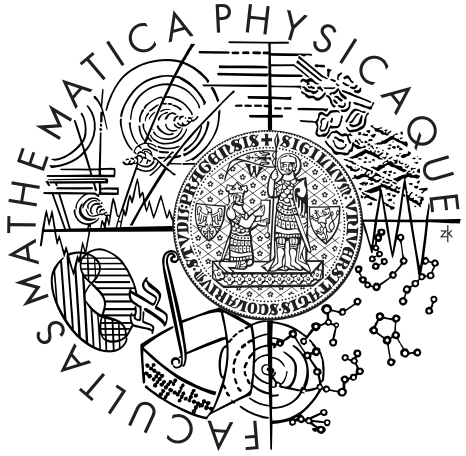
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Rudolf Tomori

Resource limiting and accounting facility for FreeBSD

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Martin Děcký

Study programme: Computer science

Specialization: Software systems

Prague 2013

I would like to thank to my supervisor Martin Děcký. He provided me with many useful tips, suggestions and helpful insight during the course of this thesis.

I would also like to thank to the FreeBSD developer Edward - the author of the racct/rctl framework in the FreeBSD kernel for being very supportive. I also thank to the FreeBSD developer Konstantin for reviewing parts of my patches.

Finally, I wish to thank to Google for supporting students working on open-source projects, my family and Janka for being patient with me while working on this thesis.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Nástroj pro FreeBSD na měření a limitování spotřeby systémových zdrojů

Autor: Rudolf Tomori

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Martin Děcký

Abstrakt: Tato práce analyzuje implementaci Linux cgroups subsystémů odpovědných za limitování procesorového času a propustnosti diskových I/O zařízení. Kromě přístupu použitého v případě Linux cgroups prezentujeme přehled a krátkou analýzu dalších možných přístupů k problému limitování procesorového času a propustnosti diskových I/O zařízení.

Na základě téhle analýzy navrhujeme rozšíření frameworku racct/rctl, který je součástí FreeBSD kernelu a je určen na měření a limitování spotřeby systémových zdrojů. Naše rozšíření umožňuje administrátorům a privilegovaným uživatelům definovat limity na propustnost diskových I/O zařízení a procentuální limity na procesorový čas pro vybraný proces, uživatele anebo FreeBSD jail.

Klíčová slova: FreeBSD, limitování spotřeby zdrojů, spotřeba CPU času, propustnost diskových zařízení

Title: Resource limiting and accounting facility for FreeBSD

Author: Rudolf Tomori

Department: Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Martin Děcký

Abstract: This thesis analyses the implementation of the Linux cgroups subsystems responsible for limiting CPU time and disk I/O throughput. Apart from the Linux cgroups approach, an overview and short analysis of other possible approaches to the problem of limiting CPU time and disk I/O throughput is presented.

Based on the analysis, the thesis proposes an extension to the resource limiting and accounting framework racct/rctl in the FreeBSD kernel. Our prototype implementation of this extension provides features that enable the administrators and privileged users to define disk I/O throughput limits and relative CPU time limits for a particular process, user or FreeBSD jail.

Keywords: FreeBSD, resource limits, relative CPU time, disk I/O throughput

Contents

Introduction

The management of resources is a vital and one of the core functions of operating systems. Let us quote the first sentence of the page about operating systems on Wikipedia:

> *"An operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs."*

It is often the case that the operating system itself knows the best how the resources should be managed to achieve the best overall system performance. Sometimes, however, some specific behaviour is desired to limit the amounts of allocated resources for some processes, users or possibly other operating-system dependent entities.

The resource allocation management decisions are usually results of complex algorithms that have evolved over time. Such algorithms may depend on user defined configurables and thus give the users of the computer a possibility to influence the resource allocation process. For example, considering the *CPU time* a resource managed by the operating system, a user can influence the amount of the *resource* allocated to a process by changing the process's *scheduling priority*.

However, the meaning of the *scheduling priority* varies from operating system to operating system. In fact, it varies from scheduler to scheduler and it can mean different things even in the case of a single operating system - if the operating system provides at least two schedulers to choose from. One such example is *FreeBSD*. Of course, there are some probably generally-accepted guidelines considering the *scheduling priority*. For example, *semantic increase* of the *scheduling priority* should increase the *CPU time* allocated to the process.

The relationship between the *scheduling priority* of a process and its allocated *CPU time* by the scheduler is not straightforward and depends on many other factors, mainly on other processes competing for the same CPU resource. As such, this relationship can not be used to reliably hard-limit the amount of the *CPU time* allocated to a process. One part of this thesis analyses possible approaches to the problem of limiting the *CPU time* allocated to a process by the scheduler. Apart from that, we also inspect the problem of enforcing the *block IO bandwidth* limits in the operating systems. We propose our own solutions for the *FreeBSD* operating system to these two problems.

To sum-up, these are the two core problems dealt with in this thesis:

1. The enforcement of the *CPU time* limits.
2. The enforcement of the *block IO bandwidth* limits.

In this thesis we firstly thoroughly examine the implementation of one possible approach to the problem of limiting resources in *Linux* - the *Linux cgroups* approach. We start with the analysis of the *Linux cgroups* from the user's point of view. Later we delve into the implementation in the kernel space and analyse the *cgroups* in general and how the *CPU limits* and the *block IO bandwidth* limits are enforced in the kernel using the *cgroups*.

After the in-depth analysis of one specific resource-limits implementation - that is of the *Linux cgroups* - we speculate later in the *Analysis* chapter about other possible approaches and their possible advantages and disadvantages. Based on this analysis, we choose the preferable implementation models that we think suit the best the needs of the two core problems presented in this thesis.

Finally, we present a prototype implementation of the *CPU limits* and the *block IO bandwidth limits* for the *FreeBSD* operating system. We also provide on the attached DVD a QEMU[1] image of a *FreeBSD* virtual machine where our prototype implementation can be easily tested.

[1]QEMU is a generic and open source machine emulator and virtualizer. For more information, see the http://www.qemu.org web page.

1. Linux Control Groups

1.1 Introduction

Linux Control Groups is a framework in the *Linux kernel* that evolved from a patch set named *process containers* that was developed by *Google* engineer *Rohit Seth* in September 2006. The development has then moved on to another *Google* engineer *Paul Menage*. The *process containers* were later renamed to *cgroups* and merged with *2.6.24* kernel.

Control Groups or *cgroups* provide a way for the computer administrator to partition all running processes into hierarchically structured groups - *cgroups*. The administrator groups together processes that share the same resource utilization requirements into a single *cgroup*. He can then specify per-cgroup resource limits.

1.2 Cgroups hierarchies

Cgroups are organized in hierarchical structures, where child *cgroups* inherit certain parameters from their parents. When a new process forks, it is inserted into its parent process's *cgroup*. The computer administrator creates *cgroup* hierarchies and defines how the processes are partitioned into individual *cgroups* that form a single hierarchy. Every *cgroup* in hierarchy specifies resource management information that are applicable to the processes in this *cgroup*.

Because the computer administrator can have different utilization requirements for different system resources, he can specify a separate *cgroup hierarchy* for every available *cgroup resource subsystem*. Different *resource subsystems* may share a single *cgroup hierarchy* if they require only a single partitioning scheme of all processes.

1.3 Cgroup resource subsystems

Cgroups are merely a partitioning scheme for all running processes. They come with some resource utilization meta-information, but they do not influence the kernel system resource management on their own. On the contrary, the different kernel *resource subsystems* access *cgroup* meta-information for the client process's *cgroup* container and provide access to the required system resource according to that meta-information.

This implies that the individual kernel *resource subsystems* must be *cgroup-aware*. Here follows a list of some of the cgroup-aware resource subsystems with their brief descriptions:

cpu Allows us to specify the precedence weight for processes inside the *cgroup* in case some of them compete with processes from other *cgroups* for CPU time.

cpuacct Allows us to track the CPU usage information.

blkio Allows us to limit the input/output access for block devices.

cpuset Allows us to restrict all processes inside a *cgroup* to certain CPUs.

memory Allows us to specify the memory limits for processes inside a *cgroup*.

devices Allows us to limit the access to devices for processes inside a *cgroup*.

freezer Allows us to suspend/resume processes inside a *cgroup*.

net_cls Allows us to assign a network class id to a *cgroup* that is propagated with packets sent/received by processes inside the *cgroup*.

1.4 Cgroup virtual file system

Every *cgroup hierarchy* is associated and directly mirrored in an instance of the *cgroup virtual filesystem*. These instances of the *cgroup virtual filesystem* serve as a handle to users with sufficient privileges to manipulate the *cgroup hierarchies* and assign the processes into the individual *cgroups*.

Every *Cgroup* is represented by a single directory in the *cgroup virtual filesystem* hierarchy. And its position in the directory tree of the filesystem hierarchy is determined by the *cgroup* position in the *cgroups hierarchy*. Various *cgroup* parameters are stored as files in the directory representing the *cgroup*. Every process that is a member of a *cgroup* has its process identifier listed in the *tasks* file of the *cgroup* directory. Assigning a process to some *cgroup* is as simple as adding the process identifier to the appropriate *tasks* file.

Because every *cgroup hierarchy* is associated with (possible more than one) *cgroup resource subsystems*, we also have a mapping between the instance of the *cgroup virtual filesystem* and *cgroup resource subsystems*. For different *resource subsystems*, there are different kinds of meta-information needed to be stored within the *cgroup*. This has the effect that directories representing *cgroups* in different hierarchies contain different types of meta-information files.

1.5 Working with cgroups

Here we show a small example where we set a memory limit for our shell using *cgroups*. We have already mounted different *cgroup virtual file system* hierarchies for different resource subsystems:

```
# rudo@rudo-laptop:/sys/fs/cgroup$ ls
# cpu cpuacct devices freezer memory
```

Now we create a new *cgroup* inside the memory filesystem:

```
# root@rudo-laptop:/sys/fs/cgroup# cd memory/
# root@rudo-laptop:/sys/fs/cgroup/memory# mkdir restricted
```

Now we set the memory limit to 100kB and add our shell instance to the restricted *cgroup*:

```
# cd restricted/
# echo 100k > memory.limit_in_bytes
# echo $$ > tasks
```

Now, on our test-bed machine, we can observe that the shell stops working properly because of the memory limit being set too low.

1.6 Using cgroups to set relative cpu time limits

The access of processes inside a *cgroup* to the *CPU resource* can be managed by the *cpu subsystem*. For our purposes of relative cpu time limits, we take a look at one *cgroup* configurable named *cpu.shares*. Here is the meaning of this configurable as provided in the *Red Hat Linux Resource Management Guide[4]*.

cpu.shares *Contains an integer value that specifies a relative share of CPU time available to the tasks in a cgroup. For example, tasks in two cgroups that have cpu.shares set to 1 will receive equal CPU time, but tasks in a cgroup that has cpu.shares set to 2 receive twice the CPU time of tasks in a cgroup where cpu.shares is set to 1.*

However, as we will see, there are some important details missing in this explanation. Firstly, it is not always the case that the access of processes to the CPU is proportional to the *cpu.shares* setting. Consider the following example proposed by *Greg Smith* in a public internet forum. We perform the following steps on our dual-core test-bed machine:

```
# root@rudo-laptop:~# cd /sys/fs/cgroup/cpu
# root@rudo-laptop:/sys/fs/cgroup/cpu# mkdir low high
# root@rudo-laptop:/sys/fs/cgroup/cpu# echo 512 > low/cpu.shares
# root@rudo-laptop:/sys/fs/cgroup/cpu# echo 2048 > high/cpu.shares
```

We have just created two *cgroups* in the *cpu resource* hierarchy. The one named *low* will contain processes that should only receive half of the cpu time of the other processes (The default value for the *cpu.shares* parameter is 1024). And the other *cgroup* named *high* would contain processes that should get twice the CPU time of normal processes and four times the cpu time of processes in the *low cgroup*.

Now we will create two example processes and add each of them to the different *cgroup*:

```
# root@rudo-laptop:/sys/fs/cgroup/cpu# yes low > /dev/null &
# root@rudo-laptop:/sys/fs/cgroup/cpu# echo $!  > low/tasks
# root@rudo-laptop:/sys/fs/cgroup/cpu# yes high > /dev/null &
# root@rudo-laptop:/sys/fs/cgroup/cpu# echo $!  > high/tasks
```

Now if we take a look at the percentages of the cpu utilization of our two processes, on our test-bed machine we see that the process in the *low cgroup* has utilization 98.3% and the process in the *high cgroup* shows 99.0%. That is not what we would normally expect after reading the previous explanation, given the *cpu.shares* parameters being set to 512 and 2048, respectively.

We can go now one step further and start another process in the *high cgroup*:

```
# root@rudo-laptop:/sys/fs/cgroup/cpu# yes high2 > /dev/null &
# root@rudo-laptop:/sys/fs/cgroup/cpu# echo $!  > high/tasks
```

Now, on our test-bed machine, if we inspect the cpu utilization percentages of our processes, we can see that the process spawned by the command `yes high2` has lower cpu usage than the process in the *low* cpu *cgroup*.

So now we see that we can not rely only on the *cpu cgroup subsystem* and expect that processes in *cgroups* with high *cpu.shares* value will get more cpu time than processes in *cgroups* with low-valued *cpu.shares* parameter.

To find out the explanation for this strange behavior, we must realize one important detail that was missing in the previous explanation. And that is, the value of the *cpu.shares cgroup* parameter is only taken into account by the scheduler if there is active cpu contention for a single cpu.

What happened in our scenario, was this: The two processes `yes low` and `yes high` got placed on separate cpu cores. That is why their cpu usage was almost identical - they did not compete with each other for the cpu time. And later, when we started another process in the *high* cpu *cgroup*, it got placed to the cpu other than the `yes low` process. That is why it was possible for the process `yes high2` to have lower cpu utilization percentage than the process `yes low`.

If we restricted the processes in the previous example to run only on a single cpu core, we would get the expected behavior.

1.7 Using cgroups to limit disk I/O throughput

Now we are going to show how computer administrators can use *cgroups* to limit the block device I/O throughput. On our test-bed machine, we will limit the read access to our hard drive. Limiting write access is similar with the only difference being that you should use the *blkio.throttle.write_bps_device cgroup* parameter instead of *blkio.throttle.read_bps_device*.

So we have the *blkio* hierarchy that contains only one root *cgroup*:

```
# root@rudo-laptop:/sys/fs/cgroup# ls
# blkio cpu cpuacct devices freezer memory
```

Now we add the limit of 1048576 bytes per second (1MB/s) for the read access of our hard drive that is identified on our test-bed machine by major and minor device numbers 8 : 0.

```
# echo "8:0 1048576" > blkio/blkio.throttle.read_bps_device
```

We have just changed the configuration of the root *cgroup* for the *blkio subsystem*. That means this setting will affect all processes on our machine, because they all are members of this single root *cgroup*. (That is because this *cgroup* does not contain any children *cgroups* and the *cgroups* in a single hierarchy must form a partition of all processes.)

Now it is time to test our new configuration. The *ifile* is just some file on our hard drive that we read:

```
# root@rudo-laptop:/sys/fs/cgroup# dd if=/ifile of=/dev/null bs=4k count=1024
# 1024+0 records in
# 1024+0 records out
# 4194304 bytes (4.2 MB) copied, 4.10918 s, 1.0 MB/s
```

This result shows that our hard drive throughput in this operation was $1.0MB/s$ and that is exactly the limit we have set.

2. The Linux kernel cgroups management

2.1 Introduction

We are going to inspect the implementation of the *Linux Control Groups* in this chapter. We will take a look at the kernel management of the *control groups* and see the relevant kernel data structures.

2.2 Interconnecting Linux tasks with cgroups

In *Linux*, the terms *task* and *process* are used interchangeably. Every *Linux process* is represented in the kernel by the `struct task_struct`. As we can see in its definition, there are two cgroup-relevant fields in this struct:

```
struct task_struct {
...
#ifdef CONFIG_CGROUPS
        struct css_set __rcu *cgroups;
        struct list_head cg_list;
#endif
...
};
```

Every *task* in the system has a reference-counted pointer to the `struct css_set` structure. This structure contains an array of reference-counted pointers to the `struct cgroup_subsys_state` objects, one pointer for each registered *cgroup subsystem* in the kernel.

```
struct css_set {
...
        struct cgroup_subsys_state
            *subsys[CGROUP_SUBSYS_COUNT];
...
};
```

And finally the `struct cgroup_subsys_state` objects, that store the per-cgroup state for the respective *cgroup subsystem*, hold a pointer to the actual `struct cgroup` structure and some additional subsystem state information.

```
struct cgroup_subsys_state {
        struct cgroup *cgroup;
        atomic_t refcnt;
        unsigned long flags;
        struct css_id __rcu *id;
...
};
```

The `struct cgroup` objects form a tree hierarchy, where every object has a linked-list of its children, a linked-list of its siblings and a pointer to its single parent:

```
struct cgroup {
...
        struct list_head sibling;
        struct list_head children;
        struct cgroup *parent;
...
        struct cgroup_subsys_state
            *subsys[CGROUP_SUBSYS_COUNT];
...
};
```

The `struct cgroup` object also stores an array of private pointers to the `struct cgroup_subsys_state` objects. This is because a single *cgroup hierarchy* may be shared by several subsystems. This situation is depicted in the figure 2.1:

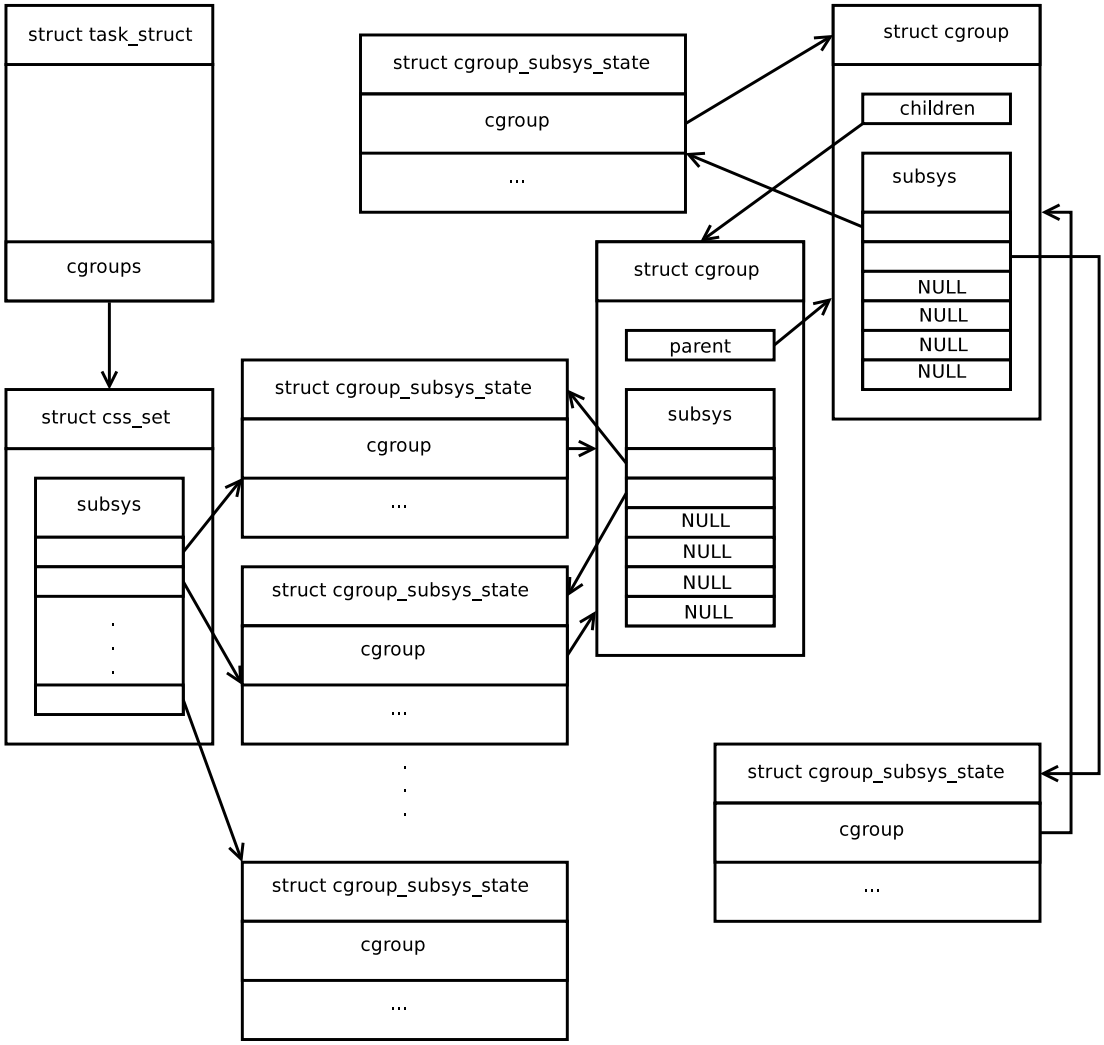


Figure 2.1: A single *cgroup hierarchy* shared by two *cgroup subsystems*.

The situation in the figure 2.1 shows a single *cgroup hierarchy* consisting of the parent *cgroup* and one child. This *cgroup hierarchy* is shared by two

cgroup subsystems. Every *cgroup* in this hierarchy has an associated `struct cgroup_subsys_state` object for every subsystem that uses this hierarchy. These subsystem state objects are also referenced from the `struct css_set` objects.

There is no direct link from the `struct task_struct` objects representing processes in the kernel to the individual *cgroups* the process is member of. For this, the pointers from the process's `struct css_set` need to be followed.

It might often be the case that several tasks in the system are members of the same *cgroups* for all registered *cgroup subsystems*. In this case, they all share a single `struct css_set` object.

2.3 Iterating over tasks that are members of a specific cgroup

As we already know, there is no direct link in the kernel between the task and the *cgroups* the task is member of. To look up the tasks that are members of a specific *cgroup*, we need to iterate over the `struct css_set` objects that are indirectly connected to the *cgroup* and then in the second phase we iterate over the tasks that share the same `struct css_set` object.

2.3.1 Associating the css_set with tasks that reference it

We want to iterate over all the processes that share the same `struct css_set` object. Now, we will understand the meaning of the second cgroup-relevant field in the `struct task_struct`, namely the `cg_list` field. This field serves as a handle by which the `struct task_struct` objects referencing the same `struct css_set` object are inserted into a double linked-list. This linked-list is anchored in the `tasks` field of the `struct css_set` structure.

Firstly, let us remind the important part of the `struct task_struct` structure representing a process in the kernel:

```
struct task_struct {
...
#ifdef CONFIG_CGROUPS
        struct css_set __rcu *cgroups;
        struct list_head cg_list;
#endif
...
};
```

Now we can see the `tasks` field in the `struct css_set` structure:

```
struct css_set {
...
        struct list_head tasks;
...
        struct cgroup_subsys_state
            *subsys[CGROUP_SUBSYS_COUNT];
...
};
```

And the `struct list_head` structure definition is quite simple:

```
struct list_head {
        struct list_head *next, *prev;
};
```

In the figure 2.2 we illustrate the `tasks` list that links the *tasks* pointing to the same `css_set`:

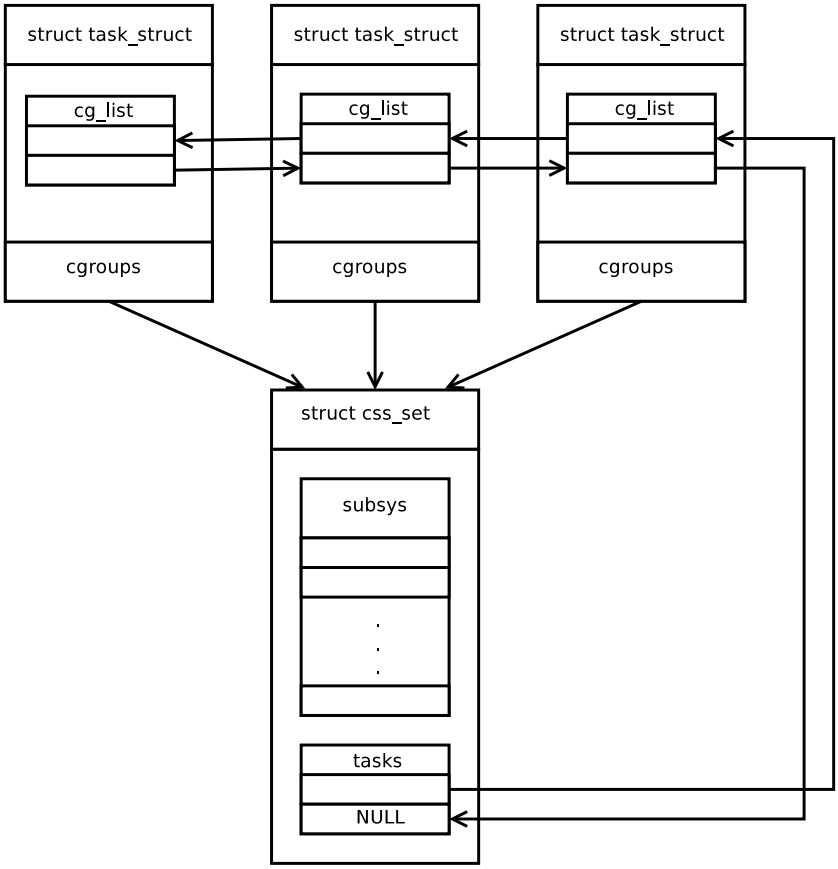


Figure 2.2: The `task_struct` objects pointing to the same `css_set`

2.3.2 Iterating the cgroups that are indirectly linked to a css_set

It is often necessary to iterate over all the *cgroups* that are linked with a single `struct css_set` object. To effectively solve this problem, the Linux kernel uses a special data structure for associating *cgroups* with *css_sets* and also for associating *css_sets* with *cgroups*.

```
struct cg_cgroup_link {
        struct list_head cgrp_link_list;
        struct cgroup *cgrp;
        struct list_head cg_link_list;
        struct css_set *cg;
};
```

One `struct cg_group_link` object associates a single *css_set* object pointed to by the `struct css_set *cg` field with one *cgroup*. This *cgroup* object is referenced by the `struct cgroup *cgrp` field. However, we need to somehow associate a single *css_set* object with all the *cgroups* it indirectly links. To achieve this, we just need a linked list of `struct cg_cgroup_link` objects.

Thus, all the `struct cg_cgroup_link` objects in this list point to the same `struct css_set` and each of them references a different *cgroup*. The `struct cg_cgroup_link` objects are connected to a linked list via the `struct list_head cg_link_list` field. And this list is anchored at the `struct list_head cg_links` member of the `struct css_set` structure. This situation is show in the figure 2.3:

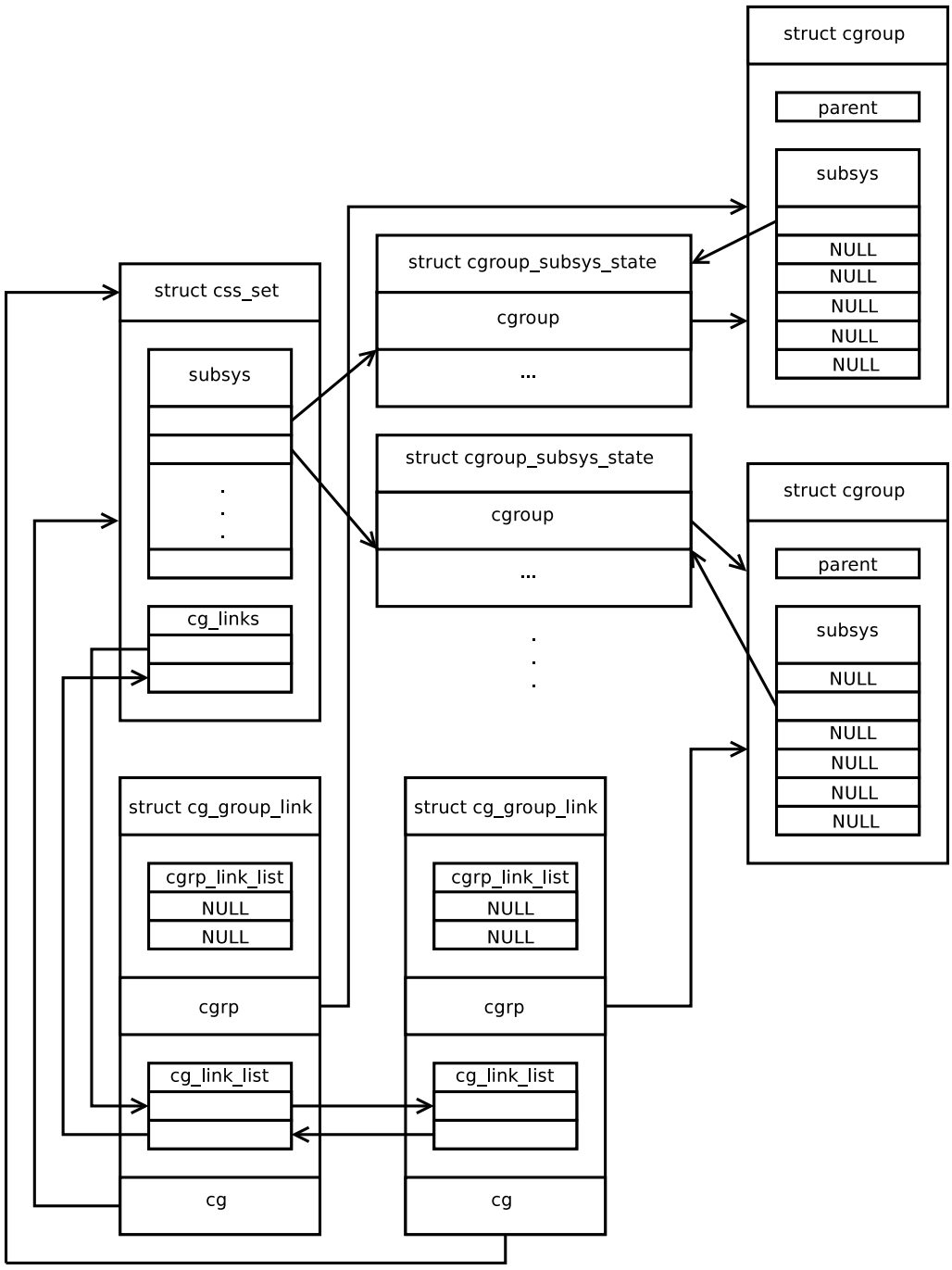


Figure 2.3: *Cgroups* indirectly linked to a *css_set*

2.3.3 Iterating the css_sets that indirectly link a cgroup

If we want to find the *tasks* that are members of a specific *cgroup*, we firstly need to find all the `struct css_set` objects that indirectly link the *cgroup*. If we know how to do that, we can then iterate over all the *tasks* associated with the *css_sets* and thus find all the *tasks* that are members of the *cgroup* in question.

The problem of iterating over the *css_sets* that are linked to a specific *cgroup* is the opposite version of the problem dealt with in the previous paragraphs. It is also solved in a similar fashion, using the `struct cg_group_link` objects. But this time, the *cg_group_link* objects are connected to a double linked-list using the `struct list_head cgrp_link_list` field. This list is anchored at the `struct list_head css_sets` field of the `struct cgroup`. This situation is shown in the figure 2.4:

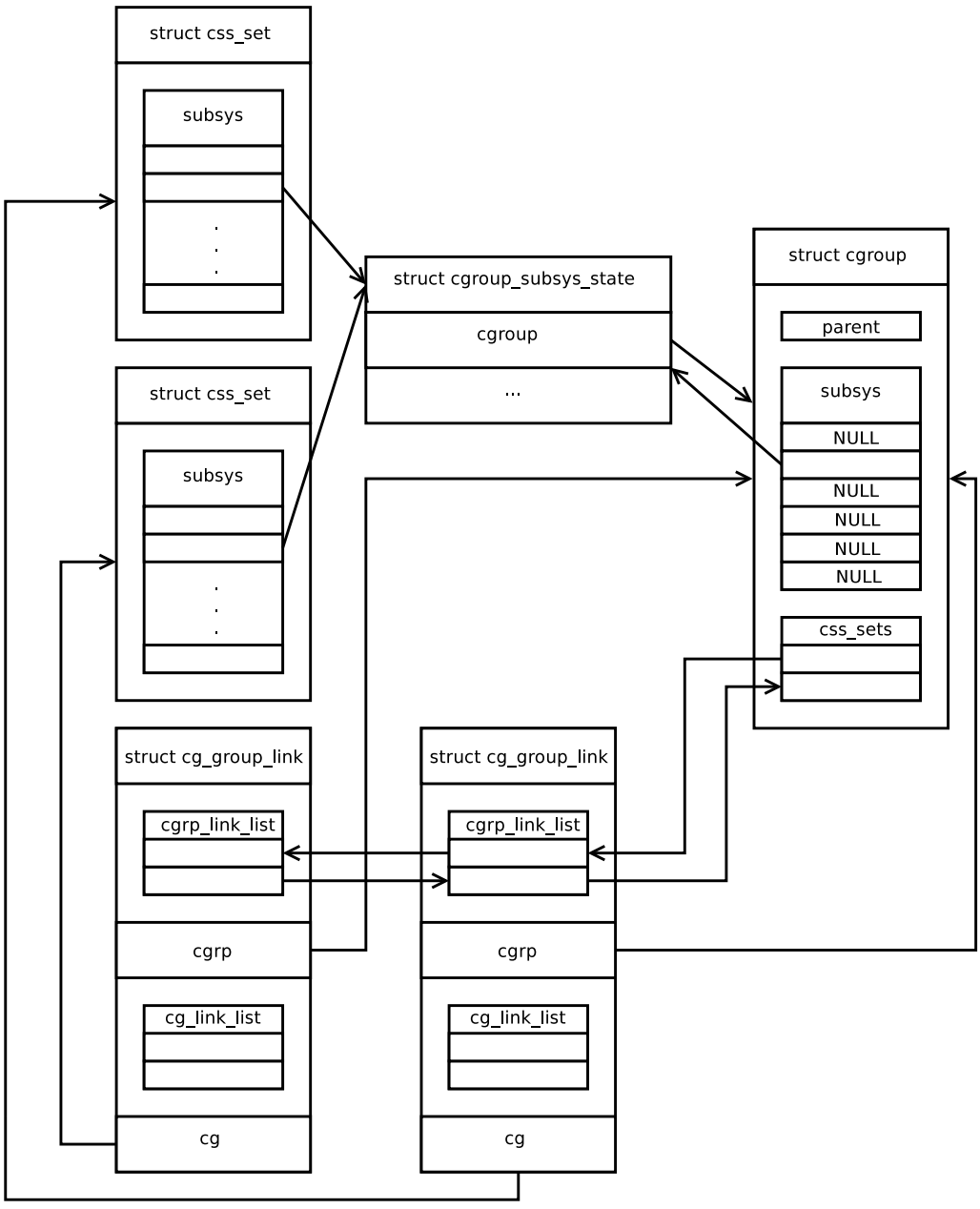


Figure 2.4: The *css_sets* indirectly linked to a *cgroup*

2.4 Creating a new cgroup

Creating a new *cgroup* under a specified hierarchy is quite a straightforward process. Firstly, the necessary `struct cgroup` object is allocated. Then, all the *cgroup subsystems* that use the *cgroup hierarchy* are iterated, and for each of them a new `struct cgroup_subsys_state` object is created and attached to the new *cgroup*. This newly created *cgroup* is also properly inserted into the *cgroup* tree data structure. After that the new directory representing the *cgroup* in the attached *cgroup filesystem* is created and properly populated.

The new, just created *cgroup* contains no tasks at all. The exception to this rule is the case when the new *cgroup* is the root *cgroup* of the hierarchy. In this case, the *cgroup* is populated with all the running *tasks* in the system.

2.5 Moving the task to a different cgroup

When a task is moved to a different *cgroup*, the task is simply attached to a different *css_set* object via its `struct css_set * cgroups` member pointer. (With the *task_struct* being inserted into the *tasks* list of the new *css_set* and removed from the old one.)

The only problem is, how to find the appropriate *css_set* - if it exists. If it doesn't, it is created. The *css_set* is found using a hash table. All *css_sets* created in the kernel have pointers to them stored in an internal hash table. The hash function is numerically computed from the pointers in the `subsys` array of the *css_set*. To find the desired *css_set*, firstly a *template array* is constructed, that contains pointers to the desired *cgroup_subsys_state* objects. This *template array* is almost the exact copy of the `subsys` array of the old task's *css_set*. For appropriate *cgroup subsystems*, however, the pointers are substituted and set to point to the *cgroup_subsys_state* objects of the new *cgroup*.

Once the *template array* is constructed, the hash value of the array is computed and the *css_set* hash table is queried to find the linked list of the *css_set* objects stored under the computed hash value. This list is then linearly searched to find the exact match.

In the following excerpt, we show how the hash function is computed and how the hash value is used to index the hash table:

```
static struct hlist_head*
css_set_hash(struct cgroup_subsys_state *css[])
{
        int i, index;
        unsigned long tmp = 0UL;

        for (i = 0; i < CGROUP_SUBSYS_COUNT; i++)
                tmp += (unsigned long)css[i];
        tmp = (tmp >> 16) ^ tmp;
        index = hash_long(tmp, CSS_SET_HASH_BITS);

        return &css_set_table[index];
}
```

3. The Linux CPU cgroup subsystem

3.1 Introduction

In this chapter we are going to explore the *Linux scheduler*. After a brief introduction to the scheduler, we will inspect how it interconnects with the *cgroups* and how the *cgroups* affect the scheduler's behaviour when it comes to enforcing the relative cpu limits.

3.2 The Linux CFS scheduler

The current Linux scheduler - *The Completely Fair Scheduler* was developed by a Hungarian Linux hacker *Ingo Molnár*. The general approach of the scheduler was inspired by Con Kolivas's work on CPU scheduling and the scheduler was incorporated into the kernel release *Linux-2.6.23* in 2007.

3.2.1 Overview of the CFS scheduler

The scheduler tracks the CPU usage of a task in what it calls the task's *virtual runtime*. This is not meant to be the exact physical time the task spent on the CPU. This *runtime* of a task is called *virtual* because it is normalised to take into account the *nice* value of the task and the *nice* values of all the other running tasks in the system.

To achieve the scheduling fairness among the tasks, the scheduler always chooses the task with the lowest *virtual runtime*. Thus, the task with the highest need for the CPU is chosen. The tasks on a single CPU are organised in a self-balancing red-black tree, with the *virtual runtime* as the key. This way, the $O(log(n))$ bounds are imposed on the tree insertion and removal operations, with n being the total number of running tasks on the CPU. This red-black tree forms a sort of a timeline of future task execution.

The scheduler is quite radical in its design, when compared with traditional Linux or BSD schedulers. There are no traditional arrays of run-queues, there is no attempt to identify interactive processes and also the notion of a fixed time slice window is gone. This is because the length of the time-slice for a task in *CFS* is task-dependent and depends on the task *nice* value and its relation to the *nice* values of the other running tasks.

3.2.2 Overview of the CFS data structures

The *CFS scheduler* organizes the runnable tasks in a per-CPU data structure `struct rq` called *runqueue*, so that the *CPU affinity* is easily achieved. The tasks are not referenced directly from this structure, instead, the `struct rq` contains two pointers to *runqueues* of two scheduling classes in the kernel - the *real-time* and the *fair scheduling class*.

The Linux scheduler can be viewed as a modular framework, with multiple *scheduling classes*, that are sequentially ordered according to the priority of the class. When the scheduler looks for the next task to run in the `pick_next_task()` function, it sequentially iterates over the scheduling classes in the order of decreasing priority until it finds some class that can provide a runnable task. The least priority scheduling class is the *idle class*. In this thesis, we are interested only in the *fair scheduling class* of the Linux scheduler. The tasks managed by this scheduling class are stored in the `struct cfs_rq` data structure. This runqueue is directly referenced from the main per-cpu runqueue `struct rq`.

```
struct rq {
...
        struct cfs_rq cfs;
        struct rt_rq rt;
...
};
```

And the `struct cfs_rq` contains the `struct rb_root tasks_timeline` field that is the root node of the red-black tree of runnable tasks that we were discussing in the previous section. The tasks are inserted into this tree by their `struct sched_entity` member field.

```
struct cfs_rq {
...
        struct rb_root tasks_timeline;
        struct rb_node *rb_leftmost;
...
};

struct task_struct {
...
         const struct sched_class *sched_class;
        `struct sched_entity se;
...
};
```

The *sched_entity* objects contain the scheduling information relevant to the task. For example, the `u64 vruntime` field stores the task's *virtual runtime*. These schedulable entities can be directly inserted into the red-black tree, because they have the `struct rb_node run_node` field:

```
struct sched_entity {
...
        struct rb_node run_node;
...
        u64 vruntime;
...
};
```

In the figure 3.1, we display the overall picture we have so far, that shows a few runnable tasks in the red-black tree of the *CFS scheduler*:

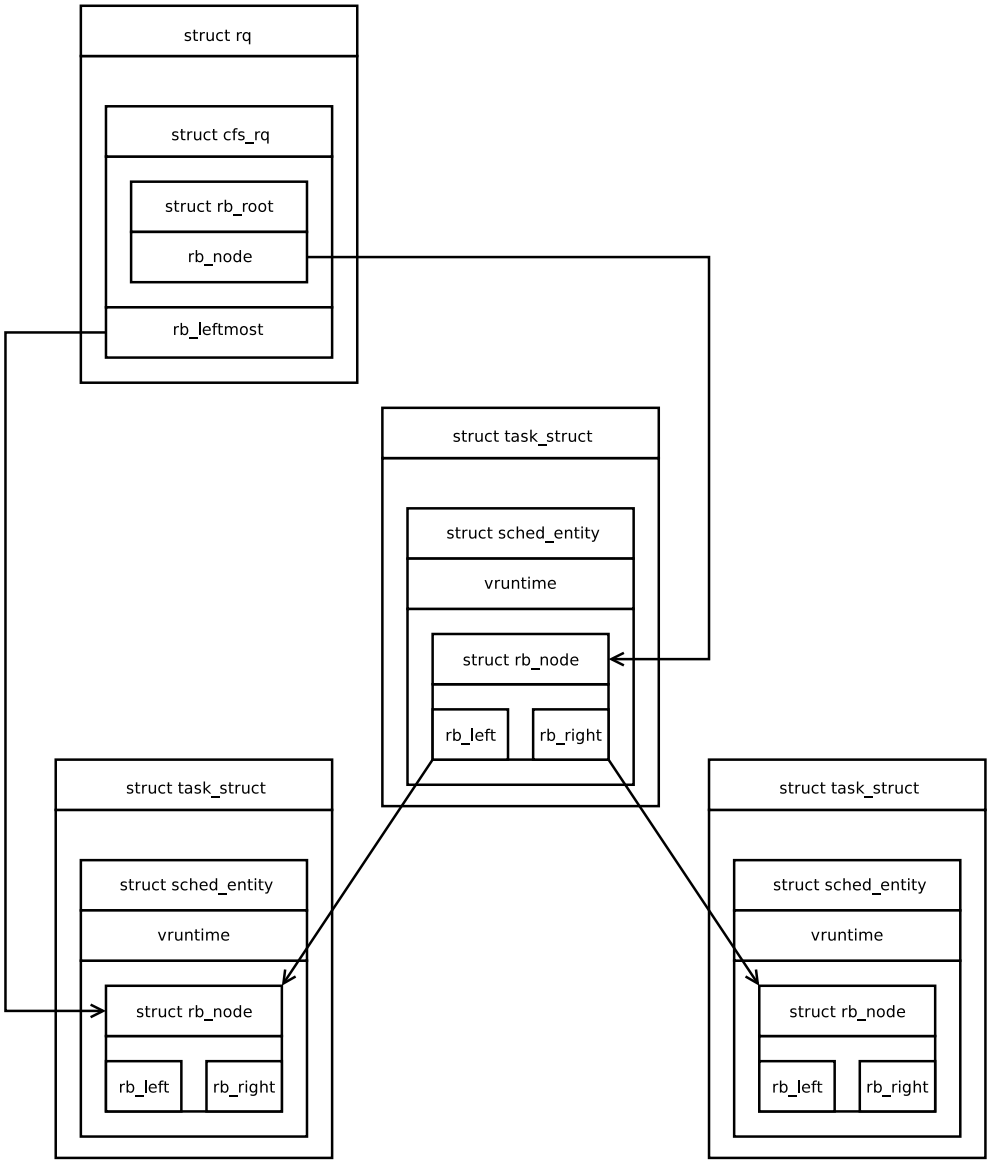


Figure 3.1: A few *runnable tasks* in the red-black tree of the *CFS scheduler*.

3.3 The cgroup extension to the CFS scheduler

The *CFS scheduler* fully supports *cgroups*. That means, the scheduler is aware of the *cgroup* partitioning of runnable tasks into hierarchical groups. And the scheduler treats the tasks in the same *cgroup* according to the tunables configured via the *cpu cgroup filesystem* and according to the *cgroup's* position in the hierarchy.

We already know that the *cgroups* are represented in kernel by `struct cgroup`. That is, however, only a general representation of the *cgroup*. The various configurable *cgroup* parameters specific to the *cpu subsystem* (or other subsystems as well) are stored elsewhere.

As we have already seen in the previous chapter, the `struct cgroup` contains the *subsys* array that stores pointers to the `struct cgroup_subsys_state` objects for all registered *cgroup subsystems*. For the *cpu cgroup subsystem*, this *cgroup_subsys_state* object is extended by the `struct task_group` object. And

that is where the various *cgroup* configurables for the *cpu subsystem* are stored. For example, it contains the `unsigned long shares` field to store the *cpu.shares* tunable of the *cpu cgroup subsystem*. The structure also contains the `struct task_group *parent` field to reflect the *cgroup* hierarchy. The situation is shown in the figure 3.2:

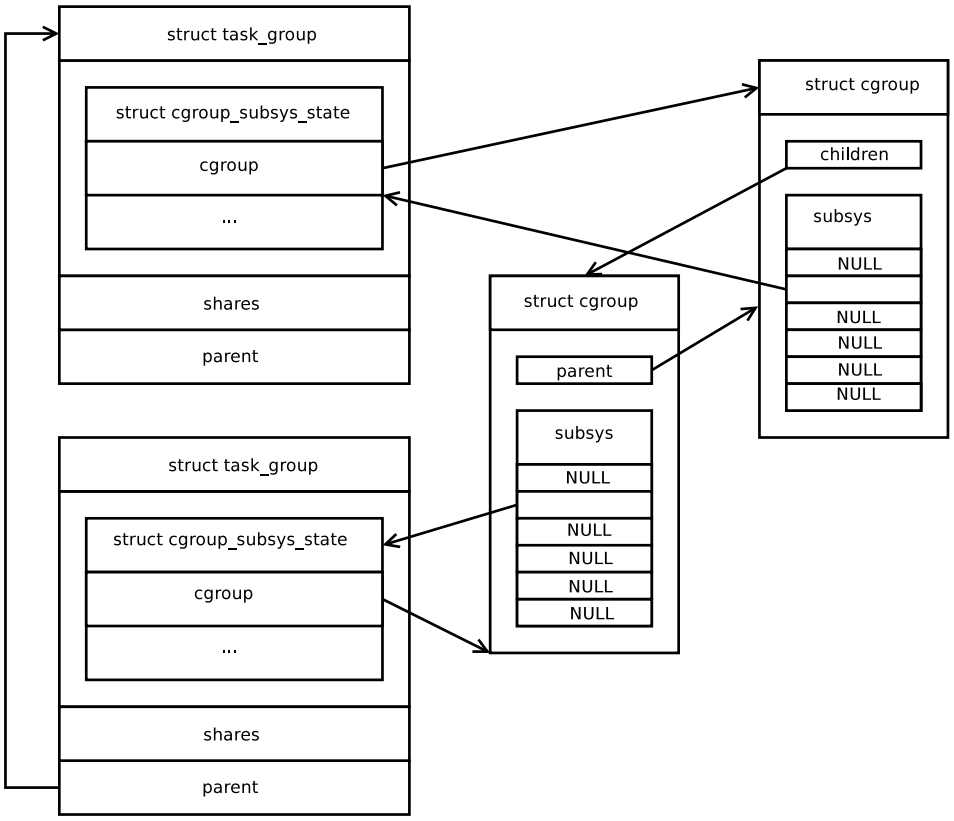


Figure 3.2: The relationship between the *task_groups* and the *cgroups*.

Here follows a short excerpt from the `struct task_group`'s structure definition:

```
struct task_group {
        struct cgroup_subsys_state css;

#ifdef CONFIG_FAIR_GROUP_SCHED
        struct sched_entity **se;
        struct cfs_rq **cfs_rq;
        unsigned long shares;
...
#endif
...
        struct task_group *parent;
...
};
```

Please, note the *se* pointer, that points to the array of pointers to the objects of type *sched_entity*. And there is also the *cfs_rq* pointer, that points to the array of pointers to the *cfs_rq* runqueues. Both of this arrays contain a single pointer for every available CPU in the system. Now, we will finally understand how the *cgroups* are treated by the scheduler.

For every available CPU, the *task_group* has a pointer to a *sched_entity* object. If the *task_group* is associated with a root *cgroup*, this *sched_entity* object is directly inserted into the red-black tree of the *cfs* runqueue of the main per-processor runqueue `struct rq`. Thus, from the scheduler's point of view, the *cgroups* are simply *sched_entities* that are treated in the similar way as normal tasks. These *sched_entities* just happen to represent a group of runnable tasks, instead of representing only a single task.

Also, for every available CPU, the *task_group* has a pointer to a separate *cfs_rq* runqueue. This runqueue stores the tasks of the *cgroup* that are running on this CPU (and also possible *sub-cgroups*) in a red-black tree. These red-black trees are organized exactly in the same way, as the main scheduler per-CPU red-black tree. That is, the sched_entity's *vruntime* field is used as the key.

We already know that the *sched_entity* objects can represent either a single task, or a group of tasks that are members of the same *cgroup*. In case the *sched_entity* represents a group of tasks, the `struct cfs_rq *my_q` member pointer of the structure points to the *cfs_rq* runqueue where the individual member tasks are stored. On the other hand, the `struct cfs_rq *cfs_rq` member pointer of the structure points to the *cfs_rq* runqueue (and thus the red-black tree) on which this entity is to be scheduled. The *sched_entity* objects also form hierarchies via the `struct sched_entity *parent` member field. In the following excerpt, we show again the definition of the `struct sched_entity` structure, this time also with the new member fields we have just learned about:

```
struct sched_entity {
...
        struct rb_node run_node;
...
        u64 vruntime;
...
#ifdef CONFIG_FAIR_GROUP_SCHED
        struct sched_entity     *parent;
        struct cfs_rq           *cfs_rq;
        struct cfs_rq           *my_q;
#endif
};
```

In the following figure 3.3, we see a task that is a member of a child *cgroup*. We show how the task is enqueued into the runqueue of the cgroup's *task_group*. Note that the task is not directly inserted into the scheduler's runqueue, but instead the hierarchical structures are used that mirror the structure of the *cgroup hierarchy*.

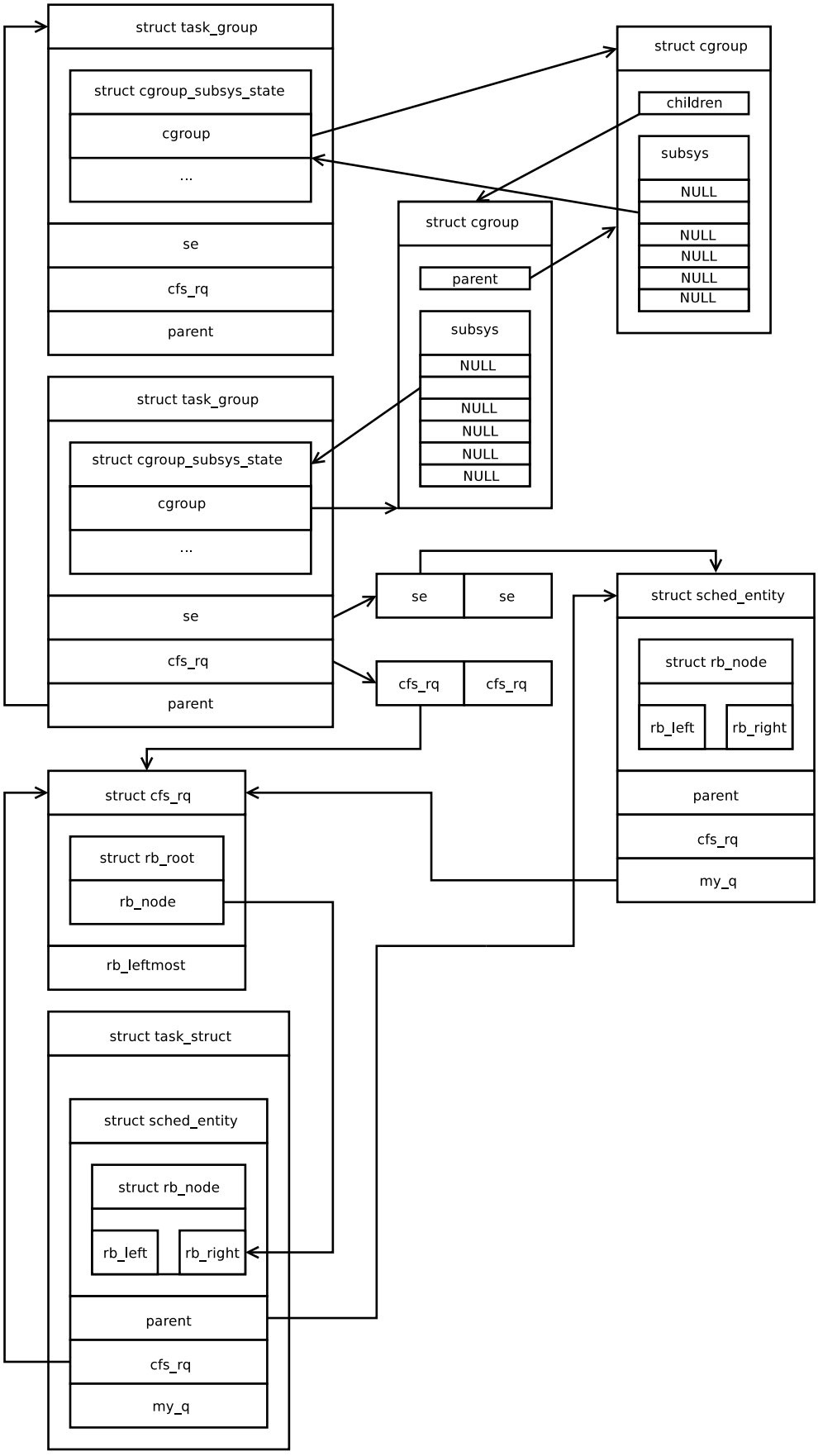


Figure 3.3: A *task* and its relationship to the *scheduler's* data structures.

In the next figure 3.4 we show the missing part of the previous situation. We can see how the main per-processor *cfs_rq* runqueue connects with the *sched_entity* of the root *cgroup's* taskgroup:

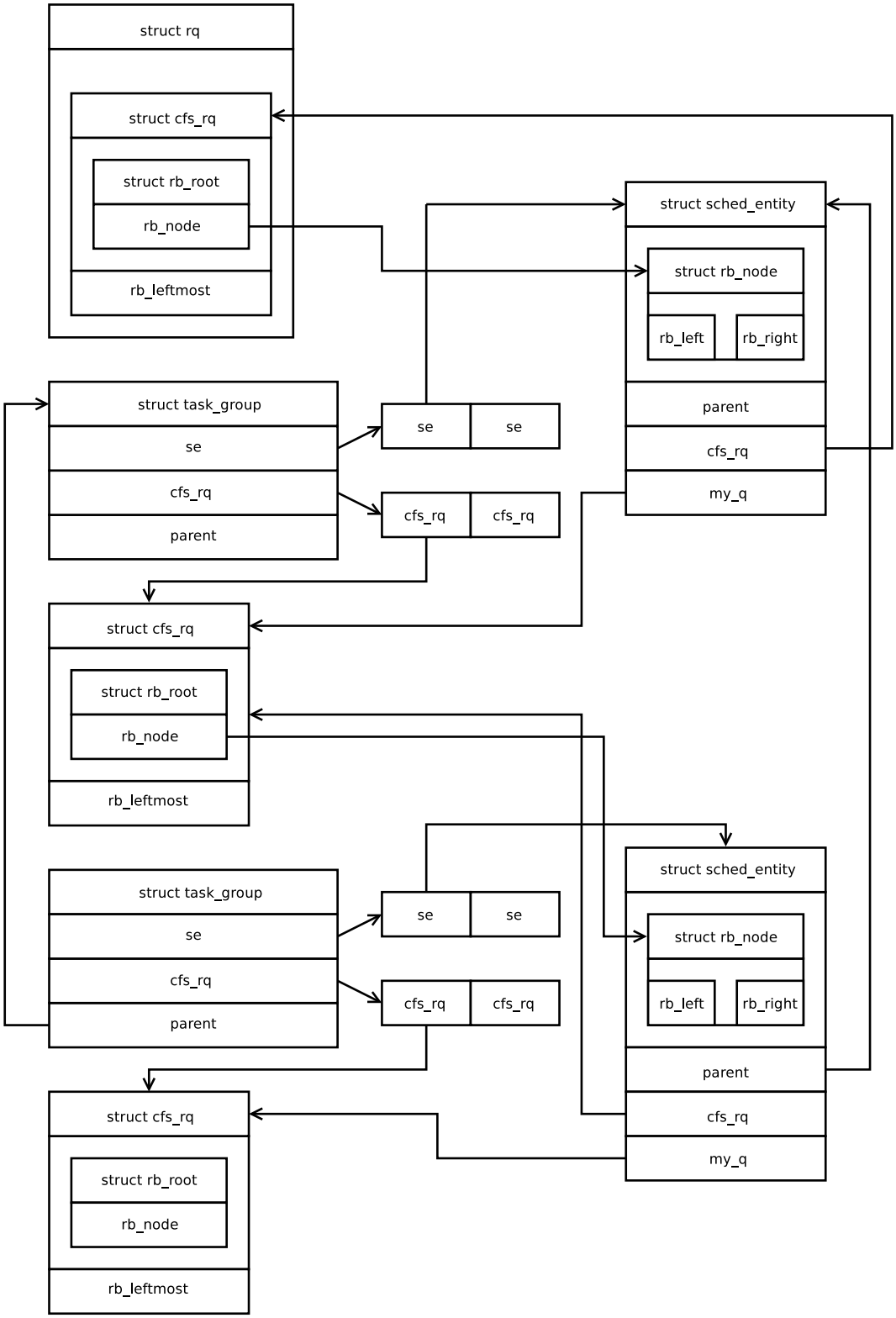


Figure 3.4: A *task* and its relationship to the *scheduler's* data structures: The missing part of the hierarchy

3.4 Enforcing the relative cpu limits using the *cpu.shares* tunable

As we have seen in the first chapter, a privileged user can specify per-cgroup relative cpu limits using the *cpu.shares* tunable of the *cpu cgroup* subsystem. We also know that in the *CFS* scheduler, the scheduling time-slice window is not a fixed value, but it is task-dependent. And this is actually the key, how the relative cpu limits specified by *cpu.shares* value are imposed. The tasks that are members of a cgroup with relatively high *cpu.shares* value will get longer time slices and thus will get more cpu time when compared to the other tasks.

3.4.1 The *load weight* of tasks and cgroups

Now we need to understand the *load weight* of a *sched_entity* object. If the *sched_entity* represents a single task, then the *load weight* of the task is determined by the task's nice value. It is defined in the kernel by the following table:

```
static const int prio_to_weight[40] = {
 /* -20 */  88761,  71755,  56483,  46273,  36291,
 /* -15 */  29154,  23254,  18705,  14949,  11916,
 /* -10 */  9548,   7620,   6100,   4904,   3906,
 /*  -5 */  3121,   2501,   1991,   1586,   1277,
 /*   0 */  1024,   820,    655,    526,    423,
 /*   5 */  335,    272,    215,    172,    137,
 /*  10 */  110,    87,     70,     56,     45,
 /*  15 */  36,     29,     23,     18,     15,
};
```

The value in the comment tracks the *nice value* of the task. We can see that tasks with lower *nice value* get higher *load weight*. The nice value zero, for example, matches the *load weight* 1024. If the *sched_entity* object represents a cgroup, then its *load weight* is roughly[1] the cgroup's *cpu.shares* value.

3.4.2 Calculating the scheduling time-slice.

We will describe the idea how the time slice window of a task is calculated. The corresponding function is implemented in the `sched_slice()` routine of the *CFS scheduler*. From the scheduler's point of view, our task is simply a *sched_entity* object that is added to a red-black tree rooted in a *cfs_rq* runqueue. Our example task is a member of a child cgroup, therefore the task is not enqueued in the main per-CPU *cfs_rq* runqueue, but it is enqueued in the appropriate *cfs_rq* runqueue of the child cgroup's taskgroup.

To calculate the length of the time slice window for our example task, we will need to travel up the *sched_entity* hierarchy until we get to the main per-CPU scheduler red-black tree and its *cfs_rq* runqueue. We begin with our task's *sched_entity* that is enqueued in the child taskgroup's *cfs_rq* runqueue.

[1] In uniprocessor case, it is exactly the cgroup's *cpu.shares* value. In SMP case, the value is slightly biased by calculations based on the CPU load. See the `calc_cfs_shares()` and `update_cfs_shares()` functions in `kernel/sched/fair.c` for more details.

Firstly, the time-slice length is set to an initial value. After that, We look at the *sched_entity* of our task and at the *cfs_rq* runqueue where the *sched_entity* is enqueued. Now, we multiply the current time-slice length value by the *load weight* of the *sched_entity* and divide it by the overall *load* of the *cfs_rq* runqueue. The overall *load* of a *cfs_rq* runqueue is the sum of *load weights* of the *sched_entities* it contains. Since we begin at the bottom of the hierarchy, the *load weight* of the *sched_entity* is the *load weight* of the task which is 1024, because our task has nice value zero.

Now, we are going to move one step up in the *sched_entity* hierarchy. Our current *sched_entity* is the *sched_entity* of the child cgroup's taskgroup. And our current *cfs_rq* runqueue is the *cfs_rq* runqueue of the root cgroup's taskgroup. Again, we multiply the current scheduling slice length with the *load weight* of the *sched_entity* and divide it by the overall *load* of the *cfs_rq* runqueue. This time, the *load weight* of the *sched_entity* is the *cpu.shares* value of the child cgroup.

We need to take one more step up in the hierarchy. Our current *sched_entity* is the *sched_entity* of the root cgroup's taskgroup. And our current *cfs_rq* runqueue is the main per-CPU *cfs_rq* runqueue. Now, we perform the same calculation according to the pattern in the previous steps. The calculation of the scheduling time slice length is complete.

4. The Linux blkio cgroup subsystem

4.1 Introduction

The Linux *blkio cgroup subsystem* is responsible for ensuring the *cgroup* limits for block I/O access. We have seen in the first chapter how system administrators can specify bandwidth limits for *read* and *write* access to block I/O devices. In this chapter, we are going to look closely at the implementation how these limits are enforced.

4.2 The Linux generic block layer

The *blkio cgroup subsystem* is implemented in the part of Linux kernel that is known as the *generic block layer*. In the following figure, we show an overview of kernel components involved in block I/O operations:

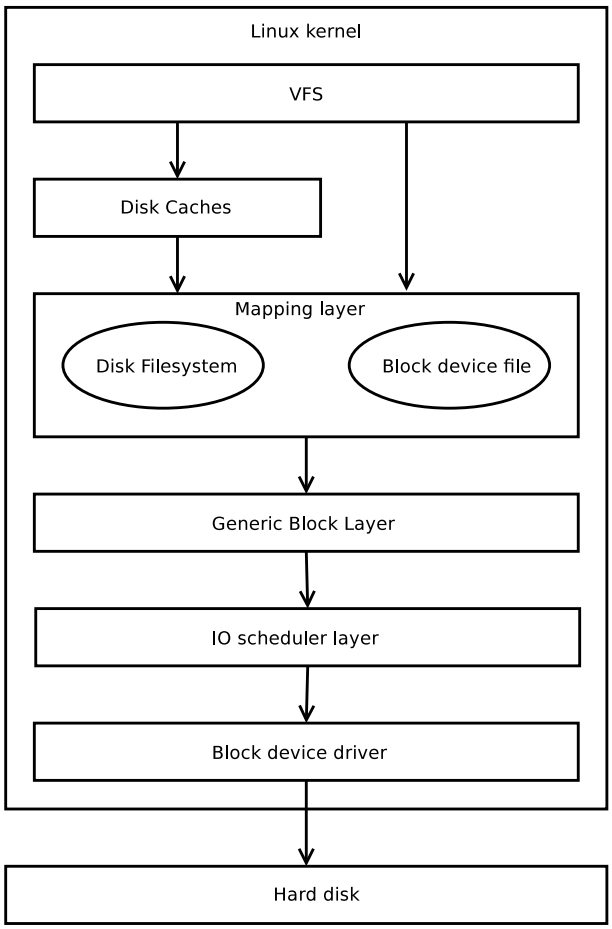


Figure 4.1: Kernel components involved in block I/O

Source: *Understanding the Linux kernel*[6]

The first kernel layer to handle *read* and *write* system calls is the *VFS virtual*

filesystem layer. This layer firstly looks into a disk cache named the *page cache* to see if the requested data is already available in the main memory. If the data is found in the cache, the slow access to the block device is avoided. If it is not the case, the request is passed on to the *mapping layer*. This layer contains the actual filesystem-specific routines to find the logical block numbers of the disk blocks containing the requested data. After this step is performed, the *Generic block layer* can start the actual block device I/O operation. The single block I/O operation is represented in the kernel by `struct bio`. We will look at this structure more closely later on in this chapter. The *generic block layer* submits these *bios* to the *IO scheduler layer* that performs rescheduling of the block I/O access requests to optimise the block device operation. Finally, the *block device driver* is responsible for the actual data transfer between the block device and the kernel.

One of the important implications of the fact that the block I/O throttling is implemented in the *generic block layer* is that the throttling will take no effect if the requested data is found to be present in the disk *page cache*. This is because the *VFS* layer does not need to consult the lower block I/O layers in that case. This can be easily proved by setting a cgroup limit for block device read access. If a file is copied from the device for the first time, the read access to the file on the device will be restricted according to the cgroups limits set. If, however, the copy command is repeated, the cgroup blkio limits will take no effect because the data will already be available in the page cache.

4.3 The bio structure

The `struct bio` is the core data structure of the *generic block layer*. It is a command descriptor that stores the block device command parameters. Basically, this data structure maps a *contiguous* set of disk sectors to an array of memory *segments*. The memory *segment* defines the location in the main memory that is used as the data source - in case of the *write* operation, or as the data destination - in case of the *read* operation.

As we have already mentioned, the `struct bio` maps a *contiguous* set of disk sectors to an array of memory *segments*. This allows to define compound commands that can be efficiently handled by a single *scatter-gather DMA transfer* - if such functionality is provided by the block device controller. If it is not the case, the single *bio* can be handled by multiple device controller data transfers.

Now, we show an excerpt from the `struct bio` definition in the *Linux kernel* sources:

```
struct bio {
        sector_t                bi_sector;
        struct block_device     *bi_bdev;
        unsigned long           bi_rw;
        unsigned short          bi_vcnt;
        unsigned int            bi_size;
        struct bio_vec          *bi_io_vec;
...
```

```
#ifdef CONFIG_BLK_CGROUP
        struct cgroup_subsys_state *bi_css;
        ...
#endif
...
};
```

The `sector_t bi_sector` parameter stores the address of the disk sector where the *read/write* operation starts. The `struct block_device *bi_bdev` identifies the block device on which the operation is to be performed. The `unsigned int bi_size` stores the number of bytes yet to be transferred in the block device operation. Finally, the `struct bio_vec *bi_io_vec` and `unsigned short bi_vcnt` parameters store the address and the length of the array of the `struct bio_vec` objects. Each such object defines a single segment in the main memory. The direction of the command (*read/write*) is determined by the `unsigned long bi_rw` parameter.

If the *blkio cgroup subsystem* is enabled, the `struct cgroup_subsys_state *bi_css` parameter joins the *bio* with the *blkio subsystem*.

4.4 Interconnecting the bio structure with the blkio cgroup subsystem

Firstly, let us remind how the general cgroup subsystem infrastructure looks from the kernel point of view:

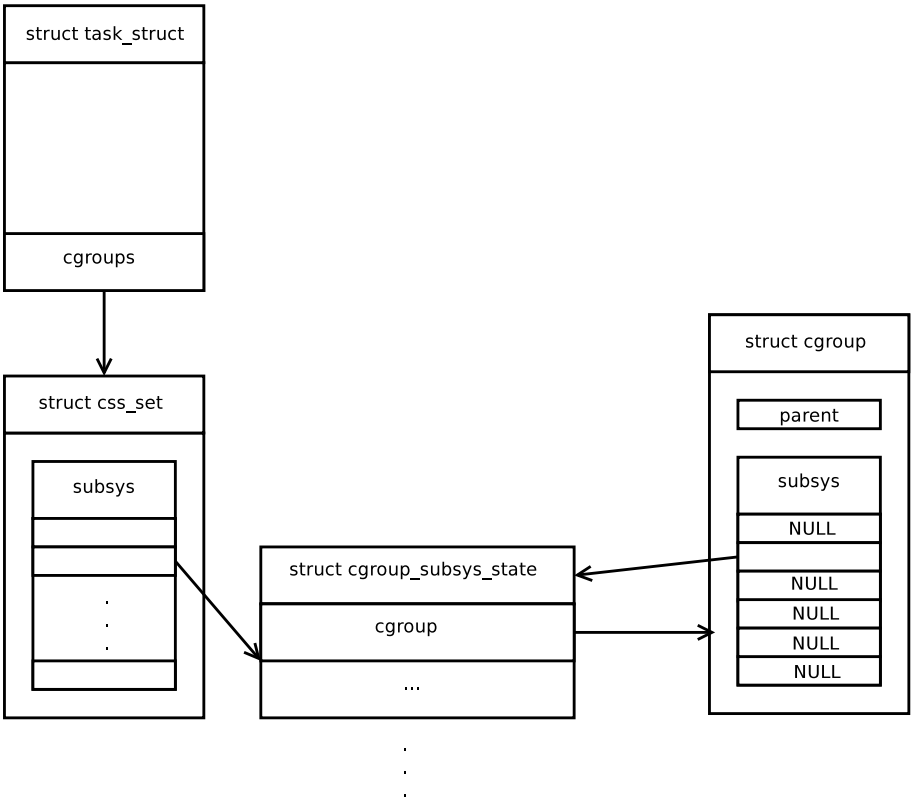


Figure 4.2: The general kernel cgroup subsystem infrastructure

In the figure, we see a *task* represented by the `struct task_struct` and how it references a *css_set* object. For every available *cgroup subsystem*, the *css_set* object contains a reference to the `struct cgroup_subsys_state` object, that in turn references the actual `struct cgroup` object. As we have also seen in the chapter about the *CPU cgroup subsystem*, the `struct cgroup_subsys_state` objects are extended in the *OOP*-fashion by subsystem-specific structures for individual *cgroup subsystems*.

In the case of the *CPU cgroup subsystem*, the `struct cgroup_subsys_state` objects were extended by the `struct task_group` structure. See the chapter about the *CPU cgroup subsystem* for more details. In the case of the *blkio* subsystem, the `struct cgroup_subsys_state` structure is extended by the `struct blkcg` structure, the so-called *block cgroup*.

Here, we show an excerpt from the definition of the *block cgroup*:

```
struct blkcg {
        struct cgroup_subsys_state      css;
        ...
        struct radix_tree_root          blkg_tree;
        ...
};
```

This structure definition is quite simple. We are going to look at the *radix tree* [1] data structure `struct radix_tree_root blkg_tree`. Logical block devices (which usually represent a real hardware block device) are represented in the kernel by `struct gendisk`. Every *gendisk* object has a *request queue* associated with it. The radix tree *blkg_tree* of the *blkcg* structure stores associations between the *block cgroups* and the *request queues*. These association objects are defined by the `struct blkcg_gq` data structure and are indexed in the tree by the *request queue* identifier. They also store data for (at most two different) *block cgroup policies*. That means, for a given *block cgroup*, there can be different *block cgroup policy* specifications for separate request queues (and thus for separate logical block devices). In this thesis, we are only interested in the *throttling block cgroup policy*.

Now we can inspect the definition of the `struct blkcg_gq` association objects that link a *request queue* with a *block cgroup* and store *block cgroup policy* specifications:

```
struct blkcg_gq {
        struct request_queue            *q;
        ...
        struct blkcg                    *blkcg;
        struct blkg_policy_data         *pd[BLKCG_MAX_POLS];
        ...
};
```

In the case of the *throttling block cgroup policy*, the `struct blkg_policy_data` structure is extended by the `struct throtl_grp` structure. These *throtl_grp* objects store the actual read/write IO limits set by the computer administrator via

[1] The Linux *radix tree* implementation is quite specific and can be thought of as a mapping function from `long` to `void*`. For more details, see [7]

the *cgroup* interface. These objects also contain linked lists of `struct bio` block device command descriptors that could not be processed immediately and their dispatch to the lower kernel layers had to be delayed because of the limits set.

To sum up, a `struct throtl_grp` object encapsulates the delayed `struct bio` block device command descriptors and IO limits for a specific *cgroup* and a specific *logical block device*. The *logical block device* is represented in this case by its *request queue*.

Now we are going to inspect the `struct throtl_grp` definition:

```
struct throtl_grp {
        struct blkg_policy_data pd;
        struct rb_node rb_node;
        unsigned long disptime;
        struct bio_list bio_lists[2];
        uint64_t bps[2];
        ...
};
```

The `struct rb_node rb_node` and `unsigned long disptime` members are used by the algorithm that ensures the timely processing of the delayed *bios*. We will inspect this algorithm in more detail in the next section. The `struct bio_list bio_lists[2]` contains two lists of delayed *bios* - one for the *read* direction and one for the *write* direction. The `uint64_t bps[2]` parameter stores the limits in the bytes per second units for both IO directions.

We have now enough information to view the important structures of the *blkio cgroup subsystem* infrastructure and the relationships between them. You can inspect them in the figure 4.3.

4.5 Ensuring the *blkio cgroup* limits

4.5.1 The *throtl_data* structure

When the kernel processes *bios*, it checks if processing the *bio* would exceed the limits set in the respective *throtl_grp* where the *bio* belongs. Should the limits be exceeded, the *bio* is attached to the *throtl_grp's* list of delayed bios for later processing.

The timely processing of the delayed *bios* is based on the *throtl_grp* data structures that store the throttled *bios* belonging to the same *cgroup* and to the same *logical block device*. The *request queue* of the *logical block device* has a reference to the `struct throtl_data` data structure which contains a *red-black* tree of the *throtl_grp* objects with *bios* enqueued for delayed processing and belonging to this block device. The tree is named *tg_service_tree*.

Thus, the *tg_service_tree* is a red-black tree of *throtl_grp* objects. Every *throtl_grp* object in the tree represents a different *cgroup* but they all belong to the same *request queue* and thus to the same *logical block device*. The *throtl_grp* objects in the tree are ordered by the *disptime* parameter that stores a suggested *dispatch time* when the *bios* blocked on the *throtl_grp* should be dispatched.

Now, let us have a closer look at the `struct throtl_data` data structure definition taken directly from the kernel sources:

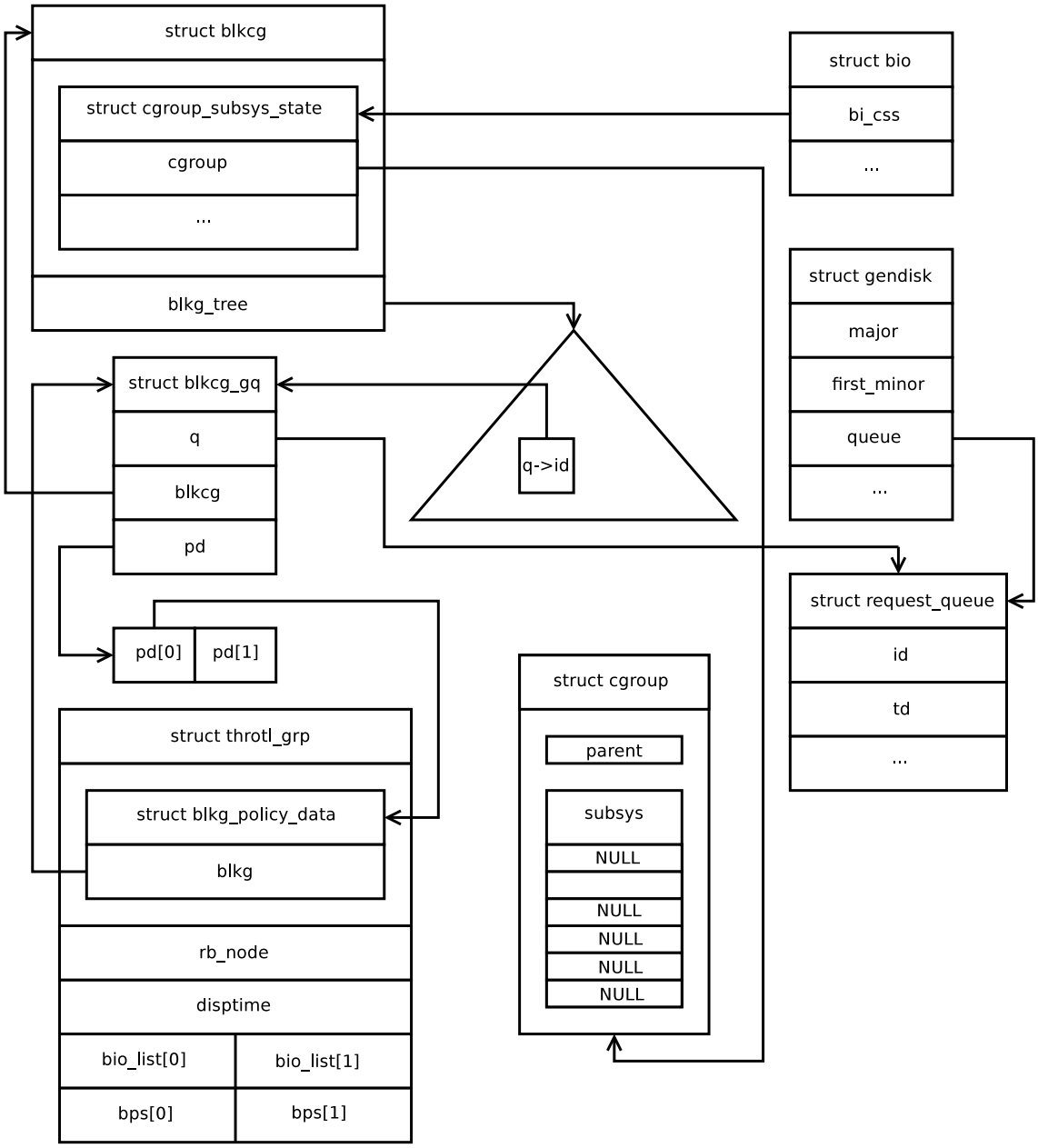


Figure 4.3: The *blkio cgroup subsystem* infrastructure

```
struct throtl_data {
        struct throtl_rb_root tg_service_tree;
        struct request_queue *queue;
        unsigned int nr_queued[2];
        struct delayed_work throtl_work;
        ...
};
```

We see here the *tg_service_tree* definition and a reference to the *request queue* of the *logical block device*. The `unsigned int nr_queued[2]` array stores the overall count of *read/write* bios delayed on the *request queue*. And the *throtl_work* member connects the *throtl_data* object with the underlying *workqueue* infrastructure that drives the timely processing of events. In the next figure 4.4 we show

the `struct throtl_data` structure and its relationship to the `struct gendisk` structure that represents a logical block device:

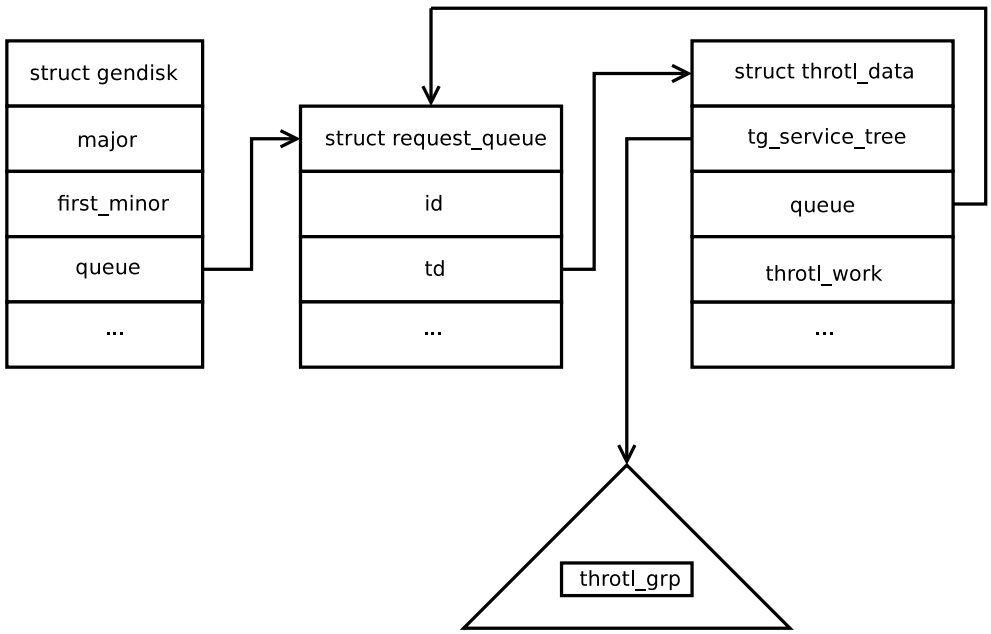


Figure 4.4: The *throtl_data* structure and its relationship to the *gendisk* structure representing a *logical block device*

4.5.2 Enqueuing the delayed *bios.*

If the kernel finds that processing the *bio* request would exceed the defined *cgroup blkio* limits, it works as follows: Firstly, the *bio* is added to the list of the delayed bios for the *read/write* direction in the respective *throtl_grp*. Then, the whole *throtl_grp* object is inserted (if it is not already there) into the associated *throtl_data's* red-black *tg_service_tree*. Now, the dispatch time of the *throtl_grp* is (re)calculated and a new bio dispatch function invocation is scheduled accordingly using the underlying kernel *workqueue* framework.

4.5.3 Dispatching the delayed *bios.*

Dispatching of the delayed *bios* starts when the underlying *timer workqueue* framework fires up the dispatch invocation function. It works by removing the *throtl_grps* off the *tg_service_tree* and dispatching the bios stored there. The maximum number of *bios* dispatched at one time is limited by a constant value so that the underlying *block device* can handle them. Because a single *throtl_grp* represents here a single *cgroup*, this could lead to the situation where only *bios* from one *cgroup* are processed and further processing would be temporarily blocked. This is avoided by setting another constant limit on the maximum number of *bios* processed from a single *cgroup* at one time. *Read* and *Write bios* are handled with different priorities. The code tries to dispatch 75% *reads* and 25% *writes* during one invocation of the deferred *bio* dispatch invocation function.

5. Analysis

5.1 Introduction

In this chapter we are going to analyse various possible approaches to the problem of limiting the *relative CPU usage* and the *block IO bandwidth*. We will consider the implications our analysis yields for the prototype implementation that will be presented in the next chapter. In the previous chapters we have provided an in-depth analysis how the *Linux* tackles the problem by employing *cgroups*. Now we will briefly sketch other possible solutions and speculate over their advantages and disadvantages.

We will start our analysis with the *relative CPU usage limits*. We will see how these limits are best expressed and what exactly they mean. Then we will analyse possible implementation approaches. The next part of this chapter will deal with *block IO limits*. Again, we will firstly see how this limits could be expressed and at which kernel layer they should be interpreted. An analysis of possible implementation methods will follow.

5.2 Specifying the *CPU usage* limits

Before we start considering the implementation details of various possible ways to enforce the *CPU usage limits*, we firstly need to make clear what these limits actually mean. We will differentiate between the so called *hard limits* and *soft limits* for the *CPU usage*.

When talking about the *CPU limits* in common computer architectures, we will also look at how these limits should be interpreted for *uniprocessor* and *multiprocessor* architectures that both now have sound support in prevalent operating systems and thus also in the *FreeBSD*.

Another important point to consider is the entity to which the limits are applied. If it is just a single thread, a single process or a group of processes.

5.2.1 *Hard* versus *soft* CPU limits

Both *hard* and *soft* CPU limits are best defined by *relative ratios*. The *ratio* can have the form of two numbers specifying the *nominator* and the *denominator* values. Or in our opinion more conveniently and more user-friendly, it can be specified by a single percentage value. For our purposes of specifying *CPU limits*, both forms are interchangeable and the percentage value form will be preferred.

Hard CPU limits for a thread mean that the thread should never get more relative CPU time than the specified percentage value. This is also true even if the CPU core should be otherwise *idle*. That is in clear contrast with the *soft CPU limits* where the thread is allowed to exceed its limits if there is no thread CPU contention in place.

In practice, both approaches are used. The parts of the *Linux kernel cgroups implementation* presented in the previous chapters are a nice example of the *soft CPU limits* approach. In our own prototype implementation, we will use the *hard CPU limits* approach, but we will also sketch a way how our implementation

could be extended to also support the *soft CPU limits*. *Soft CPU limits* are best suited for desktop environments. As noted in [8], we can see following areas where enforcing *hard* CPU limits is beneficial:

1. Hard limits can be used in *Pay-per-use* enterprise environments where customers demand a certain amount of the CPU resources and pay only for that.

2. They can be also used in virtualization environments. For example in the case of the *FreeBSD jail* - which is an operating system level virtualization schema for *FreeBSD* - the *hard limits* can be used to make sure a particular *jail* does not exceed its CPU entitlement.

3. *Hard limits* can be used to provide guarantees.

5.2.2 Specifying *CPU limits* on uniprocessor and multiprocessor architectures.

We consider a single percentage value the best way to specify *CPU usage limits*, because it is user-friendly and easy to understand. In the case of the *uniprocessor* architectures, the meaning of the limit is straightforward: For example if the limit for the process p is set to the value 50%, then during an arbitrary time interval, the thread p should not be running on the CPU for more than 50% of the time.

However, the situation can become a litter trickier in multiprocessor architectures. In this case, at least two reasonable interpretations of the single percentage value could be used. The question is, if the percentage value limit is to be applied per a single processor, or it should be applied to the overall computing power of the machine.

In our prototype *FreeBSD* implementation, we will apply the *CPU percentage limits* set for the process p per single processor. This interpretation brings us several advantages: Firstly, we can directly use the provided CPU accounting information from the *scheduler* code, because this accounting is done per single CPU. The second advantage is that our per-processor interpretation is also compatible with commonly used UNIX tools like *top* and *ps*. However, there are also tools in the *FreeBSD* operating system, that display CPU percentage utilization ratio based to the overall computing power, for example the *vmstat* tool works this way.

Another important point to note about the *multiprocessor architectures* is that the single percentage limit value is not linked to any specific *CPU chip*, nor is the limit applied separately to all the processor chips. Although a process sometimes migrates from one CPU to another, the limit is still applied the same way as it would be applied if the process was always running on the same processor. That means, if we have two processors in the machine and the *CPU limit* for the process p has been set to 50%, then the process p is allowed to run 50% of the time on one processor, or it can run for example 20% of the time on the first processor and the remaining 30% of the time on the second processor. The important thing is that the overall time the process p is running is independent of the process p's cpu affinity which could change in time. This is the way how the *cpu percentage limit* value is interpreted in our prototype implementation and it

is also compatible with the *CPU usage* accounting code already present in the *FreeBSD schedulers*.

5.2.3 Specifying the *CPU limits* for different *entities*.

In the previous section, we have exactly specified what it means if the *CPU limit* for a process p is set to some value, for example 50%. However, it is often desirable to specify the *CPU limits* for other entities like groups of processes, or *process containers* and *cgroups* in the *Linux* world. The groups of processes could be based on several criteria - for example *user id*, *jail id*, or they could be built manually as well. Such ability gives the computer administrator much greater flexibility, and its implementation cost is usually not too big compared to the benefits it brings - once the per-process limits are already implemented.

The natural way to implement such functionality is to use a simple additive function to account the *CPU* usage of the process group as the sum of the *CPU usage* values of its members. This is how we make it in our prototype implementation and the similar way is also used in the *Linux cgroups* - where the overall *cgroup scheduling weight* is distributed among all the child processes and *cgroups*.

5.3 Implementing the *CPU usage limits*

We will now evaluate various possible approaches how to actually implement the *CPU limits* enforcement. Then we will be able to choose one approach that best suits our needs for the prototype implementation presented in the next chapter.

5.3.1 Implementing the *CPU limits* at the scheduler level

Probably the most intuitive way to implement the *CPU limits* would be to create a brand new process scheduler or to adjust the already existing *FreeBSD process schedulers* to honour the *CPU limits* set. This approach is for example also used in the case of the Linux *CFS scheduler* which has full *cgroup* support. This is however easier said than done and brings some severe complications that would be beyond the scope of our resources.

Firstly, it would not be too difficult to create some new scheduler that would also honour the hard *CPU limits*. However, it would be difficult to create such scheduler that would also be *competitive* with the modern *ULE FreeBSD scheduler* that shows very good scalability results on multiprocessor machines, its scheduling decision are time-effective and also provides good interactivity experience.

We also decide not to adjust the existing *FreeBSD* schedulers to support the *relative CPU limits* because that would considerably change the logic how these schedulers work. In some cases this would cause problems to other kernel parts that rely on some scheduler invariants that are not well documented and these problems would be difficult to debug.

There are other difficulties we would need to overcome if we wanted to connect the *FreeBSD racct/rctl infrastructure*[1] directly with the scheduler code. The

[1]The *racct/rctl* framework is a resource limiting framework implemented in the FreeBSD kernel. Our prototype implementation presented in the next chapter will extend this framework to add support for the *relative CPU usage* limits and the *block IO bandwidth* limits.

important parts of the FreeBSD scheduler code run protected under the *spin locks*. On the other hand, the *racct/rctl* infrastructure is protected by *sleepable locks*. That would prevent accessing the *racct/rctl* data structures directly from the scheduler code. Yes, this issue is definitely solvable, but it also helps us to decide not to implement the *CPU limits* at the scheduler level.

On the contrary, our prototype implementation will be scheduler independent. This has the advantage that our implementation will work with both *FreeBSD schedulers* and the necessary kernel changes made will not be so dramatic.

5.3.2 Ensuring per-process *CPU limits* by manipulating the process scheduling priority

Another possible approach to enforce the per-process *CPU limits* could involve modifying the process scheduling priority. However, this approach is not possible for implementing the *hard CPU limits* in *FreeBSD*, because even a low priority process can starve the CPU usage to 100% if there are no other processes competing for the same CPU. So we will analyse here how (and if) changing the process priority could be used to ensure the enforcement of the *relative CPU limits*. The *FreeBSD* operating system contains two different schedulers and the user can choose one before the compilation of the system. The older one is the *4BSD scheduler* and the new (and the default) one is the *ULE scheduler*.

The 4BSD scheduler

The *4BSD* scheduler works by choosing the thread with the semantically highest priority in a round-robin fashion as the next thread to run. The CPU starvation is prevented by a periodic timer that adjusts the threads' priorities. So the key to the scheduling fairness in this scheduler is the algorithm that recalculates the process priorities based on their recent CPU usage. This algorithm could be modified to semantically lower the scheduling priority of a thread if the thread exceeds its *CPU limits*. This would have the effect that if there are any other runnable threads with the same priority, they will be preferred. And this is exactly what the *relative CPU limits* are for.

The ULE scheduler

The *ULE* scheduler works in a different way and it does not use the process priority to ensure the scheduling fairness. To ensure fairness, the scheduler keeps two different queues, the *current* and the *next*. Threads are chosen from the *current* queue in their priority order. When the *current* queue is empty, it is switched with the *next* queue and it all continues the same way. Interactive threads are inserted into the *current* queue, the other threads are inserted into the *next* queue.

As we can see, once every two switches of the the *current* and the *next* queues, every thread will be given a chance to run regardless of its scheduling priority. This means that the *relative CPU limits* can not be implemented for this scheduler simply by manipulating the process's scheduling priority.

5.3.3 Implementing the *hard* CPU usage limits by the *stop-and-run* technique

We are now going to consider the defensive approach that we name *stop-and-run*. It works like this: Whenever there is a processes exceeding its *CPU usage limits*, the process is paused until its CPU usage statistics fall again into the allowed range. At this point, the process can be resumed and it continues to run until the limits are again exceeded. This approach could be implemented both in the *user-space* and in the *kernel-space*.

The user-space approach

In this approach, there could be a userspace *daemon* process running in the background and periodically monitoring the cpu usage statistics of all the running processes. Once an offending process is found that exceeds its limits, the process is sent a *SIGSTOP* signal. To resume the process, it is sent the *SIGCONT* signal. This approach has a basic sense of security, because the *SIGSTOP* signal can not be ignored. This is for example how the *cpulimit*[2] tool is implemented.

This approach could be also used to implement the *CPU limits* for various *process groups*, based for example on process *real user-id*. The algorithm would work this way: If the per-user *CPU* limit has been exceeded, all processes belonging to the offending user would be temporarily paused.

The kernel-space approach

The same technique could be implemented directly in the *FreeBSD kernel*. The per-process *CPU usage* statistics can be periodically collected directly from the *FreeBSD* schedulers. As soon as the process exceeds its limits, it is forced to sleep. When its *CPU usage* falls back into the requested range, the process is awaken. This way, the limits can be implemented per-process, per-user or per any other process groups that are considered necessary.

So now comes the question why would anybody implement this technique for *FreeBSD* in the kernel. In addition to the better transparency, this solution can provide an additional benefit. It can be further extended to provide support for *soft CPU limits*. For example in the case of the *per-user soft CPU limits*, there could be added a little bit of bookkeeping code to the scheduler that would tell if on a given CPU core, there are running at least two processes with different *real user-ids*. If this is true, the limits are imposed by pausing all the processes of the offending user on this CPU core. (Other CPU cores are handled separately in the same way) However, if all the processes on the CPU core belong to the same user, the limits need not to be imposed on this particular CPU. This is in line with the *soft CPU limits* semantics.

There are some reasons why extending the user-space approach in a similar way to support *soft CPU limits* would not be practical. Firstly, process affinity can change during the process's runtime and it also usually does. (For example when the scheduler tries to balance the CPU load on multiple CPU cores). So the statistics taken from some user-space daemon saying on which CPU core the process is currently running could already be obsolete at the time they are being

[2]For more information, visit the project homepage at http://cpulimit.sourceforge.net/

interpreted. This would also increase the reaction time after which the offending processes would be paused. In the kernel-space approach, however, as soon as the scheduler schedules the process on some CPU core, it knows it immediately.

On the other hand, if we wanted to make this reaction time as little as possible, our user-space daemon could itself add a considerable amount to the CPU load of the system and that would also render the user-space solution impractical.

5.3.4 Overview of the implementation approaches for imposing *CPU usage limits*

Here we present the summary of possible approaches to our problem that we have already evaluated. We also list their most notable advantages and disadvantages.

implementation approach	advantages	disadvantages
create a new scheduler or possibly rework an existing one	clean and accurate solution	scheduler dependent, may break nice scheduling behaviour in some cases (effective workload balancing on multiprocessor architectures, scheduling fairness, favouring of interactive processes)
manipulating the scheduling priority of a process	easy to understand	scheduler dependent, does not work for the ULE scheduler without further changes to the scheduling logic
stop and run technique - userspace daemon	no kernel modifications, easy to implement, scheduler independent	not very accurate since the inherently defensive nature
stop and run technique - kernel implementation	easy to implement, scheduler independent, possibility to support also *soft CPU limits*	not very accurate since the inherently defensive nature

Based on the analysis, we choose to use the *stop and run technique* kernel-space approach in our prototype implementation that will be presented in the next chapter.

5.4 Specifying the *block IO bandwidth* limits

For our purposes, the *block IO limits* are best defined by specifying the number of bytes that are allowed to be transferred in read/write operations on a block device per second. The read and write limits can be defined independently of each other.

We need to emphasize that reading a file that is stored on some block device does not always mean that the actual data transfer between the block device and the main memory takes place. This is possible because the data stored in the file may already be cached in the main memory from the previous block device operations. So by specifying the block device read and write limits we really do want to limit only the data transfer coming to or originating from the block device. We are not interested in limiting the read and write operations as long as they involve only the data transfer between the cache memories managed by the operating system and do not involve the direct read/write data transfers between the block device.

Now, we take a look at the general architecture of the *FreeBSD block device IO subsystem* so that we can understand at which layer the specified limits will be enforced:

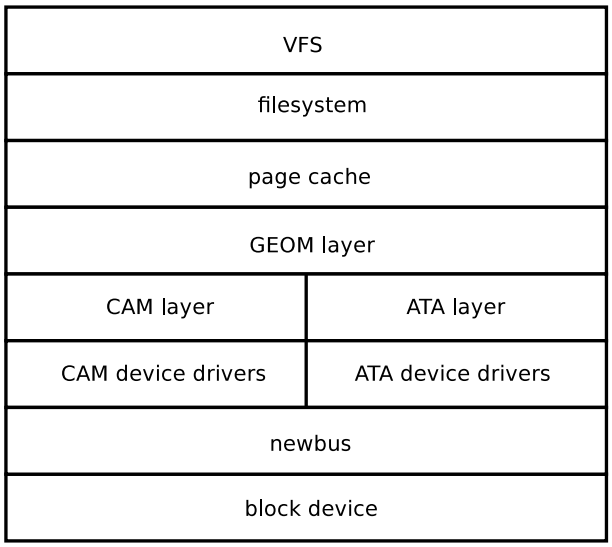


Figure 5.1: The general *FreeBSD block device IO subsystem* architecture

Source: *The Design and Implementation of the FreeBSD Operating System*[10]

We see that enforcing the limits at the *VFS layer* would not have the desired effect because this way the cached operations would be limited too. Therefore, the limits will be enforced at the *GEOM layer*. This is analogical to the *Linux cgroups blkio subsystem* implementation where the block IO limits are enforced at the *generic block layer*.

5.5 Implementing the *block IO bandwidth* limits

The problem of managing the traffic bandwidth is well known and well studied in the field of *computer networks*. We will now have a look at some common *traffic shaping* algorithms described in *Computer Networks*[11] and evaluate how they suit our needs of limiting the block device IO bandwidth.

5.5.1 The *Leaky bucket* algorithm

This algorithm is based on an analogy of a bucket with small holes at the bottom. If the bucket contains water, then it will leak at constant rate that is independent of the rate at which the water enters the bucket. Also, if the amount of water in the bucket reaches its capacity, any additional water would overflow - that means it would be lost because it does not come out of the output stream of the bucket under the hole.

A possible implementation of this algorithm includes a *queue* data structure that serves as our fixed-capacity bucket. As the data packets arrive, they are enqueued into the queue. Also, as long as the queue is not empty, the data packets are taken off the queue at constant speed.

Thus, this algorithm produces an output data stream with speed that is lower or equal to a constant value. In some network scenarios, this might be the required behaviour. But this is not what we are looking for. Firstly, we have already decided that our bandwidth limits will be specified per second. Which is quite a big time interval when it comes to computer processing. So a reasonable implementation of the *leaky bucket algorithm* could divide the imposed limit by 1000 for example and apply it per millisecond. Surely, this would provide a stable output stream to or from the block device, but this kind of limiting is actually more than we really want.

What we really want to limit is exactly specified by the allowed amount of bytes per second. And nothing more. We are not concerned about limiting any short data bursts as long as they comply with the per-second limit. That is why this solution does not suit our needs.

5.5.2 The *Token bucket* algorithm

The *token bucket* algorithm is more flexible than the *leaky bucket* algorithm and allows limited burstiness in the output data stream. This algorithm is based on *tokens* that are inserted into the bucket at *fixed rate*. A single token is added to the bucket every $1/n$ of the second. The bucket has fixed capacity of tokens. If its capacity is reached, the additional tokens are simply discarded. If an input data packet arrives, it consumes a single token from the bucket (or a number of tokens that is proportional to the data size of the packet) and the data packet is sent out. If there is no token in the bucket, the processing of the input packet is delayed until the tokens are again available.

If the input stream stands still for some time, the bucket has a chance to refill itself with tokens to its full capacity. Then, if some data arrives, it is processed in greater speed until the tokens are used up. A slight modification of this algorithm will be used in our prototype implementation.

A variation of the *Token bucket* algorithm that we will use for imposing the block IO bandwidth limits

The total token capacity of the bucket will be equal to the number of bytes allowed to be transferred to or from the device in the specified direction. There are two separate buckets - one for the read and one for the write direction. At the beginning of each time slice, the buckets will be fully filled with tokens. When the

check is performed to see how many available tokens are there in the appropriate bucket, only the portion of the tokens in the bucket is seen. This portion is proportional to the time passed since the beginning of the current time-slice. More specifically, if the one tenth of the time-slice has passed, only one tenth of the tokens is seen in the bucket. At the end of the time-slice, all tokens will be visible.

As the data packets come from or to the block device, they are passed on as long as there are sufficient visible tokens in the appropriate bucket and the necessary tokens are then consumed. If there are no enough visible tokens in the appropriate bucket, the processing of the input packet will be delayed for the time that is proportional to the size of the data that overflows the imposed bandwidth limits.

In our prototype implementation, the increasing of the amount of tokens in the bucket by only a single token would be ineffective. Such increments would need to happen quite often (The frequency being the numerical value of the bandwidth limit per second in bytes). Also, every such increment must be done inside a critical section to prevent possible data corruption. For this reason, the bucket is fully filled with tokens at the beginning of every time slice. This is not a problem, because if there are no enough tokens available in the bucket anymore, the delay time for the processing of the input packet is exactly calculated and the delayed processing of the packet is registered using an underlying timer framework. That means we are no longer dependent on the amount of available tokens being incremented by one at the fixed and steady rate.

5.6 Integrating the prototype implementation within the *FreeBSD* kernel

In the *FreeBSD* kernel, there is no such framework that would resemble the architecture of the *Linux cgroups*. This simple fact leads us to a decision if we will implement a new *cgroup framework* for the *FreeBSD* operating system. However, there is a relatively new codebase in the *FreeBSD* kernel named the *racct/rctl infrastructure* that has a similar purpose as the *Linux cgroups* - that is *limiting resource usage*. It has support for various system resources, however it lacks support for limiting the *relative CPU usage* and the *block IO bandwidth*.

We decide to take advantage of the *racct/rctl* framework and instead of developing a new *cgroup* infrastructure from scratch, we will extend this existing framework to add support for the *relative CPU usage limits* and the *block IO bandwidth limits*.

6. Our prototype implementation

6.1 Introduction

After we have performed our analysis of the *Linux cgroups subsystem* in the first chapters and after we have analysed other possible solutions in the *Analysis* chapter, we will now present in this final chapter our prototype implementation of the *relative CPU limits* and the *block IO bandwidth limits*. Our code is provided in the form of patches for the *FreeBSD-CURRENT* branch of the *FreeBSD* source tree. The *FreeBSD-CURRENT* branch is the "bleeding-edge" development branch of the *FreeBSD* operating system where most of the experiments with new features take place.

We start this chapter by presenting the analysis of our implementation of the *relative CPU limits*. Firstly, we set forth the requirements that our implementation should fulfill. Once our requirements are clearly stated, the analysis of the actual implementation will follow. The same logic will be applied in the second part of this chapter, where we present our prototype implementation of the *block IO bandwidth limits*.

6.2 The relative CPU limits

6.2.1 Implementation requirements

We impose the following requirements on our prototype implementation:

1. Our tool will be fully integrated within the *FreeBSD racct/rctl* framework designed for specifying and imposing resource usage limits in the *FreeBSD* operating system.

2. The *CPU usage percentage* calculations of our tool will be compatible with the common userland tools *ps* and *top*.

3. We will support specifying relative CPU limits per-process, per-user and also per-jail.

4. We choose the defensive algorithm for our solution, named in the section 5.3.3 on page 37 as the *stop-and-run* technique. More specifically, we implement the kernel-space approach. This brings us some advantages but also disadvantages that we agree to accept.

6.2.2 Integrating the tool within the *FreeBSD kernel* and especially the *racct/rctl framework*

The core parts of the *FreeBSD* kernel *racct/rctl framework* are implemented in the files `src/sys/kern/kern_racct.c` and `src/sys/kern/kern_rctl.c` and these files are also the place where most of our code will be applied.

The *racctd* kernel thread

The *racct* subsystem contains one kernel thread *racctd* that is responsible for periodically collecting some per-process CPU usage statistics. This is the place where we hook our code to calculate the per-process CPU usage percentage.

The *racctd* works by periodically iterating over all the processes and recalculating the per-process CPU usage statistics. To properly implement the *stop-and-run* technique, we need to iterate over all the processes in the system in two passes. In the first pass, the necessary per-process CPU usage percentage statistics are collected. In the second pass, we iterate over every single process once again and we decide if the process should be throttled. The process should be throttled if some of the CPU usage limits imposed on the process or on the group of processes where the process belongs have been exceeded.

The reason why we need two passes to implement the desired behaviour is the *throttling fairness*. If, for example, a per-user CPU usage limit has been exceeded then we want to throttle all the processes that belong to the offending user. Thus, to decide if a per-user limit has been exceeded, we need to firstly build the per-user *CPU usage percentage* statistics. Such kind of information is reliably obtained only after all the processes in the system have been inspected for their individual *CPU usage percentage* statistics in the first pass.

Throttling and waking up the offending processes

When the *CPU usage* percentage value of a process is approaching the imposed *CPU usage limit* , the process will be throttled. This is handled by the routine `racct_proc_throttle(struct proc *p)`. The per-process flag `p_throttled` is firstly set to the logical value 1 so that this process should be throttled as soon as possible - when the appropriate time comes. We also inspect the current state of the process. If the process is found to be running or it is currently placed in the scheduler *runqueue*, we request the involuntary context switch for the process by setting the `TDF_NEEDRESCHED` bit in the `td_flags` field for all the threads that the offending process contains.

If a user process is running, sooner or later the execution context of the process will switch to the kernel mode. This may happen voluntarily - by calling a system call, or involuntarily - when for example a timer interrupt occurs. In any case, the kernel will try to do its work and then return the execution context again to the user mode so that the user-space process continues running. Before the kernel returns the execution to the user-space mode, it calls the `userret()` routine. We add a check at the end of this routine to see if the process should be throttled. That is, if the `p_throttled` per-process flag is set for the current process. If the flag is set, we *sleep* and thus do not return to the user mode immediately.

When the offending process is later found to be again within its *CPU usage limits*, the process is woken up. This handles the `racct_proc_wakeup(struct proc *p)` routine. After the process is woken up, it returns to the user mode and its user-space execution continues.

Thus, the *throttling* of the offending processes and then making them later again *runnable* is achieved without directly hacking the schedulers. This makes our solution scheduler-independent and that is a nice feature since the *FreeBSD* operating system currently supports two schedulers that are completely different.

Unbounded kernel execution

It is now clear from the previous section that the actual CPU throttling in our prototype implementation is implemented by inserting a *sleep* call at the kernel-userspace boundary. This also means, the process is never throttled while executing the code in kernel mode. Only after the work in kernel mode is done and the *userret* function is executed, the process may be throttled.

Therefore, if there is for example any faulty syscall implementation in the kernel that could starve the CPU resources significantly - maybe by employing an endless loop or a similar construct - our throttling would be ineffective. Such problems are not handled in our implementation. It is not our intention to create any safeguards against the side effects of the programming errors elsewhere in the kernel. If such problems do occur, they should be handled by eliminating the root cause of the problem, not just its side effects - which in our case is the superfluous CPU usage.

6.2.3 Calculating the CPU usage percentage

Calculating the *CPU usage percentage* value of the process p is handled in our tool by the `racct_getpcpu(struct proc *p, u_int pcpu)` subroutine. The first argument identifies the process whose CPU usage we are going to determine. The second argument is our own estimate of the *CPU usage percentage* for the specified process p. Normally, we calculate the percentage value from the statistics provided by the *scheduler* and ignore this estimated value. However, if the process p is very young - that means the process's runtime so far is less than 3 seconds - the statistics provided by the scheduler are unreliable. In that case, we use our own estimate to calculate the *CPU usage* percentage value and ignore the statistics provided by the scheduler. We have found the limit of 3 seconds empirically as a good value to differentiate the cases where the statistics provided by the scheduler should be used.

Ensuring compatibility with the userland *ps* tool

Firstly, we will see how the *FreeBSD* userland *ps* tool performs the per-process *CPU usage percentage* calculations. These calculations are handled by the following lines of code:

```
double
getpcpu(const KINFO *k)
{
...
#define fxtofl(fixpt)   ((double)(fixpt) / fscale)
...
return (100.0 * fxtofl(k->ki_p->ki_pctcpu) /
        (1.0 - exp(k->ki_p->ki_swtime * log(fxtofl(ccpu)))));
}
```

Mathematically, the computation is expressed in the following fraction:

$$100 \cdot \frac{ki\_pctcpu}{FSCALE} \cdot \frac{1}{1 - e^{ki\_swtime \cdot \log \frac{ccpu}{FSCALE}}} \tag{6.1}$$

The computation depends on the following variables:

FSCALE This is the scaling factor in the *FreeBSD* kernel for performing the fixed-point arithmetic. It is defined to have value 2048. Thus, in the fixed-point arithmetic, the value 0.5 is represented as integer number 1024.

ki_pctcpu The per-process CPU usage percentage value obtained from the kernel in fixed-point arithmetic format.

ki_swtime The time in seconds that has passed since the process started or it has been last swapped in.

ccpu The kernel-defined constant value. In the *4BSD* scheduler it is used as the per-second decaying factor for the CPU usage percentage calculations. It is defined in this scheduler to the value $e^{-\frac{1}{20}}$ and is meant to decay 95% of the percentage value in 60 seconds. $((e^{-\frac{1}{20}})^{60} \approx 0.05)$. In the *ULE* scheduler, the value is not used. For compatibility with the *ps* tool, the *ccpu* kernel constant is defined in the *ULE* scheduler to the value 0.

Because the *ccpu* value is different for the *4BSD* and the *ULE* schedulers, the CPU usage percentage calculations also differ depending on the current scheduler used. In the case of the *ULE* scheduler, the *ccpu* value is defined to be 0, which simplifies the calculations significantly. Thus, in the case of the *ULE* scheduler, the computation 6.1 can be simplified as follows:

$$100 \cdot \frac{ki\_pctcpu}{FSCALE} \tag{6.2}$$

In the case of the *4BSD* scheduler, the calculation 6.1 yields the following formula:

$$\frac{100 \cdot ki\_pctcpu}{FSCALE - FSCALE \cdot e^{-\frac{ki\_swtime}{20}}} \tag{6.3}$$

Thus, to calculate the per-process *CPU usage percentage* value, we use the formula 6.2 in our tool if the *ULE* scheduler is compiled into the kernel. This is a simple formula and it is very convenient because it does not involve any floating point functions.

However, in case of the *4BSD* scheduler, things are not so easy. To properly evaluate the formula 6.3, we need to be able to calculate the exponential function. And this is a problem in kernel-space in general and especially in the *FreeBSD*. The usage of the *floating point unit* is best avoided in kernel-space because of the overhead of swapping the contents of the *FPU* registers on every context switch. For this reason, the *FreeBSD* kernel does not support the *floating point unit* operations at all, as some conversations of the *FreeBSD* developers in the *freebsd-hackers mailing list*[12] suggest. Also, the floating-point emulation for the *i386* architecture has been removed in 2004 in the *FreeBSD* version *5.2*.

To avoid using the *floating point* in our code in case of the *4BSD* scheduler, we use a table named `ccpu_exp` that is filled with the precomputed fixed-point arithmetic integer values of the following formula:

$$FSCALE \cdot e^{-\frac{k}{20}} \quad k \in 0..n \tag{6.4}$$

The *stop-and-run* technique and malicious users creating short-lived processes to avoid being noticed

The *FreeBSD racct/rctl* infrastructure is not only meant to limit the usage of various resources. It also provides facilities to *monitor* their usage. In our case of monitoring for example the *per-user CPU percentage* value and facilitating the *stop-and-run* technique, one important problem arises: Processes with short lifetime may not be noticed, because of the discrete and periodic sampling of the *CPU resource usage* done in the *racctd* thread. And this may happen even if they use substantial CPU resources in their short lifespan. The process may not be noticed if its life span is shorter than our sampling period, which is 1 second in case of the *racctd*. A malicious user could thus use substantial CPU resources without being noticed. He could achieve this by creating and appropriately scheduling short-lived CPU intensive processes.

To address this problem, whenever a process terminates, if it has not been inspected for its CPU usage by the *racctd* thread, its CPU usage percentage value is added to the CPU usage account of the appropriate user. This is also done for other necessary CPU usage accounts, like *per jail* and *per login class* CPU usage accounts. Now we see that our CPU usage accounting system also contains some superfluous CPU usage amounts caused by the already terminated processes. However, these excessive CPU usage amounts will be shortly reduced to zero. This is achieved by the automatic *decaying* of these superfluous CPU usage amounts.

Decaying the CPU usage percentage values

The excessive *CPU usage amounts* caused by the terminated processes are only present on *per user*, *per login class* and *per jail* CPU usage accounts. We refer to these types of CPU usage accounts as *container accounts*. The excessive CPU amounts are not present on individual per-process CPU usage accounts, because the per-process CPU usage accounts are kept only for living processes.

The *container CPU usage accounts* are subjected to the process of decaying. At the beginning of every *racctd* iteration, the CPU usage amounts in these *container accounts* are reduced by multiplying them by the `racct_decay_factor` variable that is always less than one and greater than zero. The value of this decaying factor is derived from the one-minute *system load average*. The decaying factor f is calculated using the following formula:

$$f = \frac{2 \cdot load\_average}{2 \cdot load\_average + 1} \tag{6.5}$$

The decaying factor f obtained from this formula has a handy property that multiplying the percentage value by this factor once a second will decay away about 90% of the percentage in $(5 \cdot load\_average)$ seconds. Another important advantage of this formula is that it does not involve any floating point functions and can be easily computed using the fixed point arithmetic in the kernel. This very same formula is also used in the code of the *4BSD* scheduler that also employs the CPU usage decaying. Now, we will sketch the rationale behind this simple formula. The formula is also explained in the comments of the source code of the *4BSD FreeBSD* scheduler in the file `sys/kern/sched_4bsd.c`.

We are looking for a decaying factor f that will decay away 90% of the percentage value p in $5l$ multiplying iterations, where l is the one-minute *system load average*. Mathematically, this is expressed in the following equation:

$$p \cdot \frac{1}{10} = p \cdot f^{5l}, \quad p > 0, \quad f \in (0, 1), \quad l > 0 \tag{6.6}$$

We assume that the percentage value $p \neq 0$ because the zero-valued percentage does not need any decaying.

$$\frac{1}{10} = f^{5l} \tag{6.7}$$

From the equation 6.7 we can get the formula for the decaying factor f:

$$f = e^{\frac{\ln(0.1)}{5l}} \tag{6.8}$$

We will approximate the expression $\ln(0.1)$ as -2.30. We also assume that l is not close to zero, more specifically that $2l > 1$. This assumption is quite realistic considering that the *CPU throttling* is usually applied under high system loads. Now we can also use the fact that $\lim_{x \to 0} \exp(x) = 1 + x$. This is what we get:

$$f = e^{\frac{\ln(0.1)}{5l}} \approx e^{-\frac{1}{\frac{5}{2.3}l}} \approx e^{-\frac{1}{2l}} \approx 1 - \frac{1}{2l} = \frac{2l-1}{2l} \approx \frac{2l}{2l+1} \tag{6.9}$$

At the end of the last equation 6.9 we have the formula 6.5, that is used in our implementation.

An illustrated example of the decaying of the CPU usage percentage values

We illustrate how the CPU usage decaying works in the figures 6.1 and 6.2. The example user *John* was running three different processes *p1*, *p2* and *p3* that were occupying 30%, 15% and 24% of the CPU resources respectively. The process *p1* was running only for a short time and thus has not been inspected in the latest *racctd* iteration. Therefore, it also does not contribute to the per-user CPU usage amount of the user *John*. The processes *p2* and *p3* have been running for a longer time and therefore their CPU usage is also mirrored in the *John's* per-user CPU usage account. Now, the process *p1* has just terminated, so its CPU percentage value has been added to the *John's* per-user CPU usage account. The green bars in the figure 6.1 show the per-user CPU usage account of the user *John*. In the beginning of the next *racctd* iteration, the John's CPU usage account will decay. The red bars in the figure 6.1 show his account after the *decaying*.

In the figure 6.2 we present the situation where the current *racctd* iteration is finished. The *racctd* iteration continued by inspecting all the running processes. Because the processes *p2* and *p3* are still running, they were also inspected and their amounts in the John's CPU usage account have been refreshed to their up-to-date values. The percentage amount of process *p1* has not been refreshed because the process does not exist any more.

You can see how the excessive amount of the percentages caused by the terminated process *p1* diminishes every *racctd* iteration from the respective container accounts.

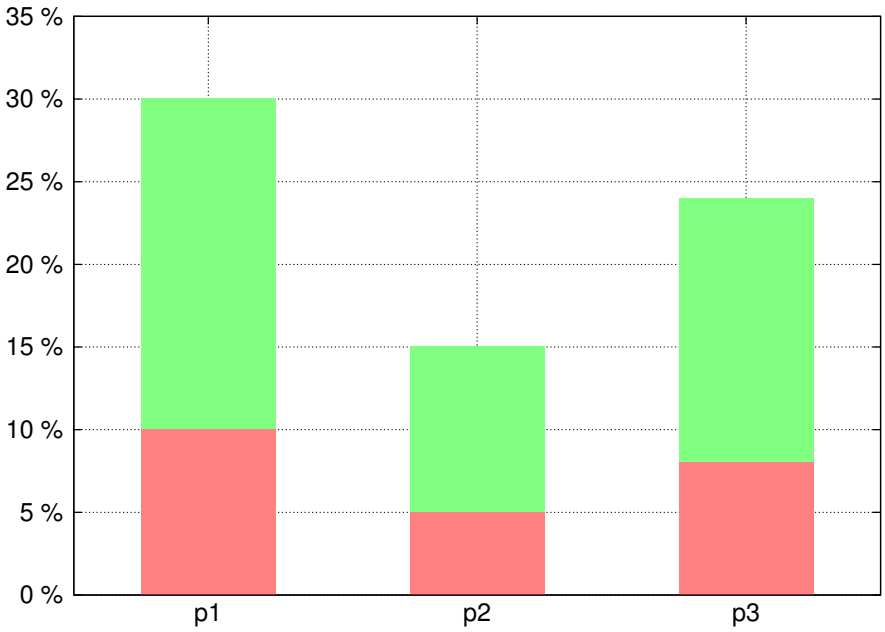


Figure 6.1: The CPU usage account of user *John* before and after the *decaying*

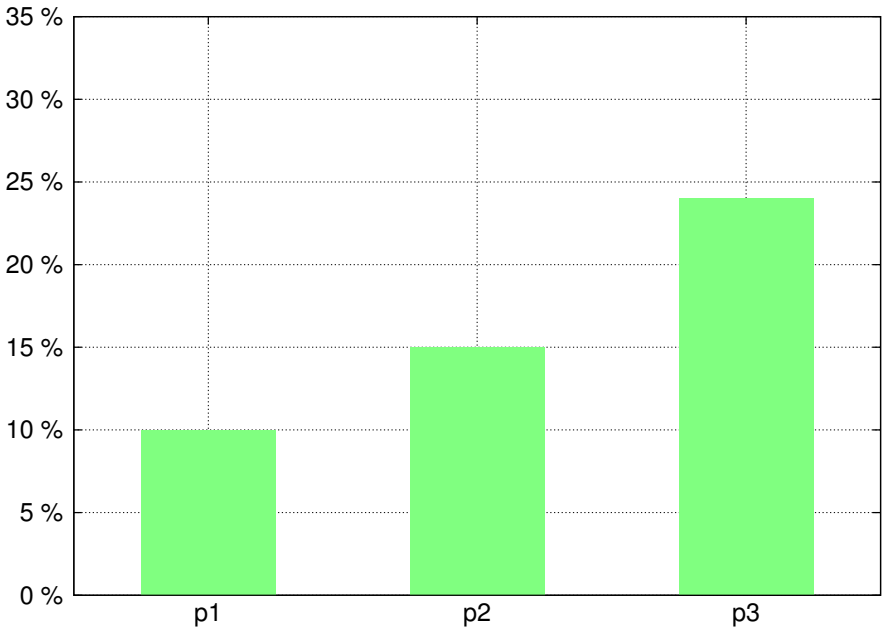


Figure 6.2: The CPU usage account of user *John* after the *racctd* iteration has finished

A note about the rounding errors in the percentage calculations

There are basically two sources of rounding errors in the percentage calculations of our prototype implementation. The first source of problems would be caused by the inherent nature of percentages - there are only one hundred of them. Storing the percentage values directly in the percentage units would be insufficient. We therefore use a nice feature of the *racct/rctl* framework and register the cpu usage resource within the framework to work internally in multiples of millions. This gives us bigger precision and less rounding errors.

The other source of rounding errors is caused by the fixed point arithmetic. Our implementation uses the precision of 11 bits right of the fixed binary point. This is de-facto a standard in the *FreeBSD* kernel fixed-point arithmetic and many other kernel parts use the same scheme.

6.2.4 Support for *relative CPU limits* specified per *process groups*

Our prototype implementation does not support only per-process CPU limits. It also supports CPU limits specified *per user*, *per login class* and *per jail*. Once we have implemented the per-process CPU usage accounting, the *racct/rctl infrastructure* takes care of the *per user*, *per login class* and *per jail* CPU usage accounting. This is done automatically as long as we use the right *API* to control the *per process* CPU usage accounting.

The *racct/rctl infrastructure* also provides us with the *API* to check if a specified process's resource usage is in line with all the limits that are applicable to the process, be them *per-user*, *per-process*, or any other limits supported by the framework. It is the *racct/rctl infrastructure's* responsibility to keep track of and check all the *rctl rules*[1] that apply to a specific process. We need not to be concerned about this - as long as the right *API* is used.

6.2.5 Simple evaluation of our prototype implementation

To evaluate the functionality of our prototype implementation to *limit the relative CPU usage*, we have created a shell script that compiles the *FreeBSD base system*. The script is supposed to be running under a testing user. Using our prototype facility, we have set the per-user relative CPU usage limit and we periodically observed the current CPU usage of the testing user. The following data displayed in the figure 6.3 was obtained when the per-user limit was set to 60%.

When using the *4BSD* scheduler during the benchmarking process, the CPU usage percentage can even exceed 100%. This is not a bug caused by our code, but a documented standard behaviour in *FreeBSD*. See the `man(1)` page for the *ps* tool in *FreeBSD* for more details. Since the defensive nature of the employed algorithms, we see that the limits are sometimes exceeded. But most of the time, they have been kept.

6.3 The block IO limits

6.3.1 Implementation requirements

We impose the following requirements on our prototype implementation:

1. Our tool will be fully integrated within the *FreeBSD racct/rctl* framework that is designed for specifying and imposing resource usage limits in the *FreeBSD* operating system.

[1] The resource limits in the *racct/rctl* framework are specified using the *rctl rules*. See the appropriate appendix section for more details about the syntax and some examples of these rules.

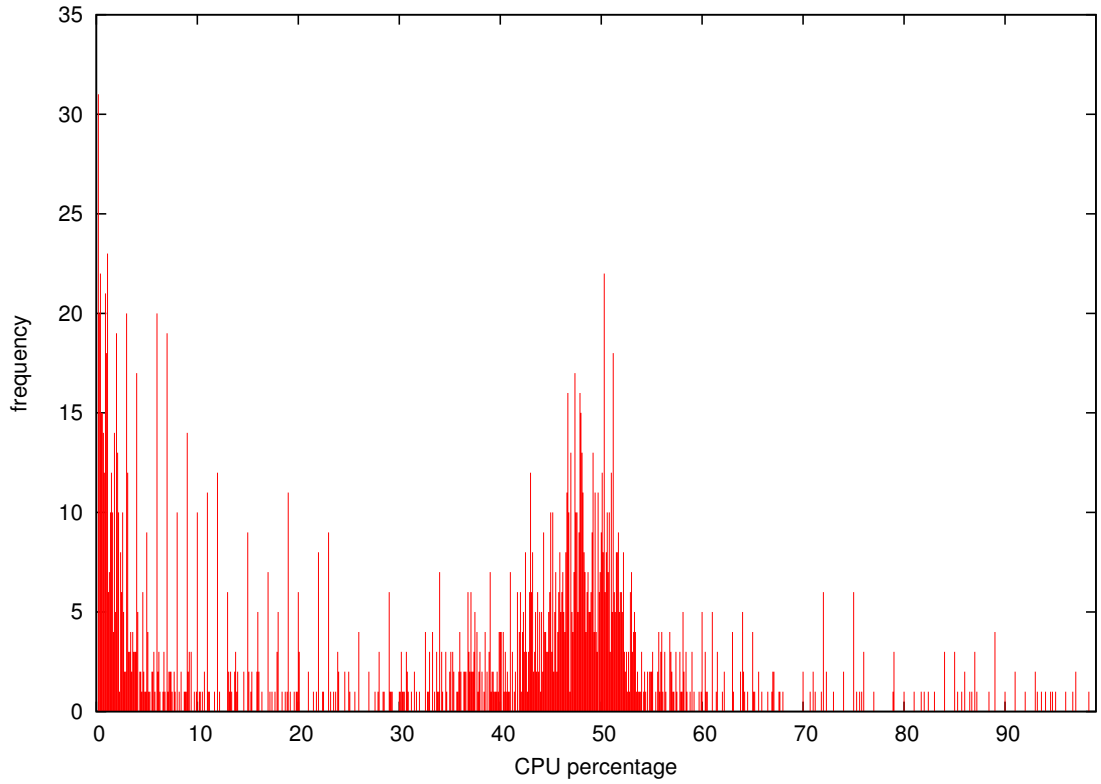


Figure 6.3: Periodically observed per-user CPU utilization during the compilation of the base system with the per-user CPU limits set to 60%. The *4BSD* scheduler is in use.

2. We will support specifying the bandwidth limits for the block devices in both the *read* and the *write* direction.

3. We will support applying the limits per-process, per-user and also per-jail.

4. The limits will only affect the data transfer originating from or directed to the block device. Read and write operations that take place only in the cached memories managed by the operating system are not affected.

6.3.2 Integrating the block IO limits within the *FreeBSD* kernel

The block device IO bandwidth limits in our prototype implementation are also implemented as an extension to the *racct/rctl* framework in the FreeBSD kernel. This framework is implemented in files `src/sys/kern/kern_racct.c` and `src/sys/kern/kern_rctl.c`. Only small changes are applied to the *racct/rctl* resource limiting framework to register names for the virtual resources that serve as identifiers for specifying the *read* and *write* block device IO limits.

Most of the work is done however on top of the *GEOM layer* that receives the block device access IO command descriptors from the upper kernel layers and processes them further. See the figure 5.1 to see the overview of the *FreeBSD* kernel block device IO subsystem architecture. The entry point to the *GEOM layer* from the filesystem layers in the *FreeBSD* kernel is the `g_vfs_strategy()` function in

the `src/sys/geom/geom_vfs.c` file. This function simply creates a *GEOM layer* command descriptor for the requested action, fills its fields accordingly and calls the `g_io_request()` function to process the newly created command descriptor. And this is the place where we hook our code.

Instead of simply calling the `g_io_request()` function to process the command, we use the *racct/rctl API* to check if the requested operation would exceed any IO limits. If the action does not violate any limits, the call to the `g_io_request()` function is made to process the IO command. However, should the requested action exceed the block device IO bandwidth limits set by the computer administrator, we store the command descriptor into a *red-black tree*-based data structure. The command descriptors in this tree are indexed by the time estimate value that says when the action should be processed to satisfy the limits. The *FreeBSD callout* timer framework is used to ensure the timely processing of the delayed command descriptors registered in the tree.

6.3.3 The *token bucket* algorithm

We have already described the variation of the *token bucket* algorithm that we use in our prototype implementation in the section 5.5.2 in the previous chapter. The whole checking if the limits are exceeded or not is done inside the *racct/rctl* framework that defines a specific *API* for this purpose. This process is analogical to our description of the *token bucket* algorithm.

We use the *racctd* thread to fill the *buckets* with the necessary *tokens*. The *racctd* thread works in a loop, each iteration taking place once a second. For every process, the *racct/rctl* infrastructure stores the numbers of bytes *read* and *written* in the last second. On every *racctd* iteration, these numbers are set to zero. This is analogical to filling the buckets with tokens. The number of tokens in the bucket can be obtained by subtracting the number of processed bytes in the last second from the overall *read/write* limit. When the command descriptors are processed in the *GEOM* layer, the appropriate number of processed bytes in the last second is increased in the *racct/rctl framework*. This is analogical to the process of consuming the available tokens.

To find out if the block IO access limit for the process p has been exceeded, the *racct/rctl* framework checks all the limits applicable to the process p. This may include limits defined per-process, but also per-user and per-jail limits. It is analogical to checking if there are enough tokens in the bucket for the operation to be performed. This is true because in our case the tokens are analogical to the yet unconsumed bandwidth.

6.3.4 Implementation details

The block device IO command descriptors that are stored in the red-black tree data structure are specified by the following structure definition:

```
struct g_sched_bio {
        struct bio *bip;
        struct g_consumer *cp;
        struct proc *p;
        STAILQ_ENTRY(g_sched_bio) gsb_link;
```

```
};
```

The `struct bio *bip` pointer is a link to the actual *GEOM-layer* command descriptor and the `struct g_consumer *cp` is a pointer to the *GEOM* consumer for the IO command. The process that caused the command is stored in the *p* variable. We need to store this pointer because we have to perform the per-process IO bandwidth usage accounting using the *racct/rctl API*. Finally, the `STAILQ_ENTRY(g_sched_bio) gsb_link` member variable enables us to keep the `g_sched_bio` objects in a FIFO queue.

In fact, the `g_sched_bio` objects are not kept directly in our red-black tree. We keep the lists of the `g_sched_bio` objects in the tree. That is because objects in the tree are indexed by their estimated dispatch time and there are often more than one command descriptors with the same dispatch time. Thus, the lists that are directly inserted into the red-black tree are defined by the following structure definition:

```
struct g_sched_bio_list {
        int dispatch;
        RB_ENTRY(g_sched_bio_list) gsb_link;
        STAILQ_HEAD(, g_sched_bio) bio_list;
};
```

The `int dispatch` member variable is self-explaining. It is used as the indexing key in the red-black tree. The `RB_ENTRY(g_sched_bio_list) gsb_link` member variable enables us to store the `g_sched_bio_list` objects in the red-black tree. And the `STAILQ_HEAD(, g_sched_bio) bio_list` member variable contains pointer to the actual queue head. The queue contains objects of the `struct g_sched_bio` type.

6.3.5 Evaluating the block device IO bandwidth limits

Firstly, we take a look at the requirements that we have imposed on our prototype block device IO bandwidth limits implementation. The implementation does integrate within the *FreeBSD racct/rctl resource limiting framework*. Also, both *read* and *write* operations are supported. Thanks to the *racct/rctl framework*, *per-user*, *per-jail* and any other container-based limits supported by the framework are also supported. Again, we only had to do the *per-process* bandwidth accounting and the rest has been taken care of by the *framework*. Because the limits are applied at the top of the *GEOM* layer, the *cached* read and write filesystem operations taking place only in the main memory are not affected. Now we can conclude that we have accomplished our goals set forth in the implementation requirements.

To evaluate our implementation, we use the *dd* utility to copy files on the disk devices. This utility conveniently shows the number of bytes processed per second.

We have set the bandwidth limit for the *read* block device access to 1MB per second. You can inspect in the figure 6.4 the speeds of the *read* operations as provided by the *dd* utility. The mean value μ for the bandwidth of the read operations is 1089217.4 bytes per second, which is less than 4% above the limit.

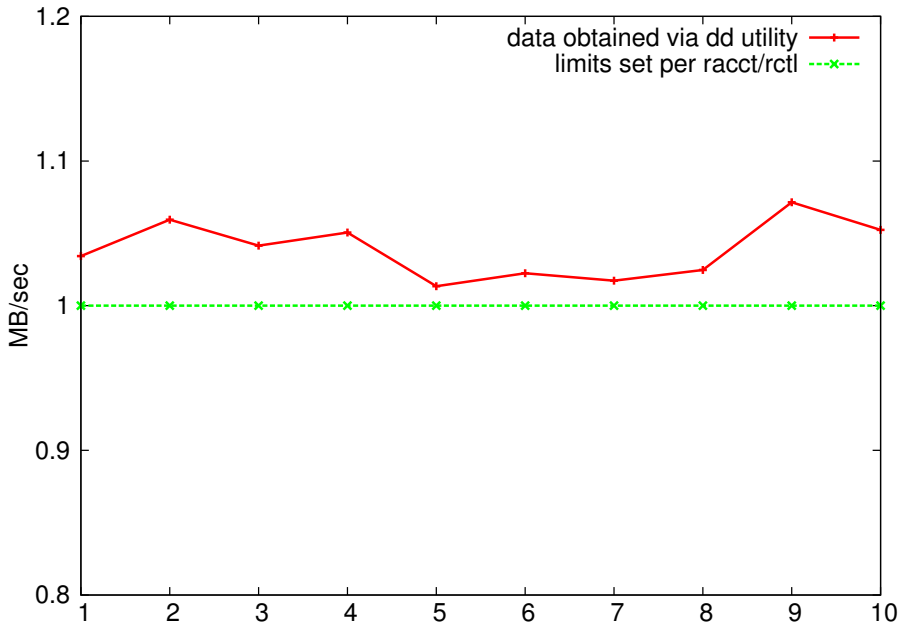


Figure 6.4: Measuring the *read* block device access using 16MB files.

In the figure 6.5, we have set the bandwidth limit for the *write* block device operations to 1MB per second. You can inspect the speeds of the *write* operations as provided by the *dd* utility.

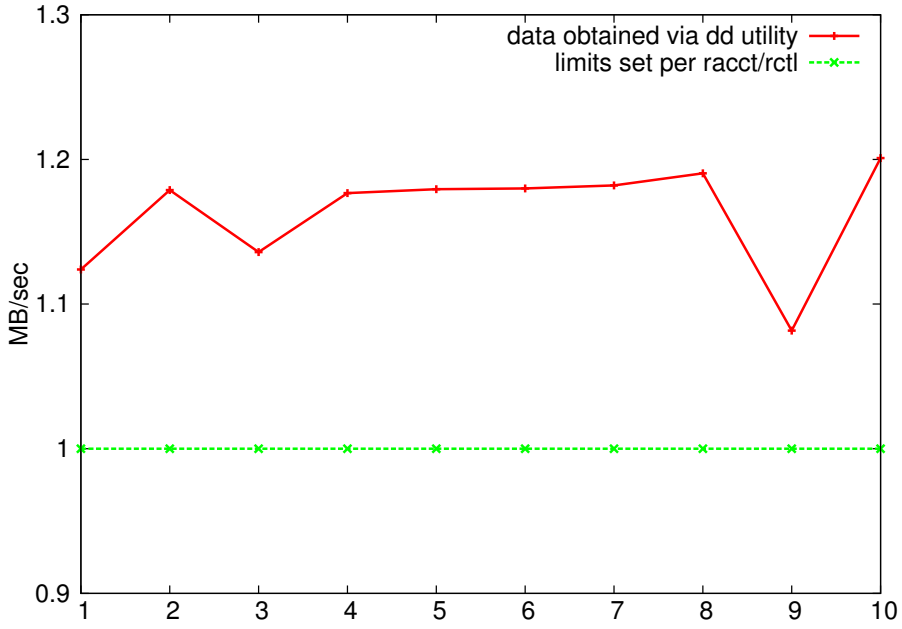


Figure 6.5: Measuring the *write* block device access using 16MB files.

The mean value μ for the bandwidth of the write operations is 1219458.6 bytes per second, which is 16.3% above the limit.

The error in the latter case of the *write* operations is much bigger than in the previous case of the *read* operations where it was less than 4%. The reason for this is simple: The *dd* utility counts the data as written sooner than the data are physically committed to the block disk device.

Conclusion

We start this thesis by examining the *Linux cgroups*. Analysing the kernel implementation of the *Linux cgroups* management and later analysing the implementation of the *cgroup subsystems* responsible for the *CPU throttling* and the *block IO bandwidth throttling* was quite time-consuming due to the complexity of the whole system.

In the next part of this thesis, the *Analysis* chapter, we try to look at the core problems of this thesis from different points of view. We also look at algorithms that are better known in other fields of computer science and are not directly related to the theory of *Operating systems*. We evaluate the pros and cons of the application of the *token bucket* and *leaky bucket* algorithms to our problems. These algorithms are usually connected with traffic shaping models of computer networks.

Probably the most challenging part of this thesis is the last part, our own implementation of the *CPU throttling* and the *block IO bandwidth throttling* for the *FreeBSD* operating system. We have decided not to port the *Linux cgroups* implementation into the *FreeBSD*. One reason for this was the complexity of the *Linux cgroups* framework, the other reason was the *racct/rctl* framework already present in the *FreeBSD* that is used for a similar purpose and that was just asking to be extended to support also the *CPU throttling* and the *block IO bandwidth* throttling.

We consider our prototype implementation stable and working. Thus, we can conclude we have successfully achieved all the goals of this thesis. Some benchmarking results are provided in the appendix section *Benchmarks*. A part of our implementation (the *CPU throttling*) has been already accepted to the *FreeBSD-CURRENT* source tree. That makes us particularly proud.

Our implementation can be easily tested without the need of compiling the whole *FreeBSD* system. We attach to this thesis a DVD that contains a *QEMU* disk image of an already preinstalled *FreeBSD-CURRENT* system with our patches already applied. More details about the DVD can be found in the appropriate appendix section. However, if you prefer not to use the prebuilt environment but to apply our patches yourself and then manually compile the *FreeBSD-CURRENT* operating system from sources, please refer to the appendix section *Building the prototype implementation*.

Since the CPU throttling patch has been already accepted to the *FreeBSD-CURRENT* source tree, our future work on the topics explored in this thesis will be based on the opinion of the *FreeBSD* community to our block IO bandwidth management patch. The *CPU throttling* implementation could be also later extended to support the soft CPU limits.

Bibliography

[1] Corbet, Jonathan. Process containers. *LWN.net.* Linux info from the source [online]. May 29 2007 [viewed 29 Oct 2012]. Available from: http://lwn.net/Articles/236038/

[2] Menage Paul. *CGROUPS.* [online]. [viewed 20 Oct 2012]. Available from: http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

[3] The Linux Kernel Organization. Block IO Controller. *The Linux Kernel Archives.* [online]. [viewed 20 Oct 2012]. Available from: http://www.kernel.org/doc/Documentation/cgroups/blkio-controller.txt

[4] Prpič Martin, Landmann Rüdiger, Silas Douglas. *Managing system resources on Red Hat Enterprise Linux 6.* [online]. 2011 [viewed 20 Oct 2012]. 3th ed. Available from: https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/index.html

[5] Corbet Jonathan. Schedulers: the plot thickens. *LWN.net.* Linux info from the source [online]. April 17 2007 [viewed 29 Oct 2012]. Available from: http://lwn.net/Articles/230574/

[6] Bovet P. Daniel, Cesati Marco. *Understanding the Linux kernel.* 3rd ed. California:O'Reilly, 2005. 944 p. ISBN-13: 978-0596005658

[7] Corbet Jonathan. Trees I: Radix trees. *LWN.net.* Linux info from the source [online]. March 13 2006 [viewed 29 Oct 2012]. Available from: http://lwn.net/Articles/175432/

[8] Bharata B Rao. *CPU hard limits.* [online]. 2009 [viewed 10 Nov 2012]. Available from: http://lwn.net/Articles/336127/

[9] Roberson Jeff. *ULE: A Modern Scheduler For FreeBSD.* [online]. Sep 2003 [viewed 7 Dec 2012]. Available from: http://www.usenix.org/event/bsdcon03/tech/full_papers/roberson/roberson.pdf

[10] Kirk McKusic Marshall, Neville-Neil V. George. *The Design and Implementation of the FreeBSD Operating System.* Boston:Addison-Wesley, 2004. 720 p. ISBN-13: 978-0201702453

[11] Tanenbaum S. Andrew. *Computer Networks.* 4th ed. New Jersey:Prentice Hall, 2002. 912 p. ISBN-13: 978-0130661029

[12] *The FreeBSD hackers mailing list.* [online]. [viewed Nov 25 2012]. Available from: http://lists.freebsd.org/pipermail/freebsd-hackers/

[13] The FreeBSD Documentation Project. *FreeBSD Handbook.* [online]. [viewed Dec 15 2012]. Available from: http://www.freebsd.org/doc/en/books/handbook/index.html

[14] The FreeBSD Documentation Project. *FreeBSD Architecture Handbook.* [online]. [viewed Dec 20 2012]. Available from: `http://www.freebsd.org/doc/en/books/arch-handbook/`

[15] The FreeBSD Documentation Project. *FreeBSD Developers' Handbook.* [online]. [viewed Dec 20 2012]. Available from: `http://www.freebsd.org/doc/en/books/developers-handbook/`

[16] The Linux Kernel Organization. *Linux kernel sources.* [online]. Version *3.6.8.* [viewed Dec 2 2012]. Available from: `https://www.kernel.org/`

[17] The FreeBSD Project. *FreeBSD sources.* [online]. Version *FreeBSD-CURRENT.* [viewed Dec 15 2012]. Available from: `http://www.freebsd.org`

Appendix A

Usage and examples

The rules syntax

The *CPU limits* and the *block IO bandwidth limits* are specified using the `rctl(8)` command in *FreeBSD*. This command is used for the management of the *rctl rules* loaded into the kernel. The *rctl rules* specify the limits for various resources managed by the kernel. We provide here an excerpt from the `rctl(8)` man page that specifies the syntax of the *rctl rules*:

> Syntax for a rule is: `subject:subject-id:resource:action=amount/per`.
>
> **Subject** defines the kind of entity the rule applies to. It can be either *process*, *user*, *login class*, or *jail*.
>
> **Subject ID** identifies the subject. It can be user name, numerical user ID, login class name, or jail name.
>
> **Resource** identifies the resource the rule controls.
>
> **Action** defines what will happen when a process exceeds the allowed amount.
>
> **Amount** defines how much of the resource a process can use before the defined action triggers.
>
> The **per** field defines what entity the amount gets accounted for. For example, rule `loginclass:users:vmem:deny=100M/process` means that each process of any user belonging to login class "users" may allocate up to 100MB of virtual memory.
>
> Rule `loginclass:users:vmem:deny=100M/user` would mean that for each user belonging to the login class "users", the sum of virtual memory allocated by all the processes of that user will not exceed 100MB.
>
> Rule `loginclass:users:vmem:deny=100M/loginclass` would mean that the sum of virtual memory allocated by all processes of all users belonging to that login class will not exceed 100MB.
>
> Valid rule has all those fields specified, except for the *per*, which defaults to the value of subject.

The rctl resource used for the *CPU throttling* is identified by `pcpu`. The resource responsible for the *block IO bandwidth throttling* in the *read* direction is `ior`. The *write* direction is identified by the rctl resource `iow`. There is only one valid *action* for these resources, and that is the `deny` action.

Examples

The *rctl rules* can be managed only by the root user. The following command loads a rule into the kernel that sets the 50% limit to the overall CPU access of processes owned by user *rctl*:

```
rctl -a user:rctl:pcpu:deny=50
```

The following command loads a rule into the kernel that sets the 1MB per second limit for the block IO device read access for processes owned by user *rctl*:

```
rctl -a user:rctl:ior:deny=1M
```

The `M` prefix stands for megabytes. The `k` prefix stands for kilobytes. If no prefix is provided, the default unit is in bytes. The following command loads a rule into the kernel that sets the 1024kB per second limit for the block IO device write access for processes owned by user *rctl*:

```
rctl -a user:rctl:iow:deny=1024k
```

The following command removes the last rule:

```
rctl -r user:rctl:iow:deny=1M
```

When a rule is being removed, you do not always need to specify the whole rule. Sometimes, *filters* can be used that are shorter to type and can comprise more than one rule. The following command removes all the rules specified for the user *rctl*:

```
rctl -r user:rctl::
```

To view all the rules currently loaded in the kernel, the root user can run the `rctl` command without providing any parameters.

Appendix B

Building the prototype implementation

Tutorial

Use the following steps to build a *FreeBSD-CURRENT* machine patched with the patches that make up our prototype implementation. This is also the way how we built the *FreeBSD* system provided on the QEMU disk image present on the attached DVD.

1. You firstly need to build the *FreeBSD-CURRENT* operating system.

   Download and install the latest *RELEASE* version of the *FreeBSD* operating system from the *FreeBSD* homepage. At the time of this writing, it is the *FreeBSD 9.1-RELEASE* version. Now, refer to the *FreeBSD handbook* to upgrade your system to *FreeBSD-CURRENT*.

   You will do the best if you upgrade your system to *FreeBSD-CURRENT* revision number `r248247`, because this is the *subversion* revision number for which the patches are provided on the attached DVD.

2. Apply the `tomor6am.patch` file provided on the DVD

   ```
   cd /usr/src
   patch -p1 < tomor6am.patch
   ```

3. Add the following options to your kernel configuration file:

   ```
   OPTIONS RACCT
   OPTIONS RCTL
   ```

4. Build the kernel

   ```
   cd /usr/src
   make buildkernel KERNCONF=YOUR_CONFIG_FILE
   ```

   Where `YOUR_CONFIG_FILE` should be substituted with the name of your custom kernel configuration file.

5. Install your new kernel

   ```
   make installkernel KERNCONF=YOUR_CONFIG_FILE
   ```

   Again, the `YOUR_CONFIG_FILE` should be substituted with the name of your custom kernel configuration file.

6. Reboot your computer. You will boot your new kernel.

Appendix C

Benchmarks

The block IO bandwidth throttling

Here we have performed disk device *read* operations by instructing the *dd* utility to read 16MB files. The benchmark was performed using two processes running under the same user. The per-user read block IO bandwidth has been limited.

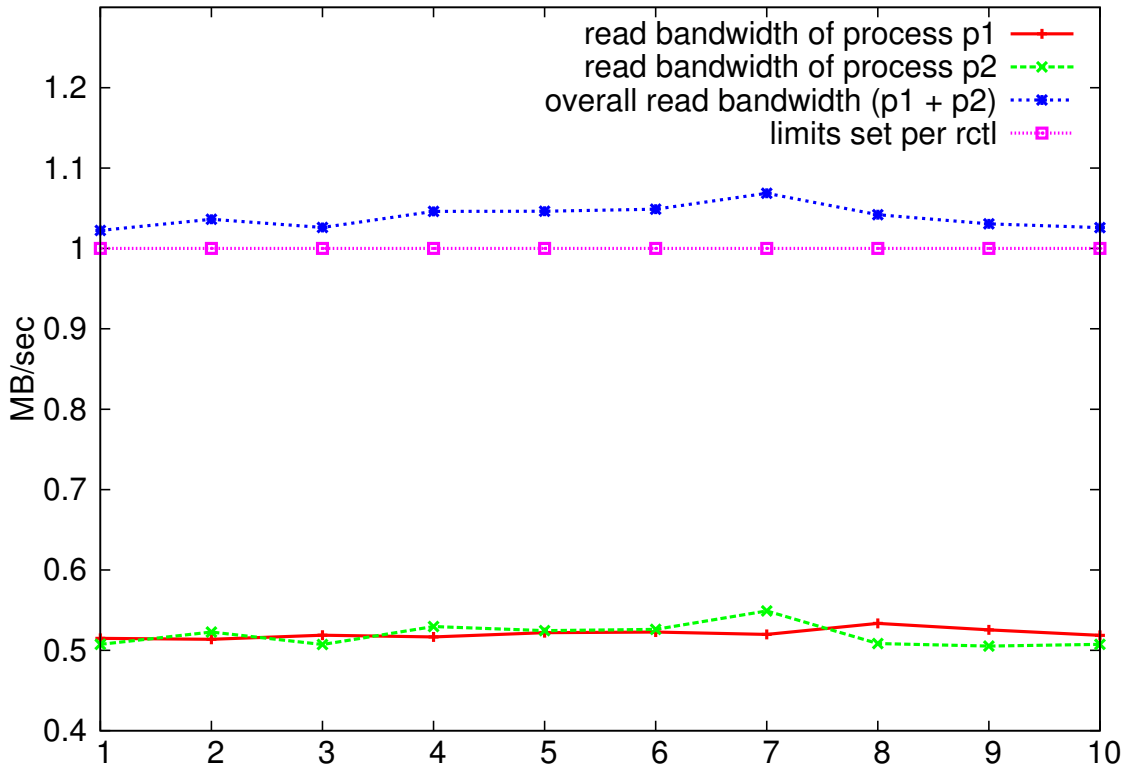


Figure 6.6: Two processes running under the *same* user have been performing *read* operations. The *per-user* read bandwidth has been limited to 1MB per second.

The CPU throttling

The following figures 6.7 and 6.8 show a situation where we have been compiling the *FreeBSD* base system under a testing user. The per-user CPU limits for the testing user have been set to 50% and 60% respectively. During the compilation process, we have periodically observed the per-user CPU usage using the *ps* utility. In both benchmarks, the *ULE* scheduler has been used.

These figures look slightly different than the figure 6.3. The reason for this is that in the previous case, the *4BSD* scheduler was employed.

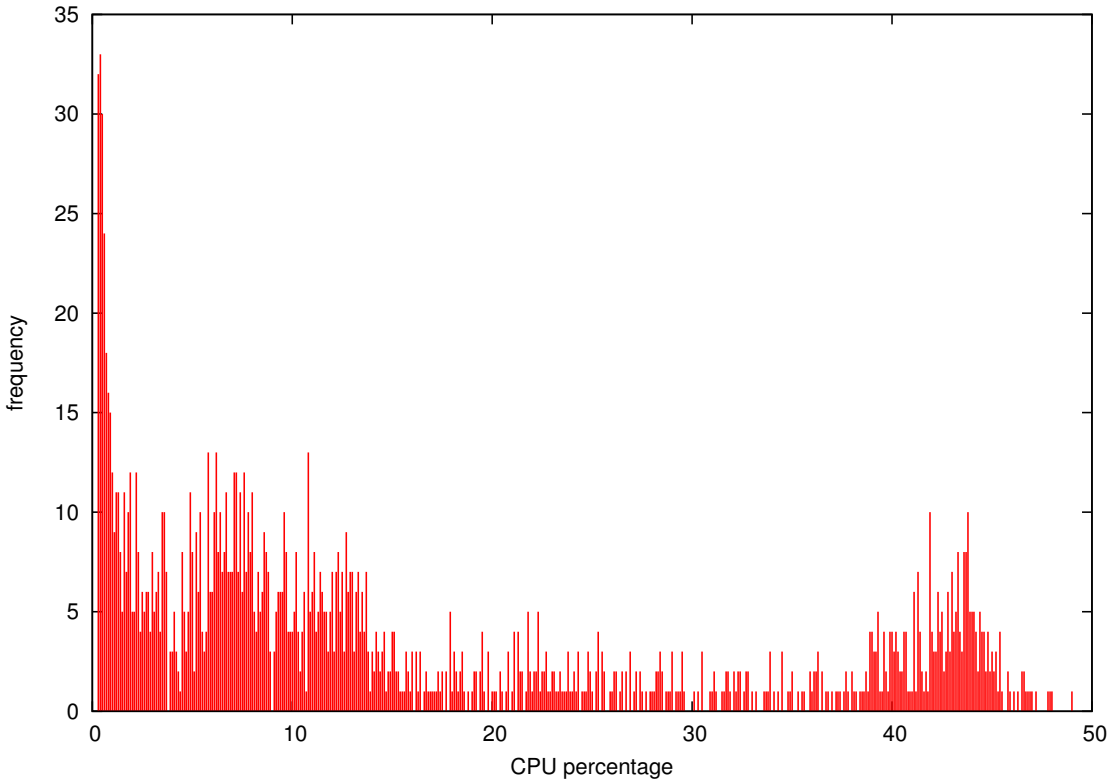


Figure 6.7: The per-user CPU limit has been set to 50%.

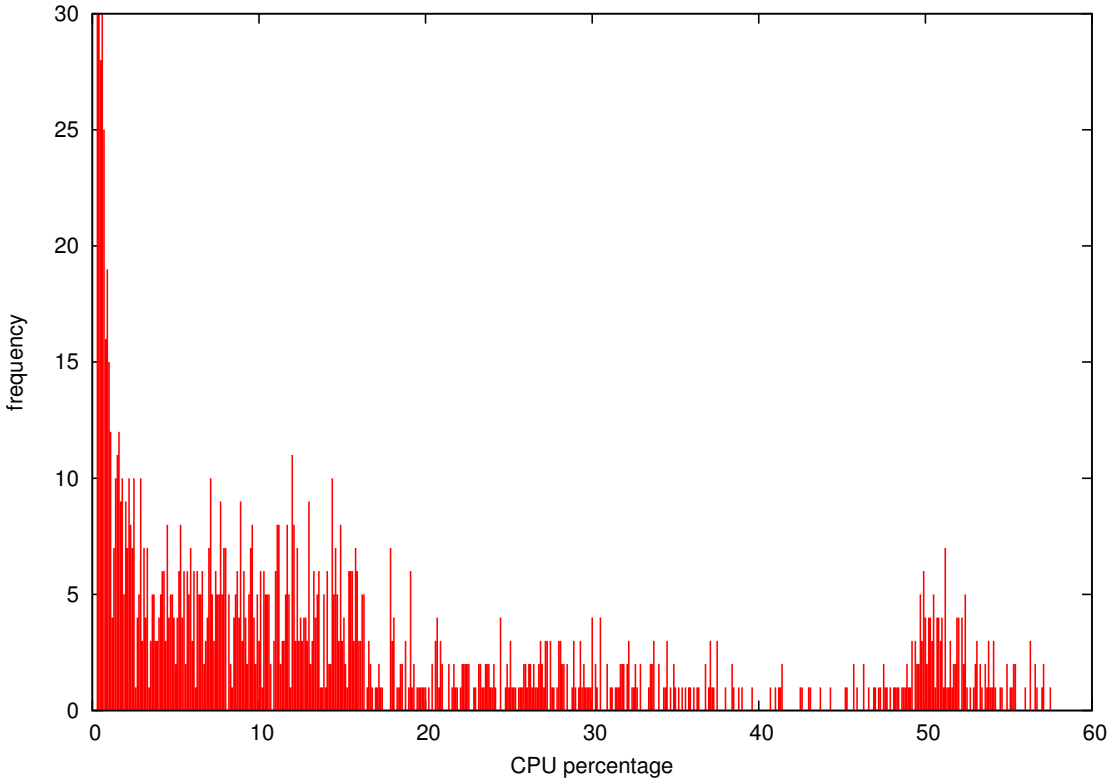


Figure 6.8: The per-user CPU limit has been set to 60%.

Appendix D

The attached DVD

The contents of the DVD

The attached DVD contains the following directories:

qemu-vm This directory contains the raw QEMU image with the preinstalled and patched *FreeBSD-CURRENT* operating system. Our prototype implementation can be easily tested using this virtual machine. Some helper scripts to run the virtual machine are also provided.

code This directory contains our prototype implementation and the relevant parts of the sources for the FreeBSD kernel.

code/patches The patches that make up our prototype implementation are included here.

code/src Parts of the FreeBSD-CURRENT kernel sources are stored here.

code/src/vanilla This directory contains files from the unpatched kernel sources for the *FreeBSD-CURRENT* operating system. They have been obtained directly from the *FreeBSD-CURRENT* svn repository.

code/src/patched This directory contains the same files as the *vanilla* directory, but after applying the patches that make up our prototype implementation.

benchmarking This directory contains some shell scripts that we have used for benchmarking.

thesis This master thesis is stored there.

More detailed descriptions of the files inside these directories can be found in the respective *readme* files.

A note about the included QEMU image on the attached DVD

Please note that you can not boot the QEMU virtual machine using the provided raw QEMU image directly from the DVD. That is because the attached DVD serves only as the *read-only* memory.

To use the image file, first copy the whole *qemu-vm* directory to your hard drive. It takes about 2GB of disk space. After that, if you have the QEMU emulator installed on your system, you should be able to run a QEMU virtual machine from the provided QEMU disk image residing on your hard drive. To boot a QEMU virtual machine using our disk image on a UNIX system, you can use the provided helper scripts in the *qemu-vm* directory.

A tutorial to test our implementation using the QEMU disk image provided on the attached DVD.

By following the steps in this tutorial you will firstly set some *rctl limits* and then observe the behaviour of the virtual machine.

1. Boot a QEMU virtual machine using the provided raw QEMU disk image. We will later need the SSH access to the virtual machine. Therefore, qemu should be instructed to listen on some port and then forward the incoming packets via the SSH port of the virtual machine.

   The `qemu-vm/fire-vm-ssh.sh` script can be used for this step. This script instructs qemu to listen on the TCP port 33333.

2. Login as the *root* user providing the password *root*.

3. Enter the `rules-mgmt` directory which is located in the home directory of the *root* user.

4. Run the `add_cpu_limit.sh` script to set the 50% CPU usage limit for the user *rctl*. You can confirm that the limit has been successfully set by running the `view_all_limits.sh` script.

5. Login to the virtual machine via ssh as user *rctl* using password *rctl*. The following command can be used for this if the qemu is listening for SSH connections on port 33333: `ssh -p 33333 rctl@localhost`.

6. As user *rctl*, change your working directory to the `~/tests/cpu` directory.

7. Run the `top` command in the virtual machine. (Not in your SSH session, but as user *root*. You will need your SSH session for other commands.)

8. As user *rctl* in your SSH session, run the `./dummy_process` binary in your working directory. You should notice in the output of the `top` command that the *dummy_process* does not consume more than 50% CPU time.

9. SSH with a new session to the virtual machine as user *rctl*. Again, run the `dummy_process` binary. You should notice in the output of the `top` command that the two processes do not consume more than 50% CPU time altogether.

10. Terminate the two running `dummy_process` binaries. As user *root*, run the `remove_all_limits.sh` script to remove the previous pcpu *rctl rule*. Now, run the `add_write_limit.sh` script to set the limit for the block IO bandwidth in the write direction to the 1MB per second. Again, check with the `view_all_limits.sh` script that only the last limit is set now. The output should read: `user:rctl:iow:deny=1048576`.

    It is important that the previous *rctl rule* responsible for limiting the CPU usage of the user *rctl* is no longer active. Otherwise, the results of the tests performing the block IO read and write operations and measuring the overall bandwidth would be biased.

11. Now, you will need to run two scripts almost simultaneously. You should now have two active SSH sessions as user *rctl* to your virtual machine. In the first session, type this command (without typing the enter key to submit the command): `./write_test.sh 1`. In the second session, prepare this command: `./write_test.sh 2`. The `write_test.sh` script is located in the `~/tests/write` directory. Now, type the enter key in both sessions to submit the commands almost simultaneously. Wait until the commands end. You should notice in the output of these commands that both of them were processing about half of a megabyte per second. This adds up to our 1 megabyte limit.

12. As the *root* user, run the `add_read_limit.sh` script to set a bandwidth limit for the block IO read operations to 1MB per second. Now, it is not necessary to remove the previous rule for the write operations. If you run the `view_all_limits.sh` command, you should see the following rule has been set: `user:rctl:ior:deny=1048576`

13. Now, in your two SSH sessions, change your working directory to the `~/tests/read` directory. In your first session, run the script `read_test_1.sh`. In the second session, run the `read_test_2.sh` script. Again, these scripts should be run almost simultaneously. You should notice in the output of these scripts that they have been again processing about half of a megabyte per second. This adds up to our 1MB limit.