Security Research

PASR

Preparatory Action on the enhancement of the European industrial potential in the field of Security research

Grant Agreement no. SEC5-PR-104600

ROBIN

Open Robust Infrastructures

Project

Deliverable D.12

Machine-Checked Specification

Due date of deliverable: 31/04/2008
Actual submission date: 15/05/2008

Start date of Activity: 31/01/2006 Duration: 31/04/2008

Organisation name of lead beneficiary for this deliverable: Radboud Universiteit Nijmegen

Revision 1

CLASSIFICATION: Public

15/05/2008

Formal Nova interface specification

Formal specification of the micro-hypervisor interface
(Robin deliverable D.12)

Hendrik Tews Tjark Weber Erik Poll
Marko van Eekelen Peter van Rossum

Radboud Universiteit Nijmegen, The Netherlands
http://www.sos.cs.ru.nl/

May 14, 2008

Revision 45

Based on revision 214 of the informal
Nova micro-hypervisor interface specification

This work has been supported by the European Union through PASR grant 104600.

1 Executive Summary

This document contains one major result of work package 4 (kernel specification and verification): a formal specification of the Nova interface.

The specification consists of three main parts: (1) a definition of an abstract internal state for the Nova micro-hypervisor, (2) a description of the operations of the hypervisor in imperative pseudo-code that accesses and modifies the internal state, and (3) a combination of big-step denotational and small-step operational semantics to give semantics to the pseudo code as state modifying functions.

The outstanding point of this work is the design and use of imperative pseudo code in the specification to describe the operations of the kernel. With the pseudo code we found an excellent compromise between the formal world of work package 4 and the programming world in work package 1 (micro-hypervisor and environment) and work package 2 (trusted computing base construction kit and application). The pseudo code is understandable at an intuitive level and can augment the Nova documentation to increase its preciseness. In fact, the description of the hyper calls in the Nova documentation [Steb, 5] already contains pseudo code to make the natural language description precise. The pseudo code enhances the informal specifications, but at the same time, when augmented with a formal definition of the kernel state space, it forms the basis of a specification of the Nova interface.

The pseudo-code programs in this document and in the Nova documentation are almost identical (apart from minor differences in concrete syntax). The remaining small differences are caused by the constant evolution of the original Nova documentation. This document is in sync with revision 214 of the informal Nova micro-hypervisor interface specification from March 14th, 2008. Later changes have not been incorporated.

During the project we were confronted with personnel problems beyond our control. Because of administrative difficulties our second postdoc in work package 4 was not able to timely renew his working permit for the Netherlands and had to quit his contract in the project in the sequel. Because of these personnel problems we did not quite reach our original goal to formalize the Nova interface specification in an appropriate tool, such as PVS or Isabelle/HOL. However, the status reached is very close. It should now be a simple exercise to formalize the specification described in this document in a theorem prover for higher-order logic, or in some other appropriate tool.

Contents

2 Introduction

This document contains a formal specification of the hyper calls of the Nova micro-hypervisor. The mathematical definitions of this document are of course self-contained. However, without knowing the Nova hypervisor in detail it will be difficult to make sense of them. In particular, we are not repeating any material from the Nova documentation [Stea, Steb]. It is assumed that the reader is familiar with these documents.

During our work on the specification it was one major goal to come up with a formal specification that could be useful for those designing and working with Nova. In theory a precise description of the interface of a system is always invaluable for designers and users. In practice however, specifications are often too complicated for designers and users to understand and use them in the time they are willing to spend.

As a step towards designers and users of Nova, we split the specification up into pseudo code describing the hyper calls of Nova (see Chapter 5), a formalization of the internal state space of Nova (see Chapter 3), and the foundation of the pseudo code (see Chapter 4). The pseudo code is imperative in nature, updating local variables or the kernel state. With very few exceptions, the fields of the kernel state are taken literally from the informal Nova specification [Steb]. Therefore the pseudo code is easily understandable for people that know Nova in detail. In fact, each hyper call in the Nova documentation [Steb, 5] is annotated with pseudo code, making the informal description in English language of the behavior much more precise.

The other two parts of the specification give a precise mathematical meaning to the pseudo code; they are much more formal. They are very likely to be ignored by the designers and users of Nova. This ignorance is well-justified, as those sections do not contribute to an intuitive understanding of the pseudo code.

To make the complete specification easier to comprehend, we do not use any fancy specification mechanisms that require additional knowledge. We only use simple, naive set theory, with which the reader is assumed to be familiar. The internal state of the kernel is defined in Chapter 3 as a tuple of functions. The kernel objects that appear inside the states are simple records.

For the pseudo code we do not define a full-blown imperative language. The trick instead is to view the expressions and statements in the pseudo code as syntactic sugar for manipulating kernel states. While this makes it slightly more difficult to decide at first glance whether a given symbol string is valid pseudo code, we are happy to accept this difficulty as our approach greatly simplifies the semantics definition given in Chapter 4.

The pseudo code consists of three kinds of statements: (1) assignments, (2) local control flow statements (such as **if**, but also the special **error** statement), and (3) the global control flow statement **block**. Assignments and local control flow statements can be given a semantics considering only the kernel state and a vector of local variables. To describe the effect of **block**, one must however take all execution contexts and their state into account. We solve the dilemma by defining an operational small step semantics on top of a denotational big step semantics: statements and expressions are described with a denotational big-step semantics with respect to the kernel state. The behavior of the complete system is then defined using an operational small-step semantics. This small-step semantics interprets the local and global control flow statements. It yields a transition system describing all possible future states of the system. Scheduling decisions and user programs (that decide which system calls are performed) are captured using nondeterminism.

The specification describes the behavior of Nova under the assumption of a one-processor system. In our pseudo code the test for non-emptiness (of a list or a queue) and the retrieval of one element is spread over usually two statements. This is obviously incorrect on a multi-processor system, if not protected with locks. The pseudo code in the Nova documentation combines test and retrieval into one utility function. In our version expressions are side-effect free, simplifying the semantics of the pseudo code considerably.

The pseudo-code programs in this document and in the Nova documentation are formulated in slightly different concrete syntaxes. Apart from this representation issue the programs are almost identical. This document and the pseudo code herein is in sync with revision 214 of the informal Nova micro-hypervisor interface specification from March 14th, 2008. The informal interface description has evolved since then, leading to small differences in both documents.

In the future this document should be merged with the informal Nova interface specification to have only one set of pseudo-code programs. The merged document could contain the mathematical foundations of the pseudo code as a technical appendix.

2.1 Notation

We write $\mathbb{N}$ for the set of non-negative integers, i.e. $\mathbb{N} := \{0, 1, 2, \ldots\}$. We write $\mathbb{B}$ for the set of Boolean values, i.e. $\mathbb{B} := \{\text{true}, \text{false}\}$.

We write $f : A \to B$ for a total function mapping elements from A to B. We write $g \colon A \rightharpoonup B$ for a partial function g from A to B. In contrast to a total function, a partial function may leave the image of some $a \in A$ undefined, that is, those a's are not mapped to any element of B. The domain of the partial function g consists of those elements of A that are actually mapped to some element of B.

We omit parentheses in function applications where this does not lead to ambiguities, using $f\ x$ for what is often written $f(x)$.

The notation $f(a \mapsto b)$ is used to denote function update for partial and total functions, i.e. if f is a partial or total function from A to B, $a \in A$ and $b \in B$, then (for $x \in A$)

$$[f(a \mapsto b)]\,(x) := \begin{cases} b & \text{if } x = a; \\ f(x) & \text{otherwise.} \end{cases}$$

A record is a tuple whose elements have been assigned names. These names can be used to access and update elements of the record. If n is the name of the i-th element of the tuple $(x_1, \ldots, x_i, \ldots)$, then $(x_1, \ldots, x_i, \ldots).\mathsf{n} = x_i$. Similar to function update we write $x(\mathsf{n} \mapsto e)$ to denote the tuple that is identical to x, except that the field with name n holds e: $(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots)(\mathsf{n} \mapsto e) = (x_1, \ldots, x_{i-1}, e, x_{i+1}, \ldots)$.

The set of finite lists (i.e. words) over the set A is denoted by $\mathrm{List}[A]$. The empty list is written $[]$, and the constructor function, which prepends one element to a list, is given by cons.

3 Kernel State

In this chapter, we describe the state space of the specification's formal model of the Nova kernel. A single state contains a set of different kernel objects, that are linked (via pointers) with each other. System calls change, create and destroy possibly several kernel objects, producing a new state. It is important here to model the side effect of kernel object modification in the right way. Consider a state s in which the kernel object o_1 has a link to the kernel object o_2, and some system or hyper call that modifies o_2 such that it becomes o_2'. Then in the result state s' the object o_1 has a link *to the modified* o_2'. To obtain this kind of side effect we explicitly model pointer structures: object o_1 will contain a descriptor d, and the state s will map d to o_2. Modifying an object then means to update the descriptor mapping function to let the original descriptor point to the new object: the state s' will map d to o_2'.

The validity of our approach hinges on an important property about object identity and the object descriptor mapping. To discuss this property we have to distinguish the real-world concepts *identity* and *equality.* In the real world identity implies equality but not vice versa: One might have two real-world objects that are indistinguishable apart from the fact that they are located at different places. On the contrary one cannot have two identical real-world objects, because if they are really identical, then they are the same and one has just one object. Complications arise from the fact that in the world of simple set theory identity and equality are identified: In the real world (or inside Nova) one can have two different but equal objects. In simple set theory objects are identical (they are the same) if and only if they are equal.

Because complications arise when switching from the real world (a running Nova instance) to the mathematical universe (a state modelling a Nova instance at a certain point in time) we have to distinguish both worlds. Whenever we say *in Nova* in the following we refer to the real world. *In the model* clearly refers to the mathematical universe.

In our modelling complications arise from side effects: If one has two equal but different objects o_1 and o_2 in Nova and o_1 is changed such that it becomes o_1' then in the model o_2 must not change, although o_1 and o_2 are the same in the model. On the other hand if inside Nova the target of two pointers of two objects o_1 and o_2 is the same object o_3, then, in the model, updates to o_3 must be visible from o_1 and o_2.

These obvious requirements lead to the following property: For two objects that are identical inside Nova there is at most one descriptor (pointer) mapped to this object in the model. The model can contain several descriptors that are mapped to the same object, but then one has correspondingly many different but equal objects inside Nova. In programming terms this property just says that the same object cannot start at

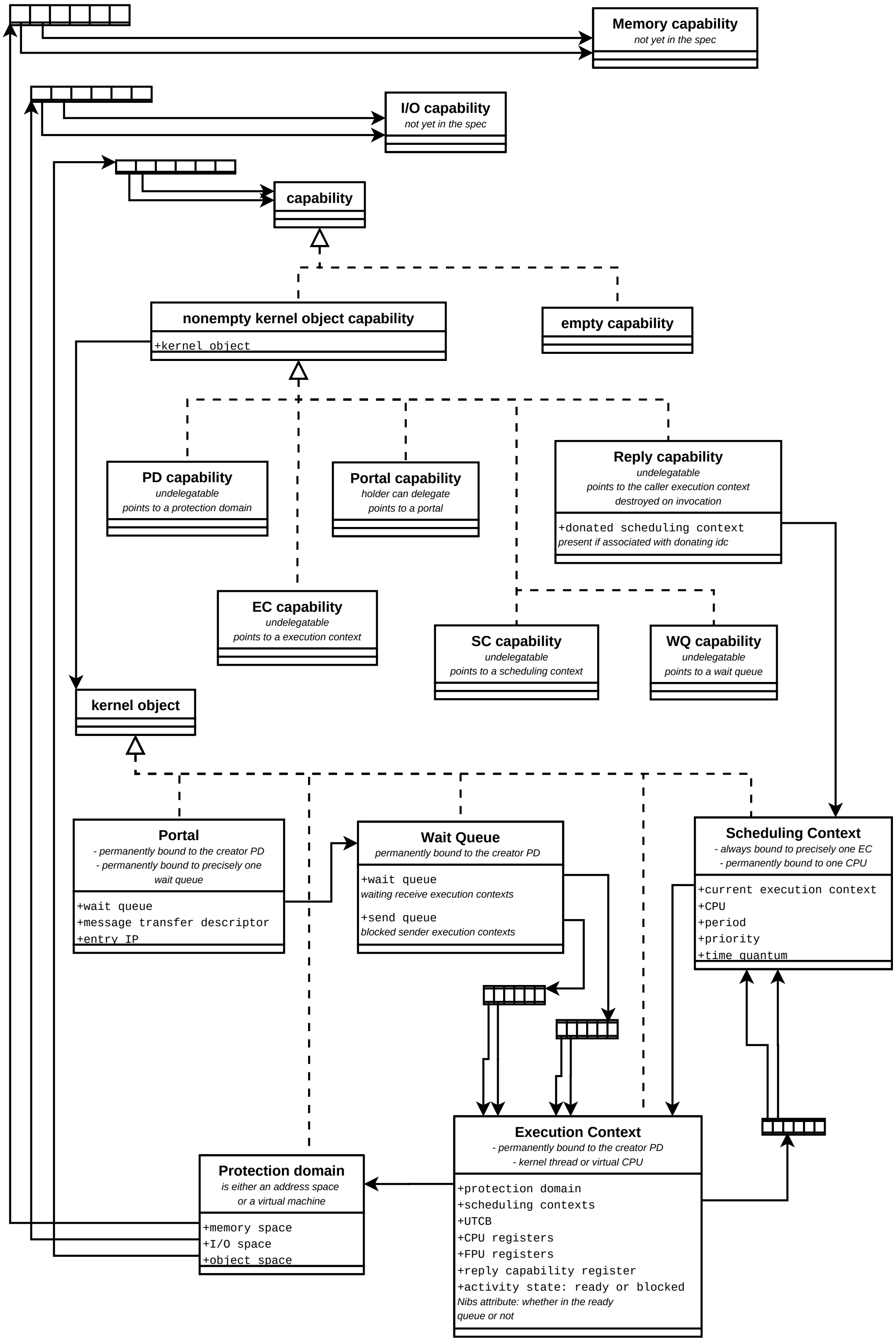
Memory capability
not yet in the spec
I/O capability
not yet in the spec
capability
nonempty kernel object capability
+kernel object
empty capability
PD capability
undelegatable
points to a protection domain
Portal capability
holder can delegate
points to a portal
Reply capability
undelegatable
points to the caller execution context
destroyed on invocation
+donated scheduling context
present if associated with donating idc
EC capability
undelegatable
points to a execution context
SC capability
undelegatable
points to a scheduling context
WQ capability
undelegatable
points to a wait queue
kernel object
Portal
- permanently bound to the creator PD
- permanently bound to precisely one wait queue
+wait queue
+message transfer descriptor
+entry IP
Wait Queue
permanently bound to the creator PD
+wait queue
waiting receive execution contexts
+send queue
blocked sender execution contexts
Scheduling Context
- always bound to precisely one EC
- permanently bound to one CPU
+current execution context
+CPU
+period
+priority
+time quantum
Execution Context
- permanently bound to the creator PD
- kernel thread or virtual CPU
+protection domain
+scheduling contexts
+UTCB
+CPU registers
+FPU registers
+reply capability register
+activity state: ready or blocked
Nibs attribute: whether in the ready queue or not
Protection domain
is either an address space or a virtual machine
+memory space
+I/O space
+object space

Figure 3.1: Kernel objects and their relation

two different addresses in memory. In the model new kernel objects are always created with a kernel-object constructor statement (see Definition 4.1.9 on page 20 below). The constructor statement adds a mapping to the new object to the state that uses new, so far unused descriptor.

The objects in the abstract kernel state and the links between them are depicted in Figure 3.1. An explanation of these kernel objects is given in the Nova documentation [Stea, Steb]. Here we are assuming complete familiarity with the Nova documentation.

Solid arrows in Figure 3.1 represent links or pointers. Any portal will, for instance, always be linked to a wait queue. Dashed arrows symbolize the *is a* relation: A wait queue is a kernel object. The three entities capability, nonempty kernel object capability and kernel object represent some kind of object super type. These super types have only been introduced to keep Figure 3.1 comprehensible. The super types do not play any role in the formal model. The concrete capabilities, such as the PD (protection domain) and WQ (wait queue) capability, inherit the kernel-object pointer field from the abstract kernel object capability. In the concrete capabilities this kernel-object pointer is well-typed: the pointer in a wait queue capability will always point to a wait queue, as indicated in the figure.

The objects in Figure 3.1 and the links are taken almost literally from the informal Nova interface specification [Steb]. Fields attributed with the comment *Nibs attribute*[1] however are our addition. They describe state that is explicitly or implicitly available in the Nova kernel, but not described in the documentation.

Some of the fields visible in Figure 3.1 do not play any role in the informal documentation, apart from being mentioned as attributes of some kernel object. Because those fields do not contribute in any way to the specification we have omitted them from our formal descriptions of kernel object (in Section 3.3) and from the pseudo code (in Chapter 5). It remains future work to refine this specification in a suitable way to incorporate those omitted attributes, once their effect is described informally.

From now on we make a clear distinction between *kernel objects* and *capabilities*. Kernel objects are only those that inherit from *kernel object* in Figure 3.1. Thus, although capabilities are objects in the abstract kernel state, they are *not* referred to as kernel objects in the following.

3.1 Descriptors

Descriptors are the specification's notion of pointers. There are only descriptors to kernel objects. Capabilities are stored in arrays or fields, therefore capabilities are accessed by array index or field name.

Definition 3.1.1 (Descriptors)**.** We define the following sets of descriptors to kernel objects:

[1] A predecessor of this document was called Nova Interface Base Specification, giving rise to the acronym Nibs to which we stick here.

- PDdesc is the set of *protection domain descriptors*;
- ECdesc is the set of *execution context descriptors*;
- SCdesc is the set of *scheduling context descriptors*;
- PTdesc is the set of *portal descriptors*;
- WQdesc is the set of *wait queue descriptors.*

We assume these sets to be infinite and pairwise disjoint. Moreover, we assume that each of these sets contains a distinguished element symbolizing the descriptor that does not refer to any object. This descriptor is referred to as `null-desc`.

Capabilities are treated differently. A capability is referenced (both in the actual kernel and in this specification) by an index into a protection domain's capability table. We model these indices as natural numbers; the only reason for the distinct names is clarity of the specification.

Definition 3.1.2 (Indices to Capabilities)**.** We define the following sets of indices to capabilities:

- $\mathsf{CD}_{\mathsf{PD}} := \mathbb{N}$ is the set of *indices to protection domain capabilities*;
- $\mathsf{CD}_{\mathsf{EC}} := \mathbb{N}$ is the set of *indices to execution context capabilities*;
- $\mathsf{CD}_{\mathsf{SC}} := \mathbb{N}$ is the set of *indices to scheduling context capabilities*;
- $\mathsf{CD}_{\mathsf{PT}} := \mathbb{N}$ is the set of *indices to portal capabilities*;
- $\mathsf{CD}_{\mathsf{WQ}} := \mathbb{N}$ is the set of *indices to wait queue capabilities.*

Reply capabilities are never referred to by index.

3.2 Capabilities

Capabilities are tokens that designate an object together with access rights to that object. They give user programs the right to perform specific actions on this object, such as using it as an argument to a system call. Capabilities are never passed around by reference. If kernel objects are donated, i.e. given to a potentially different protection domain, the kernel will always create a new capability (with possibly diminished access rights).

Definition 3.2.1 (Protection Domain Capability)**.** A *protection domain capability* is a singleton (pd), where

- $\mathsf{pd} \in \mathsf{PDdesc}$.

The set of all protection domain capabilities is denoted by $\mathsf{Cap}_{\mathsf{PD}}$.

Definition 3.2.2 (Execution Context Capability)**.** An *execution context capability* is a singleton (ec), where

- $\mathsf{ec} \in \mathsf{ECdesc}$.

The set of all execution context capabilities is denoted by $\mathsf{Cap}_{\mathsf{EC}}$.

Definition 3.2.3 (Scheduling Context Capability)**.** A *scheduling context capability* is a singleton (sc), where

- $\mathsf{sc} \in \mathsf{SCdesc}$.

The set of all scheduling context capabilities is denoted by $\mathsf{Cap}_{\mathsf{SC}}$.

Definition 3.2.4 (Portal Capability)**.** A *portal capability* is a singleton (pt), where

- $\mathsf{pt} \in \mathsf{PTdesc}$, and

The set of all portal capabilities is denoted by $\mathsf{Cap}_{\mathsf{PT}}$.

Definition 3.2.5 (Wait Queue Capability)**.** A *wait queue capability* is a singleton (wq), where

- $\mathsf{wq} \in \mathsf{WQdesc}$.

The set of all wait queue capabilities is denoted by $\mathsf{Cap}_{\mathsf{WQ}}$.

Definition 3.2.6 (Reply Capability)**.** A *reply capability* is a pair $(\mathsf{ec}, \mathsf{sc})$, where

- $\mathsf{ec} \in \mathsf{ECdesc}$, and
- $\mathsf{sc} \in \mathsf{SCdesc}$.

The set of all reply capabilities is denoted by $\mathsf{Cap}_{\mathsf{Reply}}$.

Definition 3.2.7 (Null Capability)**.** The *null capability* is a special capability (different from all other capabilities) without any data. It is referred to as `null-cap`.

Reply capabilities are only stored in the reply-capability register of execution contexts. All other capabilities are stored in the object-space array of protection domains.

Definition 3.2.8 ($\mathsf{Cap}_{\mathsf{Obj\text{-}space}}$)**.** The set of all object space capabilities, $\mathsf{Cap}_{\mathsf{PD}} \cup \mathsf{Cap}_{\mathsf{EC}} \cup \mathsf{Cap}_{\mathsf{SC}} \cup \mathsf{Cap}_{\mathsf{WQ}} \cup \mathsf{Cap}_{\mathsf{PT}} \cup \{\texttt{null-cap}\}$, is denoted by $\mathsf{Cap}_{\mathsf{Obj\text{-}space}}$.

3.3 Kernel Objects

Aside from descriptors and capabilities, the Nova kernel uses five kinds of kernel objects to store information that is relevant to this formal specification: namely protection domains, execution contexts, scheduling contexts, portals, and wait queues. In the following we define these objects as certain tuples. In these definitions we will state *default values* for some of the tuple elements. These default values are important for the semantics of the constructor statement for kernel objects (see Definition 4.1.9 on page 20). The constructor statement takes arguments for precisely those tuple elements that have no default initialization. The constructor combines its arguments with the default values in the obvious way to determine the new kernel object.

Certain attributes (such as the priority in a scheduling context) are mentioned in the informal Nova specification, but without any description of their effect (at least not in revision 214, on which this specification is based). Consequently these attributes are omitted in the following definitions.

Definition 3.3.1 (Protection Domain)**.** A *protection domain* is a singleton $(\mathsf{object\text{-}space})$, where

- $\mathsf{object\text{-}space}\colon \mathbb{N} \to \mathsf{Cap}_{\mathsf{Obj\text{-}space}}$ is an array mapping indices to capabilities. (Its default value is the function that maps all indices to the null capability, `null-cap`.)

The set of all protection domains is denoted by PD.

In any reachable kernel state the set of indices to non-null capabilities, $\{i \in \mathbb{N} \mid \mathsf{object\text{-}space}\ i \neq \texttt{null-cap}\}$, will be finite. In the Nova implementation, a protection domain also contains a memory space and an I/O space. These attributes are currently not present in the formal specification.

Definition 3.3.2 (Execution Context)**.** An *execution context* is a four-tuple $(\mathsf{pd}, \mathsf{sc}, \mathsf{reply\text{-}cap}, \mathsf{state})$, where

- $\mathsf{pd} \in \mathsf{PDdesc}$ (with no default value),
- $\mathsf{sc} \subseteq \mathsf{SCdesc}$ (defaulting to the empty set),
- $\mathsf{reply\text{-}cap} \in \mathsf{Cap}_{\mathsf{Reply}}$ (defaulting to the null capability), and
- $\mathsf{state} \in \{\mathrm{ready}, \mathrm{blocked}\}$ (defaulting to ready).

The set of all execution contexts is denoted by EC.

In the Nova implementation, an execution context also contains register values and a user thread control block (UTCB). These attributes are currently not present in the formal specification. In all reachable kernel states sc will contain only finitely many descriptors.

Definition 3.3.3 (Scheduling Context)**.** A *scheduling context* is a singleton (ec), where

- $\mathsf{ec} \in \mathsf{ECdesc}$ (with no default value).

The set of all scheduling contexts is denoted by SC.

In the Nova implementation, a scheduling context also contains a CPU identifier, a time quantum, a period, and a priority. These attributes are currently not present in the formal specification.

Definition 3.3.4 (Portal)**.** A *portal* is a singleton (wq), where

- $\mathsf{wq} \in \mathsf{WQdesc}$ (with no default value).

The set of all portals is denoted by PT.

In the Nova implementation, a portal also contains a message transfer descriptor and an entry instruction pointer. These attributes are currently not present in the formal specification.

Definition 3.3.5 (Wait Queue)**.** A *wait queue* is a pair $(\mathsf{wait\text{-}queue}, \mathsf{send\text{-}queue})$, where

- $\mathsf{wait\text{-}queue} \in \mathrm{List}[\mathsf{EC}]$ (defaulting to the empty list), and
- $\mathsf{send\text{-}queue} \in \mathrm{List}[\mathsf{EC}]$ (defaulting to the empty list).

The set of all wait queues is denoted by WQ.

The wait-queue field contains execution contexts waiting to receive (from a call/send using a portal that points to this wait queue). The send-queue field contains execution contexts currently doing a call or send (using a portal that points to this wait queue) that have to wait until a possible receiver execution context shows up. Naturally, there are no reachable kernel states where both queues are non-empty.

3.4 Kernel State

The state of the kernel is now given by five partial functions that map descriptors to kernel objects, and by one field that holds the current scheduling context.

Definition 3.4.1 (Kernel State)**.** A *kernel state* is a 6-tuple $(\mathsf{Pd}, \mathsf{Ec}, \mathsf{Sc}, \mathsf{Pt}, \mathsf{Wq}, \mathsf{current\text{-}sc})$, where

- $\mathsf{Pd}\colon \mathsf{PDdesc} \rightharpoonup \mathsf{PD}$,
- $\mathsf{Ec}\colon \mathsf{ECdesc} \rightharpoonup \mathsf{EC}$,
- $\mathsf{Sc}\colon \mathsf{SCdesc} \rightharpoonup \mathsf{SC}$,
- $\mathsf{Pt}\colon \mathsf{PTdesc} \rightharpoonup \mathsf{PT}$,
- $\mathsf{Wq}\colon \mathsf{WQdesc} \rightharpoonup \mathsf{WQ}$, and

- current-sc $\in$ SCdesc.

The set of all kernel states is denoted by State.

Not every 6-tuple in State corresponds to an actual Nova state. The reachable kernel states fulfill the following properties:

- The domains of the five partial functions are finite.
- The null descriptor `null-desc` is not in the domain of any of the five partial functions.
- Sc current-sc is undefined only when the operational semantics performs meta steps, see Section 4.2, and current-sc = `null-desc` in this case. Otherwise Sc current-sc is defined.
- Suppose d is a descriptor that is mapped (by the corresponding partial function) to a kernel object o. Then all non-null descriptors in o are in the domain of their corresponding partial function (e.g. a wait queue descriptor in o is mapped to some wait queue by Wq, etc.).

Let σ be a kernel state. The *live kernel objects* of σ are the elements in the range of either of the five partial functions in σ. The *live descriptors* in σ are the elements in the domain of either of the five partial functions in σ.

4 Pseudo-Code Semantics

In this chapter, we give a formal semantics for the pseudo code that we use to specify the behavior of system calls in the following chapter. The semantics is split into a denotational (big step) semantics for expressions and single statements, and an operational (small step) semantics for the local and global control flow. Most of this is completely standard without any surprises.

4.1 Denotational Semantics for Expressions and Simple Statements

We give a denotational semantics of simple pseudo-code statements as a partial function that maps pairs (Γ, σ) to successor pairs (Γ', σ'). Here Γ is an environment holding the values of local variables, and σ is a kernel state according to Definition 3.4.1. Formally an environment is a partial function from variable symbols to Value, the set of all values, where

$$\mathsf{Value} \;:=\; \mathsf{Cap}_{\mathsf{Reply}} \cup \mathsf{Cap}_{\mathsf{Obj\text{-}space}} \cup \mathsf{PD} \cup \mathsf{EC} \cup \mathsf{SC} \cup \mathsf{PT} \cup \mathsf{WQ}.$$

The set of all environments is denoted by Env.

We do not specify a precise grammar for our pseudo code. Such a grammar is implicit from the semantics definition below, and from the pseudo code fragments used in the following chapter.

4.1.1 Expressions

Expressions are side-effect free. They do not modify the current state, but they denote a value that will usually depend on the state. Expressions include (local and global) variable names, arguments of system calls, record field access, and literals.

Local Variables

The semantics of a local variable (or a system call argument) x is given by its value in the environment.

Definition 4.1.1 (Semantics of Local Variables and Arguments)**.** Let x be the name of a local variable, or the name of a system call argument. Then $[\![x]\!]_{(\Gamma,\sigma)} := \Gamma(x)$.

Local variables must be assigned a value before their first use. A pseudo-code fragment accessing a local variable that has not been assigned a value is ill-formed.

Field Access

If x is an expression that denotes a tuple with a named field a (where the field names are those used in the definitions in Chapter 3), we write $x.\mathsf{a}$ to select this field in x. Field access associates to the left: $x.\mathsf{a}.\mathsf{b}$ is short for $(x.\mathsf{a}).\mathsf{b}$.

Definition 4.1.2 (Semantics of Field Access)**.** Let x be an expression with $[\![x]\!]_{(\Gamma,\sigma)} = (x_{\mathsf{a}}, x_{\mathsf{b}}, \ldots)$. Then $[\![x.\mathsf{a}]\!]_{(\Gamma,\sigma)} := x_{\mathsf{a}}$ (and likewise for $x.\mathsf{b}$, ...).

Global Variables

The pseudo code can access the five partial functions and the current-sc field that are part of the kernel state (see Definition 3.4.1) via global variables.

Definition 4.1.3 (Semantics of Global Variables)**.** Let $\mathsf{x} \in \{\mathsf{Pd}, \mathsf{Ec}, \mathsf{Sc}, \mathsf{Pt}, \mathsf{Wq}, \mathsf{current\text{-}sc}\}$. Then $[\![\mathsf{x}]\!]_{(\Gamma,\sigma)} := \sigma.\mathsf{x}$.

Capability Creation Functions

For each capability type there is a constructor function that takes precisely as many arguments as there are fields in the capability. The pseudo code denotes these constructor functions in C++ style in the form `new x-cap(...)`, where `x` names one of the capability types `PD`, `EC`, `SC`, `Portal`, `WQ` and `Reply`. Note that the very similar looking `new PD(...)` is instead a kernel-object constructor statement, whose semantics is given in Definition 4.1.9 on page 20.

Definition 4.1.4 (Semantics of Capability Creation)**.**

$$\begin{aligned}
[\![\texttt{new PD-cap(x)}]\!]_{(\Gamma,\sigma)} &:= ([\![\texttt{x}]\!]_{(\Gamma,\sigma)}) \\
[\![\texttt{new EC-cap(x)}]\!]_{(\Gamma,\sigma)} &:= ([\![\texttt{x}]\!]_{(\Gamma,\sigma)}) \\
[\![\texttt{new SC-cap(x)}]\!]_{(\Gamma,\sigma)} &:= ([\![\texttt{x}]\!]_{(\Gamma,\sigma)}) \\
[\![\texttt{new Portal-cap(x)}]\!]_{(\Gamma,\sigma)} &:= ([\![\texttt{x}]\!]_{(\Gamma,\sigma)}) \\
[\![\texttt{new WQ-cap(x)}]\!]_{(\Gamma,\sigma)} &:= ([\![\texttt{x}]\!]_{(\Gamma,\sigma)}) \\
[\![\texttt{new Reply-cap(x,y)}]\!]_{(\Gamma,\sigma)} &:= ([\![\texttt{x}]\!]_{(\Gamma,\sigma)}, [\![\texttt{y}]\!]_{(\Gamma,\sigma)})
\end{aligned}$$

Predefined Functions

There are predefined Boolean functions to test a list for emptiness, and to test the type of a capability.

Definition 4.1.5 (Semantics of Predefined Functions).

$$\llbracket \texttt{nonempty? l} \rrbracket_{(\Gamma,\sigma)} := \begin{cases} \text{true} & \text{if } \llbracket \texttt{l} \rrbracket_{(\Gamma,\sigma)} = []; \\ \text{false} & \text{otherwise.} \end{cases}$$

$$\llbracket \texttt{nonempty-cap? c} \rrbracket_{(\Gamma,\sigma)} := \begin{cases} \text{false} & \text{if } \llbracket \texttt{c} \rrbracket_{(\Gamma,\sigma)} = \texttt{null-cap}; \\ \text{true} & \text{otherwise.} \end{cases}$$

$$\llbracket \texttt{is-PD-cap? c} \rrbracket_{(\Gamma,\sigma)} := \begin{cases} \text{true} & \text{if } \llbracket \texttt{c} \rrbracket_{(\Gamma,\sigma)} \in \mathsf{Cap}_{\mathsf{PD}}; \\ \text{false} & \text{otherwise.} \end{cases}$$

$$\llbracket \texttt{is-EC-cap? c} \rrbracket_{(\Gamma,\sigma)} := \begin{cases} \text{true} & \text{if } \llbracket \texttt{c} \rrbracket_{(\Gamma,\sigma)} \in \mathsf{Cap}_{\mathsf{EC}}; \\ \text{false} & \text{otherwise.} \end{cases}$$

$$\llbracket \texttt{is-SC-cap? c} \rrbracket_{(\Gamma,\sigma)} := \begin{cases} \text{true} & \text{if } \llbracket \texttt{c} \rrbracket_{(\Gamma,\sigma)} \in \mathsf{Cap}_{\mathsf{SC}}; \\ \text{false} & \text{otherwise.} \end{cases}$$

$$\llbracket \texttt{is-Portal-cap? c} \rrbracket_{(\Gamma,\sigma)} := \begin{cases} \text{true} & \text{if } \llbracket \texttt{c} \rrbracket_{(\Gamma,\sigma)} \in \mathsf{Cap}_{\mathsf{Portal}}; \\ \text{false} & \text{otherwise.} \end{cases}$$

$$\llbracket \texttt{is-WQ-cap? c} \rrbracket_{(\Gamma,\sigma)} := \begin{cases} \text{true} & \text{if } \llbracket \texttt{c} \rrbracket_{(\Gamma,\sigma)} \in \mathsf{Cap}_{\mathsf{WQ}}; \\ \text{false} & \text{otherwise.} \end{cases}$$

$$\llbracket \texttt{is-Reply-cap? c} \rrbracket_{(\Gamma,\sigma)} := \begin{cases} \text{true} & \text{if } \llbracket \texttt{c} \rrbracket_{(\Gamma,\sigma)} \in \mathsf{Cap}_{\mathsf{Reply}}; \\ \text{false} & \text{otherwise.} \end{cases}$$

Literals and Other Expressions

Literals (e.g. `0`, `1`, . . . , `true`, `false`, etc.) and other pseudo-code expressions (e.g. `x = y`, `x` **and** `y`, **not** `x`, etc.) have their obvious semantics.

4.1.2 Simple Statements

Statements usually modify the current state in some way: assignments to local variables will update the environment Γ, while other statements will change the actual kernel state σ by modifying existing kernel objects or creating new ones. Here, we only treat assignments and kernel-object constructor statements. For composite and control-flow statements see Section 4.2.1 (on page 22 below).

Assignment

There are three different forms of assignment in the pseudo code:

- local variable assignments,
- field updates of some kernel object, and
- overwriting a capability in the object space of some protection domain.

We now define their semantics. Additionally, there are statements of the form `desc-var := new ...`. Although these look like an assignment, they really are kernel-object constructor statements (see Definition 4.1.9).

Definition 4.1.6 (Semantics of Assignment to Local Variables/Arguments)**.** Let x be the name of a local variable, or the name of a system call argument, and let e be an expression. Then

$$[\![x{:=}\,e]\!]_{(\Gamma,\sigma)} \;:=\; \big(\Gamma(x \mapsto [\![e]\!]_{(\Gamma,\sigma)}), \sigma\big).$$

Definition 4.1.7 (Semantics of Field Update of Kernel Objects)**.** Let d be an expression that denotes a descriptor to a kernel object, let f be a field of that kernel object, and let e be an expression that denotes a possible value of field f. In the current state σ the descriptor d denotes a kernel object, which can be accessed as $\sigma.\xi\ [\![d]\!]_{(\Gamma,\sigma)}$, where ξ is the name of the state field corresponding to the type of d (i.e. $\xi = \mathsf{Pd}$ for $[\![d]\!]_{(\Gamma,\sigma)} \in \mathsf{PDdesc}$, $\xi = \mathsf{Sc}$ for $[\![d]\!]_{(\Gamma,\sigma)} \in \mathsf{SCdesc}$, etc.). Let x_σ denote the updated kernel object in state σ:

$$x_\sigma \;:=\; \big(\sigma.\xi\ [\![d]\!]_{(\Gamma,\sigma)}\big)\big(\mathsf{f} \mapsto [\![e]\!]_{(\Gamma,\sigma)}\big)$$

Then the semantics of field update is defined as

$$[\![d.\mathsf{f} := e]\!]_{(\Gamma,\sigma)} \;:=\; \Big(\Gamma,\ \sigma\big(\xi \mapsto (\sigma.\xi([\![d]\!]_{(\Gamma,\sigma)} \mapsto x_\sigma))\big)\Big).$$

If $\sigma.\xi\ d$ is not defined, then the semantics of the entire assignment in state σ is undefined.

Definition 4.1.8 (Semantics of Overwriting Capabilities)**.** Let d be an expression that denotes a protection domain descriptor in PDdesc, let i be an expression that denotes an index in $\mathbb{N}$, and let c be an expression that denotes a capability. Let o_σ denote the updated object space in state σ:

$$o_\sigma \;:=\; \big((\sigma.\mathsf{Pd}\ [\![d]\!]_{(\Gamma,\sigma)}).\mathsf{object\text{-}space}\big)\big([\![i]\!]_{(\Gamma,\sigma)} \mapsto [\![c]\!]_{(\Gamma,\sigma)}\big).$$

Because protection domains are singletons, the singleton (o_σ) is the updated protection domain. Then the semantics of overwriting capabilities is defined as

$$[\![d.\mathsf{object\text{-}space}[i] := c]\!]_{(\Gamma,\sigma)} \;:=\; \big(\Gamma, \sigma\big(\mathsf{Pd} \mapsto \sigma.\mathsf{Pd}([\![d]\!]_{(\Gamma,\sigma)} \mapsto (o_\sigma))\big)\big).$$

Kernel-Object Constructor Statement

A statement of the form `l := new-X(...)`, where `l` is a local variable and `X` names one of the kernel objects, is a kernel-object constructor statement. It first determines a fresh, so far unused descriptor d, and then changes the state to let d be mapped to a kernel object whose fields have values according to the arguments in the constructor statement (except for fields with default value). The restriction to local variables on the left-hand side is not really essential, it is just that in the current pseudo code, newly created kernel objects are always assigned to local variables.

Definition 4.1.9 (Semantics of Constructor Statement)**.** Let l be a local variable, let `X` be one of `PD`, `EC`, `SC`, `Portal` or `WQ`, and let ξ be the name of the state field corresponding to `X`. Let n be the number of fields of the kernel object denoted by `X` without default value, and let f_1, …, f_n be their respective field names. (If, for instance, `X` is `PD`, then $n = 0$ and the list of f_i is empty. If `X` is `EC`, then $n = 1$ and $\mathsf{f}_1 = \mathsf{pd}$.) Assume further n expressions e_1, …, e_n denoting possible values for the fields f_1, …, f_n.

Let o be the kernel object of the type denoted by `X` whose fields f_1, …, f_n have values $[\![e_1]\!]_{(\Gamma,\sigma)}$, …, $[\![e_n]\!]_{(\Gamma,\sigma)}$ respectively, and the other fields have their default value (as given by the definitions in Chapter 3). For a state σ, let d_σ be a descriptor that is not live in σ. Then the semantics of the constructor statement is as follows:

$$[\![l := \texttt{new-X}(.\mathsf{f}_1 := e_1, \ldots, .\mathsf{f}_n := e_n)]\!]_{(\Gamma,\sigma)} \;:=\; \big(\Gamma(l \mapsto d_\sigma),\ \sigma(\xi \mapsto \sigma.\xi(d_\sigma \mapsto o))\big).$$

4.2 Operational Semantics for Local and Global Control Flow

Up to now we described a simple imperative programming language. The only thing not completely standard was the state space over which this programming language is interpreted. The **if** and **while** statements are completely standard as well. However, before we can describe them we have to motivate the definition of system state.

What makes our pseudo code a bit special are the **error** and the **block** statement. The **error** statement captures abnormal exits from hyper calls for instance because of unsuitable argument values. We thereby abstract from the different error codes that the Nova hypervisor gives back in such cases. A pseudo-code program executing **error** is instantly terminated, skipping all remaining pseudo-code statements.

The behavior of the **block** statement can only be described with respect to the complete system state, consisting of a kernel state and a set of active execution contexts of which each runs a pseudo code program. The actions of these pseudo programs are interleaved. At any given point in time only one execution context makes progress. This one progressing execution context is called the *current execution context.* The other active execution contexts are suspended. Rescheduling (that is suspending the current context and letting a different one execute) happens at **block** statements and at the end of the pseudo code program.

In our specification execution contexts that are equal but physically different are identified (compare the discussion about identity and equality on page 8). Therefore, we have to speak about the *current execution-context descriptor.* It is possible to have a system state with two descriptors d_1 and d_2 that are both mapped to the same execution context. It corresponds to a Nova state with two execution contexts that agree on all the fields listed in Definition 3.3.2. Only one of those two can be the current one.

In our specification the currently live execution contexts (i.e. those that exist in a given state) are divided into the *active* and *inactive* ones. Active execution contexts are currently executing a hyper call and are associated with a pseudo code program (that they are currently executing) and a context Γ holding local variables and hyper call arguments. Inactive execution contexts are executing user code. They have no associated pseudo code program. At any point in time an inactive execution context can become active by (nondeterministically) choosing a hyper call with suitable arguments. An active execution context becomes inactive when it finished it current pseudo-code program.

For the same reason as before the distinction between active and inactive execution contexts must actually be made on the level of execution-context descriptors.

The **block** statement can be nested inside **if** or **while** statements. Therefore these statements get an operational semantics, in which the current execution context makes progress in its pseudo-code program, thereby changing the kernel state. If the current execution context hits a **block** statement it is suspended and stored back to all the other active execution contexts. Thereby one must remember its local variable context Γ and the statement following the **block** statement, such that this execution context can continue to process its pseudo-code program once it is again selected as current. After suspending the current execution context there is a nondeterministic choice: (1) a currently inactive execution context can be chosen (nondeterministically) to become active, (2) one of the active execution contexts can be chosen (nondeterministically) to become current. This execution context will then continue its pseudo code program where it was suspended before or start it (if it was just made active) until the pseudo-code program terminates or a **block** statement is executed. Choosing a new current execution context happens by nondeterministically selecting a live scheduling context that points to an execution context whose state is ready.

In the operational semantics we distinguish *pseudo-code steps* and *meta steps*. Pseudo-code steps make progress in the pseudo-code program of the current execution context. They are enabled if $\sigma.\mathsf{current\text{-}sc} \neq \texttt{null-desc}$ for the current kernel state σ. Meta steps are the activation of an execution context or the selection of a new current execution context. Meta steps are enabled if $\sigma.\mathsf{current\text{-}sc} = \texttt{null-desc}$. Executing a **block** statement sets $\sigma.\mathsf{current\text{-}sc}$ to `null-desc`, and selecting the next current execution context sets $\sigma.\mathsf{current\text{-}sc}$ to a value different from `null-desc`.

A pseudo-code program is a finite list of statements $[s_1, \dots, s_n]$, where each of the s_i can be a simple or complex statement. Occasionally we write just one s_i of a sequence of statements, for instance in the branches of **if** statements. In pseudo-code programs that are currently executing or suspended, we mark the statement to be executed next with an arrow: $[s_1, \dots, \rightarrow s_i, \dots, s_n]$.statement to be executed can be nested:

$$[\dots, \ \textbf{if}\ e\ \textbf{then}\ s_{i,1}, \dots, \rightarrow s_{i,j}, \dots s_{i,n_i}\ \textbf{else}\ \dots\ \textbf{endif}, \dots]$$

A pseudo-code program that has been processed completely is depicted as $[s_1, \dots, s_n, \rightarrow]$. Similarly we use $[s_1, \dots, \ \textbf{if}\ e\ \textbf{then}\ \dots \rightarrow\ \textbf{else}\ \dots\ \textbf{endif}, \dots]$ to indicate that the execution has reached the end of the then-block.

The set of all pseudo-code programs with next-statement marks is denoted by $\mathsf{Program}$.

Definition 4.2.1 (System State)**.** The system state is a pair $(\sigma, \mathsf{active\text{-}ec})$, where

- σ is a kernel state (according to Definition 3.4.1), and

- $\mathsf{active\text{-}ec}\colon \mathsf{ECdesc} \rightharpoonup \mathsf{Env} \times \mathsf{Program}$ is a partial function whose domain contains precisely the active execution-context descriptors, mapping them to their current local variable environment and pseudo-code program.

In the reachable system states the following properties hold:

- The active execution-context descriptors (elements in the domain of $\mathsf{active\text{-}ec}$) are contained in the live execution context descriptors in σ.

- All descriptors in all environments Γ in the range of $\mathsf{active\text{-}ec}$ are live.

- If $\sigma.\mathsf{current\text{-}sc} \neq \mathtt{null\text{-}desc}$, then the execution-context descriptor reached from $\sigma.\mathsf{current\text{-}sc}$ is the current execution-context descriptor:

  $$\mathsf{current\text{-}ec} \;:=\; (\sigma.\mathsf{Sc}\ \sigma.\mathsf{current\text{-}sc}).\mathsf{current\text{-}ec}$$

  The current execution-context descriptor shall always be in the domain of $\mathsf{active\text{-}ec}$.

- All the live but inactive execution contexts are in state ready.

A system state with $\sigma.\mathsf{current\text{-}sc} \neq \mathtt{null\text{-}desc}$ (where a current execution-context descriptor can be determined as just described) is said to *contain the current execution-context descriptor* $\mathsf{current\text{-}ec}$.

In the following two subsections we define the possible transitions $s \longrightarrow s'$ for system states s and s'. This gives a transition system describing the behavior of the Nova micro-hypervisor. In this transition system states with precisely one successor state describe deterministic behavior, where the kernel is processing a hyper call and the next action is completely determined by the current kernel state and the current pseudo-code program. There are also states with many successor states. They correspond to scheduling decisions (where there is one successor state for each possible schedule) and to hyper calls initiated by user code (there is one successor state for each possible hyper call with a well-typed argument list for each live but inactive execution-context descriptor).

4.2.1 Pseudo-Code Transitions

Sequential composition and the **if** and **while** statements are completely standard. Note that currently we don't have diverging while loops in the traditional sense, because the only while loop (in the pseudo code for call/send on page 31) contains a **block** statement.

Definition 4.2.2 (Semantics of Sequential Composition)**.** Let s be a system state with current execution-context descriptor d such that $s.\mathsf{active\text{-}ec}\ d = (\Gamma, p)$. Then, depending on p, the following transition is possible.

- $p = [s_1, \ldots, \rightarrow s_i, s_{i+1}, \ldots, s_n]$:
  Let $(\Gamma', \sigma') = [\![s_i]\!]_{(\Gamma, s.\sigma)}$, then there is a transition to the state

  $$\big(\sigma',\ s.\mathsf{active\text{-}ec}\big(d \mapsto (\Gamma',\ [s_1, \ldots, s_i, \rightarrow s_{i+1}, \ldots, s_n])\big)\big).$$

Definition 4.2.3 (Semantics of **if**)**.** Let s be a system state with current execution-context descriptor d such that $s.\mathsf{active\text{-}ec}\ d = (\Gamma, p)$. The following transitions for **if** statements are possible, depending on p.

- $p = [s_1, \ldots, \rightarrow\ \textbf{if}\ e\ \textbf{then}\ s_{i,1}\ \textbf{else}\ s_{i,2}\ \textbf{endif}, \ldots, s_n]$:

  In case $[\![e]\!]_{(\Gamma, s.\sigma)}$ equals true, there is a transition to the state

  $$\big(s.\sigma,\ s.\mathsf{active\text{-}ec}\big(d \mapsto (\Gamma,\ [s_1, \ldots,\ \textbf{if}\ e\ \textbf{then}\ \rightarrow s_{i,1}\ \textbf{else}\ s_{i,2}\ \textbf{endif}, \ldots, s_n])\big)\big).$$

  Otherwise there is a transition to the state

  $$\big(s.\sigma,\ s.\mathsf{active\text{-}ec}\big(d \mapsto (\Gamma,\ [s_1, \ldots,\ \textbf{if}\ e\ \textbf{then}\ s_{i,1}\ \textbf{else}\ \rightarrow s_{i,2}\ \textbf{endif}, \ldots, s_n])\big)\big).$$

- $p = [s_1, \ldots,\ \textbf{if}\ e\ \textbf{then}\ s_{i,1} \rightarrow\ \textbf{else}\ s_{i,2}\ \textbf{endif},\ s_{i+1}, \ldots, s_n]$ or
  $p = [s_1, \ldots,\ \textbf{if}\ e\ \textbf{then}\ s_{i,1}\ \textbf{else}\ s_{i,2} \rightarrow\ \textbf{endif},\ s_{i+1}, \ldots, s_n]$:

  There is a transition to the state

  $$\big(s.\sigma,\ s.\mathsf{active\text{-}ec}\big(d \mapsto (\Gamma,\ [s_1, \ldots,\ \textbf{if}\ e\ \textbf{then}\ s_{i,1}\ \textbf{else}\ s_{i,2}\ \textbf{endif},\ \rightarrow s_{i+1}, \ldots, s_n])\big)\big).$$

Definition 4.2.4 (Semantics of **while**)**.** Let s be a system state with current execution-context descriptor d such that $s.\mathsf{active\text{-}ec}\ d = (\Gamma, p)$. The following transitions for **while** statements are possible, depending on p.

- $p = [s_1, \ldots, \rightarrow\ \textbf{while}\ e\ \textbf{do}\ s_i\ \textbf{done},\ s_{i+1}, \ldots, s_n]$:

  In case $[\![e]\!]_{(\Gamma, s.\sigma)}$ is true, there is a transition to the state

  $$\big(s.\sigma,\ s.\mathsf{active\text{-}ec}\big(d \mapsto (\Gamma,\ [s_1, \ldots,\ \textbf{while}\ e\ \textbf{do}\ \rightarrow s_i\ \textbf{done},\ s_{i+1}, \ldots, s_n])\big)\big).$$

  Otherwise there is a transition to the state

  $$\big(s.\sigma,\ s.\mathsf{active\text{-}ec}\big(d \mapsto (\Gamma,\ [s_1, \ldots,\ \textbf{while}\ e\ \textbf{do}\ s_i\ \textbf{done},\ \rightarrow s_{i+1}, \ldots, s_n])\big)\big).$$

- $p = [s_1, \ldots,\ \textbf{while}\ e\ \textbf{do}\ s_i \rightarrow\ \textbf{done},\ s_{i+1}, \ldots, s_n]$:

  There is a transition to the state

  $$\big(s.\sigma,\ s.\mathsf{active\text{-}ec}\big(d \mapsto (\Gamma,\ [s_1, \ldots, \rightarrow\ \textbf{while}\ e\ \textbf{do}\ s_i\ \textbf{done},\ s_{i+1}, \ldots, s_n])\big)\big).$$

The **error** statement captures all error reporting functionality. Executing **error** terminates the current pseudo-code program immediately.

Definition 4.2.5 (Semantics of **error**)**.** Let s be a system state with current execution-context descriptor d such that $s.\mathsf{active\text{-}ec}\ d = (\Gamma, p)$. Then, depending on p, there is the following transition for the **error** statement.

- $p = [s_1, \ldots, \rightarrow\ \textbf{error}, \ldots, s_n]$:

  There is a transition to the state

$$\big(s.\sigma,\ s.\text{active}\big(d \mapsto (\Gamma,\ [s_1, \ldots,\ \textbf{error}, \ldots, s_n, \rightarrow])\big)\big).$$

The **block** statement suspends the current execution context. Then a new scheduling context is chosen to determine the new current execution-context descriptor. In between an arbitrary number of execution-context descriptors can be activated. We split the description of the semantics of **block** into the following definition (which only clears the current scheduling context) and Definition 4.2.7, which describes the meta steps.

Definition 4.2.6 (Semantics of **block**)**.** Let s be a system state with current execution-context descriptor d such that $s.\mathsf{active\text{-}ec}\ d = (\Gamma, p)$. Then, depending on p, there is the following transition for the **block** statement.

- $p = [s_1, \ldots, \rightarrow\ \textbf{block},\ s_i, \ldots, s_n]$:

  There is a transition to the state

$$\big(s.\sigma(\mathsf{current\text{-}sc} \mapsto \texttt{null-desc}),\ \ s.\mathsf{active\text{-}ec}\big(d \mapsto (\Gamma,\ [s_1, \ldots,\ \textbf{block},\ \rightarrow s_i, \ldots, s_n])\big)\big).$$

4.2.2 Meta Steps in the Operational Semantics

Meta steps are performed for system states without a current execution context (where $\mathsf{current\text{-}sc}$ contains the null descriptor). The meta steps give rise to a huge nondeterministic choice that reflects the freedom of the scheduler and the user mode programs (which are both not contained in the specification).

Definition 4.2.7 (Meta Steps)**.** Let s be a system state without current execution-context descriptor (i.e. $s.\mathsf{current\text{-}sc} = \texttt{null-desc}$). Then the following transitions are possible.

- Execution context activation:

  For all execution-context descriptors d, pseudo-code programs $p = [s_1, \ldots, s_n]$, and local variable contexts Γ, there is a transition if the following conditions are met.

  - The descriptor d is a live but inactive execution-context descriptor in s, and the list of scheduling contexts of $s.\sigma.\mathsf{Ec}\ d$ is not empty.
  - The program p is a complete pseudo-code program for one hyper call (as given in Chapter 5).
  - The environment Γ maps just the arguments of that hyper call to valid values according to their type. Local variables are undefined in the initial Γ.

  If these conditions are fulfilled there is a transition to the state

$$\big(s.\sigma,\ \ s.\mathsf{active\text{-}ec}(d \mapsto (\Gamma, [\rightarrow s_1, \ldots, s_n]))\big).$$

- Execution-context selection.:

  There is a transition for each live scheduling-context descriptor $\mathsf{sc\text{-}desc}$ that fulfills the following condition.

  - The execution-context descriptor $(s.\sigma.\mathsf{Sc}\ \mathsf{sc\text{-}desc}).\mathsf{ec}$ is active, and it is mapped to an execution context in state ready.

  The transition goes to the state

  $$\big(s.\sigma(\mathsf{current\text{-}sc} \mapsto \mathsf{sc\text{-}desc}),\ s.\mathsf{active\text{-}ec}\big).$$

4.2.3 Initial System State

The initial state is specified following the description in [Steb, Section 6] and following the behavior of the hyper call to create new protection domains. The initial state contains one protection domain, one execution context ec, and one scheduling context sc. In the object space of the protection domain there are precisely two non-null capabilities: at the index `predefined-capabilities` (coming from the hypervisor information page, see [Steb, Section 6.2]) there is an execution-context capability pointing to ec, and at the index `predefined-capabilities` + 1 there is a scheduling-context capability pointing to sc. The current scheduling context is sc, which determines the current execution context as ec.

Definition 4.2.8 (Initial System State)**.** We give a constructive definition of the initial system state.

- Let $\mathsf{pd\text{-}desc}$, $\mathsf{ec\text{-}desc}$, and $\mathsf{sc\text{-}desc}$ be three descriptors for protection domains, execution contexts, and scheduling contexts, respectively.

- Let $\mathsf{os}\colon \mathbb{N} \to \mathsf{Cap}_{\mathsf{Obj\text{-}space}}$ be the function that maps all indices to the null capability, i.e. $\mathrm{os}(i) = \mathtt{null\text{-}cap}$, except that an execution-context capability and a scheduling-context capability are at indices `predefined-capabilities` and `predefined-capabilities` + 1, respectively:

  $$\begin{aligned} \mathsf{os}(\mathtt{predefined\text{-}capabilities}) &= (\mathsf{ec\text{-}desc}),\\ \mathsf{os}(\mathtt{predefined\text{-}capabilities}+1) &= (\mathsf{sc\text{-}desc}). \end{aligned}$$

- Let pd be the protection domain that has os as its object space: $\mathsf{pd} = (\mathsf{os})$.

- Let ec be the execution context with the following fields:

  $$\begin{aligned} \mathsf{ec.pd} &= \mathsf{pd\text{-}desc},\\ \mathsf{ec.sc} &= \{\mathsf{sc\text{-}desc}\},\\ \mathsf{ec.reply\text{-}cap} &= \mathtt{null\text{-}cap},\\ \mathsf{ec.state} &= \text{ready}. \end{aligned}$$

- Let sc be the scheduling context that has $\mathsf{ec\text{-}desc}$ as its execution context field: $\mathsf{sc} = (\mathsf{ec\text{-}desc})$.
- Let σ be the kernel state that has the following fields:

$$\begin{aligned}\sigma.\mathsf{Pd} &= \varepsilon(\mathsf{pd\text{-}desc} \mapsto \mathsf{pd}),\\ \sigma.\mathsf{Ec} &= \varepsilon(\mathsf{ec\text{-}desc} \mapsto \mathsf{ec}),\\ \sigma.\mathsf{Sc} &= \varepsilon(\mathsf{sc\text{-}desc} \mapsto \mathsf{sc}),\\ \sigma.\mathsf{Pt} &= \varepsilon,\\ \sigma.\mathsf{Wq} &= \varepsilon,\\ \sigma.\mathsf{current\text{-}sc} &= \mathtt{null\text{-}desc}.\end{aligned}$$

  Here ε denotes the empty partial function, which is undefined for all argument values.

The initial system state is then defined to consist of σ with no active execution-context descriptors:

$$\mathsf{initial\text{-}system\text{-}state} \quad = \quad (\sigma,\ \varepsilon)$$

The initial state can only do an activation transition, making its only execution-context descriptor active and thereby choosing the first hyper call. The next transition is then an execution-context selection that sets $\sigma.\mathsf{current\text{-}sc}$ to $\mathsf{sc\text{-}desc}$, thereby making $\mathsf{ec\text{-}desc}$ the current execution-context descriptor. Then execution of the selected first hyper call starts.

5 Pseudo-Code Description of the Nova Hyper Calls

5.1 Create Protection Domain

5.1.1 Global Constants

The constant `predefined-capabilities` gives the number of capabilities that are used for exceptions and interrupts. It is announced in the hypervisor information page, see [Steb, 6.2]. The first free capability slot is at index `predefined-capabilities`.

5.1.2 Arguments

0. `self` descriptor of the execution context performing the IDC; implicit argument
1. `pd-cap` capability index of the new protection domain
2. `utcb-address`
3. `cap-range` capability range

The specification currently ignores the two arguments `utcb-address` and `cap-range`.

5.1.3 Pseudo Code

```
local variables
  new-pd : protection domain descriptor
  new-ec : execution context descriptor
  new-sc : scheduling context descriptor

// argument checking
// if invalid-utcb-address? utcb-address then error
if nonempty-cap? self.pd.obj-space[pd-cap] then error

// create new PD
new-pd := new PD()
self.pd.obj-space[pd-cap] := new PD-cap(new-pd)

// create new EC
```

```
new-ec := new EC(.pd := new-pd)
new-pd.obj-space[predefind-capabilities + 0] :=
                               new EC-cap(.kobj := ec-desc)

// create new SC
new-sc := new SC(.ec := new-ec)
new-pd.obj-space[predefind-capabilities + 1] :=
                               new SC-cap(.kobj := new-sc)

add(new-sc, new-ec.scheduling-contexts)

// delegate capabilities in cap-range from self to new-pd
```

5.2 Create Execution context

5.2.1 Arguments

0. `self` descriptor of the execution context performing the IDC; implicit argument

1. `ec-cap` capability index for the new execution context

2. `wq-cap` target wait queue

3. `utcb-address`

4. `SP` stack pointer

The specification currently ignores the arguments `utcb-address` and `SP`.

5.2.2 Pseudo Code

```
local variables
  new-ec : execution context descriptor
  wq : wait queue descriptor

// argument checking
// if invalid-utcb-address? utcb-address then error
if nonempty-cap? self.pd.obj-space[ec-cap] then error
if not is-wait-queue-cap? self.pd.obj-space[wp-cap] then error

// create EC
new-ec := new EC(.pd := self.pd)
self.pd.obj-space[ec-cap] := new EC-cap(.kobj := new-ec)

// enqueue and dispatch wait queue
```

```
wq := self.pd.obj-space[wq-cap].kobj
enqueue(wq.wait-queue, new-ec)
if nonempty? wq.send-queue then
  ec := dequeue(wq.send-queue)
  ec.state := ready
```

5.3 Create Scheduling Context

5.3.1 Arguments

0. `self` descriptor of the execution context performing the IDC; implicit argument
1. `sc-cap` capability index for the new scheduling context
2. `ec-cap` target execution context
3. `P` priority
4. `Q` quantum length

The specification currently ignores the arguments `P` and `Q`.

5.3.2 Pseudo Code

```
local variables
  ec : execution context descriptor
  new-sc : scheduling context descriptor

// argument checking
if nonempty-cap? self.pd.obj-space[sc-cap] then error
if not is-execution-context-cap? self.pd.obj-space[ec-cap] then error

ec := self.pd.obj-space[ec-cap].kobj

// create new SC
new-sc := new SC(.ec := ec)
add(new-sc, ec.scheduling-contexts)
self.pd.obj-space[sc-cap] := new-sc
```

5.4 Create Wait Queue

5.4.1 Arguments

0. `self` descriptor of the execution context performing the IDC; implicit argument

1. `wq_cap` capability index for the new wait queue
2. `donate` Boolean donation flag

The specification currently ignores the `donate` argument.

5.4.2 Pseudo Code

```
local variables
  new-wq : wait queue descriptor
  new_cap : wait queue capability

// argument checking
if nonempty-cap? self.pd.obj-space[wq-cap] then error

// create WQ
new-wq := new WQ()
self.pd.obj_space[wq-cap] := new WQ-cap(.kobj := new-wq)
```

5.5 Create Portal

5.5.1 Arguments

0. `self` descriptor of the execution context performing the IDC; implicit argument
1. `cap-portal` capability index for new portal
2. `cap-wq` index of target wait queue
3. `IP` instruction pointer
4. `mtd` message transfer descriptor

The specification currently ignores the arguments `IP` and `mtd`.

5.5.2 Pseudo Code

```
local variables
  wq : wait queue descriptor
  new-portal : portal descriptor

// argument checking
if nonempty-cap? self.pd.obj-space[cap-portal] then error
if not is-wait-queue-cap? self.pd.obj-space[cap-wq] then error
```

```
wq := self.pd.obj-space[cap-wq].kobj
new-portal := new Portal(.wq := wq)
self.pd.obj-space[cap-portal] := new Portal-cap(.kobj := new-portal)
```

5.6 Inter-Domain Communication: Send, Call

There are actually three closely related hyper calls for sending/calling: the donating call `idc-dcall`, the non-donating call `idc-ncall` and the simple send `idc-send`. We describe them here collectively (as it is done in the Nova documentation) and use an additional third argument, which can take one of the values `dcall`, `ncall` and `send`, to distinguish between the calls.

5.6.1 Arguments

0. `self` descriptor of the execution context performing the IDC; implicit argument
1. `portal-cap` target portal index
2. `mtd-send` message transfer descriptor for sending
3. `mode` one of `dcall`, `ncall` or `send`

5.6.2 Pseudo Code

```
local variables
  target : portal descriptor
  wq : wait queue descriptor
  partner : execution context descriptor
  sc : scheduling context descriptor

// argument checking
if not is-portal-cap? self.pd.obj-space[portal-cap] then error

target := self.pd.obj-space[portal-cap].kobj
wq := target.wq
while empty? wq.wait-queue do
  enqueue(wq.send-queue, self)
  self.state := blocking
  block
done
partner := dequeue(wq.wait-queue)

// transfer message to partner
```

```
if mode = dcall then
  sc := current-sc
  delete(sc, self.scheduling-contexts)
  sc.ec := partner
  add(sc, partner.scheduling-contexts)
else
  sc := null-desc

if mode = dcall or mode = ncall then
  partner.reply-cap := new Reply-cap(kobj := self, .sc := sc)

partner.state := ready

if mode = dcall or mode = ncall then
  self.state := blocking
  block
```

5.7 Inter-Domain Communication: Reply and Wait

The reply-and-wait hyper call does a reply if the reply capability register of the current execution context contains a valid reply capability. Otherwise there is no reply, and only the wait is performed.

5.7.1 Arguments

0. `self` descriptor of the execution context performing the IDC; implicit argument
1. `wq-cap` target wait queue index
2. `mtd-send` message transfer register for reply

5.7.2 Pseudo Code

```
local variables
  partner : execution context descriptor
  sc : scheduling context descriptor
  wq : wait queue descriptor
  ec : execution context descriptor

// argument checking
if not is-wait-queue-cap? self.pd.obj-space[wq-cap] then error

if is-reply-cap? self.rp then
  partner := self.rp.kobj
```

```
    sc := self.rp.sc

    // transfer message to partner

    if not sc = null-desc then
      delete(sc, self.scheduling-contexts)
      sc.ec := partner
      add(sc, partner.scheduling-contexts)

    self.rp := null-capability
    partner.state := ready

  wq := self.pd.obj-space(wq-cap)
  enqueue(wq.wait-queue, self)

  if nonempty? wq.send-queue then
    ec := dequeue(wq.send-queue)
    ec.state := ready

  self.state := blocked
  block
```

5.8 Capability Revocation

There is no description of capability revocation in [Steb] yet. Besides, a necessary prerequisite for revocation is capability donation, which is not yet described in [Steb] either. Revocation will be added to the formal specification when there is sufficient information on the subject available in the informal Nova documentation.

6 Conclusions

This document formally specifies the behavior of the Nova micro-hypervisor. The main part of the specification is given by imperative pseudo-code programs that describe how the different hyper calls affect the kernel data structures. Apart from specifying the behavior, the pseudo code serves a documentation purpose: because it is understandable at an intuitive level, it augments the natural language description given in the Nova documentation [Stea, Steb], making it much more precise.

In addition to giving the pseudo-code programs (which are almost identical to the ones in [Steb]), this document defines the abstract kernel state using only simple set theory, and it gives a formal semantics to the expressions and statements used in the pseudo code.

7 Bibliography

[Stea] Udo Steinberg. Nova architecture whitepaper. Robin deliverable D1.

[Steb] Udo Steinberg. Nova microhypervisor interface specification. Robin deliverable D2.

Index