NOVA Microhypervisor Interface Specification

Udo Steinberg
Technische Universität Dresden
Operating Systems Group
udo@hypervisor.org

January 18, 2011

Contents

Part I

Introduction

1 System Architecture

The **NOVA OS V**irtualization **A**rchitecture facilitates the coexistence of multiple legacy guest operating systems and a multi-server user environment on a single platform. The core system leverages virtualization technology provided by recent x86 platforms and comprises the hypervisor and a virtual-machine monitor.

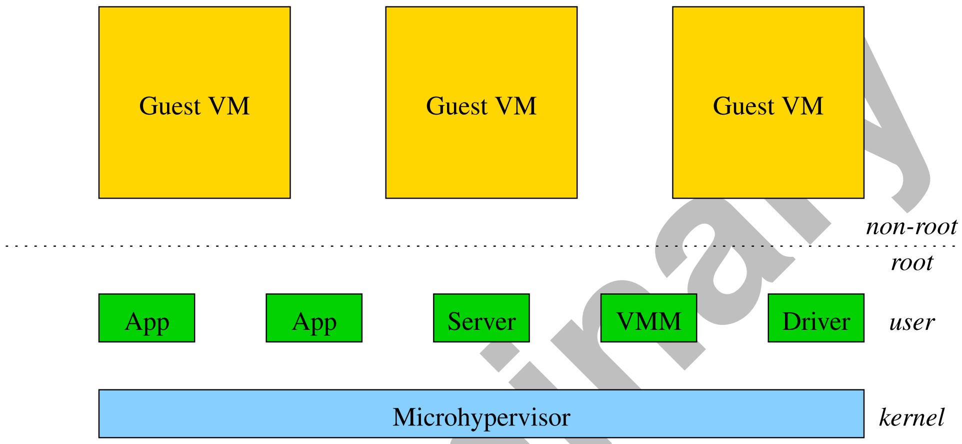


Figure 1.1: System Architecture

Figure 1.1 shows the structure of the system. The hypervisor is the only component running in privileged root/kernel mode. It isolates the user-level servers, including the virtual-machine monitor, from one another by placing them in different address spaces in unprivileged root/user mode. Each legacy guest operating system runs in its own virtual-machine environment in non-root mode and is therefore isolated from the other components. On platforms that do not provide hardware support for virtualization, legacy operating systems can be supported through use of address spaces and paravirtualization.

Besides isolation, the hypervisor also provides mechanisms for partitioning and delegation of platform resources, such as CPU time, physical memory, I/O ports and hardware interrupts and for establishing communication paths between different protection domains.

The virtual-machine monitor handles virtualization faults and implements virtual devices that enable legacy guest operating systems to function in the same manner as they would on bare hardware. Providing this functionality outside the hypervisor in the VMM considerably reduces the size of the trusted computing base for the multi-server user environment and for applications that do not require virtualization support.

The architecture and interfaces of the VMM and the multi-server user environment are not described in this document.

Part II

Basic Abstractions

2 Kernel Objects

2.1 Protection Domain

1. The protection domain (PD) is a unit of protection and isolation.
2. A protection domain is referenced by a protection domain capability $\mathrm{CAP}_{\mathrm{PD}}$ (4.1).
3. A protection domain is composed of a set of spaces that hold capabilities to platform resources or kernel objects that can be accessed by execution contexts within the protection domain. These spaces are:
    - Memory Space
    - I/O Space
    - Object Space
4. The memory space of a protection domain holds capabilities to page frames.
5. The I/O space of a protection domain holds capabilities to I/O ports.
6. The object space of a protection domain holds capabilities to the following kernel objects:
    - Protection Domain
    - Execution Context
    - Scheduling Context
    - Portal
    - Semaphore

2.2 Execution Context

1. The execution context (EC) is an abstraction for an activity within a protection domain.
2. An execution context is referenced by an execution context capability $\mathrm{CAP}_{\mathrm{EC}}$ (4.1).
3. The protection domain in which an execution context was created holds in its object space an EC capability referencing that execution context.
4. An execution context is permanently bound to the protection domain in which it was created.
5. An execution context optionally has a scheduling context bound to it.
6. There exist two flavors of execution context:
    - Kernel thread
    - Virtual CPU
7. An execution context comprises the following information:
    - Reference to protection domain (2.1)

- Scheduling context (2.3)
- UTCB (4.7)
- Event Selector Base (3.3)
- CPU registers (architecture dependent)
- FPU registers (architecture dependent)
- Reply capability register (4.1)

2.3 Scheduling Context

1. The scheduling context (SC) is a unit of dispatching and prioritization.
2. A scheduling context is referenced by a scheduling context capability CAP_{SC} (4.1).
3. The protection domain in which a scheduling context was created holds in its object space an SC capability referencing that scheduling context.
4. At any point in time, a scheduling context is bound to exactly one execution context.
5. Donation of a scheduling context to another execution context binds the scheduling context to that other execution context.
6. A scheduling context is permanently bound to exactly one CPU.
7. A scheduling context comprises the following information:
    - Reference to execution context (2.2)
    - Time quantum
    - Priority

2.4 Portal

1. A portal (PT) represents a dedicated entry point into the protection domain in which the portal was created.
2. A portal is referenced by a portal capability CAP_{PT} (4.1).
3. The protection domain in which a portal was created holds in its object space a portal capability referencing that portal.
4. A portal is permanently bound to exactly one execution context.
5. A portal comprises the following information:
    - Reference to execution context (2.2)
    - Message transfer descriptor (4.4)
    - Entry instruction pointer
    - Portal identifier

2.5 Semaphore

1. A semaphore (SM) provides a means to synchronize execution and interrupt delivery by selectively blocking and unblocking execution contexts.
2. A semaphore is referenced by a semaphore capability CAP_{SM} (4.1).
3. The protection domain in which a semaphore was created holds in its object space a semaphore capability referencing that semaphore.

3 Mechanisms

3.1 Scheduling

The microhypervisor implements a round-robin scheduler with multiple priority levels. Whenever an execution context is ready to execute, the runqueue contains the highest-priority scheduling context bound to that execution context. When an execution context blocks, the hypervisor removes the corresponding scheduling context from the runqueue.

When the hypervisor needs to make a scheduling decision, it selects the highest-priority scheduling context from the runqueue and dispatches the execution context bound to that scheduling context.

The parameters of a scheduling context influence the scheduling behavior of the system as follows:

- The priority defines the importance of a scheduling context. A higher-priority scheduling context always has precedence and immediately preempts a lower-priority scheduling context.
- The time quantum defines the number of microseconds that the execution context bound to the scheduling context can utilize the CPU when it is dispatched. A dispatched execution context consumes the time quantum of its scheduling context until the quantum reaches zero; at that point the execution context is descheduled until the time quantum has been replenished.

3.2 Communication

Message passing between protection domains is governed by portals. Each portal represents a dedicated entry point into the protection domain where the portal was created. An execution context in a protection domain can traverse any portal for which the protection domain holds a capability. Portal capabilities can be delegated to establish cross-domain communication channels.

To initiate a message-passing operation from one protection domain to another, the caller execution context presents a capability selector for a capability referencing the destination portal and a message transfer descriptor of the data to be transmitted to the hypervisor. The hypervisor looks up the portal in the object space of the source protection domain, determines the destination protection domain and loads the entry instruction pointer for that domain from the portal.

An arbitrary number of portals can be bound to a callee execution context in a protection domain. The callee provides the stack for handling one incoming request on any of these portals. If the callee is busy handling another request at the time of portal traversal, the caller blocks until the callee becomes available, unless the nonblocking flag has been specified.

Once the callee is available to handle a request and a caller exists for any portal bound to the callee, the hypervisor arranges a rendezvous and transfers the specified message from the caller to the callee.

If the request established a reply capability for the callee, the callee may subsequently respond directly to the caller through a reply operation without risking to block, because the caller is already waiting for the response.

The following forms of message passing are currently supported:

Nondonating Call

During a nondonating call, the caller execution context traverses the destination portal, rendezvouses with a callee execution context and transfers a message to it. The hypervisor establishes a reply capability in the callee protection domain. The caller does not donate the current scheduling context to the callee. The caller blocks on the instruction following the hypercall. The callee may later invoke the reply capability to send a response directly to the blocked caller. Upon receiving the response the caller becomes unblocked.

Donating Call

A donating call differs from a nondonating call in that the caller donates the current scheduling context to the callee. The donation mechanism implements priority and bandwidth inheritance from the caller to the callee. The caller blocks on the instruction following the hypercall and the callee starts executing immediately. The hypervisor also establishes a reply capability in the callee protection domain. When the callee later invokes that reply capability to send a response, the hypervisor returns the previously donated scheduling context from the callee back to the caller. Upon receiving the response the caller becomes unblocked.

Reply and Wait

The reply-and-wait operation sends a message back to the caller identified by the reply capability and revokes that capability. If the reply capability was established by a donating call, the hypervisor returns the previously donated scheduling context back to the caller. The callee blocks until the next request arrives.

3.3 Exceptions and Intercepts

When an execution context triggers a hardware exception or VM intercept, the hypervisor adds the exception number or intercept reason to SEL_{EVT} of the affected EC. If the resulting capability selector refers to a portal capability CAP_{PT}, the hypervisor arranges an implicit *dcall* hypercall for the execution context through the corresponding portal; otherwise the execution context is shut down.

The entire handling of the exception or intercept is performed using the current scheduling context of the execution context that triggered the event. Furthermore, that execution context remains blocked until the handler has replied with a message to resolve the exception or intercept.

The number of capability selectors used for exception and intercept handling is conveyed in the hypervisor information page (6.2). The translation of hardware exception numbers and intercept reasons to capability selectors is described in the processor-specific ABI (IV).

3.4 Interrupts

The hypervisor provides a semaphore per global system interrupt (GSI). An execution context waits for an interrupt by performing a *semctl(down)* hypercall to block on the corresponding semaphore. When the interrupt occurs, the hypervisor issues a *semctl(up)* operation for the semaphore.

3.5 Capability Delegation

Delegation of capabilities from one protection domain to another is performed during inter-domain communication. The execution context that sends a message specifies in its message transfer descriptor which range of capabilites from the sender's protection domain it offers to the receiver. The receiver specifies which range of capabilities it is willing to accept and where they should be installed in the receiver's protection domain.

The hypervisor computes the intersection of the sender and receiver ranges and delegates only those capabilities that are covered by both ranges. If the capabilities have access permissions associated with them, the sender may optionally reduce the permissions so that the receiver obtains less privileged capabilities.

If the capability ranges of the sender and receiver differ in size, the capability hotspot, specified by the sender, is used for disambiguation as illustrated in Figure 3.1.

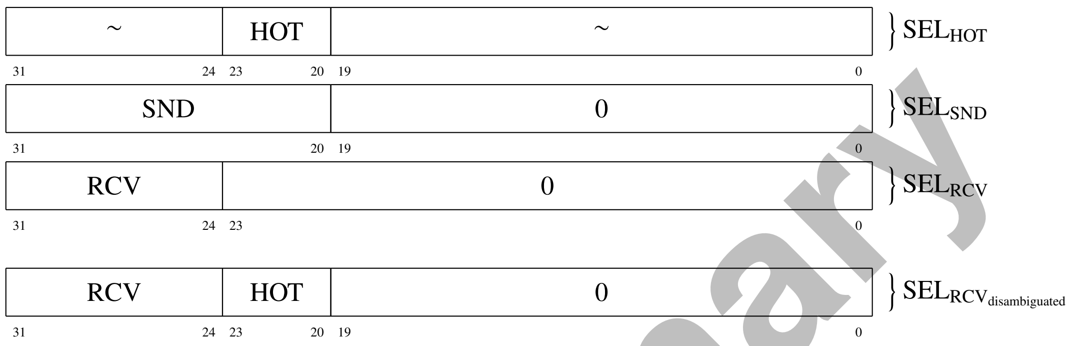


Figure 3.1: Capability Range Disambiguation

In this example, the sender has specified a capability range of order 20, starting at SEL_{SND}, whereas the receiver has specified a capability range of order 24, starting at SEL_{RCV}. There exist 2^4 possible locations in the receiver range, where the sender range could be delegated. Whenever two capability ranges differ in size, the hypervisor truncates the larger range by taking the ambiguous bits from the capability hotspot.

3.6 Capability Revocation

Accepting a capability delegation constitutes an implicit agreement that the capabilities may be revoked again at any time without the receiver's consent. Revoking a range of capabilities from a protection domain additionally revokes that range from all protection domains that directly or indirectly inherited it from that protection domain.

Part III

Application Programming Interface

4 Data Types

4.1 Capability

A capability (CAP) is a reference to a kernel object plus associated auxiliary data, such as access permissions. Capabilities are opaque and immutable to the user — they cannot be inspected, modified or addressed directly; instead user programs access a capability via a capability selector (4.2). All capabilities can be delegated and revoked as described in Section 3.5. The following types of capabilities exist:

4.1.1 Null Capability

A null capability $\text{CAP}_\emptyset$ does not reference anything and there are no permissions defined.

4.1.2 Memory Capability

A memory capability CAP_MEM references a 4KB page frame. It is stored in the memory space of a protection domain. The capability permissions are defined as follows:

1	1	x	w	r
4	3	2	1	0

r readable if set.

w writable if set.

x executable if set.

4.1.3 I/O Capability

An I/O capability $\text{CAP}_\text{I/O}$ references an I/O port. It is stored in the I/O space of a protection domain. The capability permissions are defined as follows:

1	1	1	1	a
4	3	2	1	0

a accessible if set.

4.1.4 Object Capability

An object capability references a kernel object. It is stored in the object space of a protection domain. The following types of object capabilities are currently defined:

4.1.4.1 Protection Domain Capability

A protection domain capability CAP_{PD} references a protection domain (2.1). The capability permissions are defined as follows:

sm	pt	sc	ec	pd
4	3	2	1	0

pd `create_pd` operation (5.3.1) permitted if set.

ec `create_ec` operation (5.3.2) permitted if set.

sc `create_sc` operation (5.3.3) permitted if set.

pt `create_pt` operation (5.3.4) permitted if set.

sm `create_sm` operation (5.3.5) permitted if set.

4.1.4.2 Execution Context Capability

An execution context capability CAP_{EC} references an execution context (2.2). The capability permissions are defined as follows:

1	1	1	1	1
4	3	2	1	0

4.1.4.3 Scheduling Context Capability

A scheduling context capability CAP_{SC} references a scheduling context (2.3). The capability permissions are defined as follows:

1	1	1	1	1
4	3	2	1	0

4.1.4.4 Portal Capability

A portal capability CAP_{PT} references a portal (2.4). The capability permissions are defined as follows:

1	1	1	1	1
4	3	2	1	0

4.1.4.5 Semaphore Capability

A semaphore capability CAP_{SM} references a semaphore (2.5). The capability permissions are defined as follows:

1	1	1	dn	up
4	3	2	1	0

up `semctl[up]` operation (5.4.2) permitted if set.

dn `semctl[down]` operation (5.4.2) permitted if set.

4.1.5 Reply Capability

A reply capability CAP_{RP} references a caller execution context. It is stored in the reply register of an execution context during inter-domain communication and automatically destroyed when invoked.

4.2 Capability Selector

A capability selector (SEL) is a user-visible abstract key for accessing a capability. The capability selector serves as integer index for the memory space, I/O space or object space of a protection domain. All capability selectors that do not refer to capabilities of another type refer to a null capability. For example, in Figure 4.1 capability selector 2 refers to a capability for an execution context.

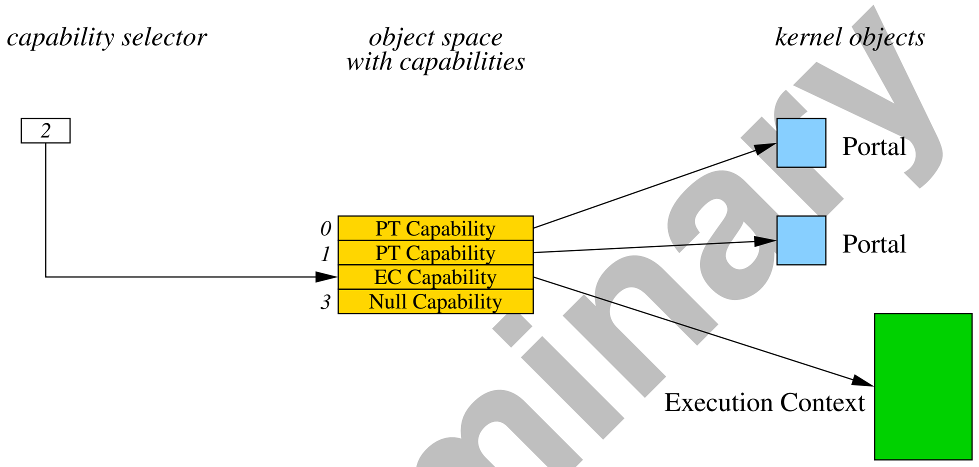


Figure 4.1: Capability Selector

4.3 Capability Range Descriptor

A capability range descriptor (CRD) refers to all capabilities of a particular type in the selector range Base ... Base + $2^{\text{Order}} - 1$. It must be naturally aligned such that Base ≡ 0 (mod 2^{Order}). During capability delegation, the permissions of the destination capability are computed as the logical AND of the permissions of the source capability and the permission mask from the capability range descriptor.

4.3.1 Null Capability Range Descriptor

A null capability range descriptor CRD_0 does not refer to any capabilities.

~	0
31 … 2	1 … 0

4.3.2 Memory Capability Range Descriptor

A memory capability range descriptor CRD_{MEM} refers to the memory capabilities located within the specified range of the memory space. Each memory capability covers 2^{12} bytes of memory.

Base	Order	Mask	1
31 … 12	11 … 7	6 … 2	1 … 0

4.3.3 I/O Capability Range Descriptor

An I/O capability range descriptor $\text{CRD}_{\text{I/O}}$ refers to the I/O capabilities located within the specified range of the I/O space.

Base	Order	Mask	2
31 … 12	11 … 7	6 … 2	1 … 0

4.3.4 Object Capability Range Descriptor

An object capability range descriptor CRD_{OBJ} refers to the delegatable capabilities located within the specified range of the object space.

Base	Order	Mask	3
31 … 12	11 … 7	6 … 2	1 … 0

4.4 Message Transfer Descriptor

The message transfer descriptor (MTD) is an architecture-specific bitfield that controls the contents of an exception or intercept message. The MTD is provided by the portal associated with the event and conveyed to the receiver in the exception or intercept message.

For each bit set to 1, the hypervisor transfers the processor state associated with that bit to or from the respective fields of the UTCB data area. The layout of the MTD and the fields in the UTCB data area are described in the processor-specific ABI (IV).

4.5 Quantum Priority Descriptor

The quantum priority descriptor (QPD) specifies the priority of a scheduling context and its time quantum in microseconds. It has the following format:

Time Quantum	0	Priority
31 – 12	11 – 8	7 – 0

Figure 4.2: Quantum Priority Descriptor

4.6 PCI Routing ID

The PCI Routing ID (RID) specifies the address of a PCI or PCI-E device and is composed of a bus number, a device number and a function number. It has the following format:

Bus	Device	Function
15 – 8	7 – 3	2 – 0

Figure 4.3: PCI Routing ID

With alternative routing-id interpretation (ARI), the format changes as follows:

Bus	Function
15 – 8	7 – 0

Figure 4.4: PCI Routing ID (ARI)

4.7 User Thread Control Block

Each execution context that acts as a kernel thread has an associated user thread control block (UTCB), which comprises a header area and a data area as illustrated in Figure 4.5.

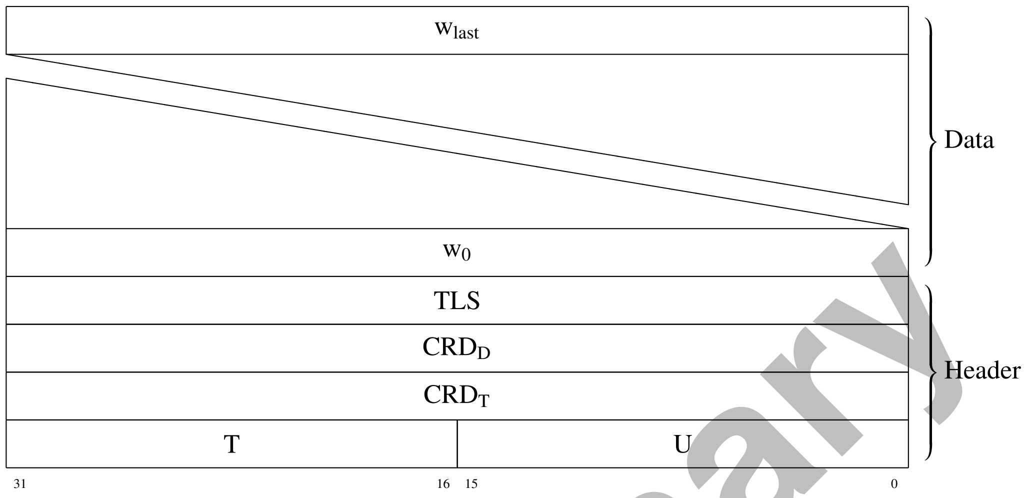


Figure 4.5: User Thread Control Block: General Layout

4.7.1 Header Area

The UTCB header fields are defined as follows:

U

Number of untyped items.

T

Number of typed items.

CRD_T

This capability range descriptor (4.3) specifies a receive window in the memory, I/O, or object space, in which the microhypervisor is allowed to perform capability translations. A null capability range descriptor effectively disables capability translations.

CRD_D

This capability range descriptor (4.3) specifies a receive window in the memory, I/O, or object space, in which the execution context is willing to accept capability delegations. A null capability range descriptor effectively disables capability delegations.

TLS

This field is never written by the hypervisor and can be used to store thread-local data.

4.7.2 Data Area

The size of the data area is defined by the size of the UTCB minus the size of the header area. An execution context uses its UTCB to send or receive messages, and to transfer typed items during capability delegation. The U and T fields in the UTCB header area define the number of untyped and typed items.

4.7.2.1 Untyped Items

The hypervisor transfers untyped items from the beginning of the UTCB data area upwards. Each untyped item occupies one message word as illustrated in Figure 4.6 For example, during a transfer of u untyped items, the hypervisor copies words $w_0 ... w_{u-1}$ from the UTCB data area of the sender to words $w_0 ... w_{u-1}$ in the UTCB data area of the receiver, without interpreting the contents of the message words.

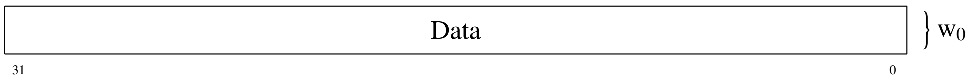


Figure 4.6: User Thread Control Block: Untyped Item

4.7.2.2 Typed Item

The hypervisor transfers typed items from the end of the UTCB data area downwards. Each typed item occupies two words. For example, during a transfer of t typed items, the hypervisor interprets words $w_{last} ... w_{last-2t+1}$ of the sender's UTCB data area. For each typed item in the sender UTCB, the hypervisor creates a corresponding typed item in the receiver UTCB. The following typed items are currently defined:

Translate:

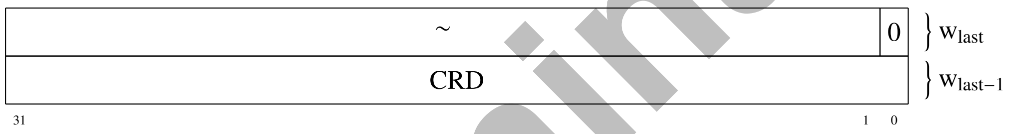


Figure 4.7: User Thread Control Block: Translate Item

If the type of the sender's CRD does not match the type of the receive window CRD_T in the receiver's UTCB header, the receiver obtains a transfer item with a null capability range descriptor.

Otherwise, the hypervisor attempts to translate the capability range specified by the base address and order in the sender protection domain to the corresponding capability range in the receiver protection domain from which it had been originally delegated. If the translation fails, e.g., because the sender range is not derived from the receiver range, the receiver obtains a transfer item with a null capability range descriptor. Otherwise the capability range descriptor describes the corresponding range in the receiver and the sender permissions for that range.

Delegate:

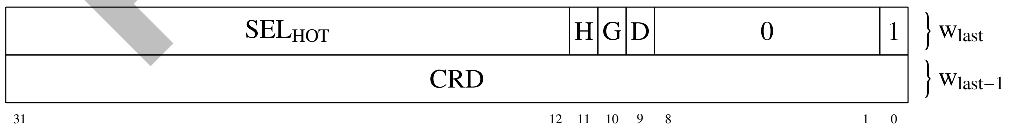


Figure 4.8: User Thread Control Block: Delegate Item

If the type of the sender's CRD does not match the type of the receive window CRD_D in the receiver's UTCB header, the receiver obtains a transfer item with a null capability range descriptor.

Otherwise, the hypervisor computes the range of capabilities to delegate from the sender to the receiver, using the hotspot SEL_{HOT} for range disambiguation, as described in Section 3.5. The capability range descriptor in the receiver's transfer item describes the contents of the receive window.

The root protection domain can control the source of a capability delegation as follows. For other protection domains this bit is ignored.

H If the bit is set, the source is the hypervisor. Otherwise the source is the protection domain itself.

For memory capability range descriptors (4.3.2), the following bits control which page tables are updated in addition to the host page table. For other capability range descriptors, these bits are ignored.

G The guest page table is updated if the bit is set.

D The DMA page table is updated if the bit is set.

5 Hypercalls

5.1 Definitions

Hypercall Numbers

Each hypercall is identified by a unique number. Figure 5.1 lists the currently defined hypercalls.

Number	Hypercall	Section
0x0	CALL	5.2.1
0x1	REPLY	5.2.2
0x2	CREATE_PD	5.3.1
0x3	CREATE_EC	5.3.2
0x4	CREATE_SC	5.3.3
0x5	CREATE_PT	5.3.4
0x6	CREATE_SM	5.3.5
0x7	REVOKE	5.3.6
0x8	LOOKUP	5.3.7
0x9	RECALL	5.4.1
0xa	SEMCTL	5.4.2
0xb	ASSIGN_PCI	5.5.1
0xc	ASSIGN_GSI	5.5.2

Figure 5.1: Hypercall Numbers

Status Codes

Figure 5.2 shows the status codes returned to indicate success or failure of a hypercall.

Number	Status Code
0x0	SUCCESS
0x1	TIMEOUT
0x2	BAD_SYS
0x3	BAD_CAP
0x4	BAD_MEM
0x5	BAD_FTR
0x6	BAD_CPU
0x7	BAD_DEV

Figure 5.2: Status Codes

5.2 Inter-Domain Communication

5.2.1 Call

Synopsis:

```
status = call (SEL_PT);
```

Parameters:

SEL$_{PT}$ Capability Selector in Caller PD: Target Portal

Flags:

0		DD	DB
3	2	1	0

DB Disable Blocking (0=blocking, 1=nonblocking)

DD Disable Donation (0=dcall, 1=ncall)

Description:

1. If the execution context (2.2), to which the target portal referenced by SEL$_{PT}$ is bound, is busy, the hypervisor considers the 'disable blocking' flag. If the flag is set, the hypercall returns with a timeout. Otherwise the caller blocks until the callee execution context becomes available.
2. The hypervisor transfers a message, whose contents is determined by the UTCB header, from the caller to the callee.
3. The hypervisor establishes a reply capability (4.1) in the reply register of the callee. The caller blocks until the callee invokes the reply capability. If the 'disable donation' flag is clear, the current scheduling context, previously bound to the caller, is donated and thereby bound to the callee.

Status:

SUCCESS

Hypercall completed successfully.

TIMEOUT

A rendezvous could not be arranged immediately.

BAD_CAP

SEL$_{PT}$ did not refer to a PT capability.

BAD_CPU

Caller and callee are not on the same CPU.

5.2.2 Reply

Synopsis:

```
reply();
```

Description:

1. If the reply register contains a reply capability, the hypervisor transfers a message, whose contents is determined by the UTCB header, to the caller execution context referenced by the reply capability.
2. If the caller had donated its scheduling context to the callee, the hypervisor binds that scheduling context back to the caller, thereby terminating the donation.
3. The hypervisor revokes the reply capability by replacing it with a null capability $\mathrm{CAP}_{\emptyset}$.
4. The callee blocks until a subsequent request arrives.

Status:

This hypercall does not return. Instead, when one of the portals bound to the execution context is called, the execution continues at the instruction pointer specified in that portal.

5.3 Capability Management

5.3.1 Create Protection Domain

Synopsis:

status = create_pd (SEL_0, SEL_{PD}, CRD_{OBJ});

Parameters:

SEL$_0$ Capability Selector in Target PD: Created PD

SEL$_{PD}$ Capability Selector in Caller PD: Target PD

CRD$_{OBJ}$ Object Capability Range Descriptor (4.3)

Description:

Creates a new protection domain in the target PD specified by SEL_{PD}. Prior to the hypercall SEL_0 must refer to a null capability. The target PD obtains in place of SEL_0 a protection domain capability that refers to the created PD. The hypervisor delegates the capability range, specified by CRD_{OBJ}, from the target PD to the created PD.

Status:

SUCCESS

Hypercall completed successfully.

BAD_CAP

SEL_0 did not refer to a null capability.
SEL_{PD} did not refer to a protection domain capability.
The protection domain capability has insufficient permissions.

5.3.2 Create Execution Context

Synopsis:

```
status = create_ec (SEL0, SELPD, CPU, UTCB, SP, SELEVT);
```

Parameters:

SEL$_0$ Capability Selector in Target PD: Created EC

SEL$_{PD}$ Capability Selector in Caller PD: Target PD

CPU CPU Number

UTCB Virtual Address: UTCB Pointer

SP Virtual Address: Stack Pointer

SEL$_{EVT}$ Capability Selector: Event Base

Flags:

0	G
3 1	0

G Global Thread (0=local, 1=global)

Description:

Creates a new execution context in the target PD specified by SEL$_{PD}$. Prior to the hypercall SEL$_0$ must refer to a null capability. The target PD obtains in place of SEL$_0$ an execution context capability that refers to the created EC. The hypervisor binds the execution context to the specified CPU. If the UTCB address is zero, the hypervisor creates a virtual CPU, otherwise it creates a thread according to the G flag. Local threads cannot have a scheduling context bound to them. They start running when they receive a request on a portal bound to them. Global threads and virtual CPUs generate a startup exception the first time a scheduling context is bound to them.

Status:

SUCCESS

Hypercall completed successfully.

BAD_CAP

SEL$_0$ did not refer to a null capability.
SEL$_{PD}$ did not refer to a protection domain capability.
The protection domain capability has insufficient permissions.

BAD_CPU

Invalid CPU number.

BAD_FTR

Virtual CPUs not supported.

BAD_MEM

Invalid UTCB address.

5.3.3 Create Scheduling Context

Synopsis:

status = create_sc (SEL$_0$, SEL$_{PD}$, SEL$_{EC}$, QPD);

Parameters:

SEL$_0$ Capability Selector in Target PD: Created SC

SEL$_{PD}$ Capability Selector in Caller PD: Target PD

SEL$_{EC}$ Capability Selector in Target PD: Target EC

QPD Quantum Priority Descriptor (4.5)

Description:

Creates a new scheduling context in the target PD specified by SEL$_{PD}$. Prior to the hypercall SEL$_0$ must refer to a null capability. The target PD obtains in place of SEL$_0$ a scheduling context capability that refers to the created SC. The hypervisor binds the scheduling context to the execution context referred to by SEL$_{EC}$ in the target PD and configures it with the parameters specified by QPD.

Status:

SUCCESS

Hypercall completed successfully.

BAD_CAP

SEL$_0$ did not refer to a null capability.
SEL$_{PD}$ did not refer to a protection domain capability.
SEL$_{EC}$ did not refer to an execution context capability.
The protection domain capability has insufficient permissions.
Binding the scheduling context to the execution context failed.

5.3.4 Create Portal

Synopsis:

```
status = create_pt (SEL₀, SEL_PD, SEL_EC, MTD_PT, IP);
```

Parameters:

SEL$_0$ Capability Selector in Target PD: Created PT

SEL$_{PD}$ Capability Selector in Caller PD: Target PD

SEL$_{EC}$ Capability Selector in Target PD: Target EC

MTD$_{PT}$ Message Transfer Descriptor (4.4)

IP Virtual Address: Instruction Pointer

Description:

Creates a new portal in the target PD specified by SEL$_{PD}$. Prior to the hypercall SEL$_0$ must refer to a null capability. The target PD obtains in place of SEL$_0$ a portal capability that refers to the created portal. The hypervisor binds the portal to the execution context referred to by SEL$_{EC}$ in the target PD.

Status:

SUCCESS

Hypercall completed successfully.

BAD_CAP

SEL$_0$ did not refer to a null capability.
SEL$_{PD}$ did not refer to a protection domain capability.
SEL$_{EC}$ did not refer to an execution context capability.
The protection domain capability has insufficient permissions.
Binding the portal to the execution context failed.

5.3.5 Create Semaphore

Synopsis:

status = create_sm (SEL_0, SEL_{PD}, CNT);

Parameters:

SEL_0 Capability Selector in Target PD: Created SM

SEL_{PD} Capability Selector in Caller PD: Target PD

CNT Unsigned: Initial Counter Value

Description:

Creates a new semaphore in the target PD specified by SEL_{PD}. Prior to the hypercall SEL_0 must refer to a null capability. The target PD obtains in place of SEL_0 a semaphore capability that refers to the created semaphore. The hypervisor initializes the semaphore counter with the value of CNT.

Status:

SUCCESS

Hypercall completed successfully.

BAD_CAP

SEL_0 did not refer to a null capability.
SEL_{PD} did not refer to a protection domain capability.
The protection domain capability has insufficient permissions.

5.3.6 Revoke Capability Range

Synopsis:

```
status = revoke (CRD);
```

Parameters:

CRD Capability Range Descriptor (4.3)

Flags:

0	SR
3 1	0

SR Self Revoke (0=only children, 1=including self)

Description:

Revokes the capabilities within the range specified by the capability range descriptor from all protection domains that directly or indirectly obtained these capabilities through delegation from the calling protection domain. If the self revoke bit is set, the capabilities will also be revoked from the calling protection domain itself. Once all capabilities to a kernel object have been revoked and no references to the kernel object exist anymore, the kernel object will be destroyed. This operation never fails but can take a long time to complete if there are many capabilities to revoke.

Status:

SUCCESS

Hypercall completed successfully.

5.3.7 Lookup Capability Range

Synopsis:

```
status = lookup (CRD);
```

Parameters:

CRD Capability Range Descriptor (4.3)

Description:

Looks up a range of capabilities in the caller's protection domain. The caller must specify a base address and type in the CRD prior to the hypercall. If a capability exists at the specified address, the hypervisor returns a completely filled CRD describing the capability range. Otherwise a null capability range descriptor is returned.

Status:

SUCCESS

Hypercall completed successfully.

5.4 Execution Control

5.4.1 Recall Execution Context

Synopsis:

```
status = recall (SEL_EC);
```

Parameters:

SEL$_{\text{EC}}$ Capability Selector in Caller PD: Execution Context

Description:

Pends an event for the specified execution context, which causes it to generate a recall exception before its next return from the hypervisor.

Status:

SUCCESS

Hypercall completed successfully.

BAD_CAP

SEL$_{\text{EC}}$ did not refer to an execution context capability.

5.4.2 Semaphore Control

Synopsis:

```
status = semctl (SEL_SM);
```

Parameters:

SEL$_{SM}$ Capability Selector in Caller PD: Semaphore

Flags:

3 – 2	1	0
0	ZC	OP

OP Operation (0=up, 1=down)

ZC Zero Counter (0=decrement, 1=set to zero)

Description:

The *down* operation blocks the calling execution context if the semaphore counter is zero, otherwise the counter is decremented or set to zero, depending on the setting of the ZC bit.

The *up* operation releases an execution context blocked on the semaphore if one exists, otherwise it increments the counter.

Status:

SUCCESS

Hypercall completed successfully.

BAD_CAP

SEL$_{SM}$ did not refer to a semaphore capability.
The semaphore capability has insufficient permissions.

5.5 Device Control

5.5.1 Assign PCI Device

Synopsis:

```
status = assign_pci (SEL_PD, RID_PF, RID_VF);
```

Parameters:

SEL$_{\text{PD}}$ Capability Selector in Caller PD: Target PD

RID$_{\text{PF}}$ PCI Routing ID: Physical Function (4.6)

RID$_{\text{VF}}$ PCI Routing ID: Virtual Function (4.6)

Description:

Assigns a PCI device to the specified target protection domain. RID$_{\text{PF}}$ identifies the physical function of the device. RID$_{\text{VF}}$ identifies the virtual function or must be set to 0.

Status:

SUCCESS

Hypercall completed successfully.

BAD_CAP

SEL$_{\text{PD}}$ did not refer to a protection domain capability.

BAD_DEV

RID$_{\text{PF}}$ or RID$_{\text{VF}}$ did not refer to a valid PCI device.

5.5.2 Assign Global System Interrupt

Synopsis:

```
status = assign_gsi (SEL_SM, CPU, RID);
```

Parameters:

SEL$_{SM}$ Capability Selector in Caller PD: Interrupt Semaphore

CPU CPU Number

RID PCI Routing ID (4.6)

Description:

Assigns the global system interrupt identified by SEL$_{SM}$ to the PCI device with the specified RID. The interrupt will be routed to the given CPU and signaled on the corresponding interrupt semaphore. For global system interrupts that are delivered through an IOAPIC, RID is ignored and should be set to 0. For devices that generate MSI or MSI-X directly to a local APIC, a misconfigured RID will cause interrupt remapping hardware to drop the interrupt.

In addition to the status, the hypercall returns a hint for the values that must be programmed into the MSI registers of the PCI device for proper operation. Refer to Section 7.5 for details.

Status:

SUCCESS

Hypercall completed successfully.

BAD_CAP

SEL$_{SM}$ did not refer to an interrupt semaphore capability.

BAD_CPU

Invalid CPU number.

6 Booting

6.1 Root Protection Domain

When the hypervisor has initialized the system, it creates the root protection domain with a root execution context and a root scheduling context.

At bootup the root protection domain is configured as follows:

Memory Space

Program Segments

The hypervisor loads the program segments of the roottask into the memory space as specified by the ELF program headers of the roottask image.

Hypervisor Information Page

The hypervisor information page is mapped into the memory space at a specific address that is passed to the root execution context during startup.

UTCB

The UTCB of the root execution context is mapped into the memory space just below the HIP.

All other regions of the memory space are initially empty.

I/O Space

The I/O space is initially empty.

Object Space

The object space contains the following capabilities:

- Capability selector EXC + 0 refers to the root PD capability.
- Capability selector EXC + 1 refers to the root EC capability.
- Capability selector EXC + 2 refers to the root SC capability.

All other capability selectors refer to null capabilities.

6.2 Hypervisor Information Page

The hypervisor information page conveys information about the platform and configuration to the root protection domain. The processor register that contains the virtual address of the hypervisor information page during booting is ABI-specific (IV). Figure 6.1 shows the layout of the hypervisor information page. All fields are unsigned values unless stated otherwise.

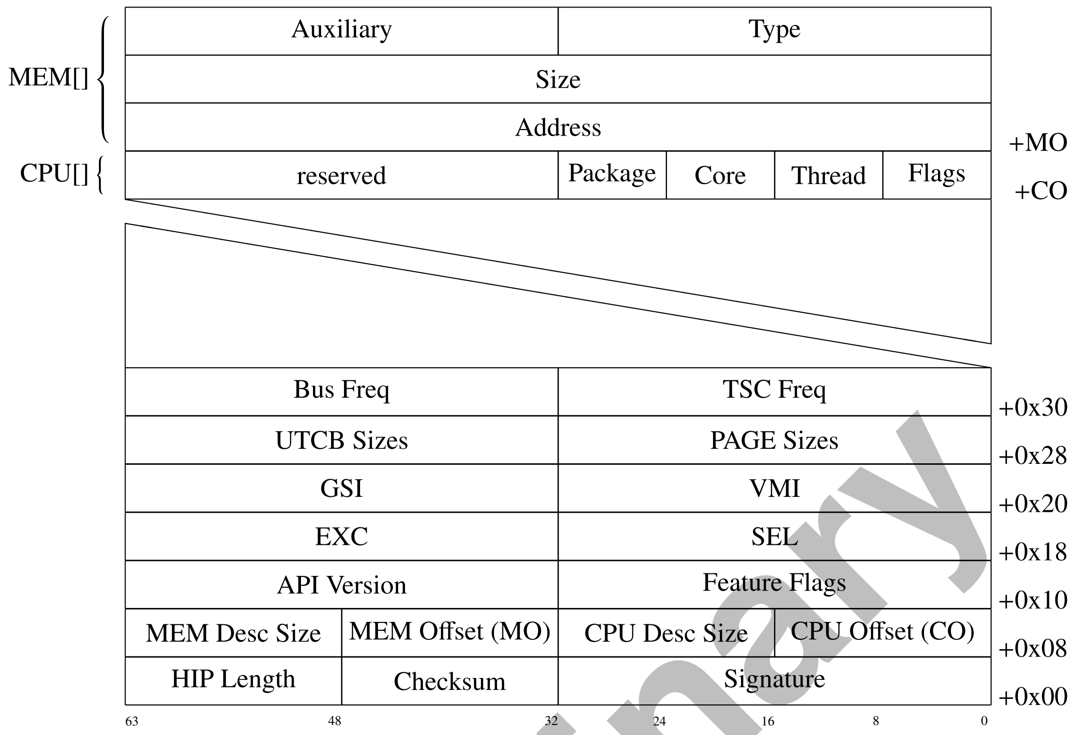


Figure 6.1: Hypervisor Information Page

Signature:

A value of 0x41564f4e identifies the NOVA microhypervisor.

Checksum:

The checksum is valid if 16bit-wise addition the HIP contents produces a value of 0.

HIP Length:

Length of the HIP in bytes. This includes all CPU and memory descriptors.

CPU Offset:

Offset of the first CPU descriptor in bytes, relative to the HIP base.

CPU Desc Size:

Size of a CPU descriptor in bytes. The number of CPU descriptors is equal to the number of CPUs supported and can be computed as follows: (MEM Offset - CPU Offset) / CPU Desc Size

MEM Offset:

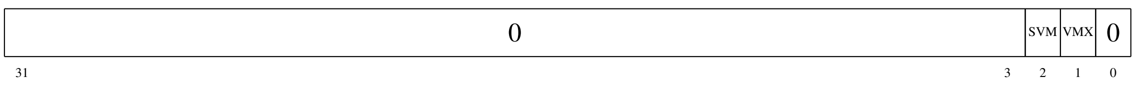
Offset of the first MEM descriptor in bytes, relative to the HIP base.

MEM Desc Size:

Size of a MEM descriptor in bytes. The number of MEM descriptors can be computed as follows: (HIP Length - MEM Offset) / MEM Desc Size

Feature Flags:

The hypervisor supports a particular feature if and only if the corresponding bit in the feature flags is set to 1. The following features are currently defined:



VMX:	Intel Virtual Machine Extensions
SVM:	AMD Secure Virtual Machine

API Version:

API version number.

SEL:

Number of available capability selectors in each object space. Specifying a capability selector beyond the maximum number supported wraps around to the beginning of the object space.

EXC:

Number of capability selectors used for exception handling (3.3).

VMI:

Number of capability selectors used for virtual-machine intercept handling (3.3).

GSI:

Number of global system interrupts (3.4).

PAGE Sizes:

If bit n is set, the implementation supports memory pages of size 2^n bytes.

UTCB Sizes:

If bit n is set, the implementation supports user thread control blocks of size 2^n bytes.

TSC Freq:

Time Stamp Counter Frequency in kHz.

BUS Freq:

Interconnect Frequency in kHz.

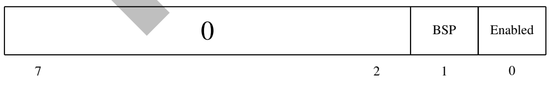
CPU Descriptor

Package, Core, Thread:

CPU multiprocessor topology information.

Flags:

CPU status flags.



MEM Descriptor

Address:

Physical base address of memory region.

Size:

Size of memory region in bytes.

Type:

Type of memory region. Note that the allocates ranges overlap the available ranges.

Type	Description	
-2	Multiboot Module	Allocated ranges
-1	Microhypervisor	
1	Available Memory	Available ranges
2	Reserved Memory	
3	ACPI Reclaim Memory	
4	ACPI NVS Memory	

Auxiliary:

Physical address of command line if type is 'Multiboot Module', reserved otherwise.

Part IV

Application Binary Interface

7 ABI x86-32

7.1 Initial State

Figure 7.1 details the state of the CPU registers when the hypervisor has finished booting and transfers control to the root protection domain.

Register	Description
CS	Selector=~, Base=0, Limit=0xFFFFFFFF, Code Segment, ro
SS,DS,ES,FS,GS	Selector=~, Base=0, Limit=0xFFFFFFFF, Data Segment, rw
EIP	Address of entry point from ELF header
ESP	Address of hypervisor information page
EAX,ECX,EDX,EBX,EBP,ESI,EDI	~
EFLAGS	0x202

Figure 7.1: Initial State

7.2 Event-Specific Capability Selectors

For the delivery of exception and intercept messages, the hypervisor performs an implicit portal traversal. The destination portal is determined by adding the event number to SEL_{EVT} of the affected execution context.

Exceptions

Number	Exception	Number	Exception	Number	Exception	Number	Exception
0x0	#DE	0x8	#DF[1]	0x10	#MF	0x18	reserved
0x1	#DB	0x9	reserved	0x11	#AC	0x19	reserved
0x2	reserved	0xa	#TS[1]	0x12	#MC[1]	0x1a	reserved
0x3	#BP	0xb	#NP	0x13	#XM	0x1b	reserved
0x4	#OF	0xc	#SS	0x14	reserved	0x1c	reserved
0x5	#BR	0xd	#GP	0x15	reserved	0x1d	reserved
0x6	#UD	0xe	#PF	0x16	reserved	0x1e	STARTUP
0x7	#NM[1]	0xf	reserved	0x17	reserved	0x1f	RECALL

VMX Intercepts

Number	Intercept	Number	Intercept	Number	Intercept
0x0	Exception or NMI[1]	0x15	VMPTRLD	0x2a	reserved
0x1	INTR[1]	0x16	VMPTRST	0x2b	TPR Below Threshold
0x2	Triple Fault[2]	0x17	VMREAD	0x2c	APIC Access
0x3	INIT[2]	0x18	VMRESUME	0x2d	reserved
0x4	SIPI[2]	0x19	VMWRITE	0x2e	GDTR/IDTR Access
0x5	I/O SMI	0x1a	VMXOFF	0x2f	LDTR/TR Access
0x6	Other SMI	0x1b	VMXON	0x30	EPT Violation[2]
0x7	Interrupt Window	0x1c	CR Access[1]	0x31	EPT Misconfiguration[1]
0x8	NMI Window	0x1d	DR Access	0x32	INVEPT
0x9	Task Switch[2]	0x1e	I/O Access[2]	0x33	RDTSCP
0xa	CPUID[2]	0x1f	RDMSR[2]	0x34	VMX Preemption Timer
0xb	GETSEC[2]	0x20	WRMSR[2]	0x35	INVVPID
0xc	HLT[2]	0x21	Invalid Guest State[2]	0x36	WBINVD
0xd	INVD[2]	0x22	MSR Load Failure	0x37	XSETBV
0xe	INVLPG[1]	0x23	reserved	0x38	reserved
0xf	RDPMC	0x24	MWAIT	0x39	reserved
0x10	RDTSC	0x25	MTF	0x3a	reserved
0x11	RSM	0x26	reserved	0x3b	reserved
0x12	VMCALL	0x27	MONITOR	0x3c	reserved
0x13	VMCLEAR	0x28	PAUSE	0xfe	STARTUP
0x14	VMLAUNCH	0x29	Machine Check	0xff	RECALL

SVM Intercepts

Number	Intercept	Number	Intercept	Number	Intercept
0x0–0xf	CR Read	0x6e	RDTSC	0x81	VMMCALL
0x10–0x1f	CR Write	0x6f	RDPMC	0x82	VMLOAD[2]
0x20–0x2f	DR Read	0x70	PUSHF	0x83	VMSAVE[2]
0x30–0x3f	DR Write	0x71	POPF	0x84	STGI
0x40–0x5f	Exception[1]	0x72	CPUID	0x85	CLGI[2]
0x60	INTR[1]	0x73	RSM	0x86	SKINIT[2]
0x61	NMI[1]	0x74	IRET	0x87	RDTSCP
0x62	SMI	0x75	INT	0x88	ICEBP
0x63	INIT[2]	0x76	INVD[2]	0x89	WBINVD
0x64	Interrupt Window	0x77	PAUSE	0x8a	MONITOR
0x65	CR0 Selective Write	0x78	HLT[2]	0x8b	MWAIT
0x66	IDTR Read	0x79	INVLPG	0x8c	MWAIT (cond.)
0x67	GDTR Read	0x7a	INVLPGA	0x8d	reserved
0x68	LDTR Read	0x7b	I/O Access[2]	0x8e	reserved
0x69	TR Read	0x7c	MSR Access[2]	0x8f	reserved
0x6a	IDTR Write	0x7d	Task Switch	0xfc	NPT Fault[2]
0x6b	GDTR Write	0x7e	FERR Freeze	0xfd	Invalid Guest State[2]
0x6c	LDTR Write	0x7f	Triple Fault[2]	0xfe	STARTUP
0x6d	TR Write	0x80	VMRUN	0xff	RECALL

[1]These events do not currently cause a portal traversal, because the microhypervisor handles them internally.
[2]These events are currently force-enabled by the microhypervisor or by hardware.

7.3 Message Transfer Descriptor

Figure 7.2 illustrates the format of the architectural bitfield of the message transfer descriptor for exceptions and intercepts, as described in Section 4.4.

0			TSC	STA	INJ	CTRL	QUAL	SYS	DR	CR	IDTR	GDTR	LDTR	TR	CS SS	FS GS	DS ES	EFL	EIP	ESP	BSD	ACDB
22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 7.2: Message Transfer Descriptor: Architectural Bitfield

The individual bits transfer the following processor state to/from their respective UTCB fields (7.4).

ACDB General-Purpose Registers EAX, ECX, EDX, EBX

BSD General-Purpose Registers EBP, ESI, EDI

ESP Stack Pointer ESP

EIP Instruction Pointer EIP, Instruction Length

EFL Flags Register EFLAGS

DS ES Segment Registers DS and ES (Selector, Base, Limit, Access Rights)

FS GS Segment Registers FS and GS (Selector, Base, Limit, Access Rights)

CS SS Segment Registers CS and SS (Selector, Base, Limit, Access Rights)

TR Task Register (Selector, Base, Limit, Access Rights)

LDTR Local Descriptor Table Register (Selector, Base, Limit, Access Rights)

GDTR Global Descriptor Table Register (Base, Limit)

IDTR Interrupt Descriptor Table Register (Base, Limit)

CR Control Registers CR0, CR2, CR3, CR4

DR Debug Register DR7

SYS SYSENTER MSRs for CS, ESP, EIP

QUAL Exit Qualification

CTRL Execution Controls

INJ Injection Info, Injection Error Code

STA Interruptibility State, Activity State

TSC TSC Offset

7.4 UTCB Data Layout

reserved	IDTR Base	IDTR Limit	reserved		+0x110
reserved	GDTR Base	GDTR Limit	reserved		+0x100
reserved	TR Base	TR Limit	TR AR	TR Sel	+0xf0
reserved	LDTR Base	LDTR Limit	LDTR AR	LDTR Sel	+0xe0
reserved	GS Base	GS Limit	GS AR	GS Sel	+0xd0
reserved	FS Base	FS Limit	FS AR	FS Sel	+0xc0
reserved	DS Base	DS Limit	DS AR	DS Sel	+0xb0
reserved	SS Base	SS Limit	SS AR	SS Sel	+0xa0
reserved	CS Base	CS Limit	CS AR	CS Sel	+0x90
reserved	ES Base	ES Limit	ES AR	ES Sel	+0x80
SYSENTER EIP	SYSENTER ESP	SYSENTER CS	DR7		+0x70
CR4	CR3	CR2	CR0		+0x60
TSC Offset		Secondary Exit Ctrl	Primary Exit Ctrl		+0x50
Secondary Exit Qual		Primary Exit Qual			+0x40
EDI	ESI	EBP	ESP		+0x30
EBX	EDX	ECX	EAX		+0x20
Injection Error	Injection Info	Activity State	Interruptibility State		+0x10
EFLAGS	EIP	Instruction Length	MTD		

7.5 Calling Convention

The following pages describes the calling convention for each hypercall. An execution context calls into the hypervisor by loading the hypercall identifier and other parameters into the specified processor registers and then executes the *sysenter* instruction.

The hypercall identifier consists of the hypercall number and hypercall-specific flags, as illustrated in Figure 7.3.

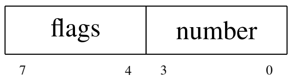


Figure 7.3: Hypercall Identifier

The status code returned from a hypercall has the format shown in Figure 7.4.

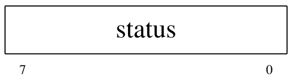


Figure 7.4: Status Code

The assignment of hypercall parameters to general-purpose registers is shown on the left side; the contents of the registers after the hypercall is shown on the right side.

Call

$SEL_{PT[31-8]}/hypercall_{[7-0]}$	EAX	**— Call ⟶**	EAX	$status_{[7-0]}$
return ESP	ECX		ECX	≡
return EIP	EDX		EDX	≡
–	EDI		EDI	≡
–	ESI	sysenter	ESI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	EDX

Reply/Wait

$hypercall_{[7-0]}$	EAX	**— Reply/Wait ⟶**	EAX	*pt→id*
return ESP	ECX		ECX	≡
–	EDX		EDX	~
–	EDI		EDI	≡
–	ESI	sysenter	ESI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	*pt→ip*

Create Protection Domain

$SEL_{0[31-8]}/hypercall_{[7-0]}$	EAX	— **Create PD** ⟶	EAX	$status_{[7-0]}$
return ESP	ECX		ECX	≡
return EIP	EDX		EDX	≡
SEL_{PD}	EDI		EDI	≡
CRD_{OBJ}	ESI	sysenter	ESI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	EDX

Create Execution Context

$SEL_{0[31-8]}/hypercall_{[7-0]}$	EAX	— **Create EC** ⟶	EAX	$status_{[7-0]}$
return ESP	ECX		ECX	≡
return EIP	EDX		EDX	≡
SEL_{PD}	EDI		EDI	≡
$UTCB_{31-12}\,/\,CPU_{11-0}$	ESI	sysenter	ESI	≡
SP	EBX		EBX	≡
SEL_{EVT}	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	EDX

Create Scheduling Context

$SEL_{0[31-8]}/hypercall_{[7-0]}$	EAX	— **Create SC** ⟶	EAX	$status_{[7-0]}$
return ESP	ECX		ECX	≡
return EIP	EDX		EDX	≡
SEL_{PD}	EDI		EDI	≡
SEL_{EC}	ESI	sysenter	ESI	≡
QPD	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	EDX

Create Portal

$SEL_{0[31-8]}/hypercall_{[7-0]}$	EAX	— **Create PT** ⟶	EAX	$status_{[7-0]}$
return ESP	ECX		ECX	≡
return EIP	EDX		EDX	≡
SEL_{PD}	EDI		EDI	≡
SEL_{EC}	ESI	sysenter	ESI	≡
MTD_{PT}	EBX		EBX	≡
IP	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	EDX

Create Semaphore

$SEL_{0[31-8]}/hypercall_{[7-0]}$	EAX	— **Create SM** ⟶	EAX	$status_{[7-0]}$
return ESP	ECX		ECX	≡
return EIP	EDX		EDX	≡
SEL_{PD}	EDI		EDI	≡
CNT	ESI	sysenter	ESI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	EDX

Revoke Capability Range

$hypercall_{[7-0]}$	EAX	— **Revoke** ⟶	EAX	$status_{[7-0]}$
return ESP	ECX		ECX	≡
return EIP	EDX		EDX	≡
CRD	EDI		EDI	≡
–	ESI	sysenter	ESI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	EDX

Lookup Capability Range

$hypercall_{[7-0]}$	EAX	— **Lookup** ⟶	EAX	$status_{[7-0]}$
return ESP	ECX		ECX	≡
return EIP	EDX		EDX	≡
CRD	EDI		EDI	*CRD*
–	ESI	sysenter	ESI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	EDX

Recall Execution Context

$SEL_{EC[31-8]}/hypercall_{[7-0]}$	EAX	— **Recall** ⟶	EAX	$status_{[7-0]}$
return ESP	ECX		ECX	≡
return EIP	EDX		EDX	≡
–	EDI		EDI	≡
–	ESI	sysenter	ESI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	EDX

Semaphore Control

$SEL_{SM[31-8]}/hypercall_{[7-0]}$	EAX	— **Semctl** ⟶	EAX	$status_{[7-0]}$
return ESP	ECX		ECX	≡
return EIP	EDX		EDX	≡
–	EDI		EDI	≡
–	ESI	sysenter	ESI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	EDX

Assign PCI Device

$SEL_{PD[31-8]}/hypercall_{[7-0]}$	EAX	— **Assign PCI** ⟶	EAX	$status_{[7-0]}$
return ESP	ECX		ECX	≡
return EIP	EDX		EDX	≡
RID_{PF}	EDI		EDI	≡
RID_{VF}	ESI	sysenter	ESI	≡
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	EDX

Assign Global System Interrupt

$SEL_{SM[31-8]}/hypercall_{[7-0]}$	EAX	— **Assign GSI** ⟶	EAX	$status_{[7-0]}$
return ESP	ECX		ECX	≡
return EIP	EDX		EDX	≡
CPU	EDI		EDI	*MSI Addr*
RID	ESI	sysenter	ESI	*MSI Data*
–	EBX		EBX	≡
–	EBP		EBP	≡
–	ESP		ESP	ECX
–	EIP		EIP	EDX