Bachelor thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Systems and Control

Jailhouse hypervisor

Maxim Baryshnikov

Supervisor: Ing. Michal Sojka, Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Systems and Control
May 2016

Acknowledgements

I would first like to thank my thesis advisor Ing. Michal Sojka, Ph.D., for his assistance and dedicated involvement in every step throughout the process. Without his great mentorship this paper would have never been accomplished. I would also like to thank my family and friends for their moral support and help and specially thank to my friend Andrey Alberstein who borrowed me his own PC for some experiments related to this work.

Declaration

I hereby declare that I have completed this thesis with the topic "Jailhouse hypervisor" independently and that I have included a full list of used references.

Prague, May __, 2016

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, __. května 2016

Abstract

This bachelor thesis is dedicated to Jailhouse, Linux-based partitioning hypervisor, which provides the software solution for asymmetric multiprocessing (AMP). This thesis describes the main concepts and operation principles of this hypervisor, contributes the implementation of the simple demo application (interacting with High Precision Event Timer) and shows how a small operating system (L4 Fiasco.OC) was ported inside the partition of this hypervisor. Another part of this work is an evaluation of shared memory hierarchy influence on real-time properties of software running under Jailhouse. Benchmarks were implemented and applied in Jailhouse partitions to investigate how significant that influence will be. Tests showed (for two CPUs interacted with each other) the approximately 220% slowdown of memory accesses bandwidth in the worst case. This is the result of partitions' competition about L3 cache which is shared among the CPU cores.

Keywords: Jailhosue, hypervisor, asymmetric multiprocesing, Fiasco.OC, shared memory hierarchy, benchmark, degree project

Supervisor: Ing. Michal Sojka, Ph.D.

Abstrakt

Tato bakalářská práce se povídá o Jailhouse hypervisoru, což je softwarový prostředek pro realizaci asymetrického multiprocesingu. V práci se popisuje základní koncepty a operační principy tohoto hypervisoru. Ukazuje se tady jak se dá naimplementovávat jednoduchou aplikaci, což má za úkol využivat High Precision Event Timer. Dal, do prostředí hypervisoru byl portovan malý operační systém (L4 Fiasco.OC), a proběhlo to úspěšně. Naposled, byl determinovan vliv sdílené paměťové hierarchie na běh programů (tj. jejích vlastnosti týkající se práci v reálném čase) v buňce hypervizoru. Pro tento účel byly navrhnuty a spouštěny benchmarky, a ty ukázali, že při využití dvou jáder z různými běžicí programy dochazelo se ke zpomalením testujemých procesů kvůli konkurenci za sdílené paměťové zdroje. A to až o 220% v nejhorším případě.

Klíčová slova: Jailhouse, hypervisor, asymetrický multiprocessing, Fiasco.OC, sdilená paměťová hierarche, benchmark, závěrečnná práce

Contents

Figures

Tables

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Control Engineering

BACHELOR PROJECT ASSIGNMENT

Student: **Maxim Baryshnikov**

Study programme: Cybernetics and Robotics
Specialisation: Systems and Control

Title of Bachelor Project: **Jailhouse hypervisor**

Guidelines:

1. Make yourself familiar with Jailhouse hypervisor and its safety-related use cases.2. Develop simple applications that will run inside Jailhouse cells (virtual machines) both on bare hardware and with a small operating system such as Erika, RTEMS or L4 Fiasco.OC.3. Use various benchmarks to evaluate the influence of shared memory hierarchy (caches, DRAM) on real-time properties of software running in different cells.4. Document your results.

Bibliography/Sources:

[1] Valentine Sinitsyn, "Understanding the Jailhouse hypervisor", https://lwn.net/Articles/578295/[2] Heechul Yun; Gang Yao; Pellizzoni, R.; Caccamo, M.; Lui Sha, "Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms," in Computers, IEEE Transactions on , vol.65, no.2, pp.562-576, Feb. 1 2016 doi: 10.1109/TC.2015.2425889[3] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna, A memory-centric approach to enable timing-predictability within embedded many-core accelerators

Bachelor Project Supervisor: Ing. Michal Sojka, Ph.D.

Valid until the summer semester 2016/2017

L.S.

prof. Ing. Michael Šebek, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, February 9, 2016

Chapter 1
Introduction

Nowadays multi-core systems are getting cheaper and more available for everyone. They became very attractive for usage in real-time systems because of high computational power. However, in most cases such systems are used for parallel execution of one task. An alternative way of application of multi-core systems is running independent tasks on each core. This solution may reduce the final price of hardware resources without affecting performance.

Such operating mode introduces several problems. The CPUs are not completely independent because they share caches, interconnect and memory bus. Code which is running on one core could interfere the code executing on other cores through shared hardware resources. Jailhouse, Linux-based partitioning hypervisor, which is described in this thesis, provides the software solution for asymmetric multiprocessing realization and attempts to solve mentioned issues.

The aim of this thesis is to study the Jailhouse hypervisor. Firstly, the concepts and operation principles will be described. I will investigate basic application which comes with Jailhouse as a standard demo and propose an additional one. Then, I will launch the small operating system L4 Fiasco.OC inside the hypervisor partition. Finally, I will run benchmarks using the ported Fiasco.OC to evaluate the influence of shared memory hierarchy on real-time properties of software running in different cells. The thesis will be finalized by discussing the achieved results.

Chapter 2
Jailhouse hypervisor

Jailhouse is Linux-based partitioning hypervisor which can run bare-bone applications or adopted operating systems in its partitions and provides substantial isolation between them. The project was started by Jan Kiszka, lead developer, as an internal research in Siemens, AG. Then, he decided to open sources in 2013. Jailhouse is still quite young (currently at version 0.5) and is under active development. It is available for ARM and x86 architectures. Project home located at `https://github.com/siemens/jailhouse`.

It is not yet another huge featured and general purpose virtualization solution such as XEN or KVM. Jailhouse is primarily focused on safety-related use-cases (industrial processes, aerospace, medicine, etc.) and supposed to be real-time capable right out of the box. So, its operation field is quite special.

This work relates to x86 version only. New features are being added very often, so e.g. cache partitioning support or command line passing options are not mentioned here.

The following sections describe basic concepts (2.1), operation related theory and practical notes about how Jailhouse might be configured and launched (2.2) and some demonstration examples (2.3).

2.1 Concepts

The main feature of this particular hypervisor is that, instead of sharing multi-core processor resources symmetrically between guests, the Jailhouse launches each guest with his resources set (cores, peripheral, memory). That concept called *asymmetric multiprocessing*. Best description of this property was given by Vitaliy Sinitsyn, one of active Jailhouse contributors:

“ ...Jailhouse enables asymmetric multiprocessing (AMP) on top of an existing Linux setup and splits the system into isolated partitions called "cells.” Each cell runs one guest and has a set of assigned resources (CPUs, memory regions, PCI devices) that it fully controls. The hypervisor’s job is to manage cells and maintain their isolation from each other.”[Val14a]

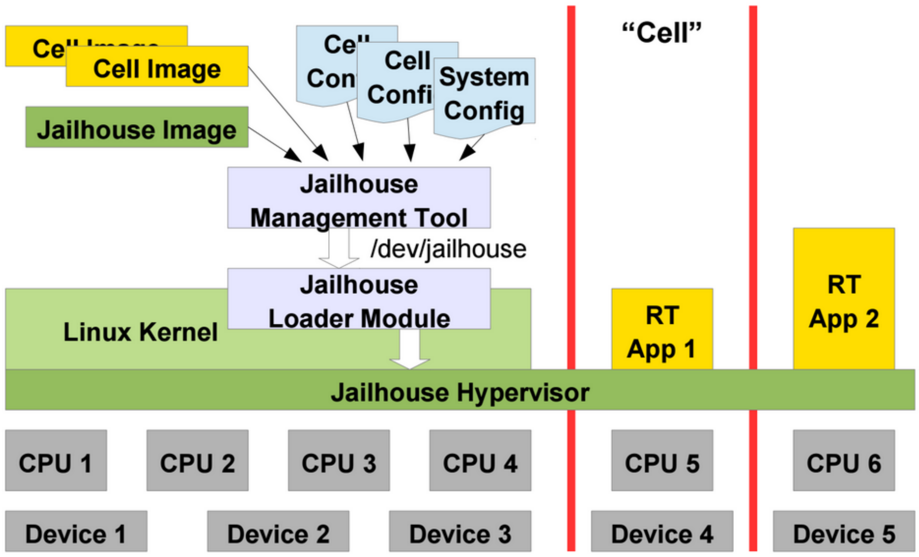

Figure 2.1: The Jailhouse architecture overview. Source: [Jan15]

However, who needs the hypervisor which has for a task only to isolate things? Well, suppose we have a multi-core system, and we want to use one core for a hard real-time task (e.g. program which controls some critical industrial process), other core for some user interaction (GUI, non-real-time application) and the rest CPUs could collect statistics from sensors. It is evident that GUI application must not influence the first core work. And this is the hypervisor job to prevent such interactions.

As was already mentioned, Jailhouse's target domain is safety-critical industrial applications. Such applications are often required to be certified by an independent authority according to numerous safety standards. The goal of the certification is to gain confidence that the system is reliable enough to perform its intended function safely. Safety standards classify safety functions into several levels (often called Safety Integrity Levels) according to the needed degree of reliability, giving more strict requirements on systems with higher criticality. It is a key to keeping the complexity of higher-criticality systems low to make their verification and certification possible. Complex systems are hard (i.e. expensive) or impossible to certify. This is the reason why Jailhouse is the very minimalistic hypervisor, containing only the functionality which is needed for proper isolation of guest systems.

The Jailhouse was developed to fit these mentioned requirements. It uses virtualization techniques to create strong isolation between guests, and this is its *only* task. It does not emulate any devices for them for example. Jailhouse does *one-to-one resource assignment* to separate resources between partitions.[Jan15] That means: if one partition has access to some I/O port, PCI device or any other resource when other partition have not. All these makes the performance of the Jailhouse very close as if tasks run on bare hardware.

Strong isolation is the reason where the roots of terminology came from. Jailhouse developers usually use the "*cell*" term to describe the partition where an "*inmate*" - guest executable binary is located. A cell with Linux that bootstraps Jailhouse and from where other cells could be managed called "root cell". I also will use that terminology in this thesis.

The diagram in Fig. 2.1 might shed some light on the Jailhouse architecture. Surely, none of the cells have got access to a device that does not belong to them, because hypervisor prevents it. Thus, real-time applications are not influenced by whatever is going on in other partitions. However, hypervisor is managed from Linux user-space by accessing to the jailhouse device driver which is able only to issue a hypercalls to the hypervisor. It is important to notice that Jailhouse is not a part of a kernel (as KVM, or VirtualBox), it runs at the lowest level. Kernel module is used there only to deliver the hypervisor binary to the reserved memory in kernel address. This process described in section 2.2.3.

2.2 Operation

This section describes the basic Jailhouse functionality, explains internal processes and also provides requirements (2.2.1) and steps which should be done to enable and start the inmate in a cell(2.2.2, 2.2.3, 2.2.4).

2.2.1 Hardware requirements

Jailhouse relies on hardware virtualization features of CPU to be fast and to simplify its code. For running on x86 architecture it requires (according to [`README.md`] from [Jan]):

- Intel system:
    - support for 64-bit and VMX
        - EPT (extended page tables)
        - unrestricted guest mode
        - preemption timer
    - Intel IOMMU (VT-d) with interrupt remapping support (except when running inside QEMU)
- AMD system:
    - support for 64-bit and SVM (AMD-V)
    - NPT (Nested page tables) - required
    - Decode Assists - recommended

   - AMD IOMMU (AMD-Vi) is unsupported now but will be required in future

- at least 2 logical CPUs

It also could be launched under QEMU with KVM mode. However, even in this case host parameters must respond the requirements mentioned above.

2.2.2 Cell configuration

Each cell (either root or non-root) must be statically configured before it launches. This configuration determines which hardware resources can the cell access. Jailhouse uses *.c files where parameters have to be assigned as fields of special C structures, which are defined into `hypervisor/include/jailhouse/cell-config.h`. For the non-root cell, this setup looks like in Listing 2.1. Comments were added to key fields to show how it should be used.

Listing 2.1: The part of a Non-root cell configuration. Fields commented.

```
#include <linux/types.h>
#include <jailhouse/cell-config.h>
#define ARRAY_SIZE(a) sizeof(a) / sizeof(a[0])

struct {
 /*The size of arrays there must correspond with the amount of
 fields of each type.*/
 struct jailhouse_cell_desc cell;
 __u64 cpus[1];
 struct jailhouse_irqchip irqchips[1];
 __u8 pio_bitmap[0x2000];
 struct jailhouse_pci_device pci_devices[1];
 struct jailhouse_pci_capability pci_caps[1];
 } __ __attribute__((packed)) config = {
.cell = {
 .signature = JAILHOUSE_CELL_DESC_SIGNATURE,
 /*Name of the cell*/
 .name = "NAME-of-non-root-cell",
 .cpu_set_size = sizeof(config.cpus),
 .num_memory_regions = ARRAY_SIZE(config.mem_regions),
 .num_irqchips = 1,
 .pio_bitmap_size = ARRAY_SIZE(config.pio_bitmap),
 .num_pci_devices = 1,
},
/*CPUs which are assigned to a cell.
<n> bit set = core <n> will be used.*/
.cpus = {
 00010010b, /* e.g., here are assigned 1st and 5th CPUs*/
},
/*Here is setup which mem regions this cell
could have access and with which rights (flags).*/
.mem_regions = {
 {
     .phys_start = 0x3f000000,
     .virt_start = 0,
     .size = 0x00100000,
     .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
      JAILHOUSE_MEM_EXECUTE | JAILHOUSE_MEM_LOADABLE,
    }
   },
   /*Several irq chips could be assigned.*/
   .irqchips = {
   {
    .address = 0xfec00000,
    .id = 0xff01,
    /*Allowed irqs. <n>-bit set = allow <n> irq.*/
    .pin_bitmap = 0xffffff,
   },
   },
   /*Those bitmasks allow a cell to access some I/O ports. */
   /*bit set = access denied, bit cleared = access allowed.*/
   .pio_bitmap = {
     [     0/8 ...  0x3f7/8] = -1,
     [ 0x3f8/8 ...  0x3ff/8] = 0, /* serial1 */
     [ 0x400/8 ... 0xe00f/8] = -1,
     [0xe010/8 ... 0xe017/8] = 0, /* OXPCIe952 serial1 */
     [0xe018/8 ... 0xffff/8] = -1,
   },
   /*PCI devices assignment.*/
   .pci_devices = {
    {
     .type = JAILHOUSE_PCI_TYPE_DEVICE,
     .domain = 0x0000,
     /*Bus, Device, and Function address*/
     .bdf = 0x00d8,
    /*It is possible to add capabilities to a device.*/
    /*Index of the first entry for this device in the array
    below.*/
     .caps_start = 0,
     .num_caps = 2,
     .num_msi_vectors = 1,
     .msi_64bits = 1,
    },
   }
   /*list of capabilities */
   .pci_caps = {
    {
    .id = 0x5,
    .start = 0x60,
    .len = 14,
    .flags = JAILHOUSE_PCICAPS_WRITE,
    },
   }
  };
```

For the root-cell the structure must have the (`struct jailhouse_system`) in the header instead of `struct jailhouse_cell_desc`, however, the rest structures are the same as in the Listing 2.1.

Listing 2.2: The header of a root cell configuration. Fields commented.

```
    .header = {
     .signature = JAILHOUSE_SYSTEM_SIGNATURE,
     .hypervisor_memory = {
     /*This is the memory area,
       where the hypervisor binary must be placed.
    This memory must be reserved when Linux starts.*/
       .phys_start = 0x3b000000,
       .size = 0x600000,
     },
     .platform_info.x86 = {
       .mmconfig_base = 0xb0000000,
       .mmconfig_end_bus = 0xff,
       .pm_timer_address = 0x608,
       /*IOMMU could be defined here.*/
       .iommu_units = {
        {
        .base = 0xfed90000,
        .size = 0x1000,
        },
   },
  },
     .interrupt_limit = 256,
 .root_cell = {
    /*struct jailhouse_cell_desc follows from there.*/
```

Unfortunately, the only option to configure the non-root cell is to do it manually. Jailhouse does not provide any interactive tool for it. However, for the root cell configuration file such tool exists. It could be generated in two steps:

1. Collect information about target system by executing:

   ```
   jailhouse config collect <name-of-arch.tar>
   ```

   This python script located into *tools* directory. It copies all from `/proc` and `/sys` and compresses it.

2. Process data on another system. This step requires `python-mako` library installed.

   ```
   jailhouse config create -r <name-of-arch.tar> <name-of-conf.c>
   ```

Alternately, if `python-mako` is available on the target system, the first step could be skipped. Just use the second command without `-r` parameter.
After creating, the configuration is compiled in raw binary, and Jailhouse operates with it when the cell is created.

2.2.3 Enabling Jailhouse

To be ready for a battle, Jailhouse needs this steps to be performed:

1. Install Linux on target system (version >=3.18)
2. Compile and install Jailhouse user-space tools on the target system (see README.md [Jan]).
3. Provide the reserved memory region by appending `memmap=` option to Linux kernel on boot. The value here must be the same as values of `.phys_start` and `.size` in the header of root cell configuration.2.2.2
4. Load the `jailhouse.ko` module into the kernel.
   This enables `/dev/jailhouse` in the system which Jailhouse user-space tools can operate with.
5. start the hypervisor by executing this:

   ```
   jailhouse enable <path/to/cell/conf.cell>
   ```

The 4th step starts the following operations (well described in [Val14a]). `jailhouse` user-space program sends the JAILHOUSE_ENABLE request to the `/dev/jailhouse`, which signals driver to call `jailhouse_cmd_enable()` `(driver/main.c)`. In this function, the driver does some validation first. It checks out CPU flags to determine which virtualization technology this CPU uses (Intel's VMX or AMD's SVM), then it does basic validation of a configuration binary (signature in the header). After that, it calls `request_firmware()` function which searches for `jailhouse-intel.bin` or `jailhouse-amd.bin` in `/lib/firmware` folder[ME]. At the next stage, driver remaps memory region reserved in step 2 to the kernel address space memory (using `ioremap_page_range(...)`), so hypervisor could be accessed from the user-space. Driver copies that binary at the start of this memory area and cell configuration right after it. Then, it calls `jailhouse_cell_create()` function, whose operation is described in section 2.2.4 in details.

The final stage of Jailhouse enabling is the CPUs initialization. This process has an excellent explanation under the same-named section in [Val14a]. Briefly, hypervisor starts it by calling `entry_hypervisor()` function for every CPU (which leads to `arch_entry` in `hypervisor/x86/entry.S`). Jailhouse needs to become an interface between cells (root cell with Linux at this early boot time) and CPU cores, so it saves system's state and then sets up its environment when the CPU0 initializes. It includes: setting up paging for the hypervisor and APIC, creating the Interrupt Description Table (IDT), creating root cell and remapping Linux memory regions and devices, and configuring of Virtual Machine Extensions (VME). It also sets up UART communication to write debug information in, so the following info (Listing 2.3) could be seen on the `ttyS0` (by default). The continuing of this process is the same for all CPUs: renew IDT and Global Descriptor Table (GDT), reset CR3 register (page table pointer) and setup Virtual Machine Control Structure (VMCS). Finally, hypervisor sends a `VMLAUNCH` instruction, and this returns the control to Linux, but since this point in time, Linux no longer runs "on bare metal" but in the "cell" (virtual machine) under Jailhouse.

Listing 2.3: Log from the "QEMU-VM" root cell initialization.

```
 Initializing Jailhouse hypervisor  on CPU 0
 Code location: 0xfffffffff0000030
 Using xAPIC
 Page pool usage after early setup: mem 43/1505, remap
65/131072
 Initializing processors:
 CPU 0... (APIC ID 0) OK
 CPU 1... (APIC ID 1) OK
 CPU 3... (APIC ID 3) OK
 CPU 2... (APIC ID 2) OK
 WARNING: AMD IOMMU support is not implemented yet
 Adding PCI device 00:01.0 to cell "QEMU-VM"
 Adding PCI device 00:02.0 to cell "QEMU-VM"
 Adding PCI device 00:1b.0 to cell "QEMU-VM"
 Adding PCI device 00:1d.0 to cell "QEMU-VM"
 Adding PCI device 00:1d.1 to cell "QEMU-VM"
 Adding PCI device 00:1d.2 to cell "QEMU-VM"
 Adding PCI device 00:1d.7 to cell "QEMU-VM"
 Adding PCI device 00:1f.0 to cell "QEMU-VM"
 Adding PCI device 00:1f.2 to cell "QEMU-VM"
 Adding PCI device 00:1f.3 to cell "QEMU-VM"
 Adding virtual PCI device 00:0f.0 to cell "QEMU-VM"
 Page pool usage after late setup: mem 180/1505, remap
65602/131072
 Activating hypervisor
```

2.2.4 Cell initialization and start process

At the moment, when user executes Jailhouse user-space application like this:

`jailhouse cell create <path/to/conf.cell>`

It reads configuration binary into memory and sends the `JAILHOUSE_CELL_CREATE` command to the driver with an address of the loaded binary which is attached. It invokes `jailhouse_cmd_cell_create()` (`driver/control.c`) which copies the mentioned binary from the user-space memory to a kernel space and performs some checks for loaded cell description (signature, size). Then it makes an image for a guest according to the taken configuration (filling up the special `struct cell` defined in `driver/cell.h` with pointers at mapped memory for guest's regions and PCI devices). After that, the driver leaves an information about the new cell in sysfs and plugs requested CPUs out of the Linux (root cell). It also "unplugs" PCI devices from Linux at this time. Jailhouse emulates PCI dummy driver

(see `jailhouse_pci_claim_release()`) to cut it out of Linux as far as real unplug could not be performed. The reason is explained in source code comments (`driver/pci.c`): “Linux will reprogram the BARs and locate resources where we do not expect them.”
Next stage starts when the driver issues the `JAILHOUSE_HC_CELL_CREATE` hypercall. Hypervisor when catching it calls `cell_create()` in `hypervisor/control.c` where it firstly gives a command for all new cell’s processors to suspend except the current one (which executes this code right now). It prevents race conditions between them[Val14b]. The following step to allocate pages for memory regions of the cell.

After that, the **cell initialization** process starts. The `cell_init()` function fills the `cpu_set` field in `cell` struct with values of I/O ports’ bitmaps and calls a routine to save (already reallocated to guest) locations and handlers of the memory-mapped devices such as PCI, IOAPIC, and IOMMU. Then, after checking that all CPUs are not owned by somebody else, the operation moves to `arch_cell_create()` (`hypervisor/arch/x86/control.c`) where begins the part which developers called *Linux “shrinking”* [Val14b]. The point here is to follow the *one-to-one assignment concept*: if the root cell has something that initializing cell wants, then access for Linux cell will be denied, and the new cell gets it. The problem appears if, for example, Linux continues to use serial port after it was assigned to another cell. Linux CPU will be parked at the very first access in that case. After resources like I/O ports, IOAPICs, IOMMUs, PCIs were reassigned, the **communication region** is configured. This is “.. a per-cell shared memory area that both the hypervisor and the particular cell can read from and write to. It is an optional communication mechanism..” [`Documentation/hypervisor-interfaces.txt`]. It also contains information about PM timer address, the number of the CPU assigned, information about the current cell state (could be Running or Running/Locked for example).

Finally, the cell will be committed to the list of cells, cell state in communication region will be set at `JAILHOUSE_CELL_SHUT_DOWN` and for every cell’s CPU hypervisor will issue `arch_cpu_resume()`.

To execute some inmate in the new cell it is needed to move it to the cell’s memory region. This is done by:

```
jailhouse cell load <name-of-cell> <inmate.bin> -a <offset-in-guest>
```

All inmates are treated as raw binaries. The size of this binary must be less or equal to the guest memory region where it will be loaded. Mechanism of transfer the file into cell’s memory is similar to previous cases. The driver sends `JAILHOUSE_HC_CELL_SET_LOADABLE` to the hypervisor, and it remaps guest regions marked as loadable to the root cell address space. The message "`Cell <name-of-cell> can be loaded.`" should be seen at this stage on the debugging serial port. After that, the driver stores binary at the given address.

And finally, to start it, user should invoke:

```
jailhouse cell start <name-of-cell>
```

It causes the hypercall `JAILHOUSE_HC_CELL_START` which, from its side, causes hypervisor's `cell_start()` perform the unmapping all loadable regions from the root cell back to the guest. Cell's state becomes `JAILHOUSE_CELL_RUNNING` and on each CPU of the cell is invoked arch_cpu_reset(). This sends fake Startup Inter-Processor Interrupt (SIPI) to each CPU in the cell. At the next #VMEXIT, guest instruction pointer will be set at `0xffff0`, and the inmate starts.

2.3 Inmate demos

Jailhouse provides a small framework which makes the development of the simple OS-less applications easier. It is not mandatory to use it, but it gives a good example of how things could be done inside a cell. That tiny library of useful functions is a C-header file `inmate.h` which defines routines for memory allocation and remapping, APIC and IOAPIC initialization, interrupt handlers setup, several interactions with PCI devices and even some basic SMP operations (`smp_wait_for_all_cpus()`, `smp_start_cpu()`).

Listing 2.4: Startup routine for every inmate demo. The part of `header-32.S`.

```
.code16
.section ".boot", "ax"

.globl __reset_entry
__reset_entry:
ljmp $0xf000,$start16


.section ".startup", "ax"

start16:
lgdtl %cs:gdt_ptr

mov %cr0,%eax
or $X86_CR0_PE,%al
mov %eax,%cr0

ljmpl $INMATE_CS32,$start32 + FSEGMENT_BASE


.code32
start32:
mov %cr4,%eax
or $X86_CR4_PSE,%eax
mov %eax,%cr4
mov $loader_pdpt + FSEGMENT_BASE,%eax
mov %eax,%cr3

mov $(X86_CR0_PG | X86_CR0_WP | X86_CR0_PE),%eax
mov %eax,%cr0

(...)
```

The example of the startup code for inmates is represented in Listing 2.4 (see `header.S` for applications running in 64-bit mode, or `header-32.S` for 32-bit mode). As it has been mentioned earlier, Jailhouse waits for the inmate entry point at address `0xffff0`. So the small trick is needed there to jump to 16-bit code section (for the GDT and protected mode flag setup). Inmate demo binaries are loaded into guest memory with the offset `0xf0000` (see loading in section 2.2.4). Those binaries are linked with consideration to this offset.

Here how it is done. The linker script (`inmate.lds`) which is shown in Listing 2.5 ensures the following. 16-bit `.startup` section, which contains those mentioned setup instructions, is bonded at the very beginning of the binary. Section `.boot` is pinned at `0xfff0`, so addition with the offset gives right entry address. Sections `.text`, `.data` and `.rodata` have their Virtual Memory Addresses (VMA) with the load offset included. However, their Load Memory Addresses (LMA) do not have it. As a result, when the binary will be placed into the memory of a cell everything will be where it is supposed to be.

(VMA means "the address the section will have when the output file is run"[lin] and LMA means "the address at which the section will be loaded"[lin].)

Listing 2.5: The linker script for inmate demos.

```
SECTIONS
{
 /* 16-bit sections */
 . = 0;
 .startup : { *(.startup) }

 . = 0xfff0;
 .boot  : {
  *(.boot)
  . = ALIGN(16);
 }

 /* 32/64-bit sections */
 . = 0xe0000;
 stack_top = .;
 bss_start = .;
 .bss  : {
  *(.bss)
```

```
   . = ALIGN(8);
  }
  bss_dwords = SIZEOF(.bss) / 4;
  bss_qwords = SIZEOF(.bss) / 8;

  . = 0xf0000 + SIZEOF(.startup);
  .text  : AT (ADDR(.text) & 0xffff) {
   *(.text)
  }

  . = ALIGN(16);
  .rodata  : AT (ADDR(.rodata) & 0xffff) {
   *(.rodata)
  }

  . = ALIGN(16);
  .data  : AT (ADDR(.data) & 0xffff) {
   *(.data)
  }

  /DISCARD/ : {
   *(.eh_frame*)
  }
 }

ENTRY(__reset_entry)
```

Thus, Section `.boot`, which has the mentioned entry point placed at `0xffff0`, has only one instruction: `ljmp $0xf000,$start16`. It causes instruction pointer to move on physical address `0xf0000`. And after, when GDT and protected mode flag are set, it jumps back to the 32-bit code, where it comes to paging bits and, finally, to the `inmate_main()` function entry (see Listing 2.4.

More details on how to create the inmate are provided below in sections 2.3.1 and 2.3.2.

2.3.1 APIC demo

APIC demo (stands for Advanced Programmable Interrupt Controller) is canonical inmate which usually used to demonstrate Jailhouse features (e.g. in [Jan15]). It is a tiny program (lied in `inmates/demos/x86/apic-demo.c`) which sets up an interrupt for the APIC timer and “measures actual time between the events happening”[Val15]. Besides that, it shows the basics of using the inter-cell communication and manipulating the cell state.

The configuration file (presented in Listing 2.8) for that cell is very laconic. It defines only two memory regions: the lowest (1 MB wide) where the inmate is loaded, and the little one (only 4 KB) is for communication. The second region has an additional flag `JAILHOUSE_MEM_COMM_REGION` to let the hypervisor know where to read/write messages. And it prints a log at the serial port 0.

Listing 2.6: Launching the `apic-demo` cell.

```
# jailhouse cell create /jailhouse/configs/apic-demo.cell
[   27.588227] smpboot: CPU 3 is now offline
[   27.610212] Created Jailhouse cell "apic-demo"
# jailhouse cell load apic-demo /jailhouse/inmates/apic-demo.bin
  -a 0xf0000
# jailhouse cell start apic-demo
# jailhouse cell shutdown apic-demo
JAILHOUSE_CELL_LOAD: Operation not permitted
# jailhouse cell shutdown apic-demo
#
```

Right after the demo starts, cell's state is set to state `JAILHOUSE_CELL_RUNNING_LOCKED`. This is done by an assignment to `comm_region->cell_state`. Usually, it means that hypervisor could not shrink this cell. After that the application calibrates the Time Stamp Counter (`inmates/lib/x86/timing.c`) and initializes APIC timer. Then handler is set for the timer's interrupt. So, when every next interrupt occurs, jitter is calculated. "Jitter is the difference between the expected and actual time (the latency), and the smaller it is, the less visible (in terms of performance) the hypervisor is."[Val15]

The program waits for a message in the communication region. If the shutdown request appears there, the program sends a message that it is not possible right now. If this request appears by the second time, `apic-demo` breaks the loop. Right before the final return `apic-demo` changes cell's status into `JAILHOUSE_CELL_SHUT_DOWN`, so the Jailhouse knows that shutdown process has gone well. Illustrations are given there: Listings 2.6,2.7.

Listing 2.7: Shorted (...) listing from apic-demo cell's operation.

```
Cell "apic-demo" can be loaded
Started cell "apic-demo"
CPU 3 received SIPI, vector 100
Calibrated TSC frequency: 3292506.587 kHz
Calibrated APIC frequency: 99773 kHz
Timer fired, jitter:    821 ns, min:    821 ns, max:    821 ns
Timer fired, jitter:   1090 ns, min:    821 ns, max:   1440 ns
(...)
Timer fired, jitter:   1261 ns, min:    821 ns, max:   1440 ns
Rejecting first shutdown request - try again!
Timer fired, jitter:   1418 ns, min:    821 ns, max:   1440 ns
(...)
Timer fired, jitter:   1306 ns, min:    821 ns, max:   1440 ns
Stopped APIC demo
Cell "apic-demo" can be loaded
```

Listing 2.8: The `apic-demo` cell configuration.(`configs/apic-demo.c`)

```
#include <linux/types.h>
#include <jailhouse/cell-config.h>

#define ARRAY_SIZE(a) sizeof(a) / sizeof(a[0])

struct {
  struct jailhouse_cell_desc cell;
    __u64 cpus[1];
    struct jailhouse_memory mem_regions[2];
    __u8 pio_bitmap[0x2000];
  } __attribute__((packed)) config = {
    .cell = {
    .signature = JAILHOUSE_CELL_DESC_SIGNATURE,
    .name = "apic-demo",

    .cpu_set_size = sizeof(config.cpus),
    .num_memory_regions = ARRAY_SIZE(config.mem_regions),
    .num_irqchips = 0,
    .pio_bitmap_size = ARRAY_SIZE(config.pio_bitmap),
    .num_pci_devices = 0,
  },

  .cpus = {
    0x8,
  },

  .mem_regions = {
    /* RAM */ {
    .phys_start = 0x3f000000,
    .virt_start = 0,
    .size = 0x00100000,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
    JAILHOUSE_MEM_EXECUTE | JAILHOUSE_MEM_LOADABLE,
  },
  /* communication region */ {
    .virt_start = 0x00100000,
    .size = 0x00001000,
    .flags = JAILHOUSE_MEM_READ | JAILHOUSE_MEM_WRITE |
    JAILHOUSE_MEM_COMM_REGION,
  },
},

.pio_bitmap = {
    [     0/8 ...  0x3f7/8] = -1,
    [ 0x3f8/8 ...  0x3ff/8] = 0, /* serial1 */
    [ 0x400/8 ... 0xe00f/8] = -1,
    [0xe010/8 ... 0xe017/8] = 0, /* OXPCIe952 serial1 */
    [0xe018/8 ... 0xffff/8] = -1,
  },
};
```

2.3.2 HPET demo

This example was implemented by me to become more familiar with Jailhouse and with the inmate's creation process. It is a demonstration of using the High Precision Event Timer (HPET) - event timer hardware which has its registers memory mapped and its interrupts routed through the IOAPIC chip. So, this is a problem where the set of `inmate.h` functions could be useful.

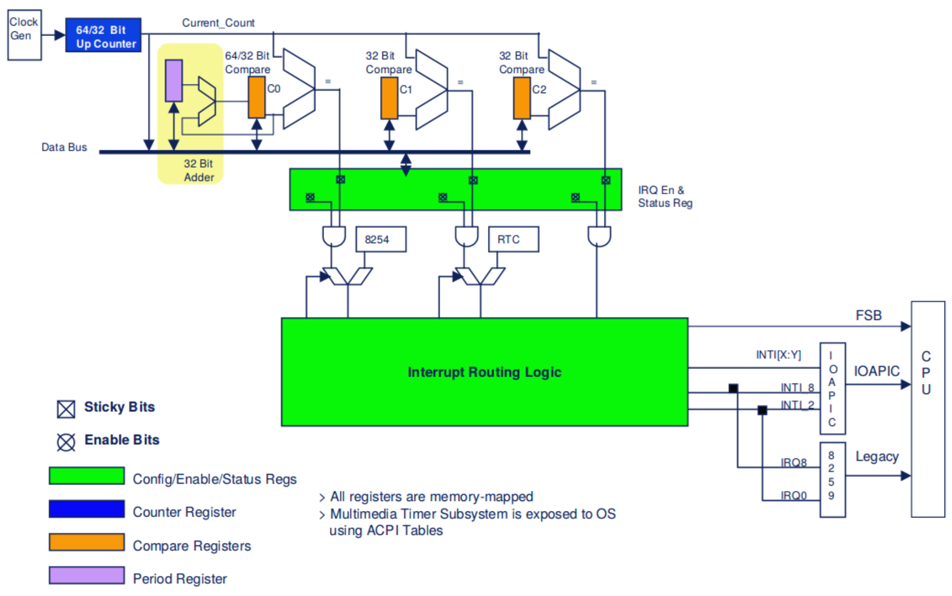

Figure 2.2: The High Precision Event Timer architecture overview. Source: [hpe]

Implementation form was inspired by the `apic-demo` and `ioapic-demo`, the set of constants was partly taken from Linux kernel sources. Of course, any step could not be done here without reading the specifications[hpe]. Please, refer to Figure 2.2 where the HPET architecture is clearly described. Briefly, HPET Architecture has several timers whose registers are mapped at the address which is found in ACPI tables. Timers must be configured through their capabilities' registers. Base parameters are the mode of an operation (periodic/one-shot), interrupt (where to route) and the *comparator*'s value field. When the value on some timer's comparator is equal to the main counter, the timer produces an interrupt.

The cell was configured in the similar way as it is done for `apic-demo` (refer to Listing 2.8), but two additional memory regions allowed there. The first one is where HPET found - [`0xfed00000-0xfed01000`], the second one is the IOAPIC space - [`0xfec00000 - 0xfec01000`]. Note, that HPET must be turned off in Linux (e.g. with appending the "nohpet" option to the kernel).

The program must find the General Capabilities Register address first. In a general case, ACPI tables must be parsed for it, but as far as we know our environment, it is not necessary (cell is configured statically, so the memory region has been already added there). So, I hardcoded the address at `0xfed00000` value (it is there in QEMU, and the same is in the majority of real cases). However to interact with this piece of memory, it is needed to remap it into the cell. And the function `map_range()`, (found in `inmate.h`), was used for that purpose. After this is done, access to that space is available. Information about the amount of registers and the address of the main counter register is taken from General register in the way how it should be done according to the spec [hpe]]. When all timers are enumerated, the program shows the basic info for everyone available. Then, legacy mode is turned on - it is just some attempt to make this demo more applicable because the specification predefines interrupts' IRQs in this mode (for the first and the second timer). This program operates with only the first three timers - all are set to the periodic mode. Finally, IOAPIC is initialized, and IRQs are assigned and, after enabling the main counter, all three comparators start to produce interrupts. Results are in Listing 2.9.

Listing 2.9: Demonstration of the HPET demo operation.

```
Created cell "hpet-demo"
Page pool usage after cell creation: mem 196/1505, remap
  65602/131072
Cell "hpet-demo" can be loaded
Started cell "hpet-demo"
CPU 3 received SIPI, vector 100

Base Address for HPET registers : 0x00000000fed00000

Timer 0 on: 0x00000000fed00100,
Timer comparator : 0xffffffffffffffff,
Interrupts where to route: 0x0000000000ff0104

Timer 1 on: 0x00000000fed00120,
Timer comparator : 0xffffffffffffffff,
Interrupts where to route: 0x0000000000ff0104

Timer 2 on: 0x00000000fed00140,
Timer comparator : 0xffffffffffffffff,
Interrupts where to route: 0x0000000000ff0104
Done preparation..

Timer 0 says hi!

Timer 0 says hi!

Timer 2 says hi!

Timer 1 says hi!
```

After all that long initialization process the program behaves the same way as apic-demo does - waits for a shutdown request from outside. And it handles three "Hello from timer <number>" interrupts.

Chapter 3
L4 Fiasco.OC launch

Running the bare-bones program inside a cell could be useful to solve simple problems, but, mostly, an operating system is required in real applications when something more complex has to be implemented (Network protocol stack, Autopilot, etc.). So, there is a motivation to port some OS as an inmate. Moreover, if Jailhouse has real-time properties, it is worth for OS to have that too.

Currently, it is possible to boot Linux in non-root cell, and the `Documentation/non-root-linux.txt` [Jan] file describes how that should be done. The kernel must be patched and configured in a specific way. User-space tool for bootstrapping that kernel exists also. However, this it would not be a real-time case still (without special kernel patches and configuration, which could be more complex).

That is why Fiasco.OC was chosen to port. It is small enough, and it does meet real-time requirements. It is quite configurable and has an environment which makes development process of the user-space applications much easier.

The following section (3.1) provides the small overview on an architecture of Fiasco.OC. Subsection (3.1.1) is dedicated to the L4 bootstrapper application which is used to launch L4-based systems. Next section (3.2) describes steps for launching Fiasco under Jailhouse.

3.1 Overview

Fiasco.OC is a microkernel-based operating system developed by the Fiasco Team at Technical University Dresden. It consists of the L4-based microkernel and user-level programs which are related to the L4 Runtime Environment (L4Re). The kernel itself is very minimalistic. Thus, it provides only base functionality as the Inter-Process Communications (IPC), creating/deleting address spaces (Tasks) and threads functionality. As it noticed about the Fiasco minimalism in [ZDM+09]: “The microkernel provides a total of seven system calls, in other words, microkernel rules the world with only 7 system calls.” All other responsibilities are lied on shoulders of L4Re.

The minimal configuration in which the OS could be launched must contain Fiasco kernel, root pager called *Sigma0*, root task *Moe* and at least one user-space application which runs on top of all it. Sigma0 provides the API for user-space program to work with memory (remapping, allocation and such). Moe, which operates above the paging manager, is a task which kernel starts in the first place. It serves more abstract interface for all other user-space applications.

The diagram, that could be found in Fig. 3.1, clearly describes the architecture of the L4 Fiasco.OC.

Detailed information about the architecture and programming references could be found in [l4-].

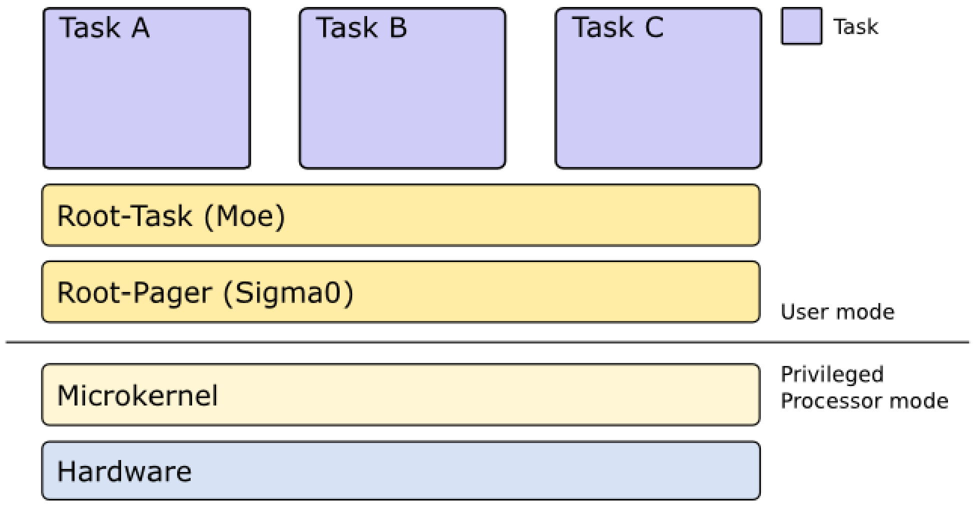

Figure 3.1: Basic Structure of an L4Re based system. Source:[l4-]

3.1.1 Fiasco bootstrapping process

Fiasco kernel itself is a Multiboot-compliant so that it could be booted for example via GRUB with modules which are added separately. However, for purposes to distribute the whole system as a single image in L4Re exists a package called L4 bootstrapper. The image could be built with it if the `make E=<entry-name>` is invoked in the L4 build system.

L4 bootstrapper also solves the problem of portability. Not only it supports being loaded by the Multiboot-compliant boot loader. It has an ability, for example, to be launched from Linux user-space (Fiasco-UX) or in XEN environment. It even supports launching from real-mode with PXElinux.

To set up boot configuration, an entry must be added to the `modules.list`. The example is in Listing 3.1.

Listing 3.1: Build entry for the helloworld example in modules.list

```
modaddr 0x01100000

default-kernel fiasco -serial_esc
default-bootstrap bootstrap

entry hello
roottask moe --init=rom/hello
module l4re
module hello
```

Bootstrapper sets up UART communication first, and then it tries to determine available system memory. If there is no faults, the bootstrapper search for modules in the image (modules were placed there as raw binaries after linking, see `bootstrap.ld.in` script in `l4/pkg/bootstrap/ARCH_x86`). Then it moves all modules behind the predefined address and jumps to the kernel start address.

Those operations mentioned above are platform specific. Different implementations are located into `bootstrap/platform` folder. For example, `x86_pc.cc` contains all necessary for the x86 PC.

3.2 Port Fiasco into cell

The following sections describe modifications which were contributed into the L4 bootstrapper (3.2.2) and into the Fiasco kernel (3.2.3) to launch it as a Jailhouse inmate. Section 3.2.1 contains information about cell configuration and host's Linux parameters. Log in Listing 3.9 presents the Fiasco.OC which successfully works in a cell.

3.2.1 Cell and host system configuration

As it has been mentioned earlier, the first step, when creating a new cell for the Jailhouse, is to configure it. Such configuration must describe resources which application requires otherwise it would not work.

The Fiasco kernel (and the bootstrap) uses:

- Ports from 0x3f8 to 0x3ff for sending debug info at the serial port.
- Port 0x80 to produce delays.
- Ports 0x20 and 0x21 when accessing to the Programmable Interrupt Controller (PIC) on the Master chip.
- Ports 0xA0 and 0xA1 when accessing to the PIC Slave chip.

- Ports from 0x40 to 0x43 when using the Programmable Interrupt Timer (PIT).
- Ports 0x60 and 0x64 when trying to operate with PS/2 keyboard.
- Memory - at least 3MB for the inmate image. The rest depends on user-space applications' requirements.

Section 2.2.2 explains how to describe it in configuration file (lies in `jailhouse/configs/fiasco-demo.c`).

Unfortunately, it is not enough just to allow this in configuration. It is also needed to ensure that Linux would not access to these ports. To avoid competition about I/O ports the following corrections were added to the Linux configuration (tested with the kernel version 4.5.0-rc4):

- Parameter `CONFIG_IO_DELAY_0XED` was turned on. This allows Linux to use the port 0xed instead of 0x80 as the I/O delay.
- Parameter `CONFIG_SERIO_I8042` was turned off to avoid all operations with PS/2 keyboard controller. Alternatively, the `i8042.nokbd` argument could be appended to the kernel command line at the boot time.

There is no need to worry about the PIC and the PIT as far as Linux uses the Advanced Programmable Interrupt Controller (APIC) and the APIC timer instead.

3.2.2 Bootstrap modification

According to what have been discussed above, an application must be built in a special way to become an inmate. Jailhouse does not provide any bootloader at all; it only sets up the instruction pointer at the address 0xffff0. Thus, the bootstrap process needs several customizations to boot the Fiasco successfully. Note please, that the following text is related to the i386 version.

An issue about the entry point difference was solved in the similar way how it is done with the demo inmates. It includes modifications of the startup code (`crt0.S`) and the linker script (`bootstrap.ld.in`). To start with, the addition (presented in Listing 3.2) were inserted into `crt0.S`. It is, basically, a part of the inmates' startup code which does a jump from the reset entry into the `.jh.boot` section in 16-bit code. There, the Global Descriptor Table (GDT) sets up and, after enabling the protected mode (bit 0 set in the CR0 register), the program counter moves to 32-bit code. There, Memory type range register (MTTR) sets up at Default Type which tells CPU that this part of memory could be cached. And then it goes to the original code of bootstrap (symbol `_start`). However, the `__reset_entry` symbol must be placed at the right address, and the linker script has to ensure it.

Listing 3.2: Startup code improvements in `crt0.S`.

```
#ifdef JAILHOUSE
 #define X86_CR0_PE 0x00000001
 #define MSR_MTRR_DEF_TYPE 0x000002ff
 #define MTRR_ENABLE  0x00000800
 #define INMATE_CS32 0x8

 .code16
 .section ".jh.boot", "ax"

 .globl __reset_entry
 __reset_entry:
  ljmp $0xf000,$start16

 .section ".jh.startup", "ax"

 start16:
  lgdtl %cs:gdt_ptr
  mov %cr0,%eax
  or $X86_CR0_PE,%al
  mov %eax,%cr0
  ljmpl $INMATE_CS32,$_start

 .code32

 .global loader_gdt
 loader_gdt:
  .quad 0
  .quad 0x00cf9b000000ffff
  .quad 0x00af9b000000ffff
  .quad 0x00cf93000000ffff
 gdt_ptr:
  .short gdt_ptr - loader_gdt - 1
  .long loader_gdt + FSEGMENT_BASE
 .align(4096)
 .global loader_pdpt
 loader_pdpt:
  .long 0x00000083
 .align(4096)

#endif //JAILHOUSE
 .section .init

 .globl _start
  _start:
#ifdef JAILHOUSE
  movl $MSR_MTRR_DEF_TYPE,%ecx
  rdmsr
  or $MTRR_ENABLE,%eax
  wrmsr
#endif //JAILHOUSE
```

Sections were relocated into the `bootstrap.ld.in` to satisfy the mentioned requirement. First of all, the `.boot` section was bound at `0xfff0`. The code which was added into `SECTIONS` is shown in Listing 3.3. Note also, that the resulting binary will be loaded with the `0xf0000` offset, so the right address for the entry will be achieved.

Listing 3.3: Placing the `.jh.boot` section in a binary.

```
#define LOAD_OFFSET (0x0)
#ifdef JAILHOUSE
#define LOAD_OFFSET (0xf0000)

. = 0;

/* 16-bit sections */
.jh-startup :  { *(.jh.startup) }

. = 0xfff0;
.jh-boot    :
{
 *(.jh.boot)
 . = ALIGN(16);
}
#endif
```

Moreover, that offset (defined in Listing 3.3 as the `LOAD_OFFSET`) must be considered when placing all other sections like e.g. `.text` and `.data`. In that case, Load Memory Address (LMA) must be specified without the offset as it is shown in the example for the `.text` section in Listing 3.4. The result of linking is presented in Listing 3.6.

Listing 3.4: Placing the `.text` section in a binary considering the load offset.

```
.text : AT (ADDR(.text) - LOAD_OFFSET)
{
 *(.init)
 *(.text .text.* .gnu.linkonce.t*)
 *(.rodata*)
}
```

The bootstrap must be built with the `REALMODE_LOADING` flag. It enables a piece of code which creates synthetic multi-boot info and uses information about the memory map provided by command line arguments. These arguments could be passed through the `BOOTSTRAP_CMDLINE` declaration so that the final image could have that build in. These build flags were appended to `Makeconf.local` as it is shown in Listing 3.5.

Listing 3.5: Additional build options declared in Makelocal.conf

```
 DEFINES += -DREALMODE_LOADING
 BOOTSTRAP_CMDLINE += -mem=1M@0x0 -mem=50M@0x100000
-maxmem=51M
```

Listing 3.6: Linked bootstrap with LMAs modified.

```
objdump -h bootstrap.elf

bootstrap.elf:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
0 .jh-startup   00002000  00000000  00000000  00001000  2**12
CONTENTS, ALLOC, LOAD, READONLY, CODE
1 .jh-boot      00000010  0000fff0  0000fff0  00003ff0  2**0
CONTENTS, ALLOC, LOAD, READONLY, CODE
2 .text         00009e24  002d0000  001e0000  00004000  2**5
CONTENTS, ALLOC, LOAD, READONLY, CODE
3 .data         00000124  002d9e40  001e9e40  0000de40  2**5
CONTENTS, ALLOC, LOAD, DATA
4 .data.module\_info 00000098  002d9f64  001e9f64  0000df64
 2**2
CONTENTS, ALLOC, LOAD, DATA
5 .bss          00004420  002da000  001ea000  0000dffc  2**5
ALLOC
6 .module\_data  000c44e8  002df000  001ef000  0000e000  2**12
CONTENTS, ALLOC, LOAD, CODE
```

3.2.3 Modifications in Fiasco kernel

It was necessary to add some changes to the source code of the Fiasco kernel. To interact with the local APIC (read/write to its memory mapped registers) it uses functions presented in Listing 3.7.

Listing 3.7: Definitions of the reading/writing to memory in Fiasco. `kernel/fiasco/src/kern/ia32/apic-ia32.cpp`

```
PUBLIC static inline Unsigned32
Apic::reg_read(unsigned reg)
{
  return *((volatile Unsigned32*)(io_base + reg));
}
PUBLIC static inline void
Apic::reg_write(unsigned reg, Unsigned32 val)
{
  *((volatile Unsigned32*)(io_base + reg)) = val;
}
```

After the compilation, there is generated an assembly instruction in format `mov $address, %eax` or `mov %eax, $address`, which is not acceptable in Jailhouse. Jailhouse cannot allow the cells to do whatever they want with the APIC, because it would allow escaping from the cell. Therefore Jailhouse has to intercept all the accesses to the APIC and allow only those that are safe. (see `apic_mmio_access(..)` in `hypervisor/arch/x86/apic.c`). As a result, there is the error `"FATAL: Unsupported APIC access"`. To avoid this situation, functions in Listing 3.7 were changed as it showed in Listing 3.8 and defined in the `inmate.h` in the inmate demos library. It forces the compiler to produce instructions in format `mov %ebx, %edx`, which is passed through the Jailhouse parser.

Listing 3.8: Improvments of reading/writing to APIC registers in Fiasco. `kernel/fiasco/src/kern/ia32/apic-ia32.cpp`

```
PUBLIC static inline Unsigned32
Apic::reg_read(unsigned reg)
{
  Unsigned32 val;
  /* assembly-encoded to match the Jailhouse hypervisor MMIO
parser support */
  void *address = (void*)(io_base + reg);
  asm volatile("mov (%1),%0" : "=r" (val) : "r" (address));
  return val;
}
PUBLIC static inline void
Apic::reg_write(unsigned reg, Unsigned32 val)
 {/* assembly-encoded to match the Jailhouse hypervisor MMIO
parser support */
  void *address = (void*)(io_base + reg);
  asm volatile("mov %0,(%1)" : : "r" (val), "r" (address));
}
```

Listing 3.9: Log from the Fiasco running Hello World demo. In a Jailhosue cell.

```
Initializing Jailhouse hypervisor  on CPU 2
Code location: 0xfffffffff0000030
Using xAPIC
Page pool usage after early setup: mem 38/16347, remap 65/131072
Initializing processors:
CPU 2... (APIC ID 4) OK
CPU 3... (APIC ID 6) OK
CPU 1... (APIC ID 2) OK
CPU 0... (APIC ID 0) OK
Found DMAR @0x00000000fed90000
Found DMAR @0x00000000fed91000
Reserving 24 interrupt(s) for device f0f8 at index 0
Adding PCI device 00:00.0 to cell "RootCell"
Adding PCI device 00:02.0 to cell "RootCell"
Reserving 1 interrupt(s) for device 0010 at index 24
Adding PCI device 00:14.0 to cell "RootCell"
Reserving 8 interrupt(s) for device 00a0 at index 25
Adding PCI device 00:16.0 to cell "RootCell"
Reserving 1 interrupt(s) for device 00b0 at index 33
Adding PCI device 00:19.0 to cell "RootCell"
Reserving 1 interrupt(s) for device 00c8 at index 34
Adding PCI device 00:1a.0 to cell "RootCell"
Adding PCI device 00:1b.0 to cell "RootCell"
Reserving 1 interrupt(s) for device 00d8 at index 35
Adding PCI device 00:1c.0 to cell "RootCell"
Reserving 1 interrupt(s) for device 00e0 at index 36
Adding PCI device 00:1c.2 to cell "RootCell"
Reserving 1 interrupt(s) for device 00e2 at index 37
Adding PCI device 00:1d.0 to cell "RootCell"
Adding PCI device 00:1e.0 to cell "RootCell"
Adding PCI device 00:1f.0 to cell "RootCell"
Adding PCI device 00:1f.2 to cell "RootCell"
Reserving 1 interrupt(s) for device 00fa at index 38
Adding PCI device 00:1f.3 to cell "RootCell"
Adding PCI device 02:00.0 to cell "RootCell"
Reserving 8 interrupt(s) for device 0200 at index 39
Adding PCI device 02:00.1 to cell "RootCell"
Reserving 8 interrupt(s) for device 0201 at index 47
Page pool usage after late setup: mem 2105/16347, remap
 16452/131072
Activating hypervisor
Created cell "fiasco-demo"
Page pool usage after cell creation: mem 2121/16347, remap
 16452/131072
Cell "fiasco-demo" can be loaded
Started cell "fiasco-demo"
CPU 1 received SIPI, vector 100
cmdline:0x2d8823, realmode_si=(nil)

L4 Bootstrapper
Build: #298 Tue May 17 17:16:45 CEST 2016, x86-32, 4.9.2
```

```
cmdline params: '-mem=1M@0x0 -mem=50M@0x100000 -maxmem=51M'
RAM: 0000000000000000 - 00000000000fffff: 1024kB
RAM: 0000000000100000 - 00000000032fffff: 51200kB
Total RAM: 51MB
Scanning fiasco
Scanning sigma0
Scanning moe
Moving up to 5 modules behind 1100000
moving module 02 { 372000-3a34e7 } -> { 1193000-11c44e7 }
 [201960]
moving module 01 { 366000-37130f } -> { 1187000-119230f }
 [45840]
moving module 00 { 318000-365337 } -> { 1139000-1186337 }
 [316216]
moving module 04 { 2fb000-317537 } -> { 111c000-1138537 }
 [116024]
moving module 03 { 2df000-2fa44f } -> { 1100000-111b44f }
 [111696]
Loading fiasco
Loading sigma0
Loading moe
find kernel info page...
found kernel info page at 0x400000
Regions of list 'regions'
[     1000,      1fff] {     1000} Kern   fiasco
[     2000,      20eb] {       ec} Root   mbi_rt
[   100000,    10f193] {     f194} Sigma0 sigma0
[   140000,    177287] {    37288} Root   moe
[   2d0000,    2de41f] {     e420} Boot   bootstrap
[   300000,    38ffff] {    90000} Kern   fiasco
[   400000,    44efff] {    4f000} Kern   fiasco
[  1100000,   1138fff] {    39000} Root   Module
API Version: (87) experimental
Sigma0 config    ip:001001ec sp:00000000
Roottask config  ip:0014020e sp:00000000
Starting kernel fiasco at 00300798

Welcome to L4/Fiasco.OC!
L4/Fiasco.OC microkernel on ia32
Rev: fb3ab8c-dirty compiled with gcc 4.9.2 for Intel Pentium
 []
Build: #11 Mon May 16 16:25:15 CEST 2016

Performance-critical config option(s) detected:
CONFIG_NDEBUG is off

Superpages: yes
Kmem:: cpu page at 2eed000 (4096Bytes)

KERNEL: Warning: ACPI: Could not find RSDP, skip init
Allocate cpu_mem @ 0xfc6f0400
FPU0: SSE AVX
```

```
Local APIC[02]: version=15 max_lvt=6
APIC ESR value before/after enabling: 00000000/00000000
Using the Local APIC timer on vector f8 (Periodic Mode) for
 scheduling
ACPI: cannot find FADT, so suspend support disabled
Absolute KIP Syscalls using: Sysenter
Enable MSI support: chained IRQ mgr @ 0xfc6f0024
SERIAL ESC: allocated IRQ 4 for serial uart
Not using serial hack in slow timer handler.
CPU[0]: GenuineIntel (6:3A:9:0)[000306a9] Model:
 Intel(R) Core(TM) i5-3550 CPU @ 3.30GHz at 3292MHz

128 Entry I TLB (4K pages)
64 Entry D TLB (4K pages) 512 Entry D TLB (4k or 4M pages)

Freeing init code/data: 24576 bytes (6 pages)

Calibrating timer loop... done.
MDB: use page size: 22
MDB: use page size: 12
SIGMA0: Hello!
KIP @ 400000
Found Fiasco: KIP syscalls: yes
allocated 4KB for maintenance structures
SIGMA0: Dump of all resource maps
RAM:------------------------
[0:0;fff]
[4:2000;2fff]
[0:3000;fffff]
[0:110000;13ffff]
[4:140000;177fff]
[0:178000;3fffff]
[0:449000;10fffff]
[4:1100000;1138fff]
[0:1139000;2eeafff]
IOMEM:----------------------
[0:3300000;fedfffff]
[0:fee01000;ffffffff]
IO PORTS--------------------------
[0:0;fffffff]
MOE: Hello world
MOE: found 47224 KByte free memory
MOE: found RAM from 2000 to 2eeb000
MOE: allocated 46 KByte for the page array @0x3000
MOE: virtual user address space [0-bfffffff]
MOE: rom name space cap -> [C:103000]
BOOTFS: [1100000-111b450] [C:105000] l4re
BOOTFS: [111c000-1138538] [C:107000] hello
MOE: cmdline: moe --init=rom/hello
MOE: Starting: rom/hello
MOE: loading 'rom/hello'
Hello World!
```

Chapter 4
Benchmarks

The following sections are dedicated to the discussion about evaluating the influence of shared memory hierarchy (caches, DRAM) on the performance of software running in different cells. Section 4.1 determines goals of such study and which results are expected to be achieved. Implementation of the testing suite is described in section 4.2. The results of the implemented tests are presented in section 4.3.

4.1 Goal

A common multi-core processor has L3 cache shared between cores, and all the rest memory hierarchy is also shared. Suppose now, that every core does the entirely different program, so, everyone needs access to the different areas of memory. In that case, the cores will compete with each other for the cache (i.e. cache trashing) and memory access time will be increased. Also, what is more important, the problem of 'bottleneck' appears. When one core wants access to memory at the same time the other core does, one of them needs to wait. A memory-intensive application on one core can significantly slow down applications on other cores.

All those mentioned issues have a negative influence on real-time and safety of a system. The reason is well explained in [BMVB]: "...current techniques for timing analysis are not effective when applied to the complex hierarchical memory system of modern many-cores. The reason is that classic real-time theory usually views memory latencies as implicit components of the worst-case execution time of tasks, and the interference among cores concurrently accessing memory is upper-bounded to provide a safe worst-case analysis.".

The goal for benchmarking the Jailhouse is to investigate how significant that influence will be in a case when one cell does accesses to memory, and so does the other cell e.g. root one. Moreover, when this value is known, it could give the overview of Jailhouse real-time properties and open the field for further improvements.

4.2 Implementation

To achieve goals mentioned in the previous section, it was decided to implement a simple benchmark that could run both inside a Jailhouse's cell and on bare hardware. Conditions of that launching must be the same in both cases on purpose to investigate if any slowdown of memory access comes while using Jailhouse.

Moreover, to demonstrate the 'bottleneck' problem the same benchmark could be used. That benchmarking program is running inside a cell. It measures an efficiency (bandwidth) of its accesses to memory of different size. So if some memory-using program starts in Linux at this time, then the inmate probably could notice a slowdown. The program which runs into Linux uses the same memory amount over and over again, thus, for the concrete accessing size an interaction brings more (or less) slowdowns into the cell. And after that, this test should be done in another direction; Benchmarking program do measurements inside the root (Linux) cell, and Fiasco makes an external interference.

That benchmarking inmate consists of the ported Fiasco.OC (described in Chapter 3) and a small program running in the Fiasco's user-space. The benefit of using this OS there is that it only needs to swap the Bootstrapper (see 3.1.1) to run the whole suite on bare hardware.

This simple benchmark was implemented as follows (code is given in Listing 4.1). At the every step, it initializes an array with pointers which store an address of the next array element. The last pointer will have the address of the first element of the array, so all that represents a cyclic linked list. The `size` variable determines how long that list will be. Then, the test is started, and the program goes through the list pointer by pointer forcing a CPU to access the memory over and over as long as the value of `REPEATS` runs out. It accesses only the memory amount of particular `size`, so it determines the load level which that test brings into the system. When there are no more repeats left, the time of the whole test is calculated, and the average time of one memory read is reported. On every next stage, the `size` is multiplied by two, and the program tries a the memory amount. The range of it is from 1 to 32 MiB. Time is measured in clock cycles using the `rdtsc` assembly instruction.

If the `ALL_WORKSETS_BENCH` macro is not defined, then, instead of changing the `size` on every step, the program uses only one given, so it generates a constant load.

Listing 4.1: The algorithm of implemented benchmark.

```
struct s {
  char dummy[56];
  struct s *ptr;
};
struct s array[0x1000000/sizeof(struct s)];
#define REPEATS (0x20000000)

static __inline__ uint64_t rdtsc(void)
{
  uint32_t a, d;
  asm volatile("rdtsc" : "=a" (a), "=d" (d));
  return (((uint64_t)a) | (((uint64_t)d) << 32));
}

int main(int argc, char *argv[])
{
  unsigned int size;
#ifdef ALL_WORKSETS_BENCH
  for (size = 1024; size <= sizeof(array); size *= 2)
#else //!ALL_WORKSETS_BENCH
  size = WORKSET_SIZE;
  if (argc - 1) {
    char * nxt_cr;
    size = strtoul(argv[1], &nxt_cr, 0);
  }
  while (1)
#endif
  {
    unsigned int i;
    for (i=0; i < size / sizeof(array[0]); i++)
      array[i].ptr = &array[i+1];
    array[ size / sizeof(array[0]) - 1].ptr = &array[0];

    i = REPEATS;
    volatile struct s *p = &array[0];
    uint64_t tic, tac;

    tic = rdtsc();
    while (i--) {
      p = p->ptr;
    }
    tac = rdtsc();
    printf("%d %llu\n", size, (tac - tic) / REPEATS);
    fflush(stdout);
  }
  return 0;
}
```

Listing 4.2: Processor characteristics on hardware that was used for tests.

```
# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Model name:            Intel(R) Core(TM) i5-3550 CPU @ 3.30GHz
Stepping:              9
CPU MHz:               3292.379
BogoMIPS:              6584.75
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-3
```

The hardware on which tests are running has the configuration described in Listing 4.2. The processor consists of 4 separate CPU cores. Each core has the 32 KiB L1 data cache and 256 KiB L2 cache. L3 cache is 6 MB wide and is shared among all cores. According to this characteristics, next test-cases are assigned: while all work sets size benchmark is running in one cell, loads of 64 KiB, 256 KiB, 1 MiB, 4 MiB, 8 MiB, 16 MiB and 32 MiB are applied in the other cell. This makes possible evaluation of critical points, where the 'bottleneck' effect could appear.

4.3 Results

Benchmarks' results are presented in Figures 4.1 and 4.2. Points of each line show how much time (clock cycles) took the average memory access while benchmark operated with the working set of particular size (on the X axis).

As it could be seen, the difference for Fiasco.OC running bare metal and above the hypervisor is not noticeable. The same applies for the Linux also. So, Jailhouse contributes almost no influence at inmates' memory access bandwidth. However, as it has been mentioned earlier, the asymmetric multiprocessing still comes for a price.

External loads which are less than 256 KiB have not affected the curve form because data of such size could be placed in L1 and L2 caches locally for each core. However, when the measured process is influenced by larger size accesses from the other cell, the slowdown of 8-10% is noticed when measured process operates with data sets less than 1 MB. This could be explained by the fact that Intel processor has an inclusive cache.

There is almost no competition, as far as the probability of access to shared memory is minimal. Unfortunately, the slowdown effect of shared L3 cache takes its place. All curves start to grow near the point of 1 MB while a sum of required memory sets becomes closer to the 6 MiB - the size of the L3 cache. It is could be seen in Fig. 4.2: the huge difference between the no-load test and 32MB test.

Surprisingly, Linux cell has got much slower accesses (peak values) than Fiasco which has only one CPU assigned.

The worst case 'bottleneck' influence achieved during tests is approximately **220 % slowdown**. This result was obtained when the benchmark run on just two cores out of four available. When the benchmark is run on all cores in parallel, we expect the slowdown to be even higher.

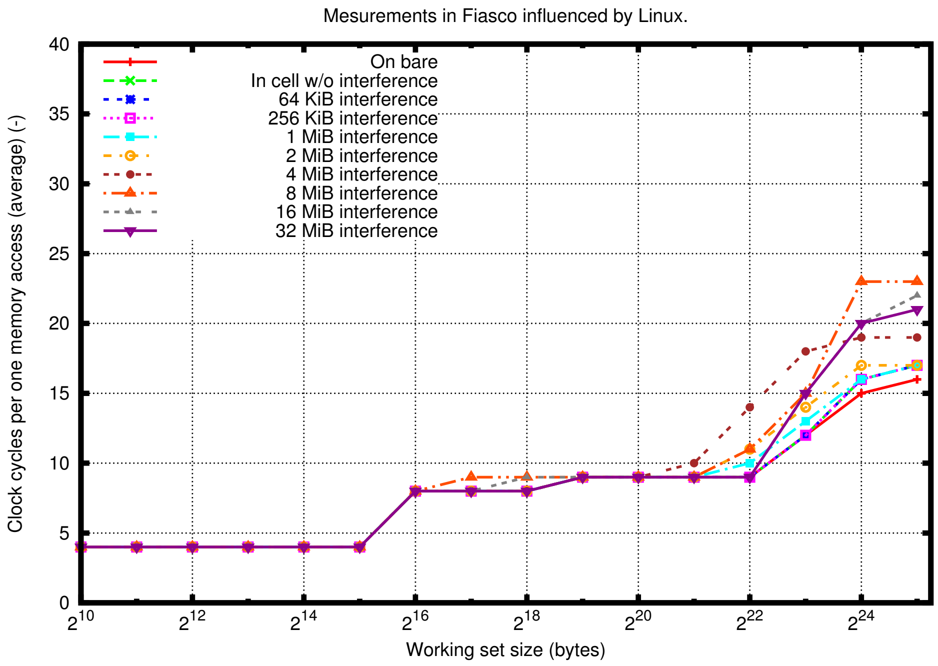

Figure 4.1: Results of measurements from the Fiasco (non-root cell) perspective.

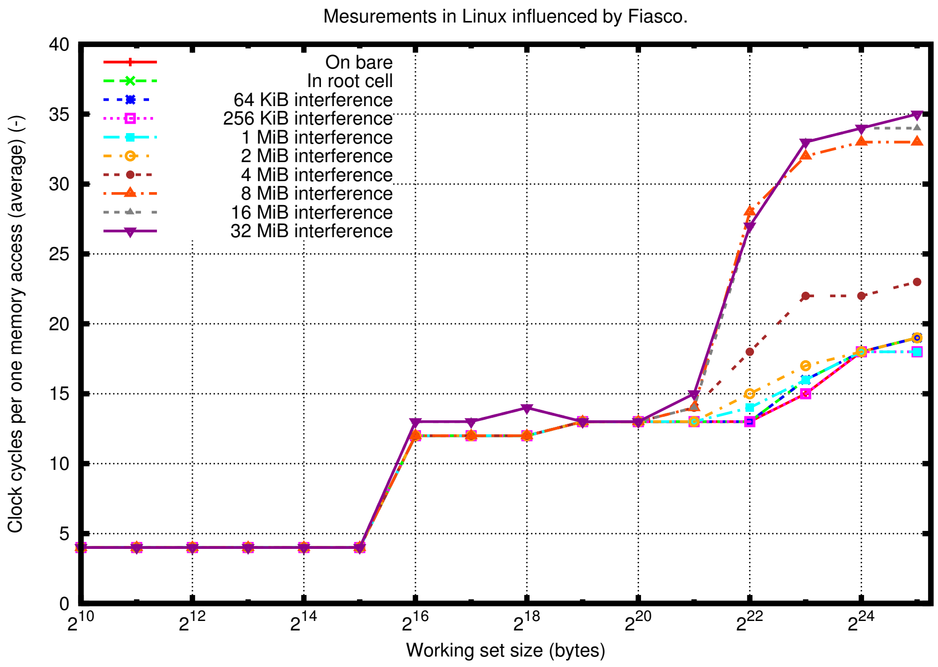

Figure 4.2: Results of measurements from the Linux (root cell) perspective.

Chapter 5
Conclusion

I have studied the concepts and operation principles of Jailhouse hypervisor and described it in this thesis. For purpose to become more familiar with Jailhouse hypervisor's environment simple demo was developed. It sets up High Precision Event Timer to send interrupts periodically or in one-shot mode and handle it. Such example shows how the OS-less program, which requires interacting with IOAPIC chip and other memory mapped hardware, was implemented using the Jailhouse `inmate` library.

L4 Fiasco.OC was successfully ported into Jailhouse cell. It is fully functional and there were almost no differences between its operation into cell and bare-metal. To achieve this the L4 Bootstrapper was modified: startup code and linker script were edited to build the binary which meets the Jailhouse requirements. Also, kernel code was slightly modified to solve a problem with APIC access.
Simple benchmarks were implemented and applied on the test-case where two cells could interfere with each other while accessing to the memory simultaneously. The benchmarking programs are made both for Linux and L4 Fiasco.OC user-spaces. This made possible to determine that a presence of shared memory hierarchy brings the slowdown of %220 in the worst case.

This work may be continued in future to bring more enchantments into all mentioned points. Improvements of this demo are possible; It could be enhanced to be more platform independent. For example, the base address of HPET might be read from ACPI tables, and available interrupts for the setup could be determined automatically. It even could be improved to the full-functional driver. Fiasco.OC port also has some field for new features e.g. enabling multiprocessor support in the cell for it, or improving the adopted bootstrapper to achieve an ability to start the x86_64 version of Fiasco kernel.

The repository containing all source code related to this work is publicly available at `http://rtime.felk.cvut.cz/gitweb/jailhouse-test.git`.

Appendix A

Bibliography

[BMVB] Paolo Burgio, Andrea Marongiu, Paolo Valente, and Marko Bertogna, *A memory-centric approach to enable timing-predictability within embedded many-core accelerators.*

[hpe] *IA-PC HPET (High Precision Event Timers) Specification*, [online]`http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf`, Accessed: 2016-05-26.

[Jan] Kiszka Jan, *Github - siemens/jailhouse: Linux-based partitioning hypervisor.*, [online]`https://github.com/siemens/jailhouse.git`, Accessed: 2016-05-26.

[Jan15] ______, *Hard partitioning for linux: The jailhouse hypervisor*, [online] `http://events.linuxfoundation.org/sites/events/files/slides/LinuxConNA-2015-Jailhouse_0.pdf`, August 2015, Accessed: 2016-05-26.

[l4-] *L4re - l4 runtime environment*, [online]`https://l4re.org/doc/l4re_intro.html`, Accessed: 2016-05-26.

[lin] *Red hat enterprise linux 4: Using ld, the gnu linker*, [online]`https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Using_ld_the_GNU_Linker/scripts.html#BASIC-SCRIPT-CONCEPTS`, Accessed: 2016-05-26.

[ME] Sainz Manuel Estrada, *Linux kernel documentation - request_firmware hotplug interface.*, [online]`https://www.kernel.org/doc/Documentation/firmware_class/README`, Accessed: 2016-05-26.

[Val14a] Sinitsyn Valentine, *Understanding the jailhouse hypervisor, part 1.*, LWN.net (2014).

[Val14b] ______, *Understanding the jailhouse hypervisor, part 2.*, LWN.net (2014).

[Val15] ______, *Jailhouse*, Linux Journal (2015).

[ZDM$^+$09] Q. Zhou, Y. Ding, N. McGuire, C. Li, G. Cheng, and B. Hu, *A case study of microkernel for education*, 133–136.