Diploma Thesis

# Cloning $L^4$Linux

Sebastian Sumpf

March 27, 2007

Technische Universität Dresden
Department of Computer Science

| | |
|---|---|
| Professor: | Prof. Dr. rer. nat. Hermann Härtig |
| Tutor: | Dipl.-Inf. Norman Feske |

## Statement

Hereby I declare that this work was produced in an autonomous fashion and that I did use the specified resources only.

Dresden, den March 27, 2007

Sebastian Sumpf

## Acknowledgments

Thanks to Krause for her extraordinary patience and the rest of the “outside world”!

# Contents

**1 Introduction** **1**
1.1 Virtualization classification . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Virtualization properties . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Comparison of isolation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.4 L$^4$Linux . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.5 Idea . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**2 State of the art** **6**
2.1 L3 – A persistent system . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2 L4 and persistence . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.3 Nomadic operating system . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.4 VMWare ESX Server . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.5 The Denali isolation kernel . . . . . . . . . . . . . . . . . . . . . . . . . . 13

**3 Design** **15**
3.1 Checkpointing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.1.1 Checkpointing preliminaries . . . . . . . . . . . . . . . . . . . . . . . 15
3.1.2 The frozen strategy . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2 Memory-resource sharing . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2.1 Copy-on-write design . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.3 Post checkpoint memory-resource sharing . . . . . . . . . . . . . . . . . . . 19
3.3.1 Partial hashing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

**4 Implementation** **24**
4.1 L$^4$Linux checkpointing – The details . . . . . . . . . . . . . . . . . . . . . 24
4.1.1 Saving the kernel state . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.1.2 Memory handling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.2 Cloning L$^4$Linux . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.3 The Freezer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.3.1 The hashing thread . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

**5 Evaluation** **34**
5.1 Cloning . . . 34
5.2 Isolation comparison . . . 35
5.3 Hash collisions . . . 36

**6 Outlook** **38**

**7 Summary** **40**

**Glossary** **41**

**Bibliography** **42**

# List of Figures

1.1 Memory sharing of $L^4$Linux clones . . . . . 5

2.1 Architecture overview . . . . . 8

3.1 Typical $L^4$Linux task and thread structure . . . . . 16
3.2 Freezer managed memory . . . . . 18
3.3 Example of a hash-level sample distribution. . . . . . 21
3.4 Lookup table search . . . . . 22

4.1 $L^4$Linux memory before checkpointing . . . . . 27
4.2 $L^4$Linux memory after checkpointing . . . . . 28
4.3 $L^4$Linux resume process . . . . . 29
4.4 Freezer usage scenario. . . . . . 30
4.5 Hash value layout . . . . . 31
4.6 Example of the search tree of Figure 3.4, showing the subtree for a hash key match in level 1 . . . . . 32
4.7 Page sharing . . . . . 32

5.1 $L^4$Linux suspend–resume latencies . . . . . 34
5.2 Comparison of isolation . . . . . 36
5.3 Page sharing . . . . . 37

## Abstract

Following the ongoing development of virtual machines, this paper deals with the “reincarnation” of virtual machines by examining their eligibility as isolation containers for single applications. During the course of the paper, we will take a closer look at additional performance costs and the required memory overhead necessary to enable the virtual machine L$^4$Linux to act as an isolation container.

# Chapter 1 Introduction

*"Nothing is as it seems."—Gerda Äagesdotter, Archmage of the Unseen*

Virtualization has regained a lot of attention during the last couple of years. Some even believe in the "reincarnation" of virtual machines [Ros04]. The term *virtual machine* (VM) dates back to the 1960s, it described a software abstraction imitating a system's hardware. Today (2007), the term compromises a vast amount of concepts and products reaching from the original definition to runtime interpreters like the Java-virtual machine or the "Common Language Infrastructure" of C#. Even todays microprocessor manufacturers offer VM support to be able to achieve a more transparent virtualization.

## 1.1 Virtualization classification

Because virtualization is implemented in software and, therefore, is located at some *layer* of a real-machines software stack, one can classify it by the location of this layer in the given software stack. In most publications this *virtualization* layer is referred to as *Virtual Machine Monitor* (VMM).[1].

**Hardware-level virtualization**. Here the virtualization layer is located right on top of the hardware. The exported virtual-machine abstraction may look and behave like the real hardware, thus enabling the software to run without modification on top (e.g., VMWare ESX Server [OH05], IBM VM/370). Alternatively, no hardware may be simulated. These VMs offer a special API that requires accommodation of the software running on top. This technique is generally called *paravirtualization*. The most notable representative is Xen [BDF$^+$03].

**Operating-system-level virtualization** describes a concept where the virtualization layer sits between the operating system and the application programs. The layer creates a so called *virtual environment*, wherein applications that are written for the

[1] Closely following [Ros04]

particular operating system are virtualized. Virtuosso [SWS06], FreeBSD Jails [Gun06], and Security-Enhanced Linux [LS01] are a few examples.

**High-level virtualization**. In this case, the virtualization layer sits as an application program on top of the operating system. There exist two types of virtualization. The first form exports an abstraction of a virtual machine and executes programs written for this VM. Java and Smalltalk are examples of this kind of VM. A completely different approach was introduced by the emulation VMMs. These VMMs emulate real machines in software, thus allowing to run unmodified software (i.e., operating systems). This approach is almost identical with hardware-level virtualization, but it implies a significant performance penalty. Examples are QEMU and Microsoft Virtual PC.

## 1.2 Virtualization properties

The described virtualization types share more or less the same set of properties.

**Software compatibility**. This term describes that all the software written for a specific virtual machine will be able to run in it. For example, an operating system running in a virtual machine may be required to accommodate parts of its code to be able to be executed in the VM in the first place. Running the same system in a different VM may demand no changes at all. Software compatibility is a kind of vague term and more a necessity than a property, but still worth noting.

**Isolation**. Virtual machines isolate programs running in them from other machines as well as real machines, the programs have no privileges outside their isolated domain. Additionally, the virtualization layer can provide performance isolation, this way resources consumed by one virtual machine do not necessarily harm the performance of other VMs.

**Encapsulation**. The virtualization layer can also by seen as a *level of indirection* that might be used to manipulate and control the execution of software running in the virtual machine. For example, VMWare and Microsoft Virtual Server use encapsulation techniques to package data and processing into a single object. Java takes advantage of encapsulation by enforcing runtime checks, thus providing a safe execution environment for Java applications.

**Performance**. Adding a layer to a system adds overhead as well. This affects the software running in the VM. The performance loss of the different types of VMs varies significantly.

## 1.3 Comparison of isolation

The previous section described basic classes and properties of virtualization. Next, we will take a closer look at the isolation property of (virtual) machines. We will limit our examination to three cases:

**Real machines**: An easy way to achieve isolation is to put everything that should be isolated on a separate real machine.

**Traditional virtual machines**: Programs are isolated by running them within an instance of a virtual machine. In most cases a complete operating system must be started. Hardware-level virtualization as well as high-level virtual-machine monitors belong to this group.

**Operating-system level**: Operating-system-level virtualization partitions a single real machine into multiple small computational partitions. The classic method of isolation here is *chroot* that basically creates an isolated directory. A newer and more sophisticated approach is SELinux (Security-Enhanced Linux), which implements the concept of *Mandatory Access control* [LS01] and is based on the “Flask” [SLH$^+$99] architecture of the NSA. All isolated instances share the same operating system.

Deploying real machines requires one physical machine for each isolation container, giving virtual machines and operating-system-level virtualization the advantage of the ability to host several isolation containers. Additionally, OS-level virtualization imposes less memory overhead than the use of VMs because one operating-system kernel is shared among all virtual environments, whereas VMs require one memory consuming OS kernel for each isolation container. OS-level virtualization is often criticized for its high administration overhead, thus raising costs for system administration. This virtualization type can also introduce the risk of an insecure system—if not configured properly.

Because of the ability of VMs and OS-level virtualization to host several guest systems, they are slow in comparison with an isolated system running at a single real machine, even though OS-level systems are generally faster than VMs because only one kernel is needed, whereas VMs must schedule one kernel per isolation environment. A high start-up latency is another VM drawback.

Isolation *granularity* measures the degree of a system that can be isolated without being ineffective (e.g., how efficiently does the isolating-system handle the isolation of a single application in comparison with a whole operating system environment). Operating-system mechanisms can be extremely small, resulting in a fine granularity and, therefore, a high amount of isolation containers can be provided by the virtualization system. The mentioned memory overhead introduced by VMs, limits the supported isolation granularity drastically. A typical system can host a few up to hundreds of virtual machines, depending on the underlying hardware, whereas a real machine offers just “one” isolation container, making it rather monolithic in terms of granularity.

When it comes to security aspects of isolation, physically separated real machines are the state of the art. OS-level virtualization suffer from the fact that the isolation environment completely relies upon the isolation policies provided by the kernel (e.g., compare *chroot* on a vanilla kernel with SELinux). The source code complexity of an OS kernel is in comparison with a virtual-machine monitor (thin layer of software) immense and, therefore, OS-kernels are hard to develop secure. An often neglected fact is the complexity of interfaces (to the “outside” world) provided by VMs, which lead to a weak isolation when overlooked.

Taking a closer look at Table 1.1, one notices mediocre overall results for virtual machines. Our vision is to improve upon the virtual-machine concept with the goal to make them more attractive for fine-grained isolation of single applications.

| | **Real machines** | **Virtual machines** | **Operating-system level** |
|---|---|---|---|
| **Cost** | High | Medium | Low |
| **Performance** | High | Medium to low | Medium to high |
| **Granularity** | Coarse | Medium | Fine |
| **Security** | High | Medium | High |

***Table 1.1:*** Summary of isolation properties ([Cow06])

**Goals**:

1. Decrease the **costs** of VMs by reducing the memory overhead per virtual-machine instance (resource sharing).
2. Decrease the start-up latency of a single virtual-machine instance (**performance**).
3. Increase the **granularity** of virtual machines. It should be possible to efficiently isolate a single application using virtual machines.

## 1.4 L$^4$Linux

The system we will use to demonstrate our concepts is L$^4$Linux([Hoh96], [Lac02]) on top of the "Dresden Real-time Operating System", in short DROPS [HBB$^+$98]. L$^4$Linux is a port of Linux to the L4 microkernel *Fiasco* and is executed as a unprivileged user-space task. It provides a paravirtualized machine with software compatibility for native Linux applications (in our terminology, these applications are called *legacy* applications).

The performance penalty introduced by L$^4$Linux is in the range from 2 % to 10 % in comparison with native Linux (AIM multiuser benchmark suite VII [HR06], [HHL$^+$97]).

## 1.5 Idea

Our fundamental idea is to boot a full grown L$^4$Linux prototype, including X11 if necessary. We will then checkpoint this prototype, resulting in a complete image of the once running operating system, in short a *frozen* image. It now is possible to create self-sufficient L$^4$Linux instances out of the frozen image. This procedure should be notably faster than starting an "out of the box" L$^4$Linux system. Each instance can then be used to execute different applications in an isolated way that is as strong as VM-based isolation. Multiple L$^4$Linux instances will share their memory using copy-on-write (COW) techniques with the frozen image as well as among each other (Figure 1.1).

With a code complexity of both the L4 microkernel and basic services of less than 50.000 lines of code, the DROPS architecture establishes a *Trusted Computing Base* and is able to provide strong isolation for multiple L$^4$Linux instances.

We will quantify the success of this project as follows:

- How much memory overhead do we need to create a running clone out of the frozen image?

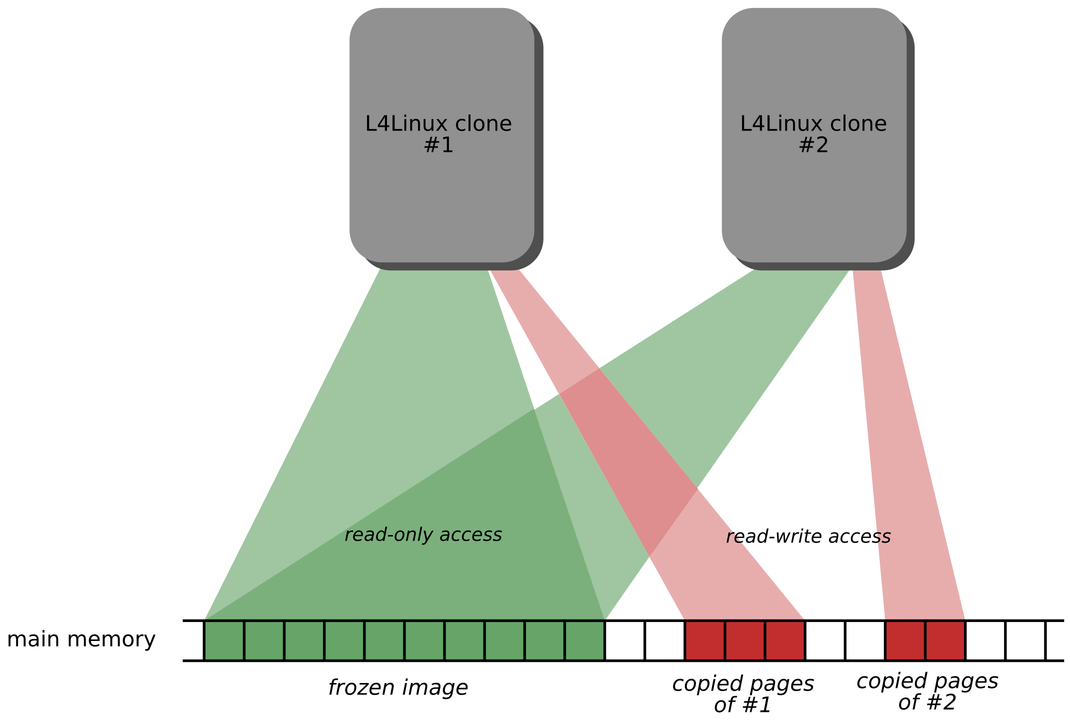

***Figure 1.1:*** Memory sharing of L$^4$Linux clones

- How far can we scale down the startup latency of a L$^4$Linux clone?

If these two parameters are comparable to application-startup time and memory consumption by the application itself, the L$^4$Linux-isolation granularity is in fact increased to application level.

# Chapter 2

# State of the art

Checkpointing and in the broadest sense persistent system object stores have a long history of research in computer science. Because one objective of this project is to checkpoint a running virtualized operating system, we now will take a closer look at checkpoint creation. $L^4$Linux is executed by an L4-microkernel; therefore, concepts of L4 and its persistent predecessor L3 will be described next. Nomadic OS [HH02] is a project whose aim is to provide the capability of operating-system migration for the $L^4$Linux-2.2 kernel. As a component of the project, a $L^4$Linux checkpoint had to be provided.

Memory-resource sharing in the VMWare ESX server [OH05] is shown afterwards. The final section describes the Denali isolation kernel, a working approach toward application-level isolation.

## 2.1 L3 – A persistent system

L3 [Lie93] is/was actually used commercially and, therefore, in real use. The first systems were shipped in 1989 and may still be in use today. "Everything is persistent[1]" is the basic assumption of L3, reasoning that persistence is defined by the lifetime of an object. This includes tasks or processes also (i.e., there are files that only last a few seconds and programs that run for weeks). For the reader not familiar with the L3 task concept and because this applies to L4 tasks as well, here some basic terminology: In L3, a task consists of:

- At least one *thread*, where a thread is defined as a running program.
- *Dataspaces*, which are virtual-memory objects. Only in L3, dataspaces are always persistent.
- One *address space*, where dataspaces are mapped into dynamically.

[1] "Everything but device drivers is persistent" (hardware related reasons).

Tasks communicate through the means of “Inter-Process Communication” (IPC), the performance of IPCs has a crucial influence on the system’s overall performance.

The outstanding concept of L3 is the handling of processes as “first class objects”. Tasks maintain data objects and control access to them. This implies that there are no data objects outside of a process. An object not owned by a task will no longer exist. So, each data object in L3 (e.g., files, databases) is implemented as a persistent tasks containing persistent data and persistent threads, giving the advantage of reduced kernel complexity and a higher degree of flexibility. A global naming scheme to locate objects is not necessarily needed because unique task identifiers are sufficient[2].

Data objects in L3 are managed either by the default or external pagers [3] and can be mapped into address spaces. Therefore, they are called dataspaces. One key component of L3 is the default pager, which implements persistence. To checkpoint the complete system, L3 introduces the notion of a *fixpoint*, a checkpoint covering the complete system, including a consistent copy of all tasks, thread, address spaces, and dataspaces taken at the very same point in time. To implement a fixpoint, all dataspaces of all tasks and all thread–task control blocks are integrated into the “dataspace of all dataspaces” at the persistent pager. This is done in a way that lazy copying can be applied to the dirty pages using copy-on-write techniques to not disturb the system performance too much.

**Benefits**:

- There is only one general persistence mechanism for files, programs, etc.
- No run-time overhead is incurred accessing persistent data.
- Program and system checkpoints are easy to implement.

For our intent, checkpointing L$^4$Linux, L3 would be the system of choice. Every fixpoint contains a consistent image of L$^4$Linux, which could be used as a basis to create a cloned version from. Of course, an external copy-on-write pager must be implemented, handling the resource sharing between the different L$^4$Linux clones. This is solvable because L3 supports “stacking” of pagers. Unfortunately, L3 is a little outdated[4] and its successor L4 was not designed as a persistent system from the start. So lets have a look at L4 and persistence next.

## 2.2 L4 and persistence

L4 is a second generation microkernel. One of the fundamental experiences with L3 was the realization that microkernels are processor dependent and are inherently not portable. In [Lie95] it is stated that “even such compatible processors as the 486 and the Pentium need different microkernel implementations (with the same API)—not only different coding but different algorithms and data structures”. For L4 the same machine

[2] This concepts is also known as *orthogonal persistence*

[3] Pagers are in this context user-level tasks.

[4] (and L$^4$Linux is not L$^3$Linux)

abstractions as in L3 apply, namely: address spaces, threads and IPC. Only seven system calls are implemented. Also, all address-space policies were removed from the kernel, which now provides a pure mechanism interface. L4 supports the recursive construction of address spaces outside the kernel, where the initial address space represents the physical memory and is controlled by the first memory manager $\sigma_0$. Pagers may be integrated with a memory manager or use a memory-managing server. For constructing further address spaces on top of $\sigma_0$, L4 provides three operations[5]:

- **Grant**: The owner of an address space may *grant* any of its pages to another space. The page is then removed from the granter's address space and included in the address space of the recipient.
- **Map**: The owner of an address space can *map* any of its pages into another address space. Afterwards, the page can be accessed in **both** address spaces.
- **Flush/Unmap**: The owner of an address space can flush any of its pages. The flushed page remains accessible in the flusher's address space, but is removed from all other address spaces, which received the page (directly or indirectly) from the flusher.

To be able to manage granted and mapped physical pages, the kernel maintains a mapping database. For further reading, please refer to [ALE+00].

As stated earlier, L4 was not designed with persistence in mind but it is possible to checkpoint the system transparently through user-level pagers. [SCL01] describes a way how to achieve this [6] using a checkpointing server, which is implemented as a L4 user-level pager.

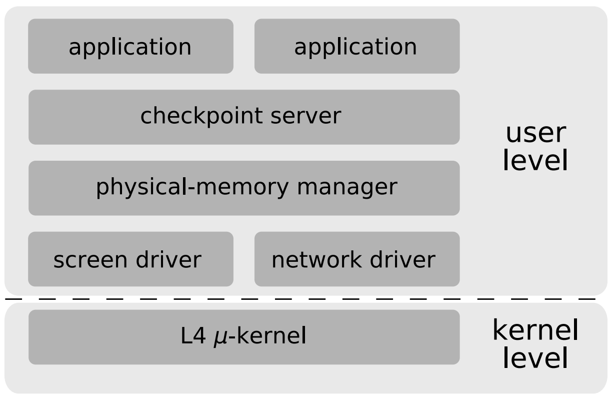

***Figure 2.1:*** Architecture overview

[5] These terms appear throughout this paper and are, therefore, described here.
[6] As in L3 drivers are not subject to checkpointing

Two checkpointing problems will be described next: *kernel checkpointing* and *main-memory checkpointing.*

### Kernel checkpointing

Interestingly enough, the L4 memory-management concept allows the checkpointing server to act as a pager for thread-control blocks (TCBs). Recall that physical page frames are not owned by the kernel, but by $\sigma_0$ . This raises the opportunity to save the state of various threads, by a simple flush of the relevant page frames at regular intervals. On next access of a TCB, the page containing the affected TCB is simply copied (COW technique), resulting in a consistent copy (or *fixpoint*) of the needed system state.

Even more interesting is the fact that the L4 design requires no other kernel state than that contained in TCBs:

- Persistent applications can make no assumptions about the ordering of in-kernel ready queues and IPC waiting queues. The kernel can, therefore, upon recovery reconstruct the queues in any order it prefers.
- Page tables and the mapping database are an integral part of the microkernel; these data structures are not subject to checkpointing. In consequence both structure will be empty upon the recovery of a task. Once a persistent task is restarted, a page fault will be raised, the kernel redirects this page fault to the task's pager. The pager, whose page tables are also empty, will in turn generate another page fault, until it subsequently reaches the checkpointing server, which is able to handle the page fault and maps the affected page into the faulting task's address space.
- [SCL01] states that other data structures have no relevance.

### Main-memory checkpointing

The checkpointing server can be easily expanded as a pager to all physical memory by locating it right above the physical-memory manager $\sigma_0$ (Figure 2.1). Being almost on top of the pager hierarchy, it automatically gains access to all user-level memory. This enables the checkpointing server to temporarily flush pages out of the user tasks address space as well as the ability to map pages read only, thus exploiting the ability to checkpoint the main memory using the same methods (COW, lazy copying) described in *kernel checkpointing* (Section 2.2). After unmapping a page, the page can safely be transfered to backup storage or to to any place, where fixpoints are stored at. The whole mechanism is heavily influenced by L3, EROS [SSF99], and KeyKOS[7] [Lan92].

Again, this could be an opportunity to checkpoint L$^4$Linux by retrieving all necessary information (frozen image) from the checkpointing server. But the DROPS environment currently does not provide a checkpointing server, for it specializes in real-time applications, where regular checkpointing intervals will decrease the ability to

[7] Not described in this paper.

make predictions on execution times. Therefore, we will now examine a more specialized approach.

## 2.3 Nomadic operating system

Nomadic OS [HH02] emerged out of the understanding that the current trend in distributed computing (e.g., clusters, or grids), which focuses on the migration of processes[8], may be insufficient, especially when dealing with loosely connected networks (as opposed to tightly connected networks, running long non-I/O intensive calculations—a typical cluster setup). [HH02] reasons about the large amount of dependencies processes are demanding from the running environment. Open files, memory shared with other processes, reliance on access to various host specific system features, are just a few examples. On the other hand, operating systems are stated to require only access to certain well defined hardware interfaces, making them more simple to migrate. For these reasons, Nomadic OS assumes that operating systems instead of just its processes, should be the unit of migration. To be more precise, the operating systems to be migrated, are running on top of an operating system (virtual-machine monitor), so actually, virtualized machines are subject to migration.

As the environment to demonstrate the research, L$^4$Linux-2.2 on top of an L4 microkernel was chosen. In L4, a lot of features usually implemented in the kernel (monolithic kernels) are handled by user-level programs, making it simpler to execute multiple operating systems on top of the microkernel.

The essential tasks of Nomadic OS follow:

- **Checkpoint** a running L$^4$Linux instance.
- **Transfer** the resulting operating-system (frozen) image over the network to a different host node.
- **Resume** L$^4$Linux operation, using the transfered image at the new host.

*Checkpoint* and *resume* operation will be examined next, for network image *transfer* is not the intention of this project.

### Checkpointing

Because the L4 environment currently does not support transparent checkpointing, *main-memory checkpointing* of L$^4$Linux is achieved in a similar way as described in Section 2.2. A user-level pager is integrated "between" L$^4$Linux and its default memory manager. Thus, all memory requests of L$^4$Linux are processed by the "nomadic" pager, giving it the opportunity to freeze all L$^4$Linux memory when required.

*Kernel checkpointing* is a little more complicated. Recall from Section 2.2 that the only thing needed to checkpoint the kernel state of a task–thread are its thread control block (TCB). The TCB can be obtained by a pager that handles TCB memory requests from

[8] See MOSIX [BL98] as a well known, example.

the kernel. While this solution is sensible for global transparent system checkpointing, it should not be used for a specialized approach like the Nomadic OS (this pager would then have access to the TCBs of all tasks in the system). For this reason, Nomadic OS saves the kernel state of $L^4$Linux in itself[9], by sending a special signal to $L^4$Linux, which causes it to save the internal kernel state of its threads (registers, stack pointer, and instruction pointer). It then stores the state in memory (which is paged by the nomadic pager). Additionally, all $L^4$Linux threads are tricked into suspension code, causing the whole operating system to cease operation, resulting in a consistent frozen image at the nomadic pager. The last running $L^4$Linux thread sends its resumption address to the Nomadic OS system.

### Resume operation

To resume[10] $L^4$Linux, Nomadic OS creates a task that begins execution at the previously received resumption address. The first thread of this task recreates all $L^4$Linux threads and user tasks. Each thread and user task then restores its previous state from the memory where it was saved at checkpoint time and resumes normal operation.

Please note that each memory access of the newly created threads and tasks will incur a page fault, which will be handled by the nomadic pager, and in turn repopulates the microkernels mapping database (Section 2.2). The reader should also be aware of the fact that $L^4$Linux threads and user tasks cannot be resumed in any arbitrary order (e.g., kernel threads must be resumed before user tasks), which introduces the necessity of synchronization during the resume operation.

Nomadic OS provides a feasible solution of the checkpoint and resume problem for a virtually running operating system, without the need of much environmental support. The procedure described here may seem very specific to the L4 environment at first sight, but a similar approach was used by one of the authors to implement operating-system migration in XENs VMM[HJ04], again using a modified Linux. As in Nomadic OS, Linux saves its own internal state, leaving the memory checkpointing to the VMM. This leads us to the assumption that checkpointing and resuming virtual-machine instances may (for different purposes) be possible at other virtual-machine monitors.

Section 2.1 to 2.3 described three different checkpointing strategies. Memory-resource sharing is another aim of this project (Section 1.3). One possible sharing strategy is shown in the following section.

## 2.4 VMWare ESX Server

Both the VMWare ESX server and its predecessor Disco[BDGR97] provide mechanisms to share memory resources. Because Disco concentrates on large scale CC-NUMA architectures and mostly deals with features like cache coherency or locality, we will content

[9] Note: This requires source code changes
[10] Simplified

ourselves with the description of memory resource sharing of the VMWare ESX Server as given in [Wal02], omitting any other technical details.

As stated in Section 1.1, the VMWare ESX Server is a proprietary hardware-level virtual-machine monitor, which is based on a kernel created by VMWare itself. Therefore, the server runs directly on top of the hardware, currently virtualizing the IA-32 architecture. It is in production use and able to host unmodified operating systems like Linux or Windows.

### Memory sharing

The ESX Server supports *overcommitment*[11] to achieve a higher degree of server consolidation. Therefore, a memory sharing strategy is required. Justification of sharing memory lies in the assumption that multiple VMs may be running instances of the same operating system and–or have the same applications or components loaded.

Elimination of redundant copies of pages (e.g., code or read-only data) is the core task of every memory-sharing strategy. To achieve this necessity, the ESX Server implements an additional page-address translation to be able to give each VM the illusion of utter physical-memory access. While a *machine address* refers to actual hardware memory, a *physical address* is a software abstraction providing the illusion of hardware memory to the VM. A mapping of physical to machine page frames is maintained by the ESX server, enabling it to map one machine page to multiple physical pages.

To be able to eliminate redundant page copies, one has to identify them in the first place. The approach chosen by the ESX Server is called "content-based page sharing", which basically assumes that "Pages with identical contents can be shared regardless of when, where, or how those contents were generated". This assumption first, eliminates the need to hook or even understand guest code and second, it provides more opportunities for sharing, for all potentially shareable pages can be identified by their "contents".

This way, the ESX Server has to implement a page comparing scenario. Because naive page matching would require an expense of $O(n^2)$, hashing is used to identify pages with potentially identical contents more efficiently. A hash value for a page summarizes the contents of the same and is used as a lookup key into a hash table containing pages already shared (or in this case marked copy-on-write). If a matching key is found, the identified pages are subject to a full comparison (being aware of possible hash collisions). On success, the found page can be shared with the compared page using COW techniques, the memory of the redundant copy is freed and, therefore, reclaimable. An attempt to write to the shared page will of course result in a fault, causing the creation of a private copy of the shared page.

Note, if no matching key is found in the hash table, the page is not marked as shared or COW, but as a *hint* page. On any future match with another page, the contents of the hint page are rehashed. If the hash is still valid, a full comparison is performed, and the pages are shared on success. If not, the stale hint is removed. This behavior

[11] The total size of memory configured for all virtual machines exceeds the total amount of actual machine memory

describes a method to identify possible read-only pages (i.e., if the hash stayed the same, it may be a read-only page).

Because it is an interesting concept and because it may be useful, here an “as short a possible” description of the ESX Server’s *ballooning* concept.

### Ballooning basics

The purpose of the ballooning concept is the ability to influence the internal memory usage of a guest-operating system externally[12]. This is useful on overcommit situations when free machine memory is rare. In this case, the memory usage of the guest OS is increased (externally), which in turn will result in the inability of large memory requests by the guest OS (because it assumes its memory is exhausted). If the machine’s memory usage drops, the balloon can be deflated, resulting in a drop of the guest OS memory usage as well.

Remember? Our principal aim is to provide a finer granularity for VM isolation (Section 1.2). The next and last system described in this chapter pursues the very same target (although in a more specialized way).

## 2.5 The Denali isolation kernel

As the title suggests, Denali’s purpose is to isolate something, namely Internet services, thus relieving Internet service authors from “the burden of acquiring and maintaining physical infrastructure”. The system is required to be both *scalable* and *secure* (i.e., provide strong isolation). To gain performance, Denali (again) takes advantage of the hardware-level virtualization approach, but this time having scalability and simplicity in mind, what leads Denali to a form of paravirtualization. Denali’s intention is to run a thin layer of software (small-kernel design) directly on x86 hardware. For the sake of performance, Denali does not emulate the complete architecture of a physical machine, but instead exposes an own instruction set similar to x86. The cost of this is, as in any paravirtualized system, the requirement of adaption by the guest-operating system to the host environment (VMM).

Denali’s setup is strongly related to the setup we use. While our system consists of an adapted version of Linux on top of a small microkernel (12K), Denali implements its own lightweight OS called *Ilwaco* on top of the isolation kernel to be able to demonstrate the completeness of its architecture. The main difference here is that Denali was designed as a virtual-machine monitor and, therefore, is focused on this task. L4 and its environment, on the other hand, provide a complete operating-system infrastructure and are open to any type of application. Here L$^4$Linux is just another user task.

### CPU virtualization

Denali claims to use standard multiprogramming techniques to multiplex the CPU across virtual machines. The kernel maintains per-VM thread structures, containing the kernel

[12] Note, this requires a patch of the OS or an additional kernel module

stack, register memory, and thread control structures. CPU multiplexing is handled by a *gatekeeper policy*, enforcing admission control by choosing a subset of active machines to admit into the system and a simple *scheduler policy* that controls context switches in round-robin order. Because Denali is based on the x86 architecture, standard compilers, linkers, etc. can be used.

### Memory management

Each Denali VM is given its own virtual-address space (4GB). But a single VM can only access a subset of this address space. The kernel itself is mapped into a protected (not accessible) portion of the address space, to avoid physical TLB flushes on VM/VMM crossings. Each VM also has a *swap region* allocated during machine startup by the kernel, which is large enough to hold the entire VM-visible address space. Swap regions are used to page out portions of the VM's address space. This way, the system periodically redistributes physical memory from inactive to active VMs. For this to work, a page replacement policy has to be provided and incorporated into the kernel[13].

Denali is an attempt to reach application-level isolation. To achieve this, an entire kernel was designed from scratch, justifying the somewhat "basic" nature of some of its policies. Nonetheless, a scalable and high performance system is provided with a (claimed) potential of running more than 10.000 virtual machines on commodity hardware.

[13] A CLOCK algorithm derivative has been implemented

# Chapter 3

# Design

This chapter will give you an overview of the solution we propose to increase the isolation granularity of L$^4$Linux applications. Sometimes, there are multiple solutions to a specific problem available.

Our idea (Section 1.5) imposes two major design challenges: First, a way to checkpoint (freeze) a running L$^4$Linux instance must be provided—subject to the first part of this chapter. Second, an efficient memory-resource-sharing strategy has to be developed and will be described through the rest of the chapter.

## 3.1 Checkpointing

Figure 3.1 illustrates the typical structure of a running L$^4$Linux system. This system consists of the L$^4$Linux task itself, which is a user task of the L4 system (L4 task). The L$^4$Linux-server task contains various threads, where the *region mapper* and *semaphore* threads are part of the *L4Env*[srg03]—the environment L4 tasks are executed in—the others are Linux kernel threads. The Linux kernel code is executed by the L$^4$Linux-server thread. Unmodified Linux applications are created by this thread. These applications are executed in separate L4 tasks, thus providing them with their own address space (Section 2.1). The recursive address-space model of L4 (Section 2.2) enables the L$^4$Linux-server thread to act as the pager of its user tasks (respectively Linux applications), while managing its main memory transparently.

### 3.1.1 Checkpointing preliminaries

Chapter 2 illustrated two checkpointing strategies applicable to the L4-based system (Sections 2.2 and 2.3). For our purposes, we will consider *memory checkpointing* and *kernel-state checkpointing* (omitting *backing-store checkpointing*). The memory-checkpointing concept in both Nomadic OS and the generalized L4 approach are similar. An external pager is placed below the pager hierarchy of L$^4$Linux (*external pager* arrow in Figure 3.1), which gains control of the memory used by L$^4$Linux. In the L4 approach, this pager was called *checkpointing server*. It additionally gained control of the memory

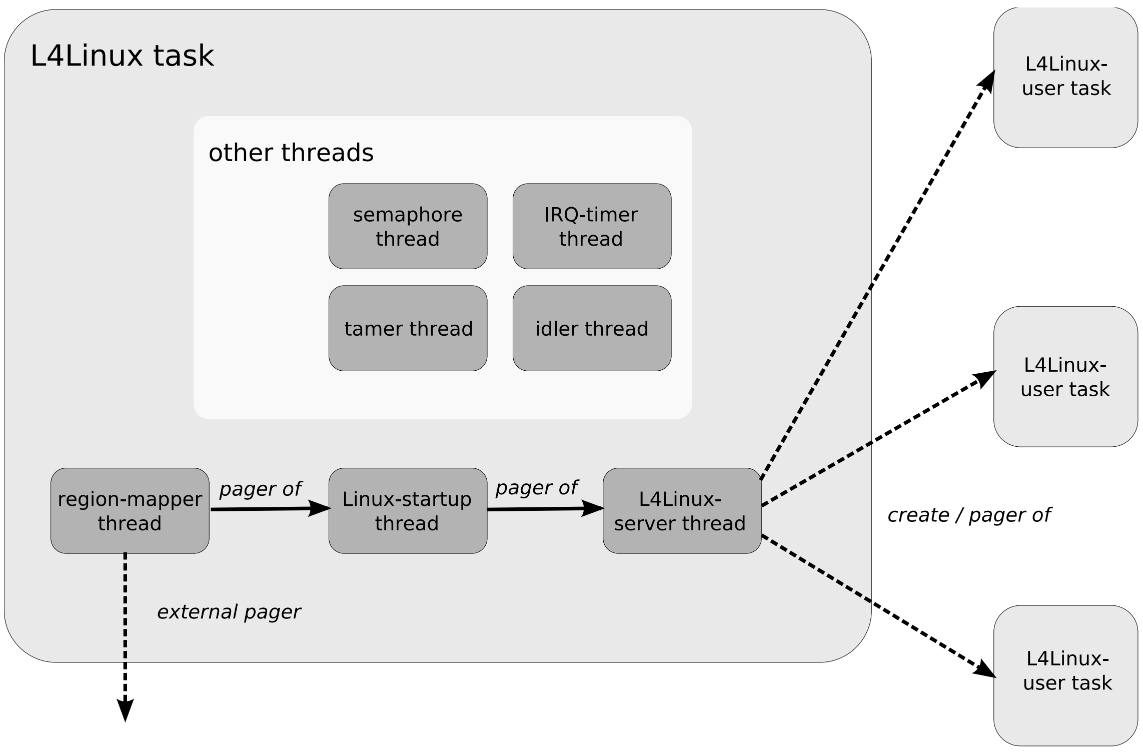

***Figure 3.1:*** Typical L$^4$Linux task and thread structure

of the user task besides L$^4$Linux. *Nomadic pager* is the term that we used for the same concept in Nomadic OS. Note that this pager has control over the L$^4$Linux main memory **only**.

Differences arise when it comes to kernel checkpointing. Recall that in the L4 approach, the checkpointing server also pages the TCB memory used by the kernel, thus enabling it to save the state of any user task in the system. Nomadic OS, in contrast lets L$^4$Linux save its kernel state internally in a nontransparent manner.

Whereas the L4 approach is a more generalized and transparent way to checkpoint user tasks, it also involves new problems. First, it is not there, yet[1]. Second, there exist the security issues described in Section 2.2. It seems out of question to implement a checkpointing server, which manages the memory of **all** TCBs in the system, just for the purpose of creating a L$^4$Linux image.

Because Drops (Section 2.2) is by design a real-time system, real-time tasks should be omitted from checkpointing because it complicates execution-time predictions. This implies the need for the microkernel to determine, which memory to use for TCBs for a certain type of application. For example, there could be one pool for real-time application and one for checkpointed applications (i.e., the checkpointing server). Changes to the microkernel are required to implement this approach.

[1] Not essential for our decision.

For these reasons, our checkpointing strategy will stick more closely to the one of Nomadic OS, mostly in the sense that the kernel state of L$^4$Linux is determined and restored by L$^4$Linux itself. We will refer to our custom pager [2] that features the checkpointing facility as *Freezer*.

### 3.1.2 The frozen strategy

Freezer is, unlike in Nomadic OS, not the pager of L$^4$Linux from the beginning. It rather represents a somewhat dual nature. At startup time the pager resides in the system waiting to be used. On checkpointing time, the Freezer takes over the memory management of L$^4$Linux (*container*). That is: L$^4$Linux copies its main memory to the Freezer task. From this memory image, now owned by the Freezer, new L$^4$Linux instances can be created. Please notice that this memory image also contains the state of the L$^4$Linux-server at checkpointing time. The Freezer pages freshly created Linux instances only.

Our design implies that kernel-state checkpointing as well as memory checkpointing is triggered by L$^4$Linux itself. This design is necessary because virtual-address spaces provide various semantics (see Section 4.1.2)—introduced with the development of the L4Env[3].

## 3.2 Memory-resource sharing

As mentioned in Section 1.5, we are now able to create self-sufficient L$^4$Linux instances using the memory image stored in the Freezer. All L$^4$Linux instances are resumed using the same checkpoint. Therefore, we assume a high page-sharing potential between freshly created L$^4$Linux instances. [Lie93] states that on average, one third of main memory contains dirty (read-write) page frames, supporting our page-sharing assumption. We will exploit this property using copy-on-write techniques. Each L$^4$Linux instance shares its read-only pages with the frozen image, whereas a write request will implicate a copy operation of the "frozen" page. The writing L$^4$Linux instance is then the exclusive owner of the freshly created page copy.

### 3.2.1 Copy-on-write design

The Freezer must be able to quickly locate and copy page frames. In L4, physical pages are identified by mapping them to virtual addresses.

An efficient way to manage page mappings is partitioning. Page tables and trees are examples of this concept. We decided to use trees (Figure 3.2).

Freezer-managed memory is partitioned by L$^4$Linux instances and page mappings. Note that the Freezer does not map virtual to physical addresses, but external virtual-addresses to its own virtual-address space (recursive address-space construction). In Figure 3.2, *Frozen* contains the previously created checkpoint (frozen image). All read requests from the running L$^4$Linux instances are satisfied using the frozen image. On

[2] Other terms will be added later on.

[3] The L4Env was not used by L$^4$Linux-2.2 and, therefore, of no concern to the Nomadic OS

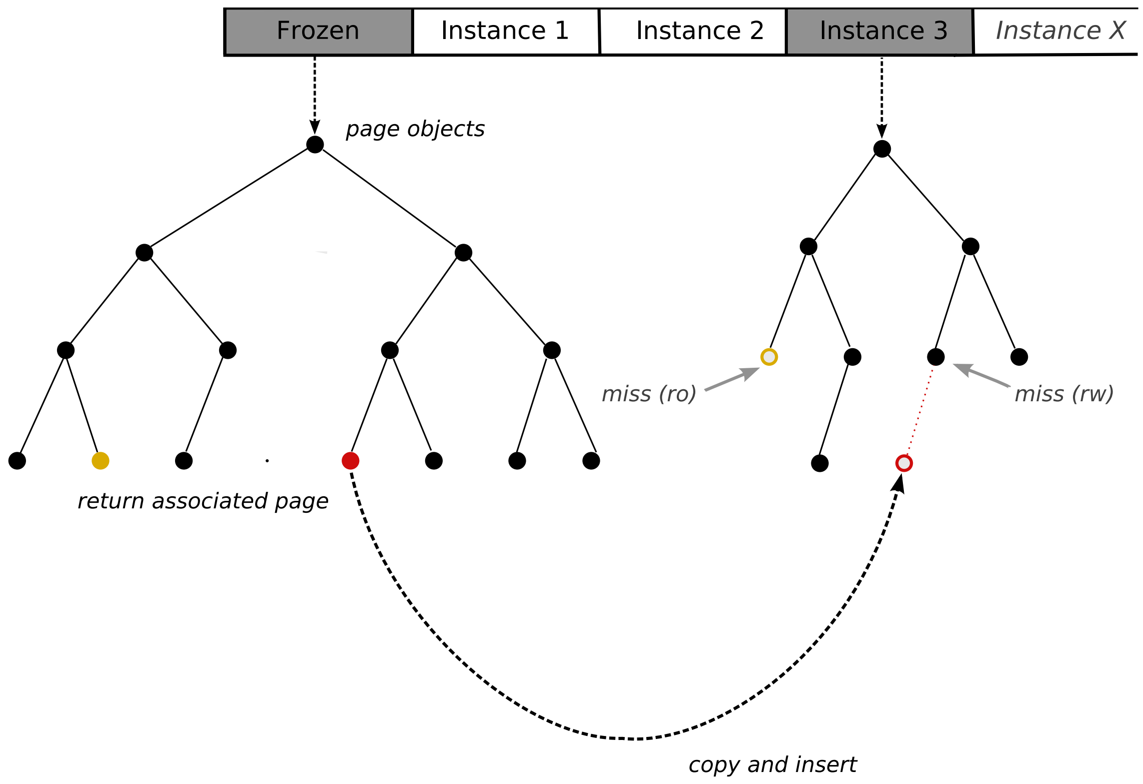

***Figure 3.2:*** Freezer managed memory

a read only page request, the pages of the faulting L$^4$Linux instance are searched, if the faulting address cannot be located the corresponding “frozen” page is returned (yellow circles in Fig. 3.2). If this behavior happens on during a write request, the corresponding “frozen” page is copied and its faulting address is added to the appropriate L$^4$Linux instance (red circles).

Each L$^4$Linux instance manages the difference to the “frozen” image (i.e., modified pages) only. We chose this design to save as much memory as possible. Alternatively, we could logically copy the complete management structures of the “frozen” image during L$^4$Linux clone startup. In Figure 3.2, “Frozen” and “Instance 3” would then be identical. All virtual addresses of the new instance are then marked as copy-on-write. Now, a write request would cause a simple copy operation of the underlying page. If assuming the mentioned one third of dirty pages in the system, an average of two thirds of the management structures of a single L$^4$Linux instance would be redundant.

Our approach is reasonable because the Freezer is designed for the purpose of application isolation. It is **not** exclusively a checkpointing server. A checkpointing server supports checkpoint creation for multiple applications at regular intervals. Different checkpoints could share their underlying memory. For such a system, it is useful to have an overview of the complete logical memory structure at each checkpoint because checkpoints may be freed, implying the need to identify pages that can–cannot be released. Because the Freezer currently supports just one checkpoint, this behavior is not a requirement.

The described behavior can also be implemented using page tables. Page tables have notable advantages over the proposed tree structure. Searching a page-table entry requires constant-time overhead, whereas binary (in our case AVL) tree search operations have an upper bound of $O(\ln n)$. Inserting elements into a binary tree may trigger unpredictable balancing operations. This never occurs when using page tables. The vital reason we decided to use trees, can be found in the fact that the Freezer manages the difference to the “frozen” image. Because every L$^4$Linux instance only manages modified Freezer pages, page tables may be sparsely occupied, resulting in a waste of memory. Whether this advantage of trees outweighs their performance penalties remains undetermined and may be subject to further research (Section 6).

A weakness of our memory-sharing strategy is the fact that each L$^4$Linux clone will subsequently perform write operations in its managed main memory, for example applications have to be loaded, user-level memory requests must be satisfied, caches are maintained, .... Therefore, the memory contents of the “frozen” image and any two running L$^4$Linux instance will diverge over time. In Figure 3.2 this behavior leads to a (possibly steady) growth of the page tree of “Instance 3”. To counter this divergence effect, we will introduce an additional memory-resource-sharing strategy next.

## 3.3 Post checkpoint memory-resource sharing

The memory-sharing potential of any two L$^4$Linux instances heavily depends upon the workload—specific to the system. A high sharing potential can be assumed for the following scenarios:

- Applications that are executed at any two L$^4$Linux instances
- Frequently used shared libraries (e.g., glibc)
- Freed L$^4$Linux main memory
- The Linux kernel including any ramdisks (e.g., initrd)

The key concept in Section 2.4 was called *content-based page sharing*, which assumed that “each” page in a system can be shared and its shareability is only defined by the “contents” of a page. This approach manifests itself in its general nature. For example, if a single L$^4$Linux instance executes multiple instances of the same application, distinct memory pages will hold identical data. We say that the contents of these pages “converge” and thus, can be stored on one shared memory page.

In Section 2.4, hashing was used for the purpose of comparing page contents. This strategy seems natural because complete page comparisons can be delayed until a matching hash key is located. If the hash keys match, a full page comparison is still necessary because the calculated hash value might as well be a hash collision. A good hash function tries to minimize the probability of hash collisions, distributing hash values uniformly for its given input *universe* $\mathcal{U}$ (usually not uniformly distributed). Additionally, our system requires that the hash function should be faster than $O(n^2)$—the expense of native comparison.

We decided to use Jenkins' hash function[4][Jer97], which offers a time-tested trade-off between quality and performance. It is designated for hash-table lookups or basic fingerprinting, but is not suitable for cryptographic uses. The hash function performs sufficiently in the well known "Uniform Distribution Test" ($\chi^2$) and in the "Avalanche Test"[5]. More on these empirically determined results is described in [Jer97]. Calculating a hash value out of $n$ input bytes requires $6n + 35$ machine instructions on IA-32 hardware, attesting Jerkins' Hash a good performance (MD4 needs $9.6n + 230$ instructions).

As mentioned in Section 2.4, the ESX Server creates a hash value for each page in the system and inserts the result into a hash table. If the generated hash value already exists, a full page comparison is initiated. On success, the affected pages are shared. On failure, a hash collision occurred and the hash value is simply ignored[6]. Notice that the ESX Sever uses **whole** memory-pages to generate hash values, leading to a constant hashing overhead, even on systems with a low page-sharing potential.

## 3.3.1 Partial hashing

Our approach tries to reduce the performance overhead required by the hashing of complete pages. For the purpose of partial hashing, we divide the hashing process in $n$ *hash levels*. Each hash level uses a different number of input samples to calculate the hash value of a page. Without the loss of generality we assume that:

$$|\mathcal{U}_n| < |\mathcal{U}_{n+1}| \tag{3.1}$$

The entropy of a single hash key correlates with the number of input samples used in the corresponding hash level and, therefore, the entropy of a hash key increases with each subsequent hash level. Whereby computational expenses rise with each level.

Generated hash values as well as page informations are inserted into a lookup table, where compare and search operations take place.

The partial-hashing algorithm:

1. *hash value* is ''
2. **For each** *hash level*
   a) Calculate *hash key* for current *hash level*
   b) Append *hash key* to *hash value*
   c) Search lookup table for an entry (partially) matching *hash value*
   d) **If** no entry was found, enter *hash value* and page information into lookup table and **terminate**

[4] Also used in [Wal02]

[5] Every input bit should affect every output bit about 50% of the time.

[6] [Wal02] states that for a static snapshot of the largest possible IA-32 memory configuration with $2^{24}$ pages, the collision probability is less than 0.01%

   e) **Otherwise if** *hash value* for *hash level* + 1, does not exist for found entry, calculate corresponding *hash value*

3. Share underlying page

The algorithm tries to identify unique page contents (*unique page*) early, thus saving computational expenses. On systems with a low page-sharing potential, this behavior will most likely result in a large number of pages that only need to be hashed to level one or two, whereas on systems with a high sharing potential, many pages will have hash keys for all hash levels.

### Example

Figure 3.3 shows a simple sample distribution for three hash levels. A 4KB page is divided into different samples. Hash level one uses bytes 1 to 64 to generate its hash key, level two bytes 65 to 1024, and level three bytes 1025 to 4096.

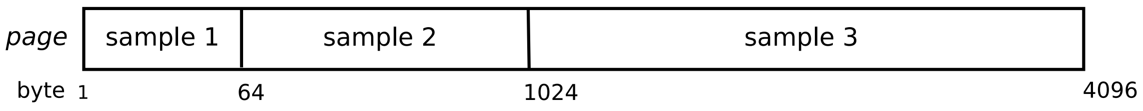

***Figure 3.3:*** Example of a hash-level sample distribution.

A property of hash functions is the occurrence of hash collisions, for the length of the generated hash key is smaller than the length of the input data. Therefore, we define an additional hash level four, which uses the trivial hash function:

$$h(k) = k \qquad k \in \mathcal{U} \tag{3.2}$$

The length of $k$ is equivalent to the page size (4KB). It is obvious that hash level four is a “dummy” hash level, performing full page comparisons.

An increase of entropy, as demanded by equation 3.1, is fulfilled. While hash level one uses a 64 byte sample (small entropy), the input entropy increases with each further hash level.

In this example, the sum of the input samples of all hash levels is equal to the page size, thus the maximum expenses for hashing are identical with the hashing costs of the ESX Server approach. Hash collisions remain unpredictable. Therefore, a full page comparison (level four) has to be performed in any case, what leads us to the assumption that hash levels sampling smaller page portions might suffice as well. Unfortunately, we lack empirical results, which are necessary to determine relevant parameters like the hash-key-storage size, the sample distribution, and the number of samples per page (Section 6).

### The lookup table

We designed the lookup table by again using a binary tree structure, giving us the advantage of a lower search overhead in hash level $n + 1$ (compared to starting over at the root), once a matching entry in level $n$ has been located (Figure 3.4).

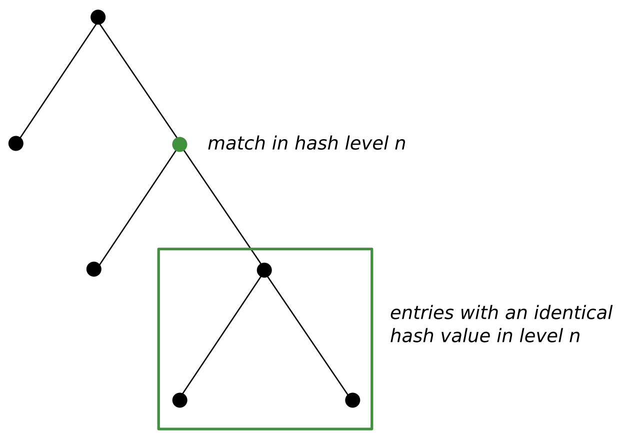

***Figure 3.4:*** Lookup table search

A match of the hash value in level $n$ occurred. When searching matching values for hash level $n + 1$, the right subtree is scanned only, beginning at the found position (green circle). The hash table also tolerates hash collisions. If any two pages produce identical hash keys for each hash level, but show different page contents, the new page is nevertheless inserted into the lookup table. When the same situation occurs again, pages with an identical hash value can be found at the next node (right path). Speaking of hash collisions, each hash level has a different hash-collision probability $p_n$. Because hash level are independent events, the overall joint probability $P_c$ of a hash collision for all hash levels is defined as:

$$P_c = \prod_{i=1}^{z} p_i \tag{3.3}$$

$P_c$ may not necessarily be smaller than the collision probability of the ESX Server approach (Section 5.3).

### Optimizations

A general task of runtime-memory sharing is the identification of read only pages. Only this page type can safely be shared. For this purpose the ESX server introduced the notion of *hint* pages (Section 2.4). Hint pages are, in ESX Server terminology, pages that own a **unique** hash value and, therefore, are not shared up to this point. Because write operations to the hint page can occur at any time, this page type needs to be rehashed on every hash-key-comparing operation. Once a hint-matching page is located and both pages are merged, the "hint" mark is removed—any further write request to the shared page will result in a copy-on-write operation.

Our approach is not as aggressive. It aims at reducing the effect of hashing on the system's overall performance by identifying *hot* pages—pages that are subject to a lot of write operations (e.g., stacks, buffers, caches, queues, . . . ). If a page produced a unique hash value it is nevertheless inserted into the lookup table, but mapped and marked as read only. A write operation to such a read only mapped page, leads to a page fault. In turn the read only tag is removed. On next hash-key comparison, the missing read only mark is detected, causing the rehashing of the page. We define an upper limit to the number of times a page can be rehashed (or a read only marked page can be written to). If this limit is exceeded, then the page is entirely removed from the hashing and page-sharing process.

Another optimization is the task of locating memory pages freed by L$^4$Linux. The problem here is that the L$^4$Linux-server task internally marks pages as free upon release. These pages are not detectable from the outside, because the contents of the freed pages still differ. To overcome this barrier, we provide a Linux kernel option that (if enabled) completely "zeros" the contents of a page upon release, thus marking it (externally) as free. The "zeroed" page concept implies two key advantages: Free pages can be shared between any two instances, as well as among any single L$^4$Linux instance. And second, because Linux always hands over "zeroed" pages to its user tasks, unused user-level memory can be shared in the same fashion.

# Chapter 4

# Implementation

Having described the design of our L$^4$Linux clone, it now is time to inspect the strategies of Chapter 3 in more detail. This chapter will for the most part deal with checkpointing–resume procedures of L$^4$Linux as well as internal *Freezer* structures. At the end, details of the "partial hashing" (Section 3.3.1) approach are presented. Mentioned L4 abstractions will not be explained; for a detailed examination of these concepts, please refer to Sections 2.1 and 2.2 as well as [Lie96], [ALE$^+$00], [Lac02], and [srg03].

## 4.1 L$^4$Linux checkpointing – The details

Before we start our examination, we must apply restrictions this project will not support. First of all, devices that cannot be shared between multiple L$^4$Linux instances must be disabled. This concerns mostly Linux-device drivers that require direct hardware access. In L4, shareable devices need a special L4 server implementation, which handles the multiplexing of requests offered by L4 tasks. An existing server of this kind is `ORe` [ore]—a network multiplexer, also available as L$^4$Linux-network-driver stub. We expanded the L$^4$Linux frame-buffer driver in a way that this limitation is of no concern to applications using `DOpE` [Fes02] or the L4 console [l4c]. Recall that device drivers are not subject to checkpointing.

Hybrid applications cannot be supported as well. A hybrid application uses Linux and L4 primitives in parallel. Currently, there is no feasible way to inform a hybrid application of an ongoing checkpointing–resume operation.

### 4.1.1 Saving the kernel state

Figure 3.1 in Section 3.1 shows a possible L$^4$Linux-runtime setup. Our challenge is to save the state information of each thread running inside the L$^4$Linux task and additionally the states of all the Linux-user applications executed in separate L4 tasks. One of our premises is, to apply as few changes as possible to the L$^4$Linux kernel, we will rather take advantage of already implemented features.

### Short excursion into Linux power management

Software-suspend (*swsusp*) support has been added to L$^4$Linux in version 2.6.17. It is an extension of the native Linux-software-suspend (to disk) mechanism. The native Linux mechanism works as follows: All user tasks and kernel threads are tricked into suspension code (called *refrigerator*), where they perform no operation, but calling `schedule()` if woken up. For this reason, Linux sets a flag in the Linux task structure, thus omitting suspended processes from future scheduling. Kernel threads have to check periodically (by calling `try_to_freeze`) if there is an ongoing suspend operation and, therefore, enter the refrigerator code voluntarily. After all processes have been frozen, the kernel notifies registered devices about the suspend operation. During the final step, the kernel takes a snapshot of the system memory and transfers this image to backing storage where it may be used at resume time.

To be able to resume from a saved checkpoint, the operating system requires a reboot and a special boot option. During the boot operation, the saved checkpoint image is read from backup storage and loaded into memory. At the end of this resume procedure, the registered devices are again notified, this time about the ongoing resume process, all the task flags are reset, causing the affected processes to leave the refrigerator code and resume normal operation.

Linux does not know anything about L4 tasks, even though the state of all processes and kernel threads is saved at the Linux-kernel stack, the L4 tasks must separately be terminated during suspend and recreated at resume time. For this purpose, L$^4$Linux registers a dummy device, which is notified during the suspend–resume operation. Recall that during suspend devices are notified **after** process states have been saved, during resume the opposite behavior applies. This way, the L$^4$Linux-dummy device is enabled to safely terminate and create L4-user tasks. Memory mappings are also restored, which we will not emphasize here. Additionally, L$^4$Linux defines an interface where functions, executed during software suspend, may be registered. This interface is used by L4 threads running inside the L$^4$Linux task.

Because of this working infrastructure, we decided to use it as a skeleton for our checkpointing purposes. We introduced a new power state *ds* (dataspace), which triggers the checkpointing procedure:

```
shell# echo ds > /sys/power/state
```

This solution is not a necessity because any other mechanism can be implemented if required (e.g., an external IPC sent to the L$^4$Linux task would be another possibility). Our approach takes advantage of the L$^4$Linux-software-suspend implementation, with the following exceptions:

- The states of **all** L$^4$Linux-sever threads are saved—not only the Linux kernel threads and user tasks
- Memory is **not** brought to backup storage
- No reboot is required to resume operation from a checkpoint

Because there are multiple suspend interfaces (Linux, L$^4$Linux device), saving the kernel state becomes a gradual process. The states of Linux processes and kernel threads[1] are saved during Linux software suspend. Others, like the *idler thread* or the *IRQ-timer thread* in Figure 3.1, implement additional suspend–resume calls, which are registered at the L$^4$Linux device.

Having performed these operations, the L$^4$Linux thread saves its own CPU state and passes control to the *startup thread.* In the setup shown in Figure 3.1, there are now only four threads remaining. The *tamer* and *semaphore* synchronization threads are in a well-defined state because nothing needs to be synchronized at this point. Therefore, these two threads can safely be halted. Also, no more page faults will be generated, thus the *region-mapper* thread can be halted as well. In the final step, the startup thread determines the resume address and the resume-stack pointer, passes these two values to the Freezer, and `exit(0)`s.

### 4.1.2 Memory handling

At this point, we should have a complete memory image of the once running L$^4$Linux task. Unfortunately, the Linux memory is freed by L4 upon exit. Because the Freezer has not yet overtaken the L$^4$Linux memory management(Section 3.1.2), it becomes necessary to transfer the memory used by L$^4$Linux to the Freezer. This operation is performed during software suspend. L$^4$Linux queries the region-mapper thread [ALE$^+$00] and iterates through its address space. A region map handles the mappings of pages to the virtual-address space of an L4 task (Section 2.2) and is started as the first thread within every L4 task. Currently, there are two types of virtual-memory regions L$^4$Linux must handle. The first kind are *dataspace* regions that map pages belonging to a specific dataspace. A page fault in a dataspace region causes the region mapper to translate the faulting virtual address into a (dataspace, offset) tuple. This tuple, is then forwarded to the appropriate memory manager. Address spaces (respectively dataspaces) also have an owner, who granted or mapped the memory pages in the first place. *Pager* regions are the second kind of memory regions. Here the faulting virtual address is **not** translated by the region mapper, but simply forwarded to the given memory manager. This leads to three cases:

1. Dataspaces owned by L$^4$Linux
2. Dataspaces **not** owned by L$^4$Linux
3. Pager regions

Figure 4.1 shows an example of these three cases. Dataspaces 20, 21, and 22 are owned by L$^4$Linux itself (1). The inherent memory manager is "DMPhys" (here dataspace manager – *DSM*). In turn, L$^4$Linux transfers the ownership of these dataspaces to the Freezer. This way the memory will not be released upon L$^4$Linux exit, also the memory manager is set to the ID of the Freezer server. Dataspaces 10 and 11 are owned by the

[1] Notice that Linux kernel threads are executed within the L$^4$Linux-server thread (Figure 3.1), using different stacks.

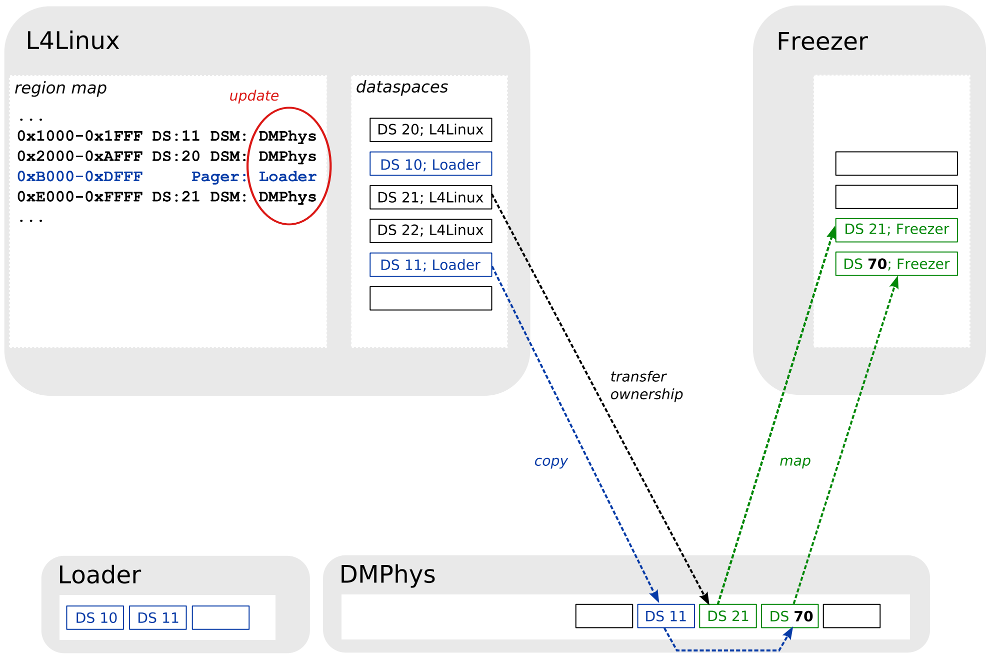

***Figure 4.1:*** L4Linux memory before checkpointing

L4Loader (2). Because L4Linux is not the owner of these dataspaces, it cannot transfer the ownership. Therefore, dataspaces 10–11 are copied, making the Freezer owner of the resulting copies (e.g., copy of dataspace 11 is dataspace 70). Figure 4.1 also shows a pager region (3), which are harder to handle. Generally spoken[2], this virtual-memory region must also be handled by the Freezer, so the memory manager of region 0xB000 has to be changed. The result of these operations is displayed in Figure 4.2, the Freezer now is memory manager and pager of all L4Linux memory. Note that this memory state is essentially the same as if the Freezer had been the pager of L4Linux since startup.

## 4.2 Cloning L4Linux

Containing a complete memory image as well as all necessary state information (within this image), the Freezer is now able to create self sufficient L4Linux instances. A new instance is started with the creation of a L4 task using the resumption address and stack pointer provided during checkpointing. As described in Section 2.2, page tables and the kernel's mapping database are empty upon task creation, causing the L4Linux clone to immediately raise page faults on memory access. Because services of the region mapper thread are not available at this point, the faulting virtual address cannot be translated into the appropriate (dataspace, offset) tuple causing the faulting address to be forwarded to a special thread within the Freezer (pager thread), where the necessary

[2] For example pager regions can lead to other memory managers, anonymous memory, or simple dataspaces

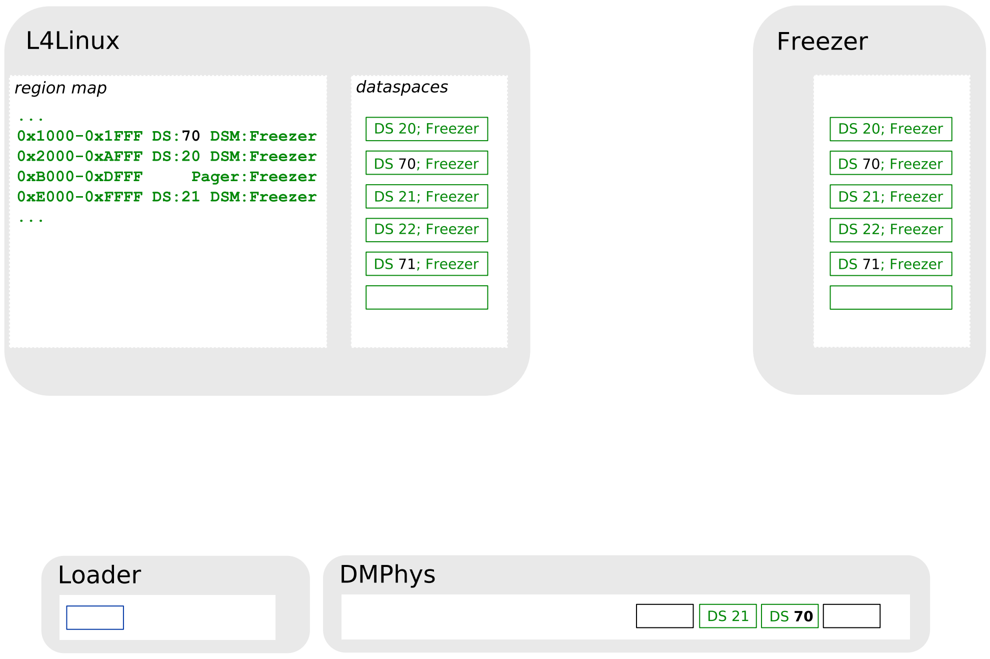

***Figure 4.2:*** L$^4$Linux memory after checkpointing

address translation takes place and the affected page (Section 3.2.1) is mapped to the L$^4$Linux instance.

### The thread-identifier obstacle

Thread IDs are an integral part of L4 and are used by L$^4$Linux. Because our clone is a freshly created task, all restored threads will have a new Thread ID. To be able to guarantee a functioning system, where for example IPCs take place, synchronization has to be performed, and page faults are raised, it becomes the job of the first L$^4$Linux thread resumed to update thread IDs. Several structures require an update. First of all, global variables containing thread IDs are adjusted. The thread library also keeps thread IDs within its TCB structures. Because we did not want to apply unnecessary changes to the L4Env, we decided to simply reinitialize the thread library. The drawback of this decision is that every resumed L$^4$Linux thread **must** register itself at the thread library.

Additionally, L$^4$Linux puts the ID of each thread on top of the thread's stack, which is an optimization aiming at not overusing the `l4_myself()` system call [Lac02]. Therefore, the first thread of an L$^4$Linux instance iterates through all thread stacks and updates the ID on top of each stack accordingly.

Local variables containing thread IDs cannot be handled at this point. We leave the update of these variables to the thread, actually using them. This can be achieved by registering a function at the L4-power-management interface (Section 4.1).

Only L$^4$Linux write requests possibly lead to copy-on-write operations at the Freezer. Therefore, the first running thread maps all L$^4$Linux memory read only, thus read requests do not trigger any page faults—somewhat improving the system performance. During the final step, the *startup thread* is created and resumed (Figure 4.3), the first thread then enters its server loop, thus enabling region map services.

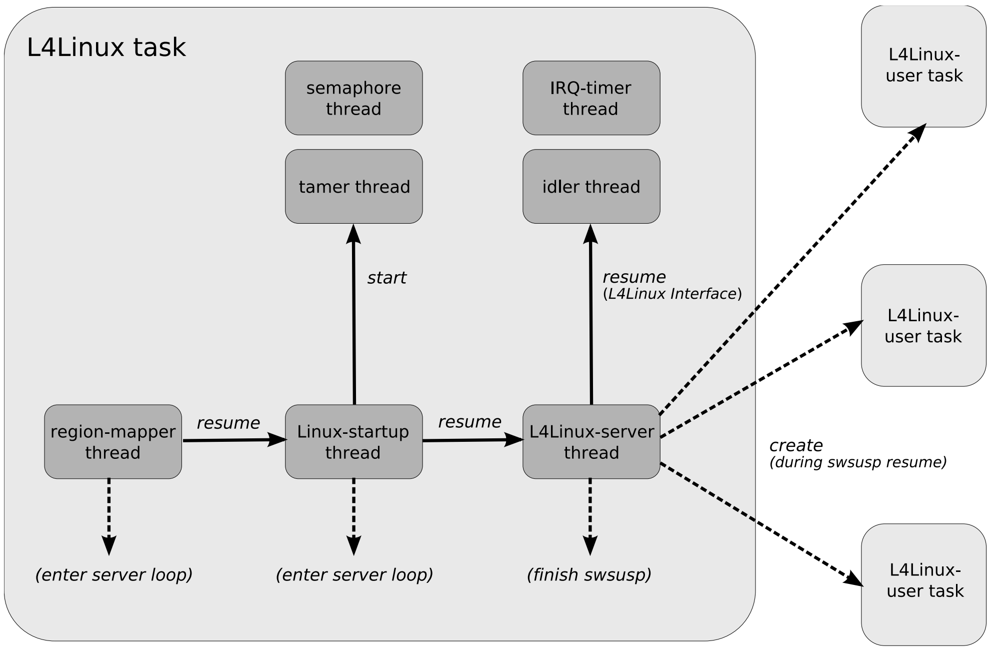

***Figure 4.3:*** L$^4$Linux resume process

The startup thread begins execution by loading its saved CPU state. Because from this point on, competitive resource access may appear, threads that handle synchronization (*tamer* and *semaphore* thread) are started before the L$^4$Linux-server thread. Finally, the L$^4$Linux-server thread is resumed and the startup thread, as before, enters normal operation.

With the resume of the L$^4$Linux kernel, the Linux-software suspend is continued until each thread and user task is back to normal operation. Because of our COW approach and the resulting read-only page mappings, L$^4$Linux was switched to a demand-paging model. This behavior is implemented by "touching" a page upon a write request **before** it is mapped to the faulting user task, leading to a write page fault as well as a COW operation at the Freezer in case the page is still mapped read only.

## 4.3 The Freezer

Having described most of the Freezer concepts, we will now have a look at some internals. The Freezer handles:

1. The checkpointing of L$^4$Linux memory
2. Page faults for a given (dataspace, offset) tuple

3. Page faults for given virtual addresses
4. Memory-resource sharing
5. L$^4$Linux-clone creation

To be able to provide this functionality, the Freezer implements three threads (Figure 4.4). The *main thread* manages task (1), (2), and (5). It defines an external IDL interface, enabling L$^4$Linux to directly communicate with the Freezer during checkpoint operation, while also providing interface calls for L$^4$Linux-clone creation.

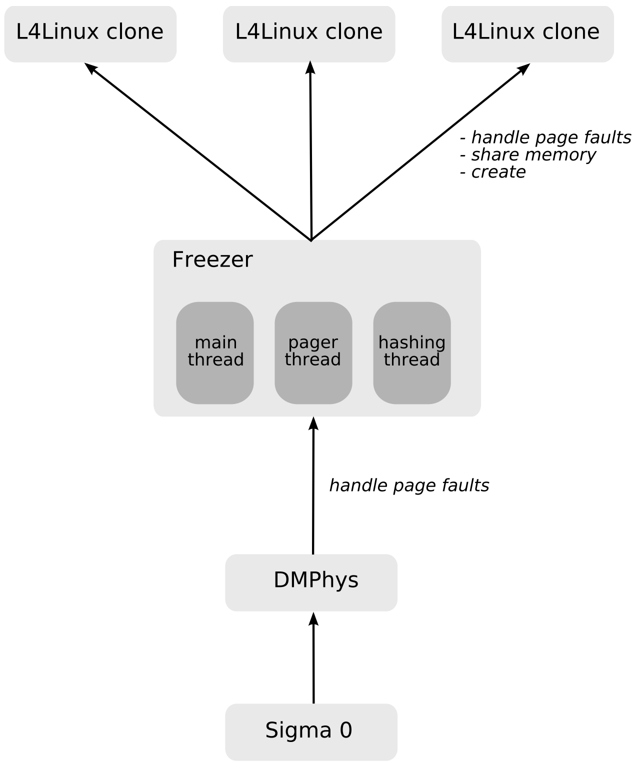

***Figure 4.4:*** Freezer usage scenario.

Page faults requiring address translations, or belonging to *pager* regions of the region map (2) are processed by the *pager thread* (Section 4.2). Finally, the *hashing thread* is responsible for memory-resource sharing (4), by continuously scanning the memory for shareable pages. Because this thread is not essential to the Freezer's operations (optimization) and because it increases the response time of the main and pager threads, we assign a low L4-system priority to the hashing thread. Note that because these threads competitively access the L$^4$Linux-server's memory, synchronization is required.

Copy-on-write operations demand additional memory dynamically allocated at run-time. For this purpose, the Freezer manages a COW-memory pool. Memory is allocated and released using 4MB slabs, the Freezer keeps track of the COW pool using free lists. Because of the well known memory-fragmentation problem, the COW pool is subject to memory compactification at exit of a L$^4$Linux clone.

### 4.3.1 The hashing thread

This thread implements the partial-hashing approach described in Section 3.3.1. We use the sample distribution and the hash levels shown in Figure 3.3.

The hashing thread features a *working queue* filled with pages by the other Freezer threads on each copy-on-write operation. This queue is checked from time to time for new page entries, if new pages are located, the hashing thread begins its operation.

#### Hash values

Each hash level produces a single hash key. As described in Section 3.3.1, the input entropy and, therefore, the computational expenses increase with each hash level. On the other hand, each level provides us with more precise informations about possibly equal page contents. Therefore, a *hash value* represents, in our context, the composition of hash keys of different entropy.

Figure 4.5 illustrates the hash-value structure we use for the three-level setup:

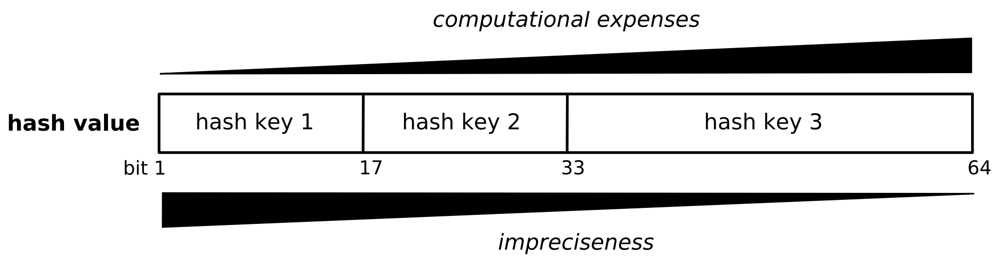

***Figure 4.5:*** Hash value layout

Because level one and two provide a smaller entropy, we decided to use a small hash-key size of 16 Bits as well, taking a higher hash-collision probability into account. Hash level 3, using a 32 Bit hash key, minimizes hash collisions aiming at the avoidance of unnecessary full page comparisons (dummy-hash level four). The resulting hash value size of 64 Bits is identical to the one of the ESX Server[3].

The order shown in Figure 3.4 must be ensured. Therefore, all hash keys within a hash value are initialized with a value of zero. This way, pages with an equal hash key in level $n$, but possibly different hash keys in level $n + 1$, form subtrees[4](Figure 4.6).

#### Page sharing

Page sharing must be integrated into our copy-own-write design (Section 3.2.1). For this purpose, we introduce *page-object references.* A page-object reference (short: page reference), is a pointer to another page object, whose underlying page is used by both the referring page object and the referred page object. To be able to keep track of page references, we implemented a simple reference count. Figure 4.7 demonstrates our page sharing approach. The hashing thread located two pages with identical contents (red

[3] The ESX Server seems to be the only comparable concept around.

[4] This effect is caused by the "$\geq$" order relation of AVL trees.

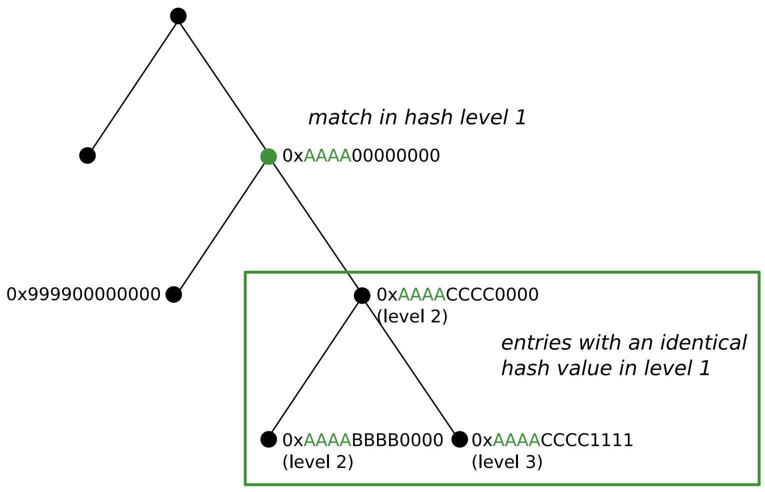

***Figure 4.6:*** Example of the search tree of Figure 3.4, showing the subtree for a hash key match in level 1

circles). Therefore, the underlying page of “Instance 4” is released and the appropriate page object becomes a pointer to the content-matching page. After this point, two possible events may occur. First, a write operation at the referring page object happens, resulting in a copy operation of the underlying page. The referring page than turns into a normal page object (i.e., becomes a black circle in Figure 4.7) and the reference count at the referred page is decreased.

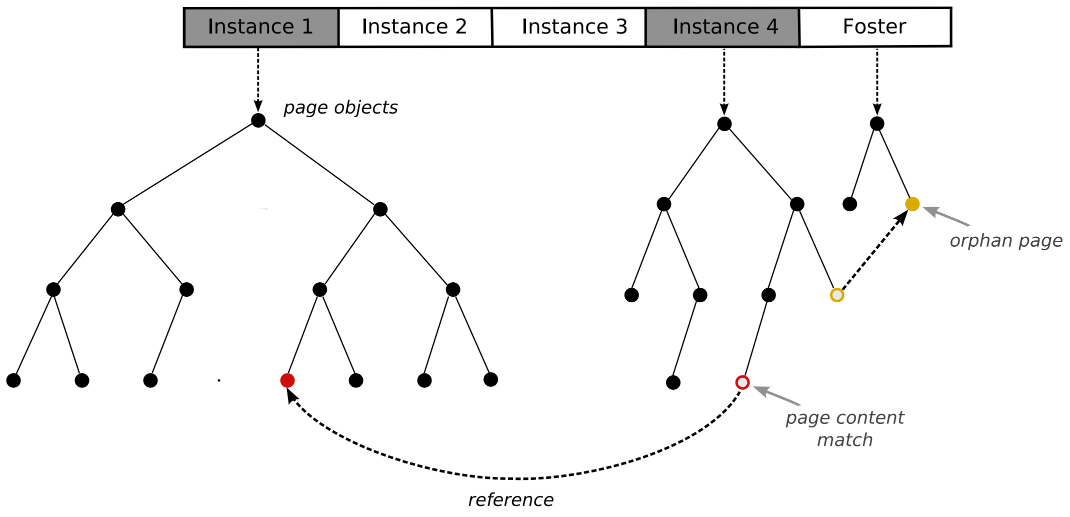

***Figure 4.7:*** Page sharing

If a write operation occurs at a referred page object, the situation gets more complicated because other pages are referring to it. In this case, we copy both the page object and the underlying page. The original page object, to which referring pages are pointing, then becomes an orphan and is moved to a dummy-Linux instance (*Foster* in

Figure 4.7), along with the original page, whereas both copies are associates with the faulting L$^4$Linux instance. In Figure 4.7, the yellow colored page object of “Instance 4” originally referred to a page object of “Instance 1”, before a write operation to the page object belonging to “Instance 1” occurred. After the resulting copy operations, the page of “Instance 4” points to the same object, but the object is now associated with the Frozen instance.

Chapter 5

# Evaluation

This chapter presents performance and memory-usage measurements of L$^4$Linux clones. All data has been gathered using a Pentium IV Williamette@1800MHz (8 KB L1 cache, 256 KB L2 cache, 512 MB DDR RAM@266 MHz).

## 5.1 Cloning

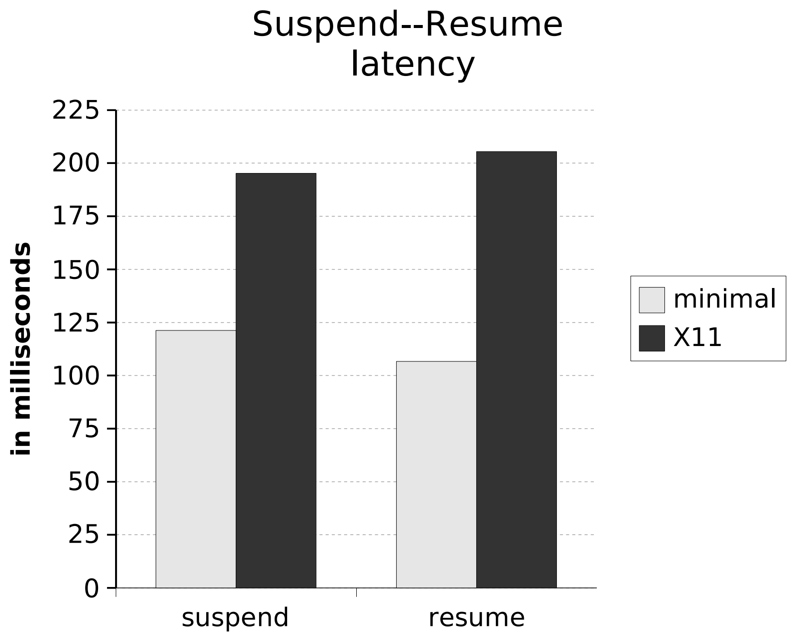

***Figure 5.1:*** L$^4$Linux suspend–resume latencies

We compare the suspend and resume performance of two L$^4$Linux setups. First, a *minimal* L$^4$Linux instance with only 64 MB of main memory and a small RAM-disk of 16 MB. Only, basic services like a login shell and `getty` are started. The second L$^4$Linux setup executes a full grown *X11* server, requires 220 MB of main memory with a RAM-disk size of 96 MB. Figure 5.1 presents the suspend–resume latencies of both setups.

Interestingly, the resume latency of the X11 setup is larger ($\approx 10ms$) than the suspend latency—not the case in the minimal setup. This behavior is caused by our copy-on-write approach, the X11 setup obviously incurs more write-page faults than the minimal approach, which lead to COW-operations and additional IPC overhead. All the more write-page faults occur, the more the resume latency is increased. Therefore, the resume latency heavily depends upon the system's workload.

Table 5.1 shows the memory usage of each setup. Note that the Frozen image is larger than the L$^4$Linux-main memory because it also contains the Linux-kernel itself as well as L4 libraries required by the L4Env.

| | **Frozen image** | Overhead 1 | **Resume** | Overhead 2 |
|---|---|---|---|---|
| **Minimal** | 74324KB | 444KB | 1084KB | 16KB |
| **X11** | 243068KB | 1392KB | 3044KB | 40KB |

***Table 5.1:*** Memory usage

*Overhead 1* is the memory usage of the Freezer's management structures, needed to handle the Frozen image. *Resume* is the additional (COW) memory, allocated during L$^4$Linux clone startup and the inherent management *Overhead 2*. As can be seen, the total overhead of the Freezer's management structures is $\approx 0.6\%$.

## 5.2 Isolation comparison

To be able to compare our virtual-machine-isolation approach with traditional operating-system isolation, we determined the total memory usage (including shared libraries) of Firefox (Version 1.0.6) right after startup, which turned out to be around 17.5 MB (*exmap* [exm]). Because we demand application-level isolation, we assume that each Firefox instance will require these 17.5 MB[1]. This gives us the opportunity to evaluate the memory savings of our sharing techniques.

For the purpose of L4-Firefox isolation, we create L$^4$Linux clones as in the X11 setup of Section 5.1. Within these clones, a single (isolated) Firefox instance is then started.

Figure 5.2 shows the memory usage when isolating 1–5 Firefox instances at the same time. In our approach (*Freezer*), a single Firefox instance obviously consumes more memory than an OS-level instance because an isolated Firefox instance consists of the memory used by Firefox plus the memory required to create a L$^4$Linux clone, leading to the difference of about 8 MB. Beginning with Firefox instance two, the (Freezer's) memory overhead of an isolated Firefox is reduced. This behavior is caused by an increase of

[1] Notice that this memory usage may not be the correct when using operating-system virtualization, because libraries may be shared between different isolation containers.

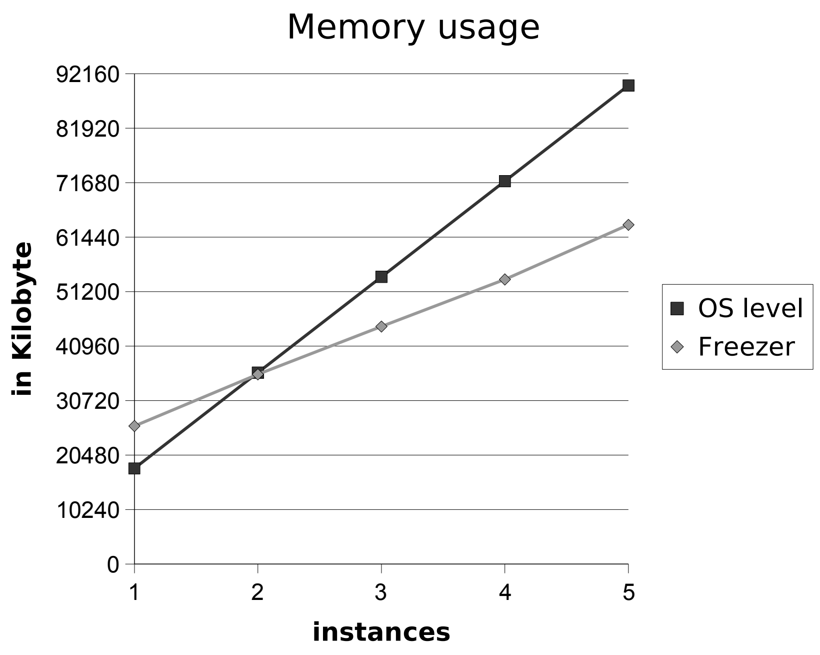

***Figure 5.2:*** Comparison of isolation

the page-sharing potential, and, therefore, by the memory-sharing strategies described in Sections 3.2.1–3.3.1. Thus, the memory allocation for each additional Firefox instance drops significantly from about 25 MB (one Firefox) to 10 MB (40%)—substantiating the already mentioned assumption that only one third of the main memory contains read–write pages.

Our partial-hashing approach (Section 3.3.1) identifies sharable pages at runtime, causing a fluctuation of the memory usage over time. Figure 5.3 illustrates this property of the Freezer for the example in Figure 5.2 . The startup of an additional L$^4$Linux–Firefox instance, for example at time 15 and 22, can clearly be identified. Write operations are satisfied by the Freezer, leading to a sudden increase of the demanded memory. At the same time the hashing thread (Section 4.3.1), locates shareable pages, thus releasing memory pressure, until at time 18 respectively 25, no more shareable pages can be located. Afterwards, the memory usage stabilizes until the next L$^4$Linux–Firefox instance is started.

## 5.3 Hash collisions

In this section we will reveal some statements about hash collisions. For this purpose, we measured the number of hash collisions that occurred in each hash level. The measured workload is the one of Section 5.2.

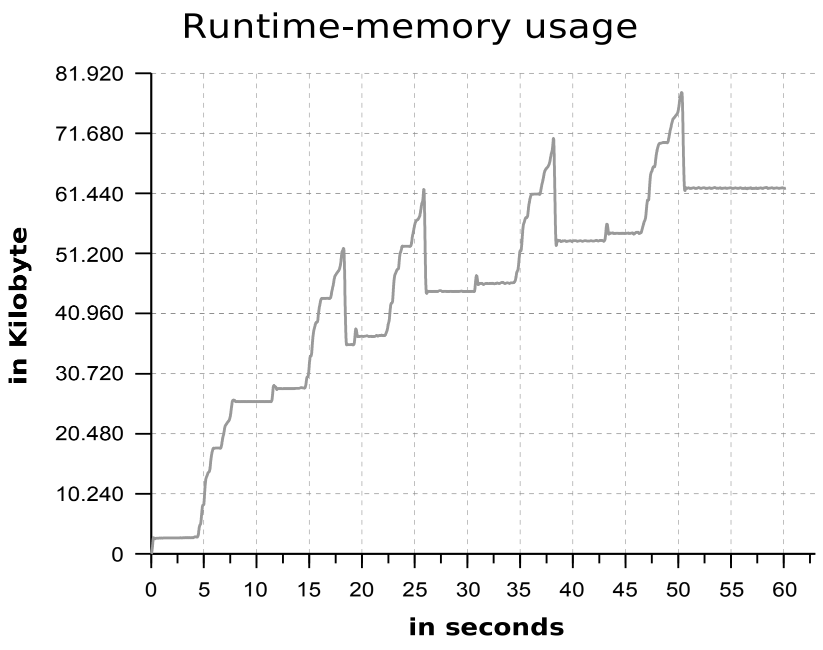

***Figure 5.3:*** Page sharing

Section 4.3.1 described the different hash key storage sizes. Section 3.3.1 presented the (currently) implemented input-sample distribution. Because each hash level increases the input-sample size and might use a different hash-key-storage size, we expect a decrease of the collision probability for higher hash levels. Table 5.2 shows the relative frequencies of hash-key collisions per hash level for the same workload. In order to validate the result, we repeated this measurement three times.

| $H_1$ | $H_2$ | $H_3$ | **Overall** |
|---|---|---|---|
| 0.1051 | 0.0659 | 0.0001 | $7 \cdot 10^{-7}$ |
| 0.1103 | 0.0670 | 0.0001 | $7 \cdot 10^{-7}$ |
| 0.0868 | 0.0576 | 0.0002 | $10^{-6}$ |

***Table 5.2:*** Relative frequencies of hash-key collisions

As expected, the collision frequency decreases with each further hash level. While level one shows a high collision rate of about 10%, level two almost halves this value by increasing the size of the input sample. Level three, which increases the sample size **and** the hash-key-storage size presents a value roughly equal to 0.01% as in [Wal02].

*Overall* is the resulting collision probability calculated using equation 3.3. All three measurements present an overall collision probability that is an order of magnitude smaller than the one described in [Wal02]. On the other hand, these values are workload–hash function dependent and may vary for different workloads.

# Chapter 6 Outlook

There are a number of topics this thesis could not analyze, mostly because of the fixed amount of time available. We will limit our outlook to three topics.

**Performance optimization** is the first. In order to compare the performance of the cloned L$^4$Linux version with the one of native L$^4$Linux, benchmarking becomes necessary. The benchmark test traditionally used at TU-Dresden is the AIM multiuser benchmark suite. This test remains to be done for cloned L$^4$Linux. Nevertheless, there are already ideas on performance optimization. As mentioned in Section 3.2.1, page tables could be used as an alternative to AVL trees in order to organize page mappings within the Freezer. From this approach we expect a possibly faster page-fault-response time (constant-time overhead during search operations), leading to a lower L$^4$Linux-clone-startup latency, as well as a better L$^4$Linux-clone performance. Nevertheless, page tables can consume a greater amount of memory, compared to trees, because they may be sparsely occupied. To be able to evaluate both approaches, both must be implemented and examined in real use.

Different hash-level-sample distributions are also of interest (Section 3.3.1). The goal here is, to minimize the number of input samples for a hash level, while preserving an acceptable hash-collision probability, thus saving computational expenses. A large hash-collision probability leads to a large number of full-page comparisons, negating the saving effect of the hash-key computation. Reasonable input-sample distributions need to be determined empirically.

The L$^4$Linux-suspend latency would be smaller if the Freezer was the pager of L$^4$Linux from the beginning (Section 3.1.2), because no memory-copy operations would be necessary. This goal is “trickier” than it appears to be. L$^4$Linux is started by the L4Loader, causing entanglements, which are not easy to solve from the outside.

A subject not discussed yet, is **admittance** control of new L$^4$Linux clones. It is impossible for the Freezer to evaluate and control memory requests of single L$^4$Linux instances, additionally, we support overcommitment. A reasonable solution is *Ballooning* (Section 2.4). Here, a new instance is allowed into the system when there is enough free memory for clone startup. If main memory is rare, then the internal main-memory usage of the L$^4$Linux clones is increased by the Freezer (externally), thus averting large memory requests by the L$^4$Linux instances. If the main memory usage drops, the internal memory usage of the clones can be decreased. Ballooning is not yet implemented.

Therefore, we recommend the deployment of *swap regions* (Section 2.5). L4's swappable-dataspace manager [Sum06], can be placed between the Freezer and the physical-memory manager, enabling the Freezer to (transparently) move pages to backup storage on main-memory exhaustion. As opposed to the solution proposed by the Denali Isolation Kernel, which uses one swap partition per virtual-address space, only one swap partition is required. This method becomes feasible with the adjustment of L$^4$Linux to a demand-paging model.

The final topic concerns **supported** features. Currently, L$^4$Linux is available for x86 and ARM platforms. Within this thesis, we implemented cloning support for the x86 architecture only. ARM support remains subject to future work.

A current limitation of the Freezer is the support of just a single L$^4$Linux checkpoint (Section 3.2.1). If we extended the Freezer to be able to handle multiple checkpoints, the Freezer could act as a checkpointing server for L$^4$Linux, giving multiple L$^4$Linux instances the opportunity to save and resume their state at any given time while performing memory-resource sharing at the same time. This feature would enable the Freezer to act as L$^4$Linux-checkpointing, memory sharing, and isolation server in parallel and in turn broaden its fields of application.

# Chapter 7
# Summary

This work proposes a real solution to the isolation problem of single applications. As opposed to the traditional isolation approach, which exploits operating-system-level virtualization, we enabled the virtual machine L$^4$Linux  to support application-level isolation (or a high isolation granularity). Featuring a startup latency of about 120 ms and a memory overhead of 1 MB, in its simplest form, the L$^4$Linux-isolation container is able to compete with the OS-level virtualization approach, while banning the Linux kernel from our *Trusted Computing Base*, thus guaranteeing a high isolation security.

For the purpose of memory-resource sharing, we developed a L4-memory manager that deploys copy-on-write as well as runtime-memory sharing techniques, working transparently between multiple L$^4$Linux-isolation containers.

Runtime-memory sharing is achieved by the *content-based page-sharing* concept, enabling our memory manager to potentially identify every shareable page within a system.

We improved upon the content-based page-sharing concept by introducing the partial-hashing algorithm, which imposes a total ordering of "fuzzy" hash keys (of page contents), aiming at the reduction of computational expenses.

The overall performance of the described system exceeded our expectations, expanding our opportunities toward application checkpointing in general as well as toward transparent memory-resource sharing for various applications.

# Glossary

**AVL tree** Adelson-Velsky-Landis tree

**Checkpoint, persistent object, frozen image** are synonyms. In this papers context it does not matter where they reside (i.e., it does not make any difference if they are stored on hard disk or in main memory)

**COW** Copy-On-Write

**DOpE** Desktop-Operating Environment

**DROPS** Dresden-Real-time-Operating System

**DS** Dataspace

**DSM** Dataspace Manager

**IPC** Inter-Process Communication

**IRQ** Interrupt Request

**L4Env** L4 Environment

**RM** Region Map

**Task and process** are synonyms

**TCB** Thread-Control Block

**TLB** Translation-Look-aside Buffer

**VM** Virtual Machine

**VMM** Virtual-Machine Monitor

# Bibliography

[ALE$^+$00] M. Aron, J. Liedtke, K. Elphinstone, Y. Park, T. Jaeger, and L. Deller. The SawMill Framework for Virtual Memory Diversity. In *Australasian Computer Systems Architecture Conference*, pages 3–11, Goldcoast, Queensland, Australia, 2000. IEEE Computer Society Press. 8, 24, 26

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press. 1

[BDGR97] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, 1997. 11

[BL98] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998. 10

[Cow06] C. Cowan. *Virtual machines, isolation, and security*, Mar 2006. 4

[exm] `http://www.berthels.co.uk/exmap/`. 35

[Fes02] N. Feske. DOpE – a grahpical user interfance for DROPS. Master's thesis, Technical Uinversity Dresden, Nov 2002. 24

[Gun06] A. S. Gunter. Jails unter FreeBSD. 2006. 2

[HBB$^+$98] Hermann Härtig, Robert Baumgartl, Martin Borriss, Claude-Joachim Hamann, Micheal Hohmuth, Frank Mehnert, Lars Reuther, Sebastian Schönberg, and Jean Wolter. DROPS: OS support for distributed multimedia applications. In *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 203–209, New York, NY, USA, 1998. ACM Press. 4

[HH02] J. G. Hansen and A. K. Henriksen. Nomadic Operating System. Master's thesis, University of Copenhagen, Dec 2002. 6, 10

[HHL$^+$97] H. Härtig, M. Hohmuth, J. Liedke, S. Schönberg, and J. Wolter. The Performance of $\mu$-Kernel-Based Systems. In *16th aCM Symposium on Operating Systen Principles (SOSP'97)*, pages 66–77, Saint-Malo, France, Oct 1997. 4

[HJ04] Jacob Gorm Hansen and Eric Jul. Self-migration of operating systems. In *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop: beyond the PC*, page 23, New York, NY, USA, 2004. ACM Press. 11

[Hoh96] M. Hohmuth. Linux Emulation auf einem Mikrokern. Master's thesis, Technical Uiniversiy Dresden, Aug 1996. 4

[HR06] H. Härtig and M. Roitzsch. Ten Years of Research on L4-Based Real-Time Systems. In *Proceedings of the Eighth Real-Time Linux Workshop*, Lanzhou, China, Aug 2006. 4

[Jer97] B. Jerkins. Alorithm Alley. *Dr. Dobbs Journal*, Sep 1997. 20

[l4c] `http://os.inf.tu-dresden.de/l4env/doc/html/l4con`. 24

[Lac02] A. Lackorzynski. L$^4$Linux on L4Env. Master's thesis, Technical Uiniversity Dresden, Dec 2002. 4, 24, 28

[Lan92] C. R. Landau. The checkpointing mechanism in KeyKOS. In *Proceedings of the 2nd International Workshop of Persistent Object Systems (POS2)*, pages 24–25, Paris, France, Sep 1992. 9

[Lie93] Jochen Liedtke. A persistent system in real use: Experiences of the first 13 years. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, Asheville, North Carolina, USA, Dec 1993. 6, 17

[Lie95] Jochen Liedtke. On Microkernel Construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, Dec 1995. 7

[Lie96] J. Liedke. *The L4 reference manual*, Sep 1996. 24

[LS01] P. Loscocco and S.Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42. USENIX Association, 2001. 2, 3

[OH05] R. Oglesby and S. Herold. *VMWare ESX Server: Advanced Technical Design Guide*. Brian Madden Company, 2005. 1, 6

[ore] `http://os.inf.tu-dresden.de/l4env/doc/html/ore`. 24

[Ros04] M. Rosenblum. The Reincarnation of Virtual Machines. *QUEUE*, pages 34–40, Jul 2004. 1

[SCL01] Espen Skoglund, Christian Ceelen, and Jochen Liedtke. Transparent Orthogonal Checkpointing through User-Level Pagers. In *POS-9: Revised Papers from the 9th International Workshop on Persistent Object Systems*, pages 201–214, London, UK, 2001. Springer-Verlag. 8, 9

[SLH$^+$99] R. Spencer, P. Loscocco, M. Hilber, D. Anderson, and J. Lepreau. The Flask Security architecture: System support for Diverse Security Policy. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123–139, Aug 1999. 3

[srg03] Operating system research group. L4Env – An environment for L4 applications. Technical report, University of Technology Dresden, 2003. 15, 24

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999. 9

[Sum06] S. Sumpf. A swap dataspace manager for DROPS. Master's thesis, Technical University Dresden, Mar 2006. 39

[SWS06] SWSOFT. An Introduction to OS Virtualization and Virtuozzo, Dec 2006. 2

[Wal02] C. Waldspurger. Memory resource management in VMware ESX server. Dec 2002. 12, 20, 37