Accounting and control of power consumption in energy-aware operating systems

Diploma Thesis

by
Martin Waitz
born April 25th, 1976, in Coburg

Department of Computer Science,
Distributed Systems and Operating Systems,
University of Erlangen-Nürnberg

Advisors:	Dr. Ing. Frank Bellosa
	Dipl.-Inf. Andreas Weißel
	Prof. Dr. Wolfgang Schröder-Preikschat
	Prof. Dr. Fridolin Hofmann

Begin:	July 31st, 2002
Submission:	January 31st, 2003

Abstract

An important task of operating systems is to schedule shared resources fairly between several parties. Precise accounting of consumed resources is the key to that goal. However, most operating systems only use very basic accounting strategies.

This thesis discusses methods for resource accounting and introduces a powerful, yet easy to use accounting model based on resource containers. The goal of this model is to always charge the party that is responsible for some resource usage. To achieve this, client-server relationships between running processes are detected. They provide an invaluable source of information which can be used to identify the entities initiating resource intensive actions. As the resource containers which are used for accounting can be nested to form a hierarchy, sophisticated accounting and scheduling policies can be formulated.

Utilizing this new accounting model, the energy consumption of the machine can be charged to the responsible entity. Accounting energy consumption is a very natural way, as every hardware component that contributes to the execution of a program is consuming energy. Several methods used to measure or estimate the energy consumption of various hardware parts are discussed, paying special attention to the main processor.

Peak energy consumption is especially important as many components have to be dimensioned according to the maximum power consumption. Reducing this peak consumption can save costs in both high-end data centers and small mobile devices. A software method for limiting energy consumption is introduced. By using the new resource model, advanced policies can be defined that allow one to control power consumption of the entire machine, individual processes or special clients and servers.

Contents

Chapter 1

Introduction

An important task of operating systems is to schedule shared resources fairly between several parties. Precise accounting of consumed resources is the key to that goal.

1.1 Importance of energy consumption

Resources accounted in today's general purpose operating systems usually include CPU, disk and network usage. These are used as they are easy to obtain: the operating system has direct control over each of them. Another resource that is getting more and more important is a bit more difficult to handle: power consumption.

Power consumption is crucial both for low- and high-end machines. Small mobile devices can only carry small batteries and have to be very careful when spending energy, as the user demands long battery lifetimes. Bigger machines usually are not battery powered; availability of energy is not limited, but as a side-product of energy-consumption, heat is getting problematic. Today's workstations need sophisticated cooling methods to keep processors and hard disks from overheating. A lot of space is needed for all those fans and heat sinks, preventing miniaturization of computer systems. As rack space is expensive, this is a problem even for data centers – performance per rack is important. Additionally, the costs of both air conditioning and (uninterruptible) power supply heavily depend on the maximum power consumption rate, making it more important to reduce it.

Of course, the best way to save energy is to build hardware which consumes only an absolutely minimum amount of energy. Hardware is always controlled by software and in order to save energy, applications have to be efficient, too.

As power consumption depends on energy efficiency and workload, there is another possibility to reduce it: change the workload. Often a machine has more than one task to do or has tasks that do not have to run at once. By choosing when each task may use the hardware, an operating system can influence power consumption of the machine. To be able to do this, a detailed policy is needed that defines which tasks are important and how much energy a task may consume.

In this way it is not possible to reduce the overall energy which got consumed after all tasks are finished, but the rate of consumption can be *shaped*. For example, an expensive background task can be throttled to achieve the desired battery lifetime for some foreground task. Hardware may run best at special usage patterns that can be enforced by software; spikes and average consumption can be controlled by the operating system to ensure that no hardware part gets overloaded.

1.2 Energy Accounting

To be able to control energy use, accurate accounting information is necessary. Only then it is possible to detect if some entity consumed too much and has to be throttled. Per-task energy accounting is needed in order to limit the tasks individually. No direct mean to obtain this information is available and the operating system has to derive it from per-machine or per-hardware consumption. That is possible when the system keeps track of everything that uses the hardware.

This section will give an introduction into several methods suitable to obtain information about the power consumption of hardware devices.

1.2.1 Direct measurement

The best way to get data about energy consumption is of course to measure it. This however requires extra hardware, which has to be installed in the machine.

Newer laptops include a *SmartBattery* [25], which can export current drain rate and capacity to the Operating System via ACPI.[1] In most implementations, the data collected is not accurate enough and is updated only infrequently. Even when accurate values are reported does this method only work while the machine is on battery power.

Of course special purpose hardware can be built to meet all requirements for accurate power consumption measurement. But this is expensive and it is not ex-

[1] Advanced Configuration and Power management Interface [7], allows the operating system to interface with the machine's firmware.

pected that such hardware will be generally available in computers within the next few years.

1.2.2 Estimation of power consumption

When no direct means are available, the power consumption rate has to be obtained indirectly. The operating system can measure the activity of several devices in the system. Combined with an energy consumption model of these devices, it is possible to calculate an estimation of the power consumption.

Devices with a high power consumption in current systems are the CPU, memory, modern 3D graphics cards and displays, hard disks and network adapters.

The easiest way to model power consumption of the main processor is to count how long it stays in its different performance states (active, idle, low-power halt, ...). More information about what is going on inside the processor can be obtained by reading performance counters or by examining the code which is executed. Performance counters [13] can count several events inside the processor and can be used to estimate which parts of the chip are active and consuming energy, even memory accesses can be counted this way.

Power consumption of hard disks and network adapters can be modeled easier as the operating system directly controls them and knows what operations are executed. Graphics cards on the other hand are handled by the display server and not by the operating system kernel. However, even with some knowledge about graphic operations, consumption of modern graphics boards would be very hard to predict as they include their own processors with undocumented firmware.

1.3 Controlling resource usage

Once the operating system has detailed knowledge about the consumption of resources, it can start to control them.

Resources like CPU time or IO bandwidth are relatively easy to handle, as each of them is controlled by a single scheduler inside the operating system. This is not true for energy consumption, which can be influenced by almost every device driver and scheduler in the system. All of them have to cooperate in order to keep some energy policy.

Schedulers have to use and update data structures that represent the total energy consumption of resource principals like processes or services. Only with such information is it possible to hold usage limits.

The main objective of this thesis is to build a resource model that allows to define sophisticated policies on energy consumption without requiring expensive changes to existing POSIX compatible applications.

Chapter 2

Related Work

Resource accounting and scheduling is a very active topic in today's computer science. This chapter will present related work dealing with energy accounting or resource accounting in general.

Resource containers are handled in more depth, as this work heavily builds upon them. Problems that were encountered while adopting resource containers for energy accounting are shown, together with possible solutions. The resulting resource model itself will be described in detail in the next chapter.

2.1 Accounting

A common way to estimate power consumption of hardware is to check device states. A constant drain rate is assumed for each state and each state transition contributes a fixed amount of energy. This approach is used to estimate energy consumption of PalmTop machines [6] and workstations [19] using state information gathered by ACPI [7]. In [19], a calibration process systematically places devices into each supported state while observing power consumption via the Smart Battery Interface [25].

The use of performance counters [13] to obtain energy consumption is studied in [4]. The work in progress [16] implements that method to estimate power consumption and temperature of the main processor.

2.2 Handling resources

Resource containers were introduced by [2] and will be discussed in detail in the next section. Cluster Reserves [1] are used to control resources in a cluster by

tuning Resource Containers on each machine. Another hierarchical resource model is described in [14], and hierarchical schedulers are studied thoroughly in [12, 22].

SGI implemented a simpler scheme to group processes called process aggregates [28], which is used by their Comprehensive System Accounting [17] for accounting.

Nemesis is a vertically structured operating system that uses libraries in such a way that the vast majority of functionality comprising an application could execute in a single process [21]. This way, energy can be accounted per process [19], without having to special-case kernel processing or user-space servers which are used in other systems.

Information about processes can also be used to automatically reconfigure the hardware to save energy with minimal impact on performance. Processor clock rate and voltage can be adjusted to match the speed of memory transfers [29, 10].

The MilliWatt project built ECOsystem which is using energy as its main resource primitive which is called Currentcy [30, 31, 27].

The effect of energy consumption on operating systems is also discussed in [3].

2.3 Resource containers

To be able to account and control resource usage (such as power consumption, processor time, memory and I/O bandwith), a data structure is needed that holds information about available resources and its consumption. Banga [2] introduced resource containers, which are such a data structure. Even though resource containers are handled by the operating system kernel, there exists a user-space Application Programming Interface (API) for applications that want to alter their resource configuration.

Concepts necessary to understand resource containers are described in this section, followed by a discussion on their shortcomings.

2.3.1 Decoupling resource accounting and processes

In order to ensure fair resource usage, Resources consumed by applications and services must be accounted by the operating system As per-process structures already exist, these are commonly used for accounting of the resources used by that process.

This effectively enforces that the protection and resource-allocation domain must be the same. Often services are split into multiple protection domains for

security reasons, yet they are working together and thus should share one resource domain. Other server processes serve different services which should belong in different resource-allocation domains.

Decoupling resource accounting from processes has the advantage that it also enables the operating system to correctly account for work done in the kernel, as the kernel can change the container used for accounting independently of the current process.

2.3.2 Capability Model

The right to consume some resources has to be protected. Applications should only be able to use resource containers when they are explicitly allowed to do so. This permission is represented to the application as a *capability*.

A capability identifies an object (much like pointers or references in programming languages) and at the same time holds the authority to use that object. They are managed by the operating system kernel.

In contrast to access control list (ACL) based security systems, which give access to all processes belonging to a user, capabilities allow fine-grained access to secured objects. Every application can have its own set of capabilities – even those started by the same user. Capabilities should only be given to those applications that really need them, following the principle of least privilege [24].

Most operating systems protect objects using a combined ACL/capabilities approach. Capabilities are handed out based on access control lists. Further operations on the object have to use that capability. The capability is a guaranty that the process already passed the ACL check. Thus expensive ACL tests do not have to be invoked for every access. Capabilities to files are named *file descriptors* in POSIX systems, they can be obtained by "opening" a file.

To user-space applications, resource containers are represented as file descriptors. Banga [2] creates a new file type which cannot be stored in file systems and is only used to name a resource container.

2.3.3 Hierarchy

Resource policies are needed at many different levels. Global limits are necessary to guarantee battery lifetime. Per-service or per-user policies are useful when these services or users do not have equal priorities or importance. And applications

themselves may want to set different policies for their components (e.g. for the background spell checker in a word processor).

To allow such multi-level policies, resource containers are arranged to form a hierarchy. Each individual resource container is enclosed by a parent container. Policies defined for one resource container also apply to all child resource containers. These children have to share all the resources available in the parent. The root container represents all resources available to the entire machine and can be used to enforce system-wide policies.

2.3.4 Resource container bindings

As a process may have access to several resource containers at once, it has to decide which one should be used. This is called the *resource binding* of a process; it can be dynamically updated by the application.

When a process has much different work to do, it may need to receive and consume resources from several containers. This is possible by adding the process to the *scheduling binding* of each resource container. Resources given to a resource container are distributed between all processes belonging to the scheduling binding.

2.3.5 Scheduling

The resource container implementation as proposed by Banga [2] uses resource containers as their scheduling primitives instead of processes. The scheduler chooses a resource container, which gets resources for consumption. Processes bound to this container may run until the assigned resources are used up, at which time resources are granted to a different resource container.

Choosing a resource container can be accomplished by a hierarchical scheduler. Each resource container with children corresponds to one scheduler node. When a node receives resources, it distributes those resources between the children of the corresponding resource container. Another scheduler may again apportion these resources if the receiving child is an inner node in the container hierarchy. Thus scheduling traverses the resource container hierarchy in a top-down manner. Resources assigned to one container are divided up between its children according to the limits or priorities configured in the containers.

When resources are given to a resource container that is part of the scheduling binding of a process, that process may run until the resources are used up.

2.4 API issues in present resource container systems

The work on resource containers greatly simplifies accurate accounting and controling of resource usage. However, there are some problems with existing resource container systems that make it difficult to deploy these systems in real life operating systems.

Changing every application is extremely expensive, but if resource containers are not correctly set up, there is no reason in using them at all. Therefore, a system is needed that transparently sets up correct and meaningful resource bindings without depending on the applications to do so. Of course, resource container aware programs must always be able to override their resource bindings if necessary.

In this section, basic problems are discussed which are very important for accounting in general and therefore have to be solved for an advanced energy counting system.

2.4.1 Resource containers are not real files

Although resource containers are represented as file descriptors, they are very different to normal files. They are even more special than sockets – neither read nor write is possible.

However, using file descriptors to reference resource containers has some benefits, as this is the only way capabilities can be used in POSIX systems. Even then, introducing a completely new file type is not necessary, a solution reusing normal file descriptors is shown in section 3.1.1.

2.4.2 Access control

Resource container file descriptors can have different access rights, similar to read or write access used for normal files:

User rights include the right to use resources belonging to this container.

Owner rights include the right to change parameters of the container, for example resource limits and priorities.

This approach is needed as some operations need to specify a resource container to which the application should not have full control.

One common example is an application that should be able to consume resources on behalf of a special resource container without being able to change the priority or resource limit stored in that container.

Another is creating a sibling container, which is often needed when an application wants to start independent activities. The application has to create a new resource container and change its parent. However, an application may not have full control over the parent of its own resource binding, as that would allow to influence other processes which share the same parent container. Banga solves this by passing a user-rights-only capability of the parent container to be able to call *set_rc_parent*().

However, there is a solution to these problems that does not depend on complicated access rights. The resource model that will be described in chapter 3 uses the container hierarchy to isolate authority and introduces a new way to create resource containers to achieve equivalent security properties.

2.4.3 Changing resource container assignment

An application is always responsible for keep its resource binding up to date.

This is especially important for server processes which have to process requests from many clients, as they may want to use different resource bindings for different clients. Depending on the type of server this is more or less difficult.

Classic Unix servers start a new process or thread for each incoming request. They have to configure the resource binding only on thread creation.

Most traditional Internet servers work this way, for example *inetd*.

Event-driven servers process all incoming requests in one big event loop. The Process must update the resource binding every time it starts to work on a different request.

An example of a popular event-driven server is the *XFree86* Display Server.

Multi-threaded servers are a mixture between those two. They create some threads prior to the arrival of requests. Each new request gets scheduled to an idle thread. If all threads are already processing requests, a new thread is created for the request. Like event-driven ones, these server threads have to change the resource binding whenever they start to process a different request.

Multi-threading is mainly used for servers where the additional fork is too expensive, but which need independent threads per request. For example the *apache* web server works this way.

To support resource containers, the application has to determine the resource binding it should use for the next request and tell the operating system about it.

```
socket := accept()
set_res_binding(rc_for(socket))
request := read(socket)
result := process(request)
write(socket, result)
```

Figure 2.1: Processing events

The *set_res_binding*() call in Figure 2.1 has to be inserted to make the application resource container aware. Note that prior to processing the request, the application of course has to read it. This can be exploited to remove the need to manually set the resource binding via *set_res_binding*() and implicitly set it on *read*() (shown in section 3.3).

However, resources can be consumed very early, even before *accept*() returns. For example, the Transmission Control Protocol [20] used by most Internet servers includes an initial handshake which consumes server-side resources. Expensive server-side work can be triggered very easily by sending a single SYN packet.[1] By flooding a machine with such packets, a malicious client could attack a server and overload it. There are protocols with a better handshake mechanism, for example the Stream Control Transmission Protocol [26] or application level protocols like the one photuris [15] uses. But as most applications still use TCP only, an operating system must handle SYN packets gracefully.

A Lazy Receiver Processing [9] implementation of the network stack helps to solve this problem. It uses a packet multiplexer which associates packets with their socket as early as possible. The resources consumed by processing of arriving packets can be accounted to the resource container associated to that socket. When there are no resources left in the resource container, the packet cannot be processed by the kernel and gets dropped.

To be able to distinguish between different clients – each with their own resource binding – Banga [2] proposes an extended *bind*() system call. With this new version of *bind*(), a socket can be configured to only accept connection from a specified source address range. The application can set up different sockets for different request sources. Even if one client tries to attack the server by flooding it, clients from a different source ranges are not affected in this model.

[1] TCP Packet with Synchronize Flag: used for initial handshake

However, the application has to use the new *bind*() system call in order to benefit from the new model. Supporting this new system call requires invasive changes to the application. To be able to distinguish between several client classes, network initialization code has to be changed as well as the configuration file handling,

A different solution to configure the resource containers which is to be used for a connection is described in section 3.3.5, using the packet filter to classify connections.

2.4.4 Separation of resource- and scheduling-binding

Banga introduces both resource- and scheduling-binding. The resource binding determines which resource container will be used to account kernel processing. A process may belong to multiple scheduling bindings which are used to determine which process should receive resources from a container.

Consider an event-driven server which processes requests in the same process. It can only work on one request at a time. Other requests thus have to wait for the last one to complete. When such an application only gets resources for the request currently processed, high priority request may be starved by a low priority request. This priority inversion can be solved by allowing a process to attach to scheduling-bindings of several resource containers.

A process can then bind to all resource containers associated with a waiting event. As all resources available to the server are used for all requests, no individual request will get stuck provided any request has resources left. But as a request consumes resources from different containers, these are also accounted to all containers involved.

Forcing requests to pay for actions triggered by other clients leads to new priority inversions. A better way to solve that problem is not to have other clients pay for excessive resources consumed by one request, but have the server do it. This is a natural choice, as it is the server that is responsible to process all requests in time anyway.

A server can achieve this semantic by adding itself to the scheduling binding of its own and the clients resource container. However, as both resource containers have resources available, they are used in equal shares. As a consequence, not all resources required by a request are entirely accounted on behalf of the client, even if assigned a container holding enough available resources.

By using a different way to bind resource containers to processes, resources used by server processes can be accounted fair. This is described in section 3.3.3.

Chapter 3

Energy-aware resource model

Even when accurate information about the power consumption of the machine is available, it is not trivial to use this information to correctly account and control the use of energy. Resource containers as introduced by Banga [2] offer a very flexible resource abstraction. This work proposes an energy-aware resource model which is based on these resource containers. However, they were originally built to account CPU usage and not energy consumption. Several improvements are needed to be able to accurately account and control power consumption. In addition to these modifications, generic ways of enhancing resource accounting are discussed.

This chapter will explain how an Application Programming Interface (API) of the new resource model has to be designed. Then, concepts of in-kernel accounting and throttling of resources are presented, followed by a new way to account resources used by server processes. A detailed view on the implementation is given in the next chapter.

3.1 Concepts for the Application Programming Interface

There are a lot of applications that need to run on a typical system. Having to modify them in order to support a new operating system is too expensive in most cases. Sometimes they even cannot be updated at all, because source code is not available for all applications.

This work introduces an application interface which is compatible to old resource scheduling models, but still allows new applications to use the full power of resource containers for accounting. That way, resource containers are transparent

for unmodified applications but resource container aware programs can tune their resource configuration.

For example, Unix processes generally assume that a child process runs independently to its parent and gets its own set of resources. This is still necessary even when resources are controlled by resource containers and not by the process itself. On the other side, the new resource model should allow to build special policies for groups of processes. If resource restrictions are defined for a process, it must not be able to break them – not even by creating new processes.

This section focuses on resource container aspects that are visible from user-space. After introducing methods to name and access resource containers, this work will describe how resource containers can be arranged in a hierarchical structure to achieve desired features. Last but not least, resource container creation is described. How they are used to account and control resource usage will be discussed in section 3.2.

3.1.1 Accessing resource containers

In order to work with resource containers, an application first has to obtain a capability to the affected container(s). Such capabilities can be stored in file descriptors.

In this work, file descriptors do not only hold a capability to a normal file, but also an extra capability to a resource container. Storing such a capability into such a file descriptor will be called *attaching* the resource container to the file descriptor. This is used both to obtain a user-space representation of the resource container and as a method to configure resource usage associated with this file descriptor.

There are several ways to acquire a capability to a resource container:

- a process can always gain a capability to its current resource binding,
- privileged processes may gain access to the root container,
- child containers can be enumerated once the application possesses a capability to their parent,
- as they are stored in file descriptors, capabilities can be transferred to other processes
  - by passing file descriptors to a child process,
  - by sending a file descriptor via $sendmsg()$/$recvmsg()$,
  - by storing it in a special file system.

The first three ways directly export references that are needed by the kernel anyway. Transferring capabilities has to be described in more detail.

When a child process is created with *fork*(), it inherits all open file descriptors from the parent process. Those file descriptors are shared between both processes until they are closed. All resource containers attached to shared file descriptors are immediately visible in both processes. Passing a resource container to a child process can be done by just attaching it to a file descriptor, and leaving that descriptor open for the child process.

Another way to share file descriptors is to send it over a socket. Unix domain sockets (also named local sockets as they don't work over a network) do not only support sending data. They allow to send user credentials and file descriptors from one process to another. Packets transmitted over Unix sockets consist of two parts: the actual message and a control message. Both are sent to the receiving process, but the operating systems interprets the control message in a special way. The control message `SCM_RIGHTS` carries a file descriptor number. If such a control message is received, this file descriptor from the sending process is duplicated. In contrast to the system call *dup*(), the newly created descriptor is inserted into the receiving process's file descriptor table. In effect, Unix sockets allow an interprocess *dup*(). With the file descriptor transferred to the receiving process, it now has access to attached resource containers, too.

Storing resource containers in normal files does not work. Resource containers always get attached to file descriptors, not to the files themselves. However, giving names to some special containers is very important. For example, everything working for a service or user should get a parent resource container belonging to that service or user. When the user logs in multiple times, his processes somehow have to obtain a capability to the parent container used in the first session. Other examples include the Resource Policy Database (section 3.3.5) or Cluster Reserves (section 6.1.4). Capability based access is impractical for these use cases, as some way to obtain these capabilities is needed.

One possibility to build an ACL based storage for resource container is a special daemon holding capabilities to all these containers. It listens for requests and eventually transfers containers to other applications via *sendmsg*() if they are allowed to do so. On login of a new user, this daemon can be contacted to retrieve the user's own parent resource container to correctly configure the resource usage of the new login shell.

Another possibility is to use a file system to store resource containers. Opening

files from a file system is the standard way to obtain file descriptors and even if no existing file system can hold resource containers, it is possible to build one that can store them at least while the machine is running. However, such a file system cannot be made persistent without having persistent processes, as all saved resource containers would not control any process otherwise. But even a file system that has to be populated with new resource containers on every startup is a great help.

Both methods have their advantages: the user-space daemon does not depend on new kernel code and the file system can easily be used by kernel code like the Resource Policy Database.

3.1.2 Container hierarchy

Providing different view levels on resources is important to support both high- and low-level accounting and control. A hierarchical structure can offer such levels.

A new level can be obtained by creating a new resource container that just combines other containers to a group. Those grouped containers are said to be children of the new container. All resources accounted to one of them are also accounted to their parent. Resource containers control all resources consumed on behalf of its descendants.

Such an approach enables two important features:

group processes by assigning resource containers that share a common parent. Such groups can be used to implement a per-user or per-service resource policy.

restrict resource allocation by setting limits or priorities in a resource container unreachable by the application. That is necessary to be able to compartment the machines resources.

When there is no way to obtain a capability to the parent of a given container, the hierarchy can be used to differentiate authority to resources. A capability to a container near the root is more powerful than a capability to a leaf container. Being able to modify a resource container empowers to control all resources consumed on behalf of it and all of its children. Therefore applications can be allowed to get a list of child containers as they already control them indirectly. On the other hand, there must not be any way to obtain a capability to the parent of a given container.

Holding a capability to the root container should only be granted to very few (if any) trusted applications, as they will get absolute power over all resources controlled by the resource container subsystem.

Restricting and controlling the resource usage of a given application is simple when creating an intermediate container crafted for this purpose. The application itself will only receive a capability to a child of this special container. It can consume resources via its own resource container, yet it cannot alter the configuration of the intermediate container. If a controlling process retains a capability to this special control container, it can even change the application's resource limit or priority at runtime.

As different levels in the hierarchy are only needed when a special feature is desired, it can be flat by default. Hierarchy features can be set up by the user-space process which needs them. It simply has to change the parent of some containers before using them. The kernel itself does not have to worry about that; it never has to increase hierarchy depth on its own. As the complexity of many operations (accounting resources, checking if there are resources left, etc.) depends on the depth of the container hierarchy, a shallow one results in a small performance benefit.

3.1.3 Container creation

New resource containers are created by cloning an existing one. This is either done explicitly by a special system call or implicitly on *fork*().

Cloning a resource container creates a new one which is situated at the same hierarchy level as the original one (Figure 3.1). The clone is not an identical copy, it inherits only some properties. It gets the same parent container and the same resource limits, but is otherwise empty: it has no children and is not used anywhere yet.

A newly forked process gets a resource container of its own for per-process accounting but shares all higher-level containers with its parent process. Both processes now have to share their available resources, as all limits are configured in parent containers. However they are not directly connected and can be scheduled independently.

Explicitly created resource containers are not bound to any process yet. The application just gets a capability to the newly cloned container and can use that for any purpose it wants to.

By always creating resource containers that are already part of the hierarchy and not allowing to detach them, the special case of a container not connected to the root can be avoided.

After creation, a resource container exists as long as there are active references to this container. References counted include file descriptors to which the container

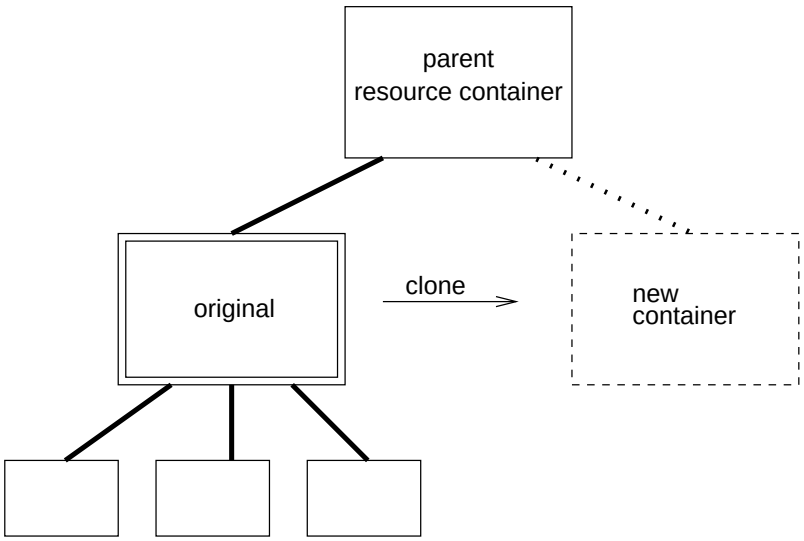

Figure 3.1: Cloning a resource container – the newly created container shares the parent and resource limits with the original one

got attached as well as process resource bindings and children containers. The last case is a bit special, as the reference to the parent container is not exported to user-space and is only used for internal bookkeeping. The opposite direction (list of children), which is exported to user-space, is not counted to avoid loops. This way, parent containers are not destroyed as long as they still have children.

3.2 Resource apportionment and scheduling

Once the operating system has detailed knowledge about the consumption of resources, it can start to control its rate. Resource consumption can be shaped by introducing limits that define how much resources may be used within some time interval.

This section describes how processes can be throttled in order to enforce resource limits, especially energy consumption. Rate limits can be defined at different time scales and for all supported resources. In addition to hard limits, the scheduler can also use resource containers to come to better scheduling decisions.

3.2.1 Enforcing resource limits

In a typical system, there are several limits that must not be broken. Most of them are “high level”-limits, not restricting a single process but groups of processes or

even the entire machine.

Using the scheduler to enforce such limits may not always work: other actions that happen without knowledge of the scheduler may use resources, too.

For example, incoming interrupts may trigger complicated kernel processing. Or the kernel may work on things unrelated to the running process while processing a system call. These actions are accounted, too; but to a different resource container than the one of the currently running process. Thus the time slice given to the current process is not affected. If enough resources are consumed by other means, the resource limit of a common ancestor may be violated (see Figure 3.2). As there is only one root container, even completely unrelated containers always have one common ancestor. Only checking that a process consumes no more than the resources which were given to it may easily result in a violation of resource limits.

This is not such a big problem when only CPU time is accounted, as almost all CPU time is under control of the scheduler. Other resources are more difficult to handle: I/O operations often happen asynchronously, energy consumption of hardware devices is not constant and may still be high some time after an application triggered an operation.

These resource consumptions may be controlled by another scheduler (I/O scheduler, network scheduler, etc.) but the CPU scheduler does not know about them. Even by making all schedulers in the entire system aware of resource containers, every scheduler can only enforce limits local to his scheduling domain. Global resource limits spanning the entire machine are more difficult to keep. In order to enforce such limits, the CPU scheduler may have to belatedly reduce the amount of resources granted to a process when resources are consumed otherwise. (Of course, the same is true for all other schedulers in the system, but this work will continue to use the CPU scheduler as an example; results can easily be applied to other schedulers)

If resource limits are to be enforced properly, all resource containers effecting the current process have to be checked for available resources. These checks have to be repeated frequently to be able to quickly respond when a resource container is depleted. Operating systems typically provide a timer tick which is used for such periodic actions. Instead of checking the remaining time slice of the process, several resource containers have to be checked each time.

Every resource container that is an ancestor of both the process's own container and some foreign resource usage may affect the running process and has to be monitored. As it is difficult to determine which ancestors are contemptible, the

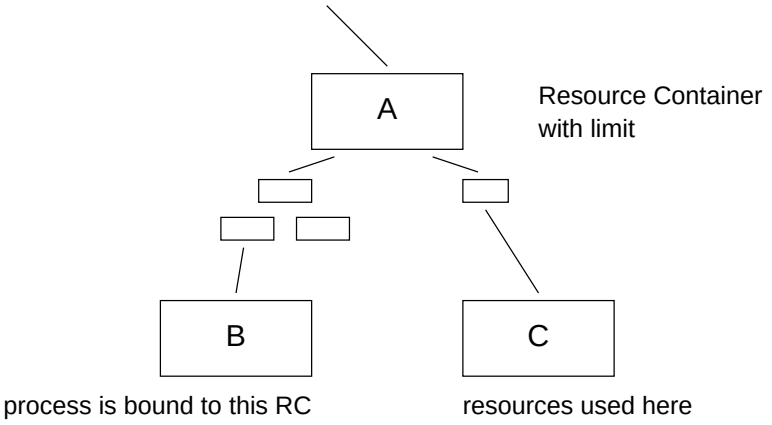

Figure 3.2: The scheduler has to reduce resources given to the process bound to container B in order to enforce limits defined in a parent container A if resources are consumed on behalf of resource container C.

easiest way is to check all of them, up to the root.

A requirement for these checks is of course that the parent containers are updated as soon as some child container consumes resources. That way every container stores the resources used inside its sub-hierarchy.

Accounting and resource checking is shown in Figure 3.3. Every resource container has a field `used` which stores all the resources consumed by this container and its children.

To be able to test if a container has available resources, every container has a field `rc.max`. A container is out of resources as soon as `rc.used` exceeds `rc.max`.

```
proc resources_consumed(rc, n) ≡
  while (rc) do
        rc.used := rc.used + n
        rc := rc.parent
  od
end
```

```
funct check(rc) ≡
  while (rc) do
        if ¬has_resources(rc)
          then exit(0)
        fi
        rc := rc.parent
  od
  exit(1)
end
```

Figure 3.3: Reliable check of resource limits: all parent containers are always updated and checked together with the one belonging to the resource binding of the running process

3.2.2 Time slices

In general, the rate of resource consumption is limited, not their total amount. A simple fixed absolute limit like the one introduced in the last section is inappropriate for this. As time proceeds, limits have to be updated, taking into account the time passed and the resources consumed.

For practical reasons, time is split up into epochs. For every epoch, there is a fixed limit of resources that may be consumed. Each resource container has an absolute resource limit per epoch. This limit is computed at the beginning of each epoch. Resources may be consumed up to this limit. Additional processing has to wait until a newly set, higher limit again allows to consume resources. (See Figure 3.4)

Periodically, resource containers have to be refreshed. Calculating the new limit is easy: it is set in a way to allow a predefined consumption within the next time slice. A possible over-consumption in the last time-slice has to be considered, too. This is done by comparing the last limit and consumption and using the lower of the two as basis for the new limit: $rc.max := min(rc.max, rc.used) + rc.limit$

Many policies define an average limit, but allow bursts to temporarily violate this limit. To be able to control the speed of these bursts, a more fine-grained limitation system can be added.

Time is independently divided into several time slices. Instead of one `rc.limit` and one `rc.max`, there now is one for each time slice. At each point in time, all limits from all time slices have to be kept (see Figure 3.5). The resource container is out of resources, if one of those is violated.

Long time slices (around 1 s) can be used to set course limits, for example controlling the average energy consumption of the machine in order to meet some intended battery lifetime. Short time slices (around 100 ms) can be necessary to reduce spikes that might overload the power supply unit.

3.2.3 Accounting multiple resources

Most operating systems have to account a lot of different resources. Several schedulers need information about what is used by each entity. System administrators want to have detailed statistics about their machines.

In some situations, one resource may be enough if it adequately describes the overall expense of an operation. Energy consumption is a good candidate for that. However, an operating system accounting several resources at once can support

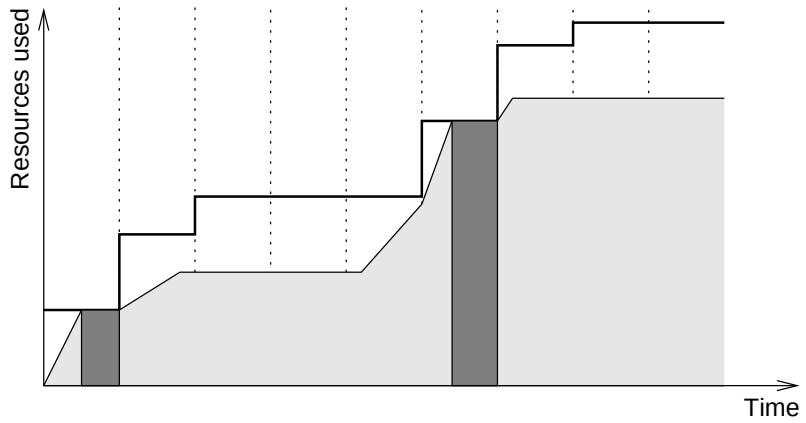

Figure 3.4: Resource limits – a process may run as long as the consumed resources (light gray) do not violate the resource limits (thick line). Everything bound to the resource container is halted if it runs out of resources (dark gray).

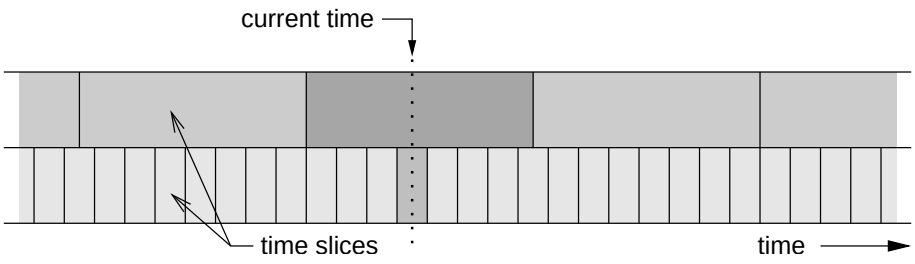

Figure 3.5: Example for a policy using two levels – both resource limits of the two highlighted time slices do apply at the same time.

more powerful resource policies. Resource containers can easily be extended to include several resources simultaneously. That way, the advantages of resource containers are used for all accounted resources. If resource container limits are enforced effectively, they are even more powerful if several resources are used, as very sophisticated policies can be defined.

Each resource container needs separate accounting information and limits for each supported resource. All of the limits defined for the various resources have to be kept.

Even if the system supports resource guarantees, applications may not be able to take advantage of them as their resource usage profile may conflict with a resource limit defined for another type of resource. Solving the reservation problem for multiple resources is intended for future work; this work will focus on limits only.

3.2.4 Resource apportionment

The resource limitation introduced in section 3.2.1 can be used independently to the scheduler(s) in the system. When resources for one process (or another schedulable entity) are used up, the scheduler has to pick a new one. This way, only the point in time the scheduler has to be called is dictated by the resource container subsystem. The scheduler can choose any process with the system still keeping global resource limits.

Thus resource apportionment is separated from resource limits. The scheduler chooses one possible resource apportionment within the configured resource limits.

This leaves very much freedom to the scheduler. As the scheduler does not have to check resource limits itself, it can implement any scheduling strategy which is optimal for the designated use of the system.

The resource containers already provide a lot of information about the resource usage of different components. This information can be utilized by the scheduler to improve scheduling decisions.

When multiple resources get accounted, there are several possibilities for making decisions:

- use one fixed resource,
- a hierarchical scheduler can use different resources at each inner node,
- calculate some "overall" resource usage and use that.

Using a single resource for scheduling may be interesting when some resources are for accounting only and should not influence the behavior of the system. One may also use different resources for different schedulers, for example build a CPU scheduler which bases its decisions on the CPU usage and an I/O scheduler, sorting requests by throughput. This scenario is very similar to what is done today in general purpose operating systems.

A lot of work has already been put into hierarchical schedulers [12, 22]. They consist of several scheduler nodes which are arranged in a hierarchy much like resource containers. Each scheduler node distributes all resources it receives between its child nodes and processes assigned to that scheduler node. Hierarchical schedulers can easily be combined with resource containers A scheduler node is used for every resource container holding more than one child or process. Thus there exists a very simple mapping between the resource container hierarchy and the scheduler hierarchy. Every scheduler node may have a policy of its own.

Instead of using a single resource for a each scheduler, it is possible to use combinations, both for hierarchical and non-hierarchical schedulers. They cannot be combined directly as each type of resource may have a totally different unit which is measured. Calculating the usage ratio between two consumers yields a number without unit. Such a ratio can be build for each resource type. The overall usage ratio of two resource containers can then be defined to be the maximum or mean value of all these per-resource ratios, making it possible to compare the resource consumption of two entities.

Each scheduler node in a hierarchical scheduler may calculate the usage ratio of all child containers compared to their parent and use these values as a basis for scheduling decisions.

Non-hierarchical schedulers can compare resource usage with overall consumption (stored in the root container). However, such an approach completely ignores the container hierarchy. A better way is to compare usage to the direct parent container again, as done for the hierarchical scheduler. As resource shares

given to parents are not taken into account by higher-level nodes, this has to be done explicitly. The minimum of the usage ratios of all parent containers is a good metric showing if some resource container received less resources than siblings. Processes bound to such a resource container should be preferred over others.

Both hierarchical and non-hierarchical schedulers do not necessarily have to be aware of resource containers, but can gain from the information provided by them.

3.3 Handling of server processes

One motivation for resource containers was to make applications responsible for resources consumed by kernel services they are using. But applications do not only use services provided by the kernel but also many which are provided by user-space. A typical operating system runs quite a lot of server processes. Those export services which can be utilized by applications. This work proposes a new way of accounting resources used by such a server.

Most clients access servers via pipes or sockets. Examples include the X Window Server, all Internet servers and increasingly desktop applications which are composed of several components. Requests sent to a server via these channels can be detected by the operating system and used for resource accounting.

First the achievable effect on the system will be shown, followed by an in-depth discussion about how it works.

3.3.1 Effect on the client process

An application can indirectly use system resources if it instructs another process to do some work. Such a procedure is often desired as only a special server process is able to complete the task because it has got more knowledge or authority. In a process-centric model, the client process is not responsible for the resources consumed by such a server.

It is difficult to detect which processes are the really expensive ones. This is both a problem for the local system administrator who is wondering why his machines are overloaded and for the scheduler which may incorrectly prefer a presumingly inexpensive task that is “only” sending requests to overloaded servers.

When accounting is based on direct *and* indirect resource consumption, then a lot of scenarios can be handled more fairly. If an application sends expensive requests to other processes and all those consumed resources are accounted to the

client, it will quickly run out of resources. The scheduler defers that client, resulting in a system that prefers finishing existing tasks over creating new ones. This helps to reduce the overall system load.

3.3.2 Effect on the server process

If resources consumed by a server process are accounted to the client, the server does not need to provide its own resources.

The scheduler now prefers those servers working for a high priority client. Resources given to the server mirror importance of requests.

However, one has to be careful that the server process does not entirely depend on the client's resources. This could lead to new priority inversions when it works for a low priority client while a high priority one waits for the server to become available again. Malicious clients could even start denial of service attacks by using special resource containers that do not hold enough resources to ever complete the request.

Obviously the server must be able to use some resources that are independent to the client. Like every other process, the server still has its own resource container. Resources available here can be used when the client runs out of resources. Those resources given to the server's own resource container should be carefully chosen to enable the server to always complete a request in time. However, too much resources do lessen the impact of client-based accounting as low-priority clients could get too much resources.

3.3.3 Resource bindings

Every server process has to keep a list of resource containers which are belonging to a clients but should be used by the server. This list is named the process's *resource binding* and is very similar to a combination of the resource- and scheduling-binding as introduced by Banga [2].

All resource containers affecting a process are stored in the resource binding. The first container in this list is used to account resources consumed by the process and is responsible to ensure resource supply. If that container runs out of resources, the next container in the list is used. As soon as resource containers regain available resources, they are preferred over containers which are listed later.

A server can use this mechanism to account resources consumed by a clients request to the clients resource container as long as there are resources left. Once

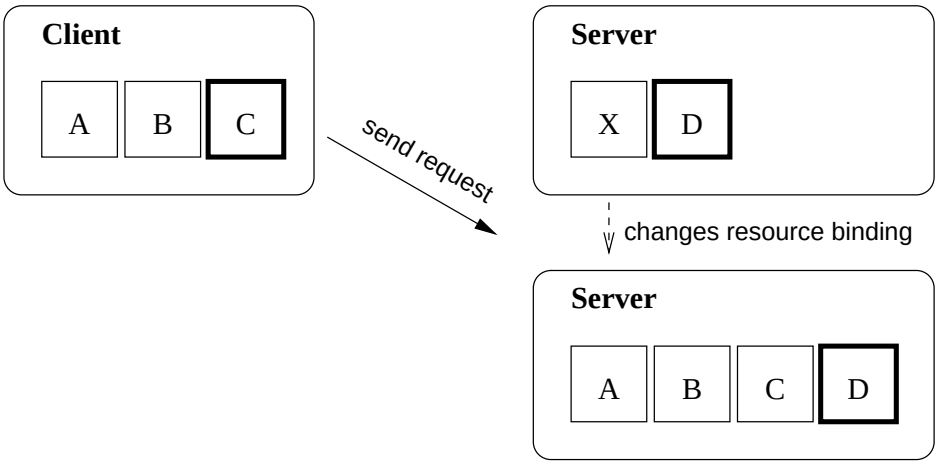

Figure 3.6: Propagating a resource binding to another process; the squares represent resource containers belonging to the resource binding. On each update, they are replaced with the binding of the client; only the one directly belonging to the process (bold square) is kept.

the client's resources are used up the server automatically falls back onto its own resource container, using resources that are reserved especially for this purpose. This is achieved by putting the resource container which is owned by the process to the end of its resource binding. Resource containers belonging to a client are always inserted at the head of the server's resource binding.

When a server starts to work on behalf of a new client, its old resource binding is discarded. Only the one resource container directly belonging to the process is kept. The resource binding of the new client is inserted into the servers resource binding, so that it now begins to consume on behalf of these new resource containers (see Figure 3.6).

3.3.4 Manually configured clients

A server application may have some special knowledge about the client(s) it is talking to. It can share that information with the operating system, to get accurate accounting of the resources used while working for these clients.

First, the server has to create or otherwise obtain a resource container for each client it is talking to. Then, every time the server process works for a different client, it has to change its resource binding.

There are two possibilities to do this:

- replace its resource binding with the new one,
- add the client's resource binding to its own.

Replacing the scheduling binding of the server should only be done if the server has direct control over the resource container used, or if this server process does not depend on being able to finish its work. Otherwise the server could starve if the container does not hold enough resources to complete a task. This method is suitable if the server process itself created the resource containers which are used for the clients. For example, an application might be internally divided into a client and server model. These parts know each other and can correctly set up the resource bindings of each sub-process. Changing a processes resource binding to a new resource container can be achieved by calling *rc_choose*() (see Appendix A.2.10).

Simply replacing the resource binding does not always work. The server process may want to keep a "backup"-container for the case that the client's resource binding runs out of resources. Therefore, the recommended way to propagate resource bindings is to combine server and client resource binding as described in section 3.3.3.

As most server processes frequently work for different clients, the resource binding update has been folded into normal request processing. This saves the extra system call for changing the resource binding. Typically, client requests are read by the server from a pipe or socket. When that happens, the resource binding can be changed without any additional direct notification. As such a behavior is not desired for every file descriptor, it has to be enabled explicitly. This is done by attaching that resource container to the file descriptor and setting a special flag (`O_SERVER`) using *fcntl*() (see Figure 3.7). Once all file descriptors that transfer client requests are correctly set up, all resource binding updates are done automatically.

Should the server receive requests via a different channel (for example shared memory, signals, et cetera), then it still has to trigger the update itself (by reading from a dummy file descriptor, e.g. one pointing to `/dev/null`).

In most cases, the flag can even be determined without the application having to call *fcntl*(). This will get more important for the next sections.

For sockets, the `O_SERVER` flag can be set automatically. Normally, clients connect to a listening server. If a *connect*() is invoked on a socket, it is considered to be the client side of the connection. All other sockets are considered to be server

1. attach a resource container to a file descriptor
2. set the O_SERVER flag for that file descriptor using *fcntl*()
3. *read*() from that file descriptor

Figure 3.7: Steps a server process has to do in order to change its resource binding

side sockets; O_SERVER is automatically set for those. The server process has to change that flags only if it uses sockets in some nonstandard way.

Determining client and server side of pipes is not generally possible. Pipes are unidirectional, making it necessary to create multiple pipes in most situations. The operating system does not know which one (if any) is used to transfer requests to a server. The only exception are named pipes. Typically, they are opened read-only only by servers. In that case, O_SERVER can be set for pipes, too.

3.3.5 Centrally manageable client associations

Attaching resource containers to file descriptors is not hard and being able to update the resource binding on *read*() greatly simplifies correct resource accounting in server processes. But for some server programs, it is very difficult to do even such a simple change. At the time of writing, there is no known application that already supports resource containers. Patching all server applications that have to run on a typical machine is too expensive or simply not possible.

Multiple different services may have similar resource policies. For example, clients in an internal network may be prioritized over those connecting from the Internet. Repeating the same configuration in every application is error-prone and difficult to maintain.

Moving resource policy into a central database kills two birds with one stone. Most operating systems already have a policy database that describes what to do with network packets: the packet filter. As requests are transfered in such network packets, the packet filter can classify requests – at least based on the source of the request. A simple extension that enables to attach resource containers to network packets can transform the packet filter into a *Resource Policy Database*.

The resource container attached to a network packet can be used if a server process reads a request from a socket. If no container has been attached to the corresponding file descriptor, but the O_SERVER flag is set, then the operating system

can look at the packet carrying the request. In effect, a default resource container can be defined for sockets. If the application does not itself supports resource containers or simply has no own information about a client, then the global default is used. Manually configured file descriptors are not affected, as these take precedence over the Resource Policy Database.

The packet filter of the operating system may not be advanced enough to correctly classify all requests. In this case the application still has to be modified to correctly configure misclassified requests. Especially if requests are to be classified based on its payload, it is difficult to do this with a packet filter.

3.3.6 Automatically obtaining client-server relationships

For locally originating packets, the resource binding for a request can be retrieved without having to depend on a Resource Policy Database. The client is simply the process sending the network packet that carries the request.

When the resource binding of the sending process is attached to every network packet then resource bindings of clients can automatically be propagated to the server process.

Most local processing can be accurately accounted this way without having to change applications or system configuration. Examples that work out-of-the-box include the X Window System, local DNS forwarders and syslog.

Only if an application wants to modify this default behavior does it have to know about resource containers.

3.3.7 Transitivity

Many processes are not a pure client or server. They can be both at the same time. A process that receives a request from some client may have to use another service in order to finish the request. For example, a mail server has to use the domain name service to be able to relay incoming mails (see Figure 3.8).

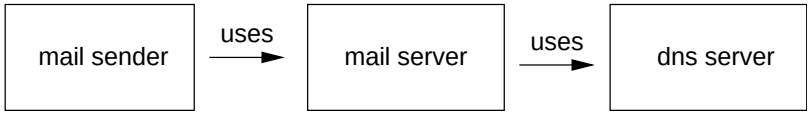

Figure 3.8: A process may be server and client at the same time, leading to transitively propagated resource bindings

As the original client indirectly triggered the DNS query, it should also pay for it, even if it has not issued that request itself. Using automatic resource container propagation as introduced in the last section, such an accounting model is possible. A sending process just has to pass on its entire resource binding, not just its own resource container. That way the resource binding of a process accumulates all resource containers of processes that act as clients or proxies for the current work.

The situation quickly becomes difficult when the graph of client-server relationships gets more complex. One process may utilize several services at once, or use the same service multiple times, perhaps while using different resource bindings itself. As an effect, resource bindings are highly dynamic, and can change often and independently.

A process acting as client may change its resource binding directly after sending a request. To avoid losing information in that case, one has to create a snapshot of the resource binding for each request. Simply referencing the sending process or its resource binding does not work, it has to be passed by value.

One has to be careful that the client-server graph does not degenerate into a tree with one client at the root that is made responsible for everything. This could happen if *all* inter-process communication would trigger a propagation of resource containers. The process receiving initial user input (e.g. the display server) could then be made responsible for all user interactions. This work however only takes into account sockets and named pipes, completely ignoring shared memory, virtual terminals, signals and other forms of communication. This is a good limitation as it covers almost all client-server relationships but does not include commands from the user interface.

3.4 Summary

This chapter described an advanced resource model built on resource containers. An API has been described that allows easy access to the resource configuration, while keeping it efficient and secure. Several methods to automate this configuration are shown, based on global administrative policies and information gathered by the system itself.

Especially client-server relationships are used to account resources to the client that triggered their consumption. This is used to obtain fair resource apportionment and meaningful per-client resource limits.

To be able to effectively control power consumption, a resource limitation system is introduced that takes into account resources used by other means.

The next chapter will show how to implement such a resource model.

Chapter 4

Implementation

The resource container system described in this work has been implemented for the Linux [18] operating system running on an Intel Pentium 4 processor. It consists both of changes to existing code and new code added to the Linux kernel, combined with a user-space library that enables applications to easily use the new features.

This chapter describes how resource containers are added and integrated into the system, followed by a detailed view on resource accounting and throttling.

4.1 Resource container

Resource accounting is broken out of process structures and given a body of its own, creating a new abstraction: the resource container. These are kernel objects that store resource consumption of some part of the system together with limits and other information (see Figure 4.1).

Many operations have to know if a resource container still has resources available. This calculation is expensive as comparisons have to be done for every supported resource and all configured time slices. As the result changes infrequently, it makes sense to cache it. To further optimize access to this field, it is put at the beginning of the resource container structure so that it can be read without having to specify an offset to resource container pointers.

The actual resource information is embedded into the resource container; one accounting structure for each supported type of resource. Information needed for resource accounting is discussed in more detail later (section 4.3).

To be able to arrange resource containers in a hierarchical structure, every container has to know its parent and children. Besides accounting and scheduling,

```
struct rc {
        /* cache for fast paths: are resources left here? */
        int has_resources;

        /* reference counter */
        /* counting: children + procs + fds using this rc */
        atomic_t count;

        /* following are protected by rwlock rc_hierarchy */
        struct rc *parent;
        struct list_head children;
        struct list_head sibling;       /* linkage for parent's children list */

        /* following are protected by rwlock rc_tasks */
        struct list_head procs;         /* processes using this rc */
        struct list_head procs_server;  /* processes serving for one of them */

        /* actual resource info, protected by spinlock res_lock */
        struct rc_resource res[RES_COUNT];

        /* only used for deferred tear-down */
        struct work_struct work_destroy;
};
```

Figure 4.1: In-kernel representation of a resource container, consisting of general information, hierarchy data and process bindings together with the actual resource data (which is shown in more detail in Figure 4.2)

process to resource container bindings are important for user-space resource management tools so that they are able to efficiently describe containers which are to be modified. To store this information, processes include pointers to the resource containers used and every resource container holds a reference to its parent. All those references to a resource container are in turn stored in linked lists belonging to the container.

In addition to the lists that allow to enumerate references to a resource container, there is an additional counter that stores the current number of references to that container, including temporary pointers. This counter is used to determine if a container is still in use or can be removed from the system. Only direct references are counted (parent and process bindings). The parent container maintains a list which does contain a reference to this child, but that reference is not included to avoid loops. That way only leaf containers are removed (possibly resulting in another container becoming a leaf container, which may be removed too).

Interrupt handlers do modify this reference counter as they update global references, but actually destroying the container involves taking write-locks (see section 4.2.1 for details on locking). Such a situation can be solved by deferring the operation and later executing it within a process context. Linux has its own kernel thread just for such purposes. A small amount of memory is needed to add the destroy operation to the list of deferred operations. Interrupt handlers however cannot reliably allocate new memory as they cannot wait for more memory to be

freed. This memory thus has to be allocated early, to be able to definitely destroy the container.

4.2 Integration into the kernel

To be useful, resource containers have to be integrated into many parts of the operating system. The scheduling system has to be updated to correctly choose tasks that still have resources available. File descriptors and network packets must be extended, so that resource containers can be attached to them.

Code executed in kernel space has to be extremely careful to keep all data correct and consistent – there is no safety-net underneath any more. Especially locking can be difficult on multi-processor machines. In addition, there is only very limited stack space and floating point arithmetic is not easily available.

4.2.1 Locking

The Linux operating system is capable of running on multiprocessor machines. These machines have multiple main processors, executing code in parallel. All processors share the system memory. When several processors want to access the same part of memory, there may be conflicts. Operations that use or modify a shared data structure have to be properly serialized.

Those code parts that operate on shared objects which have to be protected against other processors are named critical section. Consistency of the protected object can be guaranteed as long as only one processor can enter a critical section at a time. This is realized by *locking* the protected object while some processor is executing in the critical section.

Linux provides several methods to manage critical sections.

Semaphores block the current process if another processor is currently operating in the critical section. This way other processes can run on this processor in the meantime. Semaphores are best used for long-lasting critical sections. The resulting context switch makes them unsuitable for code run in interrupt context or for very short critical sections.

Spinlocks simply block the current processor as long as another one is operating in the critical section. Spinning in a tight loop works in interrupt context too, but should be limited to very short durations, of course.

Reader-writer-locks are very similar to spinlocks, but differentiate between read-only and read-write access to the protected object. Multiple readers are allowed at the same time, only writers are forced to have exclusive access.

As operations on resource containers do not require expensive calculations and might happen in interrupt context, semaphores are not used for synchronization here.

Even when spinlocks and reader-writer locks are much more efficient then semaphores, they still effect performance. Especially invalidation of a cache line and the memory transfers needed to synchronize the status of the lock cause a performance drop. One has to carefully choose the locks which are used to protect system objects. Coarse-grained locking protecting an entire subsystem results in minimum locking overhead, as only one lock has to be taken for all operations. However, lock contention can become a problem because the lock is taken very frequently and processors have to wait for the lock to be released. Fine-grained locking reduces lock contention at the drawback of increased locking overhead.

The kernel representation of resource containers consists of three parts: hierarchy information, bindings to processes and accounting information. These are accessed mostly independently, allowing different locking strategies for each of them.

The resource container hierarchy is changed relatively infrequently: only on process creation and destruction or explicitly on application request. However, parent information is retrieved very often – every time resources are accounted or checked. These asymmetric access pattern are optimal for reader-writer-locks. Lock contention is not a problem as writes are rare and one global lock protecting the entire hierarchy is enough.

Process binding information is similar: changes are infrequent and can be easily synchronized with a single reader-writer-lock.

The accounting information is changed very often. Without that asymmetry between read and write access, a reader-writer-lock has no advantages over the more simple spinlock. As locks are taken very often one could try to use fine-grained locking to reduce lock contention. However, this is difficult as each update also updates the root container. In effect, the lock for the root container behaves as a global lock. It is better to just use this one lock, than having to obtain several locks per operation.

If a critical section may be executed by an interrupt handler, then one has to disable interrupts before entering such a critical section, even on a single processor

machine. Otherwise the machine would dead-lock if a processor enters the critical section, gets interrupted and wants to enter the section again in the interrupt handler – the interrupt handler would block without giving the process which holds the lock a chance to release it.

Resource container hierarchy and process bindings are only accessed read-only by interrupt handlers. (The only exception is container destruction, which may be triggered by an interrupt handler. However, actual destruction is deferred to a special kernel thread so that the interrupt handler does not have to touch the container hierarchy.) This allows to further optimize read-locking: it does not have to disable interrupts, as there is no interrupt handler that could block (multiple reads are allowed for reader-writer-locks).

Access to accounting data has to always disable interrupts, as some interrupt handlers do modify that data.

4.2.2 Scheduler

The Linux 2.5 scheduler is already very powerful and does not have to be modified a lot to support resource limits.

- On every context switch, the resource container sub-system has to be notified; this is necessary to correctly account resources to the resource binding of the currently running process.
- Periodically, the resources of the currently running process have to be checked.
- The scheduler itself must not consider processes with a resource binding lacking available resources.
- The scheduler chooses processes by using the highest-priority runnable process. These priorities may be adjusted using statistics taken from resource containers.

All those changes are rather small and straightforward.

4.2.3 Network packets

In Linux, network packets are represented by a `struct skb`. It holds all headers, payload and other information gathered about the packet, for example the socket it belongs to. A `struct skb` is both used for incoming and outgoing network packets.

This `struct skb` is extended by a pointer to `struct rc`. That pointer identifies the resource container associated with the packet. Such a resource container can be used to configure the resource binding of a server process.

When the packet is not transmitted by hardware but delivered locally, then the `struct skb` is simply moved from the output to the input queue. Information provided by the sender is automatically available to the receiver in that case. This is a major advantage, as `skb->rc` is preserved for local connections; it can be initialized to the resource binding of the sending process to get automatic client propagation.

4.2.4 File descriptors

Every Linux process owns a file descriptor table. This table stores pointers to `struct file`. A `struct file` holds all the information necessary to access any file.

To be able to attach resource containers to file descriptors, a new field is added, `file->f_rc`. This field is used to automatically update a process's resource binding on *read*(), if the flag `O_SERVER` is set in `file->f_flags`.

The resource container attached to the file descriptor can be modified by applications using the newly introduced system call *rc_attach*(). When calling *rc_attach*(), one can choose what to attach:

- resource container belonging to the process
- root container – this is only available to processes holding the POSIX capability `CAP_SYS_RESOURCE`, which on most systems is only the case for the user *root*,
- the resource container specified by another file descriptor – capabilities can be copied that way, this is to attached resource containers what *dup*() is to file descriptors,
- a clone of another resource container – creating a new resource container that has the same parent and resource allocation as the one specified,

- child containers – these can be enumerated one after another.

Several library functions make it simple to use *rc_attach*() to create file descriptors holding the desired resource container. See the API reference for details (Appendix A.2).

4.2.5 Resource container file system

Linux already contains a lot of memory file systems; these hold all their data in main memory and don't use a backing store to make it persistent. That is very similar to what is needed for a file system storing resource containers. The `tmpfs` file system is used with some modifications:

- on file open, the resource container stored in the file system is attached to the new file descriptor,
- on file close, the resource container attached to the file descriptor is stored in the file system if the file was opened in read-write mode,
- writes to files are disabled,
- reads on files print some small statistics

The file system (named `rcfs`) is fully functional otherwise; everything one expects from a file system already works: the user can create directories and symlinks, change permissions, etc. Only does this new file system store resource containers instead of data.

4.2.6 Propagating resource bindings

Automatic update of resource bindings on *read*() has to take into account resource containers attached to the file descriptor and those provided by the object itself. While sockets or pipes can propagate resource containers based on the sender of a message, this is not true for normal files. However, updating the resource binding should work for all file types, if only to support manual switching by attaching a resource container to a `/dev/null` file descriptor.

Many objects are represented as files in Unix systems. The dispatching of file operations to the actual object is done by the Virtual File System (VFS) layer in Linux. It implements a meta file system that covers every file in the system; even files that are not reachable by mounted file systems are handled here. The

VFS layer takes care of things such as locking and integration with the rest of the kernel.

Every `struct file` contains a field `f_op`, pointing to an array of function pointers. These functions are used by the VFS layer to call the implementation for that file. This is very similar to a virtual function table used in object oriented languages.

Network sockets, which implement file operations for network objects provide a similar indirection. Once the VFS layer forwards operations to the socket layer, that operation is dispatched based on the address family and on the protocol used.

For sockets, only those low-level functions have enough knowledge to be able to retrieve the correct resource container; all higher levels do not yet work on `skb`s. The same is true for pipes, but here the low-level functions are not buried that deep; they are directly called by the VFS layer. The active resource container is changed by such a low-level function as soon as it detects a container that can be made responsible for its actions.

Resource containers configured by the applications should be preferred over those automatically gathered by low-level protocols. Updating the resource binding based on containers attached to the file descriptor has to be done by the VFS layer itself, as it has to work with every file type. This is done right after the low-level function returns, possibly overwriting any resource container changed by a low-level function.

The resource binding of the current process has to be updated as soon as the system knows the client that is responsible for activation of the process. Instead of propagating the whole resource binding of the client as described in section 3.3.3, this implementation only propagates the first resource container of that resource binding. This is a limitation that was introduced to simplify the code and hasn't been removed as the expected advantages of propagating complete resource bindings are too small compared with the work and time needed for the change. Even with resource bindings holding only two resource containers (the one inherited from the client and the one belonging to the process itself), client-based accounting works very well. To store the resource binding, each process contains two resource container references, `rc_self` and `rc_client`.

4.2.7 IPTables as Resource Policy Database

The Linux packet filter is designed to be very modular and easily extendable. These are the optimal preconditions needed to build a Resource Policy Database on top of it.

IPTables uses *chains* to determine the fate of network packets. Every chain consists of an ordered list of *rules*. These rules conditionally perform actions on the packet. Such an action is named *target* of the rule. Several possible targets are available:

- drop the packet immediately, without further processing of other rules,
- accept the packet – again skipping all other rules,
- print information about the packet to the system log,
- change source or destination of the packet to reroute it,
- use another chain for further processing of the packet
- ...

Whether a rule is activated for a given packet is determined by a list of *matches*. They allow to compare network packets with the policy. When a match does not recognize the packet, then this rule is skipped. Matches include

- interface used to send or receive this packet,
- source and destination address,
- protocol used,
- depending on the protocol: port numbers or other protocol-specific data, like state of a connection.

Both targets and matches may be extended by kernel modules. The module has to implement some functions which are then registered with the IPTables core system. These functions will be called every time a packet has to be matched or a target action is performed. The callback functions can inspect (and change for a target callback) the network packet which is stored in a `struct skb`. When it is done with the packet, a match function returns whether the packet did match or not; a target callback returns a code that determines the fate of the packet: should it

be dropped, accepted (possibly modified) without further processing or should the system continue to examine rules in the chain.

To implement the Resource Policy Database, the `struct skb` got extended to hold a reference to a resource container. The resource container which is bound to the packet can be changed by a new target (`iptables -j RC`) or examined by a new match (`iptables -m rc`).

The Resource Policy Database may be configured with the standard `iptables` utility which is used by the packet filter. It features a similar plugin mechanism to be able to communicate with the kernel modules which implement extra targets or matches. These plugins are implemented as a shared library which provides functions to parse command line arguments and marshals them into the internal data format used by the kernel extensions.

To be able to display useful information about the Resource Policy Database all resource containers are inserted by file name. Thus, a `rcfs` file system is needed and has to be populated prior to setting up the RPD.

4.3 Accounting

Each resource container stores accounting and scheduling information about some part of the system. Accounting information has to stay up to date all the time, so that it can be used for scheduling.

To be able to easily assign consumed resources, the system remembers the currently active resource container for each processor. This active container changes on every context switch or when the kernel starts to work for some other entity (e.g. processing of network packets or other asynchronous tasks).

There may be several resource accounting providers that know how to obtain the current resource usage. The active resource container is used, if such an accounting provider does not itself know which entity was responsible for the consumption.

4.3.1 Generic accounting system

Accounting is important both for very short time periods and for long running processes. Resources consumed in a fraction of a second have to be saved with enough precision while server machines have to keep track of their accumulated resource usage. As floating point numbers are not easily available for kernel code one has to resort to fixed point arithmetic. Integers have to be big enough to avoid over- and

```
struct rc_resource {
        /* absolute value: accounting information */
        u64 used;
        /* absolute values: restrictions */
        u64 max_effective; /* min( max[all] ) */
        u64 max[RES_TIME_SLICES];
        u64 last_used[RES_TIME_SLICES];
        u32 avg_usage[RES_TIME_SLICES];
        /* per time-slot limits: */
        u32 limit[RES_TIME_SLICES];
        u8 limit_scale[RES_TIME_SLICES];
};
```

Figure 4.2: Resource information stored in resource containers – once for each supported type of resource

underflows. 32bit numbers have shown to be inadequate; as a consequence, 64bit integers are used to represent resource consumption.

Together with the accumulated resource usage, other information is to be stored. To be able to constrain resource usage, there has to be the maximum allowed rate and the absolute upper limit for each time slice. The allowed rate can be specified directly (`limit`) or relative to the one of the parent container (`limit-scale`) in this implementation. To aid in scheduling decisions, this system also determines the rate at which resources are consumed on behalf of a container. The data structure used to store all this information is shown in Figure 4.2.

Every resource container can store information about multiple resource types. Each type may need different methods to obtain the current resource consumption rate. The code responsible to gather that information is named accounting provider. The accounting system exports a function that can be called when resource consumption is detected. That function updates the accounting information of the active resource container and all of its parents; type and amount of resources are given as parameters.

User-space usually prefers floating-point values that represent real physical units (for example time in seconds instead of some integer value counting internal time intervals). Applications need some information about the supported resource types to be able to correctly interpret the values gathered by the kernel, Each accounting provider has to export the unit of the resource and the scale that is necessary to convert between the internal representation and that unit.

4.3.2 CPU accounting

In Linux, CPU accounting is done by periodically sampling the running process. On every timer interrupt, the kernel determines the process which was running just before the interrupt occurred. The statistics from that process are updated. Timer interrupts happen about one hundred to a thousand times per second, depending on the kernel version and hardware architecture.

Such a method can be used for resource containers, too. When high precisions are needed, such a sampling method may not be enough, however. Processes running only for a short duration may be completely missed by the system.

Modern processors include a time stamp counter (TSC). That is a monotonically increasing counter, representing the number of clock cycles since the last hardware reset. Such a counter can be used to get very accurate information about the length of time intervals. One only has to save the TSC at the beginning of the interval and compute the difference at the end.

To correctly account CPU time, the TSC has to be read each time another resource container gets activated. The last counter value for each CPU is stored in an array. Calculating the elapsed time can easily be done by scaling the difference of the last and current value with the current clock rate of the processor.

4.3.3 Energy accounting

Even more important than the time the processor was running is the energy it consumed meanwhile. To be able to estimate the energy consumption, one has to gather information about the functional units inside the processor.

Modern processors include performance counters, which can be used to profile applications. They can count several events that are generated while executing code. These events allow to obtain information about how the code is executed.

This work uses a simple accounting provider that uses a linear combination of performance counters to estimate power consumption of the main processor. The kernel maintains a table of weights that can be configured by user-space. Accounting is done similar to the CPU accounting provider, only using a combination of performance counters instead of the single time stamp counter.

Calculation of energy consumption is based on first results from a Studienarbeit by Simon Kellner who is working on software only temperature control [16]. In order to obtain high-resolution samples of the processors temperature, power consumption is used in a mathematical model of the CPU heat sink.

4.4 Throttling

Throttling is implemented by temporarily stopping all processes that are out of resources. The system scheduler may only choose between those processes still possessing resources and has to resort to the idle task if there is no one left.

A hierarchical scheduler should have advantages here, but this work concentrates on other topics and there was not enough time to implement a hierarchical scheduler for Linux.

4.4.1 Halting processes

As long as a process does not have enough resources to continue running, it must not be scheduled. This can be achieved by setting the state of the process to `TASK_UNINTERRUPTIBLE` (See Figure 4.3 for a list of Linux process states).

Only processes which are in the state `TASK_RUNNING` are held in the schedulers *runqueue* and may receive CPU time. When a process is set into the state `TASK_UNINTERRUPTIBLE`, it is removed from the runqueue, preventing the process from being scheduled again. Even signals sent to this process are not handled in this state.

`TASK_UNINTERRUPTIBLE` is used in the Linux kernel when a hardware device is working on behalf of the process (for example, a hard-disk reading data). Such operations are difficult to stop and handling signals while the kernel is executing code in a low-level hardware driver is very difficult; therefore signal handling is simply disabled. When the hardware is done, it raises an interrupt request (IRQ). The hardware driver is notified and can then restore the process state to `TASK_RUNNING`. When determining the load of the system, then processes in the state `TASK_UNINTERRUPTIBLE` are counted, too, as the system is still working for them (even if the main processor is not involved).

Process state	Signals	Counts as load	Description
`TASK_RUNNING` (R)	processed	yes	ready to run
`TASK_INTERRUPTIBLE` (S)	processed	no	sleeping
`TASK_UNINTERRUPTIBLE` (D)	ignored	yes	waiting for hardware
`TASK_STOPPED` (T)	processed	no	stopped, e.g. by Ctrl-Z
`TASK_ZOMBIE` (Z)	ignored	no	process called $exit()$

Figure 4.3: Process states in Linux

Waiting for resources to be available is very similar to waiting for some hardware to complete a task. In the end, it *is* the hardware that supplies these resources (energy, CPU time).

As soon as the system detects that resources of a process are used up, it has to react by setting this process into the state `TASK_UNINTERRUPTIBLE`. To be able to wake up the process, it is inserted into a *waitqueue*; this is a linked list containing all processes that are waiting for some event. In most cases, the hardware driver does this by allocating a `struct wait_queue` on the process stack and linking it into the list. Afterwards, the state is set to `TASK_UNINTERRUPTIBLE` and the process stops running by calling `schedule()` – that call returns when the process state is reset (for example, by the interrupt handler of the device). At this time, the process can be removed from the linked list and the driver can return control to user-space.

This approach is not possible when the process is stopped by an interrupt handler or by `schedule()` itself – as done when limiting resource consumption. The process stack cannot be used because it is not the stack that is currently being used; interrupt context has its own stack. In addition, `schedule()` must not be called – interrupt context has nothing to do with process contexts, it cannot be scheduled away. Recursion in `schedule()` might cause a stack overflow, so using it here is not possible, too. However, one can set a flag and call `schedule()` later. By executing `set_need_resched()` one can make the kernel do this right before entering user-space again.

The process is added to the *tail* of the waitqueue, dynamically allocating the memory needed for linkage. Adding to the tail is important when several processes get stopped at the same time: the one currently running is added first, followed by the other processes. The waitqueue will be processed beginning at the head (FIFO order) when resources are available again. Those processes which are cache-hot are woken up first, reducing cache thrashing and thus reducing the overhead introduced by limiting resources.

Should the newly scheduled process be out of resources too, then the above procedure is repeated. The task is set to `TASK_UNINTERRUPTIBLE` and the scheduler will be invoked again. In the worst case all `TASK_RUNNING` tasks are out of resources and have to be stopped at the same time. But a typical system only has a few runnable tasks, most of them are sleeping. Complexity only depends on the load of the system, not on the number of processes.

4.4.2 Refreshing resources

As described in section 3.2.2, continuous time is divided into evenly sized time slices. For every time slice, there is a fixed limit of resources that may be consumed.

The limits of each resource container are updated at the beginning of each new time slice. Traversing the entire container hierarchy may be expensive on systems with a large number of resource containers.

Refreshing of containers got split up in a first implementation. Instead of traversing all of them at once, they were divided into several groups and refreshed on group at a time. That way, there is no single point that is extremely expensive, resulting in reduced latency. However, the motivation for this approach was to reduce spikes in the resource consumption as processes bound to different groups would start at different times. That didn't work satisfactory as most time, processes are waiting for some common top-level container anyway.

Instead of splitting the work, it is possible to reduce it by exploiting the container hierarchy. On each refresh, the new limit is only changed if resources were consumed on behalf of that resource container or when the rate limit changes. Resource containers that don't match these criteria do not have to be traversed. If the resource usage of a container has not changed, it is guaranteed that no child container was changed either. Those can be skipped, resulting in a faster refresh.

Processes are woken up as they receive new resources.

Chapter 5

Measurements

All measurements are using machines equipped with a 2.0 GHz Intel Pentium IV processor and 512 MB main memory. To verify energy throttling, an ampere-meter was inserted into the power supply of the CPU.

Only a few measurements could be made due to time constraints. More measurements of the system have to be made in future work.

5.1 Accuracy of energy accounting

The energy accounting system used in this work will be described in detail in [16]. It is tuned to never return an estimate that is below the real power consumption.

Depending on the workload of the applications which get executed, energy consumption can be estimated with an error of -0% to +27%. However, an accuracy of 5 to 10% can be achieved for most applications.

When the typical workload of the machine is known, energy accounting can be tuned towards this workload to improve accuracy.

5.2 Effectiveness of throttling

Measurements are shown in Figure 5.1 which shows aliasing artifacts, as both time slices used for throttling and measurement intervals are about 1 s long. Some values exceed the limit because of this effect; the mean consumption is well below the configured limit. When shorter intervals are used for measurement, aliasing effects are reduced and one can observe that power consumption quickly changes between idle and full as processes are halted and started again.

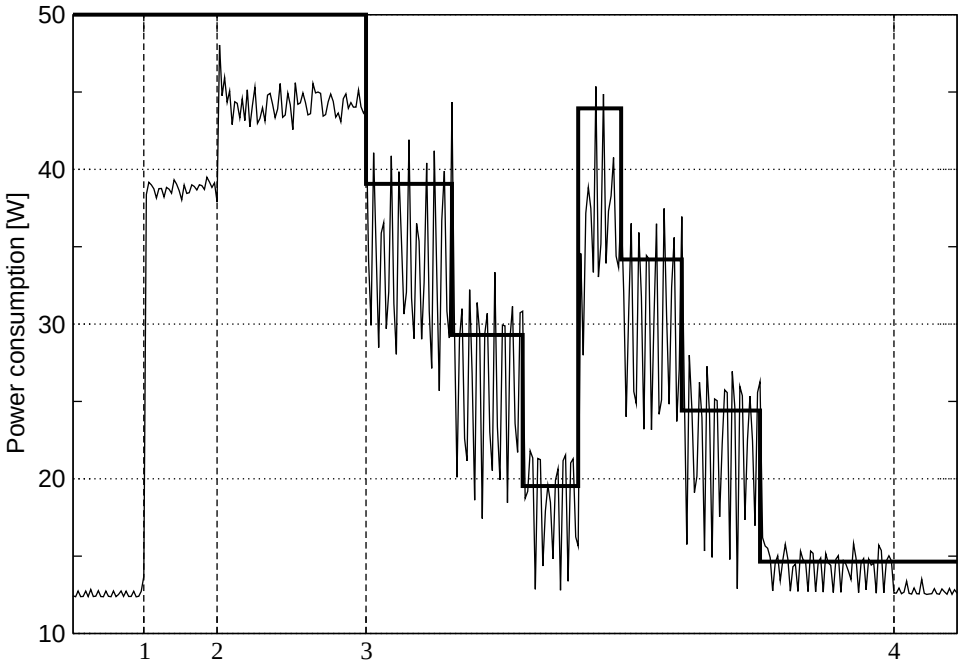

Figure 5.1: Energy consumption first rises as two programs are started (1, 2). Different resource limits (thick lines, for 1 s time slices) are used to reduce power consumption (3 - 4).

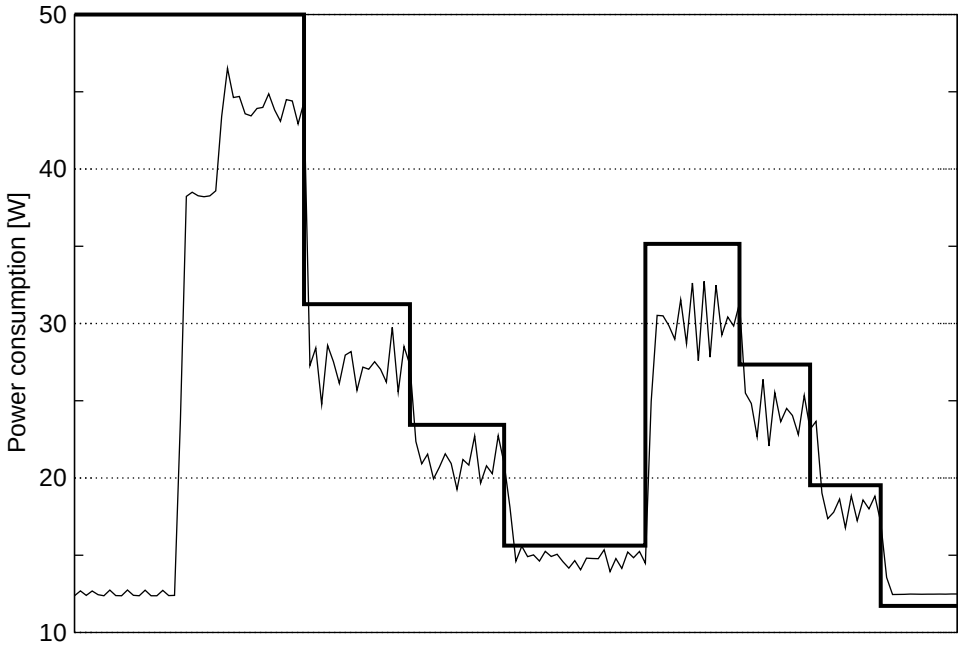

Figure 5.2: Same as Figure 5.1, but using 100 ms time slices.

Using smaller time slices for throttling results in a smoother power consumption (see Figure 5.2). As seen in these Figures, the current system is throttling a bit too much. One reason is that the system tends to overestimate power consumption. Another is that it can only reduce consumption, but cannot increase it afterwards when it detects that mean consumption is below the limit.

Using the HourGlass [23] tool, scheduling decisions and throttling can be visualized. Figure 5.3 shows how the scheduler executes several tasks that have a common resource limit. Each process is interrupted for short periods of time in order to hold the limit.

5.3 Efficiency

Implementing the ability to account and control resources adds a lot of code to the operating system kernel. Some of this new code is called very frequently, as the machine has to constantly account and check all resources that are consumed. This adds overhead to the system, slowing it down.

5.3.1 Context switch overhead

On the implemented resource container system, additional processing is needed for every context switch. The active resource container is changed, at which time performance counters have to be sampled. All resources consumed by the previous process are accounted in resource containers and it has to be checked if the process chosen by the scheduler really has resources left. All these actions add to the context switch time.

Context switch times are measured using test programs developed by [5]. Two different methods are used to trigger context switches: One of these tests uses pipes to pass single characters back and forth between two processes. A context switch is needed for each transferred character. Another test uses threads that block on a lock one after the other. Those tests can generate large number of context switches and are measuring the elapsed time.

The results (Figure 5.4) show an 43% increase when using pipes and a mean increase of 33% for the thread test.

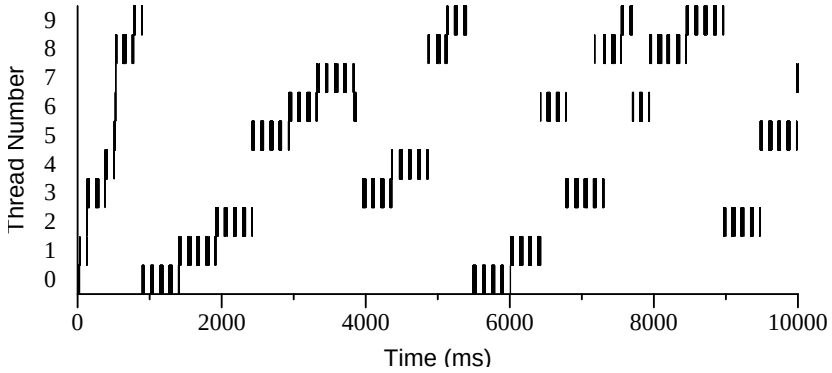

Figure 5.3: Effect of throttling on ten threads sharing a resource container with a 30% CPU limit (using 100 ms time slices). Every bar indicates that the thread is running at that time.

	pipe	2..64 threads					
Original (in μs)	13.917	8.635	8.673	9.373	9.665	9.848	10.729
CPU only (in μs)	16.437	9.662	10.808	11.011	10.984	11.209	12.481
+ Energy (in μs)	19.847	9.696	12.805	12.815	12.883	13.118	14.203
increase	43%	17%	48%	37%	33%	33%	32%

Figure 5.4: Context switch times in Linux 2.5.59 with and without resource container support, for resource container systems that use CPU accounting only and one combining CPU and energy accounting

5.3.2 Timer interrupt

Timer interrupts are used to refresh resource containers and check if the currently running process still has resources left. These actions do need a considerable amount of time, which was measured by reading the processor's time stamp counter before and after the extra code.

On a busy machine, an average of 2.6 μs per timer tick is needed for resource-container specific code. This time decreases a little bit (by 0.5 μs) when the machine is idle, as some processing is optimized away (for example, the idle task is defined to always have resources left and that check can be skipped). With energy accounting disabled, the overhead even decreases to 0.5 μs.

The test machine triggered timer interrupts at a frequency of 1 kHz, resulting in an overhead of 2.6 ms per second ($< 0.3\%$).

5.3.3 Overall performance

Data about affected parts of the system is important, but the user of that system is generally more interested in the overall performance.

A large compile job is used as real-world benchmark. It is a good mixture of many small programs that are both I/O- and computing intensive. The build is using two concurrent compilations (`make -j 2`) to increase context switches and is repeated with an identical copy of the source tree to decrease cache hits.

This test was run on both the original Linux kernel and the modified one. On the test machine, each run took an average of 6:39.2 and 6:43.2 respectively, resulting in an increase of four seconds (about 1% of the overall time).

Chapter 6

Future Work

Accounting and controlling of resources is an extensive field and it is expected that there will be a need for improvements over a long time. This work has identified several topics that still need further studies.

6.1 Accounting improvements

6.1.1 Adopt more resources to use resource containers

Only energy consumption and usage time of the main processor is used in the implementation. It has to be evaluated how the addition of more resource types (power consumption of the entire machine; memory, network and disk throughput) can improve resource policies.

6.1.2 Refund resources borrowed from a server

When a server is working on behalf of some client that already used up all resources that were available to it, then the server has to use its own resources. In effect, the client can borrow resources from the server. The current implementation does not allow the client to refund these resources.

Efficient implementation of such a refund system could improve resource accounting and limitation. However, it is considered to be difficult as the relationships between creditor and debtor can become a many-to-many graph and parent containers have to be taken into account, too.

6.1.3 Client- vs. server-based accounting

The accounting model presented in this work trades a server-based accounting against a client-based accounting. In traditional operating systems, resources are accounted per process, resulting in data about how much each server gets used. With the automatic propagation of resource containers presented here, one effectively collects data on how much resources are consumed by the individual clients.

Even if this allows precise and fair resource usage, many system administrators want to have statistics about the servers running on their machines, too. Using resource containers to collect this data is preferable, as it allows to correctly account resources used by kernel code. The system should maintain two independent counters, one for client- and one for server-based accounting.

6.1.4 Cluster support

The current implementation does not transfer information about resource containers over the network. Thus, only local client-server pairs can be detected and accounted appropriately. As cooperating processes should be located on the same machine for performance reasons anyway, this is not a big disadvantage.

However, in big environments that need load balancing, a service can be located on one or more dedicated machines. For correct accounting in such clustered environments, an extension to the TCP/IP protocols could be build, allowing processes on different machines to use the right resource container for accounting. Corresponding resource containers on each node can be combined to cluster reserves [1] to get cluster-wide accounting and resource control. Using IPv6 [8] for inter-node communication offers an easy way to embed a new IP optional destination header into the network packets transporting a request to a server. This new header can be used to carry a Cluster-global resource container identifier.

Work is especially needed on how to create, transfer and administrate these identifiers.

A straightforward approach would require that the system administrator create a set of cluster reserves which should be distributed. Every machine creates a local resource container for each of them. Whenever a client sends a request and one of these special containers contains the clients resource binding, the identifier corresponding to its cluster reserve is included in the network packet.

6.2 Performance improvements

6.2.1 Multiprocessing

The resource container hierarchy is a global data structure that has to be accessed by all processors. This will become a bottleneck if used on high-end multiprocessor machines. Especially on NUMA[1] architectures, it is too expensive to constantly transfer memory between nodes just to synchronize resource consumption.

As these machines are already optimized to keep tasks on local processors, one could build a separate resource container hierarchy for each unique node. These different copies of the hierarchy would still need to be synchronized, but that could happen less frequently. It is believed that an algorithm based on Cluster Reserves [1] will be useful here.

6.2.2 Faster energy accounting

Reading performance counters and combining them to obtain energy consumption is expensive. If a direct optimization of the code does not help, one has to study if a less fine-grained accounting system can achieve similar results; reducing the frequency at which energy consumption is calculated could lead to a significant speedup.

6.3 Uniform resource consumption

At the beginning of each time slice, all processes that receive new resources are woken up and start to run again. This leads to bursts in resource consumption. Ways to distribute resource consumption over the entire time slice (for example by reducing the CPU clock rate) have to be evaluated.

6.4 Thermal control

Energy consumption of the main CPU can be used to estimate the temperature of the processor die. By tuning the limit of the root resource container, an operating system can assure that this temperature will not rise over a critical value, preventing the processor from destroying itself if the hardware cooling system fails.

[1] Non Uniform Memory Access – every node has its own local memory, access to memory of another node is possible, but much slower

Chapter 7

Conclusions

This work describes an advanced resource model suitable for accurate accounting and limitation of energy consumption.

Using energy consumption as first class resource is a natural choice as it is a real physical value. Resource containers are used to account energy consumption. This hierarchical data structure allows one to account resources at different levels, ranging from single tasks to the entire machine.

The power consumption of the machine is controlled by temporarily halting threads. By dynamically changing limits, it is possible to enforce sophisticated policies for each resource container. The proposed controlling system also takes resources into account that were consumed by devices not under control of that system. Resource consumption that can be controlled is automatically adapted accordingly to be able to keep resource limits.

In order to effectively use all the features offered by resource containers, several methods are discussed that help to automate correct resource bindings. Care is taken to provide sensible defaults while allowing applications to override them when they have to. Even without explicit support by running applications, policies for incoming network requests can be defined. By detecting client-server relationships between local processes, resource consumption can be charged to the entity that is really responsible.

As a proof of concept, the model described has been implemented for the Linux operating system. It is shown that energy consumption can be throttled effectively while only suffering a small performance loss.

Appendix A

Application Programming Interface Reference

A.1 Command line utilities

A.1.1 viewrcs

Prints a listing of the entire container hierarchy. The user calling this utility must have *root* privileges to be able to do this.

Example output is shown below.

```
ref 37  usage 262.7J  avg 12.7W,12.62W  limit 1000000000,1000000000
  ref 4  usage 50.8mJ  task <331>viewrcs
  ref 3  usage 62.2mJ  avg ,4.688mW  server <235>nscd
  ref 10  usage 1.043J  avg 3.906mW,57.23mW  task <327>bash  server <201>syslog
  ref 3  usage 87.6mJ  avg ,1.074mW  task <325>getty
  ref 3  usage 379.6mJ  avg ,2.051mW
    ref 3  usage 103.2mJ  avg ,488.3W  task <310>cron
  ref 3  usage 772.5mJ  avg ,7.031mW
    ref 3  usage 381.2mJ  avg ,3.516mW  task <304>ntpd
  ref 3  usage 0.9725J  avg ,4.98mW
    ref 3  usage 126.3mJ  avg ,585.9W  task <299>sshd
  ref 3  usage 455.8mJ  avg ,2.539mW
    ref 4  usage 80.9mJ  avg ,195.3W  task <288>inetd  server <133>portmap
  ref 7  usage 1.25J  avg ,4.102mW
    ref 3  usage 103.3mJ  avg ,293W  server <219>ypbind
    ref 3  usage 215.8mJ  avg ,781.2W  task <222>ypbind
    ref 3  usage 11.2mJ  task <221>ypbind
    ref 3  usage 9.3mJ  avg ,97.66W  task <220>ypbind
    ref 4  usage 119.4mJ  avg ,293W  task <219>ypbind
  ref 3  usage 651.6mJ  avg ,1.953mW
    ref 3  usage 322mJ  avg ,878.9W  task <209>rpc.statd
  ref 3  usage 1.315J  avg ,3.711mW
    ref 3  usage 1.177J  avg ,3.711mW  task <204>klogd
  ref 3  usage 674.5mJ  avg ,2.051mW
    ref 3  usage 211.5mJ  avg ,390.6W  task <201>syslogd
  ref 3  usage 800J  task <140>(unknown)
  ref 3  usage 2.5mJ  avg ,293W  task <139>(unknown)
  ref 3  usage 115.3mJ  task <133>portmap
  ref 3  usage 168.7mJ  avg ,3.516mW  task <6>(unknown)
  ref 3  usage 0J  task <5>(unknown)
  ref 3  usage 2.769J  avg ,7.227mW  task <4>(unknown)
  ref 3  usage 400J  task <3>(unknown)
  ref 3  usage 0J  task <2>(unknown)
  ref 3  usage 175.2mJ  avg ,585.9W  task <1>init
  ref 7  usage 263.2J  avg 12.71W,12.58W  task <0>(unknown)
```

For each resource container one line is printed. Indentation is based on the hierarchy level. Reference counter is shown after `ref`, total resource usage after `usage`. Average values for both time slices are indicated by `avg` and the process or server(s) currently working on behalf of the resource container are displayed after `task` or `server`, respectively.

A.1.2 newrc

Creates a new level in the resource container hierarchy. This is achieved by cloning its resource container and changing the parent of its own resource container to this new one. Limits can be set in the parent container to change the amount of resources that may be consumed.

A program can be specified that will be run using this new resource binding. If no program was given, a shell will be executed.

As resource limits are configured in the parent container which is not propagated to the application, it is possible to effectively constrain the resource usage of the executed process(es).

A.1.3 measure

This utility is the resource-container equivalent to `time(1)`. It executes the specified program, creating a new parent resource container for it. Once the program exits, `measure` will print the resource usage accumulated in the resource container.

A.2 Library `librc`

A.2.1 rc_init

This function has to be called prior to other resource container operations. It obtains some information from the kernel and caches them. This way, the `rc_get_*` functions can be executed without invoking a system call.

A.2.2 rc_get_self

This function creates a new file descriptor and attaches the resource container which is owned by the process to it.

This new file descriptor is returned.

A.2.3 rc_get_nr_resources

Retrieves the number of different resource types supported.

A.2.4 rc_get_nr_time_slices

Retrieves the number of different time slices supported.

A.2.5 rc_get_resource_name

Retrieves the name of a given resource type (like “CPU”, “Energy”, etc.).

A.2.6 rc_get_resource_by_name

Opposite of `rc_get_resource_name`, returns the index of the given resource type. The name must match exactly.

A.2.7 rc_clone

Creates a new resource container by cloning an existing one. The new container is attached to a new file descriptor which is returned.

Resource limits of the original container are used to configure the new one. It shares all parent containers with its clone.

A.2.8 rc_no_limit

Removes limits set in one container. After calling this function, only limits configured in the parent container do apply.

This is useful after changing the parent of a container to be able to use the resources offered by the new one.

A.2.9 rc_set_parent

Moves one container in the hierarchy. It is removed from its old parent and will become a child of the new one. From now on, the container will receive resources from his new parent and is constrained by its resource limits.

A.2.10 rc_choose

Changes the resource binding of the process to contain the specified container. The old resource binding is discarded and this container will the process will own the new container from now on.

Resources used by the process will be accounted to the specified container and the process will be temporarily halted if the container runs out of resources.

A.2.11 rc_attach

Attaches a different container to a file descriptor. By specifying flags one can choose what to do with the file descriptor and resource container.

This low-level function is used to implement `rc_get_self`, `rc_clone` and `rc_set_parent`.

A.2.12 rc_resource_info

Obtains and changes information and parameters of a resource container. It can be used to set limits and retrieve usage statistics.

This low-level function is used to implement `rc_no_limit` and is used by `rc_init`.

Appendix B

Configuration

B.1 Resource Policy Database

B.1.1 RC target

This IPTables target will modify the resource container attached to a network packet. This way a central Resource Policy Database can be build using IPTables.

The resource container that is to be attached has to be specified using a file name that points to a resource container in a `rcfs` file system.

Further processing on that packet is done using the new resource container; if resources run out, packets are silently dropped. This can be used to reduce the bandwidth of incoming requests as TCP automatically adjusts it transmit bandwidth.

Example:

```
iptables -s 192.168.0.0/16 -j RC --set-rc /resources/internal
```

B.1.2 rc match

This match can be used to special-case packets based on their attached resource container.

This can be useful to be able to ignore the Resource Policy Database for packets that already have a resource container attached (for example the one propagated from the sender of the packet)

Example:

```
iptables -m rc --no-rc -j RETURN
```

B.2 CPU power consumption estimator

B.2.1 Model Specific Registers

Linux offers a special character device that can be used to directly access Model Specific Registers (MSR) of the CPU. Performance counters are one example of these MSRs and can be configured with this interface.

A small utility (`wrmsr`) has been build that can parse the symbolic name of the register and its intended value and configures the register accordingly.

The configuration used for measurements:

```
wrmsr MSR_IA32_TC_PRECISE_EVENT 0x0 0xFC00
wrmsr MSR_IA32_PEBS_ENABLE 0x0 0x1000001
wrmsr MSR_IA32_PEBS_MATRIX_VERT 0x0 0x1
wrmsr MSR_IA32_CRU_ESCR2 0x0 0x1000020C
wrmsr MSR_IA32_TBPU_ESCR0 0x0 0x8003C0C
wrmsr MSR_IA32_CRU_ESCR3 0x0 0x1200020C
wrmsr MSR_IA32_MOB_ESCR0 0x0 0x600740C
wrmsr MSR_IA32_FIRM_ESCR0 0x0 0x900000C
wrmsr MSR_IA32_CRU_ESCR0 0x0 0x600020C
wrmsr MSR_IA32_FSB_ESCR0 0x0 0x2600020C
wrmsr MSR_IA32_RAT_ESCR0 0x0 0x4000C0C
wrmsr MSR_IA32_MS_ESCR0 0x0 0x12000E0C
wrmsr MSR_IA32_IQ_CCCR1 0x0 0x3B000
wrmsr MSR_IA32_MS_CCCR1 0x0 0x35000
wrmsr MSR_IA32_IQ_CCCR2 0x0 0x3B000
wrmsr MSR_IA32_BPU_CCCR1 0x0 0x35000
wrmsr MSR_IA32_FLAME_CCCR0 0x0 0x33000
wrmsr MSR_IA32_IQ_CCCR0 0x0 0x39000
wrmsr MSR_IA32_BPU_CCCR0 0x0 0x3D000
wrmsr MSR_IA32_IQ_CCCR4 0x0 0x35000
wrmsr MSR_IA32_MS_CCCR0 0x0 0x31000
```

B.2.2 Linear combination

The `sysfs` file system is used in Linux to export per-device and per-driver information. The CPU device-class has been extended to include an interface to the energy accounting provider.

By writing to a special file, new MSRs can be added. Those will be included in future energy consumption of the machine. To be able to use symbolic names for the performance counters, a small utility has been written that just resolves the name adds that register to the energy consumption estimator. A weight given as float number is converted to the in-kernel integer representation at that time, too.

The following configuration was used:

```
count-msr MSR_IA32_TS_COUNTER 6.16699059
count-msr MSR_IA32_BPU_COUNTER0 7.11837835
count-msr MSR_IA32_IQ_COUNTER2 13.5536942
count-msr MSR_IA32_IQ_COUNTER1 1.73085536
count-msr MSR_IA32_IQ_COUNTER0 340.45833
count-msr MSR_IA32_BPU_COUNTER1 29.9555668
count-msr MSR_IA32_MS_COUNTER1 0.563939922
count-msr MSR_IA32_MS_COUNTER0 4.74693486
count-msr MSR_IA32_FLAME_COUNTER0 0.429620333
```

Appendix C

Getting the Source Code

Information and news about this work are available at `http://admingilde.org/~martin/rc/`.

A BitKeeper[1] repository holding the entire source code for both the modified kernel and the user-space utilities is available at `http://tali.bkbits.net/`.

All code is licensed under the GNU General Public License (GPL [11]). Use the source, Luke!

[1] *BitKeeper* – a Source Management System developed by BitMover Inc.

Bibliography

[1] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Measurement and Modeling of Computer Systems*, pages 90–101, 2000.

[2] Gaurav Banga, Peter Druschel, and Jeffrey Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating System Design and Implementation OSDI'1999*, Feb 1999.

[3] F. Bellosa. The case for event-driven energy accounting. Technical Report TR-I4-01-07, University of Erlangen, Department of Computer Science, June 2001.

[4] Frank Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Sep 2000.

[5] Edward G. Bradford. Runtime: Context switching. *IBM Developerworks*, July 2002.

[6] Todd Cignetti, Kirill Komarov, and Carla Ellis. Energy estimation tools for the palm. In *Proceedings of the 9th ACM Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems MSWiM 2000*, Aug 2000.

[7] Compaq, Intel, Microsoft, Phoenix, and Toshiba. *Advanced Configuration and Power Interface Specification 2.0a*, Mar 2002.

[8] S. Deering and R. Hinden. Internet protocol, version 6 (ipv6) specification. RFC 2460, Internet Engineering Task Force, December 1998.

[9] Peter Druschel and Gaurav Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Operating Systems Design and Implementation*, pages 261–275, 1996.

[10] Xiaobo Fan, Carla Ellis, and Alvin Lebeck. Interaction of power-aware memory systems and processor voltage scaling. In *Submitted for Publication*, October 2002.

[11] Free Software Foundation, Inc. GNU General Public License, June 1991. Version 2.

[12] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996.

[13] Intel Inc. *Model Specific Registers and Functions*.

[14] Nayeem Islam, Andreas L. Prodromidis, Mark S. Squillante, Ajei S. Gopal, and Liana L. Fong. Extensible resource management for cluster computing. In *International Conference on Distributed Computing Systems*, pages 0–, 1997.

[15] P. Karn and W. Simpson. Photuris: Session-key management protocol. RFC 2522, Internet Engineering Task Force, March 1999.

[16] Simon Kellner. Event-driven temperature control in operating systems. Studienarbeit, Friedrich Alexander Universität Erlangen-Nürnberg, 2003.

[17] Marlys Kohnke. Linux comprehensive system accounting (CSA), kernel design document. Technical report, SGI, Inc., March 2001.

[18] The Linux Kernel, `http://www.kernel.org/`.

[19] Rolf Neugebauer and Derek McAuley. Energy is just another resource: Energy accounting and energy pricing in the Nemesis OS. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, Schloss Elmau, Germany, May 2001.

[20] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[21] Dickon Reed. *The Nemesis Manual*, February 2000.

[22] John Regehr. Ph.d. proposal: Hierarchical loadable schedulers.

[23] John Regehr. Inferring scheduling behavior with hourglass. In *Proc. of the USENIX Annual Technical Conf. FREENIX Track*, pages 143–156, Monterey, CA, June 2002.

[24] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE.*, volume 63, pages 1278–1308, September 1975.

[25] SBS Implementors Forum. *Smart Battery Data Specification*, December 1998.

[26] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream control transmission protocol. RFC 2960, Internet Engineering Task Force, October 2000.

[27] Amin Vahdat, Alvin Lebeck, and Carla Ellis. Every joule is precious: A case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Sep 2000.

[28] Sam Watters. Linux process aggregates (PAGG), kernel design document. Technical report, SGI, Inc., June 2000.

[29] Andreas Weißel and Frank Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, October 2002.

[30] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Currentcy: Unifying policies for resource management. Technical Report CS-2002-09, Department of Computer Science, Duke University, 2002.

[31] Heng Zeng, Xiaobo Fan, Carla Ellis, Alvin Lebeck, and Amin Vahdat. Ecosystem: Managing energy as a first class operating system resource. Technical Report CS-2001-01, 2001.

Erfassung und Regelung der Leistungsaufnahme in energiebewussten Betriebssystemen

Eine wichtige Aufgabe von Betriebssystemen ist die gerechte Zuteilung von Ressourcen wie Energieverbrauch, Prozessorzeit, Haupt- und Massenspeicherzugriffen sowie Netzwerkdurchsatz. Um faire Entscheidungen über deren Nutzung fällen zu können ist es erst einmal nötig, den Verbrauch dieser Ressourcen exakt abzurechnen. Viele Betriebssysteme legen darauf leider noch keinen besonderen Wert.

Existierende Verfahren zum Abrechnen und Kontrollieren von Ressourcenverbrauch werden aufgezeigt und dabei besonders auf „Resource Container“ eingegangen. Diese ermöglichen es, Ressourcen unabhängig von Prozessgrenzen zu verwalten. Durch deren hierarchische Struktur kann sowohl der Verbrauch der kompletten Maschine als auch der einzelner Dienste oder Prozesse herausgefunden werden.

Bestehende Resource Container Systeme können erst mit speziell modifizierten Anwendungen ihre Vorteile wirklich ausspielen. Besonders die Zuordnung der Resource Container zu den einzelnen Prozessen ist Angelegenheit des einzelnen Programms. Dies erschwert den Einsatz solcher Systeme beträchtlich.

In dieser Arbeit wird ein auf Resource Containern aufbauendes Ressourcenmodell vorgestellt, dass einfach zu benutzen und trotzdem sehr mächtig ist. Ziel ist es, Ressourcennutzung immer dem eigentlichen Verursacher zuzuschreiben. Der häufige Einsatz von Diensten erschwert diese Zuordnung, denn die von ihnen verbrauchten Ressourcen kommen dem Auftraggeber zugute und somit sollte dieser auch dafür verantwortlich gemacht werden. Um die nötigen Modifikationen in Anwendungen möglichst gering zu halten, werden Methoden vorgestellt, um den Ressourcenverbrauch von Dienstprozessen automatisch dem richtigen Resource Container zuzuweisen. Es wird versucht, eine gute Standardkonfiguration bereitzustel-

len die von den Anwendungen bei Bedarf verändert werden kann.

Eine zentrale „Resource Policy Database“ ermöglicht es, einzelnen Aufträgen spezielle Resource Container zuzuweisen. Dazu wird der Paketfilter des Betriebssystems benutzt, um Anfragen aus dem Internet oder lokalen Netzen klassifizieren zu können. Befindet sich der Auftraggeber auf dem gleichen Rechner, kann dieser sogar direkt für die Ressourcennutzung des von ihm benutzten Dienstes verantwortlich gemacht werden.

Auch Stromverbrauch ist eine begrenzte Ressource und sollte vom Betriebssystem als solche gehandhabt werden. Durch die weiter voranschreitende Miniaturisierung und den steigenden Leistungshunger moderner Anwendungen wird Erfassung von Stromverbrauch in Zukunft immer wichtiger werden. Besonders der zu erwartende Maximalverbrauch eines Gerätes beeinflusst die Dimensionierung von Netzteil und Kühlung. Wenn der Spitzenverbrauch gesenkt werden kann, können bei mobilen Geräten bis hin zu Rechenzentren Kosten gespart werden.

Das beschriebene Ressourcenmodell wird benutzt um Energie abrechnen und kontrollieren zu können, damit ein gewählter Maximalverbrauch nicht überschritten wird. Zur Bestimmung des gegenwärtigen Stromverbrauchs werden „Performance Counter“ moderner Prozessoren hergenommen. Diese geben Aufschluss über die internen Abläufe des Prozessors. Eine Abschätzung des Stromverbrauchs wird aus einer Linearkombination von speziell ausgewählten Zählern gebildet.

Das Betriebssystem kann Einfluss auf die Aufteilung der verfügbaren Energie nehmen, indem es einzelne oder alle Prozesse kurzzeitig anhält wenn diese zu viel Leistungsaufnahme verursachen. Der Stromverbrauch vieler Rechnerkomponenten steht jedoch nicht unter der direkten Kontrolle des Betriebssystems. Dies muss bei der Vergabe von Ressourcen berücksichtigt werden; eventuell müssen einzelne Prozesse stärker beschnitten werden als eigentlich vorgesehen, damit es nicht zu einer Überschreitung des maximalen Energieverbrauchs führt.

Das beschriebene Ressourcenmodell wurde für das Betriebssystem Linux implementiert. Dabei wurde gezeigt, dass eine effektive Kontrolle des Stromverbrauchs bei nur geringfügiger Leistungseinbuße möglich ist.

In Zukunft kann das System noch weiter verbessert werden, indem zum Beispiel eine direkte Abrechnung der Ressourcen auf den Auftraggeber auch in Rechner Clustern ermöglicht wird. Auch müssen noch mehr Komponenten bei der Abschätzung des Stromverbrauchs berücksichtigt werden.