CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

BEng Thesis

Exploring IVSHMEM in the Jailhouse Hypervisor

Orientação científica: Cláudio Maia

Diana Ramos

CISTER-TR-191211

Exploring IVSHMEM in the Jailhouse Hypervisor

Diana Ramos

CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

https://www.cister-labs.pt

Abstract

Nowadays, modern multicore processors come with virtualization featuresthat provide the creation of different virtual environments inside the samemachine, which magnify its ability to use all the resources available. Thecombination of multiprocessor systems with virtualization is highly demandedby the embedded systems domain.Virtualization technologies like hypervisor software are responsible for managingvirtual machines and control their access to physical resources. Severalvirtualization technologies and hypervisors exist for different industrydomains. One of those, is Jailhouse, a static partitioning hypervisor thatpartitions the hardware resources and directly assigns applications to eachpartition, providing them with access to the actual physical resources.This hypervisor focuses on giving the applications the isolation they need;however this can be seen as a limitation as it may restrict communicationbetween applications running in different partitions. There are inter virtualmachine 19s communication mechanisms (based on networking or sharedmemory)that solves this limitation. This project aims at exploring this aspectin Jailhouse and focuses on using the Jailhouse hypervisor and ashared-memory mechanism to manage to send information between twopartitions.Specifically, it aims at: (1) understanding the jailhouse hypervisor with respectto its features. For that, demonstrations are executed on top of twoarchitectures, Intel based x86-64, using QEMU; and ARM, using a BananaPi-M1 board; and (2) understanding a shared memory-based communicationprotocol, denoted as IVSHMEM, and create a use case in which twopartitions exchange information using this protocol.Results of this work are promising as the x86-64 use case was successfullyexecuted on top of QEMU, however, the ARM use case is still an on-goingendeavour.

© 2019 CISTER Research Center
www.cister-labs.pt

Exploring IVSHMEM in the Jailhouse Hypervisor

CISTER - Centro de Investigação em Sistemas Computacionais Embebidos e de Tempo-Real

2018/2019

1161209 Diana Ramos

Exploring IVSHMEM in the Jailhouse Hypervisor

CISTER - Centro de Investigação em Sistemas Computacionais
Embebidos e de Tempo-Real

2018 / 2019

1161209 Diana Ramos

Degree in Computing Engineering

September of 2019

Advisor: **Cláudio Ribeiro Maia**

«to my family, that have always supported me»

Acknowledgments

I would like to thank my adviser Cláudio Maia, David Pereira and Pedro Santos for all the support, availability and interest in helping me in everything I needed during my project.
Moreover I would like to thank CISTER members that provided me with good moments and support and CISTER overall for the internship opportunity.

Abstract

Nowadays, modern multicore processors come with virtualization features that provide the creation of different virtual environments inside the same machine, which magnify its ability to use all the resources available. The combination of multiprocessor systems with virtualization is highly demanded by the embedded systems domain.

Virtualization technologies like hypervisor software are responsible for managing virtual machines and control their access to physical resources. Several virtualization technologies and hypervisors exist for different industry domains. One of those, is Jailhouse, a static partitioning hypervisor that partitions the hardware resources and directly assigns applications to each partition, providing them with access to the actual physical resources.

This hypervisor focuses on giving the applications the isolation they need; however this can be seen as a limitation as it may restrict communication between applications running in different partitions. There are inter virtual machine's communication mechanisms (based on networking or shared-memory) that solves this limitation. This project aims at exploring this aspect in Jailhouse and focuses on using the Jailhouse hypervisor and a shared-memory mechanism to manage to send information between two partitions.

Specifically, it aims at: (1) understanding the jailhouse hypervisor with respect to its features. For that, demonstrations are executed on top of two architectures, Intel based x86-64, using QEMU; and ARM, using a Banana Pi-M1 board; and (2) understanding a shared memory-based communication protocol, denoted as IVSHMEM, and create a use case in which two partitions exchange information using this protocol.

Results of this work are promising as the x86-64 use case was successfully

executed on top of QEMU, however, the ARM use case is still an on-going endeavour.

Keywords (Theme): embedded systems, virtualization, isolation.
Keywords (Technologies): hypervisor, Jailhouse, QEMU, Banana Pi-M1, TCP/IP, shared memory, IVSHMEM.

Resumo

Hoje em dia, os processadores modernos com vários cores já vêm com virtualização que fornece a criação de diferents ambientes virtuais dentro da mesma máquina, o que permite usar de melhor forma os recursos disponíveis. Esta combinação de sistemas multiprocessador com virtualização é altamente procurada no domínio de sistemas embebidos.

Tecnologias de virtualização como o hipervisor é responsável por gerir máquinas virtuais e controlar o acesso aos recursos físicos. Estas tecnologias e hipervisores existem para diferentes domínios da indústria. Um desses é Jailhouse, um hipervisor de particionamento estático que particiona o hardware e atribui diretamente a aplicações de cada partição, fornecendo acesso aos recursos físicos.

Este hipervisor foca se em isolar as aplicações. No entanto, o isolamento pode ser visto como uma inconveniência pois podes limitar a comunicação entre aplicações de partições diferentes. Existem mecanismos de comunicação entre máquinas virtuais (via rede ou memória partilhada) que vem a resolver esta limitação. Este projeto foca-se em explorar o Jailhouse e um mecanismo de memória partilhada de forma a que duas partições sejam capazes de receber informação uma da outra.

Especificamente, o projeto tem os seguintes objetivos: (1) perceber as funcionalidades do Jailhouse, correndo demonstrações em várias arquiteturas: Intel x86-64 utilizando o QEMU e em ARM usando a placa Banana Pi-M1 e (2) entender o funcionamento de um mecanismo de memória partilhada chamado de IVHSMEM e criar um caso de uso onde duas partições comunicam uma com a outra utilizando esse protocolo.

Os resultados deste trabalho provaram ser promissores visto que tanto o hipervisor como as demonstrações correram nas várias arquiteturas (x86-

64 e ARM) com sucesso, no entanto o caso de uso em ARM utilizando IVSHMEM é um trabalho ainda em andamento.

Palavras-chave (Tema): sistemas embebidos, virtualização, isolamento.

Palavras-chave (Tecnologias): hypervisor, Jailhouse, QEMU, Banana Pi-M1, TCP/IP, memória partilhada, IVSHMEM.

Contents

Abstract VII

Resumo IX

List of Figures XV

Acronyms XVII

1 Introduction 1

1.1 Project Context . . . 2

1.1.1 Problem Description . . . 3

1.1.2 Approach . . . 4

1.1.3 Objectives and Contributions . . . 5

1.1.4 Organization . . . 5

1.2 Report Structure . . . 6

1.3 Work Planning . . . 6

1.3.1 Meetings . . . 6

2 State-of-Art 9

2.1 Hypervisors . . . 9

2.1.1 Hypervisor's Classification . . . 9

2.1.2 The KVM Hypervisor . . . 10

2.1.3 The Xen Hypervisor . . . 11

2.1.4 The Jailhouse Hypervisor . . . 12

2.2 Inter-VM Communication . . . 13

2.2.1 TCP/IP-based Communication 13
2.2.2 Shared-Memory Communication 14

3 Jailhouse Hypervisor **17**
3.1 Jailhouse Overview 17
3.1.1 Code Organization 19
3.1.2 Jailhouse Requirements 20
3.1.3 Linux Kernel Modules 23
3.1.4 Jailhouse's Kernel Module 25
3.1.5 Jailhouse Functionality 27
3.1.6 Jailhouse's Cell States 30
3.1.7 Run Time Failing Errors 30
3.2 Jailhouse's Demonstrations 32
3.2.1 QEMU Demonstration 33
3.2.2 Banana Pi Board 39

4 IVSHMEM - The Inter-Vm SHared Memory **43**
4.1 Design of IVSHMEM 43
4.1.1 The Configuration Section 44
4.1.2 The Base Address Registers 45
4.1.3 The Register Memory 45
4.2 Message Signaled Interrupts 46
4.2.1 MSI's Functionality 47
4.3 Jailhouse's Version of IVSHMEM 47
4.3.1 IVSHMEM Demonstration 48
4.3.2 IVSHMEM Demonstration on Banana Pi-M1 51
4.3.3 Solution Implementation 52

5 Conclusion **55**
5.1 Solution Implementation 55
5.2 Limitations and Future Work 58

5.3 Final Appreciation 58

Bibliography **60**

Attachments

A Setting up Jailhouse with BPI-M1 **66**

A.1 Formatting BPI-M1 sd Card 66

A.2 Adjust U-boot 67

A.3 Cross Compiling Kernel for ARM on x86 67

A.4 Installing the Linux kernel on BPI-M1 68

A.5 Cross Compiling Jailhouse for ARM on x86 69

A.6 Installing jailhouse 69

A.7 GIC-demo's Outputs 71

B Jailhouse's IVSHMEM related files **72**

B.1 IVSHMEM Demonstration Configuration File 72

B.2 IVSHMEM Demonstration Inmate File 76

B.3 IVSHMEM 2.0 82

B.4 Parking Errors 84

List of Figures

1.1 Static partitioning example. 3
1.2 Tasks and milestones by chronologic order. 7
1.3 Meetings table. 8

2.1 Hypervisor's type 1 and 2 designs. 10
2.2 Basic architecture of the KVM Hypervisor. 11
2.3 Basic architecture of the Xen Hypervisor. 12
2.4 Basic architecture of a system using the Jailhouse Hypervisor. 13
2.5 Basic architecture of TCP/IP protocol. 14
2.6 Basic architecture of shared memory communication protocol. 15

3.1 Jailhouse bootstrap process taken from [40]. 17
3.2 Jailhouse's system architecture taken from [14]. 18
3.3 Jailhouse's package diagram. 19
3.4 Jailhouse's hardware requirements for both architectures. . . 22
3.5 Jailhouse's software requirements for both architectures. . . . 23
3.6 Loaded modules. 25
3.7 Kernel module transactions adapted from [15]. 26
3.8 Jailhouse installation Makefile. 26
3.9 Jailhouse's command line tool options. 27
3.10 Jailhouse's hypercall cell states. 30
3.11 Jailhouse cell states. 31
3.12 Jailhouse unhandled traps message. 32
3.13 Jailhouse unhandled trap console message. 32

3.14 Jailhouse unhandled traps conditions. 32
3.15 Jailhouse parking error. 33
3.16 QEMU architecture. 33
3.17 QEMU images list. 36
3.18 Bash history for QEMU's AMD systems. 36
3.19 PIC architecture taken from [22]. 37
3.20 APIC architecture taken from [22]. 38
3.21 BananaPi-M1 specification. 39

4.1 IVSHMEM or Nahanni's overview adapted from [32]. 44
4.2 IVSHMEM PCI device architecture adapted from [32] 45
4.3 IVSHMEM PCI device registers adapted from 4.3. 46
4.4 IVSHMEM PCI device doorbell adapted from 4.3. 46
4.5 Activity diagram of IVSHMEM demonstration. 51
4.6 Sequence diagram for IVSHMEM demonstration. 52
4.7 Finding the device function in ivshmem-demo inmate. 53

Acronyms

AMD Advanced Micro Devices

AMD-V Advanced Micro Devices Virtualization

AMD64 Advanced Micro Devices 64

ARM Acorn Risc Machine

APIC Advanced Programmable Interrupt Controller

BPI Banana Pi-M1

CPU Central Processor Unit

CISTER Centro de Investigação em Sistemas Computacionais Embebidos e de Tempo-Real

DTB Device Tree Binarie

DMA Direct Memory Access

DHCP Dynamic Host Configuration Protocol

EPT Extended Page Tables

FUSE Filesystem in Userspace

GIC Generic Interrupt Controller

HVM Hardware Virtual Machine

I/O Input/Output

IOMMU I/O Memory Management Unit

ISEP Instituto Superior de Engenharia do Porto

IPC Inter Process Communication

IVSHMEM Inter-VM SHared MEMory

IP Internet Protocol

IRQ Interrupt Requests

ISR Interrupt Status Register

IMR Tnterrupt Mask Register

KVM Kernel Virtual Machine

LAPIC Local APIC

MMIO Memory-Mapped for the I/O

MSI Message Signaled Interrupts

MSI-X Message Signaled Interrupts-X

NPT Nested Page Tables

OS Operating System

PCI Peripheral Component Interconnect

POSIX Portable Operating System Interface

PSCI Power State Coordination Interface

PPI Private Peripheral Interrupt

PESTI Projeto Estágio

QEMU Quick Emulator

RTOS Real Time Operating System

SPARC Scalable Processor ARChitecture

SVM Secure Virtual Machine

SPI Shared Peripheral Interrupt

SDL Simple DirectMedia Layer

SGI Software Generated Interrupt

TCP Transmission Control Protocol

UDP User Datagram Protocol

UIO Userspace I/O

VM Virtual Machine

VT Virtualization Technology

VT-d Virtualization Technology for Directed I/O

VCPU Virtual CPU

VMX Virtual Machine Extension

VNC Virtual Network Computing

1. Introduction

Virtualization is a technique that allows one to logically divide the system resources between the applications executing on top of it [51]. There are multiple ways of achieving virtualization. For instance, using desktop virtualization [1], server virtualization [2] or even hardware virtualization, the topic of study of the work reported in this document.

By definition, hardware virtualization occurs at the hardware level, that is, hardware virtualization targets hardware machine resources (such as main memory, hard disks, processor, network related hardware and peripheral devices) to create an abstraction layer for each of these resources. In a hardware virtualization setting, there is an intermediate layer between the hardware tier and software tier named the hypervisor layer [42] [32].

A hypervisor is a virtualization software that enables the creation and management of Virtual Machines (VMs) [16] often called "guests" executing over a physical machine, denoted as "host". There are many hypervisors out there in the market, each with its own purpose, as for instance Xen [55] and Kernel Virtual Machine (KVM) [54], Jailhouse [40] and many more, however in this report the attention is devoted to the Jailhouse hypervisor.

The Jailhouse hypervisor is a static partition-based hypervisor that separates the machine's available hardware resources into persistent divisions, denoted as partitions or "cells" in Jailhouse terminology, to which one may assign different types of applications, denoted as "inmates" in Jailhouse terminology. These applications include bare-metal applications or operat-

[1] Desktop virtualization is the virtualization of workstations, allowing multiple users to remotely access each one of them logically [34].

[2] Server virtualization enables the creation of multiple virtual environments with one physical server [34].

ing systems (OS), including Linux and other specific operating systems for embedded devices such as Real-time Operating Systems (RTOS)[3]. One of the features provided by Jailhouse is the isolation between partitions [24].
Figure 1.1 depicts an example of how Jailhouse partitions the hardware by the cells. In the example, there are two cells where each one has access to one core and a device (representing for instance a network card, a hard drive, a bus, etc.). In the context of the cell, one can assign inmates (i.e., applications or OSes), with the guarantee that they execute in isolation in their own environment.
Isolation has its advantages (as for instance it allows cells to execute in their own context and therefore avoid being interfered or share their resources with other cells) but at the same time it brings the challenge of how two applications in different cells can share information among them.
The project described in this report focuses on the Jailhouse hypervisor and the challenge of exchanging information between different partitions in order to transfer data between them. Due to the isolation provided by Jailhouse one has to use specific protocols to exchange information such as networking mechanisms like TCP/IP protocol or shared-memory based mechanisms like Inter-VM SHared MEMory (IVSHMEM or Nahanni)[4].
In this report the focus is on the shared-memory mechanism IVSHMEM.

1.1 Project Context

"Exploring IVSHMEM in the Jailhouse Hypervisor" is a project proposed by CISTER, a research laboratory dedicated to the research of real-time and embedded systems, and is integrated in the curricular unit of PESTI (Projeto Estágio), lectured in Instituto Superior de Engenharia do Porto (ISEP). The subject itself of hypervisors is new to me, so it is a complete challenge

[3]RTOS is a type of operating system meant to run applications that need to respond to external events in a timely fashion [53] [9].

[4]In this report it will be used IVSHMEM as a reference to this mechanism.

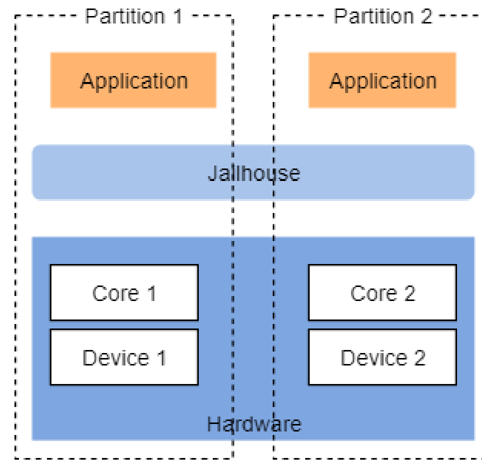

Figure 1.1: Static partitioning example.

to learn something new and popular as it is now and enter in the community with my implementation of it is very satisfying. My main goal by choosing this internship was to diversify my knowledge at a more low-level, so that I would be more receptive to more types of projects.
Alongside with the demands of this job, there are other aspects that I considered, like the type of environment that a research center would have, so that I would be thoughtful about having that experience again in the future.

1.1.1 Problem Description

As mentioned in section 1, Jailhouse is a static partitioning hypervisor that splits the machine's physical resources in divisions (or "cells") in order to assign them to different applications (or "inmates"). As also mentioned, this hypervisor guarantees isolation between the partitions which enables each application to run independently and without any interference from other cells. Yet there are cases where there is the need of exchanging information between different OSes. Thus, in these cases it is imperative to support some kind of communication channel so that partitions can send/receive information without breaking isolation.
Two types of communication mechanisms [23] can be used for this purpose:

networking communication via Transmission Control Protocol/Internet Protocol (TCP/IP), that uses the TCP/IP protocol to send data between nodes or by using shared memory between partitions (example of a protocol that leverages this feature is IVSHMEM).

TCP/IP is a protocol more suitable for those cases where the virtual machines (VMs) reside in separate physical machines, in other words, it was meant to provide a communication channel for multiple physical machines when they need to connect among themselves [41]. However, using communication via TCP/IP is more time consuming considering the data to be transferred goes through the protocol stack [5] [41].

Shared memory is more suitable to those cases where VMs reside in the same physical machine. Thus, it is more suitable to be used in the context of Jailhouse since Jailhouse creates VMs on the same physical machine. Using a shared memory protocol allows one to reduce the number of operations needed to exchange data and the changes are directly visible [41].

Having the above in mind, this project has as main goal the establishment of a shared-memory communication channel using IVSHMEM to enable partitions to exchange information without breaking the partition's isolation.

1.1.2 Approach

Currently, there is already an implementation of IVSHMEM for the Jailhouse hypervisor in x86-64 architecture, provided by Henning Schild and Jan Kiszka.[6].

Since the available demonstrations regarding x86_64 architecture of the functionality of Jailhouse are implemented with the support of Quick Emu-

[5] TCP/IP represents a group of protocol layers. Each layer supports different protocols for managing the data across the network [21].

[6] There are two repositories for Jailhouse https://github.com/siemens/Jailhouse, containing the source code, and https://github.com/siemens/Jailhouse-images, containing ready-to-use virtual machine with Jailhouse and demonstrations for several architecture types.

lator (QEMU) [7], the starting point is to learn and understand Jailhouse by following the indications provided by the Jailhouse repository [24] to set up the hypervisor. Then, the hypervisor is executed on an Intel based x86-64 architecture machine, by using QEMU, and a BananaPi-M1 (BPI) board with an ARM processor [6].

After performing the above tasks, a detailed study of the IVSHMEM shared-memory mechanism and the architecture itself (ARM) is done in order to use this mechanism along side with Jailhouse and create a use case where two partitions exchange information between them. This is expected to perform as the already existing implementation for x86_64 architecture.

1.1.3 Objectives and Contributions

This project aims to: (1) run the Jailhouse hypervisor in x86-64 architecture provided by QEMU emulator, including running already-to-use demonstrations and develop knowledge in this matter and install it in Banana Pi-M1 and run previous demonstrations as well; (2) use the shared-memory mechanism IVSHMEM to function along side with Jailhouse on Banana Pi-M1 to exchange information between two partitions.

1.1.4 Organization

CISTER is an internal research unit of ISEP that specializes in the fields of real-time cyber-physical systems and multiprocessor systems, being rewarded with the highest classification by the Science and Technology Foundation.

Currently, CISTER has international reputation built upon an extensive history of paper publications, international conferences, seminal research works and there's about 60 researchers directly involved with the centre, half of them with a PhD, reaching more than 20 nationalities [3].

[7] Open-source Emulator or virtualizer which enables the creation of virtual machines [33]

1.2 Report Structure

This report is structured in the following manner:
The current chapter, **chapter 1**, is an introductory segment that characterizes the problem and the proposed approach and describes the internship itself, including the work division and meetings.
In **chapter 2**, it will be presented the state-of-art of some hypervisors that were considered as possible solutions for the problem at hand (KVM, Xen and Jailhouse) and inter-VM communication mechanisms (TCP/IP protocol and IVSHMEM).
In **chapter 3**, it will be discussed the requirements, functionality, the device driver and one of the demonstrations available for Jailhouse. Ultimately in **chapter 4**, the IVSHMEM will be explained and ran in a x86-64 architecture via QEMU and will be discussed its implementation on a ARM processor board.
Chapter 5 is the conclusions chapter.

1.3 Work Planning

The work can be divided in five different milestones: setup (internship integration and study of the requirements), survey (study of the technologies), implementation (running previous related works and implement some modifications and the specific use case), testing (performance measurements and efficiency tests) and report, as suggested by figure 1.2.

1.3.1 Meetings

There were weekly non-formal meetings and monthly meetings in which the students discussed their work during that week or month. After the discussion of the tasks, it would be defined new tasks for the future.
The table 1.3 summarizes the topics of the meetings.

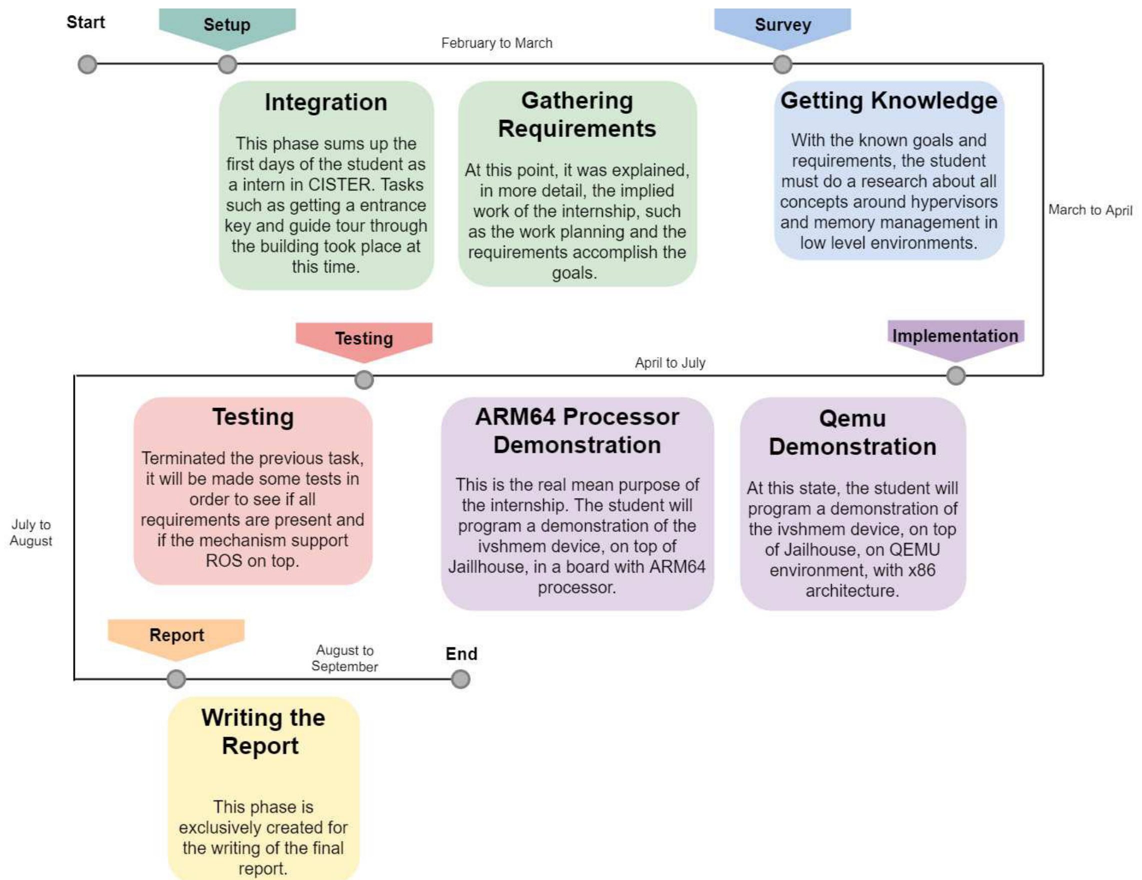

Figure 1.2: Tasks and milestones by chronologic order.

Day and Time	Members	Topics	Location
15 of March of 2019 at 10 to 11.30 a.m	• Diana Ramos; • Pedro Chaves; • Johann Knorr; • David Pereira; • Cláudio Maia.	• PESTI's tasks of each student; • PESTI's next steps; • Other matters.	CISTER
1 of April of 2019 at 13:30 to 14:40 a.m	• Diana Ramos; • Pedro Chaves; • Johann Knorr; • David Pereira; • Cláudio Maia.	• PESTI's tasks of each student; • PESTI's next steps; • Other matters.	CISTER
15 of April of 2019 at 10 to 11.30 a.m	• Diana Ramos; • Pedro Chaves; • Johann Knorr; • David Pereira; • Cláudio Maia.	• PESTI's tasks of each student; • PESTI's next steps; • Other matters.	CISTER

Figure 1.3: Meetings table.

2. State-of-Art

In this chapter it is presented some of the current most popular hypervisors, namely KVM, Xen and Jailhouse, requirements and basic functionalities (section 2.1). In addiction, it will be presented some Inter-VMs Communication mechanisms that use networking like TCP/IP protocol and a shared memory protocol named IVSHMEM (section 2.2).

2.1 Hypervisors

As mentioned in chapter 1, a hypervisor is a virtual platform (similar to an OS [32]) that enables the creation of virtual machines (VM) in the same physical machine. The main purpose is to control VMs access to the machine resources [30] [10]. A hypervisor creates a virtual layer between the physical resources and the VMs (as also denoted "guests") so that each guest has a share of virtual hardware [32].
Currently there are multiple hypervisors, thus this section will discuss the main hypervisors (as well as Jailhouse): Xen and KVM.

2.1.1 Hypervisor's Classification

Hypervisors can be divided in two types: type 1 and type 2 hypervisor. By definition, a type 1 is a bare-metal hypervisor that operates between hardware and the OS and a type 2 is considered as a hosting hypervisor, running above the OS [13].
Virtualization is a specific characteristic of the hypervisor technology and can be classified within two categories: (1) para-virtualized that uses a hypervisor as an abstraction layer between an OS and the hardware and which will imply modifications to the OS in order to communicate with the hypervisor and (2) full-virtualized which simulates the hardware from a machine and the guest OS does not need any modification [37].

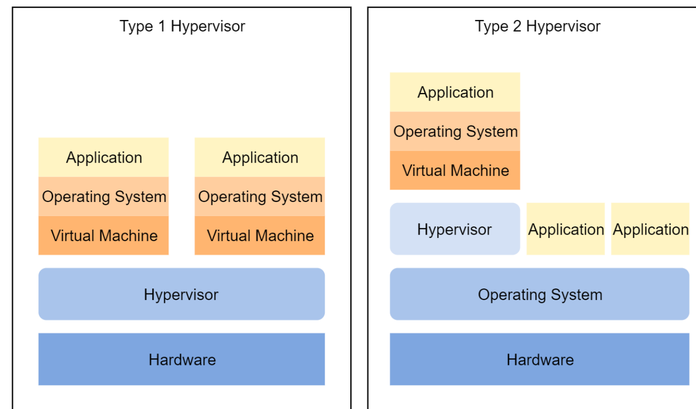

Figure 2.1: Hypervisor's type 1 and 2 designs.

2.1.2 The KVM Hypervisor

KVM, or Kernel-based Virtual Machine, is a virtualization technology, that transforms the Linux kernel into a hypervisor. The main advantage of this hypervisor is the ability to function alongside with the kernel and use kernel services, such as hard-disk write operations. It allows the hosting machine to launch multiple isolated virtual machines, by opening a device node named /dev/kvm [29], treating them as common Linux processes, as suggested by figure 2.2. Each guest has its own memory, separate from the userspace process that created it [29].

To use KVM one needs to enable virtualization in the target machine, requiring Intel processor with the Intel Virtualization Technology (VT) and the Intel 64 extensions or an Advance Micro Devices (AMD) processor with the AMD-V and the AMD64 extensions [7]. Then, two modules need to be loaded into the kernel: a host module and a processor-specific module; and an emulator. KVM takes advantage from being already part of the Linux kernel, because the kernel already has all components required to run VMs, such as memory manager or process scheduler [54]. All VMs share their resources and it is needed synchronization mechanisms in order to control

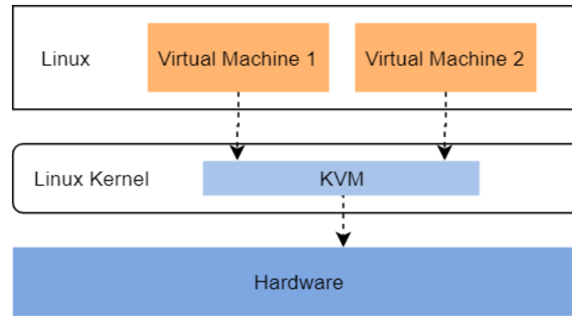

Figure 2.2: Basic architecture of the KVM Hypervisor.

resource's accesses.

KVM is not a traditional type 1 hypervisor since it transforms the Linux kernel into a hypervisor and it may seem that it provides system-level services (such as memory management), where is actually the kernel itself that provides them. It can be classified as a full-virtualized technology that can run multiple virtual machines executing unmodified Linux or Windows images [28].

2.1.3 The Xen Hypervisor

Xen is an open-source bare-metal hypervisor which enables the creation of multiple virtual machines called "domains". A privileged domain is created during boot, called dom0, and is responsible for creating another unprivileged domain, called domU. Once dom0 is shut down, the remaining domains will follow; however the opposite does not occur.

As suggested by figure 2.3, the dom0 is the only virtual machine that acts as an intermediate layer between Xen and the domU, being the only domain to interact directly with the hypervisor.

Both domains, privileged and unprivileged, are connected via para-virtualized drivers, called backend and frontend drivers, correspondingly, which gives dom0 a virtual device, visible to all other domains that it manages, and gives domU a device driver which will be used to establish the connection

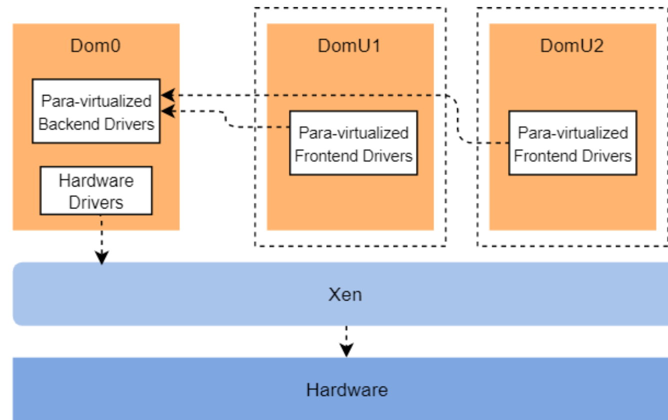

Figure 2.3: Basic architecture of the Xen Hypervisor.

with dom0. The communication is established via shared memory.
Xen represents a more concise type 1 hypervisor, where the privileged domain is created during boot. One particularity of this hypervisor is that it can support both para-virtualization and full-virtualization [55].

2.1.4 The Jailhouse Hypervisor

Jailhouse is a static partitioning hypervisor that runs with Linux, started by Jan Kiszka, a lead developer at Siemens, AG. In 2013, Siemens, AG. decided to open source it [5]. It manages partitions (called "cells"), where each one of them has a share of the computer resources, such as CPU, memory and PCI devices so, instead of having multiple guests accessing symmetrically to the resources and having no boundaries between them, Jailhouse divides the resources and assigns them to each partition, accordingly with their definitions. For instance, this hypervisor is meant to provide isolation between all partitions, such that there is no resource sharing and flexibility to run whatever software (called "inmates") they desire.
In order to launch the hypervisor, Linux is required to be installed on the system, because it is in charge of bootstrapping Jailhouse. After initialization, it is created an initial cell called the "root cell" and the hypervisor layer goes under the Linux layer. Originally, the root cell has to its possession

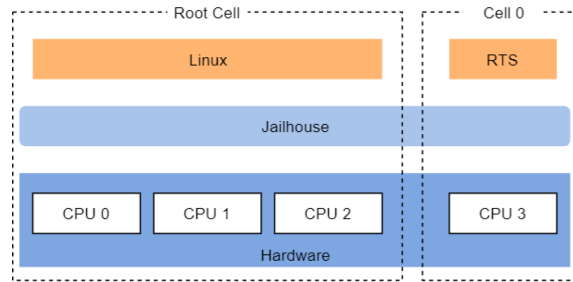

Figure 2.4: Basic architecture of a system using the Jailhouse Hypervisor.

all resources, but as more cells are created, defined resources are taken and assigned to the new cells. As demonstrated in figure 2.4, there are two cells: one root cell that runs Linux that has to its possession three CPUs and another non-root cell that run a RTOS with one CPU. In this case, if another cell were to be created with two defined CPUs, these resources would be taken from the root cell and assigned to the new cell, resulting in the root cell having 1 CPU, cell 0 having 1 CPU and the newer cell having two CPUs assigned.

Jailhouse does not fit in a standard hypervisor classification. It could be classified as a type 2 hypervisor, since it initially needs to be bootstrapped by Linux, but then, after initialization, the hypervisor goes under Linux and handles all operations, fitting in the type 1 description.

2.2 Inter-VM Communication

Although one crucial requirement of this project is to necessarily implement IVSHMEM (shared-memory) as the mechanism of communication, there are some other viable options to take into account, as for instance a networking-based mechanisms such as TCP/IP protocol.

2.2.1 TCP/IP-based Communication

TCP/IP is a networking communication protocol commonly used for nodes in different physical machines and can as well be used as a communication

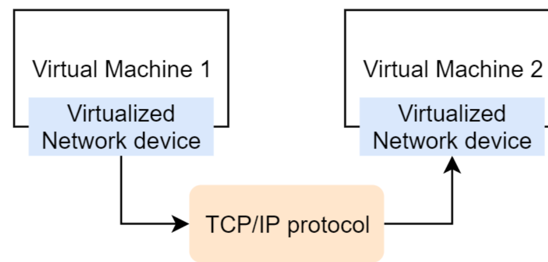

Figure 2.5: Basic architecture of TCP/IP protocol.

channel between isolated virtual machines inside the same physical machines. The data is segmented into smaller chunks (or segments), wrapped into a packet and then delivered to a certain network node. It uses two different protocols: TCP (Transmission Control Protocol), which processes and receives data from sources; and IP (Internet Protocol) that provides addresses to each node of the network.

Figure 2.5 is a representation of how two virtual machines communicate using TCP/IP via a virtualized network device.

2.2.2 Shared-Memory Communication

A traditional shared memory mechanism is translated into a specific region of memory that is mapped into several process's address space and configured by the OS resulting in a common area, visible by all of these processes. This methodology is considered to be the fastest and efficient inter-process communication mechanism [50], as suggested by figure 2.6 [8].

The main perk of using shared memory is to enable all processes to read and write directly from the memory region, where by using other methods, like TCP/IP based mechanisms, in general, there's a significant increase of overhead due to the fact that the same message has to go through multiple protocol layers.

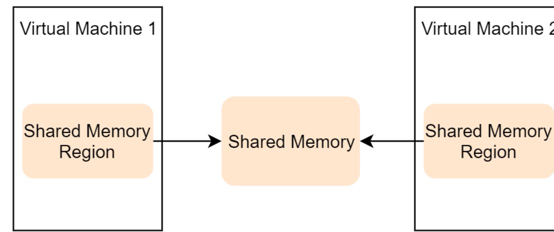

Figure 2.6: Basic architecture of shared memory communication protocol.

IVSHMEM

IVSHMEM (Inter-VM SHared MEMory), is an emulated PCI device that provides communication between co-located virtual machines (same host) by sharing memory establishing a connection between Portable Operating System Interface (POSIX[1]) shared memory on the Host and the application running inside a VM.

IVSHMEM's shared memory is visible to user-level code, which greatly simplifies the porting of existing libraries and supports pointer-based mechanisms [32] and its flexibility is proved by letting exist different memory regions between virtual machines, enforcing whatever policy is desired.

The virtual machines share the POSIX shared memory via series of mapping operations at the user level, with support for inter-VM interrupts via the Linux event file descriptor mechanism [32].

[1]POSIX is a standard interface based on UNIX that facilitates software programs to be run in different machine's operating systems [38].

3. Jailhouse Hypervisor

This chapter is related to Jailhouse and gives a general view about the project and setup of the Jailhouse kernel module, including its hardware and software requirements. This hypervisor's functionality is explained in more technical detail, incorporating some examples implemented in an emulator called QEMU (using x86-64 architecture) and Banana Pi-M1 (ARM architecture).

3.1 Jailhouse Overview

As mentioned in section 1.1.1, Jailhouse is a hypervisor that follows the static hardware partitioning ideology that is based on the fact that partitions have their own share of the physical resources of the hardware available in the system and they are not shared.
Considering that this technology provides isolation between partitions, the system is prepared to safely run bare-metal partition without compromising any of their performances, which is why Jailhouse focuses primarily in giving exclusive access to the resources for each partition.
As suggested by figure 3.1, Linux is a fundamental piece for Jailhouse, in a way that the hypervisor is activated within its environment, taking over the hardware resources and moving under the OS, becoming the root cell. When Jailhouse creates a new cell (non-root cell), Jailhouse takes back

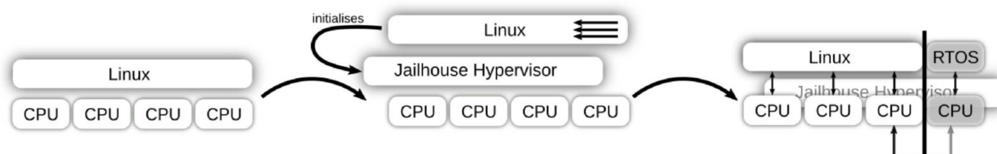

Figure 3.1: Jailhouse bootstrap process taken from [40].

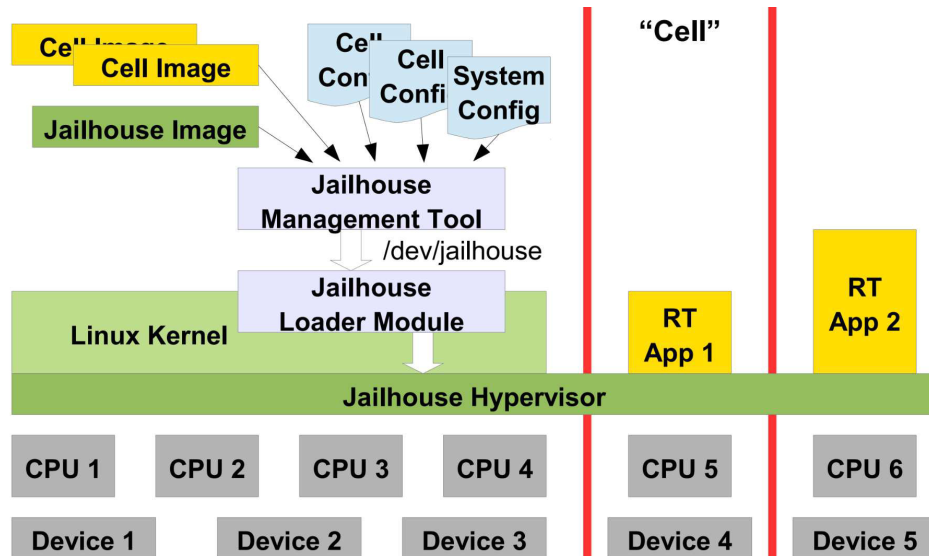

Figure 3.2: Jailhouse's system architecture taken from [14].

some resources allocated for the current root cell and assigns them to the new partition. These resources are explicitly defined in the configuration file for the new cell and includes a minimum of one CPU and a part of the memory from the root cell [40].

From all OSes, it is common to ask why Linux is the most suitable to perform this task. This OS is specially appropriate to Jailhouse's specification regarding hardware support [40], considering that it is one of the OSes that can be ran almost in every machine, which is very convenient and gives a major advantage to any developer who aims to explore the Jailhouse technology. Also, when directly assigning the devices, Jailhouse does not need device drivers, like other partitioning technologies (like Quest-V[1]), becoming lighter, system wise [40].

Figure 3.2 is a simple representation of Jailhouse's architecture, with two real-time applications, each one on their respective cell. The hypervisor

[1]Quest-V is a static partitioning hypervisor that separates kernel into sandboxes, which are a security method for isolating different programs, and each one has a part of memory, I/O and CPU resources [57]

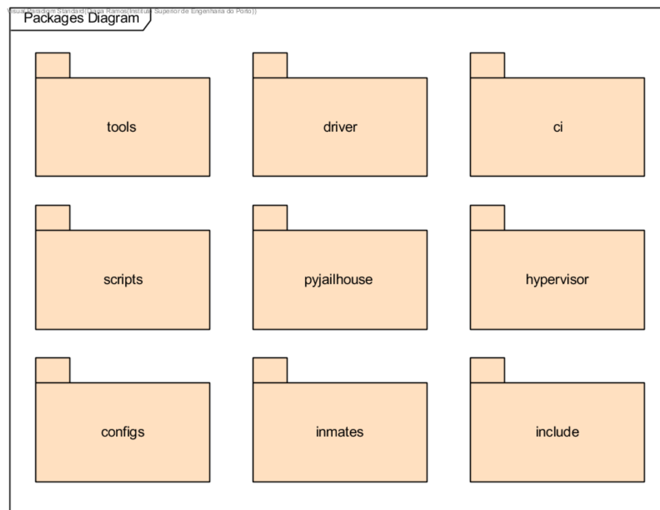

Figure 3.3: Jailhouse's package diagram.

works at the lowest-level possible with the support of a loadable kernel module (see section 4.1.2) which is in charge of managing the hypercalls[2] via a management tool.

3.1.1 Code Organization

According to figure 3.3, Jailhouse's code repository is divided by 9 packages.

1. The ci, or continuous integration build environment package, is meant to help and adapt the current system's environment in order to run Jailhouse. Since every system has its own configuration, memory regions and hardware components, a static project like this hypervisor's can not demonstrate consistency in these terms, therefore some handy scripts to help configuration tasks were created. The configuration of the Linux kernel must allow some specific functionalities enabled to run and use Jailhouse smoothly. If the configuration file

[2] A hypercall is a way to a user space application notify the OS that it needs to execute a command with higher priority [46]

(.config) in the kernel's source code is not synchronized with it, fixing the upcoming compilation/run time problems may be time consuming.

2. The pyjailhouse package is essentially a python module that extends Jailhouse's functionality [39].

3. The include package contains headers files.

4. The hypervisor package contains firmware specifics.

5. The scripts package contains version control scripts.

6. The driver package is related to the Jailhouse's loadable kernel module.

7. The tools package contain the functionality scripts for all operations under Jailhouse. Its usage will be demonstrated in the next subsection.

8. The config package, as the name suggests, is where all configuration files for the cells are stored. Since the hypervisor is supported in 3 different architectures (arm, arm64 and x86), each one has its own compatible file so that the cells (root and non-root) are created within the boundaries of memory.

9. The inmates package contains the software demonstrations that are meant to run within its equivalent cell. Each demo has different outputs.

3.1.2 Jailhouse Requirements

In order to setup Jailhouse, two main components are required: Virtualization feature for I/O and virtualization extensions pack for the processor. This section is related to this hypervisor's requirements and their importance in the context of Jailhouse.

Intel Virtualization Technology, or Intel VT, is a general feature required to enable virtualization specifically for Intel-based systems [20]. It specializes in three different areas which are: CPU (focusing in virtual machines with native run time), memory (regarding direct memory access (DMA) remapping and extended page tables (EPT)) and I/O device support (related to the peripheral system, featuring Intel Virtualization Technology for Directed I/O (denoted as VT-d)) [20].

By using VT-d capabilities, the hypervisor is able to more efficiently manage the I/O inputs and outputs, physical memory and interrupt system [19], by allowing direct assignment to I/O ports. In other words, if a certain device is trying to access some block of memory which is not allowed to, then it will be blocked by the VT-d hardware which will contribute for the resource isolation featured by the Jailhouse hypervisor.

Virtual Machine Extensions (VMX or VT-x) and the equivalent for AMD systems, Secure Virtual Machine (SVM), are defined as extension packs for the processor in order to enable virtualization [2] and includes the following:

1. EPT or Nested Page Tables (NPT): translates the physical addresses into guest physical addresses, granting the guest full control of its own page tables [1].

2. Unrestricted guest mode: determines if the guest will run in unpaged protected mode or in real address mode and will use EPT mechanism.

3. Preemption timer: mechanism that controls the preemption[3] of the guest VM's execution [2].

[3] Preemption is switching a task for another with higher execution priority.

x86-64 Architecture			
Intel System		AMD System	
Intel Virtualization Technology		------	
Virtual Machine Extensions	Extended Page Tables	Secure Virtual Machines	Nested Page Tables
	Unrestricted Guest Mode		Decode Assists
	Preemption Timer		
Intel Virtualization Technology for Directed I/O		AMD IOMMU	
At least 2 logical CPUs			

ARM Architecture	
ARM Boards	ARM64 Boards
Banana Pi	AMD Seattle / SoftIron Overdrive 3000
Orange Pi Zero	LeMaker HiKey
NVIDIA Jetson TK1	NVIDIA Jetson TX1 and TX2
ARM Versatile Express with Cortex-A15 or A7 cores	Xilinx ZCU102
emtrion emCON-RZ/G1x series based on Renesas RZ/G	NXP MCIMX8M-EVK
At least 2 logical CPUs	
ARMv7 with virtualization extensions or ARMv8	

Figure 3.4: Jailhouse's hardware requirements for both architectures.

Jailhouse has also support for ARM and ARM64 architectures and figure 3.4 lists all boards that are currently available. All boards need at least 2 logical CPUs to run a root and non-root cell as well as for x86-64 architecture.

Jailhouse also has some specific software requirements. The hypervisor needs some pre-located RAM, in order to boot and re-allocated it to future cells [24]. For x86-64 architectures, I/O Memory Management Unit (IOMMU) has to be disabled as well and the minimum Linux kernel version is 3.14. For ARM and ARM64 architectures, the Linux kernel version must be at least 3.19 and 4.7 and Linux must start in hypervisor mode and has to support Power State Coordination Interface (PSCI).

x86-64 Architecture
Direct Memory Access Remapping disabled
Pre-located RAM
Linux Kernel 3.14+

ARM Architecture	
ARM Boards	ARM64 Boards
Linux Kernel 3.19+	Linux Kernel 4.7+
Linux starts in hypervisor mode	
Support for Power State Coordination Interface	
Pre-located RAM	

Figure 3.5: Jailhouse's software requirements for both architectures.

3.1.3 Linux Kernel Modules

The Linux kernel is an aggregation of pieces of executable code that are responsible for manipulating certain parts of the hardware. It can be split in 6 parts corresponding to the following [27]:

1. **Process management:** responsible for taking into account the creation, manipulation and destruction of the Linux processes and the scheduler.

2. **Memory management:** in charge of managing the whole memory system, including controlling the resource utilization of all software components.

3. **Filesystems:** this area is related to the multiple filesystems that Linux supports and how the data is organized.

4. **Device control:** this part refers to how hardware devices interact with the system. Traditionally, each device is mapped into the memory and is uniquely controlled by code, denoted as device driver. Since a device is at hardware level and an OS executes on top it, without a device driver to monitor the hardware, the communication between

device and OS would be non-existent and the operations sent to the device would not execute properly.

5. **Networking:** this area handles the networking aspects, for instance if a system receives a packet, it must be identified and redirected to a Linux process do handle it.

Kernel Modules

Loadable modules are out-of-tree kernel modules that are developed with the intent of extending the kernel's features and exploring their potential. Modules can be loaded and unloaded at any point during run time. After compiling a module against the desired kernel, the module results in an object file which is linked to the kernel after loading it [27].

To compile an out-of-tree module, usually a main-line kernel is more suitable to compile against since is free from heavy patches provided by developers [27]. The following commands will compile and install the module:

```
make KDIR=/path/to/source/kernel
make install
```

It is worth mention that if the compilation has not a explicit location of the desired Linux kernel, by default, the kernel module will be compiled with the already running kernel.

With an already compiled module, the next step is to load it into the kernel using one of the following:

```
modprobe "module"
insmod "module".ko
```

When the kernel module is loaded, that specific piece of code is inserted into the already existing kernel's code, translating it to symbol table of the kernel, enabling other lower-level functionalities. Both "modprobe" and "insmod" are the same functions regarding loading a module, but used in different matters. The first one will search in the desired kernel directory and

```
Module                  Size  Used by
uio_ivshmem            16384  0
jailhouse              32768  0
sun4i_codec            49152  3
axp20x_pek             16384  0
snd_soc_core          155648  1 sun4i_codec
snd_pcm_dmaengine      16384  1 snd_soc_core
sun4i_backend          20480  0
snd_pcm                90112  3 sun4i_codec,snd_pcm_dmaengine,snd_soc_core
snd_timer              32768  1 snd_pcm
snd                    61440  3 snd_timer,snd_soc_core,snd_pcm
soundcore              16384  1 snd
sun4i_drm_hdmi         20480  0
cec                    40960  1 sun4i_drm_hdmi
sunxi                  20480  0
musb_hdrc              98304  1 sunxi
phy_generic            16384  1 sunxi
sun4i_drm              20480  1
sun4i_frontend         16384  2 sun4i_drm,sun4i_backend
sun4i_tcon             32768  1 sun4i_drm
sun8i_tcon_top         16384  2 sun4i_drm,sun4i_tcon
uio_pdrv_genirq        16384  0
uio                    20480  2 uio_ivshmem,uio_pdrv_genirq
```

Figure 3.6: Loaded modules.

try to find the module and load all its dependencies, whereas the second one works with static paths and will not load any dependencies and so it can result in an "unresolved symbols" insertion error [27]. After succeeding, running "lsmod" command will list all inserted modules. Figure 3.6 represents the output of lsmod, which lists all inserted modules by designation, size in bytes and how many instances of the module are being used and by which component of the system [31]. One way to remove the module is running "rmmod" command which will fail in cases while the module is still in use.

It will be created a device regarding the module that can be accessed within the /dev filesystem. Figure 3.7 is an example of how a device and an application are connected. User level applications will open the /dev filesystem which will give access to a kernel module and, consequently, control the device.

3.1.4 Jailhouse's Kernel Module

As mentioned in section 3.1, Jailhouse works with a loadable kernel module to issue hypercalls via a management tool denoted as jailhouse.ko.

Following section 3.1.3 command scheme, the kernel module is compiled

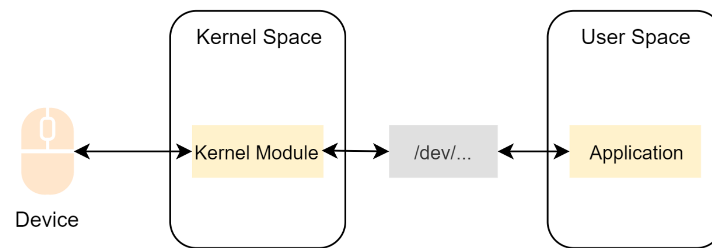

Figure 3.7: Kernel module transactions adapted from [15].

```
modules_install: modules
	$(Q)$(MAKE) $(kbuild)

firmware_install: $(DESTDIR)$(firmwaredir) modules
	$(INSTALL_DATA) hypervisor/jailhouse*.bin $<

tool_inmates_install: $(DESTDIR)$(libexecdir)/jailhouse
	$(INSTALL_DATA) inmates/tools/$(ARCH)/*.bin $<

install: modules_install firmware_install tool_inmates_install
	$(Q)$(MAKE) -C tools $@ src=.
```

Figure 3.8: Jailhouse installation Makefile.

with the above command:

```
make KDIR=/path/to/source/kernel
```

The next step should be the installation of Jailhouse on target machine, by running the command below, which follows the defined rules of figure 3.8. All modules, firmware and tools will be installed.

```
make install
```

And the kernel module can be finally loaded into the kernel running one of the following commands:

```
modprobe jailhouse
```

```
insmod jailhouse.ko
```

```
59  static void __attribute__((noreturn)) help(char *prog, int exit_status)
60  {
61          const struct extension *ext;
62
63          printf("Usage: %s { COMMAND | --help | --version }\n"
64                 "\nAvailable commands:\n"
65                 "   enable SYSCONFIG\n"
66                 "   disable\n"
67                 "   console [-f | --follow]\n"
68                 "   cell create CELLCONFIG\n"
69                 "   cell list\n"
70                 "   cell load { ID | [--name] NAME } "
71                                  "{ IMAGE | { -s | --string } \"STRING\" }\n"
72                 "             [-a | --address ADDRESS] ...\n"
73                 "   cell start { ID | [--name] NAME }\n"
74                 "   cell shutdown { ID | [--name] NAME }\n"
75                 "   cell destroy { ID | [--name] NAME }\n",
76                 basename(prog));
77          for (ext = extensions; ext->cmd; ext++)
78                  printf("   %s %s %s\n", ext->cmd, ext->subcmd, ext->help);
79
80          exit(exit_status);
81  }
```

Figure 3.9: Jailhouse's command line tool options.

3.1.5 Jailhouse Functionality

Figure 3.9 represents a piece of code from /tools/Jailhouse.c that prints a menu of available commands.

With the Jailhouse's kernel module already inserted (see section 3.1.2), the first thing to do is to enable the hypervisor by following the line 65 from figure 3.9:

```
jailhouse enable /path/to/system/configuration
```

The "enable" command from Jailhouse's management tool launches the hypervisor, passing multiple validations and creates the root cell, initializing all assigned CPUs and subsequently run it under Linux. Since the root cell will now be running with all of the resources as a whole, the next step is to create a cell configuration. Programs to be ran under a cell's environment can only use the resources that the cell makes accessible for them, nevertheless a cell configuration is needed to add and define resource's boundaries, including adding PCI devices.

The code below is an excerpt of a Jailhouse configuration file where the allocation of memory for RAM is described. It is defined a physical and virtual start addresses, the size and jailhouse's flags regarding read and write permissions. For more information of how to write a cell configuration file, consult sections B.1 and B.2 of the attachments.

```
/* RAM */ {
        .phys_start = 0x7bef0000,
        .virt_start = 0,
        .size = 0x00010000,
        .flags = Jailhouse_MEM_READ | Jailhouse_MEM_WRITE |
                 Jailhouse_MEM_EXECUTE |
                 Jailhouse_MEM_LOADABLE,
}
```

One of the following command can be used (line 65 from figure 3.9):

```
jailhouse cell create /path/to/demonstration/cell/config
jailhouse config create my_config.c
```

The "cell create" is often used for pre-created cell configuration's. The "config create" is a cell configuration generator which is intended to generate files for both root and non-root cells. This command is highly useful when implementing Jailhouse in other potential environments, where the architecture may differ from what the hypervisor is used to. Since the assignment is based on memory ranges, configuration file depends on the hardware resources and memory boundaries. Next is loading the inmate inside the cell by following the command (line 70 from figure 3.9):

```
jailhouse cell load "demo_name" /path/to/demonstration/cell/inmate
```

This command line option usually requires that the inmate is specially created for the cell that is loaded into, because the cell configuration includes all hardware resources and sufficient memory allocation for the program to

run smoothly otherwise it may occur run time fails like "unhandled traps" (see section 3.1.5) or memory access's denials (see section 3.1.5). The next step is to finally launch the cell by following this command (line 73 from figure 3.9):

```
jailhouse cell start "demo_name"
```

The "cell start" splits the memory regions defined above of the root cell and assign them to the newly-created non-root cell, which will give it run-time isolation, that is, since the root cell no longer has control over the resources that the guest has allocated, then it no longer has permission to access them.

In order to stop or shutdown a cell, the command line option must be followed as (line 74 from figure 3.9):

```
jailhouse cell shutdown "demo_name"
```

And to destroy the cell (line 75 from figure 3.9):

```
jailhouse cell destroy "demo_name"
```

Certain cells have the ability to decline a shutdown request (see section 3.1.6). Although destroying a cell does not necessarily implies it, when those cells are running, the procedure to destroy them involves shutting down first then destroying it.

There are some control command line options to be used to check the state of the cells, i.e. the "cell list" option where it gives a list of the created cells, containing its identification (both numerical and by name), the loaded inmate (if it exists), the state and failed CPUs. The syntax is the following (line 69 from figure 3.9):

```
jailhouse cell list
```

```
88  /* cell state, initialized by hypervisor, updated by cell */
89  #define JAILHOUSE_CELL_RUNNING              0
90  #define JAILHOUSE_CELL_RUNNING_LOCKED       1
91  #define JAILHOUSE_CELL_SHUT_DOWN            2 /* terminal state */
92  #define JAILHOUSE_CELL_FAILED               3 /* terminal state */
93  #define JAILHOUSE_CELL_FAILED_COMM_REV      4 /* terminal state */
```

Figure 3.10: Jailhouse's hypercall cell states.

3.1.6 Jailhouse's Cell States

Accordingly to Jailhouse's hypercalls, defined in /include/jailhouse/hypercall header file (figure 3.10), there are 5 different cell states. Figure 3.11 is a state machine diagram that represents all the possible states and their outcomes for a non-root cell, that follows sequentially all the command tool options listed above.

When creating a cell configuration, the state is set to "shutdown", since there is already an empty cell, despite not being running. After loading and starting the cell, the state is changed to "running", considering the inmate is running and producing outputs. "Running/Locked" is an inevitable case which occurs when a cell is still running but locked its configuration so that no other cell (root and non-root) can modify it or when the hypervisor attempts to shutdown the cell. This last reason is an exception for cells that implement a voting system between all cells for certain operations such as shutting down, that is, if the hypervisor requests shutting down a cell, instead of having full control, other existing cells can vote and eventually decline it. Thus, a second try should succeed. "Failed" state is common where there are some run time fails, such as unhandled traps or parking errors, which leads to CPU failing (see section 3.1.6). For this particular state, there are two alternatives: restarting the cell, setting the state back to "running" or destroying it.

3.1.7 Run Time Failing Errors

As most technologies, Jailhouse is prepared to handle run time errors. According to figure 3.11, a certain cell can be set to a "failed" state, which

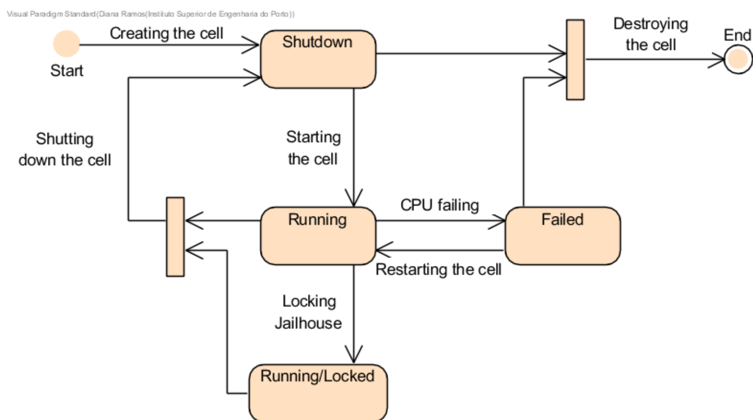

Figure 3.11: Jailhouse cell states.

normally can be lead by CPU suspension. These run-time errors are usually handled by macros called "traps". The hypervisor's trap occurs when it is trying to unsuccessfully map regions of memory. As mentioned above,the configuration files define boundaries of memory, not in a dynamic but hard-coded way. Those files are vulnerable for misconfiguration, considering that they need to be aligned with the memory for each physical device that it mentions.

There are two types of traps that are defined: unhandled and forbidden. The hypervisor passes through a bunch of validations. Figure 3.12 verifies the memory-mapped for the I/O (MMIO) accesses. In case where it results in either an MMIO_ERROR or MMIO_UNHANDLED values, the access is invalid. When trying to perform some sort of operation in memory regions that the hypervisor has not permissions, a forbidden trap appears. Accesses with appropriate permissions but not within the memory ranges will be caught as an unhandled trap. When operating on a cell with those conditions, it appears on the console some specific messages, implemented by figures 3.13 and 3.12.

Any trap leads the cell to a "failed" status. Since the CPU fails its initialization, the hypervisor suspends it and it no longer executes any code. The

```
451         case TRAP_FORBIDDEN:
452                 panic_printk("FATAL: %s (exception class 0x%02x)\n",
453                              (ret == TRAP_UNHANDLED ? "unhandled trap" :
454                                                       "forbidden access"),
455                              exception_class);
```

Figure 3.12: Jailhouse unhandled traps message.

```
 98    error_unhandled:
 99            panic_printk("Unhandled data %s at 0x%lx(%d)\n",
100                    (is_write ? "write" : "read"), mmio.address, size);
101
102            return TRAP_UNHANDLED;
```

Figure 3.13: Jailhouse unhandled trap console message.

goal is to keep the CPU waiting until the hypervisor resets its status or destroys the cell. In addition to the triggering event and trap message, a special message (figure 3.15) appears indicating which CPU failed its execution (the one(s) assigned to the triggering cell) - parking. The message is based on "parking" a CPU, i.e., setting aside on execution, on panic, which is an OS safety measure when detecting an internal fatal error [48].

3.2 Jailhouse's Demonstrations

Jailhouse has some demonstration cells and this section is related to a QEMU based demonstration (apic-demo) implemented for Intel system x86-64 based architecture and an ARM architecture based demonstration (gic-demo).

```
82          mmio_result = mmio_handle_access(&mmio);
83          if (mmio_result == MMIO_ERROR)
84                  return TRAP_FORBIDDEN;
85          if (mmio_result == MMIO_UNHANDLED)
86                  goto error_unhandled;
```

Figure 3.14: Jailhouse unhandled traps conditions.

```
1003            panic_printk("Parking CPU %d (Cell: \"%s\")\n", this_cpu_id(),
1004                         cell->config->name);
```

Figure 3.15: Jailhouse parking error.

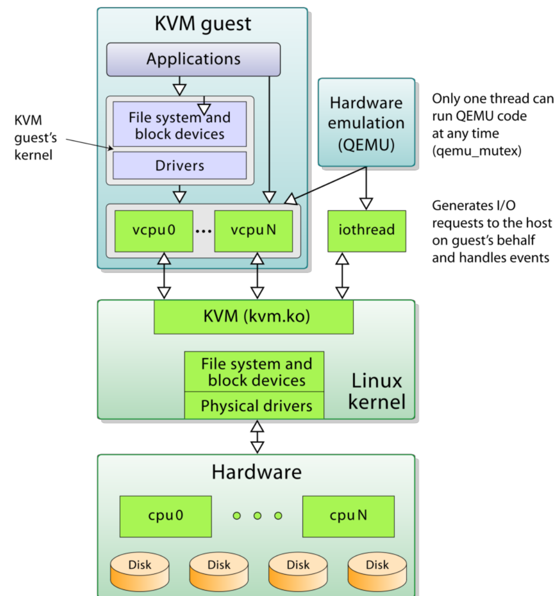

Figure 3.16: QEMU architecture.

3.2.1 QEMU Demonstration

QEMU, or Quick Emulator[4] is an emulation technology that runs virtual machines which can operate with KVM assistance and run under multiple systems such as x86, x86-64 and PowerPC and emulate numerous architectures like x86, x86-64, ARM, Scalable Processor ARChitecture (SPARC), PowerPC and Microprocessor without Interlocked Pipeline Stages (MIPS) [45].

KVM is a kernel module that is loaded into the Linux kernel in order to transform it into a hypervisor and is an important piece in QEMU. With hardware capable of using virtualization, i.e. VT-d or AMD-v, QEMU can

[4] An emulator is generally a program or device that enables the creation of a different environment within a machine, by creating its particular hardware components.

create user-space virtual machines that will be taken care of with the current hypervisor's help [18].

Figure 3.16 shows that QEMU emulates hardware and can benefit from supporting KVM, i.e. creates Virtual CPUS (VCPUs) in order to execute the intended code on the guests side. Guests are created as KVM guests that are intended to run software and communicate with VCPUs which communicates with the KVM interface inserted in the actual kernel. This emulator also sends I/O requests to the kernel module so that events regarding its virtual machines are handled.

Setting up Jailhouse with QEMU

For demonstration purposes and software validation, using an emulator such as QEMU to run Jailhouse is appropriate. Since the QEMU already provides all possible hardware devices that the hypervisor might need, this is a way to run Jailhouse almost in any environment at any time without compromising the machine itself. There are already some demonstration cells ready-to-use in multiple architectures like x86-64, arm64, arm, etc.

The first step is to get QEMU working properly. An updated version of this emulator is recommended and can be extracted by the main QEMU page (https://www.qemu.org/download/source).

The configuration script that QEMU provides, by default, builds it with every available architecture. Therefore a target list must be defined, in this case a x86-64 architecture. The Simple DirectMedia Layer (SDL) option is enabled in order to have a graphical display mode, otherwise if no display is enabled, the emulator will start and hangs at the Virtual Network Computing (VNC) server[5].

```
./configure -target-list=x86-64-softmmu -enable-sdl

make
```

[5]The VNC server is a remote way for a user to access another machine.

```
make install
```

Now that QEMU is installed, there are two paths to follow to run Jailhouse:

1. Setting up and running the hypervisor with the assistance of a side repository.

   By following the first approach, the first step is to clone the "Jailhouse-images" side repository using the git tool:

   ```
   git clone https://github.com/siemens/jailhouse-images.git
   ```

   The host machine must fulfill the following requirements [25]:

   (a) Updated Docker, which will create containers that will allow the demonstrations to run isolated, that is, the applications will not be able to see more than the resources assigned to their container;

   (b) QEMU version must be equal or higher than 2.8 for x86 image and higher or equal than 2.12 for ARM64 image;

   (c) Linux kernel version above or equal to 4.4 with KVM support (for x86 image);

   (d) On Intel, kvm_intel module loaded with parameter nested=1. This parameter will enable nested virtualization and permit that inside a KVM virtual machine it can run a hypervisor [36].

   This repository has two main parts: the build-images and the start-qemu script. In order to QEMU initialize, it needs an image prepared by the first script which allows the build for multiple environments (figure 3.17). In this particular case, the QEMU/KVM Intel-x86 virtual target is selected as the emulator to configure for this type of systems. The second script initializes the emulator with the built image (Linux kernel 4.19.56) and already display a bunch of usable ordered

```
1: QEMU/KVM Intel-x86 virtual target
2: QEMU ARM64 virtual target
3: Orange Pi Zero (256 MB edition)
4: Intel NUC (NUC6CAY, 8 GB RAM)
5: Marvell ESPRESSObin (1 GB edition)
6: Marvell MACCHIATObin
7: LeMaker HiKey (Kirin 620 SoC, 2 GB edition)
8: Avnet Ultra96
0: all (may take hours...)
```

Figure 3.17: QEMU images list.

```
1  jailhouse enable /etc/jailhouse/qemu-x86.cell
2  jailhouse console
3  jailhouse cell create /etc/jailhouse/apic-demo.cell
4  jailhouse cell load apic-demo /usr/libexec/jailhouse/demos/apic-demo.bin
5  jailhouse cell start apic-demo
6  jailhouse cell stats apic-demo
7  jailhouse cell destroy apic-demo
8  jailhouse cell linux /etc/jailhouse/linux-x86-demo.cell /boot/vmlinuz* \
9                      -i /usr/libexec/jailhouse/demos/rootfs.cpio \
10                     -c "console=ttyS0 8250.nr_uarts=1 ip=192.168.19.2"
11 ssh 192.168.19.2
12 jailhouse disable
```

Figure 3.18: Bash history for QEMU's AMD systems.

command line options (located in the history 3.18) to run Jailhouse and work with apic-demo cell and a non-root Linux cell.

```
./build-images
./start-qemu "architecture"
```

2. Starting a virtual machine from scratch, build Jailhouse and run:

   The second approach is more meticulous considering that the virtual machine is created from scratch, incorporating all hardware requirements for the hypervisor, that are concealed by the scripts of the Jailhouse-images repository. The following command will launch the virtual machine with a Linux image:

```
qemu-system-x86\_64 -machine q35,kernel\_irqchip=split
-m 1G -enable-kvm -smp 4
-device intel-iommu,intremap=on,x-buggy-eim=on
-cpu kvm64,-kvm\_pv\_eoi,-kvm\_steal\_time,-kvm\_asyncpf,
```

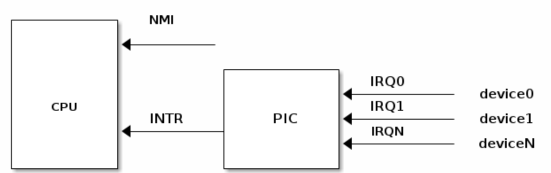

Figure 3.19: PIC architecture taken from [22].

```
-kvmclock,+vmx -drive file=LinuxInstallation.img,format=raw,
id=disk,if=none -device ide-hd,drive=disk -serial stdio
-serial vc -netdev user,id=net
-device e1000e,addr=2.0,netdev=net
-device intel-hda,addr=1b.0 -device hda-duplex
```

By setting up the Jailhouse's driver (see section 3.1) and its functionality (see section 3.1.4), the hypervisor should run correctly.

Advanced Programmable Interrupt Controller Demonstration

Advanced Programmable Interrupt Controller, or APIC, is a superior version of the Programmable Interrupt Controller (PIC), that is an unity used to program that controls interrupt's flow in the machine, which contains entries and exits of requests. From definition, an interrupt is an action triggered by peripherals or even the CPU to a CPU in order to modify software behavior [22]. Accordingly with figure 3.19, devices send interrupt requests (IRQ) to the connected PIC and it forwards them to the CPU [22].

PIC is the logical-base technology, but supposing that the system is multicore, where there are multiple CPUs to be assisted, APIC seems to be more appropriate because each core has its own local APIC (LAPIC), instead of an external connected PIC, and can manage the interrupts coming from their linked devices. This link is not direct, that is, each CPU is connected to the Interrupt Controller Communication BUS and to a I/O APIC which will receive the external events and forward them for their destination

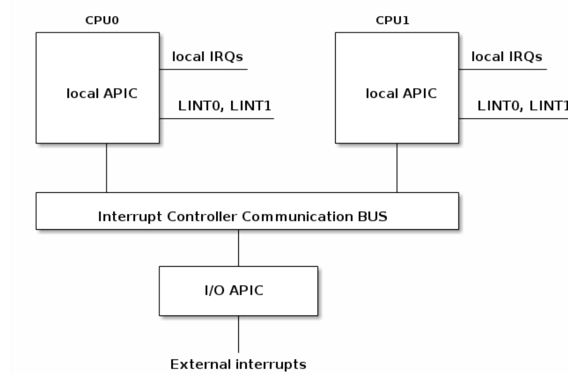

Figure 3.20: APIC architecture taken from [22].

core [22] (3.20).

Jailhouse provides multiple demonstrations specially for x86-64 architecture systems, one of them being the "apic-demo" that is based on APIC interrupt handler mentioned above. This demonstration displays a timer to be triggered at 10Hz and includes a jitter, which is measures the delay between the actual time of an event and the programmed one (event latency). Therefore, a smaller latency contributes to a more accurate hypervisor [44].

As mentioned before, QEMU images have a customized bash history (figure 3.18) which facilitates the understanding and serves as an orientation guide for what commands to follow. To run "apic-demo", the commands are:

```
jailhouse enable /etc/jailhouse/qemu-x86.cell

jailhouse cell create /etc/jailhouse/apic-demo.cell

jailhouse cell load apic-demo /usr/libexec/jailhouse/demos/apic-demo.bin

jailhouse cell start apic-demo

jailhouse cell destroy apic-demo

jailhouse disable
```

BananaPi-M1	
CPU	A20 ARM Cortex -A7 Dual-Core
GPU	ARM Mali400MP2Complies with OpenGL ES 2.0/1.1
Memory	1GB DDR3
Storage	External TF card
Network	10/100/1000 Mbit/s Ethernet
Video Output	HDMI port, multi-channel audio output LVDS
Audio Output	3.5mm jack and HDMI
USB ports	2 x USB 2.0 ports
GPIO	Power (+5V, +3.3V and GND) UART, I2C, SPI or PWM
Power Input	5V/2A via Micro USB
Size&Weight	92x60mm, 48g
Operating System	Android and Linux

Figure 3.21: BananaPi-M1 specification.

3.2.2 Banana Pi Board

Banana Pi-M1 is one of the boards that supports Jailhouse and is the chosen one to assist in this project. The specifications are represented by figure 3.21 [6].

Setting up Jailhouse with Banana Pi-M1 Board

Actually there is already a tutorial about the setup of Jailhouse on this board, although for the sake of this project's requirements, some parts of it were altered. All the full scripts used for this were added to the appendix section in the end of this document.

Since this board is a basic functional computer, its initialization depends on an injectable SD card that contains the OS. Firstly, it will contain the "bananian" SO provided by bananian team, which is a Debian image specifically adapted for Banana Pi-M1.

The next step is to update U-boot configuration file as it follows in order to boot the OS with reserved space for the hypervisor. This file must be a more recent released version that v2015.04 [43] and has two boot options

which is a main line kernel above version 4 and sunxi above 3.4. If the SD card contain only bananian OS, the boot loader will boot with it, unless a Linux kernel is found. This is only a script so if there is a boot error, the commands that are defined in the file can be manually written and this way is more likely to verify each command output and find where it fails, i.e. when the kernel image is too large. The mkimage command is used to create the image adapted for U-boot.

```
(...)  mem=932M vmalloc=512M
mkimage -C none -A arm -T script -d boot.cmd boot.scr
```

Now that the board can boot with bananian OS, the next step is to install a Linux kernel in order to run Jailhouse. Since Banana Pi-M1 has the minimum hardware requirements, obtaining a kernel within a x86-64 machine can be more efficient and less time-consuming. To do so, a Cross Compiling (CC) tool-chain (arm-linux-gnueabihf) is needed because by default the machine will compile the kernel for its architecture when the desired one is ARM.

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j8 uImage modules
dtbs LOADADDR=40008000
```

Using the sshfs comannd, which is a secure shell hosting for filesystems that uses Filesystem in Userspace (FUSE), it is possible to mount a remote filesystem. The goal is to create a mounting point, a directory, that will permit access to the files from another machine, that is, it is established a connection between two directories from different machines by network means in order to access the kernel compiled source files and install them into the SD card. The U-boot partition is updated as well in order to the bootloader loads both the Linux image (in the U-boot file is called the uImage-next) and the device tree binaries (DTBs), in this case the "sunxi-x.dtb" files.

```
sshfs <remote user>@<remote ip address>:<linux source path>  /linux-src
```

```
cp -v arch/arm/boot/uImage /boot/uImage-next
cp -v arch/arm/boot/dts/*.dtb /boot/dtb/
```

Building Jailhouse is as simple as it is mentioned before (see section 3.1.4), except that the compilation is adapted for ARM architectures.

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- KDIR=/path/to/linux
```

The installation is rather different, considering that all files are located in the x86-64 machine, the mounting point is BPI's root.

```
sshfs root@<bananapi_ip_addr>:/  /bpi_root
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- KDIR=../linux-stable
DESTDIR= /bpi_root install
```

To run Jailhouse, the compiled files must be located in the SD card, ready to use. Since is more convenient to actually have the files and not only access them by sshfs, a new command is used: sftp, which stands for ssh file transfer protocol that ships files from one machine to another by network means. A connection is established with the board and a shell-type command line opens to transfer the files via the put command.

```
sftp root@<bananapi_ip_addr>
put jailhouse-compiled.tar.gz
quit
```

Finally, the last step is to insert the Jailhouse's kernel module by using one the commands described in section 3.1.4.

Generic Interrupt Controller Demonstration

Generic Interrupt Controller, or GIC, is the APIC's version for ARM, that is, a common interrupt handler interface device for multicore ARM uni and multiprocessors [26], which receives an input signal [12] and organizes the inputs by priority and identifiers. When the interrupts are forwarded to the

destiny CPU, it reads it to match the device that triggered it, processes the event and then writes to the GIC register to inform that the interrupt was received and handled successfully.
For each CPU core, there is a CPU interface that connects to a "distributor" interface. The distributor interface is responsible for receiving, labels it and forward the interrupts where the CPU interface only receives them and signals to the connected CPU the interrupts with enough priority [4].
There are three different source interrupts: Software Generated Interrupt (SGI), Private Peripheral Interrupt (PPI) and Shared Peripheral Interrupt (SPI) [17]. PPI is an interrupt triggered to a specific processor and SPI is one that can be redirected to multiple processor, routed by the distributor [12].
GIC demonstration, implemented for ARM processors, is a timer interrupt and displays a jitter, which, once again, calculates the delay between the received interrupts at real time and the expected time.
The environment cell configuration file for BPI, included in the attachments, has 17 defined memory regions, one of them being allocated for the SPI, and 2 devices: a pci device (IVSHMEM pci device) and irqchips (GIC).

```
jailhouse enable jailhouse/configs/arm/bananapi.cell
jailhouse cell create jailhouse/configs/arm/bannapi-gic-demo.cell
jailhouse cell load bananapi-gic-demo jailhouse/inmates/arch/arm/gic-demo.bin
jailhouse cell start bananapi-gic-demo
```

4. IVSHMEM - The Inter-Vm SHared Memory

Due to the fact that Jailhouse is a partitioning hypervisor that assigns a subset of hardware resources to each partition (denoted as "cell") that it manages and each software guest (or "inmate") run without interference from other applications, it may occur situations where these cells need to exchange data and there is the need to use some sort of communication mechanism, which is one the main goals of this project.

This chapter is related to one of the existing inter-VM shared memory mechanisms implemented for QEMU called Nahanni (or IVSHMEM) and its adaption for the Jailhouse hypervisor with support for message signaled interrupts as notification protocol, including its use in a previous demonstration (in x86-64 architecture) provided by QEMU and in ARM architecture provided by BPI. IVSHMEM functionality is enabled by an Userspace I/O driver, which is one topic to be discussed in this chapter as well.

4.1 Design of IVSHMEM

The main perk of using shared memory is to enable all processes to read and/or write directly from a shared memory region, common to all involved processes, without overflowing a buffer.

By using other methods such as a TCP/IP based communication mechanism, in general, there's a significant increase of overhead, considering that a single message has to go through multiple layers and need to use more sophisticated synchronization mechanisms.

Nahanni, or IVSHMEM, as mentioned in section 2.2.2, is a project for inter-VM communication which operates with the support of a shared mapped

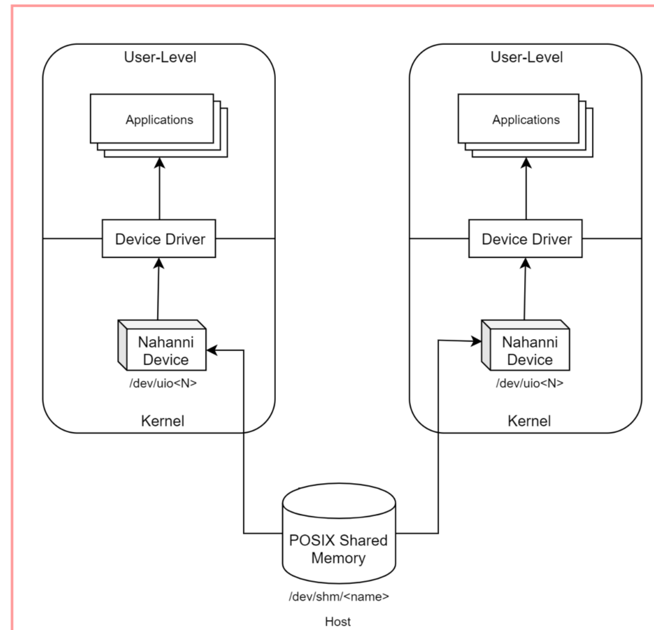

Figure 4.1: IVSHMEM or Nahanni's overview adapted from [32].

PCI device, that allows the sending of interrupts between VMs. Figure 4.1 describes IVSHMEM's architecture where there are two VMs within a host and user-level applications can access to the POSIX shared-memory via Nahanni's device under the /dev filesystem.

The IVSHMEM device (see figure 4.3) is composed by three main parts which are: the configuration section, the register memory and the shared memory.

4.1.1 The Configuration Section

The configuration section, known as "config space", has multiple sections which are visible by the OS, through the PCI bus and contains, among several information, the vendor and device ID, which allows the OS to load the correct driver for the device if the OS has available [32].

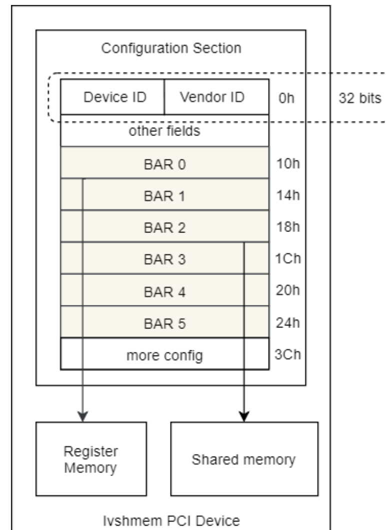

Figure 4.2: IVSHMEM PCI device architecture adapted from [32]

4.1.2 The Base Address Registers

Base address registers, or BARs, point to the regions of memory on PCI devices that are involved in the device's function. BAR0 is pointing to the register memory, which is a MMIO region [11], BAR1 is used for the MSI when enabled and BAR2 is pointing to the shared memory region [32].

4.1.3 The Register Memory

The register memory is not shared between guests. The IVSHMEM PCI device has a 256-byte register memory region.
When applications need to interact with the device driver in order to perform any tasks, such as sending interrupts, the application will read or write from/to the registers [32].

The Registers

The registers are a set of data in memory regions, measuring between 16 to 32 bits, that can be read and/or written from. The 4 registers that ivshmem uses are: Interrupt Status Register (ISR), interrupt mask register (IMR), a storage register and the doorbell (see figure 4.3).

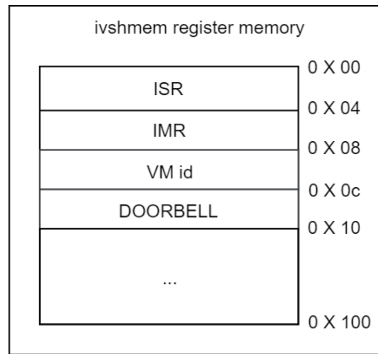

Figure 4.3: IVSHMEM PCI device registers adapted from 4.3.

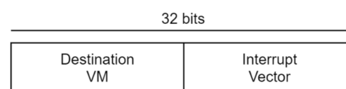

Figure 4.4: IVSHMEM PCI device doorbell adapted from 4.3.

As mentioned above, the first two registers are the main responsibles for the interrupt mechanism, in a way where both of the registers change when the nature of the interrupts are pin-based [32] [11]. Considering that the ISR is set to 1, which is triggered by an interrupt, the IMR performs a bitwise operation, resulting in a masked interrupt.
The doorbell register is composed by 2 pieces, 16-bit each, which are the "Destination VM" and the "Interrupt Vector" (figure 4.4). As well as for ISR and IMR, the doorbell also performs as a notification technology, where by using the MSI protocol, each VM that are referred in the Interrupt Vector, are going to be notified [32].

4.2 Message Signaled Interrupts

Message Signaled Interrupts, or MSI, are hardware interrupts triggered by I/O devices that support more than one interrupt per device and each is independently configured [35].
Traditional pin-based interrupts are a more complex mechanism consider-

ing that normally is shared by multiple devices so, when interrupts are triggered, the kernel is in charge of calling their handlers, resulting in a lower performance, unlike MSI [35].

Another inconvenience that rises with the use of pin-based interrupts is that it is possible that, for example, when performing a data writing operation, the interrupt may be faster than the actual operation, which will reduce performance, thus, to prevent this, the actual interrupt handler will perform an additional read to a register in order to verify if the data is already in memory, while by using MSI, the driver already acknowledges that the data is in memory when the interrupt comes [35].

4.2.1 MSI's Functionality

By default, the PCI devices are configured to use pin-based interrupts so, in order to enable this functionality, the Linux kernel must support it. The kernel must be build with the MSI-related options enabled (normally the "CONFIG_PCI_MSI") so that the following functions are available. One important note to be highlighted is that lower kernel versions than 4.8.0 are not compatible with the protocol.

MSI will allocate the maximum number of vectors supported by that PCI device (usually between 1 and 32) and forms a request by calling request_irq.

4.3 Jailhouse's Version of IVSHMEM

Considering that IVSHMEM is a shared-memory mechanism implemented for QEMU and was later adapted for the Jailhouse hypervisor, there is one major difference between them regarding the notification mechanism. The message signaled protocol implemented for QEMU's version can carry multiple interrupts at a time unlike the Jailhouse's adaptation that is one-to-one peer, i.e. one interrupt at a time.

To establish a connection between two cells, the configuration files for both cells have to define the same shared-memory region with the appropriate

permissions: read and/or write; and the same PCI device. This device must be well defined so that MSI-X[1] should function: each end of the communication channel will be emulated as a PCI device.

The hypervisor could be hosting multiple virtual machines as much as the hardware is capable to; however, when defining both PCI devices, a certain caution must be taken in order to establish the correct connections by defining a smiliar bdf identifier and the same shared-memory protocol, so that each cell can discover the devices via PCI bus.

4.3.1 IVSHMEM Demonstration

There is an IVSHMEM demonstration available only for x86-64 architecture that is programmed to send interrupts between two cells. In this case, the root cell, which is in QEMU environment and a non-root cell that is called the "ivshmem-demo".

As seen before, the hypervisor must be activated with QEMU's system cell and the IVSHMEM cell configuration must be created and loaded its inmate. Since both configuration cells have the same memory region allocated as the shared-memory and a PCI device defined, when creating the IVSHMEM cell configuration, the hypervisor will link both cells by its virtual PCI devices.

The following commands activates the hypervisor and starts the ivshmem-demo:

```
jailhouse enable /etc/jailhouse/qemu-x86.cell
jailhouse cell create /etc/jailhouse/ivshmem-demo.cell
jailhouse cell load ivshmem-demo /usr/libexec/jailhouse/demos/ivshmem-demo.bin
jailhouse cell start ivshmem-demo
```

Although a link between cells is created and the demonstrations already send interrupts, there is a piece missing in the system that is fundamental

[1] MSI-X is a more recent version of MSI which allows to more interrupts [52] [56]

to this process. By analyzing figure 4.1, a UIO device driver is needed in order to access certain kernel functionalities that are moved to userspace and the inmate could actually benefit from it by accessing the /dev/uio filesystem, i.e. read and write from/to the shared memory. To this purpose, a IVSHMEM's-based device driver was developped by Henning Schild and Cam Macdonell in order to facilitate this access.

The UIO Device

UIO was introduced to the Linux Kernel with the purpose of moving some code from the kernel and passing it to more high level layers so that user-level applications could use it.

A UIO driver is specially designed for Linux and is provided by a side git project, called the "ivshmem-guest-code", which includes the device as well as test code.

The UIO device driver configures the IVSHMEM device and requests the guest kernel to map the device memory into the kernel's address space [32]. This way, guest OSe's applications can map from kernel to user-level, which permits the kernel to map the device memory on BAR2 (the shared-memory region) into the guest kernel's virtual address space [32]. The ivshmem device is treated as a normal PCI device and will be registered under the /dev filesystem with the following format: /dev/uio"n".

The repository has three branches: master, Jailhouse and next. Nahanni is not specific for Jailhouse so trying to use the device with the main-line code is not effective. A special branch was forked from the master in order to alter and align the device within Jailhouse's requirements.

```
git clone https://github.com/henning-schild-work/ivshmem-guest-code.git
git checkout jailhouse
modprobe uio
insmod kernel_module/uio/ivshmem.ko
```

The already-to-use images for QEMU, provided by the Jailhouse team, are not capable of building a kernel module considering that the Linux modules are not imported, that is, the Linux headers. All tools could be installed via the apt command, thus, the last Linux headers installed were for a much lower version than the one that is used in the pre-installed Linux kernel, due the fact that, to simplify the QEMU demonstrations, only runtime dependencies were installed. That was sufficient to run the demonstrations and the hypervisor, although the building dependencies were hidden. This can be fixed by two means:

1. Add the following line: "IMAGE_INSTALL += linux-headers-Jailhouse" to the demo-image.bb file, located in recipes-core/images;

2. Manually transfer the Linux headers of the directory out/build/tmp/deploy/apt/Jailhouse-demo/pool/main/l/linux-$(uname -r).

Considering that the kernel module is already inserted, the uio device should now appear as a device and be listed as a PCI device. The cells should now be able to read from the shared-memory, although there is the need of using some test scripts provided by the driver:

```
apt-get install cmake
cd uio/tests/Interrupts/VM
cmake .
make
./uio/tests/Interrupts/VM/uio_read /dev/uio0 "number"
./uio/tests/Interrupts/VM/uio_send /dev/uio0 "number" 0 0
./tests/shmem_test.py
```

The activity diagram (figure 4.6) is an overview of the process from the beginning of activating the hypervisor until the expected behavior of the demonstration cell. The hypervisor needs to be activated and the uio_ivshmem

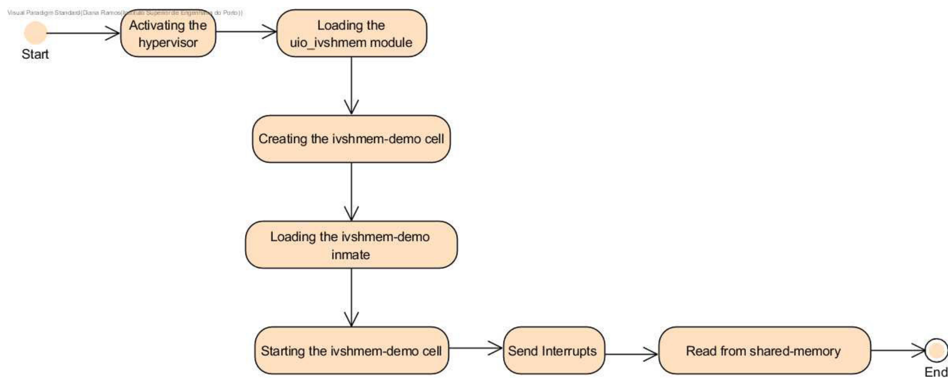

Figure 4.5: Activity diagram of IVSHMEM demonstration.

loaded so that the ivshmem-demo cell can be create, loaded with the ivshmem-demo binary and started. The inmate should receive an interrupt when the message is has been written into the shared-memory and should read it.

The driver prepares both IVSHMEM-NET[2] and traditional IVSHMEM, so one concern is to choose the right device since there is two different uio devices. If the python sripts do not eventually give any reasonable output, this might be the reason.

4.3.2 IVSHMEM Demonstration on Banana Pi-M1

Although Jailhouse has made some advancements in x86-64 architecture and already provides multiple demonstrations such as apic-demo, this hypervisor is in constant evolution and IVSHMEM, a shared-memory mechanism (uses MSI-X protocol as a notification mechanism via interruptions), has been a target mechanism to work alongside with Jailhouse within ARM environments and with great interest by the community and has been lacking for recent experiences.

[2] IVSHMEM-NET is an adaptation of the implementation of IVSHMEM for the Jailhouse hypervisor, but instead of using shared-memory, it supports networking protocols like TCP/IP.

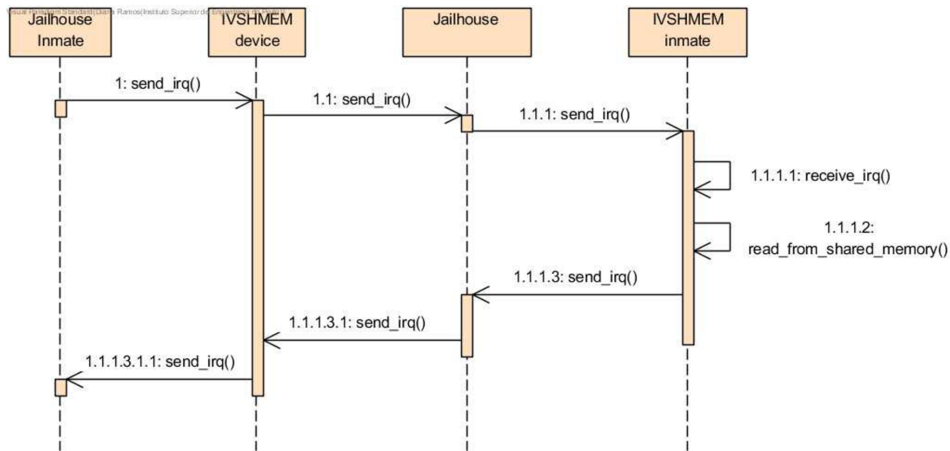

Figure 4.6: Sequence diagram for IVSHMEM demonstration.

4.3.3 Solution Implementation

The following sequence diagram represents the implementation system that was designed to run IVSHMEM under a ARM processor:

To use IVHSMEM it is required the creation of three new files (see section B.1 and B.2 from the attachments) and some modification in others. The configuration of the ivshmem-demo cell will align with the system's cell (bananapi.c) as it will define the same PCI device, bdf zone, the shared-memory region and shared-memory protocol and should be created as it follows below.

There were taken two different approaches to implement this functionality:

1. Using the uio_ivshmem driver with Linux kernel 4.13.0 and creating the necessary files based on previous IVSHMEM's demonstration files;

2. Using a recent reworked IVSHMEM driver with Linux kernel 5.2 (see section B.3 of the attachments) and build another demonstration's working configuration file.

The ivshmem-demo inmate is a binary that is loaded into the ivshmem-

```
130         while ((ndevices < MAX_NDEV) &&
131                 (-1 != (bdf =
132                         pci_cfg_find_device(pci_cfg_base, VENDORID, DEVICEID,
133                                             bdf))))
134         {
135                 printk("IVSHMEM: Found %04x:%04x at %02x:%02x.%x\n",
```

Figure 4.7: Finding the device function in ivshmem-demo inmate.

demo cell. As mentioned before, the device will appear under the UIO filesystem so that it will be a registered device. The inmate will have to manage the interrupts, map the shared-memory and BARs, find the device and send interrupts until the message is received.

The inmate must be loaded indicating the base address from where the PCI will be located as well as the interrupt request 155 and address 0x1000. These data can be found in the configuration file for BPI (.pci_mmconfig_base), which in this case is 0x2000000 and GIC is configured with pin_based to 32 and .vpci_irq_base to 123, which gives 155 for the IVSHMEM interruptions.

```
./jailhouse cell load ivshmem-demo ../inmates/demos/arm/ivshmem-demo.bin
-s "pci-cfg-base=0x02000000 ivshmem-irq=155" -a 0x1000

./jailhouse cell start ivshmem-demo
```

When following the first approach, although the uio module was successfully built against the current kernel (version 4.13.0) and inserted into the kernel, the uio_ivshmem could not be registered under the uio filesystem, therefore, it was not possible to read or write to the shared-memory nor did any interrupt could be triggered. The inmate hangs at the pci_find_device function that was needed in the main function (figure 4.7).

By following the second approach, the configuration file was based on a functional demonstration that worked with BPI (gic-demo). Although there were multiple experiments to modify the memory regions in order to run ivshmem-demo, it led to a deadlock where it would result in a parked cell due to the console's address. If this address were to be modified, another

error would occur, but if it would be removed, there would be no output. Section B.4 from the attachments lists all the parking errors and the necessary modifications to files that were needed to get to this project's point.

5. Conclusion

This chapter is related to final conclusions of the project and personal appreciations of the work. Additionally it will be introduced some features that could benefit this project and future work.

5.1 Solution Implementation

The main goal of this project was to enable a shared-memory channel to establish communication in two or more cells that the Jailhouse hypervisor could provide, based on an IPC mechanism called Nahanni, or IVSHMEM, that supported notifications via interrupts. Both cells, in this case a root and a non-root cell, have to define a PCI device and a memory region for IVSHMEM so that it can be shared between them and exchange messages. When a certain cell performs a write operation to the shared-memory, the other cell should receive a notification about it. One major step that is highly important to this process is having a device driver that takes care of these transactions called the uio_ivshmem. This uio driver is a loadable kernel module that is used to bring these low-level operations, such as write and read from the POSIX shared-memory, to the user level so that they could be implemented within the inmate space.
The approach to accomplish the goals of this project was based on the following tasks: (1) compile a Linux kernel that both Jailhouse and uio_ivshmem could be built against; (2) Explore the Jailhouse hypervisor in terms of functionality and previous implementations via QEMU, including resolving issues and (3) explore IVSHMEM functionality and be able to modify Jailhouse's files on BPI.
Although the concept perceive to be simple as it claims to be, this hypervisor is still under development and multiple patches are constantly being

released for it, in terms of Linux kernel modifications and Jailhouse itself.

The first point, regarding the Linux kernel and necessary module's compilation (Jailhouse and uio_ivshmem), was very time consuming in this project, considering that Jailhouse needs certain functionalities of the Linux kernel that are not explicitly written in the official documentation, although they have a very active mailing list that provided some information about it and I manage to build it with a correct configuration, that is, with all the minimum flags enabled. The Linux kernel version was a very crucial point as well because the hypervisor had described as a requirement a certain minimum version (Linux kernel 3.14) and the initial approach was to use the one defined in the setup for BPI (Linux kernel 4.3.3), since it was more recent than 3.14. The device driver required at least the version 4.8.0. This information was a result of multiple experiments, considering that versions above it do not implement the necessary requirements for the device driver so it would always derive from missing functions on the Linux kernel. After some research in the official Linux kernel documentation, I had to switch to another Linux version, this time to a more recent one (5.1.6) and compile with Jailhouse and uio_ivshmem flags enabled.

The second point, concerning running Jailhouse in QEMU environment, was more straight forward in a way where the goal was to run some of the Jailhouse's demonstrations and understand how to function with them, what commands Jailhouse provides and how the cells interact with each other. Using QEMU was just an initial task that was supposed to provide an immediate demonstration of the hypervisor in x86-64 architecture, without needing any particular machine. Exploring the implemented demonstrations was a way to explore the hypervisor, thus the goal was to run the ivshmem-demo. In order to run this demonstration, the uio_ivshmem was needed and since QEMU had not all the dependencies installed, the Linux headers were not updated and aligned with the current kernel version, so

the compilation of the device driver always resulted in a failed status. Other attempts were made, such as initializing manually QEMU, which resulted in a very slow process or getting an actual physical machine to run Jailhouse directly on top without any emulator. VT-d is a rather recent functionality of modern processors so, after some research about this matter, it was discovered that Intel processors that were released before 2008 would not have this capability. In a few words, there were not machines available that could run Jailhouse directly. This is why using an emulator like QEMU is very useful. By adding one line, the headers could be installed and the device could be used.

The last point was a little trickier than the others because is focused in a matter that was not very matured in the Jailhouse community and there were less help from the mailing list. There was a tutorial in the official documentation that helped me setting up everything for Jailhouse, although there were some problems when trying to use a more recent kernel and the boot partition of the sd card was too small for the amount of dtb files and u-boot image size. This was an inconvenience that needed to be solved by manually configuring it.

There was a patch related to IVSHMEM implementation on BPI that was decided that could be used as a guide. After creating, modifying and resolving some errors from these files, the uio device could not be found under the /dev/uio filesystem, therefore it could not be used. When checking the kernel log, the uio_ivshmem is active although not performing as it should. Then, following the community advises, I started working with IVSHMEM 2.0 that, at that time, had been recently announced. Following this path, a new kernel had to be compiled again with the required functionalities enabled. Since the used Linux kernel was version 4.13.0 and the Jan Kiszka's repository was using the most recent stable kernel, which at the time was version 5.2, were taken some cautions with newer or modified

configurations so that it would not interfere with Jailhouse. Since gic-demo was a functional demonstration in BPI, the ivshmem-demo was based on it and it took numerous iterations and problem solving in order to finalize it this way.

Summing up, there were three main goals: (1) Running Jailhouse in QEMU; (2) Running Jailhouse in BPI and (3) Running IVSHMEM in Jailhouse on BPI-M1. Jailhouse was ran in QEMU and some demonstrations such as apic-demo and ivshmem-demo, which gave some knowledge about their particular functionality and how it could be passed down to other platforms. Jailhouse was also successfully ran in BPI-M1 and gic-demo was also ran. Running IVSHMEM was not successfully. As mentioned before, using a first approach that included a side project's uio device driver did not gave the desired output, thus the files were correctly configured; and using a second approach which already included a reworked driver in the kernel, had many configuration problems that some ended in a deadlock.

5.2 Limitations and Future Work

One major limitation was the kernel compilation with the drivers that took a considerable amount of time to compile with an appropriate configuration.

In the future, the plan is to continue exploring IVSHMEM and make it work in a use case in order to accomplish the final goal of this internship.

5.3 Final Appreciation

This project was taken from a very different context that I am used to work with, so this was presented to me as a challenge. I have managed to work with numerous kernels and learn how a simple modification on the configuration can go a long way. I had the opportunity to learn some kernel internals as well and the functionalities that it involves. Device drivers were an important topic to learn from as well. The hot topic of the hypervisors was very explored, such as memory management mechanisms. I also gained

some agility in the command line and there are some other skills that were developed during this project that can not be very well demonstrated in this report, although I am very pleased with all the things that I have learned, developed and consolidated and it will be useful in the future.

Although one of the goals were not completely satisfactory, I am satisfied with the solution as a whole. Jailhouse is a recent technology that is still under development and entering in its area and interacting with the community helped me work on some certain topics. Running Jailhouse in multiple platforms and have actual cells giving reasonable outputs were both two successful goals.

Bibliography

[1] In: *AMD-V^{TM} Nested Paging*.

[2] In: *Intel® 64 and IA-32 Architectures Software Developer's Manual. System Programming Guide, Part 3*. Vol. 3C.

[3] *About Us*. URL: https://www.cister.isep.ipp.pt/info/. (accessed: 27.05.2019).

[4] Saravana Pandian Annamalai. *ARM INTERRUPT CONTROLLERS*. URL: http://www.embien.com/blog/arm-interrupt-controllers/. (accessed: 30.08.2019).

[5] Maxim Baryshnikov. "Jailhouse Hypervisor". Czech Technical University in Prague, 2016.

[6] *BPI-M1*. URL: http://www.banana-pi.org/m1.html. (accessed: 27.05.2019).

[7] *CHAPTER 1. SYSTEM REQUIREMENTS*. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/virtualization/chap-virtualization-system_requirements. (accessed: 2.06.2019).

[8] *CHAPTER 1. SYSTEM REQUIREMENTS*. URL: https://www.softprayog.in/programming/interprocess-communication-using-posix-shared-memory-in-linux. (accessed: 2.06.2019).

[9] *Creating Deterministic Applications (Real-Time Module)*. URL: https://zone.ni.com/reference/en-XX/help/370715P-01/lvrtconcepts/builddeterapps_rt/. (accessed: 11.09.2019).

[10] C. Dall et al. “ARM Virtualization: Performance and Architectural Implications”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 304–316. DOI: 10.1109/ISCA.2016.35.

[11] *Device Specification for Inter-VM shared memory device*. URL: https://chromium.googlesource.com/external/qemu/+/v2.4.0-rc2/docs/specs/ivshmem_device_spec.txt. (accessed: 31.08.2019).

[12] Larry D.Pyeatt. In: *Modern Assembly Language Programming with the ARM Processor*. 2016. Chap. 14 - Running Without an Operating System.

[13] R. P. Goldberg. *Architecture of Virtual Machines*. Cambridge, Massachusetts: Harvard University Cambridge, 1973.

[14] *Hard Partitioning for Linux: The Jailhouse Hypervisor*. URL: https://events.static.linuxfound.org/sites/events/files/slides/LinuxConNA-2015-Jailhouse_0.pdf. (accessed: 12.09.2019).

[15] *How to write a kernel module for Linux*. URL: http://cs.smith.edu/~nhowe/Teaching/csc262/oldlabs/module.html. (accessed: 8.09.2019).

[16] *Hypervisor*. URL: https://www.vmware.com/topics/glossary/content/hypervisor. (accessed: 24.03.2019).

[17] ideaopensource. *ARM Global Interrupt controller GIC v2*. URL: https://ideaopensource.wordpress.com/2016/08/04/arm-global-interrupt-controller-gic-v2-basic-info-wiki/. (accessed: 30.08.2019).

[18] *Intel Virtualisation: How VT-x, KVM and QEMU Work Together*. URL: https://binarydebt.wordpress.com/2018/10/14/intel-virtualisation-how-vt-x-kvm-and-qemu-work-together/. (accessed: 7.09.2019).

[19] *Intel® Virtualization Technology for Directed I/O (VT-d): Enhancing Intel platforms for efficient virtualization of I/O devices*. URL: https://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices. (accessed: 21.08.2019).

[20] *Intel® Virtualization Technology (Intel® VT)*. URL: https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html?wapkw=vt-d. (accessed: 21.08.2019).

[21] *Internet Protocol stack in Internet protocol suite (TCP/IP)*. URL: https://medium.com/@anna7/internet-protocol-layers-in-internet-protocol-suite-tcp-ip-abe038c0adde. (accessed: 31.05.2019).

[22] *Interrupts*. URL: https://linux-kernel-labs.github.io/master/lectures/interrupts.html. (accessed: 29.08.2019).

[23] Pavle Ivanovic and Harald Richter. “Performance Analysis of Ivshmem for High-Performance Computing in Virtual Machines”. In: (2017).

[24] *Jailhouse*. URL: https://github.com/siemens/jailhouse. (accessed: 24.02.2019).

[25] *Jailhouse Image*. URL: https://github.com/siemens/jailhouse-images. (accessed: 24.02.2019).

[26] jasona. *ARM Generic Interrupt Controller HOWTO*. URL: https://community.cadence.com/cadence_blogs_8/b/sd/posts/arm-generic-interrupt-controller-architecture-howto. (accessed: 30.08.2019).

[27] Alessandro Rubini Jonathan Corbet and Greg Kroah-Hartman. In: *Linux Device Drivers*. 3rd ed.

[28] *Kernel Virtual Machine*. URL: https://www.linux-kvm.org/page/Main_Page. (accessed: 27.05.2019).

[29] Avi Kivity et al. “KVM: the Linux Virtual Machine Monitor”. In: *In Proceedings of the 2007 Ottawa Linux Symposium (OLS’-07*. 2007.

[30] Deepak Kumar and Amaizo Folly Felix Magloire. “Hypervisor based performance characterization: XEN/KVM”. In: Noida, India: IEEE, 2017. URL: https://ieeexplore.ieee.org/document/8343570.

[31] *Linux lsmod command*. URL: https://www.computerhope.com/unix/lsmod.htm. (accessed: 14.09.2019).

[32] A. Cameron Macdonell. “Shared-Memory Optimizations for Virtual Machines”. University of Alberta, 2011.

[33] *Main Page*. URL: https://wiki.qemu.org/Main_Page. (accessed: 24.03.2019).

[34] Karissa Miller and Mahmoud Pegah. “Virtualization, Virtually at the Desktop”. In: (2007).

[35] *MSI-HOWTO*. URL: https://github.com/linuxkit/linux/blob/master/Documentation/PCI/MSI-HOWTO.txt. (accessed: 31.08.2019).

[36] *Nested virtualization in KVM*. URL: https://stafwag.github.io/blog/blog/2018/06/04/nested-virtualization-in-kvm/. (accessed: 14.09.2019).

[37] *Para virtualization vs Full virtualization vs Hardware assisted Virtualization*. URL: https://www.unixarena.com/2017/12/para-virtualization-full-virtualization-hardware-assisted-virtualization.html/. (accessed: 13.09.2019).

[38] *Posix Standard*. URL: https://linuxhint.com/posix-standard/. (accessed: 15.09.2019).

[39] *Python module for Jailhouse*. URL: https://wiki.qemu.org/Google_Summer_of_Code_2018#Python_module_for_Jailhouse. (accessed: 22.08.2019).

[40] Daniel Lohmann Ralf Ramsauer Jan Kiszka and Wolfgang Mauerer. "Look Mum, no VM Exits! (Almost)". In: (2017).

[41] Yi Ren et al. "Shared-Memory Optimizations for Inter-Virtual-Machine Communication". In: *ACM Comput. Surv.* 48.4 (Feb. 2016), 49:1–49:42. ISSN: 0360-0300. DOI: 10.1145/2847562. URL: http://doi.acm.org/10.1145/2847562.

[42] Gleb Reys. *HW Virtualization*. URL: https://www.unixtutorial.org/hw-virtualization. (accessed: 24.03.2019).

[43] *Setup on Banana Pi ARM board*. URL: https://github.com/siemens/jailhouse/blob/master/Documentation/setup-on-banana-pi-arm-board.md. (accessed: 29.08.2019).

[44] Valentine Sinitsyn. *Jailhouse*. URL: https://www.linuxjournal.com/content/jailhouse. (accessed: 29.08.2019).

[45] Valentine Sinitsyn. "QEMU: a Multihost, Multitarget Emulator". In: (). URL: https://www.linuxjournal.com/article/8808.

[46] Joakim Svensson and Henrik Andersson. "Virtualization in an embedded environment". In: ().

[47] *Towards an ivshmem 2.0?* URL: https://groups.google.com/forum/#!searchin/jailhouse-dev/ivshmem%5C$202.0%5C%7Csort:date/jailhouse-dev/Rvg4UR6A_gM/ApqQgX1JDAAJ. (accessed: 8.09.2019).

[48] *TROUBLESHOOTING A LINUX KERNEL PANIC AFTER PATCHING*. URL: https://www.linuxnix.com/troubleshooting-linux-kernel-panic-patching/. (accessed: 27.08.2019).

[49] *uio: Add driver for inter-VM shared memory device*. URL: http://git.kiszka.org/. (accessed: 7.09.2019).

[50] Aditya Venkataraman and Kishore Kumar Jagadeesha. “Evaluation of Inter-Process Communication Mechanisms”. In: (). URL: http://pages.cs.wisc.edu/~adityav/Evaluation_of_Inter_Process_Communication_Mechanisms.pdf.

[51] *Virtualization: an overview*. URL: https://www.ionos.com/digitalguide/server/configuration/virtualization/. (accessed: 11.09.2019).

[52] *What do the different interrupts in PCIe do? I referring to MSI, MSI-X and INTx*. URL: https://electronics.stackexchange.com/questions/76867/what-do-the-different-interrupts-in-pcie-do-i-referring-to-msi-msi-x-and-intx. (accessed: 12.09.2019).

[53] *What is an RTOS?* URL: https://www.highintegritysystems.com/rtos/what-is-an-rtos/. (accessed: 1.06.2019).

[54] *What is KVM?* Red Hat. URL: https://www.redhat.com/en/topics/virtualization/what-is-KVM. (accessed: 10.05.2019).

[55] *Xen Project Software Overview*. Xen Project. URL: https://wiki.xen.org/wiki/Xen_Project_Software_Overview. (accessed: 27.05.2019).

[56] *Xilinx PCI Express Interrupt Debugging Guide*. URL: https://www.xilinx.com/Attachment/Xilinx_Answer_58495_PCIe_Interrupt_Debugging_Guide.pdf. (accessed: 12.09.2019).

[57] Richard West Ye Li and Eric Missimer. “The Quest-V Separation Kernel for Mixed Criticality Systems”. In: (2011).

A. Setting up Jailhouse with BPI-M1

This chapter is related to a script for setting up Jailhouse with BPI-M1 based on [43].

A.1 Formatting BPI-M1 sd Card

Since BPI-M1 boots with the assistance of a sd card, it will need to go through a formatting process in order to boot with bananian OS.

```
# Get bananian OS
$ wget https://dl.bananian.org/releases/bananian-latest.zip
$ sudo apt-get update  sudo apt-get install unzip screen
$ unzip bananian-latest.zip

# Insert the sd card and search for its partition
$ sudo fdisk -l

# Write the image to sd card into the result of the search
$ sudo dd bs=1M if=bananian-*.img of=/dev/"sd card"

# Insert the sd card into BPI-M1 and establish a serial connection
$ dmesg | grep tty

# Connection via serial on Ubuntu
$ screen /dev/ttyUSB"N" 115200
# Connection via serial on Ubuntu with putty
$ sudo apt-get install putty
$ putty

# On BananaPi, login with root/pi, then expand the filesystem
$ bananian-config
```

A.2 Adjust U-boot

Mount U-boot partition

$ mkdir /p1 $ mount /dev/mmcblk0p1 /p1 $ vi /p1/boot.cmd

Create U-boot image

$ apt-get update apt-get install -y u-boot-tools

In case of firmware error

$ dpkg -i –force-overwrite /var/cache/apt/archives/firmware-misc-nonfree_20161130-5 deb8u1_all.deb

$ apt-get update apt-get install -y u-boot-tools

$ cd /p1

$ mkimage -C none -A arm -T script -d boot.cmd boot.scr

A.3 Cross Compiling Kernel for ARM on x86

$ sudo apt-get install -y git

$ cd

$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git

$ git clone https://github.com/Bananian/bananian.git

$ git clone https://github.com/siemens/jailhouse.git

Download the recommended cross-toolchain from Linaro $ cd

$ wget https://releases.linaro.org/components/toolchain/binaries/latest-5/arm-linux-gnueabihf/gcc-linaro-5.5.0-2017.10-x86_64_arm-linux-gnueabihf.tar.xz

$ tar -xf gcc-linaro-5.5.0-2017.10-x86_64_arm-linux-gnueabihf.tar.xz

Update environment path $ export PATH=$PATH$:(pwd)/gcc-linaro-5.5.0-2017.10-x86_64_arm-linux-gnueabihf/bin

Choose any Jailhouse-recommended version. Bananian Team implemented some patches for certain versions of Linux kernel v4.x

$ cd /linux-stable

$ git checkout v4.3.3

$ for i in ../bananian/kernel/4.3.3/patches/*; do patch -p1 < $i; done

$ sudo apt-get update sudo apt-get install -y build-essential libncurses5 libncurses5-dev

Enable FUSE under "File systems", needed for file transfer via sshfs and other functionalities that Jailhouse requires (virtualization, virtualization drivers, MSI and MSI-X support, etc)

$ make ARCH=arm menuconfig

$ sudo apt-get update && sudo apt-get install -y u-boot-tools

$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j8 uImage modules dtbs LOADADDR=40008000

A.4 Installing the Linux kernel on BPI-M1

On Compiling machine,

$ sudo apt-get update && sudo apt-get install -y sshfs

On BananaPi,

$ apt-get update && apt-get install -y sshfs make gcc

$ mkdir /linux-src

$ sshfs hostname@machine_ip:/path/to/linux/ /linux-src

$ cd /linux-src

$ make modules_install

Update U-boot partition

$ mount /dev/mmcblk0p1 /boot

$ mkdir /boot/dtb/

$ cp -v arch/arm/boot/uImage /boot/uImage-next

$ cp -v arch/arm/boot/dts/*.dtb /boot/dtb/

now reboot $ reboot

Verify kernel installation

$ uname -r

A.5 Cross Compiling Jailhouse for ARM on x86

On Compiling machine, $ make –version

If make version is higher or equal to 3.82, skip this part

$ sudo apt-get update sudo apt-get install -y checkinstall

$ wget http://ftp.gnu.org/gnu/make/make-3.82.tar.bz2 -O /Downloads/make-3.82.tar.bz2

$ cd tar -xf /Downloads/make-3.82.tar.bz2

$ cd make-3.82

$ ./configure –prefix=/usr

$ make

$ sudo checkinstall make install

On Compiling Machine

$ sudo apt-get update sudo apt-get install -y python-mako device-tree-compiler

$ cd

Copy the configuration header file before building

$ cp -av ./jailhouse/ci/jailhouse-config-banana-pi.h ./jailhouse/include/jailhouse/config.h

$ cd ./jailhouse

$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- KDIR=../linux-stable

A.6 Installing jailhouse

On Compiling Machine, create a mounting point in BPI-M1

$ mkdir /bpi_root

$ sshfs root@bpi_ip:/ /bpi_root

$ cd /jailhouse

$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- KDIR=../linux-stable DESTDIR= /bpi_root install

Zip all the necessary files to send to BPI-M1

$ cd tar -zcf jailhouse-compiled.tar.gz jailhouse

$ sftp -P 22 root@bpi_ip

sftp > put jailhouse-compiled.tar.gz

sftp > quit

On BananaPi

$ cd && tar -xf jailhouse-compiled.tar.gz

$ modprobe jailhouse

A.7 GIC-demo's Outputs

In this section it is presented the gic-demo's outputs followed by: the caption of each image.

```
Initializing Jailhouse hypervisor v0.11 (0-g58052a7-dirty) on CPU 1
Code location: 0xf0000040
Page pool usage after early setup: mem 56/16362, remap 0/131072
Initializing processors:
 CPU 1... OK
 CPU 0... OK
Initializing unit: irqchip
Initializing unit: PCI
Adding virtual PCI device 00:00.0 to cell "Banana-Pi"
Page pool usage after late setup: mem 68/16362, remap 5/131072
Activating hypervisor
```

Hypervisor's initialization on BPI.

```
Created cell "bananapi-gic-demo"
Page pool usage after cell creation: mem 79/16362, remap 5/131072
```

GIC-demo cell creation.

```
Cell "bananapi-gic-demo" can be loaded
```

GIC-demo inmate loading.

```
Timer fired, jitter:   833 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   708 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   708 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   708 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   833 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   916 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   624 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   624 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   624 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   708 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   708 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   708 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   624 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   624 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   708 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:  1083 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   624 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   708 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
Timer fired, jitter:   666 ns, min:   624 ns, max:  3624 ns
```

GIC-demo output.

```
ID      Name                    State           Assigned CPUs           Failed CPUs
0       Banana-Pi               running         0
1       bananapi-gic-demo       running         1
```

GIC-demo cell list output.

B. Jailhouse's IVSHMEM related files

This chapter is related to IVSHMEM implementation files added to Jailhouse. It is worth mentioned that the created files were based on a patch released on the Jailhouse mailing list by Jonas West. The inmate is the original file by this author although the configuration file is altered.

B.1 IVSHMEM Demonstration Configuration File

```
/*
 * Jailhouse, a Linux-based partitioning hypervisor
 *
 * Configuration for gic-demo inmate on Banana Pi:
 * 1 CPU, 64K RAM, serial ports 4-7, CCU+GPIO
 *
 * Copyright (c) Siemens AG, 2014
 *
 * Authors:
 *  Jan Kiszka <jan.kiszka@siemens.com>
 *
 * This work is licensed under the terms of the GNU GPL, version 2.
See
 * the COPYING file in the top-level directory.
 */

#include <Jailhouse/types.h>
#include <Jailhouse/cell-config.h>

struct {
        struct Jailhouse_cell_desc cell;
```

```
        __u64 cpus[1];
        struct Jailhouse_memory mem_regions[9];
        struct Jailhouse_irqchip irqchips[1];
        struct Jailhouse_pci_device pci_devices[1];
} __attribute__((packed)) config = {
        .cell = {
                .signature = Jailhouse_CELL_DESC_SIGNATURE,
                .revision = Jailhouse_CONFIG_REVISION,
                .name = "bananapi-ivshmem-demo",
                .flags = Jailhouse_CELL_PASSIVE_COMMREG,

                //.pio_bitmap_size = 0,
                .vpci_irq_base = 123,

                .cpu_set_size = sizeof(config.cpus),
                .num_memory_regions = ARRAY_SIZE(config.mem_regions),
                .num_irqchips = ARRAY_SIZE(config.irqchips),
                .num_pci_devices = ARRAY_SIZE(config.pci_devices),

                .console = {
                        .address = 0x01c29c00,
                        .clock_reg = 0x01c2006d,
                        .gate_nr = 23,
                        .divider = 0x0d,
                        .type = Jailhouse_CON_TYPE_8250,
                        .flags = Jailhouse_CON_ACCESS_MMIO |
                                 Jailhouse_CON_REGDIST_4,
                },
        },

        .cpus = {
                0x2,
        },
```

```
.mem_regions = {
        /* CCU */ {
        .phys_start = 0x01c20200,
        .virt_start = 0x01c20200,
        .size = 0x200,
        .flags = Jailhouse_MEM_READ | Jailhouse_MEM_WRITE |
                Jailhouse_MEM_IO | Jailhouse_MEM_IO_32,
        },
        /* GPIO: port H */ {
        .phys_start = 0x01c208fc,
        .virt_start = 0x01c208fc,
        .size = 0x24,
        .flags = Jailhouse_MEM_READ | Jailhouse_MEM_WRITE |
                Jailhouse_MEM_IO | Jailhouse_MEM_IO_32,
        },
        /* UART 4-7 */ {
        .phys_start = 0x01c29000,
        .virt_start = 0x01c29000,
        .size = 0x1000,
        .flags = Jailhouse_MEM_READ | Jailhouse_MEM_WRITE |
                Jailhouse_MEM_IO,
        },
        /* RAM */ {
        .phys_start = 0x7bef0000,
        .virt_start = 0,
        .size = 0x00010000,
        .flags = Jailhouse_MEM_READ | Jailhouse_MEM_WRITE |
                Jailhouse_MEM_EXECUTE | Jailhouse_MEM_LOADABLE,
        },
        /* IVSHMEM */ {
        .phys_start = 0x7bf00000,
    .virt_start = 0x7bf00000,
```

```
		.size = 0x000100000,
			.flags = Jailhouse_MEM_READ | Jailhouse_MEM_WRITE |
					Jailhouse_MEM_ROOTSHARED,
		},
		{ 0 },
		{
			.phys_start = 0x7bf01000,
			.virt_start = 0x7bf01000,
			.size = 0x7f000,
			.flags = Jailhouse_MEM_READ |
			Jailhouse_MEM_ROOTSHARED,
		},
		{
			.phys_start = 0x7bf80000,
			.virt_start = 0x7bf80000,
			.size = 0x7f000,
			.flags = Jailhouse_MEM_READ
				| Jailhouse_MEM_WRITE |
					Jailhouse_MEM_ROOTSHARED,
		},
		/* communication region */ {
			.virt_start = 0x80000000,
			.size = 0x00001000,
			.flags = Jailhouse_MEM_READ
			 | Jailhouse_MEM_WRITE |
			 Jailhouse_MEM_COMM_REGION,
		},
	},
	.irqchips = {
			/* GIC */ {
			.address = 0x01c81000,
			.pin_base = 32,
			.pin_bitmap = {
```

```
                                        1ULL<<(52-32),
                                        0,
                                        0,
                                        1 << (155-128),
                                },
                        },
                },
        .pci_devices = {
                {
                .type = Jailhouse_PCI_TYPE_IVSHMEM,
                .bdf = 0x00,
                .bar_mask = {
                        0xffffff00, 0xffffffff, 0x00000000,
                        0x00000000, 0x00000000, 0x00000000,
                },
                .shmem_regions_start = 4,
                .shmem_dev_id = 1,
                .shmem_peers = 2,
                .shmem_protocol = Jailhouse_SHMEM_PROTO_UNDEFINED,
                },
        },
};
```

B.2 IVSHMEM Demonstration Inmate File

```
/*
 * Jailhouse, a Linux-based partitioning hypervisor
 *
 * Copyright (c) Siemens AG, 2014-2016
 * Copyright (c) Retotech AB, 2017
 *
 * Authors:
 *  Henning Schild <hennin...@siemens.com>
```

```
 *  Jonas Westker <jo...@retotech.se>
 *
 * This work is licensed under the terms of the GNU GPL, version 2.
See
 * the COPYING file in the top-level directory.
 */
#include <inmate.h>
#include <mach.h>

#define VENDORID        0x1af4
#define DEVICEID        0x1110

#define IVSHMEM_CFG_SHMEM_PTR       0x40
#define IVSHMEM_CFG_SHMEM_SZ        0x48

#define Jailhouse_SHMEM_PROTO_UNDEFINED         0x0000

//#define IRQ_VECTOR    32

#define MAX_NDEV        4
#define UART_BASE        0x3F8

static char str[] = "Hello from bare-metal ivshmem-demo inmate!!!
";
static int irq_counter;

struct ivshmem_dev_data {
        u16 bdf;
        u32 *registers;
        void *shmem;
        u64 shmemsz;
};
```

```
static struct ivshmem_dev_data devs[MAX_NDEV];


static u64 pci_cfg_read64(u32 base, u16 bdf, unsigned int addr)
{
        u64 bar = (((u64)pci_cfg_read(base, bdf, addr + 4, 4) << 32) |
                   pci_cfg_read(base, bdf, addr, 4));
        return bar;
}

static void pci_cfg_write64(u32 base, u16 bdf, unsigned int addr, u64 val)
{
        pci_cfg_write(base, bdf, addr + 4, (u32)(val >> 32), 4);
        pci_cfg_write(base, bdf, addr, (u32)val, 4);
}

static int map_shmem_and_bars(u32 base, struct ivshmem_dev_data *d)
{
        d->shmemsz = pci_cfg_read64(base, d->bdf, IVSHMEM_CFG_SHMEM_SZ);
        d->shmem =
            (void *)((u32)(0xffffffff &
                           pci_cfg_read64(base, d->bdf,
                                          IVSHMEM_CFG_SHMEM_PTR)));
        printk("IVSHMEM: shmem is at %p\n", d->shmem);
        d->registers =
            (u32 *)(((u32)(d->shmem + d->shmemsz + PAGE_SIZE - 1))
            & PAGE_MASK);
        pci_cfg_write64(base, d->bdf, PCI_CFG_BAR, (u32)d->registers);
        printk("IVSHMEM: bar0 is at %p\n", d->registers);
        pci_cfg_write(base, d->bdf, PCI_CFG_COMMAND,
                      (PCI_CMD_MEM | PCI_CMD_MASTER), 2);
        return 0;
}
```

```
static u32 get_ivpos(struct ivshmem_dev_data *d)
{
        return mmio_read32(d->registers + 2);
}

static void send_irq(struct ivshmem_dev_data *d)
{
        printk("IVSHMEM: %02x:%02x.%x sending IRQ
        (by writing 1 to 0x%x)\n",
               d->bdf >> 8, (d->bdf >> 3) & 0x1f, d->bdf & 0x3,
               d->registers + 3);
        mmio_write32(d->registers + 3, 1);
}

static void enable_irq(struct ivshmem_dev_data *d)
{
        printk("IVSHMEM: Enabling IVSHMEM_IRQs\n");
        mmio_write32(d->registers, 0xffffffff);
}

static void handle_irq(unsigned int irqn)
{
        printk("IVSHMEM: handle_irq(irqn:%d) - interrupt #%d\n",
               irqn, irq_counter++);
}

void inmate_main(void)
{
        unsigned int i = 0;
        int bdf = 0;
        unsigned int class_rev;
        struct ivshmem_dev_data *d;
```

```
volatile char *shmem;
int ndevices = 0;

gic_setup(handle_irq);

long long pci_cfg_base = cmdline_parse_int("pci-cfg-base", -1);
if (-1 == pci_cfg_base) {
    printk("ERROR: Provide value for 'pci-cfg-base'-parameter\n"
    "(using cmdline when loading ivshmem-demo inmate:\n"
    "' -s \"pci-cfg-base=<pci_mmconfig_base>\" -a <address>').\n"
    "Check root-cell configuration file:\n"
    "config.header.platform_info.pci_mmconfig_base \n"
    "for details on a value applicable to your target system.\n");
    return;
}
printk("IVSHMEM: pci-cfg-base:0x%llx\n", pci_cfg_base);

long long ivshmem_irq = cmdline_parse_int("ivshmem-irq", -1);
if (-1 == ivshmem_irq) {
       printk("ERROR: Provide value for 'ivshmem-irq'-parameter\n"
             "(using cmdline when loading ivshmem-demo inmate:\n"
             "' -s \"ivshmem-irq=<value>\" -a <address>').\n");
       return;
}
printk("IVSHMEM: ivshmem-irq:%d\n", ivshmem_irq);

while ((ndevices < MAX_NDEV) &&
       (-1 != (bdf =
             pci_cfg_find_device(pci_cfg_base, VENDORID, DEVICEID,
                                         bdf))))
{
printk("IVSHMEM: Found %04x:%04x at %02x:%02x.%x\n",
  pci_cfg_read(pci_cfg_base, bdf, PCI_CFG_VENDOR_ID, 2),
```

```
    pci_cfg_read(pci_cfg_base, bdf, PCI_CFG_DEVICE_ID, 2),
    bdf >> 8, (bdf >> 3) & 0x1f, bdf & 0x3);
  class_rev = pci_cfg_read(pci_cfg_base, bdf, 0x8, 4);
  if (class_rev != (PCI_DEV_CLASS_OTHER << 24 |
        Jailhouse_SHMEM_PROTO_UNDEFINED << 8)) {
      printk("IVSHMEM: class/revision %08x, not supported "
             "skipping device\n", class_rev);
      bdf++;
      continue;
  }
      ndevices++;
      d = devs + ndevices - 1;
      d->bdf = bdf;
      if (map_shmem_and_bars(pci_cfg_base, d)) {
              printk("IVSHMEM: Failure mapping shmem and bars.\n");
              return;
      }

      printk("IVSHMEM: mapped shmem and bars,
          got position %p\n",
             get_ivpos(d));

      gic_enable_irq(ivshmem_irq + ndevices - 1);
      printk("IVSHMEM: Enabled IRQ:0x%x\n",
             ivshmem_irq +  ndevices -1);

      enable_irq(d);
      bdf++;
  }

  if (!ndevices) {
      printk("IVSHMEM: No PCI devices found .. nothing to do.\n");
      return;
```

```
        }

        printk("IVSHMEM: Done setting up...\n");

        {
            u8 buf[32];
            memcpy(buf, d->shmem, sizeof(buf)/sizeof(buf[0]));
            printk("IVSHMEM: %s\n", buf);
            memcpy(d->shmem, str, sizeof(str)/sizeof(str[0]) + 1);
        }

        while (1) {
            for (i = 0; i < ndevices; i++) {
                    d = devs + i;
                    //delay_us(1000*1000);
                    shmem = d->shmem;
                    shmem[sizeof(str)/sizeof(str[0])]++;
                    send_irq(d);
            }
            printk("IVSHMEM: waiting for interrupt.\n");
            asm volatile("wfi" : : : "memory");
        }
}
```

B.3 IVSHMEM 2.0

IVSHMEM 2.0 is a reworked version of IVSHMEM, specially designed for Jailhouse, that can be found in Jan Kiszka's repository (http://git.kiszka.org/linux.git/). Since the uio_ivshmem driver could be used via a side repository, the purpose was to create a patch to include it within the kernel and be found by the kernel configuration's tools.

```
git clone git.kiszka.org/linux.git
git checkout queues/Jailhouse-ivshmem2
```

Jailhouse also has a side branch for this particular kernel configuration. After cloning the repository, the branch must be switched to wip/ivshmem2 as it follows:

```
git checkout wip/ivshmem2
```

This driver "exposes the MMIO register region and all shared memory section to userspace. Interrupts are configured in one-shot mode so that userspace needs to re-enable them after each event via the Interrupt Control register. The driver registers all possible MSI-X vectors, coalescing them into the single notifier UIO provides" [49].
Accordingly to IVSHMEM 2.0 announcement, this arranged driver differs from the original in the following aspects [47]:

1. "Only two peers per link: This simplifies the implementation and also the interfaces (think of life-cycle management in a multi-peer environment). Moreover, we do not have an urgent use case for multiple peers, thus also not reference for a protocol that could be used in such setups. If someone else happens to share such a protocol, it would be possible to discuss potential extensions and their implications".

2. "Side-band registers to discover and configure share memory regions: This was one of the first changes: We removed the memory regions from the PCI BARs and gave them special configuration space registers. By now, these registers are embedded in a PCI capability. The reasons are that Jailhouse does not allow to relocate the regions in guest address space (but other hypervisors may if they like to) and that we now have up to three of them".

3. "Changed PCI base class code to 0xff (unspecified class): This allows us to define our own sub classes and interfaces. That is now exploited for specifying the shared memory protocol the two connected peers should use. It also allows the Linux drivers to match on that".

4. "INTx interrupts support : This is needed on target platforms without MSI controllers, i.e. without the required guest support. Namely some PCI-less ARM SoCs required the reintroduction. While doing this, we also took care of keeping the MMIO registers free of privileged controls so that a guest OS can map them safely into a guest userspace application".

And some extensions that were added [47]:

1. "Multiple shared memory regions, including unidirectional ones: It is now possible to expose up to three different shared memory regions: The first one is read/writable for both sides. The second region is read/writable for the local peer and read-only for the remote peer (useful for output queues). And the third is read-only locally but read/writable remotely (ie. for input queues). Unidirectional regions prevent that the receiver of some data can interfere with the sender while it is still building the message, a property that is not only useful for safety critical communication, we are sure".

2. "Life-cycle management via local and remote state: Each device can now signal its own state in form of a value to the remote side, which triggers an event there. Moreover, state changes done by the hypervisor to one peer are signalled to the other side. And we introduced a write-to-shared-memory mechanism for the respective remote state so that guests do not have to issue an MMIO access in order to check the state".

B.4 Parking Errors

This section is related to unsuccessful experiments regarding creating configuration files for BPI and ivshmem-demo. It was taken into consideration one of the already running demonstrations (gic-demo) in order to build a file

for ivshmem. Since the files needed to be aligned with the addresses and memory regions that they allocated, most of the experiences resulted in a "parked cell".

```
Unhandled data read at 0x1c25014(4)
FATAL: unhandled trap (exception class 0x24)
pc=0xc0543d64 cpsr=0xa00f0193 hsr=0x93850007
r0=0x00000030 r1=0xde074840 r2=0x00000000 r3=0xe0037000
r4=0xde074840 r5=0xde8a6068 r6=0x00000000 r7=0xc0a01eb8
r8=0x00000030 r9=0xde8a6000 r10=0xc0a69bbc r11=0xc07f905c
r12=0x1e6fe000 r13=0xc0a01e68 r14=0xc0163764
Parking CPU 0 (Cell: "Banana-Pi")

Unhandled data read at 0x1c1600c(4)
FATAL: unhandled trap (exception class 0x24)
pc=0xc0473924 cpsr=0x200f0013 hsr=0x93830007
r0=0xef60ff16 r1=0x00000021 r2=0xfffffff8 r3=0xe0029000
r4=0x0d2e6416 r5=0x00000022 r6=0xde854040 r7=0xc0a04c48
r8=0xde068000 r9=0xde0680dc r10=0xde854040 r11=0x00000001
r12=0x00000018 r13=0xde109e98 r14=0x29aaaaab
Parking CPU 1 (Cell: "Banana-Pi")

#When adding the regions "HDMI" and "RTP".

Unhandled data read at 0x1c1600c(4)
FATAL: unhandled trap (exception class 0x24)
pc=0xc047b19c cpsr=0x80010013 hsr=0x93830007
r0=0x8c31a9d6 r1=0x0000003e r2=0xfffffff8 r3=0xe0029000
r4=0xa9ff0ed6 r5=0x0000003e r6=0xdea33040 r7=0xc0b04c48
```

```
r8=0xdeb24800 r9=0xdeb248dc r10=0xdea33040 r11=0x00000001
r12=0x00000018 r13=0xde12de98 r14=0x29aaaaab
Parking CPU 0 (Cell: "Banana-Pi")

#When removing "Jailhouse_MEM_IO_32" flag from "HDMI" region.

Unhandled data read at 0x1c20060(4)
FATAL: unhandled trap (exception class 0x24)
pc=0xc03f45e8 cpsr=0x60010093 hsr=0x93870007
r0=0x20010093 r1=0x00000191 r2=0x00000060 r3=0xe0009060
r4=0xc0b2b558 r5=0x00000100 r6=0x20010093 r7=0x00000000
r8=0x00000004 r9=0xde22c000 r10=0xc0b04c48 r11=0xde0ddc00
r12=0x00000000 r13=0xde22dc98 r14=0xc03f45d8
Parking CPU 0 (Cell: "Banana-Pi")

#When removing gic-demo's clock related region.

Unhandled data read at 0x1c20088(4)
FATAL: unhandled trap (exception class 0x24)
pc=0xc03f457c cpsr=0x60010093 hsr=0x93850007
r0=0xa0010093 r1=0x00000195 r2=0x00000088 r3=0xe0009088
r4=0xc0b2a3dc r5=0xde817cc0 r6=0xa0010093 r7=0x80000000
r8=0xde24a000 r9=0xc0b04c48 r10=0x00000008 r11=0xdf054880
r12=0x00000000 r13=0xde24be68 r14=0xc03f456c
Parking CPU 0 (Cell: "Banana-Pi")

#When removing gic-demo's CCU related region.

Unhandled data write at 0x7bfe0000(1)
```

```
FATAL: unhandled trap (exception class 0x24)
pc=0xc06c770c cpsr=0x20000013 hsr=0x9000004f
r0=0xe0184000 r1=0x00023080 r2=0x00002134 r3=0xea00000d
r4=0xea000005 r5=0xea000005 r6=0xea000005 r7=0xea000005
r8=0xea000005 r9=0xe0184000 r10=0x00023038 r11=0x00000001
r12=0xea00001a r13=0xde86fe4c r14=0xea000005
Parking CPU 0 (Cell: "Banana-Pi")

#When altering gic-demo's RAM addresses.

Unhandled data write at 0x7bf00000(1)
FATAL: unhandled trap (exception class 0x24)
pc=0xc06c770c cpsr=0x20000013 hsr=0x9000004f
r0=0xe0184000 r1=0x00023080 r2=0x00002134 r3=0xea00000d
r4=0xea000005 r5=0xea000005 r6=0xea000005 r7=0xea000005
r8=0xea000005 r9=0xe0184000 r10=0x00023038 r11=0x00000001
r12=0xea00001a r13=0xde34be4c r14=0xea000005
Parking CPU 0 (Cell: "Banana-Pi")


#When altering the following lines:

/* RAM */ {
        .phys_start = 0x40000000,
        .virt_start = 0x40000000,
        //.size = 0x3bf00000,
        .size = 0x1d1fffff,
        .flags = Jailhouse_MEM_READ | Jailhouse_MEM_WRITE |
                Jailhouse_MEM_EXECUTE,
```

```
}, <-------- bananapi.c

/* RAM */ {
        /*.phys_start = 0x7bfe0000,
        .virt_start = 0,
        .size = 0x00010000,*/
        .phys_start = 0x5d1fffff,
        .virt_start = 0,
        .size = 0x1d1ffffe,
        .flags = Jailhouse_MEM_READ | Jailhouse_MEM_WRITE |
        Jailhouse_MEM_EXECUTE | Jailhouse_MEM_LOADABLE,
}, <-------- bananapi-gic-demo.c

Adding virtual PCI device 00:00.0 to cell "Banana-Pi"
Page pool usage after late setup: mem 68/16362, remap 5/131072
FATAL: instruction abort at 0x5df6e9a4
FATAL: forbidden access (exception class 0x20)
pc=0xbf000ea8 cpsr=0x80030193 hsr=0x80000086
r0=0x00000000 r1=0x00000002 r2=0xf0000000 r3=0xf0003580
r4=0xbf006688 r5=0x00000000 r6=0xbf000e3c r7=0xf0000000
r8=0x00000000 r9=0xc088c16c r10=0xc0b6ee80 r11=0x00000001
r12=0xc0b01ed0 r13=0xc0b01ed0 r14=0xf0003934
Parking CPU 0 (Cell: "Banana-Pi")

#when altering the following lines:

/* RAM */ {
        .phys_start = 0x7A3EFFFF,
        .virt_start = 0,
```

```
        .size = 0x00010000,
        .flags = Jailhouse_MEM_READ | Jailhouse_MEM_WRITE |
                Jailhouse_MEM_EXECUTE | Jailhouse_MEM_LOADABLE,
}, <------- bananapi-gic-demo.c

Cell "bananapi-gic-demo" can be loaded
Started cell "bananapi-gic-demo"
Unhandled data read at 0x1c2006c(4)
FATAL: unhandled trap (exception class 0x24)
pc=0x00001aa2 cpsr=0x200001f3 hsr=0x91830007
r0=0x00000001 r1=0x00000000 r2=0x01c00705 r3=0xfe3dff94
r4=0x01c2006c r5=0x80000018 r6=0x000020d4 r7=0x00000017
r8=0x00000000 r9=0x00000000 r10=0x00000000 r11=0x00001fc2
r12=0x00000000 r13=0x00008f10 r14=0x00001aa1

#when altering the following lines:

/* CCU */ {
        .phys_start = 0x01C1FC00,
        .virt_start = 0x01C1FC00,
        .size = 0x400,
        .flags = Jailhouse_MEM_READ | Jailhouse_MEM_WRITE |
        Jailhouse_MEM_IO | Jailhouse_MEM_IO_32,
}, <------- bananapi-gic-demo.c

Cell "bananapi-gic-demo" can be loaded
Started cell "bananapi-gic-demo"
Unhandled data read at 0x1c2006c(4)
FATAL: unhandled trap (exception class 0x24)
```

```
pc=0x00001aa2 cpsr=0x200001f3 hsr=0x91830007
r0=0x00000001 r1=0x00000000 r2=0x01c00705 r3=0xfe3dff94
r4=0x01c2006c r5=0x80000018 r6=0x000020d4 r7=0x00000017
r8=0x00000000 r9=0x00000000 r10=0x00000000 r11=0x00001fc2
r12=0x00000000 r13=0x00008f10 r14=0x00001aa1
Parking CPU 1 (Cell: "bananapi-gic-demo")
```