Crossing the endianness bridge
(or a foolish attempt at mixed-endian virtualization)

Marc Zyngier
<marc.zyngier@arm.com>

KVM Forum ’13
Foreword

This is a (hopefully short) talk about mixed-endianness virtualization support on arm64. It is NOT about:

► Finding out whether Little Endian is better or worse than Big Endian
  → We all know what the answer is, don’t we?
► The big.LITTLE architecture
  → One issue at a time, please...
► Claming we have nailed the problem for good
  → If only...
Foreword

This is a (hopefully short) talk about mixed-endianness virtualization support on arm64. It is NOT about:

- Finding out whether Little Endian is better or worse than Big Endian
  - We all know what the answer is, don’t we?
- The big.LITTLE architecture
  - One issue at a time, please...
- Claming we have nailed the problem for good
  - If only...

It is more about:

- Telling the story of why and how we got there
- Starting a discussion on how we can better support that kind of configuration
Foreword

This is a (hopefully short) talk about mixed-endianness virtualization support on arm64. It is NOT about:

- Finding out whether Little Endian is better or worse than Big Endian
  → We all know what the answer is, don’t we?
- The big.LITTLE architecture
  → One issue at a time, please…
- Claiming we have nailed the problem for good
  → If only…

It is more about:

- Telling the story of why and how we got there
- Starting a discussion on how we can better support that kind of configuration

Mandatory read:
Big and Little Endian Inside Out – Ben Herrenschmidt, LPC12
ARM: to BE or not to BE?

So what about ARM and endianness?

- Endianness agnostic architecture
- Mostly used as LE with Linux
- Recent surge in BE interest
- Linux/ARM BE port revived by Ben Dooks

Meanwhile, in Cambridge:

Too bad we can't really test this BE code...

Wonder if we could stick something in KVM...

Profit!
ARM: to BE or not to BE?

So what about ARM and endianness?

- Endianness agnostic architecture
- Mostly used as LE with Linux
- Recent surge in BE interest
- Linux/ARM BE port revived by Ben Dooks

Meanwhile, in Cambridge:

WD Too bad we can’t really test this BE code...
MZ Wonder if we could stick something in KVM...
WD ...
MZ Profit!
ARM: to BE or not to BE? (take #2)

Why this sudden surge in BE interest? Isn’t BE a dead horse anyway?
Well, maybe. But there’s always a “But”:

▶ Networking folks are now coming to ARM
▶ They are very interested in AArch64
▶ They have a lot of existing code that is:
  ▶ Old
  ▶ Crufty
  ▶ Unmaintained
  ▶ Closed source
  ▶ Certified
  ▶ Not 64bit safe
  ▶ Big-Endian only
  ▶ Did I say old and unmaintained?
ARM: to BE or not to BE? (take #2)

Why this sudden surge in BE interest?
Isn’t BE a dead horse anyway?
Well, maybe. But there’s always a “But”:
▶ Networking folks are now coming to ARM
▶ They are very interested in AArch64
▶ They have a lot of existing code that is:
  ▶ Old
  ▶ Crufty
  ▶ Unmaintained
  ▶ Closed source
  ▶ Certified
  ▶ Not 64bit safe
  ▶ *Big-Endian only*
ARM: to BE or not to BE? (take #2)

Why this sudden surge in BE interest?
Isn’t BE a dead horse anyway?
Well, maybe. But there’s always a “But”:

▶ Networking folks are now coming to ARM
▶ They are very interested in AArch64
▶ They have a lot of existing code that is:
  ▷ Old
  ▷ Crufty
  ▷ Unmaintained
  ▷ Closed source
  ▷ Certified
  ▷ Not 64bit safe
  ▷ Big-Endian only
▶ Did I say old and unmaintained?
ARM: to BE or not to BE? (take #2)

Why this sudden surge in BE interest?
Isn’t BE a dead horse anyway?
Well, maybe. But there’s always a “But”:

- Networking folks are now coming to ARM
- They are very interested in AArch64
- They have a lot of existing code that is:
  - Old
  - Crufty
  - Unmaintained
  - Closed source
  - Certified
  - Not 64bit safe
  - Big-Endian only

- Did I say old and unmaintained?
- They either don’t want to or simply cannot touch this code
BE: Do the right thing!

Rewriting the BE code to be endianness agnostic.

- Solves the problem completely
- The perfect solution
- Why the hell are we here?
BE: Do the right thing!

Rewriting the BE code to be endianness agnostic.

- Solves the problem completely
- The perfect solution
- Why the hell are we here?

Not an option.

- It would take years to achieve
- Goes against all the reasons we have mentioned before
BE: Do the right thing!

Rewriting the BE code to be endianness agnostic.
  ▶ Solves the problem completely
  ▶ The perfect solution
  ▶ Why the hell are we here?

Not an option.
  ▶ It would take years to achieve
  ▶ Goes against all the reasons we have mentioned before

It’s not fun at all anyway
BE: Compiler magic to have endian-specific accessors

```c
struct bar {
    unsigned int foo;
};

void blah(struct bar *barp) {
    barp->foo = 0xbe5afe;
}

blah:    mov    w1, #0x5afe
          movk   w1, #0xbe, lsl #16
          str    w1, [x0]
          ret
```
BE: Compiler magic to have endian-specific acccessors

```c
struct bar {
    unsigned int foo;
} __struct_layout_be;

void blah(struct bar *barp)
{
    barp->foo = 0xbe5afe;
}

blah:  mov       w1, #0x5afe
        movk      w1, #0xbe, lsl #16
        rev       w1, w1
        str       w1, [x0]
        ret
```

Lovely idea, but requires the data structure to be annotated, which defeats the whole idea of not touching the code.
BE: Compiler magic to have endian-specific accessors

```c
struct bar {
    unsigned int foo;
} __struct_layout_be;

void blah(struct bar *barp) {
    barp->foo = 0xbe5afe;
}

blah:    mov    w1, #0x5afe
         movk   w1, #0xbe, lsl #16
         rev    w1, w1
         str    w1, [x0]
         ret
```

Lovely idea, but requires the data structure to be annotated, which defeats the whole idea of *not touching the code*. 
BE: hacking the kernel to run mixed-endian userspace

Allow the kernel to deal with both LE and BE userspaces at the same time, by doing the necessary marshalling at the syscall level (à la compat-layer).

Seems like an ideal solution:

- No source code change
- No userspace change
- Focussed changes in the kernel (syscalls)
BE: hacking the kernel to run mixed-endian userspace

Allow the kernel to deal with both LE and BE userspaces at the same time, by doing the necessary marshalling at the syscall level (à la compat-layer).

Seems like an ideal solution:

▶ No source code change
▶ No userspace change
▶ Focussed changes in the kernel (syscalls)
▶ Too good to be true?

Unfortunately yes. It is perfect until you consider:

▶ Futexes
▶ Shared memory
▶ Most IPCs, actually

But we seem to get closer...
BE: hacking the kernel to run mixed-endian userspace

Allow the kernel to deal with both LE and BE userspaces *at the same time*, by doing the neccessary marshalling at the syscall level (*à la compat-layer*).

Seems like an ideal solution:

- No source code change
- No userspace change
- Focussed changes in the kernel (syscalls)
- Too good to be true?

Unfortunately yes. It is perfect until you consider:

- Futexes
- Shared memory
- Most IPCs, actually

**But we seem to get closer...**
BE: using virtualization for sandboxing

Let’s move all our BE code (including the kernel) into a VM:

- Similar to the previous solution
- No source code change
- No userspace change
- Strong isolation between BE and LE worlds
Let’s move all our BE code (including the kernel) into a VM:

- Similar to the previous solution
- No source code change
- No userspace change
- Strong isolation between BE and LE worlds
- The perfect match?
BE: using virtualization for sandboxing

Let’s move all our BE code (including the kernel) into a VM:

- Similar to the previous solution
- No source code change
- No userspace change
- Strong isolation between BE and LE worlds
- The perfect match?

Let’s investigate...
ARM: The art of BEing

ARM BE support started off with something called BE-32:

- 32bit word invariant
  - Words have the same ordering, no matter the endianness
- Byte (and 16bit) addressing differs between BE and LE
- Just a hack on the bus
- Affects both data and instructions
- Switching from one mode to another is a nightmare
  - The best path to insanity

Thankfully, we can now forget about this.
ARM: The art of BEing

ARM BE support started off with something called BE-32:

- 32bit word invariant
  - Words have the same ordering, no matter the endianness
- Byte (and 16bit) addressing differs between BE and LE
- Just a hack on the bus
- Affects both data and instructions
- Switching from one mode to another is a nightmare
  - The best path to insanity

Thankfully, we can now forget about this.
Starting with ARMv6, BE support is implemented as BE-8:

- **Byte invariant**
  - Bytes are located at the same address, no matter the endianness

- **16bit, 32bit addressing differs between BE and LE**
  - Just like on any other sane architecture...

- **Instructions and data have distinct endianness**
  - Instructions are always little endian
  - Data endianness is configurable

This is what to want to support.
Each exception level has its own endianness configuration. A taken exception automatically switches the core to the endianness of the target exception level. Endianness of the level causing the exception will be restored on exception return.

This looks like an easy fit for KVM:

- Let the guest deal with its own endianness.
- Switch back to LE when trapping into the hypervisor.

Job done! Well, almost...
ARM: Exception levels and endianness

Each exception level has its own endianness configuration.
A taken exception automatically switches the core to the endianness of the target exception level.
Endianness of the level causing the exception will be restored on exception return.

Non-Secure State

NS User Mode
NS Kernel Modes
NS Hyp Mode

Secure State

S User Mode
S Privileges Modes
ARM: Exception levels and endianness

Each exception level has its own endianness configuration

A taken exception automatically switches the core to the endianness of the target exception level

Endianness of the level causing the exception will be restored on exception return

This looks like an easy fit for KVM:

- Let the guest deal with its own endianness
- Switch back to LE when trapping into the hypervisor.
ARM: Exception levels and endianness

Each exception level has its own endianness configuration

A taken exception automatically switches the core to the endianness of the target exception level

Endianness of the level causing the exception will be restored on exception return

This looks like an easy fit for KVM:
  - Let the guest deal with its own endianness
  - Switch back to LE when trapping into the hypervisor.

Job done! Well, almost...
Beyond the core: Devices

There is, of course, more to mixed-endianness than just dealing with the CPU. Let’s talk about devices!
Beyond the core: Devices

There is, of course, more to mixed-endianness than just dealing with the CPU. Let’s talk about devices!

- Peripherals that are close to the core are always LE
  - Interrupt controller
  - Generic timers
Beyond the core: Devices

There is, of course, more to mixed-endianness than just dealing with the CPU. Let’s talk about devices!

- Peripherals that are close to the core are always LE
  - Interrupt controller
  - Generic timers
- ARM recommends that all other peripherals are wired as LE
Beyond the core: Devices

There is, of course, more to mixed-endianness than just dealing with the CPU. Let’s talk about devices!

- Peripherals that are close to the core are always LE
  - Interrupt controller
  - Generic timers
- ARM recommends that all other peripherals are wired as LE
- Linux IO accessors are constructed to byteswap on BE
Beyond the core: Devices

There is, of course, more to mixed-endianness than just dealing with the CPU. Let’s talk about devices!

▶ Peripherals that are close to the core are always LE
  → Interrupt controller
  → Generic timers
▶ ARM recommends that all other peripherals are wired as LE
▶ Linux IO accessors are constructed to byteswap on BE

This looks like a no-brainer too:
Beyond the core: Devices

There is, of course, more to mixed-endianness than just dealing with the CPU. Let’s talk about devices!

- Peripherals that are close to the core are always LE
  - Interrupt controller
  - Generic timers
- ARM recommends that all other peripherals are wired as LE
- Linux IO accessors are constructed to byteswap on BE

This looks like a no-brainer too:

- VM-accessible devices should be already dealt with
Beyond the core: Devices

There is, of course, more to mixed-endianness than just dealing with the CPU. Let’s talk about devices!

- Peripherals that are close to the core are always LE
  - Interrupt controller
  - Generic timers
- ARM recommends that all other peripherals are wired as LE
- Linux IO accessors are constructed to byteswap on BE

This looks like a no-brainer too:

- VM-accessible devices should be already dealt with
- Emulated devices should also be OK
Beyond the core: Devices

There is, of course, more to mixed-endianness than just dealing with the CPU. Let’s talk about devices!

- Peripherals that are close to the core are always LE
  - Interrupt controller
  - Generic timers
- ARM recommends that all other peripherals are wired as LE
- Linux IO accessors are constructed to byteswap on BE

This looks like a no-brainer too:

- VM-accessible devices should be already dealt with
- Emulated devices should also be OK
- Let’s look a bit closer
Interlude: Second stage translation

How does a guest access memory on KVM/arm[64]?  
► Always in control of its own page tables  
► No trapping, no subtle repainting  
► Generated address is an Intermediate Physical Address (IPA)
Interlude: Second stage translation

How does a guest access memory on KVM/arm[64]?  
- Always in control of its own page tables  
- No trapping, no subtle repainting  
- Generated address is an Intermediate Physical Address (IPA)

The hypervisor provides
- A second set of page tables (Stage-2 translation)
- Converts IPA to PA
  - Allows mapping of what the guest sees as physical memory with the real thing
- Can override attributes like cacheability, shareability
  - This is where some subtle repainting can occur...
- Can use a different page size
- Allows pages to be dynamically mapped by handling Stage-2 page faults

MMIO devices do not have a Stage-2 translation, allowing access to be trapped.
Interlude: Second stage translation

How does a guest access memory on KVM/arm[64]?  
- Always in control of its own page tables  
- No trapping, no subtle repainting  
- Generated address is an Intermediate Physical Address (IPA)

The hypervizor provides  
- A second set of page tables (Stage-2 translation)  
- Converts IPA to PA  
  - Allows mapping of what the guest sees as physical memory with the real thing  
- Can override attributes like cacheability, shareability  
  - This is where some subtle repainting can occur...  
- Can use a different page size  
- Allows pages to be dynamically mapped by handling Stage-2 page faults

MMIO devices do not have a Stage-2 translation, allowing access to be trapped.
The journey of a (wana)BE write

Let’s have a look at what happens when a BE guest writes to an emulated LE device:

▶ Guest loads the value to write in a register
▶ Guest byteswaps the value in the register
  → The bus is LE, so the value has to be swapped
▶ Guest performs the write
▶ Stage-2 translation fault, as no mapping exists at this address
  → KVM takes over
The journey of a (wana)BE write

Let’s have a look at what happens when a BE guest writes to an emulated LE device:

- Guest loads the value to write in a register
- Guest byteswaps the value in the register
  → The bus is LE, so the value has to be swapped
- Guest performs the write
- Stage-2 translation fault, as no mapping exists at this address
  → KVM takes over

At that point, we have the following information:

- The Intermediate Physical Address the guest was writing to
- The register containing the value it was writing
- The size of the access
The journey of a (wana)BE write

Let’s have a look at what happens when a BE guest writes to an emulated LE device:

- Guest loads the value to write in a register
- Guest byteswaps the value in the register
  → The bus is LE, so the value has to be swapped
- Guest performs the write
- Stage-2 translation fault, as no mapping exists at this address
  → KVM takes over

At that point, we have the following information:

- The Intermediate Physical Address the guest was writing to
- The register containing the value it was writing
- The size of the access

But the register now contains a byteswapped value, that we’ll have to byteswap again before passing it to the device emulation.
The journey of a (wana)BE read

BE guest reads from a LE device are quite similar:

- Guest performs a read from a device address
- Stage-2 translation fault, as no mapping exists at this address
  → KVM takes over

Once the device has emulated the read access, we have to:

- Byteswap the value
  - The guest thinks it reads from a LE bus...
- Shove it into the register the guest used for its access
- Resume the guest
  - The guest will byteswap the value again...

Because device emulation completely bypasses the bus (and registers have no endianness), we end up byteswapping data twice.
The journey of a (wana)BE read

BE guest reads from a LE device are quite similar:

▶ Guest performs a read from a device address
▶ Stage-2 translation fault, as no mapping exists at this address
  → KVM takes over

At that point, we have the following information:

▶ The Intermediate Physical Address the guest was reading from
▶ The register it expect the value in
▶ The size of the access
The journey of a (wana)BE read

BE guest reads from a LE device are quite similar:

▶ Guest performs a read from a device address
▶ Stage-2 translation fault, as no mapping exists at this address
  → KVM takes over

At that point, we have the following information:

▶ The Intermediate Physical Address the guest was reading from
▶ The register it expect the value in
▶ The size of the access

Once the device has emulated the read access, we have to:

▶ Byteswap the value
  → The guest thinks it reads from a LE bus...
▶ Shove it into the register the guest used for its access
▶ Resume the guest
  → The guest will byteswap the value again...
The journey of a (wana)BE read

BE guest reads from a LE device are quite similar:

▶ Guest performs a read from a device address
▶ Stage-2 translation fault, as no mapping exists at this address
  → KVM takes over

At that point, we have the following information:

▶ The Intermediate Physical Address the guest was reading from
▶ The register it expect the value in
▶ The size of the access

Once the device has emulated the read access, we have to:

▶ Byteswap the value
  → The guest thinks it reads from a LE bus...
▶ Shove it into the register the guest used for its access
▶ Resume the guest
  → The guest will byteswap the value again...

Because device emulation completely bypasses the bus (and registers have no endianness), we end up byteswapping data twice.
The aftermath

So what have we found so far:

▶ We can sanely switch the CPU endianness around VM entry/exit
▶ We can trap an MMIO access, and do the necessary byteswaps if the VM is BE

Are we there yet? Not quite.
The aftermath

So what have we found so far:

▶ We can sanely switch the CPU endianness around VM entry/exit
▶ We can trap an MMIO access, and do the necessary byteswaps if the VM is BE
▶ One interesting bit:
   → LE guest on BE host is more efficient than BE on LE
   → No need to byteswap on the host side
   → Register representation is immune to endianness change
The aftermath

So what have we found so far:

▶ We can sanely switch the CPU endianness around VM entry/exit
▶ We can trap an MMIO access, and do the necessary byteswaps if the VM is BE
▶ One interesting bit:
  → LE guest on BE host is more efficient than BE on LE
  → No need to byteswap on the host side
  → Register representation is immune to endianness change

Are we there yet?
The aftermath

So what have we found so far:

- We can sanely switch the CPU endianness around VM entry/exit
- We can trap an MMIO access, and do the necessary byteswaps if the VM is BE
- One interesting bit:
  - LE guest on BE host is more efficient than BE on LE
  - No need to byteswap on the host side
  - Register representation is immune to endianness change

Are we there yet? **Not quite.**
The case of virtio/MMIO

What is virtio?
- A framework for paravirtualized devices
- Uses shared memory between host and guest
- High performance, low overhead

What is virtio/MMIO?
- The sick brainchild of Paweł Moll <pawel.moll@arm.com>
- Allows a virtio device to be exposed to a PCI-less system
- Exposes the configuration registers as an MMIO range
- Extensively used on ARM systems (“PCI? WTF?”)
- The only way kvmtool can provide a useful device to a KVM/arm[64] guest

So what is the problem with virtio/MMIO?
The case of virtio/MMIO #2

The spec is quite unclear when it comes to endianness.

- Config space (recently) declared as LE only
- Endianness in the virtio-ring unspecified
  → Assumed to be identical between host and guest

The challenge here is to introduce mixed-endian support without breaking existing users.
As we said above, the virtio/MMIO configuration space is strictly LE. However, the Linux driver code accessing it looks like this:

```c
static void vm_get(struct virtio_device *vdev, unsigned offset,
                    void *buf, unsigned len)
{
    struct virtio_mmio_device *vm_dev = to_virtio_mmio_device(vdev);
    u8 *ptr = buf;
    int i;

    for (i = 0; i < len; i++)
        ptr[i] = readb(vm_dev->base + VIRTIO_MMIO_CONFIG + offset + i);
}
```
virtio/MMIO: Fixing the config space

As we said above, the virtio/MMIO configuration space is strictly LE. However, the Linux driver code accessing it looks like this:

```c
static void vm_get(struct virtio_device *vdev, unsigned offset, 
                   void *buf, unsigned len)
{
    struct virtio_mmio_device *vm_dev = to_virtio_mmio_device(vdev);
    u8 *ptr = buf;
    int i;

    for (i = 0; i < len; i++)
        ptr[i] = readb(vm_dev->base + VIRTIO_MMIO_CONFIG + offset + i);
}
```

While the above code works for LE guests on LE hosts, it is unlikely to give any meaningful result on a BE guest (no matter the endianness of the host)...

virtio/MMIO: Fixing the config space

Let’s change the function prototype, providing an access size:

```c
static void vm_get(struct virtio_device *vdev, unsigned offset,  
    void *buf, unsigned len, unsigned access_size)
{
    struct virtio_mmio_device *vm_dev = to_virtio_mmio_device(vdev);
    int i;

    switch (access_size) {
        [...]
    case 2: {
        u16 *ptr = buf;
        for (i = 0; i < len; i++)
            ptr[i] = readw(vm_dev->base + VIRTIO_MMIO_CONFIG + offset + i);
        break;
    }
    case 4: {
        u32 *ptr = buf;
        for (i = 0; i < len; i++)
            ptr[i] = readl(vm_dev->base + VIRTIO_MMIO_CONFIG + offset + i);
        break;
    }
    [...]
}
```

Using the proper accessors (which are LE) solves the problem.
virtio/MMIO: Fixing the config space

Let’s change the function prototype, providing an access size:

```c
static void vm_get(struct virtio_device *vdev, unsigned offset,
                    void *buf, unsigned len, unsigned access_size)
{
    struct virtio_mmio_device *vm_dev = to_virtio_mmio_device(vdev);
    int i;

    switch (access_size) {
        [...]
        case 2: {
            u16 *ptr = buf;
            for (i = 0; i < len; i++)
                ptr[i] = readw(vm_dev->base + VIRTIO_MMIO_CONFIG + offset + i);
            break;
        }
        case 4: {
            u32 *ptr = buf;
            for (i = 0; i < len; i++)
                ptr[i] = readl(vm_dev->base + VIRTIO_MMIO_CONFIG + offset + i);
            break;
        }
        [...]
    }
}
```

Using the proper accessors (which are LE) solves the problem. At the expense of quite a few changes across all virtio drivers...
virtio/MMIO: One ring to BEnd them all

The virtio-ring can contain pure data, but also:

- Structures that are part of the virtio protocol
  → vring_desc, vring_avail...
- Data Directly parsed by the device backend
  → Depend on the device protocol itself
- When guest and host don’t agree on endianness, things get a bit ugly
The `virtio-ring` can contain pure data, but also:

- Structures that are part of the `virtio` protocol
  - `vring_desc`, `vring_avail`...
- Data Directly parsed by the device backend
  - Depend on the device protocol itself
- When guest and host don’t agree on endianness, things get a bit ugly

So what do we do?

- One possibility would be to declare all data to be LE
  - Breaks existing BE users
  - Could be an option for long term future, though
- Another is to add an endianness negotiation phase to the setup protocol
  - We have to make sure this gracefully falls back to the existing behaviour with unsuspecting guests
When a guest initializes a virtio device, it engages in a “feature negotiation” phase

For each virtio queue the guest initializes:
- Guest reads “feature flags” from the host
- Guest clears flags it doesn’t support (or doesn’t understand)
- Guest writes the flags it has selected back to the host

We can leverage this negotiation phase to our benefit:
- Let’s define two new flags
  - \texttt{VIRTIO\_RING\_F\_GUEST\_LE}: LE guest
  - \texttt{VIRTIO\_RING\_F\_GUEST\_BE}: BE guest
- The host can set either or both, depending on the endianness it supports
- The guest can keep the one corresponding to its endianness, or clear both.
  - It can’t keep them both on!
virtio/MMIO: One ring to BEnd them all

So how does this work in practice:

- Host can expose whatever endianness it supports, or none
- Guest can expose the one it uses, or none
- “none” is the current behaviour...
- Very finely grained – done on a per-queue basis
virtio/MMIO: One ring to BEnd them all

So how does this work in practice:

- Host can expose whatever endianness it supports, or none
- Guest can expose the one it uses, or none
- “none” is the current behaviour...
- Very finely grained – done on a per-queue basis

Added bonus:

- Selected at run time
  - No need for a endian-specific platform emulation
- No overhead at init time
  - Part of the feature negotiation phase
- Minimal overhead at run time
  - Platform emulation locally tests the queue flag
  - No need to trap into the kernel to find out
  - No hardcoded behaviour
- Architecture independant
  - Assuming your platform is bi-endian
And the guest side patch for that is incredibly small:

diff --git a/drivers/virtio/virtio_ring.c b/drivers/virtio/virtio_ring.c
index 6b4a4db..efff20a 100644
--- a/drivers/virtio/virtio_ring.c
+++ b/drivers/virtio/virtio_ring.c
@@ -813,6 +813,14 @@ void vring_transport_features(struct virtio_device *vdev)
     break;
     case VIRTIO_RING_F_EVENT_IDX:
     break;
+
+    ifdef __LITTLE_ENDIAN
+    case VIRTIO_RING_F_GUEST_LE:
+    endif
+    ifdef __BIG_ENDIAN
+    case VIRTIO_RING_F_GUEST_BE:
+    endif
+    default:
+        /* We don't understand this bit. */
+        clear_bit(i, vdev->features);
kvmtool: Handling mixed-endianness

kvmtool is the primary tool for KVM/arm64 development, as it makes a wonderful prototyping platform:

- Implements the queue endianness extension
- Extensive changes in the queue management
  - `virt_queue__available()`
  - `virt_queue__pop()`
  - `virt_queue__get_head_iov()`
  - ...
  - most of tools/kvm/virtio/core.c, actually
- A number of backends
  - console
  - block
  - 9p
  - net (uip)
kvmtool: let there BE...

The result:

```
root@genericarmv7ab:~# cat /proc/cpuinfo
processor : 0
model name : ARMv7 Processor rev 0 (v7b)
Features : swp half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
CPU implementer : 0x41
CPU architecture: 7
CPU variant : 0x0
CPU part : 0xd0f
CPU revision : 0

processor : 1
model name : ARMv7 Processor rev 0 (v7b)
Features : swp half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
CPU implementer : 0x41
CPU architecture: 7
CPU variant : 0x0
CPU part : 0xd0f
CPU revision : 0

Hardware : Dummy Virtual Machine
Revision : 0000
Serial : 0000000000000000
```

ARMv7 BE VM running on top of an ARMv8 LE host. A lot of headache for a very unspectacular result.
What we end up with

- A virtio extension that is:
  - architecture independent
  - virtio-centric
  - easily implemented
  - efficient
  - optional

- A kvmtool implementation that is:
  - minimal
  - mostly made of bug-fixes

- A KVM/arm implementation that is 100% bug-fixes
What got fixed so far

Doing this work was an interesting opportunity to revisit some areas of the code:

- Assumptions about the endianness in the KVM/arm code
  - MMIO handling code has been substantially rewritten
  - More to come...

- Configuration register access from the guest kernel
  - Introduction of size-based accessors
  - Signature check fix

- Endianness independance of the kvmtool virtio implementation
  - virtio queue rework
  - device implementation fixes
What is left to fix

We’re not quite done yet:

- A bunch of virtio devices are still left unloved
  - scsi
  - rng
  - and probably more...

- Running LE guests on BE host
  - Just in case you didn’t have enough
  - Requires more fixes in KVM/arm code
  - Probably some more in kvmtool
  - Linux driver code should hopefully be in a decent shape
The future

Middle-endian is the way!
The future, more seriously

In parallel, work is being done on the virtio front to solve the endianness problem for good (among others). This virtio 1.0 specification would change:

- Everything is Little-Endian
  - Native endianness has been forcefully eliminated
  - Solves the problem in the long run
The future, more seriously

In parallel, work is being done on the virtio front to solve the endianness problem for good (among others). This virtio 1.0 specification would change:

- Everything is Little-Endian
  - Native endianness has been forcefully eliminated
  - Solves the problem in the long run

But this also brings its own set of problems:

- Two different driver implementations
- Two different device implementations
- Some kind of (transitionnal) compatibility mode between them
- Nothing is available now

This presentation is more about a short term solution.
Questions?