FIFO-based Event Channel ABI

David Vrabel <david.vrabel@citrix.com>

Draft H

Contents

1 Introduction

1.1 Revision History

Version	Date	Changes
Draft A	4 Feb 2013	Initial draft.
Draft B	15 Feb 2013	Clarified that the event array is per-domain. Control block is no longer part of the vcpu_info but does reside in the same page. Hypercall changes: structures are now 32/64-bit clean, added notes on handling failures, `expand_array` has its `vcpu` field removed, use `expand_array` to add first page. Added an upcall section. Added a READY field to the control block to make finding the highest priority non-empty event queue more efficient. Note that memory barriers will be required but leave the details to a future draft.
Draft C	19 Mar 2013	Queue tail is now private to Xen. Guest pages are specified by MFN in the hypercalls. Updated link/unlink algorithm to avoid races when adding an event to a queue that is becoming empty.
Draft D	7 May 2013	Only Xen writes to HEAD. Expand the state diagram to include all sub-states. Added `link_bits` field to `struct evtchn_init_control`. Clarifications to the pseudocode.
Draft E	13 Sep 2013	Control blocks are no longer required to be in the same page as the vcpu_info structures. Add padding bytes to struct evtchn_init_control.
Draft F	27 Sep 2013	READY field is now 32-bits wide. DOMCTL_set_max_evtchn replaces EVTCHNOP_set_limit. DomU now defaults to allowing an unlimited number of event channels. The toolstack should set a limit during domain creation.
Draft G	31 Oct 2013	Add section on memory barriers. EVTCHNOP_expand_array should be called after binding events. Guest clearing MASKED on a tail event is no longer valid, a hypercall is required. This allows the loop in the Xen's link() function to be bounded.
Draft H	8 Nov 2013	Block unmasking with a BUSY bit in the event word.

1.2 Purpose

Xen uses event channels to signal events (interrupts) to (fully or partially) paravirtualized guests. The current event channel ABI provided by Xen only supports up-to 1024 (for 32-bit guests) or 4096 (for 64-bit guests) event channels. This is limiting scalability as support for more VMs, VCPUs and devices is required.

Events also cannot be serviced fairly as information on the ordering of events is lost. This can result in events from some VMs experiencing (potentially significantly) longer than average latency.

The existing ABI does not easily allow events to have different priorities. Current Linux kernels prioritize the timer event by special casing this but this is not generalizable to more events. Event priorities may be useful for prioritizing MMIO emulation requests over bulk data traffic (such as network or disk).

This design replaces the existing event channel ABI with one that:

- is scalable to more than 100,000 event channels, with scope for increasing this further with minimal ABI changes.
- allows events to be serviced fairly.
- allows guests to use up-to 16 different event priorities.
- has an ABI that is the same regardless of the natural word size.

1.3 Design Map

A new event channel ABI requires changes to Xen and the guest kernels.

2 Design Considerations

2.1 Assumptions

- Atomic read-modify-write of 32-bit words is possible on all supported platforms. This can be with a linked-load / store-conditional (e.g., ARMv8's ldrx/strx) or a compare-and-swap (e.g., x86's cmpxchg).

2.2 Constraints

- The existing ABI must continue to be useable. Compatibilty with existing guests is mandatory.

2.3 Risks and Volatile Areas

- Should the 3-level proposal be merged into Xen then this design does not offer enough improvements to warrant the cost of maintaining three different event channel ABIs in Xen and guest kernels.
- The performance of some operations may be decreased. Specifically, retriggering an event now always requires a hypercall.

3 Architecture

3.1 Overview

The event channel ABI uses a data structure that is shared between Xen and the guest. Access to the structure is done with lock-less operations (except for some less common operations where the guest must use a hypercall). The guest is responsible for allocating this structure and registering it with Xen during VCPU bring-up.

Events are reported to a guest's VCPU using a FIFO *event queue*. There is a queue for each priority level and each VCPU.

Each event has a *pending* and a *masked* bit. The pending bit indicates the event has been raised. The masked bit is used by the guest to prevent delivery of that specific event.

4 High Level Design

4.1 Shared Event Data Structure

The shared event data structure has a per-domain *event array*, and a per-VCPU *control block*.

- *event array*: A logical array of *event words* (one per event channel) which contains the pending and mask bits and the link index for next event in the queue. The event array is shared between all of the guest's VCPUs.
- *control block*: This contains the meta data for the event queues: the *ready bits* and the *head index* and *tail index* for each priority level. Each VCPU has its own control block.

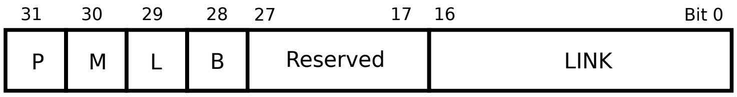

Figure 1: Event Array Word

4.1.1 Event Array

The pages within the event array need not be physically nor virtually contiguous, but the guest or Xen may make the virtually contiguous for ease of implementation. e.g., by using vmap() in Xen or vmalloc() in Linux. Pages are added by the guest as required to accomodate the event with the highest port number.

Field	Use
PENDING (P)	Event is pending
MASKED (M)	Event is masked
LINKED (L)	Event is on a queue
BUSY (B)	Xen is updating this word
LINK	The next event in the queue

Only 17 bits are currently defined for the LINK field, allowing 2^{17} (131,072) events. This limit can be trivially increased without any other changes to the ABI. Bits [28:17] are reserved for future expansion or for other uses.

4.1.2 Control Block

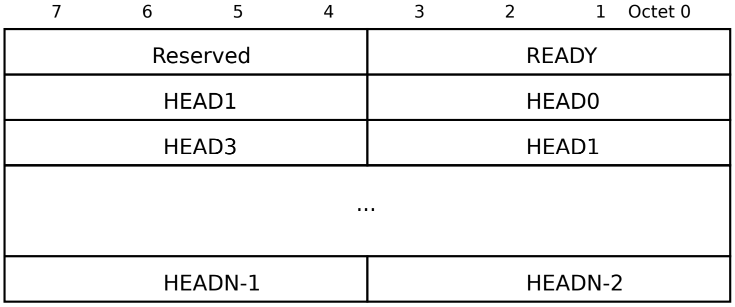

Figure 2: Control Block

The READY field contains a bit for each priority's queue. A set bit indicates that there are events pending on that queue. A queue's ready bit is set by Xen when an event is placed on an empty queue. The READY field is atomically read-and-cleared by the guest.

Bits 16 to 31 in the READY field are unused and will never be set by the hypervisor.

There are N HEAD indexes, one for each priority.

The HEAD index is the first event in the queue. empty. HEAD is only set by Xen when adding an event to an empty queue and is never set by the guest. If the queue is empty, HEAD may be zero or the last head index and the guest should not use the HEAD value until the queue has been set as READY again.

4.2 Event State Machine

Event channels are bound to a port in the domain using the existing ABI.

A bound event may be in one of three main states.

State	Abbrev.	PML Bits	Meaning
BOUND	B	000	The event is bound but not pending.
PENDING	P	100	The event has been raised and not yet acknowledged.
LINKED	L	101	The event is on an event queue.

The LINKED state has number of sub-states reflecting the events position in the list and how it may be reached by the guest.

Sub-state	Meaning
L_H	Pointed to by the control block's HEAD field.
L_GH	Pointed to by the guest's local head index.
L_L	Pointed to by the previous event in the queue.
LL_*	Event's link field points to another event.

Additionally, events may be UNMASKED or MASKED (M) in any state.

Valid transitions for UNMASKED events are shown in figure 3 on page 18.

MASKED events have all the same transitions except for:

- Ack (P → B). MASKED events should not be handled, so when a MASKED event is unlinked it should should remain PENDING until it is unmasked and then added to the event queue.
- Link (P → L). Since a MASKED event will not be handled, it does not needed to be added to the event queue until it becomes UNMASKED.

4.3 Event Queues

The event queues use a singly-linked list of event array words (see figure 1 and 4). Each VCPU has an event queue for each priority.

Each event queue has a *head* index stored in the control block and a *tail* index private to Xen. The head index is the index of the first element in the queue. The tail index is the last element in the queue. Every element within the queue has the L bit set.

The LINK field in the event word indexes the next event in the queue. LINK is zero for the last word in the queue.

The queue is empty when the head index is zero (zero is not a valid event channel).

4.4 Hypercalls

Three new EVTCHNOP hypercall sub-operations and one DOMCTL hypercall sub-operation are added:

- `EVTCHNOP_init_control`
- `EVTCHNOP_expand_array`
- `EVTCHNOP_set_priority`
- `DOMCTL_set_max_evtchn`

4.4.1 `EVTCHNOP_init_control`

This call initializes a single VCPU's control block.

A guest should call this during initial VCPU bring up.

If this call fails on the boot VCPU, the guest should continue to use the 2-level event channel ABI for all VCPUs. If this call fails on any non-boot VCPU then the VCPU will be unable to receive events and the guest should offline the VCPU.

> Note: This only initializes the control block. At least one page needs to be added to the event array with `EVTCHNOP_expand_array`.

```
struct evtchn_init_control {
    uint64_t control_gfn;
    uint32_t offset;
    uint32_t vcpu;
    uint8_t  link_bits;
    uint8_t  _pad[7];
};
```

Field	Purpose
`control_gfn`	[in] The GFN of the page containing the control block.
`offset`	[in] Offset in bytes from the start of the page to the beginning of the control block.
`vcpu`	[in] The VCPU number.
`link_bits`	[out] The number of valid bits of the LINK and HEAD fields. This will be same for all VCPUs but may change after a domain migrates.
`_pad`	Padding bytes for ABI compatibility between 32-bit and 64-bit guests. Ignored by the hypervisor.

Error code	Reason
EINVAL	`vcpu` is invalid or already initialized.
EINVAL	`control_gfn` is not a valid frame for the domain.
EINVAL	`offset` is not a multiple of 8 or the control block would cross a page boundary.
ENOMEM	Insufficient memory to allocate internal structures.

4.4.2 `EVTCHNOP_expand_array`

This call expands the event array by appending an additional page.

A guest may call this at any time after calling EVTCHNOP_init_control. Because it is not possible to known a event channel's port number until after it is bound, guests should call this after binding the event channel if the resulting port is not within the current event array.

It is not possible to shrink the event array once it has been expanded.

If this call fails, events outside of the array cannot be recieved and the guest should unbind any such events and report an error. If the first page cannot be added then the guest cannot receive any events and it should panic.

```
struct evtchn_expand_array {
    uint64_t array_gfn;
};
```

Field	Purpose
`array_gfn`	[in] The GFN of a page to be used for the next page of the event array.

Error code	Reason
EINVAL	`array_gfn` is not a valid frame for the domain.
ENOSPC	The event array already has the maximum number of pages.
ENOMEM	Insufficient memory to allocate internal structures.

4.4.3 `EVTCHNOP_set_priority`

This call sets the priority for an event channel. The event channel may be bound or unbound.

The meaning and the use of the priority are up to the guest. Valid priorities are 0 - 15 and the default is 7. 0 is the highest priority.

If the priority is changed on a bound event channel then at most one event may be signalled at the previous priority.

```
struct evtchn_set_priority {
    uint32_t port;
    uint32_t priority;
};
```

Field	Purpose
`port`	[in] The event channel.
`priority`	[in] The priority for the event channel.

Error code	Reason
EINVAL	`port` is invalid.
EINVAL	`priority` is outside the range 0 - 15.

4.4.4 `DOMCTL_set_max_evtchn`

This privileged call sets the highest port number a domain can bind an event channel to. The default for all domains is unlimited.

The limit only affects future attempts to bind event channels. Event channels that are already bound are not affected.

It is recommended that the toolstack only calls this during domain creation before the guest is started. A sensible default value for a toolstack to use is 1023 which means only a single page is required for the event array.

```
struct xen_domctl_set_max_evtchn {
    uint32_t max_port;
};
```

Field	Purpose
`max_port`	[in] The highest port number that the domain may bound an event channel to.

No sub-op specific errors are returned.

4.5 Memory Usage

4.5.1 Event Arrays

Xen needs to map every domains' event array into its address space. The space reserved for these global mappings is limited to 1 GiB on x86–64 (262144 pages) and is shared with other users.

It is non-trivial to calculate the maximum number of VMs that can be supported as this depends on the system configuration (how many driver domains etc.) and VM configuration. We can make some assuptions and derive an approximate limit.

Each page of the event array has space for 1024 events (E_P) so a regular domU will only require a single page. Since event channels have two ends, the upper bound on the total number of pages is 2 × number of VMs.

If the guests are further restricted in the number of event channels (E_V) then this upper bound can be reduced further. By assuming that each event event channel has one end in a domU and the other in dom0 (or a small number of driver domains) then the ends in dom0 will be packed together within the event array.

The number of VMs (V) with a limit of P total event array pages is approximately:

$$V = P \div \left(1 + \frac{E_V}{E_P}\right)$$

Using only half the available pages and limiting guests to only 64 events gives:

$$\begin{aligned} V &= (262144/2) \div (1 + 64/1024) \\ &= 123 \times 10^3 \text{ VMs} \end{aligned}$$

Alternatively, we can consider a system with D driver domains, each of which requires E_D events, and a dom0 using the maximum number of pages (128). The number of pages left over, hence the number of guests is:

$$V = P - \left(128 + D \times \frac{E_D}{E_P}\right)$$

With, for example, 16 driver domains each using the maximum number of pages:

$$\begin{aligned} V &= (262144/2) - (128 + 16 \times \frac{2^{17}}{1024}) \\ &= 129 \times 10^3 \text{ VMs} \end{aligned}$$

In summary, there is space to map the event arrays for over 100,000 VMs. This is more than the limit imposed by the 16 bit domain ID (∼32,000 VMs).

4.5.2 Control Block

With L priority levels and two 32-bit words for the head and tail indexes, the amount of space (S) required for the control block is:

$$\begin{aligned} S &= L \times 2 \times 4 + 8 \\ &= 16 \times 2 \times 4 + 8 \\ &= 136 \text{ bytes} \end{aligned}$$

5 Low Level Design

In the pseudo code in this section, all memory accesses are atomic, including those to bit-fields within the event word. All memory accesses are considered to be strongly ordered.

The following variables are used for the shared and selected local data structures. Lowercase variables are local.

Variable	Purpose
E	Event array.
C	Per-VCPU control block.
T	Tail index array (local to Xen).
H	Head index array (local to the guest).

5.1 Raising an Event

When Xen raises an event it marks it pending and (if it is not masked) adds it tail of event queue.

This needs to handle two main cases: the queue is empty or it is not empty. The link() function atomically ensures that the link field is only updated if the queue is non-empty.

The loop in link() must be bounded or a malicious or buggy guest could cause it to spin indefinately by repeated writes to the word. There are four valid writes a guest may do:

- clear PENDING
- clear LINKED and LINK
- set MASKED
- clear MASKED

Xen sets the BUSY bit to indicate to the guest that it may not clear the MASKED bit. The guest should wait until the BUSY bit is clear. The remaining three valid writes limit the cmpxchg() to at most 3 failures.

```
function link(t, p)
    i = 0
    E[t].busy = 1
    w = E[t]
    do
        if not w.linked
            return false
        o = n = w
        n.link = p
        b.busy = 0
        w = cmpxchg(E + t, o, n)
        i = i + 1
    while w != o and i < 4
    return true

function raise(q, p)
    E[p].pending = 1
    if not E[p].masked and not E[p].linked
        E[p].linked = 1
        if T[q] != p
            linked = link(T[q], p)
        else
            linked = false
        if not linked
            C.head[q] = p
        T[q] = p
```

Concurrent access by Xen to the event queue must be protected by a per-event queue spin lock.

5.2 Consuming Events

The guest consumes events starting at the head until it reaches the tail. Events in the queue that are not pending or are masked are consumed but not handled.

The unlink() function atomically clears LINKED and LINK and returns the LINK field.

To consume a single event:

```
function unlink(p)
    w = E[p]
    do
        o = n = w
        n.linked = false
        n.link = 0
        w = cmpxchg(E + p, o, n)
    while w != o
    return w.link

function handle_one_event(q)
    p = H[q]
    if p == 0
        p = C.head[q]
    link = unlink(p)
    H[q] = link
    if E[p].pending and not E[p].masked
        handle(p)
    return link == 0
```

handle() clears `E[p].pending` and EOIs level-triggered PIRQs.

> Note: When the event queue contains a single event we do not set the head as this would race with Xen adding a new event and setting the head.

5.3 Upcall

When Xen places an event on an empty queue it sets the queue as ready in the control block. If the ready bit transitions from 0 to 1, a new event is signalled to the guest.

The guest uses the control block's ready field to find the highest priority queue with pending events. The ready field is atomically read and cleared and or'd with a local copy.

Higher priority events do not need to preempt lower priority event handlers so the guest can handle events by taking one event off the currently ready queue with highest priority.

```
function upcall()
    r = xchg(C.ready, 0)
    while r
        q = find_first_set_bit(r)
        empty = handle_one_event(q)
        if empty
            clear_bit(q, r)
        r |= xchg(C.ready, 0)
```

Since the upcall is reentrant the guest should ensure that nested upcalls return immediately without processing any events. A per-VCPU nesting count may be used for this.

5.4 Masking Events

Events are masked by setting the masked bit. If the event is pending and linked it does not need to be unlinked.

```
E[p].masked = 1
```

5.5 Unmasking Events

Events are unmasked by the guest by clearing the masked bit. If the event is pending the guest must call the event channel unmask hypercall so Xen can link the event into the correct event queue.

The guest is not permitted to clear MASKED if BUSY is set. It should spin until BUSY is clear.

```
function clear_masked(p)
    w = E[p]
    do
        o = w
        o.busy = 0
        n = o
        n.masked = 0
        w = cmpxchg(E + p, o, n)
    while w != o

clear_masked(p)
if E[p].pending
    hypercall(EVTCHNOP_unmask)
```

The expectation here is that unmasking a pending event will be rare, so the performance hit of the hypercall is minimal.

> Note: After clearing the mask bit, the event may be raised and thus it may already be linked by the time the hypercall is done. The mask must be cleared before testing the pending bit to avoid racing with the event becoming pending.

5.6 Memory Barriers

`cmpxchg()` must have full memory barriers on both sides of the operation.

In Xen, a write memory barrier is required between writing to `C[q].head` and setting the `C.ready[q]` bit.

In the guest, a read memory barrier is required between reads of `C.ready` and `C[q].head`.

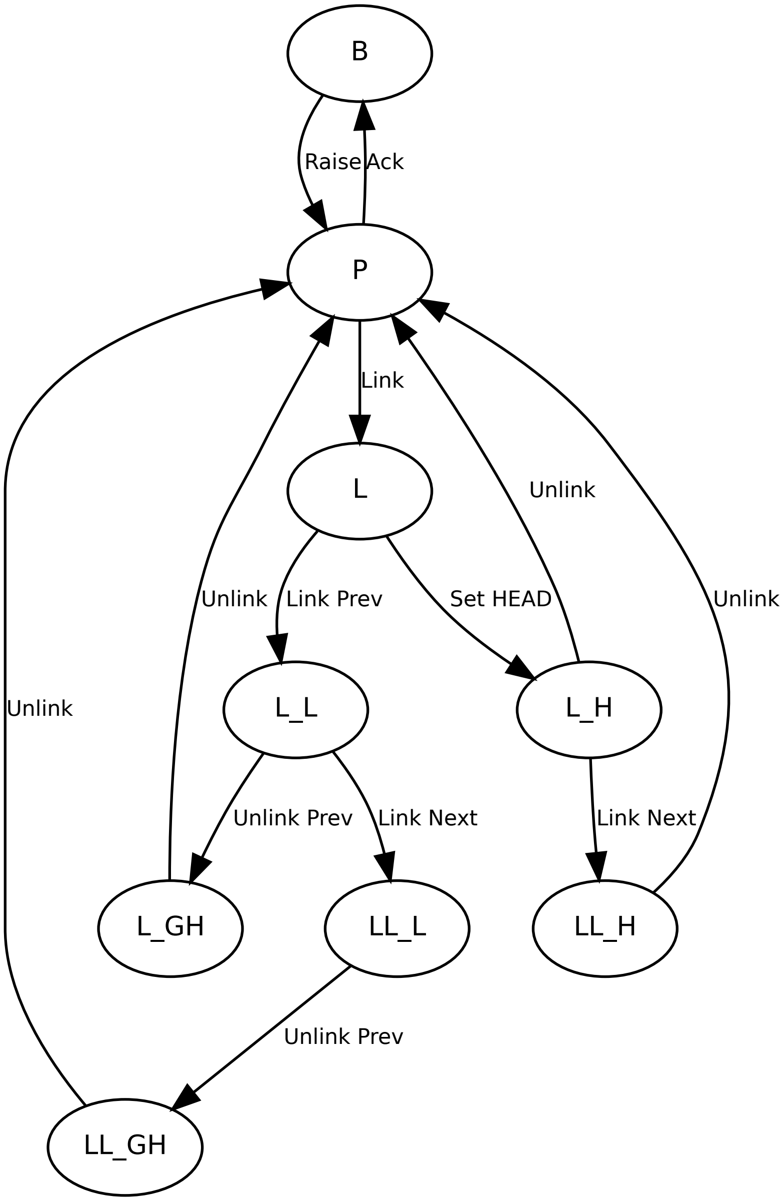

Figure 3: Event State Machine

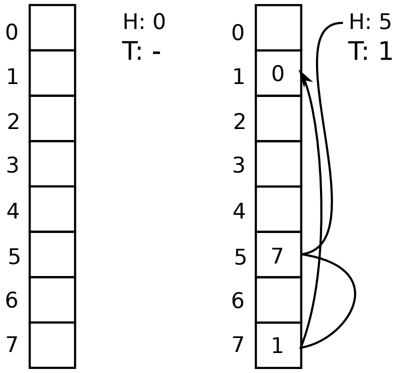

Figure 4: Empty and Non-empty Event Queues