bhyve for ARMv8

Nicolae-Alexandru Ivan
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
nicolae.ivan@stud.acs.upb.ro

January 16, 2016

Abstract

Insert abstract here.

1 Introduction

And there was light.

2 Virtualization

"Virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others."[14]

It implies having a machine on which the virtualization takes place, called a **host machine**, and a firmware or software, called a **hypervisor** or virtual machine manager, that creates the virtualized or **guest machine**.

The guest executes its code, but instead of running directly on the underlying hardware, it has to pass through the hypervisor, which may decide to run the instructions on the hardware or modify them beforehand.

2.1 Hypervisor types

Depending on how they function, there are two types of hypervisors, as classified in [13]:

- **Type-1** also called **native** is a hypervisor that runs directly on top of hardware, while the guest machines are run as processes
- **Type-2** is a hypervisor which runs inside an operating system

2.2 Types of virtualization

In order to perform efficient virtualization, most instructions must be executed directly by the hardware. Interpreting the instructions would incur a performance drop[11].

However, not all instructions can be directly executed. There are two types of such instructions[11] :

- **Control-sensitive instructions** which perform changes on hardware resources through various operations, e.g. I/O
- **Behaviour-sensitive instructions** which read the privilege level and might reveal to the guest that it is not running directly on hardware

Depending on how the framework handles sensitive instructions, there are two main types of virtualization[8]:

1. Full virtualization
2. Paravirtualization

2.2.1 Full Virtualization

This technique is composed of direct execution and binary translation. As seen in Figure 1, user code is executed directly, while guest OS requests are translated into instructions that modify the virtual hardware.

In this case, the guest OS does not know it is being virtualized. Thus, no modifications to the operating system are necessary. The hypervisor may cache the translated instructions for future use.

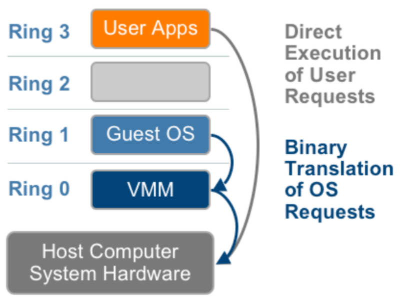
Ring 3
User Apps
Ring 2
Ring 1
Guest OS
Ring 0
VMM
Host Computer System Hardware
Direct Execution of User Requests
Binary Translation of OS Requests

Figure 1: Full Virtualization[8]

2.2.2 Paravirtulization

Paravirtualization means the hypervisor provides interfaces for operations that are not virtualizable. Because of this, guest OS instructions are now replaced by hypercalls which interact with the virtualization layer.

This approach incurs less performance penalty, but it requires modifications to the operating system which is to be virtualized. However, this also brings the advantage of being able to replace multiple consecutive sensitive instructions with a single hypercall, reducing the penalty induced by exceptions[11].

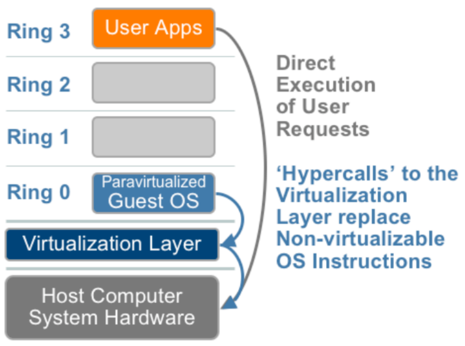
Ring 3
User Apps
Ring 2
Ring 1
Ring 0
Paravirtualized Guest OS
Virtualization Layer
Host Computer System Hardware
Direct Execution of User Requests
'Hypercalls' to the Virtualization Layer replace Non-virtualizable OS Instructions

Figure 2: Paravirtualization[8]

2.3 Hardware Assisted Virtualization

In 2006, Intel release its VT-x and AMD presented AMD-V, both introducing the notion of hardware assisted virtualization. The feature consists of a new root level of privilege in which the virtual machine manager runs. Sensitive calls trap to this level without translation or paravirtualization[8].

Memory is also virtualized in order to accommodate guest systems. This is done by an extra level of indirection in memory management. The memory management unit must map guest physical memory to host machine physical memory[2].

2.4 Virtualization on ARM

ARM is a 32-bit RISC architecture. It features one user mode and 6 kernel modes. A secure mode exists, which can be used to protect access to hardware resources and a monitor mode, which is responsible for switching between the secure and normal modes[15].

ARM virtualization works similarly to Intel and AMD, in the sense that it also introduces a new privilege level, namely the **hyp** mode. In order to virtual memory usage while in this mode, an additional memory management unit was introduced. An additional extension, called Large Physical Address Extension, was also added, extending the addressable physical memory space to 40 bits[2].

Additionally, ARM introduced a component called **virtual CPU** which allows virtual interrupts. Such interrupts can either be handled by the hypervisor, or linked directly to the physical interrupt[15].

2.5 Advantages of Virtualization for Embedded Systems

Advances in performance of embedded systems have made them a viable platform for virtualization. The main reason why virtualization on embedded systems is necessary is to

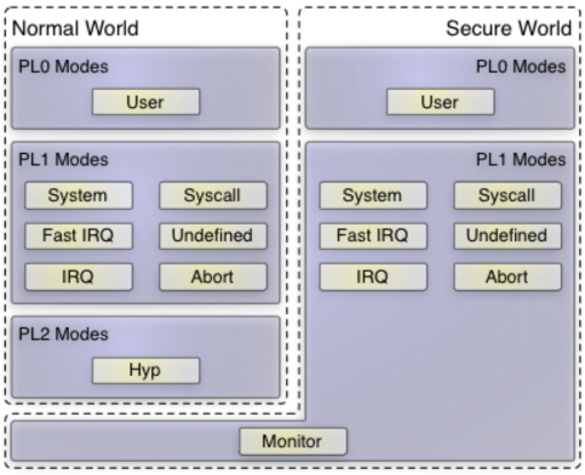

Figure 3: ARM privilege levels[2]

have multiple operating systems, each responsible for some subset of functions performed by the device.

Another important aspect of virtualization is the security benefits it brings. The isolation of one guest from another means that a vulnerability in one subsystem cannot lead to an exploit in another. This limits the possibilities a potential attacker has when trying to compromise a system[11].

The above mentioned isolation also results in the possibility of using software distributed under different license agreements in different guests, while having efficient communication between these facilitated by the hypervisor.

2.6 Limitations of Virtualization for Embedded Systems

While performance of embedded systems has increased considerably, complexity of software systems has increased as well. The fact that each guest runs a separate operating system renders the virtual machines quite expensive from a resource point of view.

Another issue with performance is brought on by the scheduling mechanism. Due to the encapsulation of virtualized systems, the hypervisor can only associate a priority for each guest system. The result is that a low priority task running inside a high priority machine will be scheduled before a high priority task inside a lower priority guest[11].

Although virtualization does bring benefits in terms of security, it has some drawbacks. The increased amount of code on which a guest operating system relies on in order to function equates to increased amount of potential vulnerabilities.

3 FreeBSD and bhyve

`BSD` stands for "Berkeley Software Distribution". It is derived from AT&T's Research UNIX operating system. In 1990, the BSD code was released without the proprietary AT&T code. Since the proprietary code contained a good portion of the kernel, it was not until 1992 that a complete operating system based on the BSD code was released - 386BSD. In 1993, the FreeBSD operating system split from the project[3].

3.1 FreeBSD vs Linux

While FreeBSD originated from the UNIX systems, Linux was built to be a similar alternative for these. Therefore, they share many mechanisms, tools and applications.

The most significat difference between the two consists in the licensing. Linux is licensed under GNU General Public License(GPL)[5]. This allows freedom to modify and redistribute the source code as long as it is also licensed under the GPL. FreeBSD is licensed under the BSD license, which is more permissive. It does not require derived work to maintain the license, but only to include a copy of the BSD license and original copyright[7].

Another difference consists of the available software. One aspect is availability of both packaged and sourced software. While most Linux distributions have little to no support for building software from source, FreeBSD provides an extensive collection of software source code which the user can customize and build. Conversely, while BSD maintainers are more conservative when modifying software packages, Linux distribution maintainers make modifications in order to improve component interconnection and management[1].

3.2 bhyve

`bhyve` is an abbreviation for BSD hypervisor. It is a type-2 hypervisor, similar to Linux KVM. As of FreeBSD 10.0, it is part of the base system. It support various guest systems, ranging from other BSD systems to Linux distributions[9].

Being a legacy-free hypervisor, it is reliant on the virtualization features of CPUs. It requires both CPU and memory virtualization technologies.

bhyve has several components[9][10]:

- **vmm.ko** - kernel module, manages VT-x stare, context switching, guest physical memory and other aspects
- **libvmmapi** - userland API
- **bhyveload** - userspace bootloader, creates vm and does initial setup, loads guest operating system
- **bhyve** - userspace run loop, emulates stdin/stdout, tap devices, block devices
- **bhyvectl** - utility that can modify virtual machine state; can also delete VMs

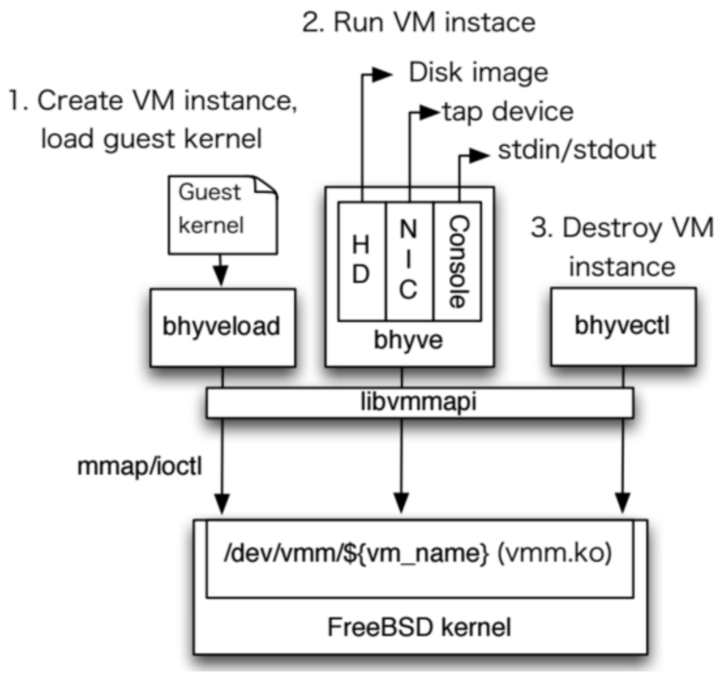

Figure 4: bhyve overview[9]

4 Experimental Setup and Results

Before actually beginning to work on porting bhyve to ARMv8 a working baseline to compare future results to was needed. To this end, an Arndale Board was used. The board is based on a Samsung Exynos5250 processor, which is part of the ARMv7 family. The two steps performed were:

- Run FreeBSD on the Arndale board
- Run a KVM guest on a Linux host running on the Arndale board

4.1 FreeBSD on Arndale

This step was done in order to better understand the process of running FreeBSD on an ARM device. It was done by following the documentation from the FreeBSD wiki[4]. The setup has two important components to build: the FreeBSD system and the bootloader.

Building FreeBSD is a straightforward process. After obtaining the FreeBSD sources, one must simply build the kernel toolchain, provide the proper configuration, after which the kernel and the system can be built.

```
# make TARGET_ARCH=armv6 kernel-toolchain
# make TARGET_ARCH=armv6 KERNCONF=ARNDALE buildkernel
# make TARGET_ARCH=armv6 buildworld
# make TARGET_ARCH=armv6 DESTDIR=/mnt installworld distribution
```

The last command installs the filesystem in the /mnt directory.

Building the bootloader is also a simple process that consists of obtaining the source code, configuring and building the software. The interesting part is preparing writing the bootloader to an SD card from which the board will boot. Three components are required:

- **The stage one binary** - also called BL1, which is proprietary and does the first steps of the booting process
- **The secondary program loader** - or SPL, which is also board specific, although built by the bootloader; this component has the role of loading the actual bootloader
- **The u-boot binary** - this is the actual bootloader; it provides the needed functionality to load the kernel

With some minor issues, this step was performed successfully.

4.2 KVM virtualization on Arndale

The goal of this step was to establish a working baseline. The approach used is described in the following guide[6].

There are many similarities to running FreeBSD on the Arndale board. Once more, the u-boot bootloader is used in order to load the kernel. However, in this case the kernel is a modified version of the 3.8 Linux kernel, with a patch set applied that enables KVM on ARM architecture. The filesystem for the host is built separately.

After booting the Linux host on the board, preparations for the running the guest systems must be made. The first part is to build `qemu`, which emulates devices and exposes KVM to userspace. Although a separate kernel has to be built for the guest system, the filesystem created for the host can be reused. Finally, the proper network configuration must be made in order to bridge the guest system network to the physical network.

This step proved to be more problematic than the previous. The main cause were modifications made in the repository for `qemu`, which resulted in attempting to compile multiple branches in order to obtain the proper binary. After several attempts, a decision was made to use the precompiled binary provided by the guide in order to save time. In the end, the process was successful.

5 Conclusion and Further Work

And that was it.

Acknowledgment

The authors would like to thank Mihai Carabaș for his support and dedication.

References

[1] A Comparative Introduction to FreeBSD for Linux Users. https://www.digitalocean.com/community/tutorials/a-comparative-introduction-to-freebsd-for-linux-users.

[2] An in-depth look into the ARM virtualization extensions. http://genode.org/documentation/articles/arm_virtualization.

[3] Explaining BSD. https://www.freebsd.org/doc/en_US.ISO8859-1/articles/explaining-bsd/index.html.

[4] FreeBSD on Arndale Board. https://wiki.freebsd.org/FreeBSD/arm/ArndaleBoard.

[5] GNU GENERAL PUBLIC LICENSE. http://www.gnu.org/licenses/gpl-3.0.en.html.

[6] KVM virtualization on Arndale development board. http://www.virtualopensystems.com/en/solutions/guides/kvm-virtualization-on-arndale/.

[7] The BSD 2-Clause License. http://opensource.org/licenses/bsd-license.php.

[8] Understanding Full Virtualization, Paravirtualization, and Hardware Assist, 2007.

[9] T. Asada. Introduction to bhyve. https://docs.google.com/viewer?url=http%3A%2F%2Fbhyvecon. to_bhyve.pdf.

[10] P. Grehan. Extending bhyve beyond FreeBD guests. http://people.freebsd.org/ grehan/talks/eurobsdcon_2013_bhyve.pdf.

[11] G. Heiser. Virtualization for embedded systems, 2007.

[12] M. T. Jones. Virtualization for embedded systems, 2011.

[13] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures, 1974.

[14] A. Singh. An introduction to virtualization. http://www.kernelthread.com/publications/virtualization/, 2004.

[15] P. Varanasi and G. Heiser. Hardware-Supported Virtualization on ARM.