Linux Riscv Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

BOOT IMAGE HEADER IN RISC-V LINUX

Author
Atish Patra <atish.patra@wdc.com>

Date
20 May 2019

This document only describes the boot image header details for RISC-V Linux.

TODO:
Write a complete booting guide.

The following 64-byte header is present in decompressed Linux kernel image:

```
u32 code0;                  /* Executable code */
u32 code1;                  /* Executable code */
u64 text_offset;            /* Image load offset, little endian */
u64 image_size;             /* Effective Image size, little endian */
u64 flags;                  /* kernel flags, little endian */
u32 version;                /* Version of this header */
u32 res1 = 0;               /* Reserved */
u64 res2 = 0;               /* Reserved */
u64 magic = 0x5643534952; /* Magic number, little endian, "RISCV" */
u32 magic2 = 0x05435352;  /* Magic number 2, little endian, "RSC\x05
↪" */
u32 res3;                   /* Reserved for PE COFF offset */
```

This header format is compliant with PE/COFF header and largely inspired from ARM64 header. Thus, both ARM64 & RISC-V header can be combined into one common header in future.

1.1 Notes

- This header can also be reused to support EFI stub for RISC-V in future. EFI specification needs PE/COFF image header in the beginning of the kernel image in order to load it as an EFI application. In order to support EFI stub, code0 should be replaced with "MZ" magic string and res3(at offset 0x3c) should point to the rest of the PE/COFF header.
- version field indicate header version number

Bits 0:15	Minor version
Bits 16:31	Major version

This preserves compatibility across newer and older version of the header. The current version is defined as 0.2.

- The “magic” field is deprecated as of version 0.2. In a future release, it may be removed. This originally should have matched up with the ARM64 header “magic” field, but unfortunately does not. The “magic2” field replaces it, matching up with the ARM64 header.
- In current header, the flags field has only one field.

Bit 0	Kernel endianness. 1 if BE, 0 if LE.

- Image size is mandatory for boot loader to load kernel image. Booting will fail otherwise.

CHAPTER

TWO

SUPPORTING PMUS ON RISC-V PLATFORMS

Alan Kao <alankao@andestech.com>, Mar 2018

2.1 Introduction

As of this writing, perf_event-related features mentioned in The RISC-V ISA Privileged Version 1.10 are as follows: (please check the manual for more details)

- [m|s]counteren
- mcycle[h], cycle[h]
- minstret[h], instret[h]
- mhpeventx, mhpcounterx[h]

With such function set only, porting perf would require a lot of work, due to the lack of the following general architectural performance monitoring features:

- Enabling/Disabling counters Counters are just free-running all the time in our case.
- Interrupt caused by counter overflow No such feature in the spec.
- Interrupt indicator It is not possible to have many interrupt ports for all counters, so an interrupt indicator is required for software to tell which counter has just overflowed.
- Writing to counters There will be an SBI to support this since the kernel cannot modify the counters [1]. Alternatively, some vendor considers to implement hardware-extension for M-S-U model machines to write counters directly.

This document aims to provide developers a quick guide on supporting their PMUs in the kernel. The following sections briefly explain perf' mechanism and todos.

You may check previous discussions here [1][2]. Also, it might be helpful to check the appendix for related kernel structures.

2.2 1. Initialization

riscv_pmu is a global pointer of type *struct riscv_pmu*, which contains various methods according to perf's internal convention and PMU-specific parameters. One should declare such instance to represent the PMU. By default, *riscv_pmu* points to a constant structure *riscv_base_pmu*, which has very basic support to a baseline QEMU model.

Then he/she can either assign the instance's pointer to *riscv_pmu* so that the minimal and already-implemented logic can be leveraged, or invent his/her own *riscv_init_platform_pmu* implementation.

In other words, existing sources of *riscv_base_pmu* merely provide a reference implementation. Developers can flexibly decide how many parts they can leverage, and in the most extreme case, they can customize every function according to their needs.

2.3 2. Event Initialization

When a user launches a perf command to monitor some events, it is first interpreted by the userspace perf tool into multiple *perf_event_open* system calls, and then each of them calls to the body of *event_init* member function that was assigned in the previous step. In *riscv_base_pmu*'s case, it is *riscv_event_init*.

The main purpose of this function is to translate the event provided by user into bitmap, so that HW-related control registers or counters can directly be manipulated. The translation is based on the mappings and methods provided in *riscv_pmu*.

Note that some features can be done in this stage as well:

(1) interrupt setting, which is stated in the next section;

(2) privilege level setting (user space only, kernel space only, both);

(3) destructor setting. Normally it is sufficient to apply *riscv_destroy_event*;

(4) tweaks for non-sampling events, which will be utilized by functions such as *perf_adjust_period*, usually something like the follows:

```
if (!is_sampling_event(event)) {
        hwc->sample_period = x86_pmu.max_period;
        hwc->last_period = hwc->sample_period;
        local64_set(&hwc->period_left, hwc->sample_period);
}
```

In the case of *riscv_base_pmu*, only (3) is provided for now.

2.4 3. Interrupt

3.1. Interrupt Initialization

This often occurs at the beginning of the *event_init* method. In common practice, this should be a code segment like:

```
int x86_reserve_hardware(void)
{
      int err = 0;

      if (!atomic_inc_not_zero(&pmc_refcount)) {
              mutex_lock(&pmc_reserve_mutex);
              if (atomic_read(&pmc_refcount) == 0) {
                      if (!reserve_pmc_hardware())
                              err = -EBUSY;
                      else
                              reserve_ds_buffers();
              }
              if (!err)
                      atomic_inc(&pmc_refcount);
              mutex_unlock(&pmc_reserve_mutex);
      }

      return err;
}
```

And the magic is in *reserve_pmc_hardware*, which usually does atomic operations to make implemented IRQ accessible from some global function pointer. *release_pmc_hardware* serves the opposite purpose, and it is used in event destructors mentioned in previous section.

(Note: From the implementations in all the architectures, the *reserve/release* pair are always IRQ settings, so the *pmc_hardware* seems somehow misleading. It does NOT deal with the binding between an event and a physical counter, which will be introduced in the next section.)

3.2. IRQ Structure

Basically, a IRQ runs the following pseudo code:

```
for each hardware counter that triggered this overflow

    get the event of this counter

    // following two steps are defined as *read()*,
    // check the section Reading/Writing Counters for details.
    count the delta value since previous interrupt
    update the event->count (# event occurs) by adding delta, and
               event->hw.period_left by subtracting delta

    if the event overflows
```

(continues on next page)

(continued from previous page)

```
        sample data
        set the counter appropriately for the next overflow

        if the event overflows again
            too frequently, throttle this event
        fi
    fi

end for
```

However as of this writing, none of the RISC-V implementations have designed an interrupt for perf, so the details are to be completed in the future.

2.5 4. Reading/Writing Counters

They seem symmetric but perf treats them quite differently. For reading, there is a *read* interface in *struct pmu*, but it serves more than just reading. According to the context, the *read* function not only reads the content of the counter (event->count), but also updates the left period to the next interrupt (event->hw.period_left).

But the core of perf does not need direct write to counters. Writing counters is hidden behind the abstraction of 1) *pmu->start*, literally start counting so one has to set the counter to a good value for the next interrupt; 2) inside the IRQ it should set the counter to the same resonable value.

Reading is not a problem in RISC-V but writing would need some effort, since counters are not allowed to be written by S-mode.

2.6 5. add()/del()/start()/stop()

Basic idea: add()/del() adds/deletes events to/from a PMU, and start()/stop() starts/stop the counter of some event in the PMU. All of them take the same arguments: *struct perf_event *event* and *int flag*.

Consider perf as a state machine, then you will find that these functions serve as the state transition process between those states. Three states (event->hw.state) are defined:

- PERF_HES_STOPPED: the counter is stopped
- PERF_HES_UPTODATE: the event->count is up-to-date
- PERF_HES_ARCH: arch-dependent usage …we don’t need this for now

A normal flow of these state transitions are as follows:

- A user launches a perf event, resulting in calling to *event_init*.
- When being context-switched in, *add* is called by the perf core, with a flag PERF_EF_START, which means that the event should be started after it is added. At this stage, a general event is bound to a physical counter, if any. The

state changes to PERF_HES_STOPPED and PERF_HES_UPTODATE, because it is now stopped, and the (software) event count does not need updating.

  - *start* is then called, and the counter is enabled. With flag PERF_EF_RELOAD, it writes an appropriate value to the counter (check previous section for detail). Nothing is written if the flag does not contain PERF_EF_RELOAD. The state now is reset to none, because it is neither stopped nor updated (the counting already started)

- When being context-switched out, *del* is called. It then checks out all the events in the PMU and calls *stop* to update their counts.
  - *stop* is called by *del* and the perf core with flag PERF_EF_UPDATE, and it often shares the same subroutine as *read* with the same logic. The state changes to PERF_HES_STOPPED and PERF_HES_UPTODATE, again.
  - Life cycle of these two pairs: *add* and *del* are called repeatedly as tasks switch in-and-out; *start* and *stop* is also called when the perf core needs a quick stop-and-start, for instance, when the interrupt period is being adjusted.

Current implementation is sufficient for now and can be easily extended to features in the future.

2.7 A. Related Structures

- struct pmu: include/linux/perf_event.h
- struct riscv_pmu: arch/riscv/include/asm/perf_event.h

  Both structures are designed to be read-only.

  struct pmu defines some function pointer interfaces, and most of them take *struct perf_event* as a main argument, dealing with perf events according to perf' s internal state machine (check kernel/events/core.c for details).

  struct riscv_pmu defines PMU-specific parameters. The naming follows the convention of all other architectures.

- struct perf_event: include/linux/perf_event.h
- struct hw_perf_event

  The generic structure that represents perf events, and the hardware-related details.

- struct riscv_hw_events: arch/riscv/include/asm/perf_event.h

  The structure that holds the status of events, has two fixed members: the number of events and the array of the events.

2.8 References

[1] https://github.com/riscv/riscv-linux/pull/124

[2] https://groups.google.com/a/groups.riscv.org/forum/#!topic/sw-dev/f19TmCNP6yA

CHAPTER

THREE

ARCH/RISCV MAINTENANCE GUIDELINES FOR DEVELOPERS

3.1 Overview

The RISC-V instruction set architecture is developed in the open: in-progress drafts are available for all to review and to experiment with implementations. New module or extension drafts can change during the development process - sometimes in ways that are incompatible with previous drafts. This flexibility can present a challenge for RISC-V Linux maintenance. Linux maintainers disapprove of churn, and the Linux development process prefers well-reviewed and tested code over experimental code. We wish to extend these same principles to the RISC-V-related code that will be accepted for inclusion in the kernel.

3.2 Submit Checklist Addendum

We'll only accept patches for new modules or extensions if the specifications for those modules or extensions are listed as being "Frozen" or "Ratified" by the RISC-V Foundation. (Developers may, of course, maintain their own Linux kernel trees that contain code for any draft extensions that they wish.)

Additionally, the RISC-V specification allows implementors to create their own custom extensions. These custom extensions aren't required to go through any review or ratification process by the RISC-V Foundation. To avoid the maintenance complexity and potential performance impact of adding kernel code for implementor-specific RISC-V extensions, we'll only to accept patches for extensions that have been officially frozen or ratified by the RISC-V Foundation. (Implementors, may, of course, maintain their own Linux kernel trees containing code for any custom extensions that they wish.)