The future of virtio: riddles, myths and surprises

Michael S. Tsirkin
Jens Freimann
Fall 2017
Virtio initialization

Virtio device
ID: 0x1                Device Features: 0x1200000000

Guest OS:
load driver for ID: 0x1

Driver:          supported features: 0x100001000
Device Features: 0x1200000000

Driver features:
0x1000000000
status

Driver features:
0x1000000000
OK / Fail
Virtio initialization

Virtio device
ID: 0x1    Device Features: 0x120000000

Guest OS:
load driver for ID: 0x1

Device Features: 0x120000000

Driver:
supported features: 0x100001000

Driver features:
0x100000000

&

Driver features:
0x100000000

status

OK / Fail
Myth #1: “changing virtio would break existing drivers”

- Really: feature negotiation can ensure compatibility
- Forward and backward
- For devices and drivers

- Let’s see it in action ...
Virtio input: add multitouch feature

- Feature bit: VIRTIO_INPUT_F_MULTITOUCH = 0
- New (multi-touch aware) device: device features = 0x1
- New driver: supported features = 0x1
- Driver features: 0x1 & 0x1 = 0x1
- Device and driver:
  \[
  \text{if (driver\_features \& (1 \ll \text{VIRTIO\_INPUT\_F\_MULTITOUCH}))}
  \]
  enable multi-touch support
- Updated device & driver: multi-touch enabled!
Compatibility: existing drivers

- Device features = 0x1
- Driver supported = 0x0
- Driver features = 0x0
- 0x0 & (1 << VIRTIO_INPUT_F_MULTITOUCH) == 0
- Device: option 1: disable multi-touch: compatible!
- Device: option 2: set status = fail
  Not worse than building a new device!
  Can suggest upgrading a driver.
Compatibility: existing devices

- Device features: 0x0
- Driver supported: 0x1
- Driver features: 0x0
- $0x0 \& (1 << \text{VIRTIO\_INPUT\_F\_MULTITOUCH}) == 0$
- Driver: option 1: disable multi-touch
- Driver: option 2: set status = fail
  Can suggest upgrading a device.
Compatibility: virtio 0.9 versus 1.0

- virtio 1.0 – made default Jul 2016
- Switched devices to a different register layout
- Gated by a feature bit:
  
  /* v1.0 compliant. */
  
  #define VIRTIO_F_VERSION_1 32

- No one noticed!
Myth #2
Changing virtio requires writing a specification

- Absolutely the right thing to do
- Does not have to be step 0!

Virtio priorities:
- Code compatibility
- IPR compatibility
- Interface compatibility
Code compatibility: avoid conflicting with others

- New device: reserve an ID. Spec patch:
  
  ```
  diff --git a/content.tex b/content.tex
  @@ -3022,3 +3022,5 @@ Device ID & Virtio Device \hline
  +23 & misc device \hline
  +\hline
  \end{tabular}
  ```

- Existing device: reserve a feature bit. E.g.:

  ```
  @@ -4800,5 +4802,6 @@ guest memory statistics
  \item[VIRTIO_BALLOON_F_DEFLATE_ON_OOM (2)] Deflate balloon on guest out of memory condition.
  +\item[VIRTIO_BALLOON_F_XXXX (3)] Reserved for feature XXXX.
  \end{description}
  ```
How to get it in the spec?

- git clone https://github.com/oasis-tcs/virtio-spec
- Edit :)
- sh makeall.sh (needs xelatex, e.g. from texlive)
- virtio-comment-subscribe@lists.oasis-open.org
- Patch: virtio-comment@lists.oasis-open.org
- If no comments – email, ask for a vote ballot
- Total time: up to 2 weeks
IPR compatibility: allow others to implement compatible devices

- Open-source an implementation
- Subscribe to virtio-dev@lists.oasis.org
- Agree to IPR rules (non-assertion mode)
- Send a copy of the patches (e.g. qemu, linux, dpdk) to virtio-dev@lists.oasis.org
- Virtio GPU at this point now.
Interface compatibility

- Document assumptions for inter-operability
- Virtio membership is not required
- Membership is open - members vote on ballots
- Hints:
  - Document device and driver separately
  - Use MUST/SHOULD/MAY keywords
  - Ask for help!
- Virtio crypto and input at this point
Myth #3
virtio has lowest possible overhead for host/guest communication

● “Efficient: Virtio devices consist of rings of descriptors for both input and output, which are neatly laid out to avoid cache effects from both driver and device writing to the same cache lines”.

● True - but is this really efficient?
Virt queue: shared memory host/guest communication
CPU caching

- Communication through shared memory requires cache synchronization (invalidate, miss, ...).

- This impacts latency.
Counting misses: no batching

- Access = cache miss
  5 cache misses per request
Counting misses: batching

- Virtio 1.0 queue layout: batching

- Batch=4 → 5 misses per batch
  1.25 misses per request
Cache miss cost

![Bar chart showing cache miss cost in nanoseconds for shared memory and shared cache across different batches. The chart displays a sharp increase in cost for the first batch, followed by a more gradual increase for subsequent batches.]
Reducing the overhead

- Information is spread across too many data structures
- Tighter packing will save cache misses.
- How about packing everything in a single data structure?
Descriptor ring

- Driver writes out available descriptors in a ring
- Device writes out used descriptors in the same ring
- Descriptor: addr, len, avail, used
- To mark a descriptor available, flip the avail bit
- To mark a descriptor as used, flip the used bit
Descriptor states

Avail = used: ok for guest to produce
Avail != used: ok for host to consume
static int used = 1;
while(desc[idx].avail == used) ← miss?
    relax();
process(&desc[idx]);
desc[idx].used = used; ← miss?
idx = idx + 1;
if (idx == size)
    Idx = 0;
    used = !!used;
Out-of-order: descriptor id

- Guest: available 9
- Host: used 1

<table>
<thead>
<tr>
<th>Avail/used</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Host: use</th>
<th>1/1</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Guest: available</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
<th>1/0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0/0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

redhat
CPU caching

- Both host and guest incur misses on access
- No batching: 2 to 4 misses per descriptor
- Batch=4:
  2 to 4 misses per batch
  4 descriptors per cache line $\rightarrow$
  0.5 to 1 misses per descriptor
- Better than virtio 1.0 even in the worst case
In-order: descriptor id

- Guest: produced 9
- Host: consumed 9

One write per batch of descriptors
Driver ensures avail != used
Request processing: comparison

- Virtio 1.0
- Virtio 1.0 shared cache
- Virtio 1.1
- Virtio 1.1 shared cache

ns vs batch
64 byte packet throughput

Virtio queue is not optimal
we will fix it
Riddle #1: event suppression

- Each queue has two event index structures
- Which descriptor should trigger an interrupt
- Can we put this in the descriptor itself?
- Should we?
- Just use polling?
Riddle #2: why powers of 2?

2 VQs * 1K descriptors fills a 32K cache

2VQs * 0.75K descriptors free cache for data
Powers of two: pseudo code

unsigned next_power_of_two(unsigned index, unsigned size)
{
    return (index + 1) & (size - 1);
}

unsigned next_non_power_of_two(unsigned index, unsigned size)
{
    return ++index >= size ? 0 : index;
}
Surprise #1: hardware is special

- Let’s assume a pass-through device implementing virtio. Shouldn’t this just work?
- Maybe – but not optimally!
- Hypervisor: processes descriptors one by one
- Hardware: can process many in parallel
- Needs to be told how many are available
- Include number of available entries in a kick
Surprise #2: writes are expensive

- PCI Express payload is full dword.
- Flipping single bits across PCIE is expensive
- In-order processing will help reduce number of writes
Summary

- Virtio 1.1 is shaping up to be a big release
  - Performance
  - Hardware offloads

- Join the fun
  - Still lots of open questions
  - Implementation and benchmarking of the new features
  - Virtio BoF tomorrow