Viengoos Developer Reference

Neal H. Walfield

December 15, 2008

Contents

Contents **iii**

1 Introduction **1**
1.1 Overview 1
1.1.1 Virtualizable Interfaces 2
1.1.2 Object Statelessness 2
1.2 Future Directions or TODO 2
1.2.1 Virtualization 3
1.3 Outline 4

I Viengoos **5**

2 Designation **7**
2.1 Capabilities 8
2.1.1 Format 8
2.2 Addressing 11
2.2.1 Address Encoding 11
2.2.2 Address Translation 12
2.3 Data Structures 14
2.3.1 addr 14
2.3.2 **addr_trans** 14
2.3.3 **object_policy** 14
2.3.4 **cap_properties** 15
2.3.5 cap 15

3 Threads and Activations **17**
3.1 Thread State 17
3.1.1 Address Space Root 17
3.1.2 Activity 18
3.1.3 Exception Messenger 18

3.1.4 User-Thread Control Block 18
3.2 Activations 21
3.3 Exceptions 22
3.4 Methods 22

4 Messengers and IPC **25**
4.1 Messages 25
4.1.1 Format 25
4.1.2 Canonical Form 26
4.2 Messengers 26
4.2.1 State 27
4.2.2 Message Transfer 28
4.3 IPC 29
4.3.1 Receive Phase 31
4.3.2 Send Phase 31
4.3.3 Return Phase 32

5 Resource Management **33**
5.1 Object Policy 33

6 Primordial Objects **35**
6.1 Objects 36
6.2 Folios 37
6.2.1 Data Structures 37
6.2.2 Methods 37
6.2.3 Convenience Functions 37
6.3 Pages 38
6.3.1 Methods 38
6.3.2 Convenience Functions 38
6.4 Threads 39
6.4.1 Methods 39
6.4.2 Convenience Functions 39
6.5 Messengers 40
6.5.1 Methods 40
6.5.2 Convenience Functions 40
6.6 Endpoints 41
6.6.1 Methods 41
6.6.2 Convenience Functions 41
6.7 Activities 42
6.7.1 Methods 42
6.7.2 Convenience Functions 42

7 Exceptions 43

8 Resource Management 45

II Runtime Environment 47

III Bibliograph 49

Bibliography 51

Chapter 1

Introduction

The text you are reading describes the Viengoos virtual machine. This text is an attempt to provide a normative reference for Viengoos, to enumerate its interfaces and to describe their behavior. It also attempts to explain the interfaces, to illustrate the motivation behind some decisions and to show the interfaces' intended uses. This interleaving of the prescriptive with the descriptive may be a source of confusion. This is unintentional and as this document evolves, such confusion will hopefully be eliminated.

1.1 Overview

Viengoos is an extensibility, object-capability system, ala Hydra [WCC+74] and EROS [SSF99].

Viengoos was built on the following ideas:

- object based,
- recursively virtualizable interfaces [PG74],
- object statelessness [TLFH96],
- no kernel dynamic allocation,
- resource accountability,
- atomic methods,
- caching [CD94]

- interrupt model [FHL+99],
- activation-based [Ros95], and
- resilience to destructive interference [Mil06]

1.1.1 Virtualizable Interfaces

By virtualizable interfaces, we mean that all kernel implemented objects can be easily proxied by a user-space task in such a way that the proxy behaves in a manner indistinguishable from the kernel implementation.

The idea is perhaps more easily explained through an example of a familiar object that is not easily virtualizable. Consider how files are implemented on Unix-like systems and suppose that one process wishes to proxy access to a file. The proxy can open the file itself and then provide another file descriptor to its clients. The question is what sort of file descriptor. A pipe could be used. In this case, the proxy will see the clients' reads and writes, however, a pipe does not support seeking and the kernel provides no way for the proxy to interpose on this operation and provide its own implementation.

1.1.2 Object Statelessness

Continuing the previous example, supposing that there was a way to cause such file invocations to be redirected to the proxy, another problem arises: proxying a file is non-trivial as the object's state machine is quite complicated. For instance, the proxy must maintain a file pointer for each client. This is because each client expects that the file descriptor it designates acts like a normal file descriptor, that is, that the file pointer is private. To work around this, the proxy must maintain a private file pointer for each client and then serialize access to the object and adjust the object's file pointer using the seek method before invoking the read method. This is required even if the proxy only wants to do some sort of bounds checking. To simplify virtualization, objects should avoid maintaining sessions: as much as is feasible, interfaces should be so designed such that a method either senses or transforms the state of the object.

1.2 Future Directions or TODO

The objects and interfaces described in this document (mostly) reflect the current implementation. There are a number of limitations requiring some thought. The

issues are outlined here.

1.2.1 Virtualization

Many of the methods are not virtualizable. Indeed, two of them are not even methods: cap_copy and cap_read are not invoked on an object but are essentially system calls. These should perhaps be modelled as thread object methods.

However, there is a more complicated problem and that is: the kernel walks the cappages and other objects to resolve an address but what should it do when it encounters an end point? The kernel cannot invoke it as then it must wait for a reply and this provides an opportunity for destructive interference. The kernel also cannot reply and tell the client to revert to some other method of resolution (or can it?). If we just fault, a process can determine whether an object is kernel or user-implemented by installing it in its address space and then trying to access it. Perhaps, the process that does the virtualization can do some tricks to provide a user-object but when the user tries to use it in its address space, by interposing on the thread object, it can the install an appropriate cappage.

As for the rest, to be able to virtualize a kernel object, the implementation needs to have access to all the information that the kernel implementation requires. This means that we either have to deliver more information or we have to adapt the interfaces.

An example of the former is to virtualize object_slot_copy_in, the implementation needs access to the source capability's subpage descriptor, policy and the weak predicates. This does not pose a fundamental problem: making this information available to the object does not violate POLA. It does require that when a message is delivered, any transferred capabilities must also include this information.

Another example is invoking an object: the object implementation also needs this information, in particular, the weak predicate and the subpage descriptor.

One place where the interfaces need to be change in a more fundamental way is object_slot_copy_out. The kernel implementation of this method does a bit copy from the designated source slot to the designated target slot. A user-implementation cannot do such a bit copy; it needs to return the capability in the reply message. To fix this, the target parameter needs to be a return value and object_slot_copy_out simply returns an appropriate capability. This raises another problem: how to then store the capability. If we store it in the receive buffer, then we still need another copy to get it where we want. If we store it directly in the address space (using a scatter/gather technique), then we need to consider the case where the target object is virtualized.

Also, for end points, we use the weak predicate to determine whether the capability interface designates the send facet or the receive facet. To properly virtualize a kernel object, we need to allow the user to control the weak predicate as normal.

1.3 Outline

This reference is divided into two parts. The first describes the kernel, how object addressing works, the primordial objects and resource management mechanisms and policies. The second part describes the sample run-time environment shipped with Viengoos.

Part I

Viengoos

Chapter 2

Designation

> "The name of the song is called 'HADDOCKS' EYES.' "
>
> "Oh, that's the name of the song, is it?" Alice said, trying to feel interested.
>
> "No, you don't understand," the Knight said, looking a little vexed. "That's what the name is CALLED. The name really IS 'THE AGED AGED MAN.' "
>
> "Then I ought to have said 'That's what the SONG is called'?" Alice corrected herself.
>
> "No, you oughtn't: that's quite another thing! The SONG is called 'WAYS AND MEANS': but that's only what it's CALLED, you know!"
>
> "Well, what IS the song, then?" said Alice, who was by this time completely bewildered.
>
> "I was coming to that," the Knight said. "The song really IS 'A-SITTING ON A GATE': and the tune's my own invention."
>
> *Through the Looking Glass*
> Lewis Carroll

Viengoos is an object-capability system. Objects are designated exclusively by way of capabilities, which are kernel-protected, unforgeable references. Capabilities are in turn designated by indexing an address space. Each thread object has a capability slot that identifies the root of its address space. When a thread invokes an object, it specifies an index. Viengoos finds the capability corresponding to this index in its address space and then dereferences the capability to obtain the object.

This chapter first describes how capabilities work, their format, and the kernel supported methods for manipulating capabilities. We then discuss addressing. Namely, how addresses are encoded, address space construction, and address resolution.

2.1 Capabilities

A capability both *designates* an object and *authorizes* access to it. (The importance of this is best illustrated by the Confused Deputy problem [Har88].) Capabilities are unforgeable in that they are kernel protected—their bit representation is never exposed—and thus can only be transferred via authorized channels.

To sense or modify an object, a thread may *invoke* it. Invocation causes a message to be sent to the object. The exact semantics of an invocation depend on the invoked object's implementation.

A capability may be delegated by transferring it in an object invocation. When a capability is transferred in such a way, the capability is copied to the receipient's message buffer. Because the receive buffer is allocated beforehand, copying does not require that the kernel allocate memory.

In Viengoos, the only way to revoke access to an object is to destroy the object.[1] By destroying the object, all capabilities designating it become invalid and act as if they designated the VOID object.

Viengoos allows user-object implementations. A user object is implemented by a process. The process allocates an end point and delegates it to clients. To use the object, a client invokes the end point. The server process is then notified that there is a message and may act on it as it sees fit.

As user objects are accessed in the same way as kernel objects, it is possible to interpose on specific objects or to fully or partially emulate the kernel from a user-space process.

2.1.1 Format

A capability is 128-bits wide and consists of the following fields:

- an object identifier (OID),

[1] Revocation can be implemented by way of Redell's Caretaker but so far, this mechanism has not been required.

- a version,
- a weak predicate (W),
- address translation directives,
  - a guard, and
  - a sub-page descriptor
- an object memory policy,
  - a discardability predicate (D), and
  - a priority

Object Identification

The OID field is used to locate an object. The OID corresponds to a block of storage on backing store. Backing store is managed by so-called backing store managers. When an object is referenced and the object is not in memory, Viengoos submits a request to page the object in to the appropriate backing store manager. Similarly, when Viengoos decides that the object should be flushed to persistent store, it sends a request to the backing store manager.

When an object is destroyed, all references to it must be invalidated. Invalidating references is difficult as it requires finding all of the references. Maintaining a linked list of capabilities referencing an object requires two additional pointers per capability. But this only suffices for in-memory objects: if a cappage is paged-out and the object is destroyed, these must be invalidated as well. To work around this problem, each object also has a version number. When a capability to an object is created, the object's version number is copied into the capability. Then, when dereferencing a capability, the capability is only considered valid if the the version numbers match. If they do not match, then the reference is known to not be valid and the VOID object is returned instead of the object instance.

The use of the version field raises another problem: it is limited in size. To avoid overflowing it and having to do a disk scavenge before being able to reuse the storage, it is imperative to control its growth. The solution EROS has used is to only bump the field if a capability designating the object goes to disk, a relatively rare occurrence, they observe, and to rate-limit that to once every few minutes [**?**].

Weak Capabilities

The data, cappage, endpoint, and activity objects implement two interfaces (facets): a so-called strong facet and a weak facet. The weak facet allows access to a subset of the functionality that the strong facet allows.

A capability designating the weak facet of a data-page provides read-only access to the object. The same applies for a cappage, however, the access is transitively removed: strong capabilities fetched via a weak capability are downgraded by the kernel to weak reference the object's weak facet. A capability designating the weak facet of an end-point only allows enqueuing messages. And, a capability designating the weak facet of an activity does not allowing changing the activity's policy.

Address Translation

In Viengoos, address spaces are composed through the arrangement of cappages; cappages act as page-tables. A thread object contains a capability slot, which is filled with the root capability. Some object methods all take a capability designating the root.

Viengoos uses a guarded page table scheme [Lie94]. To support this, capabilities contain two fields: a guard and a subpage descriptor. The guard consists of a value and a length. A subpage descriptor allows the use of only part of a capability page in address translation. It consists of a subpage count and an offset. The count indicates the number of subpages in the cappage. This value must be between 1 and 256 inclusive and be a power of 2. For example, a count of 2 means to divide the cappage into two subpages, each consisting of $256/2 = 128$ capabilities. The offset is then used to select the subpage to index. Address translation is discussed in section 2.2.2.

Object Memory Policy

To allow principals to control memory is managed, each capability contains two fields that describe the discardability and the priority of the designated object. Resource management is described in chapter 5.

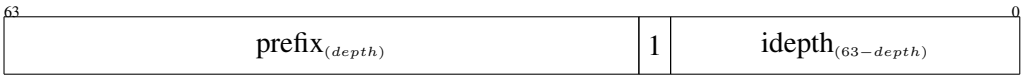
2.2 Addressing

Capabilities designated using thread-local addresses. Each thread object contains a capability slot that identifies the root of its address space. To designate a capability, a thread specifies the index of the capability in this address space.

2.2.1 Address Encoding

On Viengoos, all addresses are 64-bits wide. This is true even on 32-bit platforms. On these platforms, hardware addresses are automatically extended.

A Viengoos address consists of a **prefix** and a **depth**. The depth specifies the length of the prefix. This type of addressing allows addressing not only leaf objects but also internal nodes. (The intuition behind an addresses depth is how far into the tree to search.) The address prefix is encoded in the most significant bits of the address. This is followed by a bit with the value of 1, and then $63 - depth$ (*idepth*), which is encoded in unary.



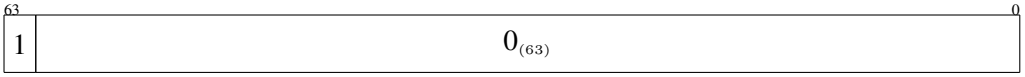
Observe that the value of idepth is the position of the least significant bit that is on.

The address with all zeros is the NULL address. The NULL address is sometimes used to denote some default action. When returned, it typically means failure.

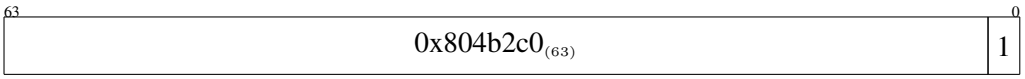
By convention, addresses are written *prefix/depth*.

Viengoos automatically translates machine addresses to the above form. The prefix is set to the machine address zero-extended to 63 bits and the depth is set to 63. For machines with 64-bits addresses, addresses with the most significant bit set are illegal.

The root capability slot is identified by the address 0/0. Its encoding is:



The address 0x804b2c0 is encoded:



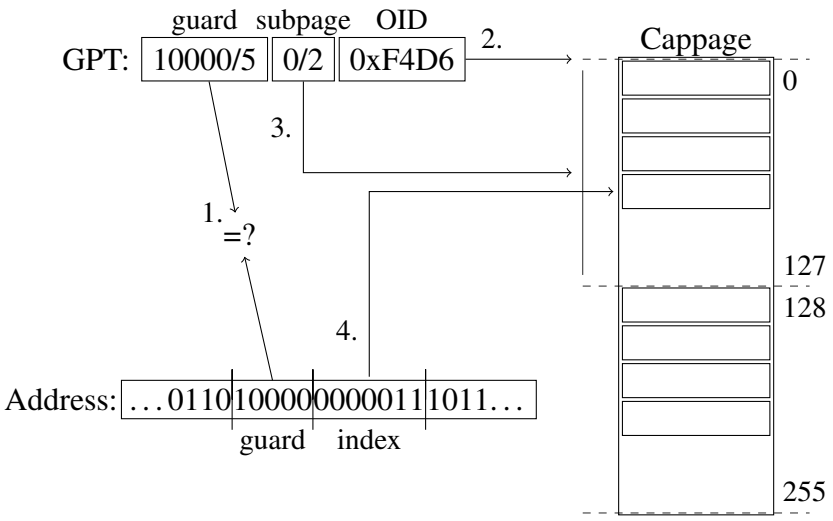


Figure 2.1: Translating part of an address using a GPT entry. The capability containing the GPT entry is at the top left in the figure, to the right is the referenced capability page, and bottom left is the address. First, the guard is compared to the address. If they match, the object is found. The subpage descriptor selects a part of the capability page, which is then indexed using the next portion of the address.

The address of the data object that contains the above byte would be the address rounded down to the nearest page size and with a depth of 63 - the logarithm base 2 of the page size. If the underlying hardware has base pages with a size of 4kb, then the address would be 0x804b000/51.

2.2.2 Address Translation

Address translation proceeds according to the following algorithm. Given an address, translation starts with the capability in the thread's address space capability slot. First, the most significant bits of the address are compared with the guard in the capability (lines 10–16). If these match, those address bits are consumed. If there are no address bits left, then the designated capability slot has been located and is returned. Otherwise, the object designated by the capability is found (line 21), divided according to the subpage descriptor in the capability and indexed using the most significant remaining bits of the address (lines 26–31). Again, the number of bits used to index the subpage are consumed. If all the bits are consumed, the capability slot has been located and is returned. Otherwise, the process is repeated with the new capability and the remaining address bits. An iteration of this process is illustrated in figure 2.1.

Algorithm 1 Capability slot lookup.

function THREAD → CAPABILITYSLOTLOOKUP($address$)
  $C \leftarrow thread.root$ ▷ The root of the address space.
  $P \leftarrow prefix(address)$ ▷ The bits to translate.
  $R \leftarrow depth(address)$ ▷ The number of bits remaining.

  loop
    if $R = 0$ **then**
      return $\&C$ ▷ C is the designated capability.
    end if

    ▷ Check the guard.
    if $R < guard\_length(C)$ **then**
      return failure ▷ Not enough bits to translate guard.
    end if
    if $guard(C) \neq P_{R..R-guard\_length(C)+1}$ **then**
      return failure ▷ The guard does not match.
    end if
    $R \leftarrow R - guard\_length(C)$

    if $R = 0$ **then**
      return $\&C$ ▷ C is the designated capability.
    end if

    ▷ Look up the object designated by the PTE.
    $O \leftarrow cap\_to\_object(C)$
    if $\neg O or typeof(O) \neq cappage$ **then**
      return failure ▷ Type mismatch.
    end if

    ▷ Index the capability page getting the next page table entry.
    $S \leftarrow 256/subpages(C)$ ▷ The subpage size.
    if $R < log_2(S)$ **then**
      return failure ▷ Not enough bits to index the cappage.
    end if
    $C \leftarrow O.caps \left[S/subpages(C) + P_{R..R-\log_2(S)+1}\right]$
    $R \leftarrow R - \log_2(S)$
  end loop
end function

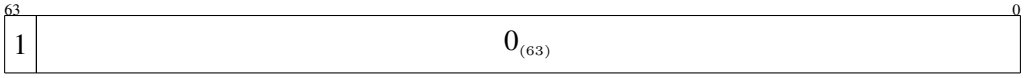
Note that a capability slot can be identified by two different names: either with or without the guard specified in the slot. This is a matter of convenience: it is useful to be able to modify the capability that designates the object at a particular address by designating the object. If this functionality were not provided, doing this would require finding the guard, which is possible but cumbersome. Moreover, the extension is quite simple.

When looking up objects, the same principle applies, however, the check if the address has been fully translated at lines 6–8 is removed. That is, it is not sufficient to specify the capability slot that designates the object, the guard must also match.

2.3 Data Structures

2.3.1 addr

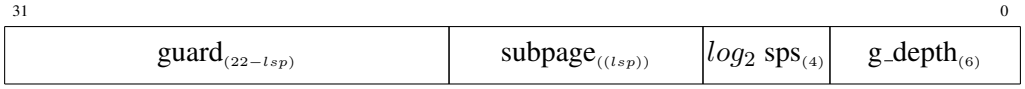
The format of an address is:



idepth is stored in unary. The depth is 63 - *idepth*.

2.3.2 addr_trans

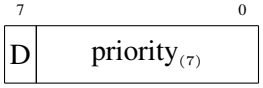
The **addr_trans** structure has the following layout:



log_2 *sps* is logarithm base 2 of the number of subpages. *subpage* is the subpage to select. It has a width of *lsp*. *g_depth* is the number of length of the guard. *guard* is the value of the guard and is zero-extended to *g_depth*. Its width is also not fixed.

2.3.3 object_policy

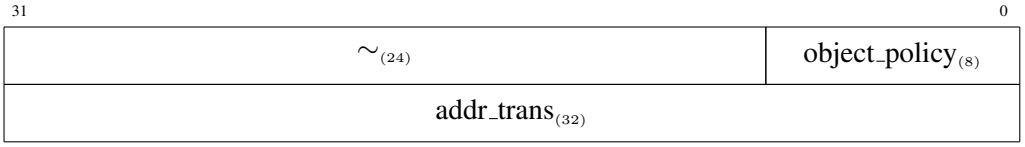
The **object_policy** structure has the following layout:



D is the discardability predicate.

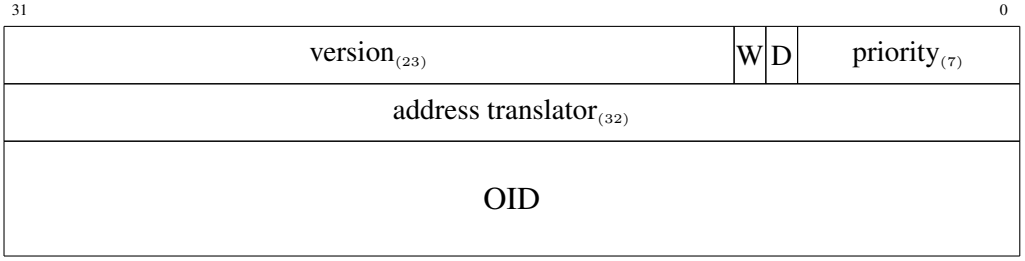
2.3.4 cap_properties

The **cap_properties** structure has the following layout:



2.3.5 cap

The following is the internal representation of a capability. Only the discardability predicate, the priority and the address translator are exposed to the user.



D is the discardability predicate. *W* is the weak predicate.

Chapter 3

Threads and Activations

A thread encapsulates an execution context. This consists of a register file, a name space, and a resource principal.

Viengoos does not implement a first-class task abstraction, which encapsulates multiple threads running in a single address space. It is possible to achieve this on Viengoos by specifying the same address space root for multiple thread objects.

In Viengoos, unlike in traditional kernels, threads are not blocking entities. Blocking is instead done by messengers, which hold and transfer messages (see chapter **??** for a description of messengers). By separating messengers from threads, it is possible to reliably wait for multiple events while the thread continues to execute and to do so in a manner that does not require the kernel or a server to block on the thread.

3.1 Thread State

A thread consists of four user-accessible capability slots: the address space root, the current activity the exception messenger and the user-thread control block (UTCB).

A thread object also contains space to save its CPU state, which it contains when the thread is not executing on a CPU.

3.1.1 Address Space Root

The address space root capability slot determines the root of a thread's address space. This address space is used in two situations. First, when a thread performs

an IPC, the arguments are identified by addresses, which are resolved in this address space. Second, this naming context determines the hardware address space. That is, it is the context in which the memory addresses to all hardware load and store instructions are resolved.

3.1.2 Activity

A thread's activity slot determines the thread's current activity. This is used to schedule the thread and to account resources that are allocated or consumed when resolving a page fault.

If the activity slot does not contain a capability designating an activity, the thread is not scheduled.

3.1.3 Exception Messenger

When a thread generates an exception, for instance, when it attempts to access a memory location for which there is no valid translation, the thread is suspended, and an exception message is generated and delivered to the thread's exception messenger. The kernel delivers execptions in non-blocking mode meaning if the exeception messenger is not ready to receive a message, the message is dropped.

3.1.4 User-Thread Control Block

The user-thread control block (UTCB) is a normal data page. It provides a conduit for the user and kernel to communicate and coordinate action. This is primarily used for managing activations and in message delivery.

activated mode The *activated mode* bit indicates whether the thread is in activated mode. An activation is only delivered if this bit is clear. Before a thread is activated, this bit is checked. If it is set, the activation is either delayed or dropped. In the former case, the messenger causing the activatation blocks on the thread. A blocked messenger may be unblocked either by a future activation or if thread_activation_collect is called. If the messenger is destroyed or the delivery aborted, the activation will not be delivered.

pending message The kernel sets the *pending message* bit when a messenger attempts to deliver a message to the thread, however, the thread is in activated mode.

```
struct utcb {
  union {
    struct {
      uintptr_t activated_mode : 1;
      uintptr_t pending_message : 1;
      uintptr_t interrupt_in_transition : 1;
    };
    uint64_t status;
  };

  uintptr_t saved_ip;
  uintptr_t saved_sp;

  uintptr_t activation_handler_sp;
  uintptr_t activation_handler_ip;
  uintptr_t activation_handler_end;

  uint64_t protected_payload;
  uint64_t messenger_id;

  /* Inline data. */
  struct {
    uintptr_t inline_word_count : 2;
    uintptr_t inline_cap_count : 1;
  };
  uintptr_t inline_words[2];
  addr_t inline_caps[1];
};
```

Listing 3.1: The user-thread control block structure.

interrupt in transition The *interrupt in transition* bit is set by the kernel when activating a thread to indicate whether the thread was activated while the instruction pointer was in the so-called *activation transition range* (see *activation handler start* and *activation handler end* below).

saved ip and **saved sp** When delivering an activation, the kernel interrupts the thread, sets the *saved ip* and *saved sp* variables to the instruction pointer and the stack pointer, sets the thread's sp and ip to *activation handler sp* and *activation handler ip*, respecitve, and then resumes the thread.

activation handler sp and **activation handler ip** When the kernel activates a thread, it sets its SP and IP to these values, respectively.

activation handler ip and **activation handler end** These variables determine the so-called *activation transition range*.

If the kernel activates a thread and its in the activation transition range (between *activation handler ip* inclusive and *activation handler end* exclusive, the kernel does not save the current IP and SP in *saved ip* and *saved sp* but sets the *interrupt in transition* bit. Using this mechanism, on many architectures, it is possible to atomically, with respect to activation delivery, clear *activated mode* and restore the interrupted stack pointer and instruction pointer without entering the kernel

protected payload When delivering a messenger's message, the *protected payload* variable is set to the protected payload of the capability that was used to invoke the messenger to send the message.

messenger id When delivering a messenger's message, the *messenger id* is set to the messenger's *message id*.

inline word count, inline cap count, inline words and **inline caps** When a messenger delivers a message inline, *inline word count* is set to the number of words (not bytes) that were transferred and *inline cap count* to the number of capabilities transferred. The *inline words* variable is filled with any data. If the message includes capabilities, they are saved sequentially in the slots specified at invocation time and the address is copied to *inline capability*. If an error occurs while transferring a capability, the corresponding elemeent in *inline capabilities* is set to ADDR_VOID.

3.2 Activations

A thread may be activated if a messenger associated with the thread transfers or receives a message. Threads are only activated by messengers and scheduling events. In the case of scheduling events, a kernel-provided messenger whose *message id* variable is set to ˜0ULL is used and delivery is done in a non-blocking manner.

Activating a thread proceeds as follows:

- Atomically, with respect to the thread's execution:
  - If the thread is activated (*activated mode* is non-zero):
    - If delivery is non-blocking, return.
    - Otherwise:
      - Block the messenger on the thread, and
      - Set *pending message* to 1.
- Suspend the thread's execution.
- Set *protected payload* to that saved in the messenger.
- Set *messenger id* to the messenger's messenger id.
- If the activation is the result of a message receipt:
  - If the message is inline:
    - Copy the messenger's data to *inline words* and *inline caps*
    - Update *inline word count* and *inline cap count*.
- Set *activated mode* to 1.
- If the thread's IP is between *activation handler ip* (inclusive) and *activation handler end* (exclusive):
  - Set *interrupt in transition* to 1.
- Otherwise:
  - Set *interrupt in transition* to 0,
  - Set *saved sp* to the thread's stack pointer, and
  - Set *saved ip* to the thread's instruction pointer.

- Set the thread's stack pointer to *activation handler sp*.
- Set the thread's instruction pointer to *activation handler ip*.
- Resume the thread.

3.3 Exceptions

Exceptions are synthesized by the kernel in response to a thread action. There is one type of exception, a fault exception.

The following actions result in the generation of a fault exception:

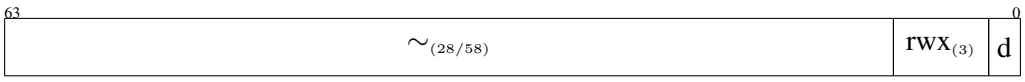
page fault A hardware load or store attempts to dereference an address for which there is no valid translation.

access fault A hardware load or store attempts to accesses an object in an unauthorized manner.

discarded A hardware load or store attempts to access an object that has been discarded.

When a thread generates an exception, the thread is suspended and a fault message is delivered to its exception messenger. If this would block, the message is discarded.

A fault message takes four parameters: the address of the fault (using Viengoos address encoding), the value of the stack pointer, the value of the instruction pointer and a fault information structure, which includes the type of access and whether the object has been discarded:



The fault message does not include a reply messenger.

3.4 Methods

thread_exregs(**cap_t** activity, **cap_t** thread,
uintptr_t flags,
in out cap_t aspace, **in out cap_t** activity,
in out cap_t utcb, **in out cap_t** exception_messenger,
in out uintptr_t sp, **in out uintptr_t** ip)

flags is a bit-wise or of the following

THREAD_EXREGS_SET_UTCB = 64
THREAD_EXREGS_SET_EXCEPTION_MESSENGER = 32
THREAD_EXREGS_SET_ASPACE = 16
THREAD_EXREGS_SET_ACTIVITY = 8
THREAD_EXREGS_SET_SP = 4
THREAD_EXREGS_SET_IP = 2
THREAD_EXREGS_GET_REGS = 1

If THREAD_EXREGS_GET_REGS is set, the current value of the address space root, activity, utcb, exception messenger, sp and ip are returned. Otherwise, the values are undefined.

If THREAD_EXREGS_SET_IP is set, the thread's instruction pointer is set according to *ip*.

If THREAD_EXREGS_SET_SP is set, the thread's stack pointer is set according to *sp*.

If THREAD_EXREGS_SET_ACTIVITY is set, the thread's activity is set according to *activity*.

If THREAD_EXREGS_SET_ASPACE is set, the thread's address space root is set according to *aspace*.

If THREAD_EXREGS_SET_EXCEPTION_MESSENGER is set, the thread's exception messenger is set according to *exception_messenger*.

If THREAD_EXREGS_SET_UTCB is set, the thread's UTCB is set according to *utcb*.

thread_id (**cap_t** activity, **cap_t** thread, **out uint64_t** id)

Return the thread's unique identifier.

thread_activation_collect (**cap_t** activity, **cap_t** thread)

Cause a blocked messenger, if any, to attempt to send an activation.

Chapter 4

Messengers and IPC

IPC in Viengoos is asynchronous with respect to thread execution. This is achieved by separating the messaging functionality from threads. To send an IPC, a program allocates a so-called messenger, loads a payload and then enqueues the messenger on a receiving messenger. When the receiving messenger accepts the message from the sending messenger, the message is copied and the threads associated with the two messengers are optionally notified of what has occurred by way of an activation.

As messengers must be explicitly allocated like any other kernel object, the required storage can be correctly accounted.

4.1 Messages

Messages can carry both data and capabilities. A message is stored in a normal data page. To send a message, the page containing the message is associated with a messenger, which is then enqueued on the target object. When the target accepts the message, the source messenger's message is copied to the target messenger's message buffer.

4.1.1 Format

The kernel interprets a message buffer according to the following format. The first 32-bits of the buffer contain the message header. This consists of a 16-bit capability address count followed by a 16-bit data count. Immediately following the header is the array of capability addresses, followed by the array of bytes.

```
struct message {
  uint16_t cap_count;
  uint16_t data_count;

  addr_t caps[cap_count];
  char data[data_count];
};
```

Listing 4.1: Message format.

When used to send a message, the capability addresses are interpreted as the location of the capabilities to send. When used to receive a message, the capability addresses are interpreted as the slots in which the received capabilities should be stored.

4.1.2 Canonical Form

Kernel objects interpret and format messages according to the following convention.

For received message, the first word of a message is interpreted as the method to invoke on the object. The remaining bytes are the methods arguments. The last capability is interpreted as the messenger to reply to.

When sending a reply, the first data word is the negation of the method identifier. The second word contains the error code. If no error occurred, the error code is set to 0.

When marshalling and unmarshalling arguments, the size of each argument is rounded up to be a multiple of the word size and zero filled; the sign bit is not extended.

4.2 Messengers

Messengers are first-class kernel objects, which are responsible for receiving and transferring messages.

A messenger references a message buffer and a thread. It can either transfer its contents to another messenger or its can wait for another messenger to send it a message. After sending or receiving a message, a messenger optionally notifies its associated thread by way of an activation. This is depicted in figure 4.1.

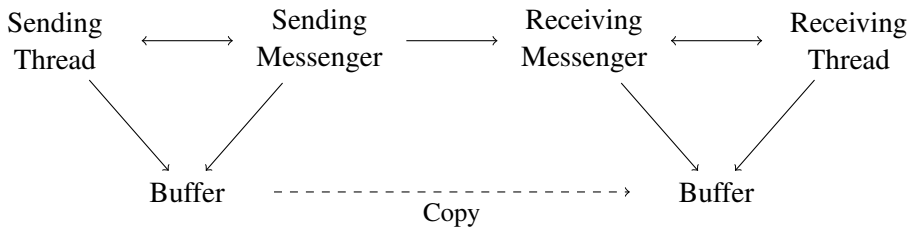


Figure 4.1: To send or receive a message, a thread associates itself and a buffer with a messenger. To send a message, it enqueues the prepared messenger on some other messenger. When the latter messenger accepts the former's message, the contents of the sending messenger's message buffer is copied to the receiving messenger's message buffer. The associated threads are then optionally activated.

To send a message, a messenger is enqueued on another messenger. The messenger's payload is only transferred to the target messenger once the messenger is unblocked.

To prevent unprocessed messages from being overwritten, messengers are blocked on message delivery. A further message is only delivered once the messenger is explicitly unblocked.

Message payload may be stored either in a message buffer or inline. This is specified in the IPC interface.

4.2.1 State

A messenger has four capability slots: a thread slot, an address space root slot, a message buffer slot and an activity slot. A messenger also contains a so-called *messenger id* field, and a blocking status.

Thread

The thread slot specifies the thread to optionally activate when the messenger transfers its message or receives a message. This is controlled via the IPC system call.

Address Space Root

The address space root specifies the address space in which to interpret the capability addresses in the message buffer.

Message Buffer

The message buffer slot identifies the message buffer.

Activity

The activity identifies the activity used to send the message.

Messenger ID

The messenger ID is a 64-bit user-settable variable that is delivered to the thread (in its UTCB) on activation. This can be used to identify a user-buffer associated with a messenger. This variable can only be read or modified by way of a strong capability.

Blocking Status

A messenger is either block, in which case any attempts to deliver a message to it will block, or it is unblocked, in which an attempt to deliver a message to it will succeed immediately.

4.2.2 Message Transfer

Messages are transfered between two messengers, a source messenger and a target messenger. A message transfer only occurs when the target messenger is not blocked. If a message transfer is attempted and the target messenger is blocked, then the transfer is either aborted (if the transfer is executed in non-blocking mode), or the source messenger is enqueued on the target messenger (otherwise).

Message transfer proceeds as follows:

- The target messenger is blocked.
- The capabilities in the source message are matched with the capability slots in the target message.

  For each pair of capability address and capability slot address, the capability and slot are looked up relative to their respective messenger's address space root. If there is a valid, writable capability slot, the source capability is copied to it. If there is no source capability, a void capability is used.

The target capability slot's address translator and policy are preserved. If the capability slot address does not resolve to a capability slot, or, the capability slot is not writable, the capability address in the target message is overwritten with ADDR_VOID.

If there are more capability slot addresses then there are capability addresses, each of the remaining capability slot addresses is overwritten with ADDR_VOID.

- The data is byte copied from the source to the destination.
- If the target messenger is set to activate its associated thread on receive, this is scheduled. Likewise, if the source messenger is set to activate its associated thread on delivery, this is scheduled.

4.3 IPC

IPC consists of three phases: the receive phase, the send phase and the return phase. All three phases are optional. Each phase is executed after the previous phase has completed. If a phase does not complete successfully, the phase is aborted and the remaining phases are not executed.

The IPC interface has the following signature:

error_t
ipc (**uintptr_t** flags,
cap_t recv_activity, **cap_t** recv_messenger, **cap_t** recv_buf,
cap_t recv_inline_cap,
cap_t send_activity, **cap_t** target_messenger,
cap_t send_messenger, **cap_t** send_buf,
uintptr_t send_inline_word1, **uintptr_t** send_inline_word2,
cap_t send_inline_cap)

The flags parameter selects which phases are executes and controls their execution. It has the following format:

The receive flags are:

R - receive phase The IPC includes a receive phase.

N - non-blocking The receive phase is non-blocking.

A - activate On message receipt, the receiving messenger activates its associated thread.

T - set thread Associate the receiving messenger with the calling thread.

S - set address space root Set the receiving messenger's address space root to the caller's address space root.

I - receive inline The receiving messenger should receive the message inline.

C - inline capability Ignored if **I** is not set. The inline message includes a capability slot at *recv inline cap*.

The send flags are:

S - send phase The IPC includes a send phase.

n - non-blocking The send phase is non-blocking.

a - activate On message delivery, the receiving messenger activates its associated thread.

t - set thread Associate the sending messenger with the calling thread.

s - set address space root Set the sending messenger's address space root to the caller's address space root.

i - send inline The sending messenger should receive the message inline.

W - inline words Ignored if **i** is not set. The number of inline words to transfer. Valid values are 0, 1 and 2.

c - inline capability Ignored if **i** is not set. The number of inline capabilities to transfer. Valid values are 0 and 1.

The return flags are:

r - return phase The IPC includes a return phase.

The remaining parameters are described below. The capability addresses are resolved in the context of the caller's address space.

4.3.1 Receive Phase

The receive phase proceeds as follows:

- *recv messenger* is looked up. If it does not designate a messenger or the designation is not strong, the IPC is aborted and EINVAL is returned.
- If the message is not inline and *recv buf* is not ADDR_VOID, the messenger's message buffer capability slot is set to the capability designated by *recv buf*.
- If the set associated thread flag is set, the messenger's thread capability slot is set to a capability designating the caller's thread object.
- If the set address space root flag is set, the messenger's address space root capability slot is set to the calling thread's address space root. The thread's address space root's address translator and policy are copied.
- If one or more messengers are blocked on *recv messenger* trying to delivery a message, the messenger which has blocked longest is selected and its payload is transferred to *recv messenger*. See section 4.2.2 for details.
- If there are no messengers blocked on *recv messenger* trying to delivery a message and the non-blocking flag is set, EWOULDBLOCK is returned. Otherwise, *recv messenger* is unblocked.

4.3.2 Send Phase

The send phase proceeds as follows:

- *send messenger* is looked up. If it does not designate a messenger or the designation is not strong, the IPC is aborted and EINVAL is returned.
- If the message is not inline and *send buf* is not ADDR_VOID, the messenger's message buffer capability slot is set to the capability designated by *send buf*.
- If the set associated thread flag is set, the messenger's thread capability slot is set to a capability designating the caller's thread object.
- If the set address space root flag is set, the messenger's address space root capability slot is set to the calling thread's address space root. The thread's address space root's address translator and policy are copied.

- *target messenger* is looked up. If it does not designate a messenger, the IPC is aborted and EINVAL is returned.
- An attempt to deliver *send messenger*'s message to *target messenger* is made. If *target messenger* is blocked and delivery is non-blocking, delivery is aborted and ETIMEDOUT is returned. Otherwise, if *target messenger* is blocked, *send messenger* is blocked on it. Otherwise the message is delivered. See section 4.2.2 for message delivery details.

4.3.3 Return Phase

The return phase proceeds as follows:

- Control is returned to the calling thread to just after the IPC call. Note that control is not returned by way of an activation.

If the IPC does not include a return phase, then thread blocks until it is next activated by an event other than a CPU available event.

Chapter 5

Resource Management

5.1 Object Policy

When an object is accessed, if the object is claimed,[1] the policy in the designating object is applied to the object.

The discardability property is a hint that Viengoos may, instead of flushing changes to disk, simply discard a frame's content. If a capability has the weak predicate set, this hint is ignored. If content discarded, the next access to the object will raise a discarded event. If an activity is discarded, all objects allocated against the activity are destroyed.

The priority property allows an activity to control the order in which the frames, which it has claimed, are released. If the content is dirty and has not been marked as discardable, the content is written to backing store. Otherwise, the frame is made eligible for immediate reuse.

The lower the numric value of the priority field, the lower the frame's priority. Frames are released in priority order. If multiple frames have the same priority, they are released in a random order unless the priority is 0, in which case, the frames are released in approximately LRU order.

[1] Claiming is discussed in **??**.

Chapter 6

Primordial Objects

> I. The world is everything that is the case.
> I.I The world is the totality of facts, not of things.
> I.II The world is determined by the facts, and by these being *all* the facts.
> I.I2 For the totality of facts determines both what is the case, and also all that is not the case.
>
> *Tractatus Logico-Philosophicus* by Ludwig Wittgenstein

This chapter describes the primordial objects implemented by the microkernel. They include folios, the unit of storage allocation, data and capability pages, threads, message buffers, end points, and activities. These objects represent the fundamental building blocks of the system; all other objects are built from compositions of these objects.

6.1 Objects

All objects are derived from the generic base object object. Each object has a number (possibly zero) of user-accessible capability slots.

cap_copy (**addr_t** principal, **addr_t** object, **addr_t** target,
addr_t source_address_space, **addr_t** source,
uint32_t flags, **struct cap_properties** properties)

Copy the capability in the capability slot *source* in the address space rooted at *source_address_space* to *object*'s slot at address *target*.

By default, preserves *source*'s subpage specification and *target*'s guard.

If CAP_COPY_COPY_SUBPAGE is set, then uses the subpage specification in CAP_PROPERTIES. If CAP_COPY_COPY_ADDR_TRANS_GUARD is set, uses the guard description in CAP_PROPERTIES.

If CAP_COPY_COPY_SOURCE_GUARD is set, uses the guard description in source. Otherwise, preserves the guard in TARGET.

If CAP_COPY_WEAKEN is set, saves a weakened version of SOURCE in *TARGET (e.g., if SOURCE's type is cap_page, *TARGET's type is set to cap_rpage).

If CAP_COPY_DISCARDABLE_SET is set, then sets the discardable bit based on the value in PROPERTIES. Otherwise, copies SOURCE's value.

If CAP_COPY_PRIORITY_SET is set, then sets the priority based on the value in properties. Otherwise, copies SOURCE's value.

cap_read (**addr_t**, principal, **addr_t**, address_space, **addr_t**, cap,
l4_word_t, type, **struct cap_properties**, properties)

Returns the public bits of the capability CAP in TYPE and CAP_PROPERTIES.

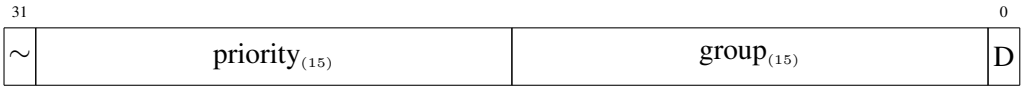
6.2 Folios

A folio is the unit of backing store allocation. A folio consists of 129 4k pages. 128 may be used to allocate objects and the remainder is a header that describes the folio itself and the individual objects.

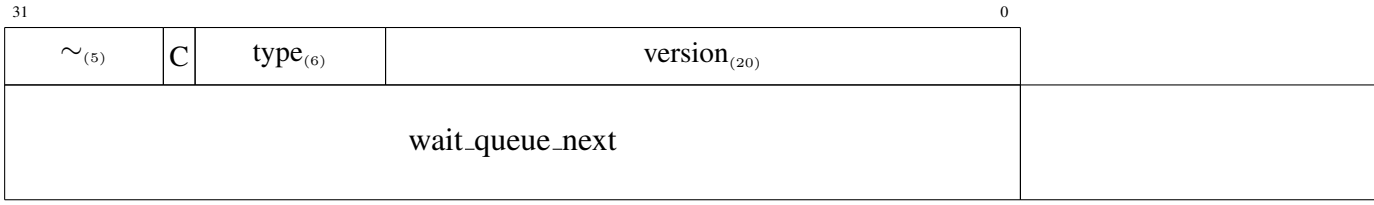
The header holds a

6.2.1 Data Structures

folio_priority



D is the discardability predicate.



6.2.2 Methods

6.2.3 Convenience Functions

6.3 Pages

Data pages and capabilities pages.

6.3.1 Methods

6.3.2 Convenience Functions

6.4 Threads

6.4.1 Methods

6.4.2 Convenience Functions

6.5 Messengers

6.5.1 Methods

6.5.2 Convenience Functions

6.6 Endpoints

6.6.1 Methods

6.6.2 Convenience Functions

6.7 Activities

An activity is a resource principal.

6.7.1 Methods

6.7.2 Convenience Functions

Chapter 7

Exceptions

Exception handling mechanism.

Chapter 8

Resource Management

Part II

Runtime Environment

Part III

Bibliograph

Bibliography

[CD94] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 179–193. USENIX Association, November 1994. 1

[FHL+99] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the fluke kernel. In *Operating Systems Design and Implementation*, pages 101–115, 1999. 2

[Har88] Norm Hardy. The confused deputy (or why capabilities might have been invented). Technical report, Key Logic, 1988. 8

[Lie94] Jochen Liedtke. Page table structures for fine-grain virtual memory. Technical Report 872, German National Research Center for Computer Science (GMD), October 1994. 10

[Mil06] Mark S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May 2006. 2

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974. 1

[Ros95] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge, August 1995. 2

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999. 1

[TLFH96] Patrick Tullmann, Jay Lepreau, Bryan Ford, and Mike Hibler. User-level checkpointing through exportable kernel state. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI 96)*. USENIX Association, 1996. 1

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974. 1