Gemini Nucleus Internals

Intel 32-Bit Architecture

Kasper Verdich Lund *verdich@users.sourceforge.net*
Guido de Jong *guidoj@users.sourceforge.net*

February 10, 2001

Contents

Part I

Architectural Overview

Chapter 1

Introduction

This document describes the internals of the IA-32 implementation of the Gemini Nucleus. It's intended as a low-level application programmer's reference, but it also provides information useful for people who are interested in the Gemini Nucleus architecture, and for those who want to port the Gemini Nucleus to other hardware platforms. The document is structured into two parts: the architectural overview and the interface description. The first part (the one you're reading right now) describes the history, philosophy, and design behind the Gemini Nucleus architecture, whereas the second part presents the Gemini Nucleus interface, including the application programming interface (API).

Chapter 2

History

The Gemini Nucleus evolved from two distinct – yet similar – operating system projects: the Apostle Project and the Elysium Project. The Apostle Project was a project to create an Open-Source operating system based on a micro-kernel. Most of the design ideas were borrowed from L4 (J. Liedkte et al), although several extensions, such as scheduler activations (T. Anderson et al), were proposed and implemented. In the words of the project initiator, J. Spencer Seidel, the goal of the project was to create an operating system, which had the following characteristics:

- Modular micro-kernel-based architecture.
- Simple device driver model.
- Networking and multi-media support.
- Intuitive, native graphical user interface.
- High run-time configurability.
- Fast and easy installation.

When the Apostle Project merged with the Elysium Project, and formed a project to create the Gemini Nucleus only the two top items in the above list had been considered. There existed a prototype kernel implementation, which supported most of the required functionality, and a model for device drivers was in the works. As an experiment the Apostle Project gave valuable input on micro-kernels and modularity to the Gemini Nucleus design process.

The Elysium Project – originally conceived by Kasper Verdich Lund – set out to design and implement an operating system based on the exokernel ideas

as proposed by D. R. Engler et al. The goal was to create an Open-Source exokernel implementation for the Intel 32-bit hardware architecture, and to empirically prove that exokernels tend to be easier to implement than traditional operating system kernels. One of the early design decisions was to structure the exokernel in components (actually separate processes) each of which supplied functionality for multiplexing different hardware resources. The most basic hardware resources, such as physical memory, input/output ports, and processing units, were multiplexed by a component called the Elysium kernel, and in a sense the Elysium operating system was a micro-kernel-based exokernel operating system.

The Gemini Nucleus builds upon the lessons learned from the two projects, and especially that although exokernels are easy to implement the design behind requires much thought, and many iterations, to get right.

2.1 L4

<<overview of the L4 project>>

2.2 Xok/ExoPC

<<overview of the Xok/ExoPC project>>

Chapter 3

Design Principles

The Gemini Nucleus is an Open-Source attempt to create a production quality operating system nucleus. This section outlines a number of design principles, which have all been applied to the current design. The principles have been – and will continue to be – used as criteria to evaluate design proposals.

Do not enforce abstractions not necessary for protection. The nucleus should avoid enforcing any abstractions, that can safely be implemented in unprivileged libraries.

Provide only a minimal set of primitives. The determining criterion is functionality, not performance. A concept is tolerated inside the nucleus only if moving it outside, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality.

<<more principles to come>>

3.1 L4 Principles

The following description of the L4 μ-kernel principles is taken from "The Performance of μ-Kernel-Based Systems" by J. Liedtke et al (pages 2-3). Although the L4 principles have been the basis for the Apostle Project, many of them are not implemented inside the Gemini Nucleus. Rather, they can easily be implemented in a library operating system on top of the Gemini Nucleus.

L4 Essentials The L4 μ-kernel is based on two basic concepts, threads and address spaces. A thread is an activity executing inside an address space. Cross-address-space communication, also called inter-process communication (IPC), is one of the most fundamental μ-kernel mechanisms. Other forms of communication, such as remote procedure call (RPC) and controlled thread migration between address spaces, can be constructed from the IPC primitive.

A basic idea of L4 is to support recursive construction of address spaces by user-level servers outside the kernel. The initial address space S_0 essentially represents the physical memory. Further address spaces can be constructed by granting, mapping and unmapping flexpages, logical pages of size $2n$, ranging from one physical page up to a complete address space. The owner of an address space can grant or map any of its pages to another address space, provided the recipient agrees. Afterwards, the page can be accessed in both address spaces. The owner can also unmap any of its pages from all other address spaces that received the page directly or indirectly from the unmapper. The three basic operations are secure since they work on virtual pages, not on physical page frames. So the invoker can only map and unmap pages that have already been mapped into its own address space.

All address spaces are thus constructed and maintained by user-level servers, also called pagers; only the grant, map and unmap operations are implemented inside the kernel. Whenever a page fault occurs, the μ-kernel propagates it via IPC to the pager currently associated with the faulting thread. The threads can dynamically associate individual pagers with themselves. This operation specifies to which user-level pager the μ-kernel should send the page fault IPC. The semantics of a page fault is completely defined by the interaction of user thread and pager. Since the bottom-level pagers in the resulting address-space hierarchy are in fact main-memory managers, this scheme enables a variety of memory-management policies to be implemented on top of the μ-kernel.

I/O ports are treated as parts of address spaces so that they can be mapped and unmapped in the same manner as memory pages. Hardware interrupts are handled as IPC. The μ-kernel transforms an incoming interrupt into a message to the associated thread. This is the basis for implementing all device drivers as user-level servers outside the kernel. In contrast to interrupts, exceptions and traps are synchronous to the raising thread. The kernel simply mirrors them to the user level. On the Pentium processor, L4 multiplexes the processor's exception handling mechanism per thread: an exception pushes instruction pointer and flags on the thread's user-level stack and invokes the thread's (user-level) exception or trap handler.

A Pentium-specific feature of L4 is the small-address-space optimization. Whenever the currently-used portion of an address space is "small", 4 MB up to 512MB, this logical space can be physically shared through all page tables and protected by Pentium's segment mechanism. This simulates a tagged TLB for context switching to and from small address spaces. Since the virtual address space is limited, the total size of all small spaces is also limited to 512 MB by default. The described mechanism is solely

used for optimization and does not change the functionality of the system. As soon as a thread accesses data outside its current small space, the kernel automatically switches it back to the normal 3 GB space model. Within a single task, some threads might use the normal large space while others operate on the corresponding small space.

3.2 Exokernel Principles

The following description of the exokernel principles is taken from "The Exokernel Operating System Architecture" by D. R. Engler (pages 21-22). The term libOS is an abbreviation for library operating system.

Separate protection from management. Exokernels restrict resource management to functions necessary for protection: allocation, revocation, sharing, and the tracking of ownership. Giving applications control over all non-protection mechanisms and policies makes the system "optimally" extensible, in that any further application customization would compromise system integrity.

In general, an exokernel strives to provide applications access to all operations that a privileged OS has. This requires providing ways for libOSes to recapture aspects of the kernel's privileged execution context, such as being tightly coupled to hardware interrupts, which they have lost by running as application software. (From another perspective, the privileged exokernel adds a layer of indirection between a libOS and hardware events. The exokernel must take steps to ensure that this layer does not preclude flexible application control of hardware.)

Expose hardware. Exokernels give applications protected access to all resources. Applications have access to privileged instructions, hardware DMA capabilities, and machine resources. The resources exported are those provided by the underlying hardware: physical memory, the CPU, disk memory, translation look-aside buffer (TLB), and addressing context identifiers. Applications have full view of resource attributes, such as the number, format, and current set of TLB mappings or the number and size of incoming and outgoing network buffers, as well as the ability to modify them (inserting TLB entries). This principle extends to less tangible machine resources such as interrupts, exceptions, and cross-domain calls.

An exokernel exposes resources and events at the lowest possible level required for protection—ideally, at the level of hardware (disk blocks, physical pages, etc.) rather than rather than [sic] encapsulating them in high-level abstractions such as files, or Unix signal or RPC semantics.

The following two principles fall from a general pattern: involve applications in important decisions rather than subjecting them to hard wired policy.

Expose allocation. Applications allocate resources explicitly. The kernel allows specific resources to be requested during allocation, giving applications control over abstraction semantics and performance decisions, such as when and which disk blocks to allocate.

Expose revocation. Exokernels expose revocation policies to applications. They let applications choose which instance of a resource to give up. Each application has control over its set of physical resources. Exokernels allow applications to revoke resources from processes they control, thereby allowing enforcement of local policies.

Protect fine-grained units. To make resource management and sharing lightweight, exokernel protection is fine-grained (e.g., at the level of disk blocks rather than partitions). Fine-grained protection reduces the overhead of mapping high-level abstractions down to the discrete resource units that an exokernel protects (e.g., mapping files to disk blocks, virtual memory to pages, network messages to message buffers, etc.). Coarse-grained protection wastes space as well as time. For example, in the context of disk, by increasing the length of disk arm sinks after increasing internal fragmentation through coarse-grained allocation. Coarse-grained protection also makes sharing clumsy, since resources cannot be shared at a finer-granularity than the exokernel protects.

Expose names. Exokernels use physical names wherever possible. Physical names capture useful information and do not require potentially costly and race-prone translations from virtual names. For example, physical disk blocks encode position, one of the most important variables for file system performance. Virtual names, in contrast, would not necessarily encode this information and would require on-disk translation tables. Using these tables increases disk accesses (extra writes for persistence, extra reads for name translation) and protecting them across reboots degrades performance (e.g., by ordering writes to disk, and causing multiple writes).

In practice, physical names also provide a uniform mechanism for optimistic synchronization. They allow locks to be replaced with checks that "expected" conditions hold (that a directory name maps to a specific disk block and has not been deleted or moved, etc.).

Expose information. Exokernels expose system information, and collect data that applications cannot easily derive locally. For example, applications can determine how many hardware network buffers there are or which pages cache file blocks. An exokernel might also record an approximate least-recently-used ordering of all physical pages, something individual applications cannot do without global information. Additionally, exokernel export bookkeeping datastructures such as free lists, disk arm positions, and cached TLB entries so that applications can tailor their allocation requests to available resources.

We have found it useful in practice to provide space for protected application-specific data in kernel data structures. For example, the structures describing an execution environment are improved if application-specific data can be associated with them (e.g., to record the numeric id of an [sic] process' parent, its running status, program name, etc.).

These principles apply not just to the kernel, but to any component of an exokernel system. Privileged servers should provide an interface boiled down to just what is required for protection.

3.3 EROS principles

The following simplified description of the EROS capability system is taken from "What *is* a Capability anyway" by J. S. Shapiro.

2. What a Capability Is

The term *capability* was introduced by Dennis and Van Horn in 1966 in a paper entitled *Programming Semantics for Multiprogrammed Computations*. The basic idea is this: suppose we design a computer system so that in order to access an object, a program must have a special token. This token **designates an object and gives the program the authority to perform a specific set of actions (such as reading or writing) on that object**. Such a token is known as a *capability*.

A capability is a lot like the keys on your key ring. As an example, consider your car key. It works on a specific car (it designates a particular object), and anyone holding the key can perform certain actions (locking or unlocking the car, starting the car, opening the glove compartment). You can hand your car key to me, after which I can open, lock, or start the car, but only on your car. Holding your car key won't let me test drive my neighbor's Lamborghini (which is just as well – I would undoubtedly wrap it around a tree somewhere). Note that the car key doesn't know that it's me starting the car; it's sufficient that I possess the key. In the same way, **capabilities do not care who uses them**.

Car keys sometimes come in several variations. Two common ones are the valet key (starts, locks, and unlocks the car, but not the glove compartment) or the door key (locks/unlocks the car, but won't start it). In exactly this way, **two capabilities can designate the same object** (such as the car) **but authorize different sets of actions**. One program might hold a read-only capability to a file while another holds a read-write capability to the same file.

As with keys, you can give me a capability to a box full of other capabilities.

Capabilities can be delegated. If you give your car key to me, you are trusting me not to hand it to somebody else. If you don't want trust me, you shouldn't hand me the key.

Capabilities can be copied. If you give me your car key, there is nothing to stop me from going down to my local car dealer and having a duplicate key made. In practice, this isn't much of a problem, because you wouldn't have handed me the key if you didn't trust me. If it comes down to desperate measures, you can change the locks on the car, making all of the keys useless. This can be done with capabilities too; it is known as severing an object, which has the effect of rescinding all capabilities. A rescinded capability conveys no authority to do anything at all.

In fact, there are only a few ways that capabilities and ordinary keys are different. The important differences are:

- Capabilities are easier to copy. No locksmith is required. Rather than hand over an original capability, the more common action is to hand over a copy of it.
- Capabilities are harder to bypass. I can break your window and hot-wire your car. Breaking into a capability system is a lot harder.
- Capabilities can have more variants. Creating a new kind of capability (for example: one that lets me inquire about the size of a file, but not read it) is simply a matter of adding a little bit of software.

Key Points: Capabilities are simple and familiar. You use them every day, and they don't surprise you very often. If you think about ordinary keys and the sorts of access controls they provide you will not go far wrong.

In order to be useful, capabilities must be unforgeable. If you could just conjure up a key to any car you wanted, they wouldn't provide much protection. Protecting capabilities from forgery can be handled in either hardware or in system software. The software approach is more convenient because it can run on an ordinary PC. EROS, which is currently the fastest capability system in existence, does this in system software, and the data suggests that there wouldn't be any real benefit to doing it in hardware.

3. Capability-Based Computer Systems

In a capability-based computer system, all access to objects is done through capabilities, and capabilities provide the only means of accessing objects. In such a system, every program holds a set of capabilities. If program A holds a capability to talk to program B, then the two programs can grant capabilities to each other. In most capability systems, a program can hold an infinite number of capabilities. Such systems have tended to be slow. A better design allows each program to hold a fixed (and small – like 16 or 32) number of capabilities, and provides a means for storing additional capabilities if they are needed. The only way to obtain capabilities is to have them granted to you as a result of some communication.

Holding a large number of capabilities is usually foolish. The goal is to make the set of capabilities held by each program as specific and as small as possible, because a program cannot abuse authority it does not have. This is known as the principle of least privilege.

In this kind of system, a program that wants to perform an operation on an object must hold a capability to that object. To perform the action, it **invokes** the capability and names the action that is to be performed. In the UNIX^{TM} operating system, for example, the system call `read(fd, buf, sz)` system call can be thought of as performing the `read` operation on the file named by `fd` (the capability), passing the arguments `buf` and `sz`. Aside from the fact that UNIX file descriptors carry some associated information about the current file offset, a UNIX file descriptor is essentially a capability.

Chapter 4

Implementation Issues

4.1 Hardware Tables

4.1.1 GDT

The number of entries in the GDT is kept down to a minimum of 8:

- Null
- Kernel CS
- Kernel SS
- User CS
- User DS
- System TSS
- NMI TSS
- Double Fault TSS

Special TSS entries for the exceptions are necessary in case of SMP systems and do not hurt otherwise.

4.1.2 IDT

The first 32 entries in the IDT (0x00-0x1f) are reserved for exceptions. The exceptions are catched and handled, if only to a bare minimum. On a page fault, changes in kernel V-space (small address spaces) are propagated first. The next 32 entries (0x20-0x3f) are reserved for IRQ's. All other entries (0x40-0xff) are free, to be used for system calls.

[comment (verdich): On a page fault any changes in kernel V-space are propagated to the faulting address space. The kernel V-space has nothing to do with the small address spaces. We need to describe the memory layout – P-space, V-space, small address spaces, etc. – somewhere.]

4.2 Processes

[comment (verdich): A process is a logical entity used exclusively for communicating, that is if you want to send information between two threads you send it the source thread sends it to the destination process. A process has a fixed number of threads (something like 128/256), and it is the threads that are scheduled. Therefore each thread needs to have prologue and epilogue upcall point, but since communication is done on the process level we only need to have a PCT upcall point per process. I suggest we have a PCT thread per process. In this way we can use the prologue upcall for the PCT thread as the process PCT upcall point. Each thread needs an address space, but an address space can be shared between any number of threads - even between threads owned by different processes. This is probably only useful in very special situations, but I see no reason why not to allow it.]

4.2.1 Process Structure

All processes are stored in a table of a fixed size (256). Although this might seem a rigid approach, it does not require dynamic memory management and is thus easier to implement. Each process has its own address space, which is implemented as separate page directories (CR3 register). Each process also has its own I/O bitmap . A process has three upcall points:

- prologue
- epilogue

- PCT

The prologue is called just before the process is scheduled. It allows the process to setup its environment. Just before the process is revoked, the epilogue is called, allowing the process to save the current enviroment. A process can call on another process through a Protected Control Transfer (PCT). At user-level this can be used to built mechanisms for Inter-Process Communication (IPC). A PCT is also used for handling events - like exceptions or hardware interrupts - at user-level.

Issue: How to implement an event? Note that it must be possible to do PCTs to several processes at once (broadcasting) and some processes might be blocked (not scheduled) and must be woken up.

4.2.2 Scheduling

For scheduling processes a so-called quantum vector is used. The vector has a fixed size (128). If a quantum in the vector is allocated, it refers to a process, which in turn can be scheduled and run on a CPU. Blocked processes are not present in the quantum vector and will therefore not be scheduled. A process can allocate any number of quantums, provided that enough free quantums are available. Hereby a minimum amount of processing time can be guaranteed. If a process does not require the rest of the quantum, it will yield in favor of other processes.

[comment (verdich): Again I suggest scheduling threads. Even further I propose that the notion of blocked process is abandoned. A process is blocked if it hasn't got any threads scheduled. Maybe we need to be able to mark a quantum as inactive ... I don't know.]

When the next process is scheduled, the I/O bitmap of the process is mapped onto the I/O bitmap indictated in the System TSS. The CR3 register is loaded with the page directory of the process, so that the process will run in its own address space.

4.3 Capabilities

4.3.1 Capability Structure

A capabilty is a key to an object, indicated by an object ID. It is vital that the object ID does not hide information. For instance page 7 should have object

ID 7. A capability also refers to an object type, so that it is possible to distinguish between page 7 and IRQ 7. The object is owned by a process, which is stored with the capability. The access rights field (a bit vector) shows what the process, holding the capability, is allowed to do with the object.

The capabilities are stored in a capability list (C-list), which is in effect a vector, the size of one or more pages. The C-list is stored at a special page directory entry, that is the same for all processes. This enables one to find the C-list easily. Each capability can in turn easily be found at the appropriate offset from the C-list start. For instance capability 7 resides at 7 * sizeof(capability) from the beginning of the C-list. Each process has read-only access to its own capability list.

[comment (verdich): We need to make sure that our capability scheme works for the small address spaces as well. If we only allow capabilities to be stored under a special page directory entry the small address spaces, that only have one page directory entry, are out of luck.]

4.3.2 Capability Management

Capabilities can not be managed directly, because a process has read-only access to its capability list. Therefore 4 system calls exist for managing the capabilties:

create	A proces can create a capablity to an object, as long as it owns the object.
duplicate	A process can duplicate a capability that it holds. The access rights is a subset of the ori
grant	A process can grant a capabilty it holds to another process.
acquire	A process can acquire a capability that is granted to it by another process.

4.4 Resource Reservation

Issue: The idea is to set up the resource reservation model similar to the capability model. Details need to be worked out.

Part II

Gemini Nucleus Interface

Chapter 5

Bootstrap

<<overview of the bootstrap interface, including the MultiBoot specification>>

<<description of modules - init process, disk block server, etc.>>

Chapter 6

Kernel API

6.1 Get Info

The `Get Info` system call is a very basic system call that allows modifying entries in a page table.

`Get Info`				
nil	`eax`		`eax`	Process ID
–	`ebx`		`ebx`	PD
–	`ecx`		`ecx`	*nil*
–	`edx`	$\longrightarrow$	`edx`	–
–	`esi`		`esi`	–
–	`edi`		`edi`	–
–	`ebp`		`ebp`	–

Table 6.1: `Get Info` Interface

Process ID The Process ID by which the current process is know inside the kernel.

Page Directory The pointer to the page directory that the current process uses.

6.2 Yield

The `Yield` system call is the most basic system call with which a process signifies that it no longer needs the remainder of its quantum.

Yield				
nil	eax		eax	*nil*
–	ebx		ebx	–
–	ecx		ecx	–
–	edx	⟶	edx	–
–	esi		esi	–
–	edi		edi	–
–	ebp		ebp	–

Table 6.2: `Yield` Interface

6.3 Set Upcalls

The `Set Upcalls` system call is a very basic system call that allows modifying the upcall points to a process.

Set Upcalls				
Prologue	eax		eax	*nil*
Epilogue	ebx		ebx	–
nil	ecx		ecx	–
–	edx	⟶	edx	–
–	esi		esi	–
–	edi		edi	–
–	ebp		ebp	–

Table 6.3: `PDE Set` Interface

Prologue The new starting address of the prologue entry point of the current process.

Epilogue The new starting address of the epilogue entry point of the current process.

6.4 PDE Set

The `PDE Set` system call is a very basic system call that allows modifying entries in a page directory.

Page Directory Capability The capability to the page directory that is to be modified.

Index An index into the page directory, specifying the entry to be modified.

PDE Set				
Page Directory Capability	eax		eax	Error code
Index	ebx		ebx	*nil*
Page Table Capability	ecx		ecx	–
Access Rights	edx	⟶	edx	–
nil	esi		esi	–
–	edi		edi	–
–	ebp		ebp	–

Table 6.4: `PDE Set` Interface

Page Table Capability A capability to the new page table to be stored at the specified index. If the capability is a NULL capability, a free page is allocated for the page table.

Access Rights Access Rights to the new page. These Access Rights are 'anded' with those specified by the page capability.

Error Codes

- `E_NONE` - No errors
- `E_INVALID_CAP_TYPE` - Not a capability to a page
- `E_NO_FREE_PAGE` - No free page available

6.5 PTE Set

The `PTE Set` system call is a very basic system call that allows modifying entries in a page table.

PTE SET				
Page Table Capability	eax		eax	Error Code
Index	ebx		ebx	*nil*
Page Capability	ecx		ecx	–
Access Rights	edx	⟶	edx	–
nil	esi		esi	–
–	edi		edi	–
–	ebp		ebp	–

Table 6.5: `PTE Set` Interface

Page Table Capability The capability to the page table that is to be modified.

Index An index into the page directory, specifying the entry to be modified.

Page Capability A capability to the new page to be stored at the specified index. If the capability is a NULL capability, a free page is allocated and mapped at the specified index.

Access Rights Access Rights to the new page. These Access Rights are 'anded' with those specified by the page capability.

Error Codes

- `E_NONE` - No errors
- `E_INVALID_CAP_TYPE` - Not a capability to a page
- `E_NO_FREE_PAGE` - No free page available

6.6 IRQ Hook

The `IRQ Hook` system call allows a process to register itself as handler of a hardware interrupt.

`IRQ Hook`				
IRQ Capability	`eax`		`eax`	Error Code
Page Directory Capability	`ebx`		`ebx`	*nil*
Handler Address	`ecx`		`ecx`	–
Stack Pointer	`edx`	$\longrightarrow$	`edx`	–
nil	`esi`		`esi`	–
–	`edi`		`edi`	–
–	`ebp`		`ebp`	–

Table 6.6: `IRQ Hook` Interface

IRQ Capability A capablity to the IRQ number the process will handle.

Page Directory Capability A capability for the page containing the page directory that the process will use when handling the hardware interrupt.

Handler Address The starting address of the IRQ handler.

Stack Pointer A pointer to the stack to be used when handling the IRQ.

Error Codes

- `E_NONE` - No errors
- `E_INVALID_CAP_TYPE` - Not a capability to an IRQ line

6.7 Set Port Access

The `Set Port Access` system call allows a process to modify its I/O port bitmap.

`Set Port Access`

Port Capability	`eax`		`eax`	Error Code
nil	`ebx`		`ebx`	*nil*
–	`ecx`		`ecx`	–
–	`edx`	⟶	`edx`	–
–	`esi`		`esi`	–
–	`edi`		`edi`	–
–	`ebp`		`ebp`	–

Table 6.7: `Register PCT` Interface

Port Capability A capability to the port which access is to be modified.

Error Codes

- `E_NONE` - No errors
- `E_INVALID_CAP_TYPE` - Not a capability to an I/O port
- `E_NO_FREE_PAGE` - No free page available

6.8 Allocate Quantum

The `Allocate Quantum` system call will try to allocate another quantum for the calling process.

Allocate Quantum				
nil	eax		eax	Error code
–	ebx		ebx	*nil*
–	ecx		ecx	–
–	edx	⟶	edx	–
	esi		esi	–
–	edi		edi	–
–	ebp		ebp	–

Table 6.8: `Allocate Quantum` Interface

Error Codes

- `E_NONE` - No errors
- `E_NO_FREE_QUANTUM` - No free quantum available

6.9 Free Quantum

The `Free Quantum` system call will free the current quantum of the calling process.

Free Quantum				
nil	eax		eax	Error code
–	ebx		ebx	*nil*
–	ecx		ecx	–
–	edx	⟶	edx	–
	esi		esi	–
–	edi		edi	–
–	ebp		ebp	–

Table 6.9: `Free Quantum` Interface

Error Codes

- `E_NONE` - No errors

6.10 Execute

The `Execute` system call allows to run a loaded ELF executable object as a new process. A capability to the new process will be stored in the capability list of the calling process.

Execute				
Start Address	eax		eax	PID/Error code
Process Capability	ebx		ebx	*nil*
nil	ecx		ecx	–
–	edx	⟶	edx	–
	esi		esi	–
–	edi		edi	–
–	ebp		ebp	–

Table 6.10: `Execute` Interface

Start Address The start address is the entry point to the ELF executable object.

Process Capability An index into the capability list of the calling process, which specifies where the nucleus should store the capability to the newly created process. If the capability list already contains a capability at the specified index it is overwritten by the new one.

Error Codes

- `E_NONE` - No errors
- `E_NO_FREE_PROCESS` - No free process available
- `E_NO_FREE_PAGE` - No free page available

6.11 Exit

The `Exit` system call terminates the current process and free all owned resources.

Exit				
nil	eax		eax	*nil*
–	ebx		ebx	–
–	ecx		ecx	–
–	edx	⟶	edx	–
	esi		esi	–
–	edi		edi	–
–	ebp		ebp	–

Table 6.11: `Exit` Interface

6.12 Kill

The `Kill` system call terminates a process and free all owned resources.

`Kill`				
Process Capability	`eax`		`eax`	Error code
nil	`ebx`		`ebx`	*nil*
–	`ecx`		`ecx`	–
–	`edx`	$\longrightarrow$	`edx`	–
	`esi`		`esi`	–
–	`edi`		`edi`	–
–	`ebp`		`ebp`	–

Table 6.12: `Kill` Interface

Process Capability Index of the capability to the process that is to be terminated.

Error Codes

- `E_NONE` - No errors

6.13 Wait

The `Wait` system call blocks the current process. The process can be woken up by a PCT.

`Wait`				
nil	`eax`		`eax`	*nil*
–	`ebx`		`ebx`	–
–	`ecx`		`ecx`	–
–	`edx`	$\longrightarrow$	`edx`	–
	`esi`		`esi`	–
–	`edi`		`edi`	–
–	`ebp`		`ebp`	–

Table 6.13: `Wait` Interface

6.14 Register PCT

The `Register PCT` system call allows a process to register a PCT entry point.

Register PCT				
Entry	eax		eax	*nil*
nil	ebx		ebx	–
–	ecx		ecx	–
–	edx	⟶	edx	–
–	esi		esi	–
–	edi		edi	–
–	ebp		ebp	–

Table 6.14: `Register PCT` Interface

Entry The PCT entry point of the current process.

6.15 PCT

The `PCT` system call does a protected control transfer to another process.

PCT				
Process ID	eax		eax	*nil*
Data 0	ebx		ebx	–
Data 1	ecx		ecx	–
Data 2	edx	⟶	edx	–
Data 3	esi		esi	–
Data 4	edi		edi	–
Data 5	ebp		ebp	–

Table 6.15: `PCT` Interface

Process ID The Process ID of the process to which the control is transfered.

Data i Some data transferred to the other process.

6.16 Create Capability

The `Create Capability` system call is a very basic system call that allows the creation of new capabilities.

Capability An index into the capability list of the calling process, which specifies where the nucleus should store the newly created capability. If the capability list already contains a capability at the specified index it is overwritten by the new one. The owner is automatically filled with the calling process ID.

Create Capability				
Capability	`eax`		`eax`	Error code
Object	`ebx`		`ebx`	*nil*
Type	`ecx`		`ecx`	–
Access Rights	`edx`	$\longrightarrow$	`edx`	–
nil	`esi`		`esi`	–
–	`edi`		`edi`	–
–	`ebp`		`ebp`	–

Table 6.16: `Create Capability` Interface

Object An index in the object list for this type of capability. The encoding through the object ID and type of the capability must not hide information.

Type Predefined type of the capability.

<<list of object types>>

Access Rights A set of flags defining the access rights on the object.

Error Codes

- `E_NONE` - No errors
- `E_NOT_OWNER` - The calling process is not and can not become owner of the object
- `E_NO_FREE_PAGE` - No free page available

6.17 Grant Capability

Grant Capability				
Capability	`eax`		`eax`	Error code
Process ID	`ebx`		`ebx`	*nil*
nil	`ecx`		`ecx`	–
–	`edx`	$\longrightarrow$	`edx`	–
–	`esi`		`esi`	–
–	`edi`		`edi`	–
–	`ebp`		`ebp`	–

Table 6.17: `Grant Capability` Interface

Capability An index into the capability list of the calling thread, which specifies the capability to be send.

Process capability. An index into the capability list of the calling thread that specifies a capability that allows granting capabilities to a specific process.

Process Id. An index into the thread array of the process specified by the *Process capability*. The (Process Id, Thread Id) tuple uniquely (at least on the local machine) identifies a thread.

Error Codes

- `E_NONE` - No errors
- `E_PAGE_NOT_PRESENT` - Page containing capability list not present

6.18 Acquire Capability

`Acquire Capability`				
Capability	`eax`		`eax`	Error code
Process ID	`ebx`		`ebx`	*nil*
nil	`ecx`		`ecx`	–
–	`edx`	$\longrightarrow$	`edx`	–
–	`esi`		`esi`	–
–	`edi`		`edi`	–
–	`ebp`		`ebp`	–

Table 6.18: `Acquire Capability` Interface

Capability An index into the capability list of the calling thread, which specifies the capability to be acquired.

Process capability. An index into the capability list of the calling thread that specifies a capability that allows acquiring capabilities from a specific process.

Process Id. An index into the thread array of the process specified by the *Process capability*. The (Process Id, Thread Id) tuple uniquely (at least on the local machine) identifies a thread.

Error Codes

- `E_NONE` - No errors
- `E_NO_FREE_PAGE` - No free page available
- `E_GENERAL_ERROR` - General error

6.19 Duplicate Capability

`Duplicate Capability`				
Capability	`eax`		`eax`	Error code
Duplicate	`ebx`		`ebx`	*nil*
Access Rights	`ecx`		`ecx`	–
nil	`edx`	$\longrightarrow$	`edx`	–
–	`esi`		`esi`	–
–	`edi`		`edi`	–
–	`ebp`		`ebp`	–

Table 6.19: `Duplicate Capability` Interface

Capability An index into the capability list of the calling thread, which specifies the capability to be duplicated.

Duplicate An index into the capability list of the calling thread, which specifies where the nucleus should store the duplicated capability. If the capability list already contains a capability at the specified index, it is overwritten by the duplicate.

Access Rights A set of flags defining the access rights on the object. The given access rights are 'anded' with the original access rights.

Error Codes

- `E_NONE` - No errors
- `E_PAGE_NOT_PRESENT` - Page containing capability list not present
- `E_NO_FREE_PAGE` - No free page available

6.20 Verify Capability

The `Verify Capability` system call is a very basic system call that allows the verification of capability of another process, without being able to read the actualy capability data.

`Verify Capability`				
PID	`eax`		`eax`	Error code
Capability	`ebx`		`ebx`	Object
Type	`ecx`		`ecx`	*nil*
Access Rights	`edx`	$\longrightarrow$	`edx`	–
nil	`esi`		`esi`	–
–	`edi`		`edi`	–
–	`ebp`		`ebp`	–

Table 6.20: `Verify Capability` Interface

Process ID An index into the thread array of the process specified by the *Process capability*. The (Process Id, Thread Id) tuple uniquely (at least on the local machine) identifies a thread.

Capability An index into the capability list of the calling process, which specifies where the nucleus should store the newly created capability. If the capability list already contains a capability at the specified index it is overwritten by the new one. The owner is automatically filled with the calling process ID.

Type Predefined type of the capability.

<<list of object types>>

Access Rights A set of flags defining the access rights on the object.

Object An index in the object list for this type of capability. The encoding through the object ID and type of the capability must not hide information.

Error Codes

- `E_NONE` - No errors
- `E_INVALID_PID` - The given Process ID does not represent a valid process.
- `E_NO_FREE_PAGE` - No free page available

6.21 Set Thread

Conceptually all processes have a fixed number of threads any number of which can be scheduled. The `Set Thread` system call is used to control the threads within a process, and it can be called by any thread that holds a capability to the process whose thread is being set.

`Set Thread`				
Process capability	`eax`		`eax`	Error code
Process Id	`ebx`		`ebx`	*nil*
Address space capability	`ecx`		`ecx`	–
Thread state offset	`edx`	$\longrightarrow$	`edx`	–
Prologue code offset	`esi`		`esi`	–
Epilogue code offset	`edi`		`edi`	–
nil	`ebp`		`ebp`	–

Table 6.21: `Set Thread` Interface

Process capability. An index into the capability list of the calling thread that specifies a capability that allows modification to a specific process.

Process Id. An index into the thread array of the process specified by the *Process capability*. The (Process Id, Thread Id) tuple uniquely (at least on the local machine) identifies a thread.

Address space capability. An index into the capability list of the calling thread that specifies a capability that allows usage of a specific address space. Using an index to a capability that specifies a small address space will allow the thread to run in a subset of the large 4 Gb address space (protected by means of segmentation), thereby eliminating expensive context-switches when transferring control to the thread.

Thread state offset. An offset to the location of the thread state in the address space specified by the *Address space capability*. The thread state is the place where a thread can communicate state information to the nucleus, without going through expensive system call interfaces. Since several threads can run in the same address space it is possible for several threads to share thread state, but the typical situation will be a unique thread state for each thread in the system.

Prologue code offset. An offset to the location of the prologue code in the address space specified by the *Address space capability*. The prologue code is the code the nucleus executes to tell a thread that it has been given a CPU time slice.

Epilogue code offset. An offset to the location of the epilogue code in the address space specified by the *Address space capability*. The epilogue code is the code the nucleus executes to notify a thread of CPU revocation.

Error Codes <<description of the error codes that `Set Thread` can return>>