the strongest objections *against* global approaches, is the scalability and cache invalidation issues. As long as an algorithm is truly global, this will be an issue, as global means sharing information across all domains. This is one of the reasons why global algorithms have received little attention from researchers over the years.

3.3.2 Partitioned

A complete opposite way of doing things than the global, is the partitioned approach. In a partitioned scheduler, each CPU will have a separate instance of the runqueue, and one task can only reside in one runqueue at a time. This removes a lot of the scalability issues related to the global approach. The Linux kernel is a very good example of this — 2.4 kernel used a global runqueue for the tasks, and this made scaling to several CPUs very difficult. In Linux v2.6, the scheduler was rewritten into a partitioned scheduler, and this removed several performance problems. This is living proof that partitioned scheduling is both scalable and fast.

However, the moment you start looking at each CPU as a separate domain, you run into bin packing problems. The bin packing problem manifests itself by making it difficult to choose which runqueue a new task should be assigned to in order to maintain a feasible schedule. In fact, it can be shown that this is an NP Complete problem in the strong sense. Some *heuristic* must be used when assigning a task to a runqueue.

FF First Fit, add the task to the first non-empty queue with enough reserve resources remaining.

BF Best Fit, of all the non-empty queues, find the queue where the least amount of resources are left after allocation.

NF Next Fit, take the previous queue and add the task to the next queue in the list.

WF Worst Fit, try to allocate the task into the queue with the greatest remaining resources.

Common for all: If no non-empty queues can accept the job, add it to the first empty queue. If no queue can hold it, fail the allocation.

Since the bin-packing is done as an approximation, the load will not be evenly distributed between the domains. To even out the load, and handle load-transients, most systems use a *load balancer* to move a task from one domain to another. This is needed in every partitioned scheduler since you *cannot* make optimal scheduling decisions, and is the cause for yet another non-deterministic, sporadic latency.

In the Linux kernel, none of these approximations are used. A real-time task is set to execute on the core where it upgraded its status from SCHED_NORMAL to SCHED_FIFO (or SCHED_RR). Then, if the load is very unbalanced, the CPU with lowest load will try to grab tasks from other, highly loaded, CPUs. The reason is as follows: if a non-rt task is allowed to run on a CPU (which it must if it is to elevate its status from SCHED_NORMAL to SCHED_FIFO), that CPU has already low load and can therefore accept a new RT-task.

3.4 Rate based scheduling - Pfair

In pfair scheduling ([1], [2], [3], [22]), tasks are required to execute at a steady rate. What this means, is that not only the deadline but also WCET will affect how the task runs. Pfair will then spread the execution of the task evenly across the period. Fig. 3.1 gives a simple illustration for how a task will be split into quantum length subjobs and each subjob scheduled individually (although

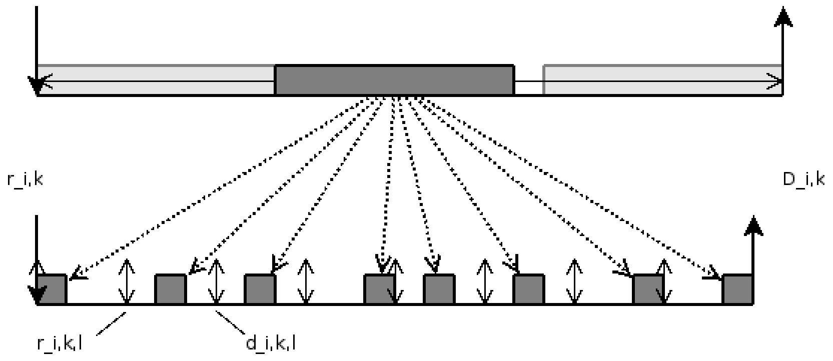


Figure 3.1: How a task will be split into subjobs and scheduled with the pfair algorithm. Unused areas are slots where subjobs from other tasks can be scheduled to run.

sequentially). Note how the gray areas indicate possible execution times in a standard EDF scheduler without preemption. This is one of the major problems with EDF, even though the task meets it deadline, you have no control over when in the period the task will execute. This results in *jitter*.

In pfair, the normal task description is extended to *subjobs*. A subjob is denoted as $\tau_{i,k,l}$. A subjob is given release, deadline and execution cost the same way a job is. The execution cost, however, is pre-determined and matches the quantum length exactly. The release and deadline is denoted $r_{i,k,l}$ and $d_{i,k,l}$ respectively. The subjobs must execute sequentially, meaning that two subjobs cannot execute on different CPUs at the same time. Scheduling decisions are taken at the timeslice boundaries, and a subjob will not be allowed to run after this boundary. The subjobs allows pfair to take the weight of T_i into consideration without a lot of extra logic. A large WCET gives may subjobs, and therefore higher bandwidth.

Note: the notation can differ, and we strive to use the same notation as the majority of real-time scheduling articles. In some articles however, T_i is used to describe the ith release of a task T, and T.p is used to describe the period. The convention in other articles, however, is to use T to describe the period, or the **T**ime between releases and P to indicate the priority. The set of all tasks is given arbitrary letters, but 'S' means the schedule.

A schedule is said to be *pfair* if the *lag* satisfies the following condition at time t after a set of subjobs have been allocated time to run:

$$lag(\tau_i, t) = \left(\frac{C_i}{T_i}\right) t - allocated(\tau_i, t) \tag{3.1}$$

$$|lag(\tau_i, t)| < |timeslice| \quad \forall \tau_i \in \tau \tag{3.2}$$

Each slice can be identified by $[t, t+1)$ and the time a given timeslice spans can be found by:

$$timespan = [t * |timeslice|, (t+1) * |timeslice|) \quad \forall t \in \mathbb{N} \tag{3.3}$$

If the execution cost of a task is not dividable by the timeslice, the last subjob will *waste* processor time[2]. Thus, the smaller the timeslice, the less resources will be wasted and the scheduler will approach the *ideal pfair scheduler*. Each subjob will have a sub-release time and corresponding deadline. These are given in t, where the release is at the start of the interval, and the deadline at the end. As an example, a release- and deadline-pair (0,0), indicates that the subjob will be released at the start of a timeslice and must finish before the end of it.

The optimal length of a timeslice must be determined experimentally. Even though slices of infinitesimal length will give an "optimal" scheduler, actual costs such as task preemption, cache warmup and time required to handle timer-events must also be taken into consideration.

3.4.1 Calculating release and deadline

$$wt(\tau_i) = \frac{C_i}{T_i} \tag{3.4}$$

$$r_{i,j,k} = \left\lfloor \frac{(l-1) * T_i}{C_i} \right\rfloor \tag{3.5}$$

$$D_{i,k,l} = \left\lceil \frac{l * T_i}{C_i} \right\rceil - 1 \tag{3.6}$$

$$w(\tau_{i,k,l}) = [r_{i,k,l}, D_{i,k,l}] \tag{3.7}$$

The weight of a task is used describe the behavior. A task cannot have weight higher than 1.00 as that would require it to run concurrently in the real sense, on several CPUs. All tasks with weight 1 can be given a dedicated CPU, and tasks with $wt(\tau_i) \geq 0.5$ are called *heavy*. The weight is found in Eq. 3.4. The pseudo release (Eq. 3.5) gives the start of the window in which a subjob must

[2]We will show later how this can be minimized, and that other non-pfair tasks can be allowed to run instead.

be released. All subjobs are expected to be released at its window boundary, and that the start of the window correspond to the start of system timeslice.The deadline gives the last timeslice in a subjobs window where it can be scheduled in order to meet the deadline. Eq. 3.6 finds this.

3.4.2 Tie breaking rules

A tie is when two subjobs have the same pseudo deadline, something that will happen quite often in a busy system. In [2], it was shown that only 2 tie-breaking parameters are needed (the disjoint bit and group deadline), and those can be computed once when the task is created. This is called PD^2 and is what we have implemented in this project.

3.4.2.1 Disjoint bit

$$b(\tau_{i,k,l}) = \begin{cases} 0, & if \left\lceil \frac{l}{wt(\tau_i)} \right\rceil = \left\lfloor \frac{l}{wt(\tau_i)} \right\rfloor \\ 1, & else \end{cases}$$

From this, we can see that the only time when a disjoint-bit will be 0, is at the very last subjob. The disjoint bit will cause the scheduler to pick the task that has not yet finished running to avoid a cascade of critical subjobs.

3.4.2.2 Group deadline

Group deadlines only make sense for heavy tasks. Any non-heavy tasks will have a group-deadline of 0. The group deadline will cause the scheduler to pick the heaviest of the two heavy tasks being compared at a tie-break. It works by looking at a set of subjobs, defined as a *group*. This group can then be scheduled almost as a single entity, and the group deadline is used to compare two subjobs from two groups. Light tasks will have a single group, namely the entire sequence of subjobs, and the group-deadline for such tasks will be 0 (forcing light tasks to be picked before heavy).

$$D(\tau_{i,k,l}) = \min\ u\ ::\ u \geq d(\tau_{i,k,l})\ \text{and u is a group deadline of}\ \tau_{i,k,l} \qquad (3.8)$$

See the pseudocode-listing 4.1 on page 47 for an example to how this value is found.

3.4.2.3 Comparing two tasks

When comparing two pfair-tasks ($\tau_{i,k,l}$ and $\tau_{j,k,l}$) to the PD^2 rule, the following rules are used to determine the relative priority between the two:

1. Start by comparing sub-deadlines:

$$\tau_{i,k,l} > \tau_{j,k,l} \text{ if } d(\tau_{i,k,l}) < d(\tau_{j,k,l}) \tag{3.9}$$

   Subjob $\tau_{i,k,l}$ will be picked if its sub-deadline is smaller than $\tau_{j,k,l}$.

2. If the sub-deadlines equal, compare disjoint bit:

$$\tau_{i,k,l} > \tau_{j,k,l} \text{ if } b(\tau_{i,k,l}) > b(\tau_{j,k,l}) \tag{3.10}$$

   $\tau_{i,k,l}$ will be picked if it's disjoint-bit is greater. In practice, this is when $\tau_{j,k,l}$ is the last subjob in the sequence and $\tau_{i,k,l}$ is not.

3. Compare the group deadline to see which task is the heaviest and closest to its deadline.

$$\tau_{i,k,l} > \tau_{j,k,l} \text{ if } D(\tau_{i,k,l}) < D(\tau_{j,k,l}) \tag{3.11}$$

All of these tests can compare values already computed, avoiding expensive list-traversals at run-time. If $\tau_{i,k,l} = \tau_{j,k,l}$, it does not matter which task is picked as they are equal in all respects. See Listing 4.2 (p. 49).

3.4.3 Reweighing

Allowing for dynamic change of weights makes the system adaptive. The theory is two-fold. One approach allows for dynamic reweighing, that is between sub-jobs, another is between job-releases. The latter is straightforward and need only verify that the change of resources is feasible. The former is a bit more involved as it must keep track of remaining subjobs and map these to the new set of subjobs. Because we cannot do fractions of subjobs, this can lead to situations where a reweighing can add almost 2 subjobs extra time for a task.

Then, the scheduler can reweigh as well. This is the same as if *every* task were to change its WCET requiring a different number of subjobs. However, the impact is large, and this is not allowed at the moment. If the scheduler wants to change the timeslice, it can only do so if *no* tasks have requested to enter SCHED_PFAIR.

3.4.4 Constraints and limitations

PD^2 offers a global, optimal approach to pfair scheduling. ERfair offers better utilization of the available time as it can allow tasks to be released early, but also introduces higher computational overhead at run-time. For this reason, ERFair will not be considered further in this project.

3.5 Summary

This chapter introduced the theory behind the pfair algorithm and discussed various approaches to multicore scheduling. Pfair is a global algorithm that executes tasks at a steady rate, thus minimizing the bin-packing problem found with partitioned schedulers, and avoiding *Dhall's effect* which affects global, multicore EDF schedulers.

Part II

Design and Implementation

Chapter 4

Overall Design of the Scheduler

"The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!' but 'That's funny...' "
— Isaac Asimov

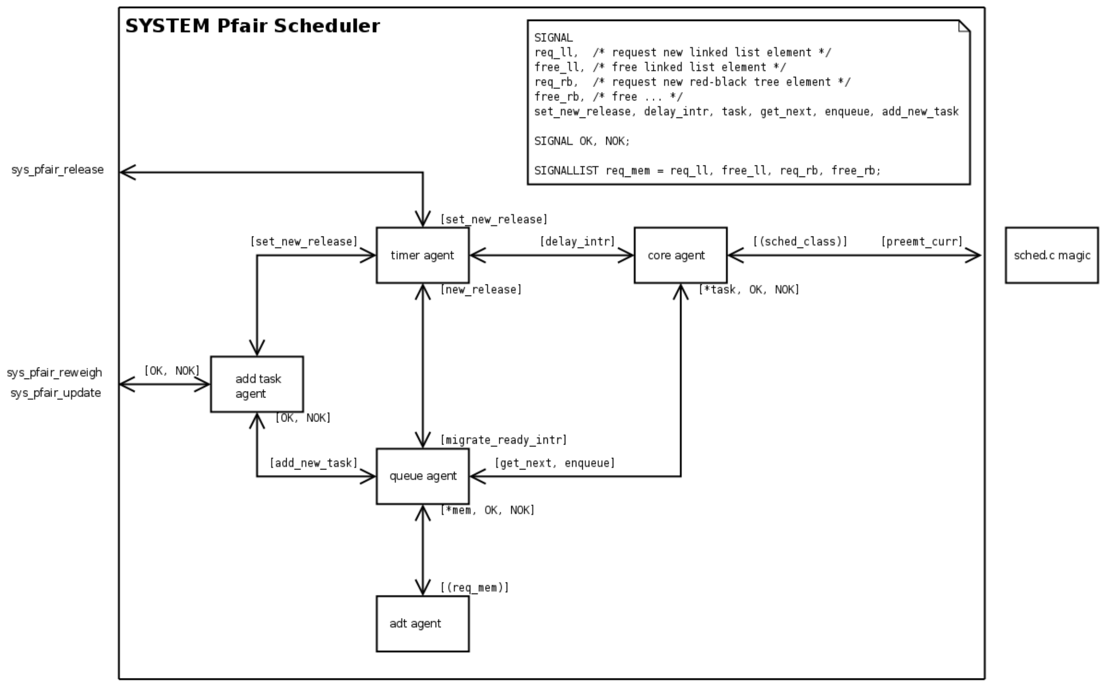
The motivation for this project was outlined in Ch. 1. Based on that, the chapters following laid down the foundation for the implementation. In Chapter 2, the Linux kernel was introduced, and in particular, in Sec. 2.2 the current scheduler was described. The theory behind this was presented in Ch. 3.

In this chapter, the goal is to present the layout of this schedulers. Sec. 4.1 describes a high-altitude overview of the scheduler. Then Sec. 4.2 follows by presenting how the tasks are handled, and then moves on to discuss how these tasks are stored in the runqueue in Sec. 4.3. The pfair scheduler is made available to user-space via a set of syscalls and SysFS entries. attributes. This is discussed in 4.6. The last major component of the scheduler is the way the main scheduler utilizes `struct sched_class` and a few other odds and ends that must be included directly in `sched.c` in Sec. 4.5. Section (4.7) describes how the kernel build system should be modified in order to add the new scheduler to the kernel according to current conventions. The chapter is then concluded in 4.8.

4.1 A birdseye view

Early on, the original plan to adopt a pure EDF scheduler was dropped due to the fact that the optimal properties of EDF breaks down on a multicore system [4]. The choice of a rate-based scheduler was found based in a series of articles ([3], [1], [21], [24], [2] and [6]).

One difficult task with all scheduling algorithms, but especially with a deadline-driven algorithm in a kernel almost entirely based on (static) priorities, are deadline inversion. As the algorithm of choice is global, an elegant solution this has been found. The proxy execution protocol is used as a basis and adapted to the global landscape. This is described further in 4.4.



```
SIGNAL
req_ll,  /* request new linked list element */
free_ll, /* free linked list element */
req_rb,  /* request new red-black tree element */
free_rb, /* free ... */
set_new_release, delay_intr, task, get_next, enqueue, add_new_task

SIGNAL OK, NOK;

SIGNALLIST req_mem = req_ll, free_ll, req_rb, free_rb;
```

Figure 4.1: Simple SDL diagram for scheduler agents with roles

Figure 4.1 gives a short overview of the scheduler. To outline the different sections of the scheduler, the term *agent* has been used. This is simply to split the behavior into different *roles*, but also to distinguish between what happens at timer-interrupts and what the task is responsible for doing. The scheduler does not contain agents, but it does implement the roles. At the core of this, is the timer agent. It will provide a heartbeat functionality with timeslice granularity, and is the main input to the pfair engine. When a new task is added, it is inserted

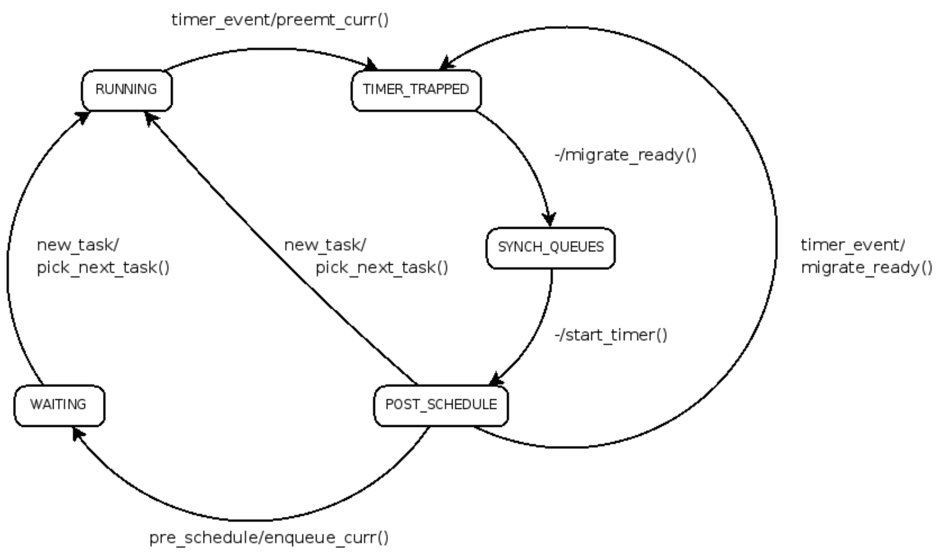


Figure 4.2: State machine for the "timer-agent" role in the pfair scheduler

into the runqueue, and the timer-agent is notified about the new task. Steps are then taken to ensure that when the task is released, a time-event will force this to happen.

The normal state, is RUNNING. When it is in RUNNING, there will be at *least one* task in the active queue. When active is empty and ready is non-empty, it will wait in POST_SCHEDULE. When a timer event occurs, the currently running task `(curr)` is marked for preemption. We have the following set of scenarios:

- `curr` is pfair task and has exhausted its timeslice and there are tasks left in active. The setup is straightforward: put current in ready and pick the next form active.
- `curr` is pfair, but active is empty. `curr` has exhausted its timeslice and should be preempted. The timer-logic will find the next release-time and then wait in POST_SCHEDULE.
- `curr` is not a pfair task. The event can only mean that a task in ready is ready to be released. It will migrate tasks from ready, preempt current and wait for a pick_next event.
- `curr` is not pfair and both queues are empty. This is a situation that can arise when several CPUs try to migrate the same task. This race-condition is actually quite difficult to handle and in the current version, it is accepted as a performance penalty as only non-pfair tasks are affected.

The details of the timer architecture discussed further in Ch. 8.

A note of warning: systems incorporating simultaneous multi-threading (like Intel's HT-technology) must be configured with this option disabled for obvious reasons.

4.2 A pfair task

All tasks in the kernel are represented by a `task_struct` found in `include/linux/sched.h` . In this struct, all resources held by the task, information about scheduling class, priority etc. are stored. Each task also contains a `sched_entity` and `sched_entity_rt` , which is what `sched_fair.c` and `sched_rt.c` use to schedule tasks. As one goal was to keep the complexity as low as possible, the current version does not create a new scheduling entity but instead embed the required fields directly.

4.2.1 New attributes in the task descriptor

All attributes concerning time, is represented as `u64` [1], and in nanoseconds. Ideally, `ktime_t` should be used, but it makes debugging easier, and some of the arithmetic simpler by not adding yet another mathematical library in the code.

`sched.h` :

`u8 pfair_state` Which state the task is in. This can be either PFAIR_READY or PFAIR_ACTIVE. The flag PFAIR_RUNNING will also be multiplexed in (bit position 1) to indicate if the task is running on a CPU or not.

`u8 pfair_periodic` The task can be either periodic or sporadic.

`struct pfair_subjob` Added twice in the form of `psj_head` and `psj_curr.` They are pointers to the head of the subjob queue as well as the current subjob (or the next to run if state is PFAIR_READY).

`struct rb_node task_node` Used in the runqueue to store the task, either in the ready or active runqueue.

`u64 period_ns` This is the base period for the task and is set when the task is updated to pfair-status. If the task is periodic, once the last subjob has finished, it will be released at `last_releasetime + period_ns` . If it is

[1] `u64` is a way of describing `unsigned long long` values, and also force the compiler to make it 64 bits, regardless of the architecture.

sporadic, it will not be allowed to be released until at least `period_ns` has passed since `last_releasetime` .

`u64 deadline_ns` The deadline dictates the dynamic priority of a task, but also how much resources that must be reserved for the task (as we will see later in Sec 4.2.3).

`u64 wcet_ns` The **W**orst **C**ase **E**xecution **T**ime (WCET) is an estimate at best as determining this value is very difficult, but in order for deadlines to have any meaning, an estimate of the execution cost must be provided. This will also determine the number of subjobs allocated to the task.

`u64 scaled_wcet_ns` To make sure the utilization does not increase beyond what is feasible, a scaled WCET is used from all tasks to provide a approximated utilization measure. This is wcet scaled up to fit into the hyper period in the runqueue. The `hyper_ns` is described in the runqueue section (Sec. 4.3), and so is the calculation of this value.

`u64 abs_release_ns` This stores the absolute release-time for the current subjob. It was introduced to minimize the need for computing. Each time two tasks in state PFAIR_READY are compared, the release-times for the current subjobs are used.

`u64 last_release_ns` Not to be confused with `abs_release_ns` , this value stores the last release of a *job*. The release-times for each subjob is computed when the task is either updated or reweighed and are stored as relative values in the subjobs. To find the absolute releasetime (or deadline) for any given subjob, the relative time added with this will give the absolute system time.

`u64 no_subjobs` The number of subjobs the task has. This is useful when the task is reweighed, as finding how many subjobs to add or remove can be done quickly.

4.2.2 Adding subjobs

The subjobs belonging to a task is the entities the pfair scheduler actually handles as elements. In itself, a subjob is not a separate task, but rather a token (or *ticket*) the job[2] holds to show the scheduler how it should be handled. The values in the subjobs are computed when the task is either updated or reweighed.

[2]A job is an instance of a task, when a task is *released*.

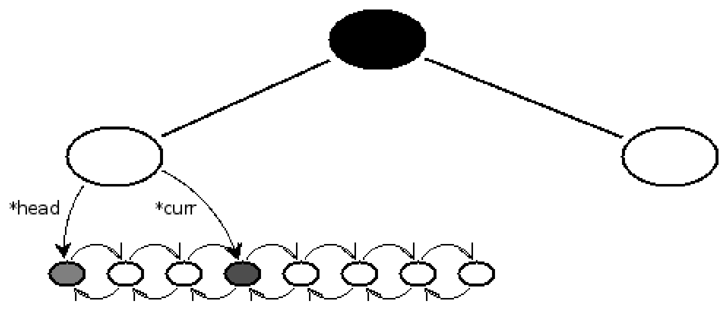


Figure 4.3: A job is stored in a tree, and the subjobs are used to sort the task to the correct place.

Elements in the `pfair_subjob` :

- `struct list_head list` : The linked-list element. This will be covered further in 6.1.
- `u32 window_length_ns` : Not to be confused with the window-length in the pfair run-queue. This value indicate the length from sub-release to sub-deadline for this subjob.
- `u32 consumed_time` : How much time has been granted to the subjob upon preemption. As a subjob can be temporarily suspended at interrupts etc, this is intended to correct for that. However, this has proved to be infeasible to conduct in practice, and the value should be removed.
- `u64 sub_release_ns` : The relative release-time of this subjob with respect to `abs_release_ns` in the task-descriptor.
- `u64 sub_dl_ns` : The relative deadline. The absolute deadline is found by adding `abs_release_ns` .
- `u64 group_dl_ns` : Exactly as the `sub_dl_ns` value.
- `unsigned char disjoint_bit:1` : The last element to complete the PD^2 rules.

All the subjobs are kept in a linked list, sorted increasingly by the release-time. As the deadline of a subjob cannot exceed the release of the next, it is also sorted by the sub-deadlines. The choice fell on linked lists as this structure provides a way of changing the size dynamically in a very easy manner. Since we keep the current subjob (or the next to be released), finding the next subjob can be done very efficiently. The kernel implementation also uses memory prefetching to bring in the next element whenever possible. This makes traversing the lists very efficient, and incrementing a single node can in most cases be done without going to memory a second time. The linked list is described in Ch. 6.1.

Any single task with a reasonable large wcet, will end up with a large number of subjobs. Add several tasks, and you suddenly get a very large number of subjobs. All these require memory to be allocated (and deallocated), with as little memory wasted as possible. The memory management layer, and especially by the SLAB allocator, handles this. This is presented in full in Chapter 5.

4.2.3 Calculation of subjob values

All the values in the subjobs, are found according to the equations described in Sec. 3.4. The original algorithm calls for 3 passes over the entire list, first to update the release-times and deadlines, then to set the disjoint-bit and finally to find the group-deadlines for the subjobs. The need for doing this 3 times (2 if you take the disjoint-bit and group-deadline together in one pass) is because the values depend on the next value in the list. However, if you change the direction you traverse the list, you can do this in *one go*. Unfortunately, this leads to a bit more complex code, as seen in listing 4.1. The actual computation can be found in `kernel/sched_pfair.c` in the source code. At the moment, the calculation is preformed in the `__pfair_subjob_calc()` function so both the update and reweigh-functions can utilize it.

Listing 4.1: *Calculation of the subjob values*

```
job_nr = no_subjobs
foreach element in list reverse:
    sub_release[job_nr]  = floor((job_nr -1)*job_period / wcet)
    sub_deadline[job_nr] = ceil(job_nr*job_period/wcet) -1
    if last_subjob:
        sub_disjoint[job_nr] = 0
    else
        sub_disjoint[job_nr] = 1
    if task_is_heavy
        if last_window_length == 3 and this_window_length == 2
            last_groupdl = this_subdl
        sub_group_dl[job_nr] = last_groupdl
    job_nr - 1
```

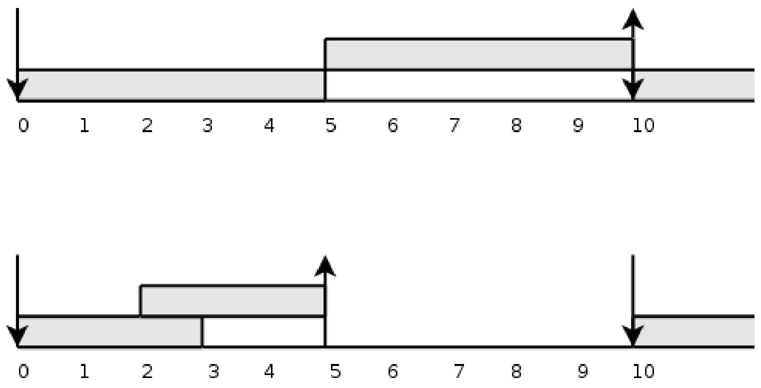


Figure 4.4: Two schedules. The first with deadline equal period, the second with a much shorter deadline, resulting in a much “tighter” schedule.

This might seem like a non-trivial way of finding the subjob values, as the actual release time must be computed from the absolute releasetime of the job and added with the subjob release time. However, it has one very important feature: when a job is released, regarding of the number of subjobs, only the releasetime for the job needs to be updated. This provides O(1) release-time overhead, something that is *very* important in a real system. In other words, optimize for the common case.

The resulting values can be added graphically as shown in Figure 4.4 for two different tasks.

4.2.4 Comparison of tasks

A normal scheduler compares the relative importance between two tasks by looking at their priorities and conducts a pure numerical comparison between the two. With deadlines, and especially with pfair, this comparison becomes more complicated. In particular, the compare-function is needed when a new task is going to be inserted into either one of the runqueues. Both trees (see Sec. 4.3 for how the scheduler keeps the tasks in the runqueues) are kept sorted based on properties in the tasks.

The task can be in STATE_READY, meaning it has released a job, but no subjob is currently ready to run. The other is STATE_ACTIVE meaning that a subjob is ready to run, it has been *released*. Looking at the ready-state first, the comparison is straightforward. Take the absolute release-time for the current subjob in and compare with another, similar task. PFAIR_ACTIVE is a bit more involved, and this is the actual PD^2 rules:

1. Compare the absolute deadlines and pick the task with the shortest deadline.
2. If the deadlines match, test the disjoint bit. If they differ, the task with the bit set "wins".
3. If they both have equal disjoint-bit, compare the group deadline.

Listing 4.2: *Comparison of two pfair tasks*

```
PFAIR_COMPARE_TASK(t1, t2)
    if t1 is PFAIR_READY
        return release(t2) - release(t1)

    if sub_deadline(t1) == sub_deadline(t2)
        if disjoint(t1) == disjoint(t2)
            return group_deadline(t2) - group_deadline(t1)
        return disjoint(t2) - disjoint(t1)
    return sub_deadline(t2) - sub_deadline(t1)
```

Optimizations can be done here. For instance, if both have cleared the disjoint bit, they are both the last job, and hence the group deadline will also be equal. This is not done in the actual implementation as it would lead to a more complex comparison and the gained speedup is negligible compared to transversing the tree.

4.3 The pfair runqueue

The kernel runqueues are per- CPU, but the pfair runqueue (`prq`) is global. Each `rq` has a reference to `prq` . As mentioned in 1.1, *resource reservation* is used to ensure that tasks requirements can be met. Since the kernel does not support floating point arithmetic, we need to approximate this. By using a large hyper-period and then scale all tasks to approximately this length, the estimated cost for an entire hyper-period can be found. This can then be added to `consumed_time_ns` to store the total, reserved resources thus far.. This is an less-than ideal approach, but it is fast and fairly accurate.

4.3.1 Added fields to the standard runqueue

In the kernel, each CPU is given a dedicated runqueue, and a set of values have been added here:

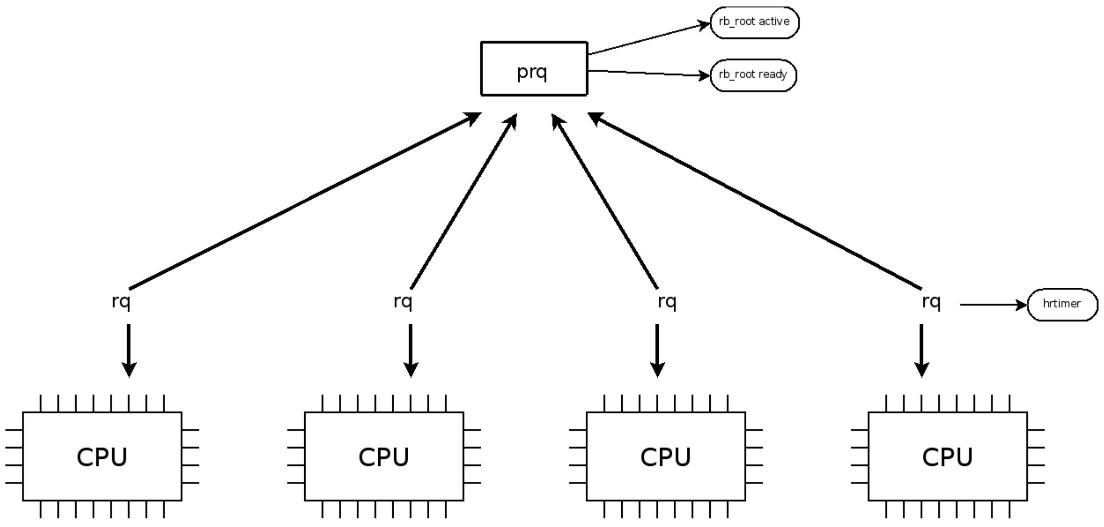


Figure 4.5: The relation between the local runqueues and the global pfair runqueue, and some of the important fields in both.

`struct pfair_rq *prq` This is a reference to the global pfair runqueue. The reason for adding this to the local runqueue, and not as a global, static value, is that we want to retain the option of dividing a domain into several "sub-domains".

`struct hrtimer pfair_timer` Each CPU need to have a dedicated timer. At regular intervals, the timer will generate events to trigger a rescheduling of tasks. This will enforce the timeslice granularity for the tasks, and also make sure that tasks are released when they are supposed to be.

`unsigned int pfair_enabled:1` If pfair is enabled or not. This can be used to turn off the scheduler.

`unsigned int pfair_running:1` Indicates whether or not the scheduler is running (if it has any tasks in any of the runqueues).

`unsigned int timer_enabled:1` Flag used to indicate if the timer is enabled. When new tasks arrive, this is used to quickly determine if the timer should be reset (an expensive operation), or if the timer can be started blindly.

`struct task_struct * pi_resource_holder` This is the task holding the resource we are currently blocked at. This is for implementing PEP.

4.3.2 Values in the pfair runqueue

The pfair runqueue holds a set of variables used to manage the behavior. This list describes the elements, and how they are used in the kernel.

`spinlock_t lock` To protect the structure from concurrent access, it has been given a lock. It is a simple spinlock, as the expected time of holding the lock will be short, and we really do not want to block while trying to schedule a new task.

`u32 pfair_running` Flag that can be modified via the SysFS interface to turn the scheduler on and off. This is to provide a flexibility for the system administrators.

`struct rb_root active` The runqueue for the active tasks in the system.

`struct rb_root ready` While the jobs wait for the next subjob to be released, they are kept in the ready-queue.

`u32 active_count` Instead of traversing the entire tree in order to find the number of tasks, this is a quick path to obtain this. It is updated upon task migration (from the ready-queue to the active and vice versa).

`u32 ready_count` Same as `u32 active_count` .

`u64 next_release_ns` In order to quickly determine when the timer will fire, this is a value holding the next timeout value. By looking at this, the system can determine if the current timer should be canceled or if it provides adequate service.

`u64 hyper_ns` When accepting new jobs (and allowing tasks to reweigh), the utilization must be found. The hyper_ns is the length of a hyper-period, and is chosen to be larger than most periods. By scaling the period of a task up to this size, the multiplier can be used in conjunction with WCET to create an estimate of the expected load for the task in the hyper period.

`u64 consumed_time_ns` This is the combined, scaled WCET for all the accepted pfair-tasks in the system.

`u32 timeslice_ns` The length of each time quanta given to the jobs.

`ktime_t timeslice_delay` A `ktime_t` representation of the timeslice. It can be computed quickly whenever needed by `ns_to_ktime(time_in_ns)` , but it is easier and more efficient to store it directly as this is a value that does not change often.

`u32 busy_limit_ns` If a job finishes before the current timeslice is exhausted, other non-pfair tasks can be allowed to run. However, if the remainder of the timeslice is below this limit, it does not make sense to change the tasks, as a preemption would remove the new task before it is allowed to run. This attribute is used to tune the performance in such settings, telling the scheduler to spin instead of schedule out a new task.

4.3.3 Storing tasks in the runqueues

As shown in Fig.4.5, the pfair runqueue has **two** runqueues, one for active tasks, and one for tasks waiting to be released. Motivated by the CFS, the usage of red black trees was adopted. Since the runqueues are global, these structures can potentially be required to hold a large number of tasks. It is therefore very important that insertion, removal and balancing of these trees can be done efficiently. The usage of red black trees are described in Chapter 6.2, and they are kept sorted by using the compare-function introduced in 4.2.4.

4.4 Solving the deadline inversion problem

Whenever there are shared resources in a system, there will be lock contention, race conditions and possible deadline misses. The kernel must make sure requested resources are freed as quickly as possible. The need for locks are obvious, a given resource must be protected from concurrent access. The nature of the lock can be either short-term, e.g. a *spin-lock* or long-term, e.g. a mutex (a binary semaphore) where the task may suspend awaiting release of the resource. The problem arises when you have (at least) 3 tasks in the system, A, B and C. A has the highest priority, C the lowest. C holds a shared resource R which then A requires. A is then suspended waiting for C to release R. Now B becomes runnable. Since B has higher priority than C, B is allowed to run, blocking *both* C and A.

Several ways of solving this have been proposed over the years, where the Priority Inheritance Protocol (PIP) is one of the better. In PIP, when A is blocked by C holding R, C is given A's priority until R is released. That way, C will be allowed to run even if B becomes runnable, speeding the release of R. This works fine for systems based on priority, and recent work has extended this to the realm of deadlines [11]. BandWidth Inheritance (BWI) works by extending the allocated resources given to a task, to the holder of a lock. Each task has a period, execution cost and deadline. They are given a certain portion of the total bandwidth, namely $\frac{execution\ cost}{period}$. Then, when a task holds a resource and depletes its budget, it will tap into another, requesting task's budget until the resource is freed.

In our setup, we use a slight variation of BWI called the proxy execution protocol, PEP. The difference between BWI and PEP is that in BWI, the holders budget is depleted first, and once empty it will run on the A's resources. In PEP, C will run as normal, using its own budget. When A attempts to acquire the lock, it blocks, but it is *not* removed from the runqueue. Instead, it is treated as any normal task. When A is scheduled to run, C will run instead, but using A's budget. This gives an added bonus to C for holding something A wants. In BWI, a task would be penalized for holding a wanted resource.

It is at this point, that we can see the pfair-scheduler excel. Simply because we have total control over *all* tasks at *all* time. By adding a memory reference to the task holding the resource, the scheduler can automatically trace down the path until it finds the task eligible for execution. If the task is already running, it can deduce this from the `pfair_state` flag. The time allocated to the holder of a resource on behalf of the requester is then accounted for on the requestors budget (meaning task A will pay to allow C to run so C will release R faster). This compartmentalize the extra resources given to C to the combined size of A and C's budget.

As a final note. If C is *not* a pfair-task, it will still run in a pfair context. This means that *every* task in the kernel can hold the resource, and because the resources granted will be accounted for on A's budget, it does not matter if C uses normal priorities, the problem is reduced to picking another *task*, not *type of task*. This means that PEP in effect can be extended to span *all* tasks in the system, not only the pfair class. At the moment, it is of considerable interest to implement this in the Linux kernel, especially in conjunction with the RT-preemption patch and its rt-mutexes.

4.5 Main scheduler interface

All scheduling classes are required int implement all (or a large subset of) the functions found in `struct sched_class` . This allows the kernel to call a set of functions without knowing which class it belongs to. In some cases, this restrict what a scheduling class can do, but in most cases, it makes the job of integrating a new class much simpler. The most important functions are also covered in Ch. 9.

4.6 Adding a gate to the kernel

To update a task from any scheduling policy to pfair, the `__sched_pfair_update()` function will do this. It is mapped to the syscall `sys_sched_pfair_update()` in `sched.c` . This is needed in order to allow tasks to gain pfair privileges. A set of tests are performed (capability, validity of parameters and whether or not the scheduler can serve the request) before the task is allowed into the pfair runqueue. When a task has gained pfair status, it must be *released* in order to be granted time on the CPU. This is either done automatically (if the task is periodic), or the task must call `sched_pfair_release()` itself. Either way, the task will not be allowed to run before a time equal the period has passed since last release. Finally, the scheduler allows a task to change its parameters. This is called reweighing and is made available via `sched_pfair_reweigh()` . The technical details around implementing syscalls are covered in Ch. 7.2.

To tune the scheduler, several attributes have been added to the /sys directory. These are covered in Ch. 7.

4.7 Building the kernel

When the kernel is going to be built, it is an absolute requirement that the pfair-scheduler can be enabled or disabled via the kernel configuration system. By wrapping the pfair-related code in " `#ifdef CONFIG_SCHED_PFAIR` ", the compilation will result in no pfair-related code[3] with *no* added code or run-time overhead. This, and some of the other options and tunable values are covered in Ch. 10.1.

4.8 Summary

This chapter gave an outline of the new scheduling class, and the new scheduling policy SCHED_PFAIR. A thorough discussion about the new fields added to both the runqueue and the task descriptor was given. The timer state-machine was also covered and explained. The next chapters will delve into how the different subsystems work, and how the scheduler uses these. Each system has also been tested with a dedicated module, and those are also covered. Where this chapter gave the outline and general idea, the following chapters will provide the nitty-gritty details.

[3]Except the syscall-wrappers.

Chapter 5

Memory management

"Memory is necessary for all the operations of reason."
— Blaise Pascal

Until now we have only looked at background, theory and design. The scheduler requires several specialized datastructures, most of which requires memory to be allocated dynamically.

In user space, memory is easy[1], but once you enter kernel context, things get a bit more hairy. For one, the *entire* kernel shares the same address-space. Where each user-land thread has its own memory space, the kernel threads share theirs. A memory error can therefore affect other, completely unrelated parts of the kernel, making tracing bugs very difficult. Often the kernel cannot sleep or suspend while waiting for memory either, which makes swapping other pages out a bit more complicated.

In this chapter we will at how memory is allocated with `kmalloc` and freed with `kfree` . We then proceed to look at the SLAB allocator and why this is a good choice for some of the datastructures in the pfair-scheduler.

[1] "Easy" must be taken with a grain of salt as C and memory has always been, and will always be, an error-prone area.

5.1 Background

Memory is treated as pages of memory. Each page is normally 4kB or 8kB depending on the underlying hardware. Both `kmalloc` and `kfree` , and then also the SLAB layer, use the buddy system to obtain and release memory. The buddy system is a system used for managing large areas of memory. The details around this is omitted, as this is beyond the scope of this project ([19, Ch. 11] and [7, Ch. 8,9] gives a very thorough discussion about memory management, allocation and implementation).

5.1.1 `kmalloc` and `kfree`

`kmalloc` is used to allocate memory (kernel memory allocation), and `kfree` is its counterpart, it will free previously allocated memory. Both `kmalloc` and `kfree` are found in `include/linux/slab.h` .

5.1.1.1 Allocating the correct size of memory

As the buddy system deals with page frames, it cannot handle variable length blocks. As it would be redundant[2] for `kmalloc` to implement direct memory management, this is left for the buddy system. To waste as little memory as possible, `kmalloc` maintains a table of blocksizes (see Table 5.1), and when a new request for memory is issued, `kmalloc` (and in turn, the buddy system) will try to return the nearest fitting block of memory, rounding *up* in size.

5.1.1.2 Allocating the memory at the correct time

When allocating memory, `kmalloc` must be given a flag to indicate the type of memory. We will not cover this in depth, other than presenting the two (composite) flags most often used:

GFP_ATOMIC is high priority allocation that cannot block. If other pages must be swapped out before free memory is returned, `kmalloc` will fail and return `NULL`. When allocating memory from regions that cannot sleep, i.e. interrupt context or other time-critical sections, this is the preferred choice. However, it will fail to allocate memory long before GFP_KERNEL.

GFP_KERNEL is the "normal" allocation flag. `kmalloc` will block the issuer if other pages needs to be swapped out. This is the most common flag when allocating memory in the kernel.

[2] "Redundant" translates directly into increased complexity and less maintainability.

list	bit width	memory size	list	bit width	memory size
0	5	32 B	6	11	2,048 B
1	6	64 B	7	12	4,096 B
2	7	128 B	8	13	8,192 B
3	8	256 B	9	14	16,384 B
4	9	512 B	10	15	32,768 B
5	10	1,024 B	11	16	65,535 B
			12	17	131,072 B

Table 5.1: List of memory blocks handled by `kmalloc` . New memory will be granted, rounded up to the nearest available block and managed via the buddy system

5.1.1.3 Freeing memory

This is done by `kfree` and will free a memory region allocated by `kmalloc` . The kernel will not try to monitor which pages belongs to which threads so it can deallocate memory belonging to a dead thread. To avoid memory leakage, all allocated memory should be freed when not in use anymore.

5.1.1.4 Using `kmalloc` and `kfree`

To allocate a memory area, `kmalloc` is used. It will return a memory pointer (to `void`) or NULL upon failure:

```
void * kmalloc(size_t size, int flags)
```

To free memory, `kfree` accepts a pointer to a memory region. Note that kfree is “NULL-safe”, but it is not advisable to free the same memory twice, or memory not allocated by `kmalloc` :

```
void kfree(void *mem_ptr)
```

5.1.2 The SLAB allocator

When allocating a large number of objects[3], `kmalloc` can, and probably will, waste a lot of space. As a result, developers often devote a small part of their larger system to managing memory. By allocating a large block of memory they can organize the memory in such a way that only a small amount is wasted. Unfortunately, this code tends to grow in complexity and steals focus from the more important task — namely the purpose of the project at hand. For this reason, the SLAB layer was invented. Originally found in Sun's Solaris, the SLAB layer is a way of allocating a large amount of memory in *caches* and return just the memory needed when asked for. Each *cache* is divided into a set of caches, and each cache allocated one or more physical pages of memory.

Frequent allocation and deallocating of objects The layer has a memory-*cache*, and when a new object is allocated, the layer first attempts to acquire the object from the cache. If unsuccessful, the layer allocates a large *slab* of memory, and then returns just the correct amount of memory from this region. Deallocation returns the object to the cache, so that a new allocation-request can be served from the cache.

Centralized free-list implementation As the kernel has a single system for a free list, the kernel can monitor the SLAB and optimize memory usage.

Cache line aware By using coloring of the elements, the allocator can minimize the cache conflicts between objects. This helps reduce cache thrashing. Another benefit of allocating objects as tightly together as possible, and not in areas with power of 2 size, is to reduce the address collisions.

Optimal usage of available memory The layer can be told the size of the objects to allocate. This allows it to optimize for memory footprint, reducing the wasted memory greatly.

Multiprocessor aware The slab layer has a per-CPU cache as well, called the "slab local cache", a cache for each CPU.

The SLAB-layer (Fig 5.1, p. 59) creates one cache for each object, and each cache is composed of one or more slabs. The slabs are one (or possibly several) page of memory. A slab can be *empty*, *partial* or *full*. When a slab is empty, no objects have been allocated from the slab. A partial slab has 1 to $n-1$ allocated objects, where n is the maximum number of objects a slab can allocate).

[3]When talking about objects in the kernel, one normally indicates an allocated instance of a `struct` for a particular purpose.

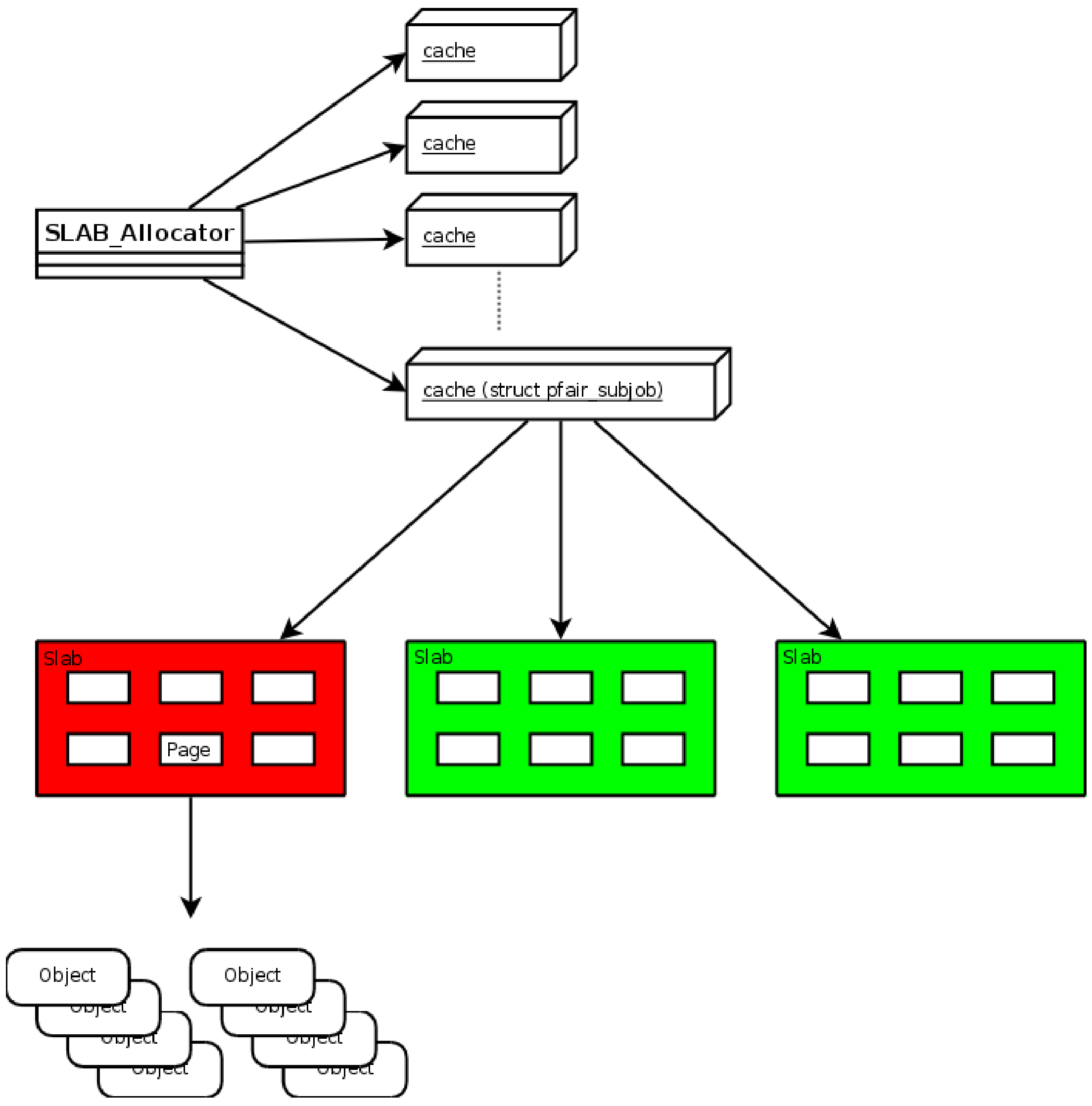


Figure 5.1: The SLAB layer with a set of initialized caches, and the pfair cache consisting of 3 slabs, each with 6 pages. Red indicates full, orange is partial (not shown) and green is empty.

The whole point with the cache, is to store unused objects instead of directly freeing them. When objects are allocated and deallocated frequently, this or when the object has a size different than the power of 2 list found in `kmalloc`.

5.1.2.1 SLAB, SLUB and SLOB

SLUB is an improvement to SLAB. Over the years, the SLAB layer has proved its worth in the kernel. It is quite complex, and some kernel hackers found it to be insufficient. As a result, the *SLUB* allocator was proposed, a replacement for SLAB. SLUB has several improvements, most notably removal of several of the queues and no metadata in the cache it self (it is kept in the allocator itself, not inside the cache areas). Instead of an array of objects, SLAB uses a linked list of free objects. When a request is made, the first available object is returned (the head of the list). If the slab is full, a new page is requested, and an object is returned from the new page.

The SLOB allocator is a very, *very* simple allocator to use in small embedded systems. SLOB does not try to do cache alignment, coloring or other cache techniques. This makes it lightweight, but it suffers from internal fragmentation as it does not have a sufficiently advanced allocating algorithm.

The allocator is chosen at compile time. The API is the same (hence the "drop-in replacement"), the difference is how memory is treated. The layer is still denoted "the SLAB Layer", and the main API-file is `include/linux/slab.h`. Depending on the kernel configuration, `slab.h` will include either a) `slab_def.h` b) `slub_def.h` or c) `slob_def.h` and since Linux-2.6.22, SLUB has been the default allocator.

5.2 Kernel interface

As the startup routines in the kernel will initialize the SLAB Layer, we will not cover direct SLAB Layer manipulation here. We will instead look at how to create, use and destroy a SLAB cache. This section tries to explain the parameters. The usage can also be viewed in App. B.1.

5.2.1 Cache create and destroy

To create a slab cache, a pointer to the cache is needed. The type is `kmem_cache_t` and is returned by `kmem_cache_create`. The create function takes the following set of arguments:

`const char *name`
: A string with the name of the cache.

`size_t size` The size of each element (object) in the cache.

`size_t align`
: Offset of first object in the slab. This is to change the initial alignment within the cache.

`unsigned long flags`
: (optional) This sets the cache behavior. No particular flag must be set. The complete list can be found in [17, line 19-24]. To increase performance, the cache can be told to align the objects to different cache-lines (SLAB_HWCACHE_ALIGN or SLAB_MUST_HWCACHE_ALIGN).
This will increase performance, but memory is wasted as the allocator adds empty regions if the objects align.

`void (*)(void *)`
: Used to be constructor for the cache, in case special operations need to be performed. As no one actually needed this reference, it is moving towards deprecation, and thus, the empty function signature. This should always be set to NULL.

On success, `kmem_cache_create` returns a pointer to the allocated area. This function can sleep, so a cache should not be implemented from an ISR.

To destroy a cache, `int kmem_cache_destroy(kmem_cache_t *cache)` can be used. The result is almost identical to `kfree`. Note that all objects allocated from the cache *must* be returned before the cache can be destroyed.

5.2.2 Allocate memory for an object

As with `kmalloc`, the `kmem_cache_alloc()` takes a flag as argument. The flag is the same as for `kmalloc`, and in effect, either GFP_KERNEL or GFP_ATOMIC is used. The difference in the function signature, is that `kmem_cache_alloc()` takes the slab-cache pointer as argument, whereas `kmalloc` takes the size.

```
void * kmem_cache_alloc(kmem_cache_t *cachep, int flags)
```

5.2.3 Deallocate, return to cache

As with `kfree`, a pointer to the allocated object must be provided, and since we are working with the SLAB-cache, a pointer to the correct slab must also be provided:

```
void kmem_cache_free(kmem_cache_t *cachep, void *objp)
```

5.3 SLAB kernel interface

The SLAB Layer adds complexity to the scheduler, no questions asked. So *why* use the SLAB Layer when, if datastructures are devised carefully, `kmalloc` performs just fine? The pfair scheduler will create a set of *subjobs* for each task, and all of these will be chained together in a linked list (see Ch. 6.1 for how linked lists are implemented and used). The subjobs were presented in listing 4.1 on page 47

A short list of reasons to why the SLAB Layer is a good choice:

a) A large number of *pfair subjobs* will be allocated and eventually deallocated, making the SLAB Layer an ideal choice.

b) SLAB is fast. When we initialize a pfair task, we want to wait for as short time as possible.

c) It will reduce memory fragmentation. Each subjob is a small struct, containing only a few variables.

The cache will be used when a task is either *updated*, i.e. changes scheduling policy to SCHED_PFAIR or when a reweighing changes the number of subjobs (i.e. the WCET changes).

5.4 Small test module

To verify that the layer was created and used correctly, a small module was created. It can be found in Appendix B.1 and is also included in the attached files (on the CD-ROM, `code/slab/slab.c`) To compile, insert and remove:

```
make
sudo insmod slab.ko
sudo rmmod slab
dmesg | tail
```

The usage is fairly straightforward. It is important to use `kmem_cache_free()` when deallocating objects. If `kfree` is used, the kernel will panic.

5.5 End result

In the scheduler, the SLAB layer is used whenever a new task is updated to SCHED_PFAIR. Based on the period, deadline and worst case execution time (wcet), a set of subjobs are created. The number of subjobs depend on the timeslice used by the scheduler (normally 1 ms) and WCET. All of this was shown in Sec. 4.2.3.

In the end, once SLAB was understood, the required code was only a few lines. This is as expected - anything more would have been disastrous. The challenge lies in understanding the system and using it properly, and the implemented code shows this.

5.6 Summary

In this chapter we looked at how the SLAB allocator can optimize the memory footprint and speed up frequent allocation and deallocation. This will be further used by the datastructures presented in the next chapter.

Chapter 6

Advanced data structures

"A computer program is organized complexity."
— Edsger W. Dijkstra

This chapter will present two of the advanced datastructures used in the scheduler for managing subjobs and storing the jobs in the runqueues. The chapter starts out by presenting linked lists with background, kernel API and a sample module before the role in the scheduler is described in more detail. Then red-black trees are covered in very much the same manner.

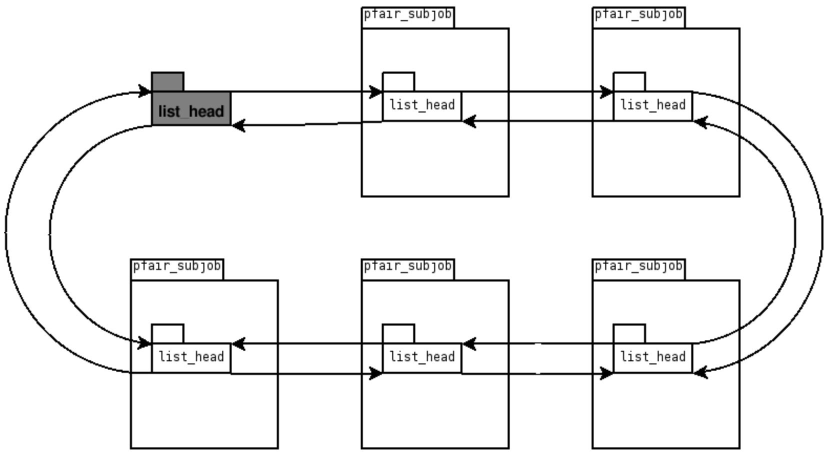


Figure 6.1: A circular, doubly linked list of 5 `pfair_subjob` elements.

6.1 Linked lists

As described in Sec. 4.2.2 and shown in Fig. 4.3, the subjobs will be stored in a linked list. Linked lists compared to arrays can reduce memory footprint as you do not have to bring the entire array into memory, you can fetch only parts of it, furthermore, you can create the *exact* length, something very difficult with arrays (pagesize etc).

6.1.1 Background

The linked list in the kernel is a doubly linked, circular list. A doubly linked list is a list where each node contain a reference to the next and previous element in the list. A circular list, is where the last element point to the first (and the first to the last if it is doubly linked).

The kernel structure for linked lists are found in `include/linux/list.h` , and in particular the struct `list_head` is the actual list-node. This is used “in place”, this means that `struct list_head` lies directly in the encapsulating data-structure, not via a memory reference. This saves the kernel an extra level of memory abstraction. By using the `container_of` macro, it is trivial to obtain the correct struct from a list_head pointer. Figure 6.1 show how the 5 subjobs are linked together via `list_head` and how they are embedded directly into the subjob.

One of the most common operations done on a linked list, is traversal. To speed this up, the kernel tries to prefetch the next list-element when fetching one.

As it is very rare for a traversal to skip one element, this can be done safely, and will in most cases speed list traversal considerably.

6.1.2 Intended role

The linked list will be used to store the subjobs for each task. As we always have a reference to the currently running subjob, finding the next subjob (when the task is removed from the active queue and placed in the ready queue, the next subjob with its release time is found) is straightforward and is done in O(1) time. This is a very important point, and this is the sole reason for choosing linked list instead of red-black trees.

6.1.3 Kernel interface

All the related macros and functions can be found in `include/linux/list.h` , the following list is the functions (and macros) used in `sched_pfair.c` . The full syntax is not given in all examples as some are nearly identical for some. The supplied module should give sufficient insight where the syntax is not shown.

- `INIT_LIST_HEAD` **:** This macro initializes a `list_head` struct so it can be used in a linked list.
- `list_entry` **:** Given a `list_head` , this will find the encapsulating struct. The syntax is similar to the `container_of` macro:

  ```
  struct your_struct *foo = list_entry(list_ptr,
                                       struct your_struct,
                                       list);
  ```

  Given the memory reference to the list, `list_ptr` embedded with variable name "list" in `struct your_struct` , the correct struct will be returned.
- `list_first_entry` **:** Given the head, this returns the first **entry** in a list. It is very similar to `list_entry` .
- `list_add_tail` **:** This will, not surprisingly, add a new element to the tail of the provided list.
- `list_is_last` **:** Test to see if the given `list_head` is the last element in the list.
- `list_for_each_entry` **:** Will iterate over a linked list in a loop and break once returned to the head of the list.

- `list_for_each_entry_reverse` : Identical behavior to `list_for_each_entry` , except for the direction.

6.1.4 Test module

In order to understand the list properly and avoid memory reference errors, a small module was created. It creates a set of list elements, keeps them in sorted order based on a primary and a secondary key (much like how subjobs will be compared between tasks) and traversed it before freeing the memory.

The result can be found in App. B.2. Even though the actual code is very small, and the usage is very basic, this small module help identify several fundamental problems and removed several flaws from the linked list implementation in the scheduler code.

6.1.5 End result

The functions dealing with subjobs are the ones using the list-functionality. A set of higher level functions have been created that handle the day-to-day list basics.

- Initializing the list. This is done in the same function that adds elements (as initializing and adding is normally done at the same time).
- Adding new elements to the list is done when it is either updated to SCHED_PFAIR or when it is reweighed. The function needs the task to add tasks to, and the number of subjobs. Note: this function does not edit the elements, it merely ensures that the given task has `total_jobs` number of subjobs available in the list.

  `__pfair_create_subjobs(struct task_struct *task, u32 total_jobs)`
- Removing elements is done when the tasks is taken out of the pfair-system (either it changes scheduling class or it stops) or it reweighs the task and ends up with fewer subjobs:

  `__pfair_free_subjobs(struct task_struct *task, u32 jobs_to_rem)`
- Traversing the list, or incrementing the curr-pointer one step. This happens when a subjob has finished and the system readies the job for the nest sub-release. The function `__advance_psj()` moves `&task->psj_curr` to the next in the list.

- Resetting the subjob-list. `__reset_psj(struct task_struct *task)` handles this by setting the `curr` reference to the first element in the subjob list. Note: a common mistake here would be to set `curr` to `head` . As we have mentioned earlier, the correct way is to use `last_first_entry()` , which will return the element `head` refers to.

These are the main subjob-handling functions. On top of this, there are functions that compare the subjobs and order the trees. As these are not directly relevant to the linked list, they are omitted here.

6.2 Red black trees

6.2.1 Background

A red-black tree is a special case of a binary tree. It is used to store comparable data in a way that makes lookup as well as insertion very efficient. The most famous attribute with red-black trees, is that they are approximately balanced at all times. A binary tree is said to be balanced if the height of the left subtree of a node is never more than $\pm$ 1 of the right subtree. Because of this, a balanced tree will never be higher than 2lg(n+1). Since the pfair scheduler is global, we must assume that a large number of tasks will be inserted in either one of the to queues. Having a data-structure that efficiently inserts and retrieves elements when the number of tasks grow, is crucial.

6.2.2 Kernel interface

The kernel red-black tree implementation is optimized for speed and reduced memory footprint. This means that it does not have all the abstract methods one would like, nor is it as easy to use at one would like. It incorporates the same basic idea found in the linked list, that the node is embedded directly in the enclosing data structure. By doing this, the kernel save one memory dereference, minimizing time spent moving data in from memory to cache.

Furthermore, there are no top-level search, insert and delete functions. The implementing module must provide these. This is simply because the red-black trees will be used in a variety of settings, making it impossible to write a general, performant API for this. What you get, however, is raw speed and unparalleled control over the data-structure.

Of all the various functions provided, only a few of the "core functions" are needed, namely:

- `struct rb_node *rb_first(struct rb_root *tree);`
  Find and retrieve the leftmost element in the tree (or NULL should the tree be empty). The rest of the logic have to traverse the tree anyway, and all that they require, is the root.

- `void rb_erase(struct rb_node *node, struct rb_root *root)`
  This will remove the supplied node from the tree starting in `root` and rebalance the tree after it is done.

6.2.3 Small test module

The red black trees are pretty much the most advanced data structure used in the scheduler. Only the hrtimer interface surpass it in complexity and size. In Appendix B.3 a small tree has been implemented. Especially the function `rbt_insert()` illustrates the point about keeping track of memory references.

6.2.4 End result

As previously mentioned, the scheduler keeps two separate queues, one for active and one for ready. Both are kept sorted based on their subjobs, and even though the sorting differs based on the state, the interface is pretty generic. The extra overhead in the compare-function is well worth the gained abstraction in this setting.

The most important functions are

- `void pfair_task_insert(struct rb_root *root, struct task_struct *new)`
  Inserts a task into the runqueue starting in `root`. It will use the compare-function described in Listing 4.2, and move iteratively[1] through the tree. One notably different property about this function, is that as most trees choke on identical keys, this function does not. Two identical tasks indicate the exact same subjob signature, something that might happen frequently. To solve this "extra" tie-breaking parameter[2] by prioritizing the task already in the tree. By doing this consistently with all new tasks arriving, no task will ever be starved due to a large number of new, identical tasks.

[1]The "standard" way of traversing a tree, is recursively, but as the kernel has very limited stack, avoiding recursive function-calls is an absolute requirement.

[2]pfair solves this arbitrarily as the tasks are considered equal

- `struct task_struct * pfair_remove_task(struct task_struct *task)` Since we have the task, we do not need to traverse the task. All we need to do is unlink it and ask the library to rebalance the tree.

- `void pfair_migrate_ready(struct rq *rq)` This is perhaps *the* most important function in the queue-manager. This will move all tasks ready to be released from the ready queue to the active. It will draw heavily upon the preceding tasks.

In some areas, the queues are tapped into directly, in particular in the pick-next logic. The reason for doing this directly is to avoid needless function call overhead, and picking the next task is, compared to other tree-related operations, basic.

6.3 Summary

In this chapter we looked at two of the major datastructures. It is clear that they are complex and at times difficult to use. Especially the red black trees have proved to be a challenge. However, once mastered and properly used, they provide unparalleled control and abstraction. By drawing on exact knowledge about the hardware, the kernel can prefetch regions of memory, effectively masking out much of the memory latency found in cold-cache scenarios.

Chapter 7

Linking to userspace

"A system call is the mechanism used by an application program to request service from the operating system based on the monolithic kernel or to system servers on operating systems based on the microkernel-structure."
— `http://en.wikipedia.org/wiki/System_call`

Up until now, we have only looked at how the different datastructures can be utilized in the kernel. In this chapter we create the connection between the task in one end and the scheduling code in the other. This is done via a set of system calls and several attributes in the `/sys` directory exported via SysFS.

7.1 SysFS and `kobject`

7.1.1 `kobject`

One of the new things introduced when the kernel moved from 2.4 to 2.6, was a new approach to how a device was represented and described in the kernel. The aim was to minimize code duplication, to list all devices in a tree-like fashion where devices depending on other devices were ordered in the tree. An easy way of linking a device to its driver. Finally, the ability to walk the tree from the leafs all the way up to the root, powering down the nodes (devices) in the correct order.

As power management in the kernel are a global event, the kernel need a generic and simple way of turning off the power on every device when the systems enter a power-save mode. On top of that, this *must* be done in a very specific order. E.g. an USB hub must be powered down before the PCI bus is turned off. We focus on the `sysfs`-entries in this chapter. Knowing that the `kobject` lies at its foundation, we can abstract that away. A more thorough guide to kobjects[1] can be found in [19, Ch. 17].

7.1.2 SysFS

SysFS, normally mounted at `/sys` is a filesystem, that at first glance, look very much the same as `/proc`. Like `/proc`, `sysfs` is a way of user mode applications to access kernel data structures. Unlike the `/proc`-directory, `sysfs` is more organized, forcing each device to inherit other devices, thus building a tree of the devices. Each directory is represented as a `kobject` and each file in a directory maps directly to an attribute (which in turn is normally tied to a variable).

7.1.3 Kernel interface

As with all other systems in the kernel, the SysFS interface is extensive and we therefore only presents a small subset of all the functions. The entire interface can be found in `include/linux/sysfs.h`.

7.1.3.1 Create the kobject

The `kobject` can be found as a folder in `/sys/`. To create a new `kobject`, the sysfs-subsystem has a dedicated function:

[1] `http://lxr.linux.no/linux/Documentation/kobject.txt`

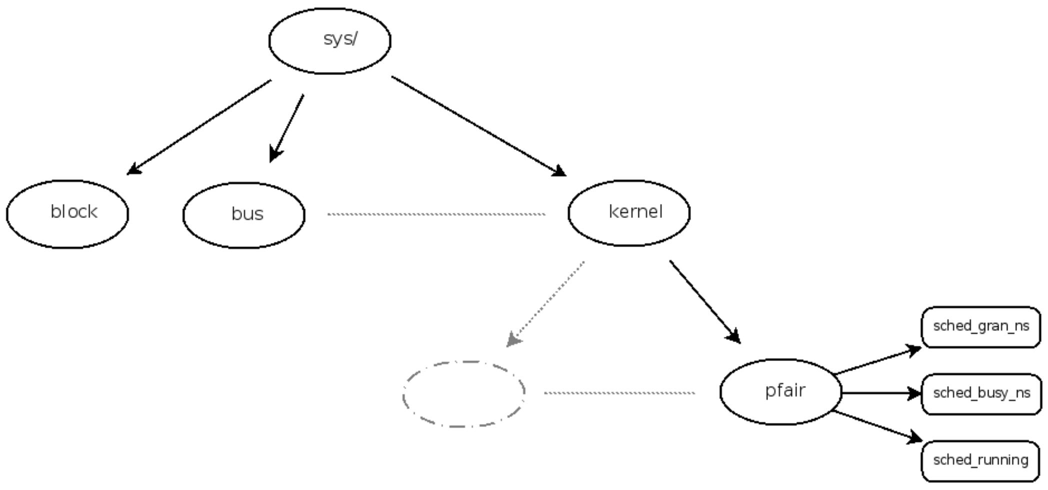


Figure 7.1: A subset of the objects and attributes found in `/sys` in any standard GNU/Linux distribution

```
struct kobject * kobject_create_and_add(const char *, struct kobject *)
```

The first parameter is the name with which it will be represented in `/sys/` and the second is the parent- `kobject` . If, for instance, the kernel kobject (`kernel_kobj`) is passed and the supplied name is “pfair”, a new directory will be added in `/sys/kernel/pfair/` . If no existing `kobject` is passed (i.e. NULL), the new object will be created in the root of `/sys/` . Figure 7.1 gives a tree-view over how the different sysfs objects and attributes are connected.

7.1.3.2 Create the files

Before an attribute can be used, it must be initialized. The proper way of doing this, is via the `__ATTR` macro. The macro needs the name of the exposed attribute, the file-mask and two functions, one for converting the value into a returned string and one for converting a supplied string into the correct datatype. These functions are normally described as the *show* and *store* functions.

```
static struct kobj_attribute sgattr =
          __ATTR(sched_idle_ns, 0644, pfair_show, pfair_store);
```

This links the attribute to the name “ `sched_idle_ns` ”, it gives the user root (root will own all files in /sys) `rw` -access, the rest of the world `read-only` . When someone tries to read the attribute, the function `pfair_show` will be queried, and when a value is written to the attribute, `pfair_store` . To connect the attribute `sgattr` to a kobject:

```
sysfs_create_file(kobj, &sgattr.attr);
```

The function will return a non-zero value upon error and should be properly handled.

7.1.3.3 Show/store functions

SysFS is about exposing attributes to the user-space, and it requires functions for showing and storing values. When initializing the attributes, references to functions to handle these must be given. If an attribute does not need either reading or writing, the corresponding reference can be set to NULL.

The required signature for the show-function is:

```
ssize_t show(struct kobject *, struct kobj_attribute *, char *buffer)
```

The returned `ssize_t` is the size of data written to the buffer. The kobject is the kobject where the sysfs-entry belong. The function should confirm the name of the attributes as well as the length of the data received.

```
ssize_t store(struct kobject *,
              struct kobj_attribute *,
              const char *buffer,
              size_t size)
```

The store function is nearly identical to the show function. The difference is the `const char *buffer` and the size added to the list of parameters. The `const` keyword help the compile catch possible errors where the buffer is modified by the store-function. The size gives the size of the data passed to the function. The function is required to return the amount of data read.

7.1.4 Sample module

To test how `sysfs` worked and open up for faster prototyping, a module was created. it can be found in App. B.4 and shows the basic show-store functionality. It will add two attributes in `/sys/kernel/pfair` . When values added, the result will be printed to `/var/log/kern.log` [2].

[2]At least on a Debian-like distribution. Behavior has not been verified in other distributions (alternatively, use ' `dmesg` '.

7.1.5 Final result

In the scheduler, `kernel/sched_pfair.c` , 3 attributes have been exposed in `/sys/pfair/` . Originally it was planned to expose this under the kernel-directory, but the `sysfs`-interface to the kernel is initialized after the scheduler is brought online. This would therefore require delayed initialization. Because of this, it is just added to `/sys/pfair/`. We could end up with 6 functions for handling this, however, as the attribute-struct contain the name of the attributes, we have multiplexed the handling of all three attributes into one general show-function and one store-function. These are named `pfair_show` and `pfair_store` and are both located in `sched_pfair.c` .

7.2 System calls

System calls, or " `syscalls` ", is the standard way of allowing userspace code to trigger events in the kernel. Syscalls in Linux is fast, and this is largely due to the fact that the infrastructure is very efficient, but also because there are so *few* syscalls. Adding a syscall to Linux is something the kernel community is very reluctant to do, and there have to be a very good reason for this. Even though the internal kernel API is declared unstable as a design-feature, the *external* API *must* remain stable. This means that once a syscall is added, it cannot ever be removed.

7.2.1 Syscall background

To understand how `syscalls` work, we need to look at how the kernel is started. On x86 architecture, a set of soft-interrupts, or "system gates" are available, each can be given a unique vector and access-rights ([7, Ch. 4, App. A]). `0x80` is given to syscalls, and userspace is granted access to raising this interrupt[3]. When this happens, the processor will immediately jump to a predefined address and execute the code found there. We say that the interrupt is *trapped.* It is here that the system call entry is located. For further discussion about how the rest of the syscalls are organized, please see [19, Ch. 5] and [7, Ch. 10].

Once in the syscall-entry, it will look at the registers. When calling a syscall, the syscall-number must be placed in `%eax` . This number is then used in a table-lookup to find the proper syscall. Once found, the required number of parameters are copied to the stack from `%ebx` , `%ecx` , `%edx` , `%esi` , `%edi` and `%ebp` . As the number of general registers are limited in `x86` hardware, this is maximum

[3] `asm('int 0x80');`

number of parameters. If more are needed, or the number is required to change later on, a *value by reference* should be used. If an address to a `struct` is passed, an unlimited number of parameters can in theory be passed.

7.2.2 Required change

In this project, we only had access to x86 hardware. The following discussion therefore only applies to this. However, it should not be too difficult to add these changes to other platforms, as the only thing that changes, is where you add the table-lookup address.

In x86, you need to edit the file `arch/x86/kernel/syscall_table_32.S` . Here you will find a long list, and at the bottom you must add the new syscall-names. It is very important not to add it between existing lines, as that would change the unique number given to each syscall, breaking user-space. The function must be given the prefix `sys_` , and the function created to trap the interrupt, must be created with a special macro, where all input-arguments must be specified.

7.2.3 Final result

To add a syscall, you need to find a place to add it. Since all the other scheduler-related syscalls are located in `sched.c` , the pfair-related syscalls were placed here as well. Since we can compile away all code `sched_pfair.c` , and once a `syscall` is added, it *cannot* go away, the syscall-entry must be placed in a file not included based on a compile-switch. To add a syscall, the correct way is to use the supplied macros with the correct amount of arguments. When asking for 2 arguments, the correct macro is `SYSCALL_DEFINE2` . The syntax is slightly different as type and variable-name is separated. In the body of the syscall, the following action is then performed:

1. Test to see if the user has CAP_SYS_NICE, which is normally the root-user.
2. Make sure `pid` is valid, i.e. greater than 0, and that the supplied parameters are valid (not a zero-value reference).
3. Copy the data safely from user-space to kernel-space via `copy_from_user()`
4. Find the task identified by `pid` and store the task descriptor.

When all of this is done without errors, the syscall delegates control to `__sched_pfair_[update|reweigh]()` which handles the actual syscall behavior. These functions can be found in `sched_pfair.c` and will update a non-pfair task to pfair or reweigh an existing pfair-task. The syscall and delegated function for `sched_pfair_release()` is basically identical and will not be covered. The delegated functions are described in Sec. 9.3.

7.3 Summary

In this chapter we had a look at ways of interacting directly with the scheduler. First we looked at the tuning-knobs provided by sysfs-attributes exposed in the `/sys` -directory. Then we moved on to look at the task-wise interface to the scheduler, the `syscalls` .

Chapter 8

Time management

"Time is the most undefinable yet paradoxical of things; the past is gone, the future is not come, and the present becomes the past even while we attempt to define it, and, like the flash of lightning, at once exists and expires."
— Charles Caleb Colton

The previous chapters have discussed various data structures and memory management. We now turn our attention to how the kernel manages time. The pfair scheduler revolves around frequent timer interrupts thus having a flexible, robust and optimized timekeeping system is an absolute requirement.

This chapter is organized as follows: first we list the requirements pfair place on the timer (Sec. 8.1). Sec. 8.2 then look at how the timer system has evolved over the years and how the current system can meet these requirements. Sec 8.3 looks at the `hrtimers` API before we in Sec. 8.4 presents two small modules to determine the error in the timers, and how various types of load affect this. The other module looks at starting timers on other cores. The final result in the scheduler can be found in Sec. 8.5. The chapter is concluded in 8.6.

8.1 Background and requirements

When it comes to time management, the kernel has evolved over the years. As the kernel continues to increase the number of supported hardware units, having a generic framework for time-related devices is essential. Part of the rework seen in Linux-2.6.16, was the generic abstraction layer, providing a simpler way of integrating new hardware devices into the existing timekeeping framework.

The current timekeeping infrastructure is separated into 3 parts; *a*) the timeofday layer, *b*) the clockevents layer and *c*) hrtimers. In this project, only `hrtimers` are of interest, although the three are strongly interconnected. See [12] and [23] (and section 8.2) for a discussion about `hrtimers` and the timekeeping architecture.

The general design of the scheduler was discussed in Ch. 4. No specific requirements has yet been placed on the time-keeping architecture except a period timer interrupt. We now turn our attention to these details:

High resolution — it should be possible to achieve sub-jiffy resolution for the timer-ticks (it should not be tied to the kernel variable `HZ`).

Accurate and efficient — it should trigger as accurately as possible, and as little time as possible should be spent inside the timer infrastructure. It should also be easy to adjust for drift.

Dynamic — it should be possible (and easy) to change the frequency of the the timer without rebooting, or recompiling the kernel.

Per-CPU timer — make it easy to synchronize the timers (either all at the same boundary, or a "staggered" approach where each timer is spaced evenly apart inside the length of a window.

Low impact — when not in use, it should be possible to remove the timer interrupts all together without affecting other parts of the system. This will make it possible to compile kernels with pfair support and not experience degraded performance when no pfair tasks are running.

As we will see in Sec. 8.2.2, `hrtimers` can do all of this, and with the RT-preemption patch, some of the related performance and latency issues are minimized.

8.2 The timer infrastructure

The problem with timers is that all they do is to make sure the event is not released *before* the specified time. To keep the event as close to the specified time as possible, the timer-event infrastructure must be efficient and able to handle a large number of queued time-events without noticeable degraded performance. This is clearly a difficult and complex task.

8.2.1 Cascading Timer Wheel (CTW)

It is worth the time to look at how the previous timer architecture was implemented. Not only because there are still kernels out there with `CTW`, but also because this illustrates some of the major obstacles met in when implementing a efficient, scalable and robust timer.

In 1997, the CTW was included in the kernel, and was then a huge improvement to the old O(N) linked list implementation. The CTW contains 5 runqueues each holding a separate slice of the time-frame in a set of *buckets*. The first queue contains timers that will expire within the next 255 ticks[1]. The remaining queues all contains 64 buckets.

Listing 8.1: *How the timers are stored, migrated and released*

```
foreach timertick
    if queue[0][counter[0]] not empty
       release(queue[0][counter[0]])
    counter[0]++
    if counter[0] == 256
       for n in range 1 4
          counter[n-1] = 0
          expire_bucket(queue[n][counter[n]], queue[n-1])
          counter[n]++
          if not counter[n] == 64
             break
```

Looking at Listing 8.1, the the counter is incremented every tick, and the corresponding bucket in `queue[0]` is emptied. When the counter overflows, queue 0 is empty, and the counter for queue 1 is incremented. All tasks in this bucket in `queue[1]` is then expired and are moved down into queue[0]. This is handled by `expire_bucket()` as it takes all timers stored in bucket `queue[n]` and moves

[1]A tick is normally the periodic timer-tick event, often denoted HZ and must be set at compile-time.

Queue	**Start**	**End**	**Cascading Expiry (1000Hz)**
0	0	$2^8 - 1$	1ms
1	2^8	$2^{14} - 1$	256ms
2	2^{14}	$2^{20} - 1$	$\approx$ 16 sec
3	2^{20}	$2^{26} - 1$	$\approx$ 17 min
4	2^{26}	$2^{32} - 1$	$\approx$ 18 hrs 36 min

Table 8.1: Table of the 5 queues with jiffy-range and expiry period. Each queue contains one bucket for each element in the bit-range (256 for queue 0, 64 for the last 4) Note that when queue 4 expires every 18 hour, every queue also expires.

down to `queue[n-1]` , placing each timer in the correct bucket. Note that when a timer interrupt occurs, it is not necessarily bucket 0 that contains the timers, but bucket `counter[0]` . This is a code-complication that removes the need for moving tasks through the buckets at each timer interrupt. The added complexity can be confusing at first, but it is well worth the complexity. Since the number of buckets are a factor of the same integer (i.e. they are not relatively prime), all queues will eventually migrate tasks at the same tick (2^{26} seconds). See column Cascading Expiry in Table 8.1. This can introduce unpredictable delays in the timer system, and is the reason for the name — *Cascading* timer wheel.

8.2.2 High resolution timers - hrtimers

The hrtimers provide a generic and abstract interface to the timer infrastructure. The system can change timer hardware (clocksource) at run-time, and return the timer granularity to the user, providing not only a very flexible, configurable and robust system, but also a way for the user do *adapt* to different hardware.

Scalability is always an important issue. One of the major drawbacks with CTW was how the cascading effect could ruin predictability (and performance). This effectively put a cap on how many active timers the system could support. By moving the timers to a per-cpu queue, and using a red-black tree to store a time-ordered list, the number of timers pr. CPU decreases, and the cascading effect vanishes. Granted, to walk a red black tree on every timer expiry is not efficient, and for that reason the systems also uses a separate list to give the expiry code fast access to the next event. By using the red black tree as fundamental

Timer frequency	**Granularity**	**32 bit**	**64 bits**
100 Hz	10 ms	≈ 1 year 130 days	≈ 5.8 billion years
1000 Hz	1 ms	≈ 49 days 17 hours	≈ 584 million years
10 kHz	100 μs	≈ 4 days 23 hours	≈ 58 million years
1 MHz	1 μs	≈ 1 hour 11 min	≈ 584,558 years
1 GHz	1 ns	≈ 4.3 sec	≈ 584 years

Table 8.2: Comparison between 32 bits- and 64 bits length for timer variable with respect to timer granularity and maximum delay (approximate)

storage, insertion can be done quickly "regardless"[2] of queue-size.

Granularity and extendability is one of the big improvements. With the clockevents- and timeofday-architecture, legacy code still dependent upon the HZ value will work as expected. If the underlying hardware does not support more than HZ resolution, neither will `hrtimers`. However, if high resolution hardware timers are available, `hrtimers` can utilize these without changing HZ, rebooting or recompiling the kernel thus dramatically improving the granularity.

Maximum timer period for the old CTW was approximately 50 days when the HZ ran at 1000Hz. In many cases, that was more than adequate. With the new high resolution requirements, this quickly becomes a limitation.

From table 8.2, we can see that if we want 1μs resolution, using 32 bits severely limits the period, and for nanosecond granularity the result is even worse[3]. It is also evident that for a 100Hz timer, using 64bits is a complete waste of space as no computer will run for 5 *billion* years.

As the new timer variable is 64 bits, and Linux is supported on both 64bits and 32bits architecture, time is wrapped in `struct ktime_t.` By using compile-defines and typedefs, `ktime_t` will be represented as a "pure" 64 bit integer on 64 bit architecture, and a struct containing seconds since the epoch and nanoseconds since the last second, both in 32 bits integers. The timer architecture has macros for dealing with `ktime_t` , abstracting away the underlying representation. When nanosecond granularity is no longer adequate, the hrtimers are so modular that

[2]it will use O($lg_2 N$) time

[3]4.3 seconds vs. 584 years 203 days 19 hours 44 minutes 9 seconds 709 ms 551 μs 616 ns

a lower bound can be set "relatively easy", and by using `ktime_t` , no other code needs to change.

8.3 The timer API

This section will cover initialization, start, callback functions and cancel. It will also describe how to start a timer on another CPU. Together with Sec. 8.4, this will provide an adequate introduction to the `hrtimers` API. `hrtimers` has a lot of functionality, but we focus on what we need, see the full documentation[4] and API[5].

Central to `hrtimer` , is the datatype, `struct hrtimer` . This struct contains all the elements need to store and retrieve the timer, as well as a pointer to a callback function[6].

8.3.1 `hrtimer_init()`

Before a timer can be used, a memory region must be reserved, and the region must be initialized. Memory allocation is done via `kmalloc` (or a SLAB cache), see Ch. 5 for details about memory management.

This function takes 3 arguments:

struct hrtimer *timer a pointer to a hrtimer struct where the data-fields are stored. A normal convention is to embed this directly into another struct used to keep various data one wants to pass to the callback function.

clockid_t clock_id , for instance CLOCK_MONOTONIC

enum hrtimer_mode mode - relative to "now" or absolute compared to the epoch (1. Jan 1970), for instance HRTIMER_MODE_REL for a relative timer.

```
enum hrtimer_restart timer_callback(struct hrtimer * hrt)
```

The callback-function is stored in `hrtimer.function` and must be set *after* initialization and *before* start (to be precise; before the timer expires). Whenever the timer expires, this function will be called, and the parameter `struct hrtimer *hrt` is the struct given as argument when the timer was initialized.

[4]http://lxr.linux.no/linux+v2.6.29/Documentation/timers/highres.txt
[5]http://lxr.linux.no/linux+v2.6.29/include/linux/hrtimer.h
[6]http://lxr.linux.no/linux+v2.6.29/include/linux/hrtimer.h#L100

8.3.2 `hrtimer_start()`

This function is also used for re-releasing timers. It takes 3 parameters and returns an integer indicating success (0) or failure (1):

- A pointer to the timer data type (`struct hrtimer *timer`), normally embedded within a data-element (in the same convention used for both linked lists and red black trees).
- The delay for the timer (absolute or relative to "now" found in `struct ktime_t`
- `enum hrtimer_mode` which can be either HRTIMER_MODE_ABS or HRTIMER_MODE_REL.

8.3.3 `hrtimer_cancel()`

This is used to cancel a pending timer. It takes a pointer to a `struct hrtimer` as its sole parameter. If the timer has already expires, the function returns 0, 1 if the timer was active.

8.3.4 Remote timers

An SMP-capable kernel has a set of functions for running functions on one, a subset or all of the cores available. The function `smp_call_function_single()` will run a function on a specified CPU. The module presented in Appendix B.6 does just this.

However, even though starting a remote timer is easy, the real problem arises when the kernel experiences load. It is at the moment, no deterministic way of starting a timer. One risk that the function call will be delayed for a long time, and starting accurate timers can therefore be a challenge.

8.4 Sample module

In App. B.5, a module for measuring the latencies for timers are presented. Some very basic test-results showed that for a standard Linux v2.6.29-kernel, the latency could be from approximately 2 μs to 40μs. Granted, a latency of 40μ is not a very large delay, but since pfair require 1ms resolution, we have a latency of nearly 4%, possibly at *every* tick. The same test was conducted with the RT-preempt patch. This proved to be a lot less susceptible to the load. The average delay was more consistent, varying between 5 and 7 μs.

Another module was also created to investigate the possibility of starting timers on other CPUs. This feature is essential for the pfair scheduler. The code proved to be quite simple, but it also proved susceptible to kernel load. This module did not measure average latency, as the main goal was to provide a proof-of-concept for remote timers.

8.5 Final results

Obtaining accurate timers have proved to be very difficult. This did not come as a surprise, but the amount of uncertainty did. If the pfair scheduler is going to work, *very* accurate timers are required. Not only to keep the deadlines, but also to prevent unbounded drift.

The final result in the kernel code proved to be next to impossible to implement properly. The minor details of starting a timer is not the problem, nor is running the actual pfair-timer event machine (described in Sec. 4.1 and in Fig. 4.2 on page 43. The challenge lies in accurately calculating the drift and counter this. As we will see in later chapters, several other flaws have been discovered in the pfair-algorithm, leading to its demise. For this reason, further work on the timers were suspended until these other problems were resolved. They were not, however, and therefore the final implementation of the timers has not been completed.

8.6 Summary

This chapter started out by giving an introduction to how timer-events were and is managed in the kernel. It then moved on to describe the kernel-API and two sample module. Sadly, the scheduler code for managing the timers have not been completed, for reasons not solely to blame on the timer infrastructure.

The next chapter will look at the core part of the scheduler, and several other, major issues leading to a stop in the implementation will be discussed.

Chapter 9

Implementation of the core scheduler

"You have to seek the simplest implementation of a problem solution in order to know when you've reached your limit in that regard. Then it's easy to make trade-offs, to back off a little, for performance reasons. You can simplify and simplify and simplify yet still find other incredible ways to simplify further."
— Steve Wozniak

In part I, the motivation for the project and general layout of the scheduler was presented. Then, in the start of part II, Ch. 4 dived into the actual design and how the various components should be created and interconnected. This chapter, being the last chapter in part II, looks at how the "core" pfair scheduler has been devised, tested and implemented.

9.1 Core algorithm

The core algorithm is a bit loosely defined term. Several of the key-elements have already been covered. We now turn our attention to the required calculations and interaction between sub-jobs, jobs, runqueues and user-space.

9.1.1 Calculations for the subjobs

To better understand the algorithmic problems with the scheduler, a user-land version was created. It initialized a set of tasks with different execution cost, deadline and period, and proved to be an effective way of tweaking the calculation of the subjob values, but also for finding the hyper-period to estimate the overall utilization. As the kernel does not support floating point arithmetic, it was an ideal place to test numerical approximation.

This of course, took a considerable amount of time, but in the end a lot of time was also saved. The implement-, compile-, test- and refine-cycle is much shorter for a normal application than for the kernel (the prototyping went a lot faster). Even with distributed compilation, a single cycle takes at least 10 minutes. With a normal C-program, this is done in a manner of seconds. The whole program is attached, under `code/pfair_window/` .

9.1.2 Testing the elements

When the calculations were verified and found correct, work started on the other areas, working through documentation, small test-modules and adding more code to `sched_pfair.c` . However, verifying this along the way proved to be very difficult. Not only because the prototyping cycle is very long, but also the lack of debugging-support. Granted, both kgdb, debugging over the network, stack-dumps and other tools can be used, but none of these provide the flexibility and speed that can easily be incorporated into a user-land application.

Then, in April 2009, LinSched was discovered. Originally released in Dec. 2008, LinSched has been applied to Linux v2.6.23.14, one of the earliest versions of CFS. This means that most of the fields in `sched_class` are present. LinSched uses parts of the Linux library directly to emulate a kernel scheduling environment in user-space. Being able to use linked lists, trees and parts of the math-library directly, proved invaluable. Several subtle (and not-so-subtle) bugs were traced down via this simulator, especially in the `rb-tree` -related code. However, two major components were missing, making LinSched unable to aid the construction all the way:

1. Timers: the lack of any timers meant making sure the time-critical code actually worked.
2. Only one `task_struct` were created for the entire schedule. This caused some confusion at first as one would normally expect that each task received an unique `task_struct` .

Clearly, element 1 can be emulated without a lot of work. A basic estimate landed the required work at about 2 working days. Clearly, the second element could be changed within a day, allowing for a few hours tracing down unexpected bugs as this would change the LinSched internals. However, at the time, 3 days were simply too much. Furthermore, LinSched was only used to validate the datastructures. The calculation did receive some optimizations (e.g. the list was traversed in reverse). In total, LinSched can be an invaluable tool when devising new scheduling algorithms, but it needs to be updated to a more current version, and it should emulate some sort of timing interface. The code written for LinSched has not been directly attached, as the change between a LinSched-compatible `sched_pfair.c` and the official `sched_pfair.c` are almost negligible.

9.2 The expected interface from the main scheduler

The main scheduler, expects each scheduling class to implement a set of functions and store pointers to these functions in a `struct sched_class` . Of all these functions, we only discuss the most important functions here. The rest of the functions are documented in `sched_pfair.c` .

The whole pfair-engine is driven by the timer. This was described in Sec. 4.1 and 8.5. At the end of the timer state-machine, the flag `TIF_NEED_RESCHED` is set in a task to be preemted. When control was about to leave kernel-mode, this would trigger `schedule()` which is where we are conceptually right now. Figure 9.1 shows 3 of the related functions in `sched_class` being called from `schedule()` . Note that `pre_schedule()` leads nowhere. This is one of the functions that can be implemented voluntarily, and we do not need it.

9.2.1 `put_prev_task_pfair()`

Called before a task is about to be removed from the CPU, once this is called, we know that the next step will be to query for a new task. We use this function

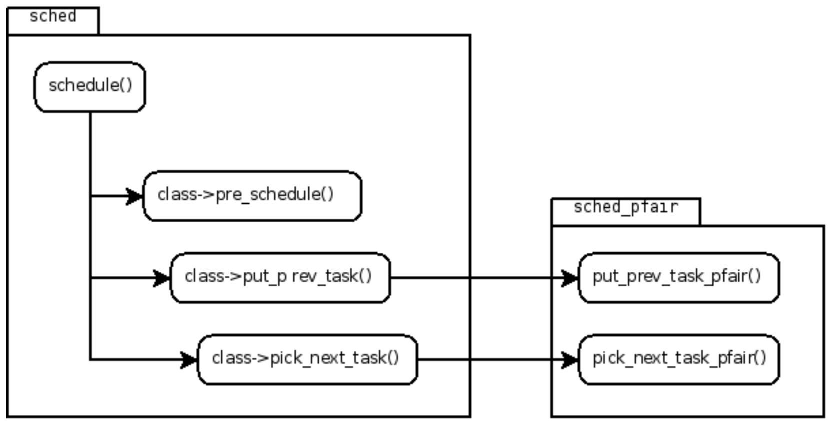


Figure 9.1: A small subset of the `sched_class` functions which are used by `schedule()`

to store the task back into the runqueue. We must do this, because `schedule()` does not call `enqueue()` or any other function to indicate that we should save the task. If we do not add the task in *this* function, it will be lost to the scheduler[1].

The function will test if the `TIF_NEED_RESCHED` flag is set. If it is, it will go through some steps:

1. advance the `psj_curr` one position.
2. If `psj_curr` has overflown:
   a) If task is periodic, reset `psj_curr` and move `release_time_ns` one period forward
   b) If task is not periodic, it has overflown and an error should be signaled. At the moment, only a warning is issued.
3. Change `pfair_state` to PFAIR_STATE_READY
4. It is inserted into the ready-queue where it will be placed according to the compare-function (Listing 4.2).

9.2.2 `pick_next_task_pfair()`

After `put_prev_task_pfair()` , `schedule()` will call `pick_next_task_pfair()` . This function is expected to return the task best suited to run, in other words,

[1]Of course, tasks are also stored in a big list, but this is for other purposes and the pfair scheduler does not have access to this list.

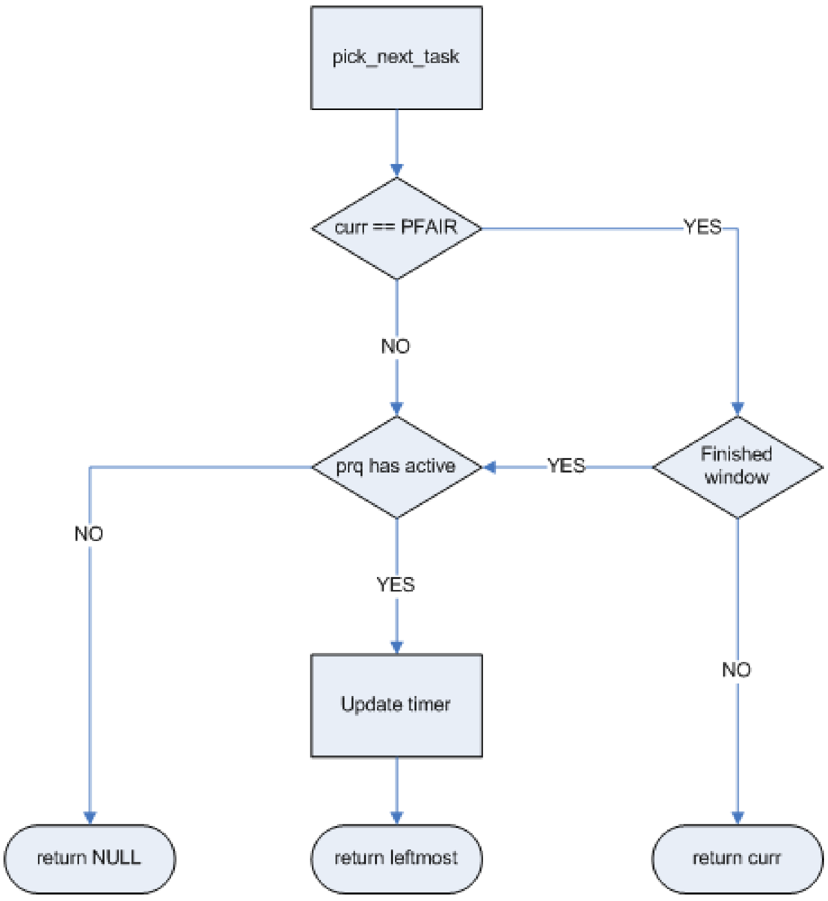


Figure 9.2: Flowchart describing the choice taken in `pick_next_task_pfair()`

the highest priority task available. As this function can be called out of place, i.e. not at the time we desire, we must test to see if the currently running task should be kept running, or if a better task is suited. The flowchart in Fig. 9.2 shows this.

When we are in this function, we know that

1. `curr` was pfair and has exhausted its timeslice
2. `curr` was not pfair, but another pfair-task has entered `PFAIR_STATE_ACTIVE` and must be allowed to run.
3. `curr` is any odd task (type probably pfair, but special cases can be proved to exist) and has finished.

9.3 The syscall interface

In Ch. 7 and in particular in Sec. 7.2, the user-land interface to the scheduler was described. The syscalls were introduced, but only the "gateway" found in `sched.c` . We now look at the functions called from these gateways, now that the input has been validated, tested and properly copied into kernel-space.

9.3.1 `__sched_pfair_update()`

This is when a task enters `SCHED_PFAIR` . After all the testing in the syscall-gateway, this function moves on to test the content of the variables. This was not done in the gateway-function to encapsulate the data. All the gateway-function does, is to verify that the memory-reference is valid and that it successfully copies data from user-space to kernel-space.

When a set of parameters successfully passes these tests, the number of subjobs are set to 0 and the periodicity is set according to the parameters. Then, control is passed to `__sched_pfair_reweigh()` .

9.3.2 `__sched_pfair_reweigh()`

The reweigh-function does a lot of the same things the update does. In fact, so much of the same, that update uses reweigh. They could very well have been multiplexed together (remember that the kernel community is very reluctant to add new syscalls), but for clarity they have been kept apart.

First off, it will test to see if the number of new jobs are fewer than before. If so, the task will *release* resources, and this can be granted immediately. If more resources are needed, things will be a bit more complicated. A lot of the logic is handled by `__sched_reserve_util()` , especially the requests for resources. Then, if resources are successfully reserved, the subjob-values are computed as described in Sec. 4.2.3 and Listing 4.1.

9.3.3 `__sched_pfair_release()`

The release-function is used to start a new cycle of the task, or to release a new job and is very simple. It tests to see if the offset is 0, and if it is, it will trigger a periodic release. Otherwise it is an aperiodic release. This requires a time in the future, and no earlier than the period.

9.4 The acceptance function for new tasks

As already mentioned, the `__sched_reserve_util()` is the acceptance function. This is the embodiment of the resource reservation approach discussed in Ch. 1 (Sec. 1.1) and gives us a simple and effective way of guaranteeing that we can meet the deadlines.

Since we want to guarantee a hard real-time system, we must make sure no transients can cause other deadline-misses. We therefore use the deadline as basis (since we do not allow for deadlines longer than the period). We then find the scaling factor so we know how many jobs the task will release within a hyper-period. The scaling factor is then used to multiply the `wcet_ns` so we get `wcet_scaled_ns` . If this can be added to `consumed_time_ns` without this exceeding `hyper_ns * num_online_cpus()` , we can allow the task to enter as pfair.

9.5 Summary

In this chapter, we looked at how the core part of the scheduler has been implemented. The most crucial functions to the inner workings has been discussed, and it is now time to tie every piece together.

Part III

Assembly and Evaluation

Chapter 10

Bringing it all together

"Nature laughs at the difficulties of integration[1]"
— Pierre-Simon Laplace

The first section (Sec. 10.1) lists all the small elements needed, but not large enough to justify a dedicated chapter. Most of the subsystem integration was described in the preceding chapters, Sec. 10.2 aims at tying all of those "superstrings" together. This section is quite short, as most of the problems encountered here, are difficult to describe as other than "compile errors" and variable faults. The chapter is concluded in Sec. 10.3 with a short discussion about how the integration process went.

[1]The author is aware that Pierre-Simon Laplace was describing calculus. Had he lived today, he would no doubt have had the quote extended to span computer software integration as well.

10.1 Other practical tasks

10.1.1 Add `kconfig`

The kernel build system is a complex, yet elegant system. It consists of a set of Kconfig-files, and these files have a special syntax. This syntax is described in full in [26].

The configuration menu is compiled and presented to the user, in several different ways. The most intuitive is the graphical interfaces, with `xconfig` and `gconfig` having the highest abstraction. The most common is perhaps `menuconfig`, which uses the ncurses library in order to present a graphical user-interface that requires very little system resources, and also works well via a terminal.

10.1.1.1 The new config variables

For the pfair scheduler, a new `Kconfig.pfair` was created in the `kernel/` source folder. As we only had `x86` hardware to test on, the file was included in `arch/x86/Kconfig` , in the same section as CONFIG_SCHED_MC (Multi-core scheduler support) under "Processor types and features". Note that SCHED_PFAIR depends on SCHED_SCHED_MC ("Multi-core scheduler support") so this feature must be enabled for pfair to be available[2].

- CONFIG_SCHED_PFAIR — The main "switch" for including the scheduler into the code. If this code is not set, *no* new code is added to the kernel source (except for the syscall-wrappers, see 7.2.3.

- CONFIG_SCHED_PFAIR_TIMESLICE — The length of the timeslice window pfair will use. It is currently set to a default 1ms, and can be changed (from 100 us to 100000 us) at compile time.

- CONFIG_SCHED_PFAIR_BUSY_LIMIT — Initially, this element was added to tune the limit for when the scheduler would return and allow a lower priority task (any other scheduling policy), or if the scheduler should just busy-loop until the timeslice boundary was reached. As busy-looping in the kernel is very ugly, this option will not be used.

- CONFIG_SCHED_PFAIR_SYSFS_R — Enable the sysfs interface to some of the pfair status variables, including the timeslice, in a read-only (ro) manner.

[2]It does not make sense to include a multi-core scheduler in a single-core system.

- CONFIG_SCHED_PFAIR_SYSFS_W — When sysfs is enabled ro, this option is made available, and the pfair values can now be changed, giving system administrators the option of changing the timeslice when the kernel runs.

- CONFIG_SCHED_PFAIR_DEBUG — Increase the verbosity of the kernel logging done by pfair from little to quite a bit. This should not be used in a production environment.

10.1.2 Allocating memory for the core

The scheduler is one of the first parts of the kernel to be initialized. As a result, the memory management subsystem is not yet available. In the `sched_init()`-function [3] the scheduler uses `alloc_bootmem()`, and its usage is not trivial. In fact, it leads to so many problems when used wrongly, that the current trend is to move *away* from `alloc_bootmem`. The current idea is to move the memory initialization earlier in the startup-process [4].

The pfair scheduler is not needed by any of the early kernel threads, we can wait until the memory management is initialized properly. The scheduler is actually initialized in *two* steps. The first being `sched_init()` , the second is `sched_smp_init()` . The latter also iterates over all the per-CPU runqueues. Luckily, `sched_smp_init()` is run *after* the memory-subsystem has been initialized. First, the global pfair runqueue is created (`struct pfair_rq * prq`), and the pfair subsystem initialized. Then, each runqueue is connected to `prq`. This way, each runqueue has a pointer, `rq->prq`.

10.2 Adding the pieces together

This is perhaps the most difficult task of all. Largely due to `LinSched` , several issues were discovered and corrected before this step. Then, by turning on all the warnings in the compiler, some erroneous variable usage was discovered and corrected. Numerous errors were encountered at startup and normal run. These turned out to be trivial in most cases, but took a long time to correct (largely due to the long prototype cycles).

Finally, and in many ways the "final nail in the coffin" for the scheduler, was the inability to get accurate and staggered timers from the timer subsystem. Without accurate timers, it will not be possible to accurately schedule a large

[3] `http://lxr.linux.no/linux+v2.6.29/kernel/sched.c#L8298`
[4] `http://marc.info/?l=linux-kernel&m=124265695228027&w=2`

number of tasks, and without staggering and drift-correction, the contention for the global runqueue will render the system completely unusable.

10.3 Summary

Most of the scheduler has now been described and pieced together. In most scenarios, it works without crippling the kernel. However, the timers render the scheduler useless and it cannot be used properly at this time. The next chapter will discuss what went wrong, and also look into why this really does not matter, as there are other, more fundamental problems with pfair.

Chapter 11

Summary and Conclusions

"There is no greater mistake than the hasty conclusion that opinions are worthless because they are badly argued."
— Thomas Huxley

We have now reached the end of the road, and a total evaluation is in order. As already stated on several occasions, the scheduler itself proved to be difficult to assemble properly. A lot of blame has been placed on the timers. This requires some modification. It is not the timer-system itself, but rather the fact that once you add timers, things become very delicate, very fast. When the timers cannot provide the required accuracy, a scheduler like pfair suffers greatly and the integration becomes very fragile.

Section 11.1 evaluates the scheduler implementation process. Sec. 11.2 then briefly covers deadline inversion. In Sec. 11.3 the goals of the project are discussed and evaluated. Then, to rectify some of the rather depressing results, a modified version of a scheduling algorithm, first published in August 2008 is presented in Sec. 11.4. The algorithm as well as a way of adding it to the kernel is described in full. The report is then concluded in Sec. 11.5.

11.1 Evaluation of scheduler

During the project, several people, some of the active kernel developers, have objected to the usage of a global algorithm, claiming that it will not scale due to memory traffic and lock contention. This would only get worse once you started to add several cores, not just 2 or 4. All of this is true, but it can also be minimized by staggering the timers. This again sparked the argument that if you need very accurate, staggered timers, a lot of resources will be tied into keeping the timers accurate. And, even if you stagger the timers, if you run this on a NUMA-machine, it does not matter how accurate your timers are. Again, this is true, but when you look at the potential gain from using a global algorithm that can achieve almost full utilization, this argument weakens. A partitioned EDF scheduler must cap the utilization prohibitively low to avoid *Dhall's effect* . Furthermore, a lot of the added overhead in a global scheduler can be compared to the extra overhead the load balancing logic will require in a partitioned setup.

1. The main problem with pfair, is preemption. If it is one thing that will steal cycles from a running system, it is switching tasks. What pfair does, is implementing fine-grained multithreading - in software.

2. The next reason for why pfair is a bad choice in real-life, is because it will waste a lot of resources. By only scheduling at timer boundaries, it cannot utilize the remainder of a timeslice, causing *another*, totally needless, task switch.

3. It cannot accept tasks with deadline closer to release than one timeslice, and the number of subjobs will be rounded *up*. This inability to support arbitrary offsets makes the scheduler impractical in real life.

4. Very strong dependency upon the timers. If the accuracy of the timers varies even a little, the pfair scheduler will become unstable and introduce more error and non-determinism.

11.2 Deadline Inheritance

Initially, this was not part of the project. However, it was early realized that a deadline driven scheduler would never be accepted into the kernel source tree without some support for this. At the time of this writing, the proxy execution protocol (introduced in 4.4) looks like a very good candidate, not only for pfair (or another deadline-driven scheduler), but for the *entire* kernel. Some rudimentary support for this has been incorporated in the design, but the major effort that remains, is to add logic that detects when a task blocks on a protected resource.

Delegating the CPU to the holder of the task is straightforward, and is already implemented in the pfair-scheduler.

11.3 Evaluation of project goals

The main goal was to implement a multi-core aware, deadline driven real-time scheduler. This has been fulfilled to some extent. Most of the subsystem is in place, the core logic is, as far as it can be verified without proper testing, implemented.

Based on work in [4], the pfair algorithm was chosen. This was because, at the time, it looked like the only way of utilizing most of the available resources in a multi-core system. During the project, it was realized that the introduced overhead from frequent task-switches would reduce this gain. The cache problem was largely ignored because once you even if it takes time to move the task-descriptor from one cache to another (which is the whole motivation for processor affinity), the task has been preempted anyway, and the cache is cold. The required accuracy of timers in the kernel lead to another problem, one which was very difficult to circumvent.

One of the goals that truly succeeded, was understanding the kernel source, configuration and compilation system. During these months, the gained knowledge of these systems has been staggering. The same goes for discussing the class with the kernel community, or a subset thereof. Even though several opposed the pfair-algorithm, the point of adding a deadline-driven scheduler has been accepted.

Tracking LKML in order to keep abreast of recent changes has involved a lot of time. Once the email-filtering became accurate enough, this proved to be one of the better tasks, starting each day with 30 minutes of email reading.

Identifying all the relevant subsystems, however, has not been quite as easy. A lot of time was spent searching and reading documentation. Some motivation was found in related code in the kernel, others found after discussions on IRC and yet other were found after several careful searches on the Internet.

The userland interface has been completed and verified. Both the syscalls and the SysFS interface works as expected. The fact that most literature regarding `syscalls` are outdated with respect to export and where to include the table-entries lead to some confusion at first.

The scheduler-implementation has been discussed on several occasions already. Again, I would like to recap; the scheduler is mostly finished. What prevents it from working is the timer-subsystem. As this introduced several very subtle bugs, and that they were very susceptible to load, made this task *the* hardest task.

Since the scheduler was not fully implemented, and it was realized at a late stage in the project, that it would never be accepted into the kernel, the testing has been omitted. The next section presents a new and revised algorithm where several of the design-choices are taken based solely on experience gained in this project. If included, tools like `cyclitest` [1] can be extended to handle deadline-tasks as well.

At the start of this project, it was believed that adding code to something so central as the scheduler, was possible in a matter of months. Not only was this wrong, it even bordered to the ignorant. What has been learned over these past few months however, are invaluable knowledge about how the kernel community works, how kernel development is conducted and how code is structured. The gained knowledge is truly astonishing, even though several of the initial goals failed.

11.4 A more mature approach

As stated, pfair is not the right way to go, and the main argument against pfair, is the excessive number of task-switches.Motivated by [8] where Buttazzo showed that EDF will lead to few task-switches, a way of extending EDF looked like the best approach. This section therefore presents a variant of the *Modified Least Laxity First* algorithm [20].

11.4.1 Preliminaries

The scheduler is a global approach, but it should be noted that for large systems, it should be divided into a clustered scheduler, where each domain is treated as a smaller, global domain. It uses deadlines as the base unit, because this is what gives the application developers the greatest expressiveness. This way, the scheduler can scale to very large, complex and dynamic systems, too difficult or impossible for offline tools to analyze and assign static priorities to.

[1]`http://rt.wiki.kernel.org/index.php/Cyclictest`

The problem today with global algorithms, is something called *Dhall's effect*[10]. It is possible to device a schedule of light tasks that can cause a deadline driven scheduler to fail, even at low utilization. The proposed scheduler avoids this by using a term called *time_to_failure* as base element for scheduling decisions.

The reason why multi-core EDF experiences this effect, is because they schedule solely on deadline and fail to take the *importance* into consideration. This is not required in a system with only a single core, as the utilization and the assumption that the period equals the deadline provide this. In a multi-core system, the deadline is important, but it cannot be used alone. Some algorithms use *weight* instead, but this assumes that the task will not be preempted once it has started running. If it is preempted, the logic fails and one risks to miss deadlines. Thus, a way of describing not only deadlines and weight, but also granted resources must be available. The next sections show how this is done, and how it can be done in an efficient manner.

This algorithm is a derivative of Modified Least Laxity First [20] and Earliest Deadline Critical Laxity [15]. Neither of these can accept fully utilized schedules. The next proposal can, as far as the author has been able to verify, accept this. A set of assumptions are needed, as well as notation. This is covered next.

11.4.2 Assumptions and notation

We assume a set of periodic tasks in a hard real-time system. We assume that no extra overhead is added in an unpredictable manner (this can be approximated by using the RT Preemption patch) and that the estimated execution cost for the tasks are an upper bound. We distinguish between soft and hard tasks by allowing soft real-time tasks to use the period as basis for the utilization, even if the deadline is shorter. This opens up for deadline tardiness as we can have situations where several tasks have a high priority to run thus forcing one or more tasks to miss their deadlines (we say we have *transients*). For hard real-time schedules, we use the shorter of the two as we cannot accept deadline tardiness.

We use the same notation as most EDF-literature, and we set the maximum utilization to the number of CPUs (M)

- Let a schedule $\mathbb{S}$ consist of a set of τ tasks.
- $\mathbb{M}$ is the set of tasks currently scheduled to run on any of the m processors.
- Let a single task be denoted τ_i and there be n tasks, $i \in [1, n]$. Let the jth release of a task be a *job* $\tau_{i,j}$.

- Each task have a period T_i that describes the minimum interval between releases.
- D_i describes the relative deadline for a job after release, and absolute deadline is $d_{i,j}$ and $D_i \leq T_i$
- The estimated computation cost (WCET) is denoted C_i, $C_i \leq D_i$.
- The individual and total utilization is then

$$U_i = \frac{C_i}{min(T_i, D_i)} \tag{11.1}$$

$$U = \sum_{i\in[1,n]} U_i \leq M \tag{11.2}$$

- Absolute release-time is $r_{i,j}$ and the time between two releases is *at least* T_i, e.g. $r_{i,j+1} \geq (r_{i,j} + T_i)$.
- The allocated time for a given job is given by $C_A(\tau_{i,j}, t)$, and describes the amount of processor time allocated to the task at time t.
- Let the unallocated computation time, or remaining time, for the currently active job $\tau_{i,j}$ at time t be $C_R(\tau_{i,j}, t) = C_i - C_A(\tau_{i,j}, t)$
- The relative time to failure, ttf and the absolute time of failure, F, is then the time left for when a task can be deferred execution and still be able to meet the deadline:

$$\begin{aligned} ttf(\tau i, j, t) &= (d_{i,j} - t) - C_R(\tau_{i,j}, t) \\ &= (d_{i,j} - C_R(\tau_{i,j}, t)) - t \\ &= F(\tau_{i,j}, t) - t \end{aligned} \tag{11.3}$$

  Note the need to have t as part of ttf. This is solely a convenient way of describing the (relative) ttf at a specific point in time, when a certain amount of CPU time has been granted.

11.4.3 Earliest Failure First

We have denoted the algorithm as Earliest Failure First as it will always pick the task with the lowest Failure value.

Let the initial set of tasks eligible to run be denoted $\mathbb{S}$ and let the total utilization of the set be less than M, i.e $U \leq M$. Find $ttf(\tau_{i,j}, t) \forall \tau_{i,j} \in \mathbb{S}$ and

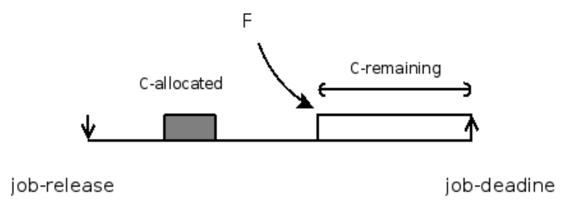


Figure 11.1: Figure showing some of the values needed by the EFF-scheduler and how they relate.

arrange this in a list sorted ascending on $ttf(\tau_{i,j}, t)$ (see Fig. 11.2). Take the first m tasks and assign to the available processors. Create a list of all tasks assigned a CPU, sorted by $ttf_R(\tau_{i,j}, t)$ in descending order. As all tasks will update their allocated time, the relative time to failure will flow the same way, thus the largest ttf_R will accept being scheduled out best. The $ttf(\tau_{i,j}, t)$ for the tasks on the CPU will stay constant, whereas the values for the queued tasks will decrease as t progresses. For every task, set a timeout at $C_R(\tau_{i,j}, t)$ in the future. When running, preempt any task that does not finish when $C_R(\tau_{i,j}, t) = 0$. This indicates a computational overflow and is an erroneous condition. When a task finish and relinquish the CPU, take the next task in the waiting queue with lowest $ttf(\tau_{i,j}, t)$. When a new task arrives, compute the time to failure and compare it to the running tasks. If the time to failure is larger than the smallest existing value, insert it into the list. If it is smaller, preempt the task with largest ttf_R.

We only preempt tasks when they either exceed allocated computation time or another task is released with immediate need for computational resources. Instead of testing all tasks, we preempt the task with the highest $ttf(\tau_{i,j}, t)$ unless the slack, or time to failure is 0. If that is the case, we will have deadline-miss and the new task is added to the head of the queue.

This means that no tasks in the queue will reach 0 time to failure unless we have a schedule with utilization higher than M. If this is the case, we are running a soft real-time schedule and tardiness can be bounded. This will not affect the scheduling decisions.

At tie-breaks:

- The new task has higher $F(\tau_{i,j}, t)$ than the first task in the queue and can be added in the list, compared to the other tasks with the time-to-failure value.
- The new task has lower $F_(\tau_{i,j}, t)$ than the head of the list. If $ttf(\tau_{i,j}, t)$ is

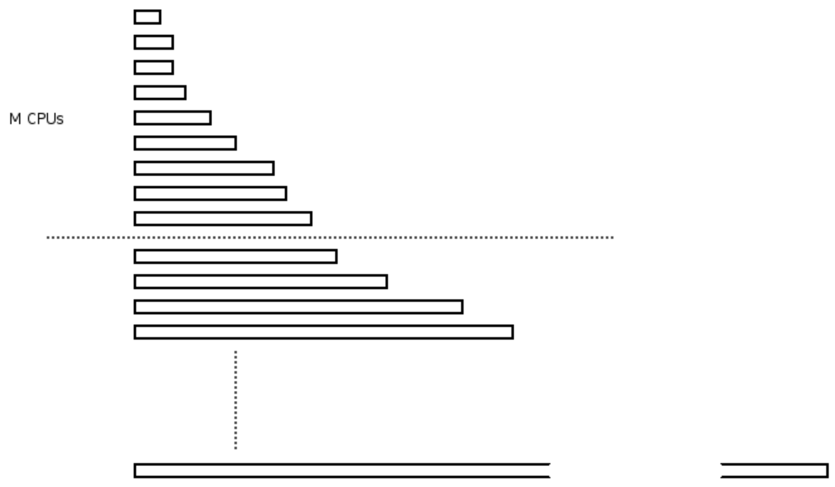


Figure 11.2: A set of tasks, represented by their relative time-to-failure (ttf) values.

lower than the highest of the running tasks, it is scheduled to run on that CPU and ttf-CPU list is re-sorted.

- Otherwise it is added at the head of the wait-queue.

11.4.4 Implementation optimizations for the Linux kernel

This scheduler has been designed based on the experience from this project. The need for a single timer per CPU maps nicely to hrtimers. As the scheduler avoids preemptions unless absolutely necessary, means it wastes as little resources as possible. Datastructures should be easy to pick, and the way the deadlines are used, both relative and absolute, means that comparison of tasks, but running and queued, can be done without expensive integer additions, but as simple integer comparisons. Since we keep two values for time-to-failure, one relative, which makes comparison of running tasks cheap and one absolute which means we do not have to update the queued tasks before they are added to the CPU, comparison can be done fast. All of these features should make the algorithm efficient to implement. That it is global makes it easy to add PEP support, and it avoids load-balancing, yet another expensive operation.

11.4.5 Future work

Future work should focus on handling sporadic tasks as soft real-time tasks that can sustain bounded yet constant deadline tardiness as well as formally prove the correctness of the scheduler. It should also be implemented into the current Linux kernel, with the RT-Preemption patch applied.

11.5 Summary

This chapter concludes the report. We have now not only introduced the scheduler, the kernel, its subsystems and this extension. We have also evaluated the failure of the implementation and pointed to some reasons why this happened.

The inclusion into the kernel repository has also been covered, and finally, EFF was described. This algorithm may prove to solve some problems found with deadline driven algorithms on multicore platforms, and the author has high hope that it will be implemented and included in the kernel repository.

Bibliography

[1] Anderson, J., and Srinivasan, A. Pfair scheduling: beyond periodic task systems. *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on* (2000), 297–306.

[2] Anderson, J., and Srinivasan, A. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *Real-Time Systems, 13th Euromicro Conference on, 2001.* (2001), 76–85.

[3] Anderson, J. H. A new look at pfair priorities. Tech. rep., University of North Carolina, September 1999.

[4] Austad, H. A survey of real-time scheduling algorithms for the linux kernel. Tech. rep., Norwegian University of Science and Technology, December 2008.

[5] Baker, T. An analysis of edf schedulability on a multiprocessor. *Parallel and Distributed Systems, IEEE Transactions on 16*, 8 (Aug. 2005), 760–768.

[6] Björn B. Brandenburg, J. M. C., and Anderson, J. H. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium* (December 2008), pp. 157–169.

[7] Bovet, D. P., and Cesati, M. *Understanding the Linux Kernel*, third ed. O'Reilly Media, Incorporated, 2006.

[8] Buttazzo, G. Rate monotonic vs. edf: Judgement day. *Real-Time Systems 29*, 1 (2005), 5–26.

[9] Corbet, R., and Kroah-Hartman. *Linux Device Drivers*, third ed. O'Reilly Media, Inc., 2005.

[10] Dhall, S. K., and Liu, C. L. On a real-time scheduling problem. *Operations Research 26*, 1 (Jan. - Feb. 1978), 127–140.

[11] Faggioli, D., Lipari, G., and Cucinotta, T. An efficient implementation of the bandwidth inheritance protocol for handling hard and soft real-time applications in the linux kernel. *OSPERT08* (2008).

[12] Gleixner, T., and Niehaus, D. Hrtimers and beyond: Transforming the linux time subsystem. In *Proceedings of the Linux Symposium, 2006, Ottawa, Canada* (2006), pp. 333–346.

[13] Hirata, K., and Goodacre, J. Arm mpcore; the streamlined and scalable arm11 processor core. In *ASP-DAC '07: Proceedings of the 2007 conference on Asia South Pacific design automation* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 747–748.

[14] Joël Goossens1, S. F., and Baruah3, S. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems 25*, 2-3 (November 2003), 187–205.

[15] Kato, S., and Yamasaki, N. Global edf-based scheduling with efficient priority promotion. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on* (Aug. 2008), pp. 197–206.

[16] Kroah-Hartman, G. *Linux Kernel in a Nutshell*, first ed. O'Reilly Media, Inc., December 2006.

[17] Lameter, C. *slab.h.*

[18] Liu, C. L., and Layland, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM 20*, 1 (1973), 46–61.

[19] Love, R. *Linux Kernel Development*, second ed. Novell Press, 2005.

[20] Oh, S.-H., and Yang, S.-M. A modified least-laxity-first scheduling algorithm for real-time tasks. In *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on* (Oct 1998), pp. 31–36.

[21] Philip Holman, J. H. A. Group-based pfair scheduling. *Real-Time Systems 32*, 1-2 (February 2006), 125–168.

[22] Sanjoy K. Baruah, Johannes E. Gehrke, C. G. P. Fast scheduling of periodic tasks on multiple resources. In *In proceedings of the 9th International Parallel Processing Symposium* (April 1995), pp. 280–288.

[23] Schultz, J., Aravamudan, N., and Hart, D. We are not getting any younger: A new approach to time and timers. In *Proceedings of the Linux Sympsium, 2005, Ottawa, Canda* (2005), pp. 219–232.

[24] Srinivasan, A., and Anderson, J. H. Optimal rate-based scheduling on multiprocessors. *J. Comput. Syst. Sci. 72*, 6 (2006), 1094–1117.

[25] Steven Rostedt, D. V. H. Internals of the rt patch. In *Proceedings of the Linux Symposium* (July 2007), pp. 161–172.

[26] Torvalds, L., Ravnborg, S., Dunlap, R., and Zippel, R. *http://lxr.linux.no/linux/Documentation/kbuild/kconfig-language.txt*.

Index

Glossary

Acceptance test	A test used by the scheduler to determine if a new task should be allowed to enter the schedule. If it fails the acceptance test, it is barred from execution on the CPU(s), 29
Agent	A separate entity with a well defined role, performing a task, or a set of tasks in collaboration with other agents, 43
Bin packing problem	An NP-Complete problem. The problem arises when you have several elements and only a finite set of resources to map to. In order to use each resource as effectively as possible, the elements must be ordered in a way so that they are split evenly among the resources, 31
BitKeeper	Distributed SCM, the first source-control management system used by Linux instead of tarballs and patches, 11
CFS	Completely Fair Scheduler, a scheduler using the concept of a "fair share" for each task, assigning a portion of the processors time and making sure each task get the chance to run for the share's time on the CPU, 18
CTW	Cascading Timer Wheel, 83
Critical (sub)job	A (sub)job which *must* be scheduled to run immediately in order to meet its deadline, 35

EDF Scheduler	Earliest Deadline First Scheduler that picks the next task based on an ordered list in ascending order, picking the task that reaches the deadline first. If two or more tasks have identical deadlines, the tie is broken arbitrarily, 29
Exordium	A beginning or introductionary part. Where the reader is prepared for rest of the discussion, and the orator lays out the purpose of the discourse, 1
Forked project	When a subset of the developers in a project (or an outside group of developers) copy the project and start working in a different direction than the original project, 13
GPOS	General Purpose Operating System, 21
Generic IRQs	A layer providing a generic interrupt handling abstraction layer for device drivers, 21
HZ	The frequency of the timer interrupt, a periodic "heartbeat" which must be set at compile time, 83
Hard real-time	When a task cannot miss a single deadline, 6
ISR	Interrupt Service Routine, code located at a specified address ran when a interrupt triggers, 21
Jitter	Variance in response-time for a task, 32
MMU	Memory Management Unit Memory is the part of the processor that is responsible for protecting system resources from unwanted access and also adding the capability for handling virtual memory, 56
Medea	One of the test machines. 2 P4 Xeon 2.40GHz HT, 2GB DDR SDRAM ECC, 2x 36GB 10k IBM Ultrastar SCSI disks, 2x Broadcom 10/100/1000 Mbit NIC, 10

Memory page	A block of memory (RAM) of fixed length which is the smallest divisible unit of memory the MMU will handle. Whereas a memory region will be contiguous in virtual memory, but often fragmented in physical, a page will always be contiguous in both, 56
NACK'ed patch	When a submitted patch is turned down from the responsible maintainer, 12
NUMA	Non Uniform Memory Access, typical for large clusters and SMP machines. Often denoted HPC or "Supercomputers", 104
PEP	Priority Execution Protocol, allowing a task A to run on the expenses of task B in order to release a resource as quickly as possible, 53
PEP	Proxy execution protocol, allowing a blocked task to act as proxy for another task, to speed up the time taken to release a shared resource., 42
PIP	Priority Inheritance Protocol, when a low-priority task A inherits a higher priority task B's status in order for A to run and release a shared resource requested by B, 52
Page frame	A block of memory (RAM) used to store a virtual page, 56
Procmail	Procmail is a mail delivery agent (MDA) or mail filter, a program to process incoming emails on a computer, widely used on Unix systems. — `http://en.wikipedia.org/wiki/Procmail`, 12
RTOS	Real Time Operating System, 6
Race condition	A situation where two or more individual processes (or CPUs) compete for the same resource, and where the order in which access is granted, is not ordered in a deterministic way. This will therefore introduce non-deterministic, and very subtle, latencies, 31

Rate Monotonic Scheduling	A scheduling algorithm where a tasks priority is inferred from the period relative to other tasks, 29
Real Time System	A system that is required to react to external stimuli, including the passing of time, within a time interval dictated by the environment, 28
Resource Reservation	A technique for minimizing or avoiding all together, missed deadlines in a real-time system, 6
Role	A defined behavior for an actor or entity, 43
SLOB Allocator	Simple List Of Blocks, a simple, lightweight SLAB allocator, intended for embedded systems, 60
Sched entity	A `sched_entity` is a schedulable element, 19
Scheduling Policy	A set of "rules" indicating how a set of tasks will be scheduled at run time, 18
Shaky	One of the test machines. P4 2.40GHz, 1GB RAM, 40GB Disk, 10/100Mbit NIC, 10
Spinlock	A lock guarding a shared resource which access should take a very short time. A requesting task will "spin" (busy wait) until the lock is available, in effect halting the execution of that task, or any other on that CPU in the foreseeable future, 23
Staggered scheduler	A staggered scheduler, is a multi core scheduler that spread the scheduling decisions taken on each CPU out in time to avoid that several CPUs try to acquire the resource (runqueue), thus giving rise to a *race condition*, 31
Syscalls	A syscall, or system call, is a way of adding function calls accessible for userspace into the kernel. By raising soft-interrupt `0x80` (x86 arch), the kernel will trap the signal and start executing in kernel-mode, 41

Trapping	The action taken by the kernel when a soft-interrupt triggers a jump in the instruction stream on the CPU, 77
Utilization	A measure for how much work the scheduler and CPU is experiencing. It is given as a fraction of the number of CPUs, 28
clockevents layer	A subsystem in the timer infrastructure, distributing clockevents to all parts of the kernel, 82
hrtimers	High Resolution Timers, a simple yet powerful interface for setting (a)periodic timers in the kernel, 82
jiffy	A term used to describe a short amount of time, in the Linux kernel, it is used to describe the length of a timer-tick (1-10ms), the time between periodic timer interrupts, 82
kobject	A data struct, it is the glue that holds much of the device and SysFS-model together., 74
timeofday layer	A generic layer giving the kernel a standardized interface to time, 82
timeout	the intentional ending of an incomplete task after a time limit considered a long enough for it to end normally, 83
timers	a device used to measure the amount of time something takes, or a device that can be set to sound an alarm after a certain amount of time., 83
vmlinux	The compiled and linked kernel image, 10

Part IV

Appendix

Appendix A

Distributed and cached compilation

This explains how to configure and enable an efficient and distributed build-process to minimize delay and increase productivity by using `distcc` and `ccache` together with `gcc` .

A.1 Cached compilation - `ccache`

In practice - every file you compile is checked against a cache. Or, rather, the check-sum of the preprocessed file[1] is matched against the cache. If the preprocessed file matches a file, the corresponding object-file is simply copied. If not, `gcc` compiles the file, and `ccache` stores the hash and the object-file for later usage. This means that only files that have actually changed, will be recompiled.

A.2 Distributed C/C++ compiler - `distcc`

`distcc` does what `ccache` does not - it spreads out the load to several machines over the network. No shared file-system is needed. `distcc` tries to move the job to several machines to *distribute the load.*

A.3 Basic setup and usage

Setup is straightforward. You install `distcc` (and `ccache` if you are so inclined). No special configuration is needed other than a "per use" configuration.

[1]The result from `gcc -E` .

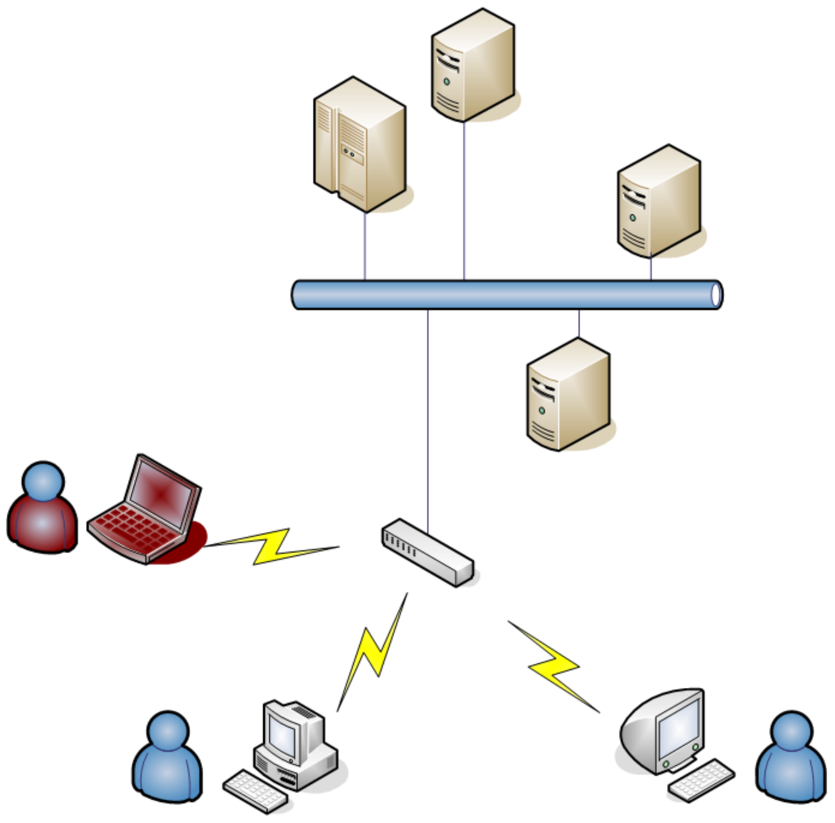


Figure A.1: An example layout of a master-machine and number of satellite computers (servers and other workstations) used for compiling. The red user is the user directing the cluster, the *master*.

`ccache` does not need any configuration. You can tweak settings like the depth of the cache-tree, the size of the cache and the location. A good rule of thumb is to make the cache large enough to contain all cached files, and move it to the fastest disk in your system.

`distcc` requires a bit of configuration. First off you need nodes in your cluster. A node is a machine that accept jobs from the master. The master is the machine where the compilation is started from.

Once `distcc` is installed on the hosts, the master can export jobs in two different ways. The first, and most secure, is to invoke `ssh`. This will tunnel each job. No other configuration of the node is required, `distcc` will start `gcc` properly via `ssh`. The downside of this approach, although it is notably the most secure, is the overhead that `ssh` incurs. The other approach is to configure each node to run a `distcc-daemon`. This is faster, but the pre-compiled files will be sent over the network unencrypted, leaving the compilation exposed to man-in-the middle attacks.

For either one of the setups, the master needs to be told where it can find the nodes. The easiest way is to configure $HOME/.distcc/hosts on the form:

```
# Sample ~/.distcc/hosts file

# Order the examples with the most powerful machine first
# local computer
localhost

# Machine running with distcc started in daemon mode.
# Do not start more than 4 parallell jobs on th enode
192.168.0.1/4

# machine running distccd started with
# distccd --daemon  -j8 --allow 192.168.0.1
# Note that distccd handles maximum job, do not add /n here
192.168.0.2

# ssh to hosts with defined workqueue
username@your.host.com/4
@your.other.host.com/8
```

- Even though it is not necessary to configure `ccache` or `distcc` in any particular way, you can. In that case, look at the webpage of the project[2] or the man-page.
- Both can be configured to replace `gcc` entirely (which will cause `ccache`/`distcc` to be invoked every time you compile **anything**. However, this is not always what you want, and for that reason, it is not done here (i.e. we use `CC=''distcc ccache gcc''` in the make invocation).
- `distcc` must be configured to use separate hosts. You can do this in two different ways, where the preferred one, is to update the `/.distcc/hosts` .
- Finally, `distcc` requires passphrase-less ssh-keys. This is due to the fact that `distcc` cannot be configured to pass along neither password nor passphrase.

Example A i)

1. Create ssh-key silently to standard file
   `ssh-keygen -q -t rsa -b 2048 -f /.ssh/id_rsa`
   Note that this will create a key **with** passphrase (which is always a good idea). To let distcc use this, you must either create without passphrase (add -N ") or use key-agent.

[2]`http://ccache.samba.org/`

2. Configure /.distcc/hosts
3. Distribute ssh-key from the master to every node in the cluster that requires `ssh`-login
4. Do a `make -j32 CC=̈distcc ccache gcc̈` and watch the compilation fly.

Appendix B

Module code

This chapter contains the module code from the different sections. This code is also included in the attachment (on the CD-ROM). The makefiles for the different modules are not show. These are attached, but will not be discussed. See [9] for an excellent coverage of these.

B.1 Slab allocator

```c
/*
 *                              slab.c
 *
 * This module demonstrates the usage of the slab allocator. The result is
 * printed to kern.log at insertion and removal. It does not provide any other
 * interface. The sole purpose is to make kernel context readily available.
 *
 *
 * Copyright 2008-2009 Henrik Austad <henrik@austad.us>
 * Norwegian University of Science and Technology
 *
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; see the file COPYING.  If not, write to
 * the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/slab.h>


/* struct object
 *
 * Small struct to demonstrate how the slab can allcoate memory.
 * It creates a singly linked list and a status counter. The size is set so
 * small (8 bytes) that kmalloc will waste a lot of space (24 bytes pr. object).
 */
struct object {
        struct object *next;
        unsigned long counter;
};

static struct kmem_cache *object_cache;
static struct object *head;

/* slab_test_init - module initialization function
 *
 * This function will create a kmem_cache, with hardware cache alignment, and
 * allocate a set of struct object from it. To illustrate, each object will be
 * given a unique number in increasing order.
 */
static int __init slab_test_init(void)
{
        int c = 0;
        struct object *new, *prev;
        object_cache = kmem_cache_create("ObjectCache",
                                         sizeof(struct object),
                                         0, SLAB_HWCACHE_ALIGN, NULL);
        if (!object_cache)
                goto faulty_exit_1;
        printk(KERN_INFO "Object-cache successfully created!\n");

        head = (struct object *)kmem_cache_alloc(object_cache, GFP_KERNEL);
        if (head) {
                printk(KERN_INFO "Got object-memory from slab-cache\n");
                head->next = NULL;
                head->counter = 0;
        }
        new = head;
        printk(KERN_INFO "OK, got the head, creating list, size of each element is %u\n",
               sizeof(struct object));
        while (c++ < 10) {
                prev = new;
                new = (struct object *)kmem_cache_alloc(object_cache, GFP_KERNEL);
                if (!new)
                        break;
                prev->next = new;
                new->counter = prev->counter + 1;
        }
        printk(KERN_INFO "Allocated %d objects from the cache\n", c);
        return 0;
faulty_exit_1:
        return -ENOMEM;
}

/* slab_test_exit - module exit function
 *
 * When module is removed, free and unlink all elements in the list and return
 * the cache to the kernel.
 */
static void __exit slab_test_exit(void)
{
        struct object *tmp, *old;
        if (object_cache) {
                if (head) {
                        tmp = head->next;
                        while(tmp) {
                                old = tmp;
                                tmp = tmp->next;
                                old->next = NULL;
                                printk(KERN_INFO "release: %lu\n", old->counter);
                                kmem_cache_free(object_cache, old);
                        }
                        kmem_cache_free(object_cache, (void *)head);
                        printk("Freed head and returned it to the cache\n");
                }
                kmem_cache_destroy(object_cache);
        }
        printk(KERN_INFO "Goodbye world\n");
        return;
}

/* notify the module interface which functions that are init and exit */
module_init(slab_test_init);
module_exit(slab_test_exit);

/* some extra info about author an license (if not GPL, kernel will be "tainted") */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Henrik Austad");
```

B.2 Linked Lists

```
/*
 *                              linked_list.c
 *
 * Demonstrate the usage of the linked_list system in the kernel. This module
 * will allocate a set of list elements and keep them in sorted order, first on
 * the uid-key, and then on uid2.
 *
 * Copyright 2008-2009 Henrik Austad <henrik@austad.us>
 * Norwegian University of Science and Technology
 *
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; see the file COPYING.  If not, write to
 * the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/list.h>

struct test_struct {
        int uid;
        struct list_head list;
        int uid2;
};
static struct test_struct *head;
static int list_size;


/** compare_test - compares to test_structs
 *
 * @ts1      : first element (struct) to compare
 * @ts2      : second element
 *
 * returns
 * negative : ts1 is smaller than ts2
 * zero     : they are equal (or one of them is NULL and comparison does not make
 *            sense).
 * positive : ts1 is greater than ts2
 */
int compare_test(struct test_struct *ts1, struct test_struct *ts2)
{
        if (!ts1 || !ts2)
                return 0;
        if (ts1->uid == ts2->uid)
                return ts1->uid2 - ts2->uid;
        return ts1->uid - ts2->uid;
}
```

```

/** remove - remove element from the list
 *
 * 1 on success, 0 on failure (i.e. not found)
 */
int remove(int uid1, int uid2)
{
        struct test_struct *ts = NULL;
        struct list_head *lh = NULL;
        unsigned char found = 0;
        list_for_each(lh, &head->list) {
                ts = list_entry(lh, struct test_struct, list);
                if (ts->uid == uid1 && ts->uid2) {
                        found = 1;
                        break;
                }
        }
        if (found && ts) {
                list_del(&ts->list);
                kfree(ts);
                list_size--;
                return 1;
        }
        return 0;
}

/**
 * add_new - add a new element to the list
 *
 * This function shall create a new test_struct, initialize the embedded
 * list_head and insert it into the queue. It will do this in sorted order.
 *
 * @uid  : primary uid (key)
 * @uid2 : secondary key
 *
 * Return 0 on sucess, negative on fault
 */
int add_new(int uid, int uid2)
{
        unsigned char added     = 0;
        struct test_struct *ts  = NULL;
        struct list_head   *lh  = NULL;
        struct test_struct *tmp = (struct test_struct *)
                kmalloc(sizeof(struct test_struct), GFP_KERNEL);

        if (!tmp)
                return -ENOMEM;
        if (!head) {
                printk(KERN_ALERT "Cannot add to non-existing list!\n");
                return -EFAULT;
        }
        tmp->uid = uid;
        tmp->uid2 = uid2;
        INIT_LIST_HEAD(&tmp->list);
        if (list_empty(&head->list))
                list_add(&tmp->list, &head->list);
        else {
                list_for_each(lh, &head->list) {
                        ts = list_entry(lh, struct test_struct, list);
                        if (compare_test(tmp, ts) < 0) {
                                list_add_tail(&tmp->list, lh);
                                added = 1;
                                break;
                        }
                }
                if (!added) {
                        /* if the number is greater than all the other elements
                         * in the list, it will not be added during the normal
                         * loop, and hence, we need to add it before head, or
                         * at the tail, as this is a circular list */
                        list_add_tail(&tmp->list, &head->list);
                }
        }
        list_size++;
        return 0;
}

/**
 * print_all - print all elements in the list
 *
 * This function will print all elements in the list in a standard way. This is
 * great for debugging, but if the list is very large, this adds a lot of text
 * to kern.log (and dumping large amount of text in kernel-mode is generally
 * frowned upon).
 *
 * @params : void
 * @returns: void
 */
void print_all(void)
{
        struct test_struct *ts;
        struct list_head *lh;

        if (!head) {
                printk("No head defined!\n");
                return;
        }
        if (unlikely(list_empty(&head->list))) {
                printk(KERN_INFO "List is empty!\n");
                return;
        }

        list_for_each(lh, &head->list) {
                ts = list_entry(lh, struct test_struct, list);
                printk(KERN_INFO "UID: %d UID2: %d\n", ts->uid, ts->uid2);
        }
}


/**
 * llist_init - init linked-list module
 */
int __init llist_init(void)
{
        int c = 0;
        int limit = 7;
        head = (struct test_struct *)kmalloc(sizeof (struct test_struct), GFP_KERNEL);
        if (!head)
                return -ENOMEM;
        list_size = 0;
        head->uid  = 0;
        head->uid2 = 0;
        /* init head */
        INIT_LIST_HEAD(&head->list);

        /* create a set of tasks */
        for (;c<limit; c++)
                if (add_new(c%5, c*2))
                        break;
        add_new(1,23);
        add_new(5,99);
        add_new(1,0);
        printk(KERN_INFO "Removed 1,1:  %s\n" , (remove(1,1)  ? "OK" : "NOK"));
        printk(KERN_INFO "Removed 1,23: %s\n", (remove(1,23) ? "OK" : "NOK"));
        printk(KERN_INFO "Removed 6,23: %s\n", (remove(6,23) ? "OK" : "NOK"));

        return 0;
}

void __exit llist_exit(void)
{
        struct test_struct *ts = NULL;
        int counter = 0;

        /* if the list is not too long, print all to show that it is sorted */
        if (list_size < 10)
                print_all();

        if (head) {
                while(!list_empty(&head->list)) {
                        ts = list_first_entry(&head->list, struct test_struct, list);
                        list_del(&ts->list);
                        kfree(ts);
                        counter++;
                }
                printk(KERN_INFO "Destroyed %d elements from the list\n", counter);
                kfree(head);
        }
        printk(KERN_INFO "Goodbye, I'm no longer linked\n");
}

/* set init, exit and moduleinfo */
module_init(llist_init);
module_exit(llist_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Henrik Austad");
```

B.3 Red black trees

```
/*
 *                              rbtree_mod.c
 *
 * Demonstrate the usage of rbtree.h in the kernel.
 *
 *
 * Copyright 2008-2009 Henrik Austad <henrik@austad.us>
 * Norwegian University of Science and Technology
 *
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; see the file COPYING.  If not, write to
 * the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/rbtree.h>
#include <linux/string.h>

/* emulate the pfair_subjob and keep a sorted tree based on these subjobs */
struct rb_task {
        unsigned long long subdl;
        unsigned long long wcet;
        unsigned long long deadline;
        unsigned int pid;
        struct rb_node node;
};
static struct rb_root active = RB_ROOT;

/**
 * print_rbtask - print an rb_task to dmesg
 *
 * @rbt:          The rb_task to print
 */
static inline void print_rbtask(struct rb_task *rbt)
{
                printk(KERN_INFO "[%d] %llu %llu %llu\n",
                        rbt->pid,
                        rbt->subdl,
                        rbt->deadline,
                        rbt->wcet);
}

/**
 * compare_raw   - compare a rb_task to a set of values and determine if they are
 *                 equal according to a predefined set of rules
 * rbt:          - The rb_task to test
 * subdl:        - matching sub_deadline
 * wcet:        - matching worst case execution time
 * deadline:    - matching deadline (major deadline)
 *
 * The compare is optimized for the tree-setup, and 'smaller' nodes should be
 * placed to the left. Hence, if rbt's values are smaller than the provided
 * values, negative is returned. 0 if equal and positive if rbt should be moved
 * right.
 *
 * We need this in a dedicated function, as compare_tasks() is not the only
 * function using compare_raw()
 */
static inline int compare_raw(struct rb_task *rbt,
                      unsigned long long subdl,
                      unsigned long long wcet,
                      unsigned long long deadline)
{
        if (rbt->subdl != subdl)
                return rbt->subdl - subdl;
        if (rbt->deadline != deadline)
                return rbt->deadline - deadline;
        if (rbt->wcet != wcet)
                return wcet - rbt->wcet;
        return 0;
}

/**
 * compare_tasks- compare two rb_tasks and determine difference between them.
 *
 * rb1:         - First rb_task
 * rb2:         - Second rb_task
 *
 * The compare is optimized for the tree-setup, and 'smaller' nodes should be
 * placed to the left. If rb1 is smaller than rb2, negative is returned. This
 * function uses compare_raw, and thus, the same rules apply.
 */
int compare_tasks(struct rb_task *rb1, struct rb_task *rb2)
{
        if (!rb1 || !rb2)
                return 0;
        return compare_raw(rb1, rb2->subdl, rb2->wcet, rb2->deadline);
}

/**
 * rbt_search   - search the tree and find a task with matching attributes
 *
 * @root        - the root of the tree
 * @subdl       - sub_deadline
 * @wcet
 * @deadline
 * @pid         - this is the key element, as this is actually the only unique
 *                attribute. However, the tree is kept sorted by the other
 *                attributes, so we need all of these in order to find the
 *                node.
 *
 * If rbt_search finds a matching task, it is returned, otherwise NULL.
 */
struct rb_task * rbt_search(struct rb_root *root,
                            unsigned long long subdl,
                            unsigned long long wcet,
                            unsigned long long deadline,
                            unsigned int pid)
```

```
{
        int compres = 0;
        struct rb_node *tmp = root->rb_node;
        struct rb_task *rbt = NULL;
        while(tmp) {
                rbt = container_of(tmp, struct rb_task, node);
                compres = compare_raw(rbt, subdl, wcet, deadline);
                if (compres == 0 && rbt->pid == pid)
                        return rbt;
                /* rbt smaller than values, go right */
                if (compres <= 0)
                        tmp = tmp->rb_right;
                else
                        tmp = tmp->rb_left;
        }
        return NULL; /* not found */
}
/**
 * rbt_insert   - insert a new rb_task into the tree.
 *
 * @root        - the root of the tree
 * @rbt         - the new node to insert into the tree.
 *
 * This function differs slightly from other rbtree insertion methods as it
 * allows for several 'identical' tasks to be inserted into the tree. This is
 * because 2 distinct tasks can have the *exact* same profile, even if they are
 * separate tasks.
 *
 * If rbt is already present in the tree, it will detect this and abort. Note,
 * it will only detect this if it *is* a duplicate, or if it happens to
 * encounter another tasks, but with identical pid. The latter should not pose a
 * problem, as pid is guaranteed to be unique.
 */
void rbt_insert(struct rb_root *root, struct rb_task *rbt)
{
        struct rb_node **tmp_link = &(root->rb_node);
        struct rb_node *parent = *tmp_link;
        struct rb_task *tmp;
        int compres  = 0;
        while (*tmp_link) {
                parent = *tmp_link;
                tmp = rb_entry(parent, struct rb_task, node);
                if (unlikely(tmp->pid == rbt->pid)) {
                        printk(KERN_ALERT "Trying to insert a task already "\
                               "present in the tree!\n");
                        return;
                }
                compres = compare_tasks(rbt, tmp);
                if (compres < 0)
                        tmp_link = &((*tmp_link)->rb_left);
                else
                        tmp_link = &((*tmp_link)->rb_right);
        }
        rb_link_node(&(rbt->node), parent, tmp_link);
        rb_insert_color(&(rbt->node), root);
}

/**
 * rbt_delete   - remove an rb_task from the tree and return it.
 *
 * @root:       the root of the tree
 * @subdl:       current subdeadline for the task
 * @wcet:        Worst Case Execution Time
 * @deadline:    Full deadline
 * @pid:         PID of the task
 *
 * This function searches through the tree to find the task with pid_t pid. If
 * found, it will be unlinked from the tree and returned.
 *
 * It uses the parameters as a guide to navigate through the tree, so if these
 * are off, it will *not* find the task.
 *
 * This method highlights one of the key problems with the rb-tree (as well as
 * a lot of other data-structures) - if you have to use something different as
 * search-key, things quickly becomes tedious and error-prone.
 */
struct rb_task * rbt_delete(struct rb_root *root,
                            unsigned long long subdl,
                            unsigned long long wcet,
                            unsigned long long deadline,
                            unsigned int pid)
{
        struct rb_task *remove = rbt_search(root, subdl, wcet, deadline, pid);
        if (remove)
                rb_erase(&remove->node, root);
        return remove;
}
/**
 * rbt_delete_full      fully remove the rb_task from the tree
 *
 * If a matching tasks is found, not only is it removed from the tree, the
 * memory is freed as well.
 */
void rbt_delete_full(struct rb_root *root,
                            unsigned long long subdl,
                            unsigned long long wcet,
                            unsigned long long deadline,
                            unsigned int pid)
{
        struct rb_task *remove = rbt_delete(root, subdl, wcet, deadline, pid);
        if (remove)
                kfree(remove);
}

int __init rbtree_init(void)
{
        struct rb_task *tmp = NULL;
        struct rb_task *sres = NULL;
        int c = 20;

        /* Create a set of elements for the tree. */
        while (c-->0) {
               tmp = (struct rb_task *)kmalloc(sizeof(struct rb_task), GFP_KERNEL);
               if (!tmp)
                       goto rb_err;
               tmp->subdl = c%13;
               tmp->wcet = (c*10) %19;
               tmp->deadline = (c*50)%29;
               tmp->pid = c;
               rbt_insert(&active, tmp);
        }

        /* make sure it works, search and remove a task guaranteed to be there. */
        tmp = (struct rb_task *)kmalloc(sizeof(struct rb_task), GFP_KERNEL);

        /* If this fails, just drop out of the init, most of the tree is
         * populated anyway */
        if (!tmp)
                return 0;
        tmp->subdl      = 5;
        tmp->wcet       = 50;
        tmp->deadline   = 250;
        tmp->pid        = 500;
        rbt_insert(&active, tmp);
        sres = rbt_delete(&active, 5, 50, 250, 500);
        if (sres) {
                printk(KERN_INFO "Got match to remove!\n");
                print_rbtask(sres);
                kfree(sres);
        }
        return 0;
rb_err:
        printk(KERN_ALERT "Error in allocating memory for rb_task\n");
        return -ENOMEM;
}

void __exit rbtree_exit(void)
{
        struct rb_task *rbt = NULL;
        struct rb_node *rb1 = rb_first(&active);
        struct rb_node *victim;

        printk(KERN_INFO "Cleaning up the rb-tree, removing all nodes\n");
        while (rb1) {
                rbt = rb_entry(rb1, struct rb_task, node);
                print_rbtask(rbt);
                victim = rb1;
                rb1 = rb_next(rb1);
                if (victim) {
                        rb_erase(victim, &active);
                        kfree(container_of(victim, struct rb_task, node));
                }
        }
        printk(KERN_INFO "Done cleaning up the tree. Ready for exit.\n");
}

module_init(rbtree_init);
module_exit(rbtree_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Henrik Austad");
```

B.4 Sysfs module

```
/*
 *                               sysfs_module.c
 *
 * Add new kobject ('pfair') under /sys/kernel/ and add two new attributes
 *      - sched_gran_us
 *      - sched_idle_us
 * These are exposed 0644, and root may change the values. Changes will be
 * logged to kern.log
 *
 * Copyright 2008-2009 Henrik Austad <henrik@austad.us>
 * Norwegian University of Science and Technology
 *
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; see the file COPYING.  If not, write to
 * the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <linux/kobject.h>
#include <linux/string.h>       /* for memset, strcmp */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/sysfs.h>
#include <linux/slab.h>         /* for kmalloc */

/* the granularity of the scheduler, i.e. the length of each timeslice given to
 * the subjobs */
long sched_gran_us = 0;

/* if time left before next hrtimertick in pfair-scheduler is less than this,
 * run a busy loop until the time is right, if it is more, invoke the
 * schedule() to find another task to run for a few microseconds before the
 * bell goes off again. */
long sched_idle_us = 0;

static ssize_t
pfair_store(struct kobject *kobj, struct kobj_attribute *attr,
            const char *buffer, ssize_t size)
{
        unsigned long res = 0;
        ssize_t length_read = -1;
        if (strcmp(attr->attr.name, "sched_gran_us") == 0) {
                length_read = sscanf(buffer, "%lu\n", &res);
                if (length_read > 0 && res > 1000) {
                        sched_gran_us = res;
                }
        }  else if (strcmp(attr->attr.name , "sched_idle_us") == 0) {
                length_read = sscanf(buffer, "%lu", &res);
                if (length_read > 0 && res >= 0)
                        sched_idle_us = res;
        } else {
                printk(KERN_ALERT "Unknown sysfs-entry given to sysfs_module\n");
        }
        /* return length_read; */
        /* to avoid endless loops, return the size and be done with it */
        return size;
}

static ssize_t
pfair_show(struct kobject *kobj, struct kobj_attribute *attr,
           char *buffer)
{
        if (strcmp(attr->attr.name, "sched_gran_us") == 0)  {
                return sprintf(buffer, "%lu", sched_gran_us);
        }
        else if (strcmp(attr->attr.name , "sched_idle_us") == 0)
                return sprintf(buffer, "%lu", sched_idle_us);
        return 0;
}

/* create the attributes and initialize with __ATTR */
static struct kobj_attribute sgattr =
        __ATTR(sched_gran_us, 0644, pfair_show, pfair_store);
static struct kobj_attribute siattr =
        __ATTR(sched_idle_us, 0644, pfair_show, pfair_store);
static struct kobject *kobj;

static int __init loc_init(void)
{
        int retval = 0;
        /* Create a folder in /sys/kernel/ */
        kobj = kobject_create_and_add("pfair", kernel_kobj);
        if (!kobj)
                return -ENOMEM;
        retval = sysfs_create_file(kobj, &sgattr.attr);
        if (retval)
                goto error_exit;

        retval = sysfs_create_file(kobj, &siattr.attr);
        if (retval)
                goto error_exit;

        /* retval = sysfs_create_group(kobj, &attr_group); */
        return retval;
error_exit:
        printk(KERN_ALERT "Error in initializing sysfs_module!\n");
        kobject_put(kobj);
        return retval;
}

static void __exit loc_exit(void)
{
        kobject_put(kobj);
}

module_init(loc_init);
module_exit(loc_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Henrik Austad");
```

B.5 Timer module

```
#include <linux/hrtimer.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/timer.h>

/*
 *                               timer_module.c
 *
 * Small test-module for hrtimers
 *
 * Copyright 2008-2009 Henrik Austad <henrik@austad.us>
 * Norwegian University of Science and Technology
 *
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; see the file COPYING.  If not, write to
 * the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.
 */

ktime_t time_offset;
ktime_t last_release;
unsigned long delay_ns;
#define REL_BUFF_SIZE_BITS 10
#define REL_BUFF_SIZE 1<<REL_BUFF_SIZE_BITS

/* to calculate the difference between desired delay and actual delay */
static signed long long *diff_buffer;
static unsigned int diffb_index;

struct container {
        struct hrtimer hrtimer;
        ktime_t last_release;
        unsigned long long last_release_ns;
        /* delay is adjusted in order to try to match delay_ns set by user/init */
        ktime_t delay;

        /* the actual delay in ns wanted */
        unsigned long long delay_ns;
        signed long long min;
        signed long long max;
        unsigned long long handled_interrupts;
};
struct container *data_container;

#define NS_IN_US 1000
#define NS_IN_MS NS_IN_US * 1000
#define NS_IN_SEC NS_IN_MS * 1000

```

```
/**
 * hrtimer_handler - hrtimer callback function
 *
 * This is the function that will be called on every timer event.
 *
 * @hrt:                 the registred struct hrtimer
 * @hrtimer_restart :    enum, if HRTIMER_RESTART, the hrtimer subsystem will call
 *                       the handler again immediately
 */
enum hrtimer_restart hrtimer_handler(struct hrtimer *hrt)
{
        struct container *c = NULL;
        int cpu = -1;
        unsigned long long last_time_ns = 0;

        if(!hrt) {
                printk(KERN_ALERT "[timer_module] No data received in "\
                       "hrtimer_handler!\n");
                return HRTIMER_NORESTART;
        }
        c = container_of(hrt, struct container, hrtimer);
        if (!c) {
                printk(KERN_ALERT "[timer_module] container is null, "\
                       "even though we have the struct hrtimer\n");
                return HRTIMER_NORESTART;
        }

        /* schedule next timer now to avoid noise from this function
         *
         * WARNING: If delay is set so low that the function uses longer time
         * than the delay, this will cause a cascading effect and the system
         * will crash as the kernel-stack will overflow.
         */
        if (hrtimer_start(&c->hrtimer, c->delay, HRTIMER_MODE_REL))
                return HRTIMER_RESTART;

        diffb_index = ++diffb_index % REL_BUFF_SIZE;
        /* save last release so we can compute the diff */
        last_time_ns = c->last_release_ns;
        c->last_release = hrtimer_cb_get_time(hrt);
        c->last_release_ns = ktime_to_ns(c->last_release);
        diff_buffer[diffb_index] = c->last_release_ns - last_time_ns - c->delay_ns;

        if (c->delay_ns > 100*NS_IN_MS) {
                cpu = smp_processor_id();
                printk(KERN_INFO "[timer_module] Time: %llu on CPU %d, diff: %lld\n",
                       ktime_to_ns(c->last_release),
                       cpu, diff_buffer[diffb_index]);
        }
        if (c->max < diff_buffer[diffb_index])
                c->max = diff_buffer[diffb_index];
        if (c->min > diff_buffer[diffb_index])
                c->min = diff_buffer[diffb_index];
        c->handled_interrupts++;

        return HRTIMER_NORESTART;
}

/* initializing function
 *
 * Before starting the timer, create and initialize the data-elements needed.
 */
static int __init timer_init(void)
{
        int res = 0;
        data_container = (struct container *)
                kmalloc(sizeof(struct container), GFP_KERNEL);
        if (!data_container) {
                printk(KERN_ALERT "[timer_module] "\
                       "Could not allocate memory to data_container!\n");
                return -ENOMEM;
        }
        data_container = (struct container *)
                memset(data_container, 0, sizeof(struct container));

        /* set default values in the data_container */
        data_container->delay_ns        = 1 * NS_IN_MS;
        data_container->delay           = ns_to_ktime(data_container->delay_ns);
        data_container->last_release    = ktime_get();
        data_container->last_release_ns = ktime_to_ns(data_container->last_release);
        data_container->max             = 0;
        data_container->min             = data_container->delay_ns;
        data_container->handled_interrupts = 0;


        /* Create and initalize the diff_buffer, where we calculate the offset
         * and errors while the timer is running */
        diff_buffer = (signed long long *)
                kmalloc(sizeof(unsigned long long) * REL_BUFF_SIZE, GFP_KERNEL);
        if (!diff_buffer) {
                kfree(data_container);
                printk(KERN_ALERT "[timer_module] "\
                       "Could not allocate memory to diff_buffer!\n");
                return -ENOMEM;
        }
        diff_buffer = (signed long long *)
                memset(diff_buffer, 0, sizeof(signed long long) * REL_BUFF_SIZE);
        diffb_index = 0;


        hrtimer_init(&data_container->hrtimer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
        data_container->hrtimer.function = hrtimer_handler;
#ifdef CONFIG_PREEMPT_RT
        data_container->hrtimer.irqsafe = 1;
#endif
        res = hrtimer_start(&data_container->hrtimer,
                            data_container->delay,
                            HRTIMER_MODE_REL);
        printk(KERN_INFO "[timer_module] Started timer %s\n",
               (res) ? "NOK" : "OK");
        return res;
}

/* timer_exit
 *
 * When the module is removed, it will print out the global min and max error
 * (the difference between actual release and desired release). The resources
 * held are then freed and the module exits.
 */
static void __exit timer_exit(void)
{
        int index = 0;
        signed long long max, min;
        signed long long total_error = 0;
        printk("[timer_module] Removing timer\n");
        if (data_container) {
                hrtimer_cancel(&data_container->hrtimer);
                /* Calculate the avg delay for the last REL_BUFF_SIZE timers */
                if (diff_buffer) {
                        max = diff_buffer[0];
                        min = diff_buffer[0];
                        for(;index<REL_BUFF_SIZE;index++) {
                                total_error += diff_buffer[index];
                                if (max < diff_buffer[index])
                                        max = diff_buffer[index];
                                else if (min > diff_buffer[index])
                                        min = diff_buffer[index];
                        }
                        printk(KERN_INFO "[timer_module] delay: %llu "\
                               "global max: %lld local max %lld "\
                               "global min: %lld local min %lld "\
                               "avg: %lld size: %d intr: %llu\n",
                               data_container->delay_ns,
                               data_container->max,
                               max,
                               data_container->min,
                               min,
                               total_error>>REL_BUFF_SIZE_BITS,
                               REL_BUFF_SIZE,
                               data_container->handled_interrupts);
                        kfree(data_container);
                } else {
                        printk(KERN_ALERT "[timer_module] release: !data-container\n");
                }
        }
        printk(KERN_INFO "[timer_module] Goodbye world\n");
        return;
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Henrik Austad");
MODULE_DESCRIPTION("Timer-functionality test module");
module_init(timer_init);
module_exit(timer_exit);
```

B.6 Bouncing timer module

```c
#include <linux/hrtimer.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/ktime.h>
#include <linux/sched.h>
#include <linux/timer.h>
#include <linux/time.h>

/*
 *                              timer_bounce.c
 *
 * Small test-module for hrtimers that bounce between CPUs
 *
 *     +---------+        +---------+
 *     |  CPU 0  |  -->   |  CPU 1  |
 *     +---------+        +---------+
 *         ^                   |
 *         |                   v
 *     +---------+        +---------+
 *     |  CPU 2  |  <--   |  CPU 3  |
 *     +---------+        +---------+
 *
 * Copyright 2008-2009 Henrik Austad <henrik@austad.us>
 * Norwegian University of Science and Technology
 *
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; see the file COPYING.  If not, write to
 * the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.
 */

/* simple container for  the timer
 *
 * @hrtimer      - the timer for the given CPU
 * @period       - time between each event (from CPU to CPU).
 * @last_started - absolute time (gettimeofday()) when the time last triggered
 *                 on *this* CPU.
 */
struct container {
        struct hrtimer hrtimer;
        ktime_t period;
        struct timespec last_started;
};
static struct container *timers[CONFIG_NR_CPUS];

#define NS_IN_US 1000
#define NS_IN_MS NS_IN_US * 1000
#define NS_IN_SEC NS_IN_MS * 1000

```

```

/**
 * remote_timer_start - start a timer on another CPU
 *
 * This function will trigger a timer on *another* CPU to start.
 */
void remote_timer_start(void *data)
{
        struct container *c = (struct container *)data;
        if (c) {
                hrtimer_start(&c->hrtimer, c->period, HRTIMER_MODE_REL);
        }
}

/**
 * hrtimer_restart - timer callback function
 *
 * When a timer event triggers, this callback function will be called, with
 * struct hrtimer *hrt as argument.
 *
 * We update current time and shows the time since last time.
 */
enum hrtimer_restart hrtimer_handler(struct hrtimer *hrt)
{
        int cpu, next_cpu;
        struct timespec time_now, time_delta;
        cpu = get_cpu();
        put_cpu_no_resched();

        /* find next CPU */
        next_cpu = (cpu+1) % num_online_cpus();

        /* Get time since last time the timer triggered on this CPU, and update
         * so we remember until next time */
        getnstimeofday(&time_now);
        time_delta = timespec_sub(time_now, timers[cpu]->last_started);

        printk(KERN_INFO "Time since last timer on this CPU (%d): %lld\n", cpu, timespec_to_ns(&time_
        timers[cpu]->last_started = time_now;

        /* single-core setups does not really like smp_call_function_single */
        if (next_cpu == cpu)
                hrtimer_start(&timers[cpu]->hrtimer, timers[cpu]->period, HRTIMER_MODE_REL);
        /* call function on specific CPU */
        else if (smp_call_function_single(next_cpu, remote_timer_start, timers[next_cpu], 1))
                printk(KERN_ALERT "Could not restart timer, aborting\n");

        return HRTIMER_NORESTART;
}

/**
 * timer_bounce_init - init the timer bouncing
 *
 * This initializes the boucing. A timer is started on a CPU and when that
 * triggers, the handler will schedule a timer on the next timer, effectively
 * turning then timing into a token passed around between the cores.
 */
static int __init timer_bounce_init(void)
{
        int cpu, this_cpu;
        int num_cpus;
        u64 period;
        struct timespec time_now;

        /* get current cpu and release lock without triggering resched */
        this_cpu        = get_cpu();
        put_cpu_no_resched();
        cpu             = this_cpu;
        num_cpus        = num_online_cpus();
        period          = 1*NS_IN_SEC;
        getnstimeofday(&time_now);
        printk(KERN_INFO "Available cpus: %d\n" , CONFIG_NR_CPUS);
        printk(KERN_INFO "online cpus: %d\n"    , num_cpus);
        printk(KERN_INFO "This CPU: %d\n"       , cpu);
        printk(KERN_INFO "Period: %llu\n"       , period);

        /* walk the cpus and init the timers */
        for_each_online_cpu(cpu) {
                timers[cpu] = (struct container *)kmalloc(sizeof(struct container), GFP_KERNEL);
                if (timers[cpu]) {
                        timers[cpu]->period = ns_to_ktime(period);
                        timers[cpu]->last_started = time_now;
                        hrtimer_init(&timers[cpu]->hrtimer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
                        timers[cpu]->hrtimer.function = hrtimer_handler;
#ifdef CONFIG_PREEMPT_RT
                        timers[cpu]->hrtimer.irqsafe = 1;
#endif
                        printk(KERN_INFO "Adding timer on CPU %d in %llu\n",
                               cpu, ktime_to_ns(timers[cpu]->period));
                } /* endif */
        } /* end for_each */

        /* start the timer on this CPU immediately */
        if (hrtimer_start(&timers[this_cpu]->hrtimer, ns_to_ktime(0), HRTIMER_MODE_REL))
                printk(KERN_ALERT "Could not start timer\n");
        return 0;
}

/**
 * timer_bounce_exit - close down module
 *
 * Stop all timers and free the struct container in the array
 */
static void __exit timer_bounce_exit(void)
{
        int c = 0;
        for(;c<CONFIG_NR_CPUS;c++) {
                if (timers[c]) {
                        hrtimer_cancel(&timers[c]->hrtimer);
                        kfree(timers[c]);
                }
        }
        printk(KERN_INFO "[timer_bounce] Module removed\n");
}


MODULE_LICENSE("GPL");
MODULE_AUTHOR("Henrik Austad");
MODULE_DESCRIPTION("Timer-bouncing test module");
module_init(timer_bounce_init);
module_exit(timer_bounce_exit);
```

Appendix C

Various scripts

This appendix lists some of the scripts used in the project.

C.1 kinstall.sh

```
#!/bin/bash
#
# Script for automating the build and installation of new kernels.
#
# This script is based on work by Greg Kroah-Hartman,
#               http://www.kroah.com/lkn/
# In the book Linux Kernel Development.
#
# It has been modified and extended to provide some more feedback to the
# user by Henrik Austad, 2009
#
# This work is licensed under the Creative Commons Attribution-ShareAlike
# 2.5 License. To
# view a copy of this license, visit
# http://creativecommons.org/licenses/by-sa/2.5/ or send a letter
# to Creative Commons, 543 Howard Street, 5th Floor, San Francisco,
# California, 94105, USA.

if [ ! $# -eq 1 ];then
        echo "Need linux-target folder"
        exit
fi

# enter target
cd $1

echo "Making modules_install"
make install
sudo make modules_install

# Fin version tags:
for TAG in VERSION PATCHLEVEL SUBLEVEL EXTRAVERSION ; do
        eval `sed -ne "/^$TAG/s/ //gp" Makefile`
done
SRC_RELEASE=$VERSION.$PATCHLEVEL.$SUBLEVEL$EXTRAVERSION

ARCH=`grep "CONFIG_ARCH " include/linux/autoconf.h | cut -f 2 -d "\""`

# Copy kernel-image, System.map to /boot
sudo cp -v arch/x86/boot/bzImage /boot/vmlinuz-$SRC_RELEASE
sudo cp -v System.map /boot/System.map-$SRC_RELEASE

echo "Installing $SRC_RELEASE for $ARCH, fixing grub"
if [ -f /boot/initrd.img-$SRC_RELEASE ]; then
        rm -v /boot/initrd.img-$SRC_RELEASE
fi
sudo /usr/sbin/update-initramfs -c -k $SRC_RELEASE
sudo update-grub
```

C.2 trigger_script.sh

```
#!/bin/sh
# read from config-file in ~/.trigger
#
# Written by:
# Henrik Austad <henrik@austad.us> 2009
# Norwegian Uninversity of Science and Technology
# Department of Engineering Cybernetics
#
# This script is released under GPLv2

# This small script will move to the designated folder, and checkout out
# the provided branch in $HOME/.trigger/trigger.conf
# It will then proceed to build the (kernel). It is, at the moment,
# hardcoded to use the kernel.

# if some other instance is running, we drop. If the already running
# instance does not catch the instnace, we can trigger it manually. No
# need to be fancy-schmanzy
lockfile=$HOME/.trigger/lock
if [ -f $lockfile ]; then
    echo "Lock ($lockfile) already present, closing" >&2
    exit
fi
touch $lockfile

# read branches from file
cfile=$HOME/.trigger/trigger.conf
if [ ! -f $cfile ]; then
    echo "$cfile non-existant, closing"
    exit
fi

time=`date +%s`
tmp_err="tmperr_$time"
tmp_msg="tmpmsg_$time"
tmp_name="tmp_$time"

# Redirect stderr and stderr, save for later usage (so we can echo to
# stderr/out and get the mail formatted properly.
exec 3>&1
exec 4>&2
exec 2>$tmp_err
exec 1>$tmp_msg

function test_build()
{
    b=$1
    p=$2
    pushd $p
    git checkout -f $b
    make distclean
    cp -v ../pfair_config .config
    echo "Building branch $b"
    echo "Building branch $b">&2
    if [ ! -f config ]; then
        echo "No config found, copying from /boot" >&2
        cp /boot/config-`uname -r` config
    fi

    yes '' | make oldconfig > /dev/null
    echo "Building kernel, 32 jobs in parallell, using distcc"
    time make -j40 all CC="distcc ccache gcc" > /dev/null

    # Do the module-magics
    rm -rf /home/henrikau/tmp_modules > /dev/null
    time make INSTALL_MOD_PATH=/home/henrikau/tmp_modules modules_install
    popd
}

# read through config and parse each line separately
old_ifs=$IFS
IFS=$'\n'
tmp_sdate="`date +%F\ %H:%M:%S`"
for line in `cat $cfile`;
  do
  if [ -n "$line" ] && [[ $line != \#* ]]; then
      echo "valid line ($line), testing for branch and path"
      branch=`echo $line | cut -d ' ' -f1`
      path=`echo $line | cut -d ' ' -f2`
      echo "Got branch $branch and path $path"
      if [ -d $path ]; then
          # test to see if branch-name contains build
          echo "supplied path is valid"
          if [ "`echo $branch | sed -n '/^build.*/p'`" ]; then
              test_build $branch $path
          else
              # This should be avoided by post_update, but add line in
              # case we forgot some crazy special case
              echo "$branch is not a build-branch. Skipping">&2
          fi
      else
          echo "$path does not exist" >&2
      fi
  fi

  echo  "Removing '$line' from $cfile: "
  sed -i "s:$line::g" "$cfile"
done
tmp_edate="`date +%F\ %H:%M:%S`"

((secs=`date +%s -d "$tmp_edate"`-`date +%s -d "$tmp_sdate"`))

IFS=$old_ifs

# Construct the body of the email
tmpfile=`mktemp trigger.XXXXXX`
echo "" >> $tmpfile
echo "The trigger_script.sh has just finished running at `hostname`," >> $tmpfile
echo "and this is the output generated from the system" >> $tmpfile
echo "" >> $tmpfile
echo -ne "Start:\t$tmp_sdate\n" >> $tmpfile
echo -ne "End:\t$tmp_edate\n" >> $tmpfile
echo -ne "Total:\t" >> $tmpfile
echo -ne - | awk '{printf "%d:%d:%d","'"$secs"'"/(60*60),"'"$secs"'"%(60*60)/60,"'"$secs"'"%60}' >> $
echo -ne " ($secs seconds)\n" >> $tmpfile
echo "" >> $tmpfile
echo "" >> $tmpfile

cat $tmp_msg >> $tmpfile
echo "" >> $tmpfile
```

```
cat $tmp_err >> $tmpfile
# Send result to the owner of the job. Assume proper delivery (possibly
# via a .forward)
cat $tmpfile | mail -s "Result of running the automated building script at `hostname`" `whoami`

# Cleanup
rm -f $tmp_msg
rm -f $tmp_err
rm -f $lockfile
rm -f $tmpfile

# remove empty lines in the config-file and return stderr and stdout
# just to be safe.
sed -i '/^$/d' $cfile
exec 1>&3
exec 3>&4
```