Rose

Capability-based persistent Ada OS

First-name Last-name

July 8, 2021

Contents

List of Figures

List of Tables

Preface

1

Introduction

2

Capabilities

2.1 Introduction

A capability represents permission to perform one or more operations. The possession of a capability for an operation O is both necessary and sufficient for performing O.

For the process, a capability is a 32-bit word, with no particular structure. The kernel uses the process capability to index a private table of capability structures. Each structure defines the capability type, and the object to which it applies.

Capability structures are stored in pages. The first 32 capabilities used by a process are stored directly in the kernel process structure; the rest are stored in private pages mapped to the process.

2.2 Capability Types

Type	Description
Null	The null capability; produces an error if used
Page Object	Controls access to physical memory
Schedule	
Process	
Interface	
Endpoint	
Set	
Kernel	

Table 2.1: Capability Types

2.2.1 Generic Capability Layout

A capability structure is 16 bytes long. The particular layout depends on the capability type. Common fields are as follows.

31 12	11 8	7	6 4	3 0
Identifier	Use$_{(4)}$	I	Flags	Type
Endpoint				
Payload				

The `Identifier` is an identifier which must be matched when using a capability. It is ignored unless the `I` flag is set.

`Use` is the remaining use count. If this is zero, the capability can be used any number of times. Otherwise, each use of the capability decrements `Use` by 1, and when it reaches zero, the capability is overwritten with the null capability.

`Flags` are capability-specific flags.

The `Endpoint`, if non zero, is an index into the kernel-private endpoint array.

The `Payload` is a type-specific payload.

2.2.2 Endpoint Capabilities

An endpoint capability is used to invoke an endpoint on the remote object. If the process is currently listening on the capability's endpoint, the receive invocation will return with the corresponding invocation record.

Endpoint Capability Layout

31 16	15 8	7	4	3 0
Identifier	Use$_{(4)}$	I	0	0 1 0 1
Endpoint				
Object Id				

2.2.3 Page Object Capabilities

A page object gives access to a shared buffer. Three endpoints are implemented: `copy-to-buffer`, `copy-from-buffer`, and `map-device-memory`.

Page Object Capability Layout

31 16	15 8	7			4	3 0
Identifier	Use$_{(4)}$	I	0	R	W	0 1 0 0
Endpoint						
Page Object Id						

2.2.4 Process Capabilities

A process capability is used to communicate with a running process. Two actions are possible, depending on the value of the `Op` field. If Op is zero, the capability returns the process's default interface. If Op in one, the capability sends the capabilities in the invocation to the process.

Process capabilities are normally given to the parent process by the kernel when the child is launched. The parent can then query the child, or send it capabilities. For example, when the Petal shell executes a program, it uses the process capability to send the default input, output, and error streams (amongst other interfaces).

Process Capability Layout

31 16	15 8	7	4	3 0
Identifier	Use$_{(4)}$	0	Op	0 1 0 0
0				
Process Object Id				

2.2.5 Boot Capabilities

These capabilities are only used during system initialisation. At boot, the kernel launches the process *init*, which is pre-configured with a script which starts the rest of the system. Boot capabilities are used to accomplish this.

Boot Capability Endpoints

Operation	Arguments
Launch Boot Module	Module index, caps

Table 2.2: Boot capability endpoints

Boot Capability Layout

31 … 16	15 … 8	7 … 4	3 … 0
Identifier	Use$_{(4)}$	0	1 0 0 0
0			
0			
0			

2.2.6 Low Level Capabilities

These are capabilities which allow device drivers to talk to hardware. The are necessarily architecture-dependent. Some examples are shown below.

Port IO Layout (i686)

This capability allows a process to execute the equivalent of `out addr, reg` and `in addr` instructions.

31 … 16	15 … 8	7	6 … 4	3 … 0
Identifier	Use$_{(4)}$	0	Sz	1 1 1 0
Endpoint				
First Port				
Last Port				

The Sz field specifies 1-, 2-, 4- or 8- byte data for port in/out instructions (i.e. number of bytes per instruction is 2^{Sz}).

The possible endpoints are as follows.

Endpoint	Op	Description
1	port out	Each invocation word is sent to First Port
2	port in	Each receivable word is set by an in on First Port
3	port out range	invocation data specifies (port offset, value) pair for port-out
4	port in range	invocation data specifies port offsets, result written to received words

Invocation word format

The sent words layout for the port-out-range endpoint varies depending on the size of the data. For 8-bit data, each sent word consists of the value in byte 0, and the offset in byte 1. 16-bit data is encoded into bytes 0 and 1, with the offset in byte 2. 32-bit data is encoded in up to three groups of five sent words. The first word of each group has four port offsets, while the second, third, fourth and fifth words contain the corresponding data for each port.

For the port-in-range endpoint, each sent word contains a port offset, and when returning the received words contain the result of the `in` instruction. If the received word count is larger than the sent word count, the extra offsets will all be zero. Extra sent words are ignored.

In all cases, an offset of 255 indicates that no data should be sent. Any offset that is not within the port range is also ignored.

2.3 Invocation

A capability is invoked by making a system call using an *invocation record.*

2.3.1 Invocation Record

31 24	23 20	19 16	15 12	11 8			5	4	3	2	1	0
reserved	Use	CC	RWC	SWC	W	P	E	S	R	Y	B	C
Capability												
Reply Capability												
Shared buffer length												
Shared buffer address												
Up to 15 capabilities												
Up to 15 argument words												

The `RWC` field is the number of words which the sender is prepared to receive in response to the invocation. A reply to this message cannot contain more words than specified here, although it can contain less.

The `SWC` field is the number of sent words.

The `CC` field is the number of sent words.

The `Capability` field contains the capability being invoked. The `Reply Capability` field is set by the kernel, and gives the target of an invocation a capability on which to send a reply.

If any capabilities are sent, the `Use` field limits the number of times they can be used. An allocation count of zero means there is no limit.

Id	Name	Description
W	Write	If the P flag is set, allow the receiver to write to the buffer
P	Pages	Send one or more shared buffers
E	Error	An error occurred (code is in first data word)
S	Send	Unprompted send on a capability
R	Receive	Willing to receive messages
Y	Reply	This is a reply to an earlier send
B	Block	The sender will block until the next message
C	Cap Construct	Create a reply cap for the message response
SWC	Sent Word Count	SWC contains the number of sent words
RWC	Receive Word Count	RWC contains the maximum received words
CC	Capability Count	Number of sent words which are capabilities

Table 2.3: Invocation Flags

3

Processes

A process is named by a launch capability. Invoking the capability executes the process.

Object ids in the range 0 to $2^{24} - 1$ represent processes.

Process id 1 is used to refer to the Kernel.

3.1 Launch Capability

3.2 Initial Capabilities

The initial environment is inherited from the parent process, along with capabilities defined when the process is installed.

The generated interface libraries rely on caps 1, 2, and 3 being the ones from table 3.1.

The standard libraries rely on cap 4 being a capability for the `Cap_Set` interface. The set referenced by this cap is defined by Table 3.2. The environment accessed by the AdaĖnvironment_Variables package contains an entry for each *Id* in the table, with the index of the corresponding capability in the cap set.

#	Cap	Type	Description
1	Destroy	Method	Ends process
2	Create Endpoint	Method	Endpoint constructor
3	Create Cap_Set	Method	Cap set constructor
4	Argument_Caps	Interface	Cap set containing supplied caps

Table 3.1: Standard process capabilities

#	Id	Interface	Description
1	Standard_Input	Stream_Reader	Standard input stream
2	Standard_Output	Stream_Writer	Standard output stream
3	Standard_Error	Stream_Writer	Standard error stream
4	Current_Directory	Directory	Start directory
5	Heap	Heap	Heap management
6	Clock	Clock	System clock

Table 3.2: Standard capability environment

3.3 Example

Consider the following excerpt from a shell session:

```
user$ cat readme.txt
```

A launch capability for the `cat` program is found in the shell's environment. A new environment is created, setting environment capabilities for input/output streams. The argument is interpreted as the name of a file, a `stream-reader` capability to this file is added to the environment.

3.4 Environment

The environment is supplied as a table of entries starting at a known address.

4

Memory

Physical memory is managed by the memory manager. Page objects are managed by the page object manager. A process belongs to a particular space bank, which represents a range of page objects. Space banks are hierarchical. A page object is a direct member of at most one space bank. At the top of the space bank hierarchy is the root space bank, which encompasses the entire page object id range.

Each process executes in a flat memory space starting at zero. These are the virtual pages. Virtual pages are mapped or unmapped. A mapped virtual page references a corresponding physical page. A physical page may be mapped by any number of virtual pages. This mapping is not persisted.

4.1 Example

The command `ls` lists the contents of a directory. A user shell normally has a launch capability for `ls`. When this capability is invoked, a new process is created. A snapshot of part of the memory state might look something like table 4.1.

Page Object	Virtual Address	Physical Address	Description
$ls + 0$	`0000 0000`	`4020 C000`	1st page of ls text segment
$ls + 1000$	`0000 1000`	`ABCD 2000`	2nd page of ls text segment
$ls + 2000$	`0000 2000`		3rd page (not paged in yet)
$base + 3000$	`BFFF F000`	`D020 1000`	stack page (in user space map)

Table 4.1: Example memory snapshot after launching `ls`

Every physical page is either free, or mapped to a page object. This mapping is not persisted. The memory manager keeps track of free memory pages and memory page ⇔ page object mappings.

4.2 Memory Interface

A memory capability can be used to map a page object id. This makes the page object available to a running process, although it does not necessarily exist in physical memory yet. When a process is launched, four page object ranges are mapped: the code segment (read-only, executable), the text segment (read-only), the data segment (read/write) and the stack segment (read/write).

If the memory server already knows about a read-only page object id, the map is not updated.

A stack segment page object id is always new, and is therefore always recorded in the memory manager.

A data segment page object may be initialised or uninitialised.

If an initialised read/write page object id is mapped, and the memory manager already has the page object mapped read-only, the page object is mapped to the read-only version. If it is later written to, a copy will be made and the page object will be re-mapped.

An uninitialised read/write page object is recorded (same as a stack segment page).

5

Persistence

The active state of the system is defined by the page object state, which is overridden by the swap state, which is overridden by the memory state.

5.1 Checkpoint Objects

A page object is a single page (where the size of the page is architecture-dependent). Object identifiers for page objects range from `ff00 0000 0000 0000` to `ffff ffff ffff ffff`. A checkpoint state contains all page objects from a particular system state.

The swap state is a copy of the page object state, updated by page writes which have occurred since the last checkpoint. A process addresses its own memory using normal virtual memory conventions; these hardware pages are mapped by the kernel to page object ids, which can index swap and page object state directly.

The memory state is a cache of the swap state.

5.2 Checkpoint Operation

Persistence is implemented by a regular checkpoint operation.

5.2.1 Pages

Memory pages are always allocated read-only. The first write to a page (if it is allowed) causes a page fault which switches the page to read/write and marks it as dirty. When a dirty page is swapped out or synchronised, it is written to swap and its page object id is recorded in a list.

5.2.2 Process

When a checkpoint begins, a new, empty, changed page object list is created. All memory pages are marked as read-only, and all dirty pages are written to swap (via synchronisation). The former changed page object list is scanned, and the changed pages are copied. A new checkpoint is created, which references the previous checkpoint, the most recent checkpoint state, and page content copies.

If a page write occurs while the checkpoint process is running, a page fault will result (because all pages are read-only at this point). If dirty pages are still being written, the faulting process will be blocked until the changed page object list scan starts.

5.2.3 Merge

From time to time, the most recent checkpoint is merged into a full page object state, which will be used for subsequent checkpoint operations.

There exists at least one consistent checkpoint state. The last operation of a checkpoint merge writes an identifier which certifies the new checkpoint state. When restoring a checkpoint, if this identifier is missing or malformed, the previous checkpoint state is used instead.

5.2.4 Pseudocode

Initialising Checkpoint

```
checkpoint:
   lock
      Checkpoint_State := Write_Dirty_Pages

      for Page of Writable_Page_List loop
         if Page.Dirty then
            Page.Write_To_Swap
         end if
         Page.Writable := False
      end loop
      Writable_Page_List.Clear
      old_list := current_list
      current_list := new Changed_Page_Object_List
      Checkpoint_State := Copy_Changed_Swap
   end lock
```

Copying Changed Blocks

```
for Page_Id of Changed_Page_Object_List loop
   Target_Block := Next_Changed_Page_Block
   Copy_Block (Active_Swap (Page_Id), Target_Block)
   Append_Checkpoint_Reference (Page_Id, Target_Block)
end loop
```

```
Commit_Checkpoint (Change_Page_Object_list)
```

Merging Checkpoint

```
Copy_Current_Page_Object_State
for Checkpoint of Incremental_Checkpoint_List
   for Page_Id of Checkpoint.Page_Id_List
      Copy_Block (Page_Id.Checkpoint_Block,
                  New_Page_Object_State.Block (Page_Id))
   end loop
end loop

Commit_And_Activate (New_page_Object_State)
```

5.3 Restore

To restore from a checkpoint, the most recent consistent checkpoint state is copied to swap. Then, for each checkpoint based on this state, the previous checkpoint is recursively applied, followed by each changed page object. Any required pages can be loaded from swap, and the system will proceed normally.

5.4 Drive Layout

5.4.1 Partition Example

This is an example of how to partition a 64G disk.

Partition	Size	Description
Boot	100MB	Grub, kernel image, boot modules
Swap	16G	Active swap partition
State	16G	Last checkpointed state
Change	16G	Changes since last full checkpoint
Incremental	12G	Changes since last incremental checkpoint

Table 5.1: Example 100G disk layout

5.4.2 Boot partition

This partition contains the boot sector, grub, a kernel image and enough modules to launch a minimal system. The init module looks for a checkpoint, and if found, restores from it. Otherwise, the installation process is started.

5.4.3 Swap and checkpoint state partition

These partitions are formatted identically. Each block in the partition corresponds to exactly one page object; in particular, the block number or'd with the page object type specifier.

5.4.4 Change Partition

The change partition is divided into a header block, a range of change file blocks, and a range of changed page object blocks.

The change partition is cleared after a checkpoint merge.

Header Block

Offset	Description
`0000`	Magic number
`0004`	Reserved
`0008`	Checkpoint Identity
`0010`	Checkpoint Start Timestamp
`0018`	Checkpoint Finish Timestamp
`0020`	First checkpoint block
`0028`	Checkpoint block count
`0030`	Working Checkpoint Identity
`0038`	Working Checkpoint Start
`0040`	Working checkpoint first block
`0048`	Next free checkpoint block
`0050`	Next free page block

Table 5.2: Change Partition Header

Checkpoint Block (4K)

Offset	Description
`0000`	Magic number
`0004`	Reserved
`0008`	Checkpoint Identity
`0010`	Next checkpoint block
`0018`	Reserved
`0020`	Page Object Id 1
`0028`	Page Object Block 1
`0030`	Page Object Id 2
`0038`	Page Object Block 2
`....`	
`1FE0`	Page Object Id 509
`1FE8`	Page Object Block 509
`1FF0`	Checkpoint Identity
`1FF8`	Magic number

Table 5.3: Checkpoint block (4K)

6

Interfaces

6.1 Standard Interfaces

6.1.1 Rose

The *Rose* interface provides operations common to all other interfaces. It is implicitly inherited by every interface.

Operations

- **Destroy** Destroys the object associated with the capability.

6.1.2 Rose.Capability

Operations on capabilities.

Inherits

Rose

Operations

- **Create Entry** Manufactures a capability which can call the given endpoint

```
function Create_Entry
  (Endpoint : Rose.Objects.Endpoint_Id)
  return Capability;
```

- **Create Endpoint** Creates a new endpoint and returns its identity.

```
function Create_Endpoint
  return Endpoint_Id;
```

- **Receive Any** Respond to an invocation of any known capability.

```
procedure Receive_Any
  (Called_Endpoint : out Endpoint_Id;
   Reply_Cap       : Capability;
   Parameters      : Invocation_Record);
```

- **Create Set** Manufactures a capability which represents a set of capabilities.

```
function Create_Set
  return interface Rose.Capability_Set;
```

6.1.3 Rose.Capability_Set

Operations on capability sets.

Inherits

Rose

Operations

- **Insert** Adds a capability to the set

```
function Insert
  (Cap : Capability);
```

- **Delete** Delete a capability from the set.

```
procedure Delete
  (Cap : Capability);
```

6.1.4 Rose.Streams

Operations on streams.

Inherits

Rose

6.1.5 Rose.Streams.Read

Operations for reading from streams

Inherits

Rose.Streams

Operations

- **Read** Reads a buffer from the stream

```
procedure Read
  (Buffer : out Storage_Array;
   Length : in out Storage_Count)
```

6.1.6 Rose.Streams.Write

Operations for writing to streams

Inherits

Rose.Streams

Operations

- **Write** Writes a buffer to the stream

```
procedure Write
  (Buffer : Storage_Array;
   Length : Storage_Count)
```

6.1.7 Rose.Launch

Launching processes.

Operations

- **Launch** Launches the process named by this capability. Sent caps are copied to the process. Returns a capability which implements the Processes interface for this process.

```
function Launch
  (Caps : Array_Of_Capabilities)
  return interface Rose.Process;
```

6.1.8 Rose.Process

Interface to processes.

Operations

- **Launch** Launches the process named by this capability. Sent caps are copied to the process.

```
function Launch
  (Caps : Array_Of_Capabilities)
  return Capability;
```

6.1.9 Rose.System.Create

Interface for creating capabilities.

Operations

- **Create Capability** Manufactures a capability based on the representation given in the sent words.

```
function Create_Capability
  (Header : Word_32;
   Payload_1, Payload_2, Payload_3 : Word_32)
  return Capability;
```

Derived Operations

- **Create Interface Capability** Manufactures an interface capability for the given endpoint and object.

```
function Create_Interface_Capability
  (Endpoint : Endpoint_Id;
   Object   : Object_Id)
  return Capability
is (Create_Capability
     (16#0000_0005#, Endpoint,
      Low_Word (Object), High_Word (Object0)));
```

6.1.10 Rose.System.Cap_Copy

Interface for copying capabilities from other objects.

Operations

- **Copy Capability** Manufactures a capability which is a copy of an existing capability in the given object.

```
function Copy_Capability
  (Header : Word_32;
   Payload_1, Payload_2, Payload_3 : Word_32)
  return Capability;
```

6.1.11 Rose.System.Boot

Boot module interface

Operations

- **Launch** Starts a boot module.

```
function Launch
  (Module_Index : Positive;
   Caps         : Capability_Array)
  return Object_Id;
```

6.1.12 Rose.System.Device_Memory

Interface for setting up memory-mapped devices.

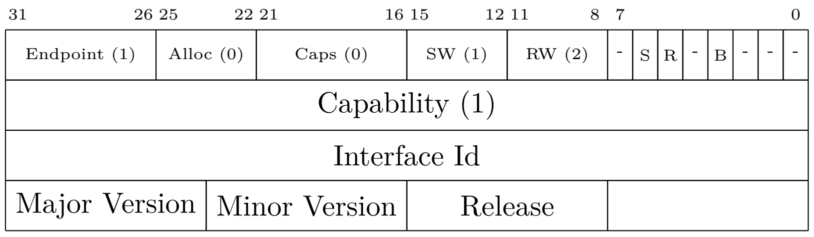

Figure 6.1: Interface capability request

Operations

- **Reserve** Reserves an area of memory for devices. Pages within this area can be acquired by device drivers using a page object cap.

```
procedure Reserve
  (Base  : Physical_Address;
   Bound : Physical_Address);
```

Name	Identity	Implemented By	Notes
Meta	1	All	A process usually has this as capability 1
Cap Copy	2	Kernel	Used during installation to subvert security
Kernel	3	Kernel	
Procmem	4	`mm`	Paged process memory management
Registry	5	registry	Registry of interface identifiers
Signal	6	any	Send and receive signals
Authentication	7	passwd	
Account	8	passwd	
Make Executable	9	exec	
Executable	10	ELF binaries	
Process	11	running processes	
Login	12	login	
User	13	user	

Table 6.1: Standard System Interfaces

Meta Interface

The *meta* interface is used for managing capabilities. Sending to this interface creates new capabilities, or sends existing capabilities to other objects. It can also be used to get a capability for an object's default interface.

Endpoint	Name	Operations	Description
1	Create Interface Cap	Send	Creates and returns a new interface capability for the sending object
2	Create Endpoint Cap	Send	Creates and returns a new endpoint capability for the sending object
3	Create Page Object Cap	Send	$data_0$ contains page address $data_1$ contains the readable and writable flags
5	Create Singleton Receive	Send	
6	Create Cap Set	Send	
16	Receive on provided caps	Receive	Maximum receive caps is 6
17	Receive on any known cap	Receive	
31	Exit	Send	End the sending process.

Table 6.2: The meta interface

The process memory manager interface

A server implementing the `procmem` interface can be notified when a process starts, when it ends, and when it requests access to virtual addresses which it does not currently have.

It is expected that the server will have access to a page mapper interface, which will allow it to map and unmap pages as required.

6.1.13 Authentication Interfaces

Name	Identity	Implemented By	Notes
Login	12	login	
User	13	user	

Table 6.3: Standard Authentication Interfaces

6.1.14 File Interfaces

Name	Identity	Implemented By	Notes
File System	100	rfs, xmlfs, isofs	
Directory	101	rfs, xmlfs, isofs	
File	102		

Table 6.4: Standard File Interfaces

6.2 Interface Definition Language

7

Servers

7.1 Boot Servers

When the kernel is ready to schedule its first process, it chooses the first boot module. This is the *init* program. Its job is to load the rest of the boot modules, and find the most recent consistent checkpoint from which to restore.

Init requires a particular set of launch caps (see Table 7.1). Each of these capabilities is invoked in order. Init runs with a low priority, causing it to block while each boot server initialises.

Init is configured via a Petal script (see Figure 8.1 for a small example). This script is compiled to an Ada package spec, which is with'd by the `init` source.

Cap	Name	Description
1	meta	Standard meta cap
2	launch	Cap for launching boot modules
2	copy	Cap for copying process caps
3	construct-port	Cap for constructing port caps

Table 7.1: Launch caps for *init* process

Pid	Name	Interface	Description
2	init	none	Installs system image and launches boot servers
3	console	stream-writer	writes to boot terminal
4	hd0	storage	boot hard drive
7	mem	mem	memory manager
8	checkpoint	checkpoint	checkpointing process
9	proc	proc	process manager

Table 7.2: Boot Servers

7.1.1 Memory Manager

7.1.2 Process Manager

8

Boot Sequence

8.1 Overview

The boot process initialises the kernel and starts the first process, `init`.

8.2 Initial Processes

8.2.1 `init`

The kernel gives the init process a single `create` capability, which is used to create launch and copy capabilities for the boot modules.

8.2.2 console

The console program implements a stream writer interface, and sends anything it receives to its device. For rose-x86 it writes directly to the console buffer, which is mapped by the kernel at boot time. Normally this is the first server launched by `init`.

8.2.3 cap set

The cap set process allows other processes to create a set of capabilities, which can be referenced using a single capability. Once this process is running, the remaining boot processes are not limited to four capabilities each.

8.2.4 timer

The kernel provides a single timeout, so there is a timer server which multiplexes multiple timers onto it.

```
procedure Init
  (Create    : interface Meta)
is
   Console : constant interface Process :=
     Launch.Launch_Boot_Module (2);
   Writer  : constant interface Stream_Writer :=
     Cap_Copy.Copy_Cap (Console, 1);
begin
   Output_Buffer := Start_Init;
   Writer.Write (Output_Buffer, Start_Init'Length);
end Init;
```

Figure 8.1: Sample Petal script for init

8.2.5 mem

The standard memory manager is launched with a memory capability, which can be queried for the available physical memory layout, and invoked to map and unmap virtual addresses to physical ones. It implements the `memory manager` interface, which can add and remove processes, and handle page faults.

8.2.6 pci

Scans the PCI bus.

8.2.7 ata

Drive for ATA/ATAPI storage.

8.3 First Boot

8.4 Restore Boot

8.5 Scripts

The boot process is controlled by a compiled Petal script. A sample script is shown in figure 8.1.

9

Petal

9.1 Generating init Scripts

A subset of Petal can be used to generate an Ada package suitable for use as an init script. The Petal source is processed by an Aquarius plugin.

```
$ aquarius -f -i init.petal -a init
```

Figure 9.1: Command for generating init script package

```
procedure Init
  (Meta_Cap    : Capability;
   Launch_Cap  : Capability;
   Copy_Cap    : Capability;
   Create_Cap  : Capability)
is
   Console_Ready       : constant String := "init: console ready" & NL;
   Memory_Buffer       : Page;
   Page_Object_Cap     : Capability;
   Console_Process_Cap : Capability;
   Console_Memory_Cap  : Capability;
   Console_Write_Cap   : Capability;
begin
   Page_Object_Cap :=
     Meta_Cap.Endpoint (3).Invoke
       (Memory_Buffer'Address, Memory_Buffer'Storage_Size);
   Console_Memory_Cap :=
     Create_Cap (Driver_Page_Capability, 16#000B_8000#);
   Console_Process_Cap :=
     Launch_Cap (Meta_Cap, Console_Memory_Cap);
   Console_Write_Cap :=
     Copy_Cap (Console_Process_Cap, 1);
   Memory_Buffer.Write (Console_Ready);
   Console_Write_Cap (Page_Object_Cap, Console_Ready'Length);
end Init;
```

Figure 9.2: Sample init Petal source

Appendix A

Architecture Specifics

A.1 i686

A.1.1 Invocation Record

The first four words of an invocation are placed in registers. The remaining words, if any, are pushed onto the stack.

Register	Parameter
`eax`	Control word
`edx`	Used by *sysenter* instruction
`ebx`	Invoked capability
`ecx`	Used by *sysenter* instruction
`esp`	
`ebp`	
`esi`	First argument
`edi`	Second argument

Table A.1: i686 invocation record

A.1.2 Page Table

Uses 4K pages. Per-process virtual memory is 3G. Page object ids are 31 bits long, allowing up to 2^{43} bytes or 8TB of state.

A.1.3 Persistence

A.2 x64