KVM VCPU Requests

Overview

KVM supports an internal API enabling threads to request a VCPU thread to perform some activity. For example, a thread may request a VCPU to flush its TLB with a VCPU request. The API consists of the following functions:

```
/* Check if any requests are pending for VCPU @vcpu. */
bool kvm_request_pending(struct kvm_vcpu *vcpu);

/* Check if VCPU @vcpu has request @req pending. */
bool kvm_test_request(int req, struct kvm_vcpu *vcpu);

/* Clear request @req for VCPU @vcpu. */
void kvm_clear_request(int req, struct kvm_vcpu *vcpu);

/*
 * Check if VCPU @vcpu has request @req pending. When the request is
 * pending it will be cleared and a memory barrier, which pairs with
 * another in kvm_make_request(), will be issued.
 */
bool kvm_check_request(int req, struct kvm_vcpu *vcpu);

/*
 * Make request @req of VCPU @vcpu. Issues a memory barrier, which pairs
 * with another in kvm_check_request(), prior to setting the request.
 */
void kvm_make_request(int req, struct kvm_vcpu *vcpu);

/* Make request @req of all VCPUs of the VM with struct kvm @kvm. */
bool kvm_make_all_cpus_request(struct kvm *kvm, unsigned int req);
```

Typically a requester wants the VCPU to perform the activity as soon as possible after making the request. This means most requests (kvm_make_request() calls) are followed by a call to kvm_vcpu_kick(), and kvm_make_all_cpus_request() has the kicking of all VCPUs built into it.

VCPU Kicks

The goal of a VCPU kick is to bring a VCPU thread out of guest mode in order to perform some KVM maintenance. To do so, an IPI is sent, forcing a guest mode exit. However, a VCPU thread may not be in guest mode at the time of the kick. Therefore, depending on the mode and state of the VCPU thread, there are two other actions a kick may take. All three actions are listed below:

1. Send an IPI. This forces a guest mode exit.
2. Waking a sleeping VCPU. Sleeping VCPUs are VCPU threads outside guest mode that wait on waitqueues. Waking them removes the threads from the waitqueues, allowing the threads to run again. This behavior may be suppressed, see KVM_REQUEST_NO_WAKEUP below.
3. Nothing. When the VCPU is not in guest mode and the VCPU thread is not sleeping, then there is nothing to do.

VCPU Mode

VCPUs have a mode state, `vcpu->mode`, that is used to track whether the guest is running in guest mode or not, as well as some specific outside guest mode states. The architecture may use `vcpu->mode` to ensure VCPU requests are seen by VCPUs (see "Ensuring Requests Are Seen"), as well as to avoid sending unnecessary IPIs (see "IPI Reduction"), and even to ensure IPI acknowledgements are waited upon (see "Waiting for Acknowledgements"). The following modes are defined:

OUTSIDE_GUEST_MODE

The VCPU thread is outside guest mode.

IN_GUEST_MODE

The VCPU thread is in guest mode.

EXITING_GUEST_MODE

The VCPU thread is transitioning from IN_GUEST_MODE to OUTSIDE_GUEST_MODE.

READING_SHADOW_PAGE_TABLES

The VCPU thread is outside guest mode, but it wants the sender of certain VCPU requests, namely KVM_REQ_TLB_FLUSH, to wait until the VCPU thread is done reading the page tables.

VCPU Request Internals

VCPU requests are simply bit indices of the `vcpu->requests` bitmap. This means general bitops, like those documented in [atomic-ops] could also be used, e.g.

```
clear_bit(KVM_REQ_UNHALT & KVM_REQUEST_MASK, &vcpu->requests);
```

However, VCPU request users should refrain from doing so, as it would break the abstraction. The first 8 bits are reserved for architecture independent requests, all additional bits are available for architecture dependent requests.

Architecture Independent Requests

KVM_REQ_TLB_FLUSH

> KVM's common MMU notifier may need to flush all of a guest's TLB entries, calling kvm_flush_remote_tlbs() to do so. Architectures that choose to use the common kvm_flush_remote_tlbs() implementation will need to handle this VCPU request.

KVM_REQ_VM_DEAD

> This request informs all VCPUs that the VM is dead and unusable, e.g. due to fatal error or because the VM's state has been intentionally destroyed.

KVM_REQ_UNBLOCK

> This request informs the vCPU to exit kvm_vcpu_block. It is used for example from timer handlers that run on the host on behalf of a vCPU, or in order to update the interrupt routing and ensure that assigned devices will wake up the vCPU.

KVM_REQ_UNHALT

> This request may be made from the KVM common function kvm_vcpu_block(), which is used to emulate an instruction that causes a CPU to halt until one of an architectural specific set of events and/or interrupts is received (determined by checking kvm_arch_vcpu_runnable()). When that event or interrupt arrives kvm_vcpu_block() makes the request. This is in contrast to when kvm_vcpu_block() returns due to any other reason, such as a pending signal, which does not indicate the VCPU's halt emulation should stop, and therefore does not make the request.

KVM_REQ_OUTSIDE_GUEST_MODE

> This "request" ensures the target vCPU has exited guest mode prior to the sender of the request continuing on. No action needs be taken by the target, and so no request is actually logged for the target. This request is similar to a "kick", but unlike a kick it guarantees the vCPU has actually exited guest mode. A kick only guarantees the vCPU will exit at some point in the future, e.g. a previous kick may have started the process, but there's no guarantee the to-be-kicked vCPU has fully exited guest mode.

KVM_REQUEST_MASK

VCPU requests should be masked by KVM_REQUEST_MASK before using them with bitops. This is because only the lower 8 bits are used to represent the request's number. The upper bits are used as flags. Currently only two flags are defined.

VCPU Request Flags

KVM_REQUEST_NO_WAKEUP

> This flag is applied to requests that only need immediate attention from VCPUs running in guest mode. That is, sleeping VCPUs do not need to be awaken for these requests. Sleeping VCPUs will handle the requests when they are awaken later for some other reason.

KVM_REQUEST_WAIT

> When requests with this flag are made with kvm_make_all_cpus_request(), then the caller will wait for each VCPU to acknowledge its IPI before proceeding. This flag only applies to VCPUs that would receive IPIs. If, for example, the VCPU is sleeping, so no IPI is necessary, then the requesting thread does not wait. This means that this flag may be safely combined with KVM_REQUEST_NO_WAKEUP. See "Waiting for Acknowledgements" for more information about requests with KVM_REQUEST_WAIT.

VCPU Requests with Associated State

Requesters that want the receiving VCPU to handle new state need to ensure the newly written state is observable to the receiving VCPU thread's CPU by the time it observes the request. This means a write memory barrier must be inserted after writing the new state and before setting the VCPU request bit. Additionally, on the receiving VCPU thread's side, a corresponding read barrier must be inserted after reading the request bit and before proceeding to read the new state associated with it. See scenario 3, Message and Flag, of [lwn-mb] and the kernel documentation [memory-barriers].

The pair of functions, kvm_check_request() and kvm_make_request(), provide the memory barriers, allowing this requirement to be handled internally by the API.

Ensuring Requests Are Seen

When making requests to VCPUs, we want to avoid the receiving VCPU executing in guest mode for an arbitrary long time without handling the request. We can be sure this won't happen as long as we ensure the VCPU thread checks kvm_request_pending() before entering guest mode and that a kick will send an IPI to force an exit from guest mode when necessary. Extra care must be taken to cover the period after the VCPU thread's last kvm_request_pending() check and before it has entered guest mode, as kick IPIs will only trigger guest mode exits for VCPU threads that are in guest mode or at least have already disabled interrupts in order to prepare to enter guest mode. This means that an optimized implementation (see "IPI Reduction") must be certain when it's safe to not send the IPI. One solution, which all architectures except s390 apply, is to:

- set `vcpu->mode` to IN_GUEST_MODE between disabling the interrupts and the last kvm_request_pending() check;
- enable interrupts atomically when entering the guest.

This solution also requires memory barriers to be placed carefully in both the requesting thread and the receiving VCPU. With the memory barriers we can exclude the possibility of a VCPU thread observing !kvm_request_pending() on its last check and then not receiving an IPI for the next request made of it, even if the request is made immediately after the check. This is done by way of the Dekker memory barrier pattern (scenario 10 of [lwn-mb]). As the Dekker pattern requires two variables, this solution pairs `vcpu->mode` with `vcpu->requests`. Substituting them into the pattern gives:

```
CPU1                                      CPU2
=================                         =================
local_irq_disable();
WRITE_ONCE(vcpu->mode, IN_GUEST_MODE);    kvm_make_request(REQ, vcpu);
smp_mb();                                 smp_mb();
if (kvm_request_pending(vcpu)) {          if (READ_ONCE(vcpu->mode) ==
                                              IN_GUEST_MODE) {
    ...abort guest entry...                   ...send IPI...
}                                         }
```

As stated above, the IPI is only useful for VCPU threads in guest mode or that have already disabled interrupts. This is why this specific case of the Dekker pattern has been extended to disable interrupts before setting `vcpu->mode` to IN_GUEST_MODE. WRITE_ONCE() and READ_ONCE() are used to pedantically implement the memory barrier pattern, guaranteeing the compiler doesn't interfere with `vcpu->mode`'s carefully planned accesses.

IPI Reduction

As only one IPI is needed to get a VCPU to check for any/all requests, then they may be coalesced. This is easily done by having the first IPI sending kick also change the VCPU mode to something !IN_GUEST_MODE. The transitional state, EXITING_GUEST_MODE, is used for this purpose.

Waiting for Acknowledgements

Some requests, those with the KVM_REQUEST_WAIT flag set, require IPIs to be sent, and the acknowledgements to be waited upon, even when the target VCPU threads are in modes other than IN_GUEST_MODE. For example, one case is when a target VCPU thread is in READING_SHADOW_PAGE_TABLES mode, which is set after disabling interrupts. To support these cases, the KVM_REQUEST_WAIT flag changes the condition for sending an IPI from checking that the VCPU is IN_GUEST_MODE to checking that it is not OUTSIDE_GUEST_MODE.

Request-less VCPU Kicks

As the determination of whether or not to send an IPI depends on the two-variable Dekker memory barrier pattern, then it's clear that request-less VCPU kicks are almost never correct. Without the assurance that a non-IPI generating kick will still result in an action by the receiving VCPU, as the final kvm_request_pending() check does for request-accompanying kicks, then the kick may not do anything useful at all. If, for instance, a request-less kick was made to a VCPU that was just about to set its mode to IN_GUEST_MODE, meaning no IPI is sent, then the VCPU thread may continue its entry without actually having done whatever it was the kick was meant to initiate.

One exception is x86's posted interrupt mechanism. In this case, however, even the request-less VCPU kick is coupled with the same local_irq_disable() + smp_mb() pattern described above; the ON bit (Outstanding Notification) in the posted interrupt descriptor takes the role of `vcpu->requests`. When sending a posted interrupt, PIR.ON is set before reading `vcpu->mode`; dually, in the VCPU thread, vmx_sync_pir_to_irr() reads PIR after setting `vcpu->mode` to IN_GUEST_MODE.

Additional Considerations

Sleeping VCPUs

VCPU threads may need to consider requests before and/or after calling functions that may put them to sleep, e.g. kvm_vcpu_block(). Whether they do or not, and, if they do, which requests need consideration, is architecture dependent. kvm_vcpu_block() calls kvm_arch_vcpu_runnable() to check if it should awaken. One reason to do so is to provide architectures a function where requests may be checked if necessary.

Clearing Requests

Generally it only makes sense for the receiving VCPU thread to clear a request. However, in some circumstances, such as when the requesting thread and the receiving VCPU thread are executed serially, such as when they are the same thread, or when they are using some form of concurrency control to temporarily execute synchronously, then it's possible to know that the request may be cleared immediately, rather than waiting for the receiving VCPU thread to handle the request in VCPU RUN. The only current examples of this are kvm_vcpu_block() calls made by VCPUs to block themselves. A possible side-effect of that call is to make the KVM_REQ_UNHALT request, which may then be cleared immediately when the VCPU returns from the call.

References

[atomic-ops] Documentation/atomic_bitops.txt and Documentation/atomic_t.txt

[memory-barriers] Documentation/memory-barriers.txt

[lwn-mb] (*1*,*2*) https://lwn.net/Articles/573436/