Harman

Device Virtualization for Connected Vehicles

CVM2RB Virtio Devices Programmer's Guide

Software Version 12.0

Doc. Rev. 3.0.8-sqy / None

Reference: None

Date: [Publish Date]

Table of Contents

1 Integrating Virtio Device in CVM2RB 4
1.1 Extending the Command Line 4
1.2 Instantiating Virtio Devices 4
2 Implementation of a Virtio Device 6
2.1 The VirtioDevice Trait 6
2.2 Activating the Device 8
2.3 The Main Device Thread 9
2.4 Processing the Virtqueues 11
2.5 Miscellaneous Implementation Details 12
2.5.1 Interrupt resampling events 13
2.5.2 Resettable virtio devices 13

1 Integrating Virtio Devices in CVM2RB

The set of virtio devices enabled in a specific instance of the cvm2rb process depends on how cvm2rb was invoked.

When cvm2rb starts, it parses the command line and builds an internal representation which takes the form of a `Config` object. The `Config` type is defined in the `cvm2rb/src/config.rs` source file. It contains all the information required to determine what devices should be enabled in the current configuration as well as the parameters to configure these devices (e.g. the disk image used by a `virtio-block` device).

The following sections provide developers with guidelines for:

- Extending cvm2rb's command line with additional parameters.
- Integrating a new virtio device in cvm2rb by instantiating it during cvm2rb startup.

1.1 Extending the Command Line

The list of all arguments supported by cvm2rb is defined by the `arguments` array created in the `parse` function of the `cvm2rb/src/cmdline.rs` source file. The definition of existing arguments shall be treated as examples to derive new definitions.

Extending the `arguments` array is enough for cvm2rb to accept a new argument when parsing the command line but we still need to define what happens when this argument is recognized.

For every valid argument identified on the command line, cvm2rb invokes the `set_argument` function defined in the `cvm2rb/src/cmdline.rs` file. This allows the developer to further parse the optional argument value and record the result in the `Config` object. For this purpose, the `Config` type (see `cvm2rb/src/config.rs`) may need to be extended to include additional configuration information.

Please refer to the current implementation of the `set_argument` function for some advanced examples.

1.2 Instantiating Virtio Devices

The code to setup the virtual devices emulated by cvm2rb is located in the `cvm2rb/src/setup.rs` source file. In particular, the `create_virtio_devices` function is responsible for instantiating all the virtio devices required by the configuration.

A common practice in cvm2rb is to define a device-specific creation function that deals with the details of instantiating a particular virtio device. All these device creation functions serve as helpers for the generic `create_virtio_devices` function.

The `virtio-rng` device creation function is included below.

```
fn create_rng_device() -> DeviceResult {
    let dev =
        virtio::Rng::new(
            virtio::base_features(ProtectionType::Unprotected)
        ).map_err(Error::RngDeviceNew)?;

    Ok(VirtioDeviceStub {
        dev: Box::new(dev),
    })
}
```

The main responsibility of this function is to create a `virtio-rng` device object. This is achieved by invoking the constructor for the corresponding type (i.e. the `virtio::Rng::new` function). The function then wraps the virtio device into a `VirtioDeviceStub` object and returns it.

A more advanced virtio device than `virtio-rng` would likely require extra configuration information to instantiate the device. For instance, a `virtio-net` device needs a TAP interface, a `virtio-block` device needs a disk image…

All these configuration parameters typically come from cvm2rb's command line, and the device-specific setup functions can retrieve these parameters from the `Config` object. Please refer to the `create_block_device` or `create_tap_net_device` functions for concrete examples.

The `create_virtio_devices` function collects all the virtio devices created by the device-specific creation functions and assemble them into a `Vector` object. The following code shows an extract from the `create_virtio_devices` function implementation.

```
for dev_path in &cfg.virtio_input_evdevs {
    devs.push(create_vinput_device(dev_path)?);
}
```

Here, the list of input devices specified when starting cvm2rb is retrieved from the configuration and the corresponding `virtio-input` devices are created. cvm2rb will later connect all the virtio devices returned by the `create_virtio_devices` function to the emulated PCI bus.

To put it in a nutshell, after the developer has implemented a new device in cvm2rb, enabling this device requires:

- Defining a new command line argument to enable this device and extend the `Config` object with additional parameters for this device.
- Adding a device-specific creation function as described in this section.
- Modifying the `create_virtio_devices` function to invoke this device-specific creation function when the device is enabled in the current configuration.

2 Implementation of a Virtio Device

This chapter presents the APIs offered by the cvm2rb virtio framework to virtio devices developers. We discuss cvm2rb's virtio devices execution model and we present the common patterns used when developing these devices.

To support this discussion, we use code examples taken from cvm2rb `virtio-rng` device implementation. The reader can find the entire source code for this device in the `devices/src/virtio/rng.rs` file of cvm2rb source tree.

2.1 The VirtioDevice Trait

A virtio device running in cvm2rb is represented by an object that implements the VirtioDevice trait. This trait is the main interface between the generic parts of the virtio framework and the devices themselves.

The virtio framework handles common operations on the virtio devices such as the virtio PCI transport layer and relies on the VirtioDevice trait for device-specific operations.

As part of implementing the VirtioDevice trait, the device must provide an implementation for the following methods.

- The `device_type` method returns the virtio device ID provided to the guest when probing the emulated PCI bus to discover available virtio devices.

  Note that when using PCI transport this information is encoded in the PCI device ID.

  ```
  fn device_type(&self) -> u32 {
      TYPE_RNG
  }
  ```

- The `queue_max_sizes` method returns a slice object containing the maximum queue sizes for all the virtqueues of this virtio device. The slice must have as many elements as the number of virtqueues used by the device.

  The example below shows the `queue_max_sizes` method for the `virtio-rng` device which only supports a single virtqueue of at most 256 descriptors.

  ```
  const QUEUE_SIZE: u16 = 256;
  const QUEUE_SIZES: &[u16] = &[QUEUE_SIZE];

  fn queue_max_sizes(&self) -> &[u16] {
      QUEUE_SIZES
  }
  ```

- The virtio framework core handles the configuration of the virtio devices by the guest OS. Once the device is ready, i.e. after all the virtqueues have been initialized, the framework invokes the `activate` method to turn the device on and make it start processing incoming requests.

  The `activate` method is by far the most complex method of the interface between the virtio framework and the device. This method will be discussed in detail in the following section.

- The `keep_rds` method returns a vector of file descriptors that must be kept open when the device is sandboxed using the minijail framework to ensure that the device works correctly.

```
fn keep_rds(&self) -> Vec<RawDescriptor> {
    let mut keep_rds = Vec::new();

    if let Some(random_file) = &self.random_file {
        keep_rds.push(random_file.as_raw_descriptor());
    }

    keep_rds
}
```

  The `virtio-rng` device only needs to ensure that it still has access to the underlying entropy source (i.e. the `/dev/urandom` device) after being sandboxed.

  Note that this method returns a vector of RawDescriptor objects which is cvm2rb's internal representation of UNIX file descriptors.

In addition to these mandatory methods, there are a few optional callbacks that can be implemented by a device to support additional virtio features.

- A device may offer optional features that the driver is free to accept or not. These optional features are presented to the driver via the virtio feature bits. During the device initialization, the driver queries the set of features supported by a specific device and indicates to the device the subset of features it wants to enable.

  To offer optional features, a virtio device shall implement the `features` method which returns the set of features it implements.

```
fn features(&self) -> u64 {
    self.virtio_features
}
```

  A device which provides optional features shall also implement the `ack_features` method. This method is invoked by the framework when the device enable features on the virtio device. The example below shows a simple implementation of this method that filters out invalid values and records the enabled features in the device object.

```
fn ack_features(&mut self, value: u64) {
    self.acked_features |= (value & !self.virtio_features);
}
```

  Please refer to the `virtio-net` implementation in cvm2rb (see the `devices/src/virtio/net.rs` file) for an example of virtio device that extensively uses optional features.

- The virtio specification allows devices to provide a device-specific configuration space that is used for rarely-changing or initialization-time parameters.

  Accesses to the device-specific configuration space are not handled by the framework itself and are dispatched to the device through the `read_config` and `write_config` methods of the VirtioDevice trait.

  These methods are defined using the following prototypes:

```
fn read_config(&self, offset: u64, data: &mut [u8]);
fn write_config(&mut self, offset: u64, data: &[u8]);
```

  The `offset` argument is the offset in bytes from the beginning of the device-specific configuration space that the guest is accessing.

The `data` argument is a slice object that is either zero-initialized and should be filled by the device for the `read_config` method, or contains the data to be written to the configuration space for `write_config`. In both cases, the slice length, i.e. `data.len()`, indicates the size of the access in the device-specific configuration space.

2.2 Activating the Device

Once the driver has fully initialized the virtio device, the framework invokes the activate method whose prototype is shown below.

```
fn activate(
    &mut self,
    mem: GuestMemory,
    interrupt: Interrupt,
    queues: Vec<Queue>,
    queue_evts: Vec<Event>,
);
```

This method is invoked with the following arguments:

- `mem`: A handle to the guest memory (see the GuestMemory type) that the driver uses to access any object allocated by the guest OS in its own address space (e.g. the virtqueues and the virtio buffers).
- `interrupt`: The Interrupt object associated with this device. This object is the interface offered by cvm2rb framework to enable virtio devices to interrupt the driver.

  They are two reasons for a virtio device to interrupt the driver:

  - when some buffers have been added on a virtqueue used ring, or
  - when the device configuration has changed.

  An Interrupt object provides two different methods to cover both use cases: `signal_used_queue` and `signal_config_changed`.
- `queues`: A vector of Queue object that represents the device virtqueues. A Queue object provides methods that the driver can use to extract buffers from the available ring and return them to the used ring after processing a request.
- `queue_evts`: A vector of Event objects (one per virtqueue) which serves to dispatch queue notification events (i.e. kick notifications) to this device.

A slightly simplified version of the `activate` method of the `virtio-rng` device is included below.

```
fn activate(
    &mut self,
    mem: GuestMemory,
    interrupt: Interrupt,
    mut queues: Vec<Queue>,
    mut queue_evts: Vec<Event>,
) {
    if queues.len() != 1 || queue_evts.len() != 1 {
        return;
    }

    let queue = queues.remove(0);
```

```
    if let Some(random_file) = self.random_file.take() {
        let worker_result =
            thread::Builder::new()
                .name("virtio_rng".to_string())
                .spawn(move || {
                    let mut worker = Worker {
                        interrupt,
                        queue,
                        mem,
                        random_file,
                    };
                    worker.run(queue_evts.remove(0));
                    worker
                });

        match worker_result {
            Err(e) => {
                error!("failed to spawn virtio_rng worker: {}", e);
                return;
            }
            Ok(join_handle) => {
                self.worker_thread = Some(join_handle);
            }
        }
    }
}
```

The first thing that the `activate` method does is to check that the number of virtqueue provided to the device matches its expectations to ensure that the device will work correctly.

The device then starts a thread that implements the core logic of the virtio device, for `virtio-rng`, it means filling available buffers with random data. This must be done from a separate thread for the reasons explained below.

Once a virtio device has been activated, the queue notifications events (i.e. kick notifications) are delivered to the device using the Event channels provided as the `queue_evts` argument of the `activate` method.

In this model, the device must poll these channels for incoming events. However, the `activate` method is invoked in the context of cvm2rb's main thread which dispatch incoming I/O events from the guest OS amongst all the devices running in cvm2rb. Starting to poll the `queue_evts` channels from the `activate` method would prevent cvm2rb from performing this dispatching. For this reason, the virtio device must return from the `activate` method as soon as possible and must start a separate thread to handle this polling activity.

2.3 The Main Device Thread

All the virtio devices in cvm2rb follow the common pattern of creating a `Worker` object to hold all the data needed by the device's main thread.

For the `virtio-rng` device the `Worker` type is defined as:

```
struct Worker {
    interrupt: Interrupt,
    queue: Queue,
    mem: GuestMemory,
    random_file: File,
}
```

A worker object is created by the `activate` method when starting the `virtio-rng` main thread.

All the worker objects in cvm2rb provide a `run` method which implements the thread event loop. For `virtio-rng` the implementation is around the line of:

```
fn run(&mut self, queue_evt: Event) {
    #[derive(PollToken)]
    enum Token {
        QueueAvailable,
    }

    let wait_ctx: WaitContext<Token> = match WaitContext::build_with(&[
        (&queue_evt, Token::QueueAvailable),
    ]) {
        Ok(pc) => pc,
        Err(e) => {
            error!("failed creating WaitContext: {}", e);
            return;
        }
    };

    'wait: loop {
        let events = match wait_ctx.wait() {
            Ok(v) => v,
            Err(e) => {
                error!("failed polling for events: {}", e);
                break;
            }
        };

        let mut needs_interrupt = false;
        for event in events.iter().filter(|e| e.is_readable) {
            match event.token {
                Token::QueueAvailable => {
                    if let Err(e) = queue_evt.read() {
                        error!("failed reading queue Event: {}", e);
                        break 'wait;
                    }
                    needs_interrupt |= self.process_queue();
                }
            }
```

```
        }
        if needs_interrupt {
            self.interrupt.signal_used_queue(self.queue.vector);
        }
    }
}
```

Once again, we present a simplified version of the `virtio-rng` code that leaves out the boilerplate code which comes with implementing a device in cvm2rb and focuses on the virtio aspects.

In cvm2rb, a virtio device must poll the various queue event channels provided at device activation time. To help with this polling, the framework provides the WaitContext types to enable a virtio device to monitor for incoming events on multiple channels. A WaitContext object is essentially a wrapper around the Linux `epoll` mechanism.

The `run` method starts by creating a WaitContext object and populates it with all the event channels it must wait for. For the `virtio-rng` device there is only a single virtqueue we need to poll from.

A more advanced device such as `virtio-net` would need to add all the virtqueues used by the device (i.e. the receive, transmit and control queues) to the WaitContext. In addition, a `virtio-net` device would typically add the TAP interface to the WaitContext so that the device gets notified when new packets are coming from the TAP interface and need to be transmitted to the driver.

After creating the WaitContext object, the `run` method enters the events processing loop where it waits for incoming events from the WaitContext. Each time it receives a notification on the `virtio-rng` device's virtqueue, it invokes the `process_queue` function.

2.4 Processing the Virtqueues

The `process_queue` function implements the core logic of the `virtio-rng` device. Whenever this function is invoked, it fills all the available buffers from the device virtqueue with random data coming from the underlying entropy source (i.e. the `/dev/urandom` device), before returning these buffers to the used ring.

The below code illustrates this process.

```
fn process_queue(&mut self) -> bool {
    let queue = &mut self.queue;

    let mut needs_interrupt = false;
    while let Some(avail_desc) = queue.pop(&self.mem) {
        let index = avail_desc.index;
        let random_file = &mut self.random_file;
        let written = match Writer::new(self.mem.clone(), avail_desc)
            .map_err(|e| io::Error::new(io::ErrorKind::InvalidInput, e))
            .and_then(|mut writer| writer.write_from(random_file,
std::usize::MAX))
        {
            Ok(n) => n,
            Err(e) => {
                warn!("Failed to write random data to the guest: {}", e);
                0
            }
```



```
        };

        queue.add_used(&self.mem, index, written as u32);
        needs_interrupt = true;
    }

    needs_interrupt
}
```

As we mentioned earlier, in cvm2rb, virtqueues are represented by Queue objects which provide methods to operate on virtqueues. In the above example we demonstrate the use of:

- `queue.pop(&self.mem)`, to pick the next available buffer from the virtqueue if any. The buffer is also removed from the virtqueue so that subsequent calls to the `pop` method will not return it.
- `queue.add_used(&self.mem, index, written as u32)` which returns a virtio buffer to the driver by adding it to the used buffers ring.

Note that both methods require a handle to the guest memory which is used internally to access the virtqueue memory. Keep in mind that with virtio, the virtqueues are allocated by the driver in its own address space.

The cvm2rb virtio framework also provides some helpers to read and write data in virtio buffers. These helpers hide from the programmer the details of the actual buffer representation (e.g. chained buffers and indirect descriptors). They are exposed through the Reader and Writer types.

The `virtio-rng` device does not use device-readable buffers so the above example only uses the Writer type.

A Reader or Writer object is associated with a specific virtio buffer, returned from the available list by the `Queue::pop,` method when creating it:

```
Writer::new(self.mem.clone(), avail_desc)
```

Since virtio buffer are located in the guest OS memory, the Writer object also needs a handle to the guest memory.

The Writer type provides a wide variety of methods to support writing different kind of objects into a virtio buffer which are described in the API documentation. In this example we use:

```
writer.write_from(random_file, std::usize::MAX))
```

to copy at most `std::usize::MAX` bytes from the random source into the virtio buffer. In this particular example it is very likely that the `write_from` method reaches the end of the virtio buffer before `std::usize::MAX` bytes are written.

We have only covered the Writer helper here, and the use of Reader objects follows similar patterns. Please refer to the API documentation for an exhaustive list of methods supported on Reader objects. Also refer to other virtio devices such as `virtio-console` or `virtio-net` for usage examples.

2.5 Miscellaneous Implementation Details

We have now covered all the basics of a real virtio device running in cvm2rb. The following sections discuss further details which have been deliberately put aside while presenting the `virtio-rng` device implementation.

2.5.1 Interrupt resampling events

In the previous sections, we have presented a simplified version of the `virtio-rng` `Worker::run` method. The main difference with the actual implementation resides in the set of event channels that are monitored through the WaitContext.

In addition to waiting for events coming from the `virtio-rng` device virtqueue, the actual implementation also considers interrupt resampling events. This feature is inherited from the original crosvm implementation. It is used by KVM to notify the device that a specific interrupt line needs resampling (e.g. on end-of-interrupt signal), giving the device an opportunity to re-assert the interrupt line at that time.

All the virtio devices in cvm2rb includes similar code to deal with interrupt resampling events because the original implementation has not been modified. However, this feature is not used in cvm2rb and the virtio devices will never receive such events.

2.5.2 Resettable virtio devices

Most virtio devices in cvm2rb support being reset by the driver (i.e. by writing 0 to the virtio status register). To enable this feature, cvm2rb's virtio devices follow a common pattern.

- When the device is activated, it creates a custom event channel that is used to dispatch *kill events* from the cvm2rb main thread, which runs virtio control-plane for all the devices, to the device-specific thread.

  This channel is based on an Event object:

  ```
  let (self_kill_evt, kill_evt) =
    match Event::new().and_then(|e| Ok((e.try_clone()?, e))) {
      Ok(v) => v,
      Err(e) => {
          error!("failed to create kill Event pair: {}", e);
          return;
      }
    };
  self.kill_evt = Some(self_kill_evt);
  ```

  One end of this channel is stored in the virtio device object itself and the other end is passed to the device worker thread.

- In the device worker thread `run` method, the *kill event* channel is added to the WaitContext. When the device thread receives a signal on this channel, it bails out of the event loop and terminates.

- The device implements the optional `reset` method from the VirtioDevice trait which is used by cvm2rb virtio framework core whenever the device needs to be reset.

  The implementation of this method is trivial and consists in:

  1. sending a *kill event* to the device thread, then
  2. waiting for the thread to terminate.

The following code shows a simplified version of this method:

```
fn reset(&mut self) -> bool {
    if let Some(kill_evt) = self.kill_evt.take() {
        if kill_evt.write(1).is_err() {
            error!("{}: failed to notify the kill event",
                   self.debug_label());
            return false;
        }
    }

    if let Some(worker_thread) = self.worker_thread.take() {
        match worker_thread.join() {
            Err(_) => {
                error!("{}: failed to get back resources",
                       self.debug_label());
                return false;
            }
            Ok(worker) => {
                self.random_file = Some(worker.random_file);
                return true;
            }
        }
    }
    false
}
```