Finalizing booting requirements for a guest running under bhyvearm

Nicolae-Alexandru Ivan, Mihai Carabas
Automatic Control and Computers Faculty
University POLITEHNICA of Bucharest
Emails: nicolae.ivan@stud.acs.upb.ro.ro, mihai.carabas@cs.pub.ro

***Abstract*—**

Keeping track of time is an invaluable resource in modern software systems. The vast majority of existing CPUs posses various clocks and timers in order to accommodate time related mechanisms required by software. These same needs apply to virtualized environments, where the guest operating system uses time based events. To this end, a virtualized timer is required. This research project describes implementing such a timer in FreeBSD for the ARMv7 architecture.

***Index Terms*—FreeBSD, bhyve, hypervisor, ARMv7, GIC, vGIC, interrupts, Cubieboard2, Allwinner A20**

I. Introduction

In the current stage, a guest running under the ARM FreeBSD hypervisor (bhyve-arm) isn't able to boot due to lack of a virtual timer implementation and issues with VFP (vector floating point) and WFI instruction. This paper will tackle mainly the timer virtualization and also the remaining issues to boot a guest.

Timed events are a core element of many software systems. Their utility ranges from pre-empting processes while in kernel space to scheduling events in high level programming in user space. It is clear that these types of functionality are also desirable when running software in a virtualized environment.

The need for keeping time has brought about the introduction of new timer hardware, such as the Programmable Interval Timer(PIT), the Real Time Clock(RTC), the Advanced Configuration and Power Interface(ACPI) and the High Precision Event Timer(HPET), each with their own utility.

The above mentioned as well as most other hardware timers have the same basic functionality, as described by Figure 1. An oscillator produces a precise frequency signal. Each cycle of the oscillator updates the counter. When reaching a specific value, it generates an output signal. Usually, this signal is a interrupt that lets the CPU know that some amount of time has passed. Depending on the specific type of timer, there may be additional components.[9]

In the next sections, the following topics are discussed: section two describes the state of the art - how other systems virtualize timers, section three goes into detail concerning the implementation, and the final section concludes with results and plans for further development.

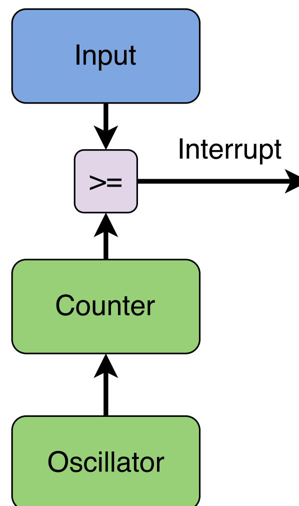

Figure 1. Timer functionality

II. State of the Art

Timer virtualization depends heavily on the underlying architecture. Some hardware platforms have virtualization extensions meant to facilitate guest interaction with hardware features as the timer. Other platforms lack such support and, in this case, the hardware device must be emulated entirely.

A. Platforms without Timer Virtualization Extension

Platforms such as the widely used x86 have no support for timer virtualization. This means that the virtualization infrastructure must emulate all access to the timer, as well as the all interrupts produced by it. Various techniques can be used to achieve this, as follows.

1) VMware: VMware uses proprietary technology, allowing the guest operating system to fall behind and catch up as necessary, without losing functionality. The time which is visible to guest systems is called apparent time. An in-depth description of the functioning of each of the virtual timers present in VMware can be found in the following VMware paper.[9]

2) Xen: Xen approaches this issue differently. It uses paravirtualization - the guest is aware that it is functioning inside

a virtualized environment. The guest kernel is modified to contain a Clock Event Device which schedules events trough the use of hypercalls. The resulting interrupt will be caught in the hypervisor and delivered to the appropriate virtual machine.[3]

3) Linux KVM: KVM supports both fully virtualized and paravirtualized virtual timers. The fully emulated virtual timer uses high resolution timers in the Linux kernel to keep track of guest timer events. When a timer is fired inside KVM, it is flagged that timers have to be taken into consideration upon entering the guest.

As an alternative, KVM also provides a paravirtualized kvm-clock which may be used as a clocksource by guest operating systems.[7]

4) bhyve: bhyve, the FreeBSD hypervisor, supports fully emulated timers for platforms that have no hardware support for virtual timers. Similar to KVM, the bhyve implementation uses the existing high performance event timers from kernel space to handle guest timer events.

B. ARMv7

ARMv7 offers hardware support for virtual timers, thus rendering both the performance penalty on the guest and the amount of work required inside the hypervisor minimal. The guest is allowed to interact directly with the hardware, with no intervention from the hypervisor. Still, the virtualization infrastructure must perform certain operations to ensure correct functionality of the guest.

KVM has a working implementation of virtual timers for ARM. This implementation was used as a reference when implementing the FreeBSD virtual timer.[6]

III. IMPLEMENTATION

Before discussing the actual implementation, the architecture of ARMv7 timer is presented and, also, a very high level overview of the Generic Interrupt Controller is made. These are necessary in order to understand the implementation. Additionally, a summary of encountered issues is made in the ending of this section.

A. ARMv7 Generic Timer Architecture

The Generic Timer present on the ARMv7 platform is a standardized timer which can be used as a system clock. Aside from the usual counter, which in this case is referred to as physical counter, the ARM Generic Timer may also contains a virtual counter, which can be used by virtual machines for time-keeping purposes. The physical counter is at least 56 bits wide and updates at a constant frequency in the range 1-50MHz. The virtual counter holds the value of the physical counter minus a 64-bit offset.

An implementation of the Generic Timer with Virtualization Extension provides four timers per CPU[5]:

- Non-secure PL1 physical timer
- Secure PL1 physical timer
- Non-secure PL2 physical timer
- Virtual timer

Each of the above provides an interrupt signal. Additionally, each has a set of three registers: a CompareValue register - which is a 64-bit unsigned upcounter, a TimerValue register - which is a 32-bit signed downcounter, and a 32-bit Control register.[5]

B. Virtual Generic Interrupt Controller

ARMv7 platforms use a Generic Interrupt Controller in order to manage interrupts. The GIC keeps track of which interrupts are enabled, prioritizes incoming interrupts and delivers them to the appropriate CPU.[4] Since there is no hardware support for a virtual GIC, it must be emulated by the hypervisor. This means that any access to the GIC from within the guest, as well as any interrupt that should be delivered to the guest must pass through the emulated controller, also called vGIC.

C. Virtual Timer Implementation

Before describing the implementation, the table below describes the registers used.[5]

Name	Description
CNTV_CTL	Virtual Timer Control register. Used by the guest to interact with the timer hardware
CNTV_CVAL	Virtual Timer CompareValue register
CNTHCTL	Controls access to the physical registers. In particular, the PL1PCTEN and PL1PCEN are used to disable access to the physical timer registers
CNTVOFF	Virtual Offset register - specifies value to be subtracted from physical counter in order to obtain virtual counter

Figure 2 constitutes an overview of the workflow.

The workflow of the virtualization process is as follows:

1) At guest initialization (state 0), the CNTVOFF register is initialized with the current value of the physical timer, rendering the virtual counter 0 for the newly created virtual machine
2) Before entering the guest (transition from state 1 to state 2), the hypervisor internal state for the virtual timer is checked in order to determine whether any interrupts

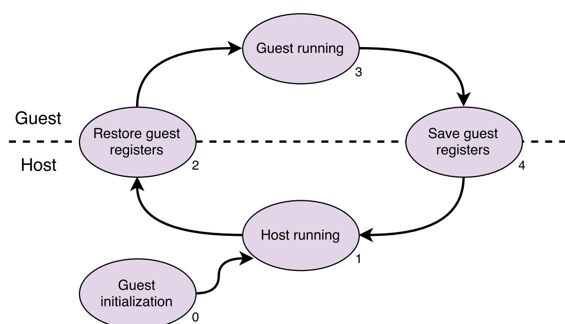

Figure 2. Virtual Timer Workflow

should have been triggered by the timer and need to be injected by the vGIC
3) Upon entering the guest (state 2), the hypervisor enables the virtual timer if necessary, disables access to the physical timer, and restores the CNTVOFF register for the current virtual machine, as well as restoring CNTV_CVAL and CNTV_CTL
4) While running, the guest accesses the virtual timer with no intervention from the hypervisor (state 3); any interrupts triggered here are sent to the vGIC, which will inject them accordingly
5) When exiting the guest, the CNTV_CVAL and CNTV_CTL registers are saved (state 4) and the hypervisor internal state is updated (transition from state 4 to state 1
6) The host continues to use the physical timer until the guest is run again, when the process resumes at step 2

Notice that only the CompareValue register is saved. This is because both read and write operations on the TimerValue register are translated into reads and writes on the CompareValue.

The implementation of the aforementioned flow is relatively straight forward. The internal structure for a virtual machine is used to memorize whether the virtual timer is enabled in the respective guest and to store the value of CNTVOFF. Similarly, the values for CNTV_CVAL and CNTV_CTL are stored within a per-cpu structure.

One noteworthy aspect is determining whether an interrupt needs to be injected upon re-entering a guest. When syncing the internal bhyve state with the hardware state, it is first checked whether the counter has reached the CompareValue already. If so, the interrupt is injected on the next guest entry. Otherwise, the remaining number of cycles is calculated and a callout[1] event is scheduled. If the callout is executed before running the virtual machine again, the interrupt is injected. In the case where the guest is executed again before the callout, the latter is simply cancelled.

D. Encounterd Issues

One major issue encountered was caused by desynchronising a number of assembly symbols from the C code. The saving and restoring of the timer registers is done directly in assembly code, in which the respective symbols are used to calculate the memory locations of various fields from the hypervisor internal structures. The incorrect offsets used in these calculations corrupted other internal fields and eventually caused the virtual machine to crash.

Another blocking issue, which at the time of writing this paper has not yet been solved, concerns the vGIC. Although the interrupt injection flow is triggered and executes correctly, the guest does not receive the interrupt. Most likely, not all necessary guest registers are updated.

E. Vector Floating Point

Another kernel subsystem which needs to be initialized as part of the boot process is the Vector Floating Point (VFP) module.

Vector Floating Point is a coprocessor which supports arithmetic operations on floating point numbers. It has a set of control registers, as well as a bank of registers which are used to store operands.

When changing worlds between host and guest, the state of this coprocessor needs to be saved. However, due to the limited use of the VFP architecture, it can be assumed that saving/restoring its state is not required at every entry/exit. Instead, the trapping mechanism is once again used. When an access to the floating point coprocessor occurs, the current state is saved and the saved state for the guest is loaded. When returning from the guest, the reverse operation is performed.

F. WFI Handler

At the very end of the boot process, FreeBSD executes a wait for interrupt (WFI) instruction. This blocks the execution until an interrupt is received by the CPU. Normally, this happens very quickly due to the timer interrupt arriving.

On the virtualized system, this instruction requires special handling. The simple method is to ignore the instruction and continue execution. However, taking into account that both the virtual interrupt controller and virtual timer were already implemented, there was no reason to choose this option.

Therefore, the hardware behaviour was mimicked as closely as possible: the virtual cpu was set to sleep and wake up periodically in order to check whether any new interrupts had arrived. While the virtual cpu is asleep, the hypervisor yields control of the cpu so that other processes may execute[2]. Upon receiving an interrupt, the virtual cpu resumes normal execution.

IV. RESULTS

Timed events are a core element of many software systems. Their utility ranges from preempting processes while in kernel space to scheduling events in high level programming in user space. It is clear that these types of functionality are also desirable when running software in a virtualized environment.

Through the virtualization of the ARM Generic Timer described in the previous section, the guest operating system is able to achieve time-keeping and scheduling functionality close to what a system running directly on the underlying hardware.

Therefore, a key part of the operating system has been implemented with behavior which closely mimics that of the physical component.

The final result is to boot a minimal FreeBSD guest. Below is the output for each of the steps of running a virtual machine.

First, the kernel module needs to be loaded.

Listing 1. Loading the vmm module

```
# kldload boot/kernel/vmm.ko
vgic0: <Virtual Generic Interrupt
    Controller> on gic0
vgic0: Cannot setup Maintenance Interrupt
    . Disabling Hyp-Mode... 0
```

There is still an unresolved issue regarding registering the maintenance interrupt. The issue is circumvented by executing the handler upon switching from guest to host context. Also, until the issue is resolved, hyp-mode is not disabled by this failure, as it would prevent the rest of the guest execution.

The second step is creating the virtual machine. This is done using the `bhyveload` utility. At the end of this step, the virtual machine is set up and the guest code is loaded into memory.

Listing 2. Creating the virtual machine

```
# bhyveload -k kernel.bin test
lpae_vmmmap_set n: 4096 27904
lpae_vmmmap_set n: 4096 23808
lpae_vmmmap_set n: 4096 19712
lpae_vmmmap_set n: 4096 15616
lpae_vmmmap_set n: 4096 11520
lpae_vmmmap_set n: 4096 7424
lpae_vmmmap_set n: 4096 3328
lpae_vmmmap_set n: 4096 4096
```

Finally, the `bhyve` utility is used to commence the execution of the guest.

Listing 3. Booting the virtual machine

```
# bhyve -b test
initarm: console initialized
 arg1 kmdp = 0xc170fbd0
 boothowto = 0x00000000
 dtbp = 0xc1654568
 lastaddr1: 0xc1734000
loader passed (static) kenv:
 no env, null ptr
KDB: debugger backends: ddb
KDB: current backend: ddb
Copyright (c) 1992-2017 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989,
    1991, 1992, 1993, 1994
        The Regents of the University of California.
             All rights reserved.
FreeBSD is a registered trademark of The FreeBSD
    Foundation.
FreeBSD 12.0-CURRENT #0 52c583799(projects/bhyvearm)
    -dirty: Fri Jul 14 20:23:04 EEST 2017
    root@bsd:/usr/obj/arm.armv6/root/git-bhyvearm/
        sys/FVP_VE_CORTEX_A15x1_GUEST arm
FreeBSD clang version 4.0.0 (tags/RELEASE_400/final
    297347) (based on LLVM 4.0.0)
WARNING: WITNESS option enabled, expect reduced
    performance.
WARNING: DIAGNOSTIC option enabled, expect reduced
    performance.
CPU: ARM Cortex-A15 r2p0 (ECO: 0x00010000)
CPU Features:
  Multiprocessing, Thumb2, Security, Virtualization,
     Generic Timer, VMSAv7,
  PXN, LPAE, Coherent Walk
Optional instructions:
  SDIV/UDIV, UMULL, SMULL, SIMD(ext)
LoUU:2 LoC:3 LoUIS:2
Cache level 1:
 32KB/64B 2-way data cache WB Read-Alloc Write-Alloc
 32KB/64B 2-way instruction cache Read-Alloc
Cache level 2:
 512KB/64B 16-way unified cache WB Read-Alloc Write-
     Alloc
real memory  = 134217728 (128 MB)
avail memory = 101703680 (96 MB)
arc4random: no preloaded entropy cache
random: entropy device external interface
ofwbus0: <Open Firmware Device Tree>
gic0: <ARM Generic Interrupt Controller> mem 0
    x2c001000-0x2c001fff,0x2c002000-0x2c003fff on
    ofwbus0
gic0: Cannot find Virtual Interface Control
    Registers. Disabling Hyp-Mode...
gic0: pn 0xe8, arch 0x0, rev 0xe, implementer 0x800
    irqs 128
intr_pic_register(): PIC 0xc2207100 registered for
    gic0 <dev 0xc2633b80, xref 1>
intr_pic_claim_root(): irq root set to gic0
generic_timer0: <ARMv7 Generic Timer> irq 0,1,2,3 on
     ofwbus0
Timecounter "ARM MPCore Timecounter" frequency
    24000000 Hz quality 1000
Event timer "ARM MPCore Eventtimer" frequency
    24000000 Hz quality 1000
cpulist0: <Open Firmware CPU Group> on ofwbus0
cpu0: <Open Firmware CPU> on cpulist0
cryptosoft0: <software crypto>
NULL mp in getnewvnode(9), tag crossmp
Timecounters tick every 1.000 msec
WARNING: WITNESS option enabled, expect reduced
    performance.
```

```
WARNING: DIAGNOSTIC option enabled, expect reduced
    performance.
md0: Embedded image 18251776 bytes at 0xc0475f94
Trying to mount root from ufs:/dev/md0 []...
warning: no time-of-day clock registered, system
    time will not be set accurately
Jul 14 17:00:51 init: login_getclass: unknown class
    'daemon'
sh: cannot open /etc/rc: No such file or directory
Enter full pathname of shell or RETURN for /bin/sh:
    random: unblocking device.

Expensive timeout(9) function: 0xc04294b0(0xc2641600
    ) 0.022796458 s
Cannot read termcap database;
using dumb terminal settings.
#
#
```

V. Conclusions and Further Work

The project achieved its goal of completely booting a minimal FreeBSD guest operating system running inside bhyve. In order to reach this objective, a number of mechanisms were implemented. These include: the virtual generic interrupt controller, virtual timer, support for guest vector floating point operations and other less notable changes. This paper tackled especially the virtual timer.

A. Further Work

The bsd kernel contains multiple mechanisms for scheduling events to be run at a future time. The current callout system may be replaced with another mechanism if the latter offers better performance or improves code maintainability.

There are a number of directions which can be pursued for long term future development. These include: adding more hypervisor components to support more virtualization features, implementing support for symmetric multiprocessor (SMP) enabled guests.

At the time of writing this paper, there is an ongoing process with the FreeBSD community to integrate the changes proposed by the current project into the upstream repository. Alexandru Elisei started to create intermediarry patches to split-out on arch-dependent/independent the current bhyve code (libvmmapi, bhyve and vmm module). After this is done, we can create review requests for the actual ARM code.

References

[1] FreeBSD callouts. https://www.freebsd.org/cgi/man.cgi?query=callout&apropos&sektion

[2] FreeBSD msleep. https://www.freebsd.org/cgi/man.cgi?query=msleep&apropos=0&sektio
&manpath=FreeBSD+7.0-RELEASE&format=html.

[3] B. Adamczyk and A. Chydzinski. Achieving High Resolution Timer Events in Virtualized Environment. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4503740/, 2015.

[4] ARM. Cortex-A15 Technical Reference Manual, 2011.

[5] ARM. ARM Architecture Reference Manual - ARMv7-A and ARMv7-R edition, 2014.

[6] C. Dall and J. Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. https://www.cs.columbia.edu/ nieh/pubs/asplos2014_kvmarm.pdf, 2014.

[7] T. Gleixner and D. Niehaus. Hrtimers and Beyond: Transforming the Linux Time Subsystems. https://www.landley.net/kdocs/ols/2006/ols2006v1-pages-333-346.pdf, 2006.

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

[9] VMware. Timekeeping in VMware Virtual Machines. https://www.vmware.com/files/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf, 2011.