LEHRSTUHL FÜR BETRIEBSSYSTEME

Univ.-Prof. Dr. habil. Thomas Bemmerl

# Entwurf und Implementierung einer Virtualisierungsschicht für den Betriebssystemkern „MetalSVM”

# Design and Implementation of a Virtualization Layer for the Operating System Kernel “MetalSVM”

Jacek Galowicz
Matrikelnummer: 285548

**Bachelorarbeit**
an der
Rheinisch-Westfälischen Technischen Hochschule Aachen
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Betriebssysteme

RWTH AACHEN UNIVERSITY

Betreuer: Dr. rer. nat. Stefan Lankes

# Contents

# List of Figures

# List of Tables

# List of Listings

# 1. Introduction

In the last decades computer programmers were used to the fact that their old programs did automatically run faster when executed on new computer generations. This fact based on the condition of new computers' higher clock frequencies which allowed for executing more processor instructions within the same time. The growing clock frequencies also led to growing heat generation and power consuption which in turn became a problem for both laptop computers and bigger server farms.

As a reaction, the increase of processor frequency in new computers did nearly stagnate a few years ago, but so did not the increase of the number of cores: Even today's laptop computers have up to about ten of them. Programmers need to rewrite their programs to take advantage of this.

Such computer's hardware either implements the *symmetric multiprocessing* (SMP) or the *message passing* architecture. In SMP systems every core has an identical view to the system's memory space. This way the programmer only has to make sure that the calculations his program does are properly parallelizable. He does not have to care about which part of his parallelized program has to run on which core as the cores have an identical view onto the memory space. This paradigm is called *shared memory programming*. Since every core has its own cache, the hardware needs to provide a mechanism to keep shared memory consistent between the different caches. Commonly referred to as *cache coherence*, this mechanism involves serialization of memory access. With a growing number of cores, this also results in heavily growing demands to the memory-bandwidth.

Message passing computer systems solve this problem by providing every core with its private memory space and bus. However, having outpaced SMP systems in computer clusters in terms of prevalence, they are harder to program: Besides parallelizing, the programmer has to explicitly move the data between the core's memory spaces to where it is used. In the long term, microprocessor manufacturers seem to regard symmetric multiprocessing as a dead end.

When hardware of future generations will not provide an SMP-typical view on the system's memory, it will be a crucial task to achieve a shared memory environment with cache coherence by software. Although message passing applications scale very well, shared memory can be beneficial in situations with dynamically changing memory access patterns. Shared memory platforms are also easier to deal with when balancing the system load. This is where Intel's *Single-chip Cloud Computer* (SCC) joins research. The SCC is an experimental microprocessor prototype with 48 cores on its die (see Figure 2.1 on page 3) which do not support any cache coherence. It is built to study the possibilities of future generation many-core computer architectures. Most research is done in examining and demonstrating the use of the

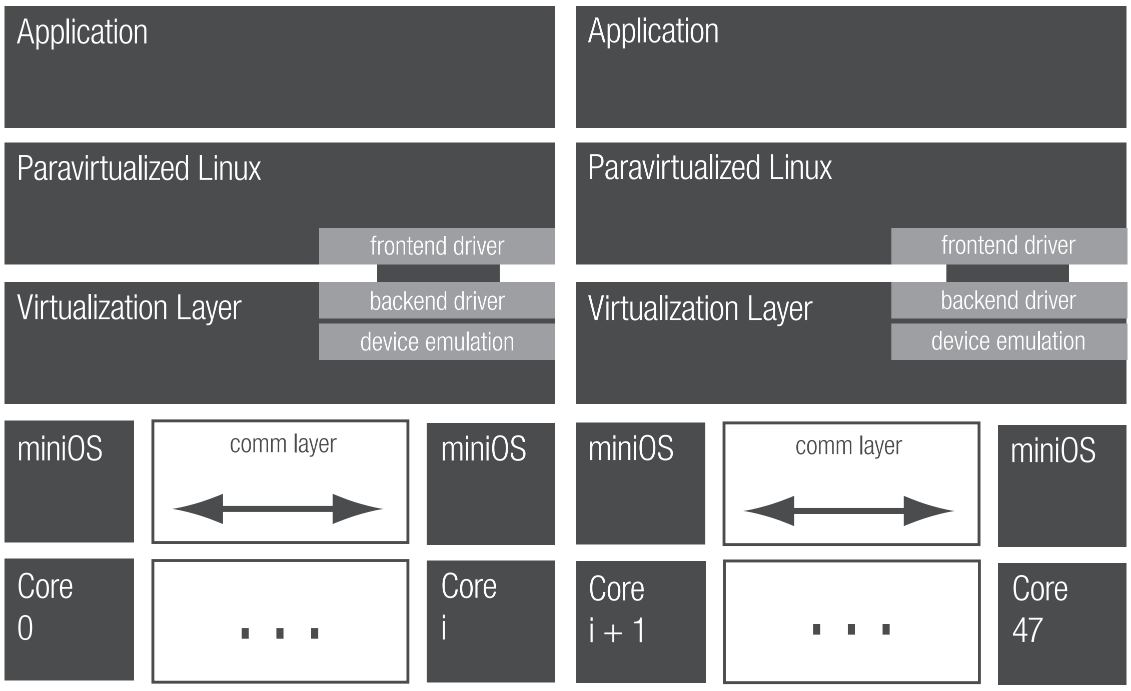


Figure 1.1.: Establishing coherency domains

SCC as a message passing architecture using the external and on-die memory for communication.

An attractive way of establishing a shared memory environment on this platform is adding a *virtualization layer* between message passing operating system kernels running on the cores and a single operating system kernel. The last-mentioned will find itself on a large virtual SMP computer system. This shall be achieved with a special small operating system running on each core providing the virtualization environment for an unmodified Linux kernel (see Figure 1.1). To enable the transparent use of a linear address space by the virtualized Linux kernel, the smaller kernels below the virtualization layer have to manage data exchange between the cores' different address spaces.

The focus of this thesis is implementing this virtualization layer in the operating system kernel MetalSVM, involving the port of an existing hypervisor implementation from the Linux kernel. This preliminary work allows later extension to a system managing coherency domains spanning over multiple cores. The thesis begins with describing MetalSVM and the Intel SCC together composing the target platform in Chapter 2. A general explanation about virtualization and virtualization mechanisms with focus on the Intel x86 architecture is given in Chapter 3. In this chapter the distinction between full- and paravirtualization is illustrated to point out the latter as most useful for the project and presents hypervisors in Linux which implement this approach. Finally, it explains the choice of which hypervisor to port to MetalSVM. Chapter 4 gives very detailed insights into the actual implementation.

# 2. MetalSVM on the Intel SCC

MetalSVM is a small monolithic operating system kernel. Initially it shared the largest part of its codebase with *EduOS*, a very basic operating system kernel developed for educational purposes. Both EduOS and MetalSVM have been developed at the Chair for Operating Systems of RWTH Aachen University since 2010. The name "MetalSVM" is a mix of the term *bare metal* and the abbreviation *SVM* standing for "Shared Virtual Memory".

The SCC's 48 P54C cores are arranged pairwise on $6 \times 4$ tiles on the microchip's die. Each of them is equipped with each 16 kB L1 data and instruction cache as well as 256 kB L2 cache. Up to 64 GB of external memory are provided by four memory controllers surrounding the grid. Due to the architecture's 32 bit limitation only 4 GB of the memory can be addressed per core. A mechanism referred to as *lookup table* (LUT) maps every core's physical 4 GB address space to the system's up to 64 GB of extended memory. There is one LUT per core. Each one contains 256 entries, which map memory segments of 16 MB size to the corresponding address in the core's address space. Possible memory locations for mapping are the private memory, local memory buffers, configuration registers, the system interface, power controller, and system memory. The LUTs can be dynamically reconfigured at runtime, what is especially interesting for possible implementations of shared virtual memory. Every tile also has a small amount (16 kB) of additional memory, namely the *message passing buffer* (MPB). This fast on-die memory can be used for more efficient communication between the cores, as classic architectures usually

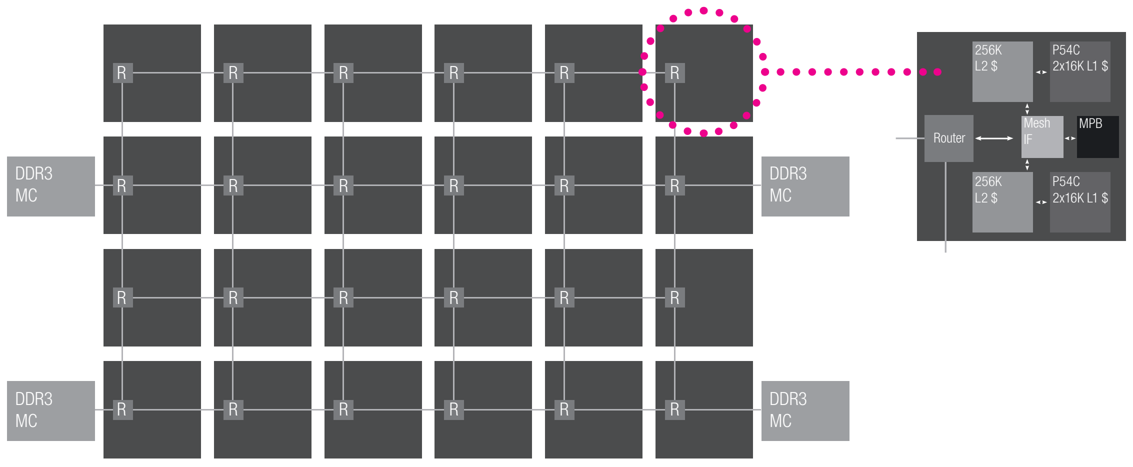


Figure 2.1.: Schematic top view SCC block diagram

only provide the relatively slow off-die memory for this purpose [10].

The idea to provide shared virtual memory on message passing architectures has already been realized in clusters before. Previous approaches are, for example, *Treadmarks* [2], *ScaleMP* [1], *Cluster OpenMP* [8], and *vNUMA* [3]. These differ in various aspects: Some of them use virtualization to provide a transparent memory space spanning over the cores/nodes in a cluster, others provide this functionality on user space level. The majority of projects tried to establish shared virtual memory on non-uniform memory access platforms. Those platforms are adversely affected by communication latency, which is problematic in situations with dynamically changing memory access patterns and excessive synchronizing with locks. Finally, all approaches have proven to be extremely sensitive to network latency and suffered from lacking programming standards [6].

The SCC with its potential for low-latency-communication and uniform memory access characteristics offers the possibility to try this concept without the limitations, which are represented by latency and data locality on non-uniform memory access platforms. Furthermore, its special registers enable for optimizing the lock synchronization between cores. MetalSVM is specially suited for fast low-latency inter-core communication exploiting the SCC's message passing buffers. The establishment of shared virtual memory between the 48 cores is currently in development. With a virtualization layer the MetalSVM cores will be able to cooperate together hosting a single Linux kernel. Using virtualization, cores could even be multiplexed or turned off completely for saving energy. Another feature is the planned virtualization of I/O - presented solutions merely virtualized the memory. The Linux kernel will be able to manage the SCC's resources, just like it would on an ordinary SMP architecture. Cache coherence will then be achieved by software [13].

# 3. Hypervisors in Virtualization

This chapter explains virtualization and describes the role of the hypervisor within the virtualization of a computer system. After that, the Intel x86 architecture is considered as the virtualization platform to compare two different virtualization approaches and their adequacy for this project. At last different Linux hypervisors are introduced, outlining the choice for implementing lguest in MetalSVM.

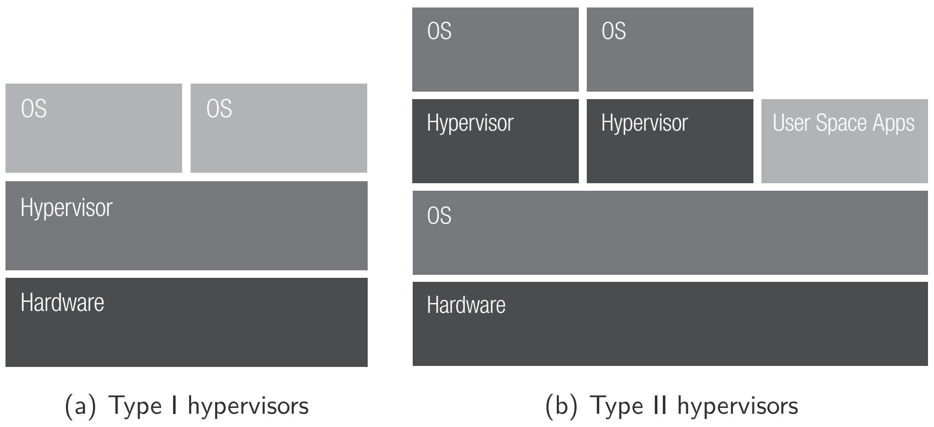
**Definition 1 (Computer Virtualization)** *Virtualizing a computer means creating virtual instances of its hardware which do not physically exist.*

All applications running on such a virtual computer system must run with the identical effect as if they were running on a real machine. In this case, the virtualizing machine is also called *host machine*, while the virtual computer system is the *guest machine*. The program running on the host machine which is creating the virtual computer system environment is referred to as the *virtual machine monitor* (VMM) or *hypervisor*.

In 1973, Robert P. Goldberg distinguished between two types of hypervisors [5]:

**Type I** Bare metal hypervisor (Examples: Xen, KVM, lguest)

**Type II** Extended host hypervisor running under the host operating system (Examples: QEMU, Bochs)



(a) Type I hypervisors (b) Type II hypervisors

Figure 3.1.: The two different types of hypervisors

Type I hypervisors run the virtual machines directly on the hardware. The examples Xen, KVM and lguest do virtualization within kernel space with highest processor privileges. KVM and lguest consist of kernel modules and a user space program [11] [15]. Xen is different and matches the Type I model more obviously: It provides a virtualization layer directly running on the bare metal machine (see also Figure 3.3 on page 11). Even the operating system instance which is used to manage the virtual machine configuration is being run as a guest [19]. This kind of hypervisor is often used in mainframe systems when the real machine is not used for anything else than virtualizing its guests to achieve high performance. Production applications will then be run on the guest machines.

Type II hypervisors are just user space applications being run on the host operating system. They are used when efficiency is only a minor constraint, but can be more flexible by being able to emulate a different processor/computer architecture for example. The lack of efficiency comes from emulating every instruction in software step by step [7].

Performance is an important constraint for this project. Thus, in the following Type II hypervisors will not be minded. The next section analyzes performance aspects of two different methods of Type I hypervisors to find the most suitable kind of hypervisor.

## 3.1. Full Virtualization Versus Paravirtualization

The traditional idea of virtual machine monitors provides an architecture completely identical to the model architecture being virtualized. This method is called *full virtualization*. On the one hand the guest operating system will then be able to run within this environment without the need for modification, but on the other hand this brings much overhead: Because the performance-aware user does not want to emulate every instruction in software, as much instructions as possible should be directly executed on the hardware. The problem here is that many computer architectures' design-goals did not imply virtualization. Operating systems designed for those will usually need to execute a certain set of privileged operations during normal operation. Those have to be *trapped and emulated* in software by the hypervisor. Especially Intel's x86 architecture fits into this group of architectures [14].

Trapping and emulating means switching back from the guest context to the hypervisor whenever the guest tried to execute an instruction which involves privileges it does not have. In general, the hypervisor does not know when and which privileged instructions occur in the future. Therefore, it depends on the programmer to find out how the architecture allows for stopping the guest's execution in the right moment. On some architectures this can end up in complicated workarounds and surely will slow down execution speed.

If the guest operating system can be modified to improve the performance, it can be called into question if it is still necessary to virtualize the whole architecture completely. The *Denali isolation kernel* for example was one of the first VMM

| **Full Virtualization** | **Paravirtualization** |
|---|---|
| no change in guest OS | guest OS must be ported |
| slow due to trap & emulate overhead | fast - no instruction trapping & emulation |
| complex emulation of real hardware | simple & fast virtual device interfaces establishing (VirtIO [16]) |
| operating from user space without privileges (mainly Type II) | embedded into kernel (Type I) |

Table 3.1.: Comparison between full- and paravirtualization

designs with the creation of a more abstract, but still x86-based architecture in mind. The guest kernel did not have to be compiled with special compilers, as it was developed for the x86 architecture instruction set, but it did not contain any privileged operation instruction calls either. This way the hypervisor's overhead of trapping and emulating in software was reduced significantly. The lack of privileged instruction set use was achieved by letting the guest kernel know it is a guest and make it call the hypervisor whenever it needs a privileged operation to be performed.

One important kind of and good example for privileged operation is hardware I/O: A dominant cause for traps and software emulation can be credited against emulating hardware devices. For this purpose Whitaker et al. designed simple hardware device abstractions of their physical counterparts: Drivers for such virtual devices can be split into frontend and backend parts. The frontend part lies within the guest kernel and uses buffers to communicate with the backend driver, which is managed by the hypervisor. Whenever hardware I/O is needed the guest kernel fills the device buffer and issues one single hypervisor call. The overhead of executing a sequence of I/O-instructions which would have to be emulated step by step can be avoided this way. Privileged operations and the overhead of their emulation are therefore kept out of the VMM implementation.

This method is called *paravirtualization* [18] [17]. A summarized comparison between full virtualization and paravirtualization is found in Table 3.1. Calls of a paravirtualized guest operating system to its hypervisor are referred to as *hypercalls*. For every privileged action, one special hypercall must exist. This is part of the common interface host and guest must implement.

The next section reveals in more detail why Intel's x86 architecture is complicated to virtualize. Furthermore it exemplifies where in the application flow paravirtualization achieves reduction of both overhead and complexity within the hypervisor by modifying the guest.

## 3.2. Virtualizing the Intel x86 Architecture

It was already hinted at before, that the Intel x86 architecture is notoriously difficult to virtualize due to its design lacking focus on virtualization. The problems which need to be solved by a hypervisor on this architecture are listed here.

All assumptions about the difficulty of virtualizing x86 processors are not true for current generations due to their support of new techniques for native *hardware virtualization*: With *Intel VT* and *AMD-V*, system programmers have the possibility to run guests within a hardware-provided virtual environment with several overhead-lowering advantages [4]. This thesis covers virtualization mainly for the Intel SCC processor. The SCC's cores originate from the Intel P54C architecture which is too old to support hardware virtualization features. Therefore, these will be ignored here.

The following subsections describe typical x86-mechanisms which need to be handled by the running operating system. A guest operating system also has to handle them, but should not be allowed to access the corresponding instruction calls. After considering the costs of trapping and emulating all of them, it becomes clear why paravirtualization and the guest's virtualization-awareness allow for very much optimization. Instead of trapping every single privileged instruction the guest executes, it is possible to bundle everything what needs to be accomplished by the corresponding code section into one hypervisor call. The following information has been gathered from Intel's official software developer's manual [9] and J. Robin's and C. Irvine's analysis of the Intel Pentium as a virtualization platform [14].

### 3.2.1. Segmentation

A more traditional approach than *paging* (virtual memory) to organize physical memory distribution and access privileges of multiple processes on the x86 platform is *segmentation*. Segmentation enables isolating code, data and stack areas per process from each other. Segments define offset and range within memory space, privilege level, etc. Still until today, the use of paging on 32 bit x86 CPUs is optional – segmentation is not.

So, even with a paging-enabled flat address space the code-, data- and stack segment registers have to be set properly at any time. The code segment, for example, is used by the processor to check if a process is allowed to continue if it tries to use privileged instructions.

Each segment is addressed by an index number in a descriptor table listing all available segments - the *Global Descriptor Table* (GDT). Before a segment is used it has to be described in the GDT. Loading/Storing the GDT with the instructions "sgdt" and "lgdt" demands operating at the highest privilege level. A guest machine must not be allowed to edit entries in the GDT itself directly, because this could be exploited for reaching a higher privilege level. There is also a *Local Descriptor Table* (LDT) for thread-local segments. Thus any access to GDT/LDT needs to be trapped and emulated, what is slowing down execution speed.

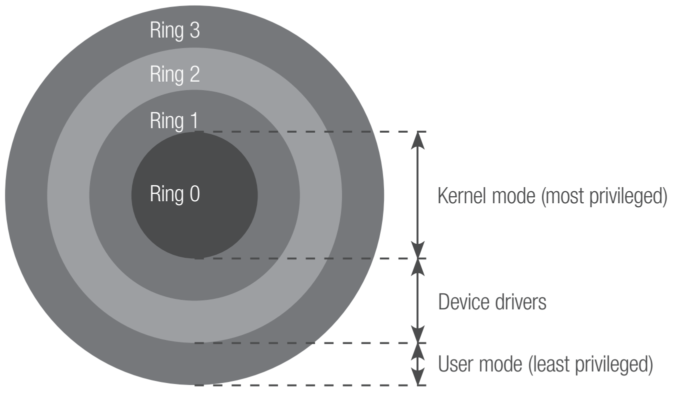


Figure 3.2.: Privilege rings in the Intel x86 architecture

### 3.2.2. Privilege Levels, Rings

The x86 platform partitions privileges to four levels, numbered from 0 to 3. Level 0 stands for the highest privilege level — 3 for the lowest. On Intel CPUs, privilege levels are referred to as *rings* (see Figure 3.2). Usually, the operating system kernel's stack-, data- and code segment use privilege level 0 while user space applications use level 3 for each segment. The segment's descriptor denotes the *descriptor privilege level* number its owners obtain after loading it into a segment register. Loading a segment with a specified descriptor privilege level requires a certain minimum privilege level.

Of course one will not want to let the guest operating system run in ring 0. Most hypervisors choose ring 1 for their guests to enable automatic trapping on privileged operations. The hypervisor must keep track of the segment descriptors to be able to secure ring 0 from the guest. Having trapped a guest which tried to manipulate a segment descriptor, the hypervisor will need to check if the lowest privilege number the guest assigns to its own segment descriptors is equal or above 1.

### 3.2.3. Virtual Memory

Virtual memory/paging involves the CPU using specific data structures maintained by the operating system. The architecture provides special registers (CR0 – CR4) for this purpose.

Again, access to this registers is restricted and must be trapped and emulated slowing down the virtual machine's execution speed. However, the whole page tree structure exposed to the MMU needs to be maintained separated from the guest's page tree. The guest's page tree should be considered unsafe as the guest must be prevented from ever mapping host memory to its address space. The guest could otherwise exploit paging to gain full access to the host system.

### 3.2.4. I/O Operations

The guest should not be allowed to do any I/O operations on the hardware directly and should rather operate on virtual devices. Therefore the hypervisor must trap any I/O access and either simulate the non-existence of I/O devices or emulate virtual ones. The latter method means the overhead of a full software implementation of the according hardware interface.

### 3.2.5. Interrupt Virtualization

As the guest should not have control over all interrupts, both *Interrupt Descriptor Table* (IDT) and *EFLAGS* register (manipulated by the "`pushf`", "`popf`", "`cli`" and "`sti`" instructions) need to be isolated from its access.

Manipulation attempts on the IDT and instructions manipulating the EFLAGS register therefore need to be trapped. Interrupts can be delivered to the guest by setting it into an interrupt situation with preconfigured stack- and segment configuration. This conserves the hypervisor's control, but slows down each interrupt.

Optimization can still be accomplished by delivering interrupts from the hardware directly into the guest without waking the hypervisor. It is important that the hypervisor ensures safe conditions when it installs the according IDT entry.

## 3.3. Hypervisors in Linux

The first free type I hypervisor implemented for Linux was Xen, released in october 2003 at Cambridge.[1] It supports both full virtualization and paravirtualizion. Xen matches the type I hypervisor definition very well: It implements a layer running directly on the hardware and thus it is positioned even beneath the operating system which is used for administration of the virtual machines [19].

Another virtualization solution for Linux is KVM ("Kernel-based Virtual Machine") which was released in October 2006. Also being a type I hypervisor, it does not match the type I definition as clearly as Xen. There is no *KVM layer* below the host operating system. KVM VMs will be run within an instance of QEMU utilizing a KVM Linux kernel module. This way, execution speed and privilege critical parts of the VM can be run on the bare metal machine by the host kernel [11].

The simplicity of loading a single set of modules (KVM) instead of reconfiguring the whole system to support virtualization (Xen) caused KVM's popularity to raise over Xen's.

Guest Linux kernels running under both hypervisors need to reimplement privilege-critical kernel functions to use hypercalls to delegate privileged tasks to the hypervisor. On the Kernel Summit in 2006[2] (still before KVM was published), Paul Russell (known under the name of *Rusty* Russel) proposed a simple solution to unify

[1]Official announcement: `http://sourceforge.net/mailarchive/message.php?msg_id=5533663`

[2]`http://lwn.net/Articles/191923/`

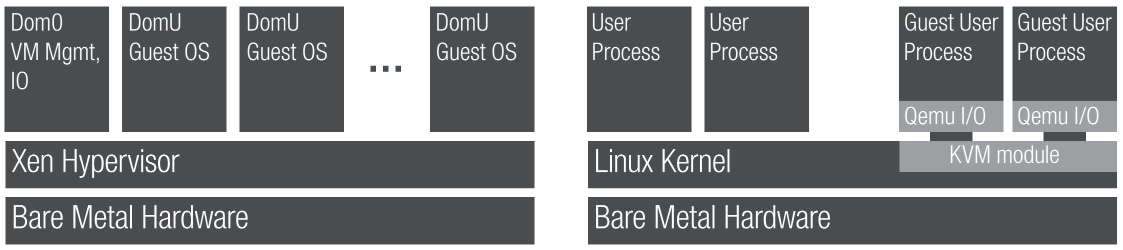


Figure 3.3.: Structural differences between the Xen and KVM hypervisors

the paravirtualization interface in Linux as a guest-side kernel: A structure called `paravirt_ops` embodying a set of function call pointers to system-critical functions.[3] Those can be configured at boot time after detection of the right hypervisor to match its interface. This way the same kernel can run on a real machine or on one of the supported hypervisors without recompiling, which until today are Xen, KVM, VMWare's VMI and *lguest* [12].

## 3.3.1. Lguest

Lguest is somewhat different compared to the other hypervisors. Rusty Russell wrote it at first to show how simple and useful his `paravirt_ops` proposal is. In the end it turned out to be a complete hypervisor with I/O-drivers for block devices and network. Without features besides the core functionality of a hypervisor and implemented with just about 5 000 lines of code, it is quite spartan. Just like KVM, it brings its own Linux kernel module and guest kernel loader terminal application. It is very limited due to its lacking support for SMP at guest side, 64 bit kernels and suspend/resume features. Nevertheless it will run on virtually any x86 CPU because it does not depend on Intel VT or AMD-V – KVM does [15].

Another simplicity-enabling design choice of lguest is its lack of an *application binary interface* (ABI): It was not developed around a bold and complicated interface, whereas ABIs define strict and extensive conventions like data types, sizes and alignments. This makes lguest perfect for experimenting.

Being a 32 bit architecture and not providing Intel VT extensions, the Intel SCC perfectly matches into lguest's set of target platforms. Also, due to its simplicity and comfortably modifiable design, it was chosen as the right hypervisor for being ported to MetalSVM.

[3]Introduced by patch: `https://github.com/torvalds/linux/commit/d3561b7`

# 4. Design and Implementation of the Hypervisor for MetalSVM

Lguest's implementation in MetalSVM is very similar to the original implementation in Linux. This chapter explains the implementation in the MetalSVM kernel and outlines structural differences to the Linux kernel. Lguest does in general consist of five parts – namely the context switcher, the user space loader application, the general hypervisor module itself, the `paravirt_ops` structure, and VirtIO front- and backend drivers.

The *hypervisor module* embodies the host's functionality to maintain the virtualization environment for the guest kernel. Both host and guest share the *context switcher* code page which manages the change between their execution contexts. Implementing the paravirtualization interface between host and guest, the `paravirt_ops` structure and the functions/procedures it points to are located in the guest kernel. Device driver I/O is implemented at host and guest side in the backend and frontend part of the VirtIO driver implementation. Finally, the virtualization functionality is provided to the user via the user space *loader application*. Table 4.1 gives a brief picture of which part of the code is executed at host or guest side:

| **Host side** | **Guest side** |
|---|---|
| Context Switcher | |
| User Space Loader | paravirt_ops |
| Hypervisor module | |
| VirtIO backend driver | VirtIO frontend driver |

Table 4.1.: Lguest's main parts

Context switcher and loader will be explained in their own sections later. The Hypervisor module is more complex and is split into several sections, while more general explanations will be found here.

To make clear how the parts work together, the application flow starting with the boot of the real machine will be described in sequential order until the guest operating system shell is started:

1. The reset pin of the real machine is released.
2. MetalSVM starts booting.

3. MetalSVM initializes the whole system and calls `lguest_core_init()`[1] at last.
4. The lguest initialization call creates a character device at `/dev/lguest` in the file system tree.
5. The host system is initialized and user space applications will be started now.
6. Lguest_loader is the user space application which does the virtualization in cooperation with the MetalSVM kernel.
7. After lguest_loader initialized the virtual memory environment for the guest, it informs the MetalSVM kernel using the `/dev/lguest` interface.
8. Kernel side initialization for the new guest is done.
9. Lguest_loader will now enter its main loop where it reads from `/dev/lguest` and handles its output continuously. (see Figure 4.4 on page 23)
10. The virtualization itself happens in kernel space during each `read(/dev/lguest)` call: In general, the hypervisor switches to guest execution and handles interrupts and other events afterwards. This is done in a loop which is explained in the following.
11. The `read()` system call returns whenever any virtual device I/O (VirtIO) needs to be processed. System crashes and shutdowns also end up this way.

The virtualization loop in kernel space within `run_guest()` (see Listing B.1) mentioned in step 10 is visualized in Figure 4.1. Rectangles with rounded corners denote actions on which `run_guest()` either breaks out of the loop to return to user space, or to stop execution.

- `lguest_arch_run_guest()`[2] performs the context switch to the guest system. Any interrupts must be disabled before entering.
- Whenever an interrupt occurs `lguest_arch_run_guest()` returns. Traps will be processed in `lguest_arch_handle_trap()`[2], interrupts have to be delivered to the guest at the next context switch.
- Hypercalls are processed by `do_hypercalls()`[3]. As it is possible to bundle multiple asynchronous hypercalls within a list, they will be processed in a loop.
- A *notify* event occurs on virtual guest device I/O. In this situation `run_guest()` returns back to user space, as the backend drivers are not implemented in the kernel.

[1]defined in `metalsvm/drivers/lguest/lguest_core.c`
[2]defined in `metalsvm/drivers/lguest/x86/core.c`
[3]defined in `metalsvm/drivers/lguest/hypercalls.c`

- If the last interrupt stopping the guest machine was an interrupt which should be delivered back to it, this will be done just before entering the guest again: The interrupt can be delivered by forming the guest stack just like it would look in case of a real interrupt.
- Another cause to return to user space are cases which cause the guest machine to stop – like crashes, shutdowns, and reboot attempts.
- The virtualization process can be made subject to being rescheduled, if the guest halts in inactivity waiting for further interrupts, unless there are interrupts waiting to be delivered.

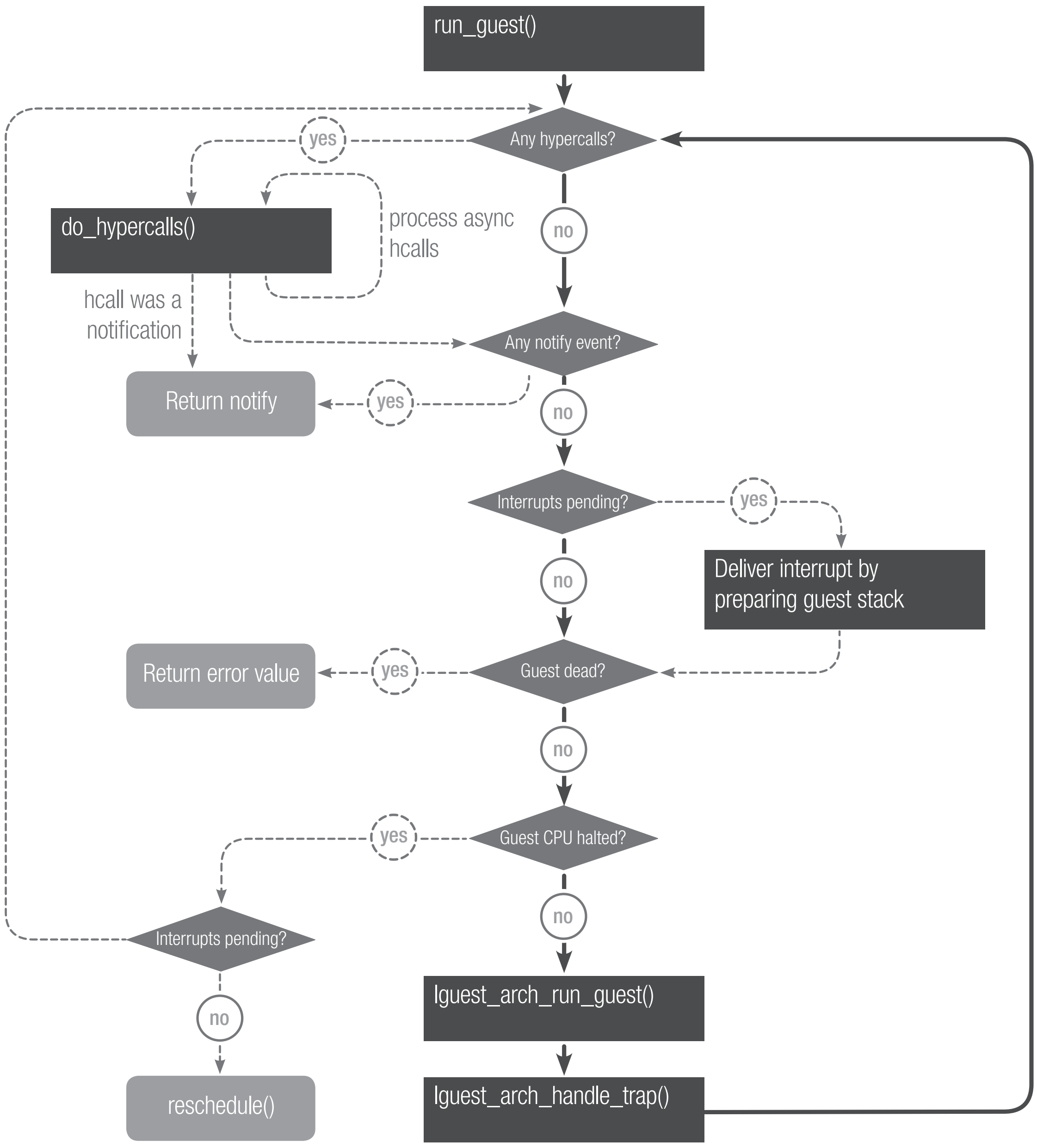


Figure 4.1.: Flow of lguest's main loop

## 4.1. Hypervisor Initialization

Before being able to provide a virtualization interface for the user space guest loader, some initialization in the MetalSVM kernel is done at boot time. Just like UNIX operating systems do, MetalSVM provides a `/dev` directory in the filesystem where it keeps user space accessible files representing devices. The corresponding lguest-file is a character device both accepting and emitting character streams called `/dev/lguest`. This is done in the `lguest_core_init()`[4] procedure.

The whole initialization process consists of 4 parts:

1. Loading the switcher pages
2. Initializing the part of the pagetables every guest initially uses
3. Creating the character device file `/dev/lguest` in the file system
4. Initializing architecture-specific data structures of both host and future guests

### 4.1.1. Loading the Switcher Pages

The *switcher pages* area in the host's and guest's memory space consists of three pages: One page containing the actual code which manages the context switch between host and guest and two further pages which are for storing host and guest states. If the host is an SMP system, the both pages for storing host/guest states exist once per every CPU in the system, because multiple guests can be run at the same time.

Initializing those means allocating memory for an array keeping pointers to every page and for the pages themselves. The first allocated page is then used for containing the switcher code. It will be copied from where it is initially located in the kernel to the beginning of this page.

After that, the first switcher page containing the context switch instructions is mapped to `SWITCHER_ADDRESS` on every cpu's address space. The per-CPU-pages are mapped behind this first switcher page on the corresponding CPUs. `SWITCHER_ADDRESS` is a constant value which is set to 0xFFC00000 in the most cases, but can be different. As those pages also will be mapped in the guest address space at the same address location, the very end of the 32 bit address space is an appropriate location.

### 4.1.2. Initializing Page Tables

Early at boot the guest kernel will be instructed not to use any page from between `SWITCHER_ADDRESS` to the very end of the memory space. This leads to the same few last page directory entries in every guest space if several guests are running at the same time. The corresponding page tables are created now.

[4]defined in `metalsvm/drivers/lguest/lguest_core.c`

To secure the host state keeping memory structures while the CPU is operating in guest context, the switcher page containing the host state is mapped read-only in the guest's page tables. Of course the context switcher code page is mapped read-only, too. These page directory and page table entries will be shared between all guests. The guest kernels will not map their own pages into this area.

### 4.1.3. Creating /dev/lguest in the Filesystem

MetalSVM provides functions and data structure definitions to enable the abstraction of system functions to file system I/O character streams. Therefore the `/dev` filesystem node is found by using `finddir_fs(fs_root, "dev")` and then a new node with the filename `lguest` is initialized and installed within the former one.

The special attributes of this file system node are its *character device* type and the corresponding pointers to functions for the system calls `open()`, `close()`, `read()` and `write()`. Utilizing these, lguest provides the interface for communication with its user space application.

### 4.1.4. Architecture-Specific Initialization

The initial interrupt descriptor table of all guests is the same. Before being ready to use, the interrupt handler's address pointers within every entry need to be relocated with the offset of the actual switcher page address.

Afterwards, the IDT, GDT and Task State Segment (TSS) descriptors are set up to values which are the same for every initial guest. The TSS contains a bitmap denoting which I/O ports the corresponding running process is allowed to use. This bitmap's fields are all set to a value representing the meaning "*none*", because the guest must not do any I/O by itself.

While switching the context and generally later in guest kernel space, host/guest kernels use the same segment numbers for their code and data segment (notably `LGUEST_CS` and `LGUEST_DS`). Their descriptors are configured with the usual flags corresponding to code/data segments, but the guest version of those is not configured to make the CPU operate on privilege ring 0, of course. These descriptors are configured here and the host version is installed into the host GDT.

Every context switch from host to guest will be started with an `lcall` instruction later. As the only parameter to this instruction is a structure keeping the target code's address offset and segment, this structure is initialized now to the switcher code address and `LGUEST_CS` segment.

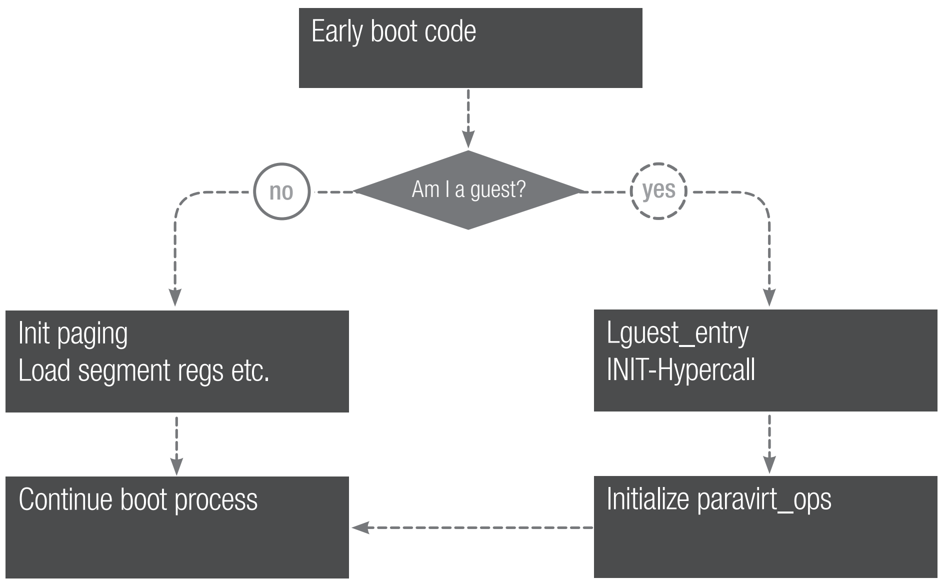


Figure 4.2.: Hypervisor-recognition at boot time

## 4.2. Guest Boot and paravirt_ops

Of course, as in paravirtualization host and guest have to cooperate, the guest must know it is a guest running under control of a hypervisor instead of running alone on the bare metal system.

Early at boot, the Linux kernel will enter its `startup_32` routine[5]. It clears the uninitialized C variables by writing zeros over the whole *BSS* range. Then it moves the boot header structure (`boot_params`) as well as the kernel command line string to another place in the memory. If paravirtualization guest support was enabled at compile time, it will then check the `boot_params` structure's `subarch` field[6] (see Listing B.2).

If this field's value is 1[7] then it recognizes itself as a guest of the lguest hypervisor and calls `lguest_entry`[8]. This procedure steps over the initialization of paging etc. since this is already done by the hypervisor (see Figure 4.2). The first thing to happen in `lguest_entry` (see Listing B.3) is the initialization hypercall to the host. The only parameter is the address of the guest's `lguest_data` structure. Now the host can initialize the guest's clock, frequency setting and inform the guest about its allowed memory dimensions by writing this information back into that structure.

Back in guest kernel space, the kernel reaches its first instructions generated of C-code while entering `lguest_init()` (see Listing B.4).

The `paravirt_ops` structures' function pointers are set to the right paravirtual-

[5]defined in `linux/arch/x86/boot/compressed/head_32.S`
[6]Introduced by patch: `https://github.com/torvalds/linux/commit/e5371ac`
[7]Value 0 stands for a normal PC architecture without any hypervisor, 2 stands for Xen
[8]defined in `linux/arch/x86/lguest/i386_head.S`

ization procedure calls. The kernel is now able to cooperate with its host and will not try to execute instructions which require a higher privilege level. Between some more hardware registration the virtual console device is registered and callbacks for crash situations and shutdown are set.

Now the `i386_start_kernel()`[9] procedure is called and the kernel finally continues executing its usual boot code.

[9]defined in `linux/arch/x86/kernel/head32.c`

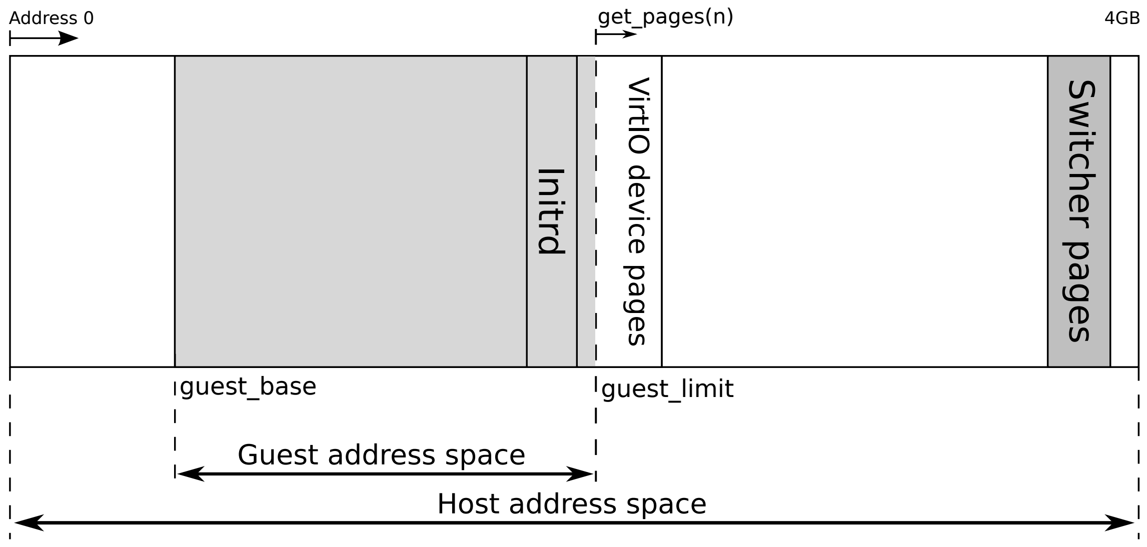


Figure 4.3.: Memory layout of host and guest

## 4.3. Lguest Loader

The whole virtualization is initiated by a simple user space process. In this implementation the program is called *lguest_loader*. It takes the following command line parameters:

- Guest kernel image file system path
- The guest's initial ramdisk image file system path
- The guest's physical memory size
- Guest kernel command line parameters (optional)

Subsequently, it will allocate the guest's later physical memory using `malloc()`: This sector's size is the size parameter the user provided plus the *device pages* (default constant value is 256 pages) size. This area needs to be aligned to page boundaries. In the following, the address of the beginning of this area will be denoted by `guest_base` and the size of this area without the device pages by `guest_limit` (see Figure 4.3).

For the user space process the guest's physical address space is just this linear range. Hence, to calculate from host-virtual to guest-physical addresses it is only required to subtract `guest_base`—or to add this value to convert a guest-physical address to host-virtual.

Loading the kernel image file is a straightforward process: At first, the file is checked for *magic strings* indicating it is an *ELF* file, then its *ELF-header* is checked. Finally, the *ELF load sections* are loaded to the proper positions in the guest-physical memory. The initial ramdisk is loaded to the end of the guest's memory area.

At current development state, the loader only implements a virtual console device which is initialized then. Due to the complexity of the *VirtIO* interface, details on the interface itself will be explained later. The virtual interfaces utilize buffers which need to be mapped at host- and guest-side. The function `get_pages()` obtains memory for this buffers by pushing the `guest_limit` border deeper into the VirtIO device pages area.

The last step before telling the hypervisor about the guest is providing a proper `boot_params` structure. This structure is placed at guest-physical address 0 and contains the following information:

- An *e820 memory map* with an entry specifying the guest's physical memory size
- A pointer to the kernel's command line parameters (which are copied into the guest memory just behind the `boot_params` structure)
- If provided, the initial ramdisk image's position and size
- The `hardware_subarch` field (as already described in the section before) is set to *1* (Indicating the presence of the lguest-hypervisor). Early at boot time, the guest kernel will check this value and recognize that it is running under a hypervisor
- There is also a variable `loadflags` where the `KEEP_SEGMENTS` flag is enabled to hold the booting kernel off reloading the segment registers since the hypervisor will have it set properly already

Telling the hypervisor about the new guest is done by writing information to the `/dev/lguest` file: This information consists of 16 bytes which contain the following:

- an enumeration value `LHREQ_INITIALIZE` to denote the format of the next written bytes
- the host-virtual `guest_base` address
- the number of pages between `guest_base` and `guest_limit`
- the guest-physical start address of the kernel executable.

In the meantime, in kernel space, the hypervisor allocates all structures it needs to manage the guest and initializes the guest's registers (by choosing the right segments, EFLAGS, instruction pointer, etc.), its GDT and initial page mapping in the shadow pages. Before boot, the initial page mapping contains identity mapping for all guest-physical pages. Linux kernel space mappings begin at an offset of `PAGE_OFFSET` which in x86 is at address 0xC0000000 and are initially mapped to page frames 0 and following. The *switcher pages* (see Figure 4.3) are mapped at the same address as in the host.

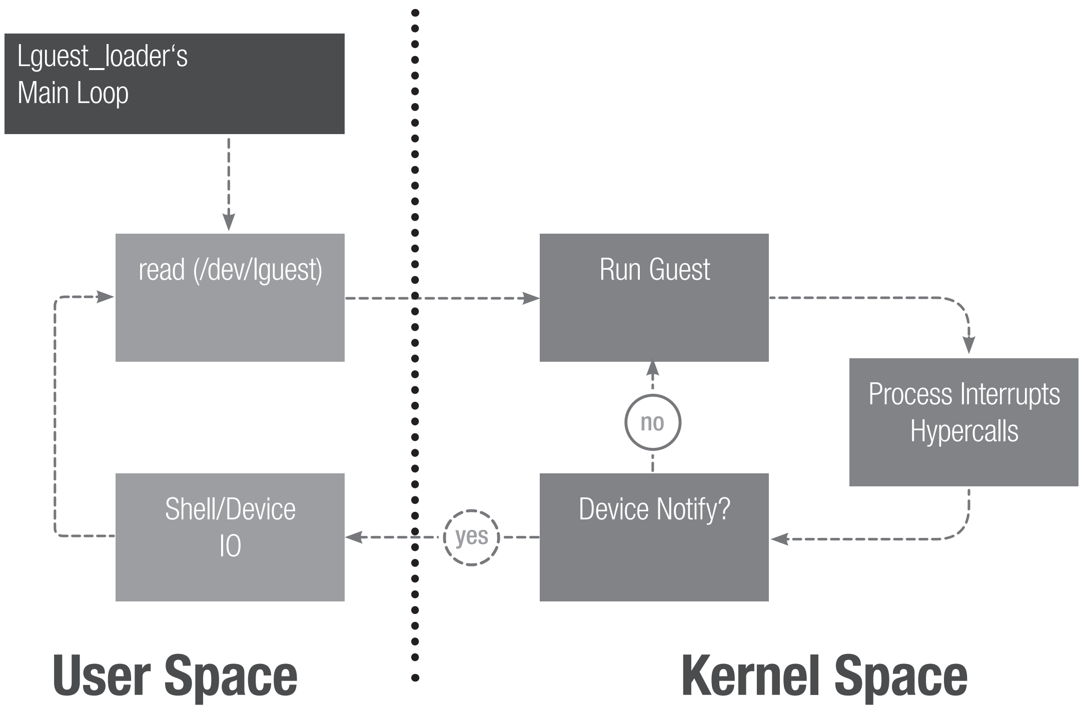


Figure 4.4.: Flow of lguest_loader's main loop

After returning to user space from the `write()` call to `/dev/lguest` everything is ready to start the guest machine. Lguest_loader will now enter its `run_guest()` procedure which merely contains an infinite loop where a `read()` call on `/dev/lguest` and processing code for its output is located. (see Figure 4.4)

For the whole lifetime of the guest system, the virtualization will take place while lguest_loader is blocking on the `read()` call. This call returns to user space whenever critical error messages or virtual device events for device output occur or if the guest just wants to shut down or reboot. The guest kernel also emits boot messages this way before its virtual console is initialized.

## 4.4. Host-Guest Context Switch

Switching the context from host to guest and back requires saving all register states of one machine and restoring the state of the other machine. On the x86 architecture this also involves loading the right segments, a different GDT, IDT and TSS as well as switching the MMU to another page tree. In the end the whole context switching procedure should also be NMI-safe. The code which manages this task is usually mapped to address 0xFFC00000, because this way the last page directory entry can be used for the switcher code and the state structures of host and guest. The Linux kernel does also have an internal variable `__FIXADDR_TOP` which is used to specify the last mappable virtual address. Both Linux host and guest kernels do this to protect this area. Due to architectural restrictions, MetalSVM uses address 0xFF800000.

To switch from host to guest, the hypervisor just needs to call the `run_guest_once()` procedure (see Listing 4.1) which first calls `copy_in_guest_info()` to copy the guest state information into a data structure mapped just behind the switcher page. Every core on the system has its own structure of this kind. On single core systems this is rather uninteresting, as this procedure does only copy conditionally: Either if a different guest than before is run on the same core or if a guest is moved to another core.

```
 1  static void run_guest_once(struct lg_cpu *cpu, struct lguest_pages *pages)
 2  {
 3      unsigned int clobber;
 4
 5      /* Copy over guest-specific data into this cpu's struct lguest_pages */
 6      copy_in_guest_info(cpu, pages);
 7
 8      /* Set trapnum to impossible value. In case of NMI while switching
 9       * this will help to identify the situation */
10      cpu->regs->trapnum = 256;
11
12      asm volatile("pushf;␣lcall␣*lguest_entry"
13              : "=a" (clobber), "=b" (clobber)
14              /* EAX contains the pages pointer,
15               * EBX contains physical address of guest's page directory */
16              : "0" (pages),
17                "1" ( cpu->lg->pgdirs[cpu->cpu_pgd].pgdir)
18              : "memory", "%edx", "%ecx", "%edi", "%esi");
19  }
```

Listing 4.1: run_guest_once() procedure (metalsvm/drivers/lguest/x86/core.c)

The `trapnum` variable within the core's `regs` structure (line 10 in the code listing) is used to save the last interrupt number from which the guest was interrupted. A value of 256 is impossible, because the x86 only allows for numbers 0 to 255. This is used to detect NMIs which occured during the context switch: If an NMI occurs, the machine will immediately switch back just like on any other interrupt. After returning from `run_guest_once()` the hypervisor will call `lguest_arch_handle_trap()` which will detect the NMI and drop it. It is unlikely that the next host-guest switch fails, too.

Finally, in line 12 of the code listing, the switcher code itself is called. The *eax* register is set to the current core's `struct lguest_pages`'s address and *ebx* is set to the guest's page directory address. `lguest_entry` is a structure containing the switcher code's entry offset and the segment selector `LGUEST_CS` the code shall be executed on. Now, the `switch_to_guest` assembler routine (see Listing 4.2) which just has been called shall be regarded line by line:

**Lines 3f** push the host's segment registers and frame pointer onto the stack. They will be popped back when returning to the host.

**Lines 10f** The task state segment register must be pushed onto the stack. Linux' original lguest implementation does not need to do this, because the Linux kernel uses only one task state segment per core. MetalSVM utilizes hardware task switching, therefore this TSS's descriptor number can be one of several possible ones and needs to be saved.

**Line 13** The host's stack pointer is saved in the `lguest_pages` structure.

**Lines 15f** As *eax* contains the `struct lguest_pages`' address, it is only necessary to add the offset that the `regs` field has within the struct to get the address of the guest's set of registers. The stack pointer is set to this address - this way the guest stack instead of the host stack is modified by a possibly occuring NMI.

**Lines 19f** load GDT and IDT register values from `lguest_pages`.

**Lines 23f** The guest's TSS register must be loaded before switching the CR3 register to the shadow page tree. Intel x86 CPUs alter the TSS descriptor by setting a *used-bit* after loading with the "`ltr`" instruction. To protect from loading a TSS twice, issuing the "`ltr`" instruction on a TSS with a used-bit, which is already set, provokes a general protection fault. Therefore, this change in the host's TSS has to be reverted immediately. After changing the CR3 register to the shadow page trees, the `struct_page` which has been used to save the host's state is only mapped read-only to isolate host from guest. The used-bit of the previous TSS is not changed automatically by issuing the "ltr" instruction - it is more common to switch TSS registers at a task switch with a far jump denoting both the next task's instruction pointer and its TSS. This is not possible at this point.

**Lines 26f** load the host-GDT address into the *edx* register. So, $edx + ecx + 5$ is the address of byte 5 in the host's TSS descriptor, where bit 2 is the *used-bit* which needs to be cleared.

**Line 29** loads the shadow page tree address into the CR3 register. Now the page with the host state structure is mapped read-only and the guest cannot harm it.

**Lines 31f** All the guest registers are simply popped off the stack.

**Line 43** steps over the `trapnum` and `errcode` fields in the `lguest_regs` structure. These fields are needed elsewhere to save interrupt number and error code before switching back to the host.

**Line 45** Entering the guest code looks like returning from an interrupt. Finally, the context switch is complete.

```
1   ENTRY(switch_to_guest)
2
3       pushl   %ss
4       pushl   %es
5       pushl   %ds
6       pushl   %gs
7       pushl   %fs
8       pushl   %ebp
9
10      str     %ecx
11      pushl   %ecx
12
13      movl    %esp, LGUEST_PAGES_host_sp(%eax)
14
15      movl    %eax, %edx
16      addl    $LGUEST_PAGES_regs, %edx
17      movl    %edx, %esp
18
19      lgdt    LGUEST_PAGES_guest_gdt_desc(%eax)
20
21      lidt    LGUEST_PAGES_guest_idt_desc(%eax)
22
23      movl    $(GDT_ENTRY_TSS*8), %edx
24      ltr     %dx
25
26      movl    (LGUEST_PAGES_host_gdt_desc+2)(%eax), %edx
27      andb    $0xFD, 0x5(%edx, %ecx)
28
29      movl    %ebx, %cr3
30
31      popl    %eax
32      popl    %ebx
33      popl    %ecx
34      popl    %edx
35      popl    %esi
36      popl    %edi
37      popl    %ebp
38      popl    %gs
39      popl    %fs
40      popl    %ds
41      popl    %es
42
43      addl    $8, %esp
44
45      iret
```

Listing 4.2: Host-guest switcher code (metalsvm/drivers/lguest/x86/switcher_32.S)

Switching back from guest to host is very similar, hence not explained here except for a detail: Ensuring to find the address of the host's state structure within `struct lguest_pages`. This must success even if an interrupt occurs while restoring the guest's registers within the switch process from host to guest. This situation is tricky:

Listing 4.3 shows the definition of `struct lguest_pages` and two assembler lines extracted from the routine which switches back from guest to host. Whenever an interrupt occurs at guest's side the TSS sets the stack pointer back to within `struct lguest_regs` in `struct lguest_pages`. This way, trap number, error code and registers are automatically saved into the `lguest_regs` structure by the CPU. Now it is a crucial task to find the read-only host state structure to restore the host context. The only hint lies within the stack pointer: As it was set to the last ring 0 stack position in `struct lguest_regs` by the TSS, it is only a matter of offset-calculation to find the read-only host state structure.

The only problem left is that it is not safe to assume that the stack has the same well-defined height on every context switch back to the host. If a fault occurs while the guest registers are popped off the stack while switching from host to guest, a simple offset-calculation to find the read-only host state structure will not work out any longer. This problem is solved by aligning `struct lguest_pages` at page-boundary and putting its `struct lguest_regs` to the very end of the first page it occupies. The space between is filled with spare bytes. With this layout, the stack pointer's lowest 12 bits can be just cleared out to obtain the `lguest_page`'s base address (see the assembler code in the listing). Using this as base address, simple offset calculations can be applied again to find the read-only host state structure.

Combined with comprehension of this detail the rest of the context switch code switching back to the host is finally analogous to the switch from host to guest.

```
/* lguest.h */
struct lguest_pages {
    /* This is the stack page mapped rw in guest */
    char spare[PAGE_SIZE - sizeof(struct lguest_regs)];
    struct lguest_regs regs;

    /* This is the host state & guest descriptor page, ro in guest */
    struct lguest_ro_state state;
} __attribute__((aligned(PAGE_SIZE)));

/* switcher_32.S */
    /* ... */
    movl    %esp, %eax
    andl     $(~(1 << PAGE_SHIFT - 1)), %eax
    /* ... */
```

Listing 4.3: Definition of the lguest_pages structure and restoring the stack pointer during guest-host context switch

## 4.5. Segments

Segments are the heart of lguest's guest-isolation functionality, because the loaded code segment defines at which privilege level the CPU is currently running on. It is very important to keep track on the changes the guest tries to write to the global descriptor table. If necessary the hypervisor needs to reject or alter them to hold the guest off from obtaining the host's privilege level.

The highest privilege level the guest can obtain during its lifetime is the ring 1 privilege level. To achieve this, the guest's kernel space code and data segments have to be manipulated. These and the task switch segment (TSS) as well as the doublefault TSS need to be secured from any write-access by the guest.

### 4.5.1. Special Segments for the Guest's Kernel Space

Most important for kernel space are the `LGUEST_CS` and `LGUEST_DS` segments which are to be used by the code and data segment registers.

`LGUEST_CS`'s segment descriptor has the same flags like the `KERNEL_CS` segment descriptor on which Linux kernel space code is usually running on. The only difference is the descriptor privilege level flag which is set to a value of 1 to let the guest run on ring 1. The same applies respectively for `LGUEST_DS` and `KERNEL_DS`, which is generally used for data-, extra- and stack-segment registers. The host kernel has its own versions of `LGUEST_CS` and `LGUEST_DS` with ring 0 set as descriptor privilege levels. These are loaded on entering the context switch code.

As the TSS preserves the old code segment and determines which segment shall be loaded to the stack-segment register etc., it is considered to be too powerful to be manipulated by the guest. The hypervisor ensures that it never switches the guest to ring 0.

### 4.5.2. Managing GDT Manipulation Requests

The guest has no write-access to the GDT because it is mapped read-only in its virtual address space. Therefore there are two hypercalls available for it to manipulate segment descriptors: `LHCALL_LOAD_GDT_ENTRY` and `LHCALL_LOAD_TLS`.

`LHCALL_LOAD_GDT_ENTRY` takes an 8 byte segment descriptor as parameter. The hypervisor will then install it into the GDT which is destined to be active while guest execution. Of course the hypervisor will not accept to install segment descriptors which could enable the guest in obtaining full privileges.

If the proposed segment descriptor would give the guest full privileges the hypervisor just overwrites the descriptor privilege field (DPL) with a value of 1 standing for ring 1. The *accessed-bit* within the descriptor's type field will be set by the hypervisor, too. If it is not set in the moment the guest tries to load the segment, the CPU would try to set it and fail, because it has insufficient privileges.

Mentioned fields the hypervisor changes within the segment descriptor are shown in Figure 4.5. There are four segment descriptors to which the guest has no access,

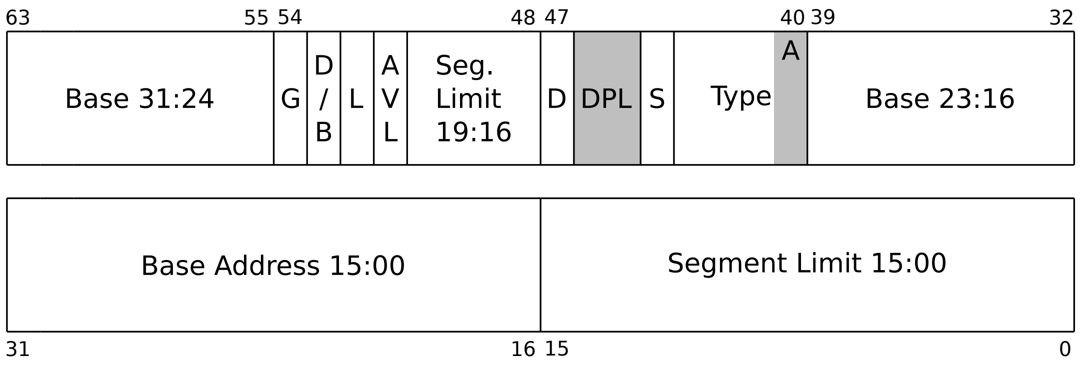


Figure 4.5.: Structure of a segment descriptor [9]

thus the hypervisor rejects any attempt to change them: The TSS, `LGUEST_CS`, `LGUEST_DS` and doublefault TSS segment descriptors.

`LHCALL_LOAD_TLS` is a hypercall for optimization, as the three thread-local storage segment descriptors need to be changed on every context switch and will be processed by the hypervisor in a row.

## 4.6. Interrupts

The guest machine depends on the interrupt and exception functionality of the CPU, just like any other machine. Of course, the host must stay in full control over the interrupt/exception mechanism at any time to ensure the system's security and integrity. Also, the host needs to be able to reschedule the guest's virtualization process to serve its other processes with CPU time equally well.

Initially, all interrupts/exceptions which occur while the guest is running cause a context switch back to the host. Depending on the kind of interrupt it is delivered to the host just as if it occured while the host was running. Exceptions are delivered to the hypervisor for further analysis. To enable this distinction, a special interrupt descriptor table is installed when switching the context to the guest: Its vector number entries point to three different handlers:

**Vector numbers 0, 1, 3...31, 128** Most of these are architecturally defined exceptions, which are issued by the CPU if errors occur or special instructions are used. The operating system has to handle them to keep the ability to run properly. Handling these interrupts mostly involves the need for higher privileges. Thus, the interrupt/exception number is pushed onto the stack and delivered to the hypervisor.

Vector number 128 is typically used for system calls, which are issued by the guest's user space processes, so they should not be delivered to the host operating system but instead sent back to the guest.

**Vector number 2** Used for Non-maskable interrupt: The handler for this does nothing else than returning to normal execution of the guest code.

**Vector number 32...127, 129...255** Being issued by some external cause or by software, interrupts assigned to vector numbers within this range need to be delivered to the host operating system and not the hypervisor code itself. For this purpose the context is switched back like usual, but in the following the host's IDT is searched for the current interrupt's IDT descriptor to jump to the right handler. After handling the interrupt the hypervisor continues execution on host side as if it returned from switching back from the guest as usual.

### 4.6.1. Managing IDT Manipulation Requests

The guest can issue an `LHCALL_LOAD_IDT_ENTRY` hypercall to the host to tell where it wants an interrupt or exception to be delivered for handling. The hypervisor will then write the segment descriptor the guest provides into the guest-IDT. Before the hypervisor accepts a segment it checks some attributes and its privilege level:

- The guest is not allowed to change interrupt descriptors of the vector numbers 2, 8, 15 and `LGUEST_TRAP_ENTRY` (commonly 0x1f). These are for NMI, double

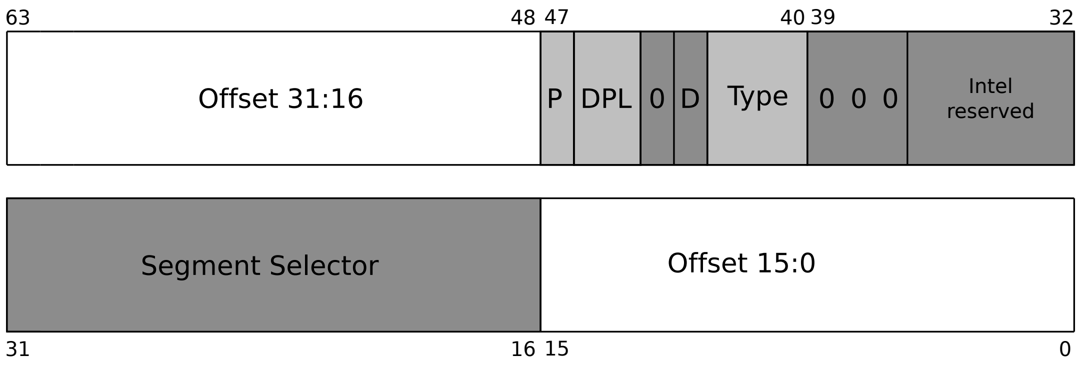


Figure 4.6.: Structure of an interrupt descriptor [9]

fault, the *spurious interrupt*[10], and lguest-Hypercall. The guest will never have to handle them by itself.

- Solely *interrupt*- and *trap-gate*-types are allowed as segment descriptors.
- Just in case the guest tries to install an interrupt descriptor with a privilege level lower than 1 (that means assigning a higher privilege than allowed to the interrupt handler) the hypervisor overwrites this flag to ring 1 priority.

All in all, the hypervisor copies only the interrupt handler's address, present-bit, privilege level and descriptor type into the IDT. Segment descriptors with an unset present-bit are zeroed out completely. As seen in Figure 4.6 the fields which are completely overwritten by the hypervisor are colored dark grey and the fields which are just checked are colored light grey.

### 4.6.2. Guest-Interrupt Handling in the Hypervisor

Immediately after every context switch back to the host, the hypervisor calls the procedure `lguest_arch_handle_trap()`[11] to determine the kind of interrupt/exception which caused the guest execution to stop. This procedure does in general consist of one switch-case construct handling the following Intel-defined interrupt vector numbers:

**Vector number 13** General protection fault - This can be caused by the following:

- I/O instructions - As Linux' paravirtualization structure is not absolutely complete, the guest tries to probe I/O ports at boot time. The hypervisor lets the guest just skip them and write return values to the guest's registers with all bits set which is the common value to signalize that this I/O port is not used.

[10]Spurious interrupts do occur because of noise and other conditions. They are not supposed to need special handling.

[11]defined in `metalsvm/drivers/lguest/x86/core.c`

- The `vmcall` instruction - If the Intel VT extension is not available or available but not initialized (lguest does not support it, therefore it is not initialized), this instruction causes a general protection fault. It will be redirected to the exception 6 handler.
- If the general protection fault was not due one of the causes above it will be redirected to the guest later.

**Vector number 6** Invalid opcode - If this exception was caused by a `vmcall` instruction this is either a rerouted vector number 13 or 6 on a CPU not supporting the Intel VT extension. In both cases the 3 bytes long instruction "`vmcall`" will be patched with another 3 bytes "`int 0x1f; nop`". In any other case it is delivered back to the guest later.

**Vector number 7** Device not available fault - This happens if a guest process tried to utilize the floating point unit. The hypervisor manages its register state storing and restoring.

**Vector number 14** Page fault - The hypervisor will refresh the corresponding shadow page tree to fulfill the MMU's demand. If it cannot find a corresponding entry within the guest's page tables while it is trying to achieve this, the exception will be delivered to the guest which has to refresh its own page tree.

**Vector number LGUEST_TRAP_ENTRY (0x1f)** Hypercall - The guest issued a hypercall and its parameters lie within the processor's all-purpose registers which already have been saved in the guest processor state structure. The hypervisor sets a pointer to this structure of parameters which will be checked in its hypercall-processing section later.

If the hypervisor decides to send one of the interrupts/exceptions above back to the guest or an interrupt occured which has a vector number lower than 32 but not listed above, it will try to deliver it to the guest now.

### 4.6.3. Delivering Interrupts Back to the Guest

Interrupt delivery to the guest works by simulating the processor's real actions in the case of an interrupt/exception before letting the guest execute its code again. The hypervisor achieves this by forming the stack in the same way the processor does when it is interrupting the execution.

There are different cases and types of interrupt/exception which the stack has to reflect before calling the handler routine:

- It makes a difference on which privilege level the processor is running before it is being interrupted:

  If it is running in kernel space, the processor will save the *EFLAGS* register, code segment and instruction pointer onto the stack and jump to the handler (see Figure 4.7(a)).

As user space processes do not have sufficent privileges to handle interrupts/exceptions, a privilege level switch to a higher privilege (lower ring number) is necessary[12]. The TSS arranges a switch of both the stack pointer and stack segment to the kernel stack. While the user space stack stays unchanged, the former user space stack segment and stack pointer are pushed onto the kernel stack. After this the same registers are pushed onto the stack just like in the kernel space interrupt case described before. (see Figure 4.7(b))

- Vector numbers 8, 10 to 14, and 17 are used for exceptions which supply error numbers. These are pushed onto the stack by the processor after saving the register states. The hypervisor has to distinguish if it needs to simulate this, too.

After handling, the stack's state will be rolled back to the state from before the interrupt has occured by using the `iret` instruction. It is the last instruction of every interrupt/exception handler in general.

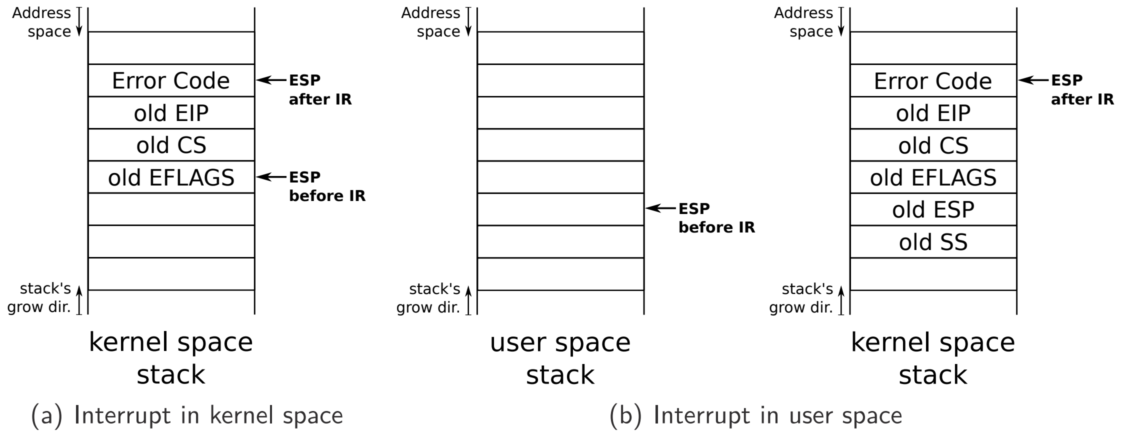


Figure 4.7.: Stack layouts before and after the occurence of an interrupt

## 4.6.4. The Guest's Possibilities to Block Interrupts

Usually on the x86 architecture interrupts are enabled and disabled with the `sti` ("set interrupt-bit") and `cli` ("clear interrupt-bit") instructions. These instructions manipulate the EFLAGS register, thus require a privilege level the guest does not have.

The lguest-guest's interrupts are generally delivered to the host first. Therefore the guest can either tell the host to selectively disable the delivery of specific interrupts or to completely disable the delivery of interrupts at all. For this purpose

[12]The privilege level during the interrupt is defined in the segment descriptor which is chosen by the interrupt descriptor

there is a field `irq_enabled` and a bitmap field `blocked_interrupts` within struct `lguest_data` to which both host and guest have access.

The hypervisor uses the function `interrupt_pending()`[13] to determine if there are interrupts to be delivered to the guest. Whenever an interrupt already has occured – and thus needs to be delivered to the guest – but its bit in `blocked_interrupts` is set, it will be ignored. The delivery will take place later when the guest has cleared the block-bit again. If the `irq_enabled` field's bit 8 is unset (which is also the interrupt-bit in the EFLAGS register) the function `interrupt_pending()` will not return any interrupt vector number at all. The lguest Linux kernel provides its own procedures at guest side to replace the architecture-specific set/unset calls for the interrupt-bit. Selective interrupt control is abstracted behind a simple configurable lguest IRQ controller "chip" interface.

Another last problem is keeping the `irq_enable` field (which simulates the guest's EFLAGS register) consistent after issuing the `iret` instruction. As already described before, the `iret` instruction is used to return from interrupts and pops the values stored on the stack back into the according registers. The last stack value to restore is the EFLAGS value which will not be written to the EFLAGS register, because the guest's privilege level is insufficient. However, the value needs to be written back to the `irq_enabled` field within struct `lguest_data`.

In the Linux kernel the usual `iret` call at the end of each interrupt handler is encapsulated into a procedure which is part of the collection of procedures the `struct paravirt_ops` points to. Lguest replaces it by the code seen in Listing 4.4. This procedure does nothing more than reading the EFLAGS value from the stack and writing it to `lguest_data.irq_enabled`.

```
ENTRY(lguest_iret)
    pushl   %eax
    movl    12(%esp), %eax
lguest_noirq_start:
    movl    %eax,%ss:lguest_data+LGUEST_DATA_irq_enabled
    popl    %eax
    iret
lguest_noirq_end:
```

Listing 4.4: lguest_iret code (linux/arch/x86/lguest/i386_head.S)

It is possible that an interrupt occurs between writing to the `irq_enable` field and issuing the `iret` instruction, but this would lead to faulty behaviour. Thus, this operation needs to be atomic. The labels `lguest_noirq_start` and `lguest_noirq_end` mark the area which shall be protected from interrupts. Before delivering interrupts, the hypervisor calls `try_deliver_interrupts()`, which checks if the guest's instruction pointer has a value between both label's values. If this condition matches then the hypervisor will not deliver the interrupt.

[13]defined in `metalsvm/drivers/lguest/interrupts_and_traps.c`

### 4.6.5. Optimization: Direct Interrupt Delivery

Letting interrupts and exceptions go to the host and then redelivering them to the guest takes the cost of a guest-host-guest context switch plus the cost of forming the stack with `set_guest_interrupt()`. It is desirable to have an optimization for that. Especially system calls issued by the guest's user space applications do not need to be delivered to the host - under this circumstance it is pure overhead to deliver such interrupts over this detour.

A guest is not aware about which interrupts or exceptions are sent to it directly and which are not. The hypervisor distinguishes to which vector number this optimization may be applied when it refreshes the guest's IDT entries. Differentiation between directly deliverable and not directly deliverable interrupts/exceptions takes both kind of interrupt descriptor and the vector number into consideration:

The first important attribute of interrupt descriptors which needs to be distinguished is its gate type. Interrupt descriptors describe either an *interrupt-gate* or a *trap-gate*. Bits 39 to 41 denote the descriptor's gate type: The value 0x6 denotes the interrupt-gate type while 0x7 denotes the trap-gate type. Only trap-gate descriptors are allowed to be delivered to the guest directly. For other descriptor types the processor disables interrupts before entering the handler function of the corresponding vector number - what the guest is not allowed to do. Hence, only trap-gate descriptors are optimizable. The second constraint is the right vector number: External interrupts are always supposed to be handled by the host, as the guest has no real hardware to deal with. Thus, Interrupts with vector numbers higher or equal to 32 are never delivered to the guest directly. The system call vector is the only exceptional case, because it has vector number 0x80 and shall be delivered directly to the guest always.

Within the allowed range of vector numbers 0 to 31 there is a subset of interrupts and exceptions, which is still to be delivered to the host always:

**Vector numer 6** This is used for the invalid opcode fault. The host needs to see this first, as it is possible that it must substitute the "`vmcall`" instruction by "`int 0x1f; nop`".

**Vector number 7** If a process within the guest tries to use the floating point unit it has to be initialized/restored for this purpose. The hypervisor has to do that with full privileges, because it involves writing into the CR0 register, what the guest is not allowed to.

**Vector number 13** General protection fault - The hypervisor has to emulate the I/O-behaviour of non-existing devices behind the I/O-ports at boot, so it needs to see general protection faults before the guest, too.

**Vector number 14** Page fault - The hypervisor needs to see page faults at first, because it maintains the shadow pagetables which are the ones the MMU really reads from.

**Vector number LGUEST_TRAP_ENTRY (0x1f)** This interrupt vector is used by hypercalls. It does not make sense to deliver it to the guest, of course.

```
1  void copy_traps(const struct lg_cpu *cpu, idt_entry_t *idt,
2          const unsigned long *def)
3  {
4      unsigned int i;
5
6      for (i=0; i < 256; ++i) {
7          const idt_entry_t *gidt = &cpu->arch.idt[i];
8
9          /* Do not alter indirect traps */
10         if (!direct_trap(i)) continue;
11
12         /* direct delivery only for trap-gates */
13         if (idt_type(*gidt) == 0xF)
14             idt[i] = *gidt;
15         else
16             default_idt_entry(&idt[i], i, def[i], gidt);
17     }
18
19 }
```

Listing 4.5: copy_traps code (metalsvm/drivers/lguest/interrupts_and_traps.c)

The hypervisor calls `copy_traps()` (as seen in Listing 4.5) before entering the guest whenever the IDT was changed before. The function `direct_trap()` does the differentiation between directly deliverable and not directly deliverable interrupts/exceptions. When checking for the gate type in line 13 the code does not use the trap-gate code 0x7 but 0xf instead, because the gate-size-bit is taken into consideration, too. In general, it is always set in 32 bit Linux.

### 4.6.6. Timer Supply

Not only for scheduling the guest needs configurable interrupts. In general the guest kernel decides at what point of time it wants to be interrupted or woken up. For this purpose it issues the time difference between this chosen point of time and current time in nanoseconds to the host, using the `LHCALL_SET_CLOCKEVENT` hypercall.

In the original lguest implementation in Linux the hypervisor takes advantage of the *high resolution timer* infrastructure. Such a high resolution timer is represented by a configurable structure (`struct hrtimer`[14]) containing primarily the timestamp at which it shall be triggered and a pointer to the handler function which then shall be called.

The hypervisor sets the point of time the event shall be triggered at to the time the guest desires. The hypervisor's handler function for this event does merely set the first bit in the hypervisor's guest-interrupt bitmap. This denotes that an interrupt with vector number 0 shall be delivered to the guest next time it is being run.

[14] defined in `linux/include/linux/hrtimer.h`

As MetalSVM did not provide any timer interface at the time when the hypervisor was implemented, one had to be created. The high resolution timer interface like seen in the Linux kernel is quite sophisticated, as it enables the user to choose events within a granularity of nanoseconds. Supplying timer interrupts with such a fine granularity is not necessary. It will merely make the guest run slower if the intervals are too sparse. The usual timestamp values the guest requests to receive an interrupt at mostly lay in orders of milliseconds or hundreds of microseconds in the future. Therefore MetalSVM is still able to supply interrupts in the right intervals in the majority of cases. Without modifying MetalSVM's own interrupt configuration it was possible to install a small and simple timer interface into the `timer_handler()`[15] procedure being periodically called by the APIC timer which is configured to 100 Hz.

The implementation can be seen in Listing B.5: Timer events are organized in a linked list where new items can be added by the hypervisor for example.

[15]defined in `metalsvm/arch/x86/kernel/timer.c`

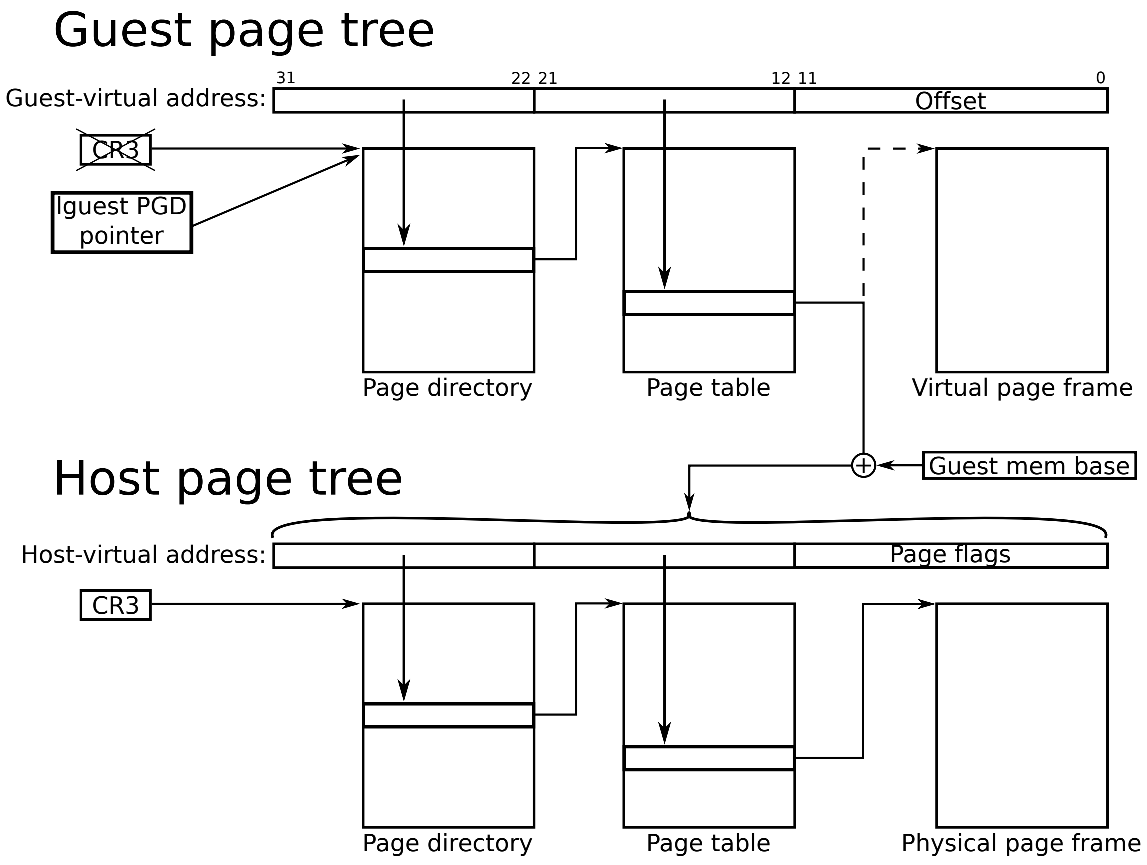


Figure 4.8.: Shadow paging mechanism explained

## 4.7. Shadow Paging

Paging is an efficient way to create virtual address spaces for programs and being able to isolate them from each other. Furthermore, additional features like read/write-protection and access-/dirty-bits enable great flexibility within the operating system's ways to handle memory pages.

Of course all of these features are desired in a virtual computer, but to use paging some privileged operations are needed which should not be exposed to the guest. Anyway, host and guest need different flags for the same pages, because the host may for example want to swap unused pages of a guest out of memory. This involves reading/setting flags the guest should not even know about.

This problem is solved by a technique called *shadow paging*: The guest kernel has its own page trees like usual, but it will only convert guest-virtual to guest-physical addresses which are merely host-virtual. The whole guest-virtual to host-physical-calculation works the following way (see Figure 4.8):

1. Some guest process tries to access memory which is mapped in the guest page tree but not yet recognized by the MMU (TLB-miss).

2. A page fault occurs, but the exception is not delivered to the guest kernel – a

context switch to the host kernel space is done.

3. The host kernel recognizes the nature of the exception and will find the guest's page tree behind a special pointer which is used by the guest instead of the CR3 register. It traverses through the guest page tree to find out the guest-physical address and the flags the guest stores for it.

4. The host-virtual address is just linear the guest-physical address plus the guest's memory base address.

5. The host kernel will look up the right physical address with its native page tree (it does not use the shadow page tree for itself) and fill it into the shadow page tree. At this time there is some flag checking, too, as the guest may show faulty behaviour.

6. After the next context switch to the guest the original guest-process continues just like after an ordinary page fault.

For setting page directory entries or page table entries in the shadow page table the guest kernel has to do a hypercall. It is especially useful that hypercalls are *batchable*. This way the guest can prepare a whole list of entries to be written and cast out one hypercall to map them all at once limiting the amount of overhead. The host will not keep an unlimited amount of shadow page trees for every guest process, because it has neither knowledge nor control over how many processes the guest has running. The number of shadow page trees is limited to a constant number. Currently, when a new shadow page tree is needed (being signalized by the guest using a hypercall), the host kernel will overwrite a randomly chosen existing shadow page tree.

Whenever the guest wants to flush the MMU this is an expensive operation, because it will cast a hypercall resulting in the host kernel emptying the whole shadow page tree. A lot of page faults will occur until the most page table entries are mapped again, but this way the shadow paging is kept quite simple and it is unlikely that the next guest process needs them all.

If the host cannot determine a guest-virtual to guest-physical translation within the guest's pagetables, it delivers the page fault to the guest. Of course, another host-guest context switch is needed until the MMU can finally translate accesses to this page.

The function `demand_page()` represents the heart of the portrayed mechanism and will be explained in more detail (see Listing 4.6):

**Lines 9-28** The host reads the guest's page directory and checks if it is present and the specific entry is valid. If the equivalent entry in the shadow page directory is not present, a new one will be allocated and the same flags applied.

**Linues 32-45** Now the host reads the page table entry within the guest's corresponding page table, checks if it is present and valid and if the corresponding guest process is allowed to access the page.

**Lines 47-50** The host has to set the flags within the guest's page tables the MMU would normally set. This includes access and dirty flags.

**Lines 55-63** With writing to the shadow page tree and guest page tree the address translation is finished.

**Lines 9, 35, 63** `lgread()`/`lgwrite()` are convenience-functions for reading/writing from/to guest-virtual addresses.

Whenever `demand_page()` returns 0, the page fault is sent back to the guest which then has to set an appropriate page table entry.

```c
int demand_page(struct lg_cpu *cpu, uint32_t vaddr, int errcode)
{
    uint32_t gpgd;
    uint32_t *spgd;
    uint32_t *gpte_ptr;
    uint32_t gpte;
    uint32_t *spte;

    gpgd = lgread(cpu, gpgd_addr(cpu, vaddr), uint32_t);
    if (!(gpgd & PG_PRESENT)) {
        return 0;
    }

    spgd = spgd_addr(cpu, cpu->cpu_pgd, vaddr);

    if (!((*spgd) & PG_PRESENT)) {
        /* There is no shadow entry yet, so we need to allocate a new one. */
        uint32_t ptepage = get_zeroed_pages(1);
        if (!ptepage) {
            kill_guest(cpu, "Allocating␣shadow-pte␣page␣-␣out␣of␣memory.");
            return 0;
        }
        check_gpgd(cpu, gpgd);

        /* Copy the flags from guest pgd into shadow pgd as we don't
         * need to change them. */
        *spgd = (uint32_t)(ptepage | (gpgd & 0xfff));
    }

    /* Step further to lower level in the guest page table.
     * Keep this address because it might be updated later. */
    gpte_ptr = (uint32_t*)gpte_addr(cpu, gpgd, vaddr);

    /* Actual pte value */
    gpte = lgread(cpu, (uint32_t)gpte_ptr, uint32_t);

    /* If there is no guest-virtual to guest-physical translation,
     * we cannot help. */
    if (!(gpte & PG_PRESENT)) return 0;

    if ((errcode & 2) && !(gpte & PG_RW)) return 0;

    if ((errcode & 4) && !(gpte & PG_USER)) return 0;

    check_gpte(cpu, gpte);

    gpte |= PG_ACCESSED;
    if (errcode & 2) gpte |= PG_DIRTY;

    spte = spte_addr(cpu, *spgd, vaddr);

    release_pte(*spte);

    /* If this was a write, we need the guest page to be writable */
    if (gpte & PG_DIRTY)
        *spte = gpte_to_spte(cpu, gpte, 1);
    else
        /* Unset the writable-bit to be able to return here later
         * on the next write attempt. */
        *spte = gpte_to_spte(cpu, gpte & ~PG_RW, 0);

    /* Write the guest pte entry back with updated flags. */
    lgwrite(cpu, (uint32_t)gpte_ptr, uint32_t, gpte);

    return 1;
}
```

Listing 4.6: Shadow paging code (metalsvm/drivers/lguest/pagetables.c)

## 4.8. Hypercalls

As the guest kernel is running on ring 1 and therefore is not privileged to perform many architecture-specific operations, it has to signalize which action it needs the host to perform. This is similar to system calls which are used whenever a user space process needs to call the kernel to do something. When a guest kernel calls the host kernel this is referred to as *Hypercall*.

Hypercalls exist for the following operations:

- Flushing a batch of asynchronous hypercalls
- Initialization of the lguest guest-structure host and guest share
- Shutdown/Restart the guest
- Virtual memory operations (Set PGD, PTE, new page table, Flush MMU)
- Load GDT, TLS or interrupt descriptor table entries
- Set a new kernel stack
- Set TS flag for the floating point unit
- Prompt the host to send pending guest-interrupts
- Send a notification to the host if the guest crashed or to signalize pending IO for virtual devices

### 4.8.1. How the Guest Casts a Hypercall

Casting a hypercall is very similar to casting a system call: The caller fills the parameters into the *eax, ebx, ecx, edx, esi* registers and issues an interrupt. The first parameter in the *eax* register always identifies the kind of hypercall (one from the list above) – the other registers are free to use for the semantics of the specific hypercall.

Lguest uses interrupt number 0x1f for hypercalls. KVM-guests use hypercalls with a very similar structure, but instead of issuing an interrupt the CPU instruction `vmcall` (available on CPUs with with the Intel VT extension) is used. The lguest developers decided to do it the same way to have more code in common with KVM. Thus, every lguest-guest initially casts hypercalls with the `vmcall` instruction. The hypervisor will then overwrite the instruction with a software interrupt instruction. This way lguest is not bound to CPUs which provide the Intel VT extension.

Another important difference between system calls and hypercalls is that user space processes are not allowed to cast hypercalls, of course.

## 4.8.2. Processing Hypercalls in the Hypervisor

After issuing a hypercall, the guest execution is stopped immediately and the context is switched back to the hypervisor. Within the context switch back to the host the parameter values in the registers are saved onto the guest state saving stack. Later in the hypervisor those register values can be easily mapped to a structure `hcall_args` which enables comfortable access. Processing the hypercall is done in the main virtualization loop before re-entering the guest as seen in Figure 4.1.

The guest can cast hypercalls by either issuing an "`int 0x1f`" or "`vmcall`" instruction. Unfortunately a `vmcall` causes a general protection fault on CPUs which do not provide the Intel VT extension. This is significantly slower then just casting a software interrupt. Because of this, the host kernel analyzes if a general protection fault was caused by a `vmcall`. In this case, the host kernel overwrites it with "`int 0x1f; nop`", because it is a 3 byte instruction.

Processing the hypercall itself is done within the procedure `do_hcall()`[16]. Again, this code looks very similar to how system call handling procedures are implemented in general. It embodies a large switch-case-statement to dispatch the kind of hypercall which is just delivered as an enumeration value written to the *eax* register by the guest.

## 4.8.3. Asynchronous Hypercalls as an Optimization

Because some kinds of hypercall (e.g. virtual memory operations) have to be used very often in one line they can be used as both synchronous or asynchronous calls to reduce the overhead of often repeated host-guest-switching.

Synchronous hypercalls are just done by filling given arguments into registers and casting an interrupt. The parameters of asynchronous hypercalls in contrast are stored in a ring buffer with elements just as broad as `struct hcall_args`. Pending asynchronous hypercalls in this ring buffer will be processed by the host before every context switch within the `do_async_hcalls()`[16] procedure. The implementation of this ring buffer is very primitive: The hypervisor keeps a variable containing the index of the last processed asynchronous hypercall. Just processed Hypercalls are marked as empty by writing 0xff into a status array at the same index like the processed hypercall has in the ring buffer.

For each asynchronous hypercall the hypervisor copies the parameters structure out of the ring buffer to where `do_hcall()` expects its parameters and calls it then. After having lined up a batch of asynchronous hypercalls the guest will just issue the `LHCALL_FLUSH_ASYNC` hypercall. This hypercall does nothing more than just stopping guest execution, as the host will process all pending hypercalls before entering the guest again.

[16]defined in `metalsvm/drivers/lguest/hypercalls.c`

## 4.9. VirtIO Implementation

At this time the Linux Kernel supports several virtualization systems, both as guest and host. Next to Xen, KVM and lguest there are User Mode Linux, VMWare's *VMI* and different solutions by IBM. Each of them except KVM and lguest has its own implementation of virtual devices for the guest while KVM and lguest use the *VirtIO* driver interface.

The VirtIO interface was initially designed for paravirtualization but it can also be ported to full-virtualized guests as VirtIO does also provide a standard *virtio-over-PCI* interface. VirtIO's ultimate goal is becoming *the* standard framework for I/O virtualization in hypervisors. Besides high efficiency by design it does not try to emulate complex hardware – it merely lets host and guest exchange I/O data over buffers with a low amount of overhead. The implementation of device I/O takes part in a frontend and backend part of the driver which is implemented in the guest and respectively the host who are communicating using VirtIO. Every implementation of the VirtIO interface consists of a *virtqueue* transport mechanism implementation and the *device abstraction API*.

The following subsections explain the general principles and data structures of the VirtIO API, followed by some implementation details in MetalSVM.

### 4.9.1. The VirtIO Device Abstraction API

Any device is represented by an instance of `struct virtio_device` which has another member `struct virtqueue_ops` containing several pointers to functions as an abstraction layer.

Before use the virtio driver registers itself. Recognition of device type and features is achieved with 32 bit fields for device type and vendor followed by the device configuration. The configuration takes place in the implementation of the functions `struct virtio_config_ops` points to.

The group of configuration operations consists of methods to access the following four parts of functionality:

**feature-bits** The host proposes features to the guest by setting specific feature-bits. If the guest wants to use a feature it acknowledges this back to the host.

**configuration space** A structure read- and writable by both host and guest to share configuration values of the device. (e. g. the MAC address of a network interface)

**status-bits** An 8 bit variable to enable the guest signalize the device status to the host

**reset operation** This enables to reset the device and re-initialize all its configuration and status-bits.

**virtqueue** A function returning a populated structure containing I/O-queues called `find_vq()` which is explained in the following.

After successful configuration the device I/O will employ the virtqueues it obtained from the `find_vq()` `config_ops` function. Devices can have a different number of virtqueues, e. g. separated ones for input and output. Access to the virtqueue is provided by another abstraction layer represented by `struct virtqueue_ops` (see Listing 4.7).

```c
struct virtqueue_ops {
    int (*add_buf)(struct virtqueue *vq,
                   struct scatterlist sg[],
                   unsigned int out_num,
                   unsigned int in_num,
                   void *data);
    void (*kick)(struct virtqueue *vq);
    void *(*get_buf)(struct virtqueue *vq,
                     unsigned int *len);
    void (*disable_cb)(struct virtqueue *vq);
    bool (*enable_cb)(struct virtqueue *vq);
};
```

Listing 4.7: The virtqueue access function abstraction

Typically buffers containing specific I/O data are added to the virtqueue with the `add_buf()` function and then the other side (e. g. host, but other guests are possible, too) is notified by the `kick()` procedure initiating a context switch from guest to host. Multiple buffers can be batched this way. The other side calls `get_buf()` to consume this data.

The `enable_cb()` and `disable_cb()` calls are for enabling/disabling interrupts which occur on the arrival of new data. This does not guarantee that interrupts are disabled – these calls are merely supposed to signalize that interrupts are unwanted to optimize the device's throughput.

## 4.9.2. virtio_ring: Linux' Virtqueue Implementation

Linux implements the virtqueues which are needed by the VirtIO API using *ring-buffers* which are in broad use within many highspeed I/O implementations.

This specific virtio_ring implementation involves use of three data structures which are also illustrated in Figure 4.9:

**descriptor table** This structure contains all length and (guest-physical) address pairs of used buffers and also provides a field for flags and a pointer for chaining elements together. The flags are used to signalize if the `next` pointer is valid and whether this buffer is read-only or write-only.

***available* ring** The array inside this structure represents a ring of descriptor table indices of buffers which are ready to consume while the index field is free-running and marks the beginning. The flags field is used to suppress interrupts.

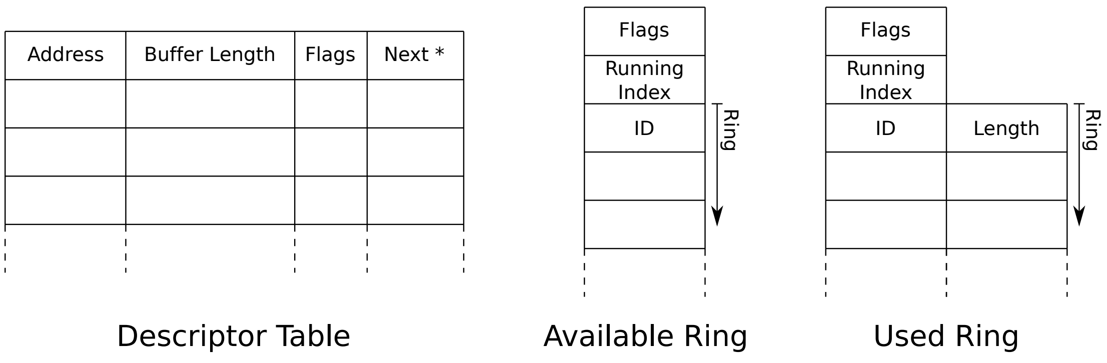


Figure 4.9.: VirtIO ring structures

***used* ring** Being similar to the *available* ring, it is used by the host to mark which descriptor chains have been consumed already.

C structure definitions of this data structures can be found in Listing B.6.

## 4.9.3. VirtIO in MetalSVM

Since MetalSVM just implements the hypervisor and no special devices were added to lguest, the only part to be implemented of VirtIO was the driver's backend. In the original Linux implementation of lguest, the hypervisor-part of the kernel also contains VirtIO-related code. Linux-lguest optimizes device event management using particular *event file descriptors* which do not exist in MetalSVM. Thus, the backend implementation of the VirtIO drivers is merely found within the user space part of MetalSVM's lguest hypervisor implementation – the lguest_loader application.

When Linux boots as an lguest-guest it assumes the existence of a VirtIO console device. Therefore the lguest_loader application does supply the console device to match this as the minimal device configuration. The application keeps a linked list of VirtIO devices with a pointer named `devices`. During the initialization before booting the guest the procedure `setup_console()` is called. It adds a new device called `console` to the device list. The corresponding enumeration value in the `type` field within the device descriptor structure finally characterizes this device as a console device. Providing both input and output functionality, the console device uses two separate virtqueues which are added to the device structure.

Early at boot the guest kernel will recognize the device list which is mapped into its address space at a specific location and issue a notification hypercall to the hypervisor. Since notifications are directed to the user space process, it gets a guest memory space address from the `read()`-call it was blocking on while the hypervisor did the virtualization. Before the guest kernel recognizes and initializes its VirtIO devices, it also uses notification hypercalls to output early boot messages. In this case the user space process just reads a pointer value pointing to the output string within the guest's memory range. If the pointer value is below the initial guest memory limit[17] it points to an output string. Otherwise it points to a virtqueue which has new buffers to consume what then is done in the device-specific way. The console device will just print the buffers onto the screen and mark them as used.

Console input as well as any other input from outside to the guest is written into buffers of the virtqueue. Write operations to buffers can be batched before the user space loader writes an interrupt request to `/dev/lguest`. The hypervisor will then issue an interrupt with a device-specific vector number to the guest.

[17] which was pushed to higher addresses when adding new devices via `get_pages(n)`, see the lguest loader subsection

# 5. Evaluation

At current time, the implementation of the lguest hypervisor in the MetalSVM kernel is fully functional and works accurately on both the Intel SCC and usual x86-hardware.

Since the developers of the Linux kernel do good work in respect to the execution speed and overall system-efficiency, it is quite challenging to reach or even beat the original lguest hypervisor's performance. However, the MetalSVM kernel's small size and specialized target environment allow for certain optimizations. Booting on usual x86 hardware, MetalSVM is able to show up the Linux guest's shell input prompt (which marks the end of the boot process) within one second from the beginning of the host's boot process.

Several measurements have been taken to analyze the hypervisor's performance in comparison with its original implementation in Linux: The cost of a host-guest context switch (one way), page faults in the guest's user space and several system calls as well as a real-life application. The performance of the host-guest context switch is very important, because it occurs forth and back on nearly every interrupt[1], exception and hypercall. Fortunately, MetalSVM's implementation is faster than the implementation in Linux, although the switcher's code had to be adapted (instructions were added) to work under MetalSVM. Page faults are very expensive in both Linux' and MetalSVM's lguest implementation. Two host-guest context round trips are needed between the MMU's page fault exception occurence and the user space's final successful access to the corresponding mapped page. The lguest developers are also aware of this design-based circumstance.

## 5.1. Measurement Details and Interpretation

The same guest kernel and ramdisk were run under both hypervisors for being able to present a fair performance comparison between both implementations. The guest kernel was compiled with a minimalistic configuration from Linux *vanilla* sources of version 2.6.32. To provide a spartan system environment, the initial ramdisk in use did merely contain a small set of tools compiled from the *Busybox* project[2] and the applications which performed the actual measurements. The applications were compiled statically with the *diet libc*[3]. Both hypervisors can be loaded with this set of kernel and ramdisk without the need of further modification. Test platform was

[1]Especially system calls are optimized by not being delivered to the host first.

[2]`http://www.busybox.net`

[3]`http://www.fefe.de/dietlibc/`

a computer with an *Intel Celeron 550* CPU, which has a clock frequency of 2 GHz and 1024 kB L2 cache.

Table 5.1 shows the tick count of each mechanism/function mentioned before and the ratio between both implementation's costs. In every measurement the "`rdtsc`" processor instruction was used to record the tick count.

| Measured | Hypervisor | | Ratio $\frac{MSVM}{Linux}$ |
|---|---|---|---|
| | Linux | MetalSVM | |
| Host-guest context switch | 1 406 | 1 347 | 96 % |
| Page fault | 40 426 | 31 978 | 79 % |
| getpid() | 1 039 | 626 | 60 % |
| fork() | 446 224 | 301 831 | 68 % |
| vfork() | 163 421 | 117 536 | 72 % |
| pthread_create() | 3 678 968 | 40 022 838 | 1 088 % |
| jacobi solver | $156.07 \cdot 10^9$ | $98.95 \cdot 10^9$ | 63 % |
| jacobi solver (2 instances) | $316.68 \cdot 10^9$ | $198.96 \cdot 10^9$ | 63 % |

*Values measured in processor ticks*

Table 5.1.: Performance measurements taken in the guest system

### 5.1.1. Host-Guest Context Switch

This is the only measurement which has actually been taken in kernel space at host side: The guest kernel was modified to issue 1 000 empty hypercalls at early boot. On host side, the tick count between the first and last hypercall was taken and averaged. The half of the average tick count can then finally be considered as the cost for one context switch from host to guest or the other way around, because both procedures are very symmetric.

Although MetalSVM takes a lower tick count in switching the context than Linux, the difference is not of magnificient dimension. However, MetalSVM's higher performance in this aspect is explained with lower overhead in the hypervisor's main loop, where one cycle has to be taken. It does not need to respect the same aspects of Linux' complex structure like the original implementation has to.

### 5.1.2. Page Fault

This and the following measurements have been taken in the guest's user space. The cost of a page fault was measured by an application which allocates an array with the size of 10 240 pages (40 MB). If the array was not initialized before, striding with simple write instructions through it with a step size of page size (4 kB) provokes a page fault on every step. The tick count of a full array traversal, divided by the number of loop iteration steps, represents the tick cost of one page fault.

Linux provides a sophisticated mechanism for swapping guest pages out of the memory onto the hard disk. MetalSVM does not provide this feature, thus saves a significant amount of overhead.

### 5.1.3. System Calls

As the first system call for being measured `getpid()` was chosen. It involves a minimum of payload-code being executed at kernel space side, thus being able to approximate the cost of an empty system call. Both `fork()` and `vfork()` as well as `pthread_create()` system calls are especially interesting. Every system call is issued 1 000 times to average the tick count later. In terms of address space replication, `fork()` is the most stressing candidate for the hypervisor's shadow paging mechanism, because address space replication in the guest involves a lot of page allocation and release in the hypervisor. MetalSVM uses a pool of pages to accelerate this. The acceleration is observable at nearly any task switch, as this involves the hypervisor releasing and allocating a whole page tree in the majority of cases[4]. The performance of `vfork()` in Linux as a MetalSVM guest is still better, but not as significantly better as `fork()`, compared to the Linux hypervisor implementation. `vfork()` does no address space duplication which works faster under MetalSVM, therefore its performance gain is not as high as in `fork()`. `pthread_create()` is obviously slow when run under the MetalSVM hypervisor. Tests have shown, that the low resolution of the current timer implementation is slowing down the guest very much. The tick count of pthread creation is reducible with a finer granularity of the timer infrastructure. A timer implementation in MetalSVM allowing the guest for choosing interrupt intervals in a magnitude of tens of microseconds should improve the performance significantly.

### 5.1.4. The Jacobi Solver Application

The Jacobi algorithm is used to solve systems of linear equations. For this purpose the Jacobi application creates a random $128 \times 128$ matrix of double precision floating point fields. Determining the solution vector took about 425 000 iterations. The measurement was taken several times and the average tick count was calculated afterwards.

The floating point unit's registers are stored and restored by the hypervisor. If several processes use it for calculations, the guest has to cast hypervalls for this to the hypervisor between nearly every task switch. Two instances of the application were run to see if this has an impact on the performance. Since the guest kernel uses the hypercall mechanism's batching feature, unnecessary overhead is avoided successfully.

[4]The hypervisor keeps only a limited number of page trees to switch between.

# 6. Conclusion

The objective to implement a virtualization layer for the operating system kernel *MetalSVM* was achieved. Linux is relatively frugal to paravirtualize due to its `paravirt_ops` structure and *VirtIO* virtual driver implementation. Therefore, merely a hypervisor with a sufficient set of features had to be chosen and ported for this project.

MetalSVM is now equipped with a kernel extension enabling for paravirtualizing Linux kernels without any need to modify them. The presented implementation shows a better performance than the original implementation in Linux in many aspects.

Lguest was initially not designed for virtualizing kernels of versions differing from the host's, therefore the specification of an ABI was omitted for the sake of simplicity. This thesis shows that this constraint is not as strong as expected. Actually because of not being tied to such an ABI, this hypervisor implementation will be easier to adapt to match the overall project's constraints. The performance issues shown in the evaluation are rooted in the fact that the young operating system MetalSVM is different from Linux in many regards. Especially the absence of a high resolution timer infrastructure highlights a possible construction site for optimizations in the future.

A branch of MetalSVM's code base supporting shared virtual memory between multiple cores does already exist. This together with the current lguest implementation sets the foundation for future research and development to accomplish a virtual SMP architecture supplying cache coherence by software.

# A. Background Knowledge

**ABI** *Application Binary Interface*: Describes conventions like data types, sizes and alignment to define an interface

**bare metal** Software is said to be run on bare metal if there is no further abstraction layer between itself and the underlying hardware

**Cache Coherence** consistence of cached data originally stored in memory shared by multiple cores

**ELF** *Executable and Linkable Format*: The common file format of executables, object files and shared libraries under Linux and several UNIXes

**GRUB** *Grand Unified Bootloader*: Multiboot specification conform boot loader mainly used to boot UNIX-like systems (`http://www.gnu.org/software/grub/`)

**Kernel** Main component of a computer operating system managing the system's resources for applications

**Kernel Module** The Linux kernel provides the possibility to load code at runtime to extend its functionality

**Magic String** A special byte string containing specific values hopefully too improbable to occur accidentally – Often placed at special positions in files or other structures which need to fulfill a specific format for verification and/or recognition

**MMU** *Memory Management Unit*: Translates virtual addresses to physical addresses using the page tables the operating system provides and its *Translation Lookaside Buffer*

**NMI** *Non-Maskable Interrupt*: Some interrupts cannot be masked with the IF flag using the standard instruction "cli" for masking interrupts

**Page** On systems implementing the concept of virtual memory the whole memory range is partitioned into blocks (then called *pages*) of fixed size

**SMP** *Symmetric Multiprocessing*: A class of computer architectures where multiple cores share the same main memory space and are controlled by a single operating system instance

# B. Code Listings

```c
int run_guest(struct lg_cpu *cpu, unsigned long *user)
{
    /* cpu->lg->dead points to NULL or to an error string */
    while(!cpu->lg->dead) {
        unsigned int irq;
        int more;

        /* The trap handler lguest_arch_handle_trap(cpu) does
         * set cpu->hcall in case of a hypercall by the guest. */
        if (cpu->hcall) {
            do_hypercalls(cpu);
        }

        /* Check if there are device notifications to be delivered */
        if (cpu->pending_notify) {
            *user = cpu->pending_notify;
            return sizeof(cpu->pending_notify);
        }

        irq = interrupt_pending(cpu, &more);
        if (irq < LGUEST_IRQS) {
            try_deliver_interrupt(cpu, irq, more);
        }

        if (cpu->lg->dead) break;

        if (cpu->halted) {
            if (interrupt_pending(cpu, &more) < LGUEST_IRQS) {
                /* Do not reschedule if there
                 * are interrupts left. */
            }
            else {
                reschedule();
            }
            continue;
        }

        /* Run guest with IRQs disabled. */
        irq_disable();
        lguest_arch_run_guest(cpu);
        irq_enable();

        /* There has been some trap we handle now.
         * (page fault for example.) */
        lguest_arch_handle_trap(cpu);
    }

    return -ENOENT;
}
```

Listing B.1: Lguest's main loop (metalsvm/drivers/lguest/lguest_core.c)

```
/*
 * 32-bit kernel entrypoint; only used by the boot CPU.  On entry,
 * %esi points to the real-mode code as a 32-bit pointer.
 * CS and DS must be 4 GB flat segments, but we don't depend on
 * any particular GDT layout, because we load our own as soon as we
 * can.
 */
__HEAD
ENTRY(startup_32)

    /* ... [most code commented out here: BSS initialization] ... */

#ifdef CONFIG_PARAVIRT
    /* This is can only trip for a broken bootloader... */
    cmpw  $0x207, pa(boot_params + BP_version)
    jb default_entry

    /* Paravirt-compatible boot parameters.  Look to see what architecture
        we're booting under. */
    movl pa(boot_params + BP_hardware_subarch), %eax
    cmpl num_subarch_entries, %eax
    jae bad_subarch

    movl pa(subarch_entries)(,%eax,4), %eax
    subl __PAGE_OFFSET, %eax
    jmp *%eax

bad_subarch:
WEAK(lguest_entry)
WEAK(xen_entry)
    /* Unknown implementation; there's really
       nothing we can do at this point. */
    ud2a

    __INITDATA

subarch_entries:
    .long default_entry     /* normal x86/PC */
    .long lguest_entry      /* lguest hypervisor */
    .long xen_entry         /* Xen hypervisor */
    .long default_entry     /* Moorestown MID */
num_subarch_entries = (. - subarch_entries) / 4
.previous
#else
    jmp default_entry
#endif /* CONFIG_PARAVIRT */
```

Listing B.2: Linux' startup_32 routine (linux/arch/x86/kernel/head_32.S)

```
ENTRY(lguest_entry)
    /* Initialization hypercall to host */
    movl  $LHCALL_LGUEST_INIT, %eax
    movl  $lguest_data - __PAGE_OFFSET, %ebx
    int  $LGUEST_TRAP_ENTRY

    /* Set up the initial stack so we can run C code. */
    movl  $(init_thread_union+THREAD_SIZE),%esp

    /* Jumps are relative: we're running __PAGE_OFFSET too low. */
    jmp lguest_init+__PAGE_OFFSET
```

Listing B.3: Linux' lguest_entry routine (linux/arch/x86/lguest/i386_head.S)

```c
__init void lguest_init(void)
{
    pv_info.name = "lguest";
    pv_info.paravirt_enabled = 1;
    pv_info.kernel_rpl = 1; /* Running on privilege level 1, not 0 */
    pv_info.shared_kernel_pmd = 1;

    /* A high amount of paravirt_ops is set. */

    /* Interrupt-related operations */
    pv_irq_ops.save_fl = PV_CALLEE_SAVE(save_fl);
    pv_irq_ops.restore_fl = __PV_IS_CALLEE_SAVE(lg_restore_fl);
    pv_irq_ops.irq_disable = PV_CALLEE_SAVE(irq_disable);
    pv_irq_ops.irq_enable = __PV_IS_CALLEE_SAVE(lg_irq_enable);
    pv_irq_ops.safe_halt = lguest_safe_halt;

    /* Setup operations */
    pv_init_ops.patch = lguest_patch;

    /* Intercepts of various CPU instructions */
    pv_cpu_ops.load_gdt = lguest_load_gdt;
      /* ... */

    /* Pagetable management */
    pv_mmu_ops.write_cr3 = lguest_write_cr3;
      /* ... */

    /* etc. */

    setup_stack_canary_segment(0);

    switch_to_new_gdt(0);

    /* the guest got this maximum address in the first hypercall */
    reserve_top_address(lguest_data.reserve_mem);

    lockdep_init();

    /* Register the panic hypercall */
    atomic_notifier_chain_register(&panic_notifier_list, &paniced);

    /* disable probing for IDE devices */
    paravirt_disable_iospace();

    /* ... [ More CPU initialization ] ... */

    /* The first virtual device is the console which is set here */
    add_preferred_console("hvc", 0, NULL);

    /* Register a preliminary console output method */
    virtio_cons_early_init(early_put_chars);

    pm_power_off = lguest_power_off;
    machine_ops.restart = lguest_restart;

    /* Paravirt guest initialization is complete. */
    /* From now on the normal Linux boot code is called */
    i386_start_kernel();
}
```

Listing B.4: Linux' lguest_init routine (linux/arch/x86/lguest/boot.c)

```c
/* This structure definition is taken from MSVM/include/metalsvm/time.h */
typedef struct _lrtimer {
    void (*function)(struct _lrtimer*);
    uint64_t expires;
    uint64_t delta;
    int periodic;
    struct _lrtimer *next;
} lrtimer;

/* ... */

static void timer_handler(struct state *s)
{
    uint32_t i;

#if MAX_CORES > 1
    if (smp_id() == 0)
#endif
    {
        timer_ticks++;

#ifdef CONFIG_LGUEST
        lrtimer *tm = lrtimer_list.first;
        while (tm != NULL) {
            if (tm->expires != 0 /* enabled? */
                && tm->expires < timer_ticks /* Expired? */
                && tm->function != NULL /* Function not null? */) {
                tm->function(tm);
                /* Reset it if periodic or disable. */
                if (tm->periodic) lrtimer_setalarm(tm, tm->delta, 1);
                else tm->expires = 0;
            }
            tm = tm->next;
        }
#endif
    }
}
```

Listing B.5: modified timer_handler() code (metalsvm/arch/x86/kernel/timer.c)

```c
/* Virtio ring descriptors: 16 bytes. These can chain together via "next". */
struct vring_desc {
    uint64_t addr; /* Guest-physical */
    uint32_t len;
    uint16_t flags;
    uint16_t next;
};

struct vring_avail {
    uint16_t flags;
    uint16_t idx;
    uint16_t ring[];
};

struct vring_used_elem {
    uint32_t id;  /* Index of start of used descriptor chain. */
    uint32_t len; /* Total length of the descriptor chain
                          which was used (written to) */
};

struct vring_used {
    uint16_t flags;
    uint16_t idx;
    struct vring_used_elem ring[];
};

struct vring {
    unsigned int num;
    struct vring_desc *desc;
    struct vring_avail *avail;
    struct vring_used *used;
};
```

Listing B.6: VirtIO structure definitions (metalsvm/include/virtio/virtio_ring.h)

# Bibliography

[1] ScaleMP. http://www.scalemp.com.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[3] M. Chapman and G. Heiser. Implementing transparent shared memory on clusters using virtual machines. *Memory*, (April):383–386, 2005.

[4] J. Fisher-Ogden. Hardware support for efficient virtualization. 17:2008, 2006.

[5] R. P. Goldberg. *Architectural Principles for Virtual Computer Systems.* PhD thesis, Harvard University, 1973.

[6] G. Heiser. Many-core chips - a case for virtual shared memory. *MMCS*, pages 5–8, 2009.

[7] IBM Systems. *Virtualization Version 2 Release 1*, 2005.

[8] Intel Corporation. *Cluster OpenMP User's Guide Version 9.1*, 2006.

[9] Intel Corporation. Intel ® 64 and ia-32 architectures software developer's manual volume 3a : System programming guide, part 1. *System*, 3(253666):832, 2010.

[10] Intel Labs. *SCC External Architectire Specification (EAS) Revision 1.1*, 2010.

[11] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium Volume 1*, Ottawa, Ontario, Canada, June 2007.

[12] J. Nakajima and A. K. Mallick. *Hybrid-Virtualization - Enhanced Virtualization for Linux.* 2007.

[13] P. Reble, S. Lankes, C. Clauss, and T. Bemmerl. A Fast Inter-Kernel Communication and Synchronization layer for MetalSVM. In *Proceedings of the 3rd MARC Symposium, KIT Scientific Publishing*, Ettlingen, Germany, July 2011.

[14] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, pages 10–10, Berkeley, CA, USA, 2000. USENIX Association.

[15] R. Russell. lguest: Implementing the little linux hypervisor. In *Proceedings of the Linux Symposium Volume 2*, Ottawa, Ontario, Canada, June 2007.

[16] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42:95–103, July 2008.

[17] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[18] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36:195–209, December 2002.

[19] XenSource Inc. *Xen User's Manual for Xen v3.3*, 2008.