Introduction to the Sartoris Microkernel Architecture

Design concepts and implementation notes for the x86 platform.

by Santiago Bazerque and Nicolás de Galarreta

for Sartoris version 2, June 28, 2008.

Abstract

The Sartoris project aim is to develop a portable policy-free microkernel that provides a simple processor abstraction, yet a powerful enough one to support a full featured operating system. Therefore, boot primitives and system calls to perform crude process creation, (shared) memory management and asynchronous message-passing are the only features directly supported by the kernel. The rest of the functionality of the operating system, including the process-manager, memory-manager, filesystems, device drivers, ... are to be implemented outside of the kernel.

Contents

Chapter 1

Microkernel design

1.1 General considerations

Any operating system designed to run modern hardware must overcome the (rather complicated) details of the management of the central processors and their memory management units. In most processor architectures, this implies using specific machine instructions and maintaining a set of data structures in memory from which the hardware can obtain the current state of the system. Furthermore, the handling of interrupt requests and the construction and maintenance of the page tables are themselves nontrivial tasks, increasing the overall complexity of the system.

Therefore, it seems reasonable to encapsulate the routines that interact directly with the processor, and that is what the Sartoris microkernel intends to do. Moreover, in order to provide a suitable abstraction to the operating system, the microkernel does not need to provide many other functions. The system calls provided by Sartoris can be grouped as follows:[1]

> **Task and thread management.** These system calls cover the creation of processor tasks and threads. Sartoris does not, however, load processes from disk or perform any scheduling of the CPU. These chores are to be performed by services running on top of the microkernel (usually, a process-manager task). Under Sartoris, an interrupt

[1] Every processor architecture has unique features that must be handled in the kernel (i.e. performance monitoring hardware). A small number of architecture specific system calls will provide the necessary cpu dependent functions.

handler is just a special thread that is invoked by the microkernel every time the corresponding interrupt request line is raised. Hence, the scheduling thread may be created by binding an ordinary thread to the timer interrupt.

Memory management. This subsystem presents an abstracted view of the paging mechanism of the underlying processor, as well as an implementation of inter-task memory sharing through Sartoris SMOs (shared memory objects), which are chunks of binary data that can be easily accessed from multiple tasks. Again, the microkernel does not perform any paging directly (except in exceptional cases like interprocess communication, where some temporary mappings may be created and later destroyed during the execution of a system call). It just provides support for the easy creation and modification of page tables, (mostly) independently of the processor page table format and peculiarities.

Message passing. The kernel provides an asynchronous messaging system, in the form of a set of ports assigned to each task. Each port functions as a mailbox where fixed-sized messages from other tasks are received. These system calls cover the creation, deletion and management of ports.

Before each of the subsystems is further described, the main issues considered in the design will be reviewed:

The microkernel should present a simple yet effective abstraction of the processor to the operating system. The services offered by the microkernel should be clearly defined and easy to understand.

In order to obtain effective portability, a suitable interface that encapsulates the architecture-dependent sections of the kernel code has to be defined.

Kernel services should be provided in a policy-independent way, whenever this is possible. The design of the operating system should not be over-restricted by the underlying microkernel architecture.

The design of the microkernel should allow an efficient implementation of the most common operating system functions, considering the inherent constraints that the microkernel architecture imposes to the system.

A secure execution envirnoment must be provided for the operating system.

1.2 Introduction and sources organization

A kernel may be defined as the set of functions that it implements. The full Sartoris system call set is declared in the file include/sartoris/syscall.h.

The architecture-independent part of the system calls implementation is contained in the files in the subdirectory mk. These files implement the algorithms and data structures that are independent of the platform the microkernel is being compiled for. The object files obtained from these sources have to be linked with the implementation of the low level functions for the target processor, which should be arch/target/kernel/cpu-arch.o, where target is a symlink to the desired architecture directory. The set of GNU makefiles provided with the sources takes care of the complete compile and link cycle. The low-level interface is described in section 1.7.

1.3 Tasks and threads

All the processing (even interrupt handling) in a Sartoris based system is done in the context of a task, and a thread within that task.
A task is composed by a virtual address space (defined by a set of page tables) and communication mechanisms (ports and shared memory objects). It is identified uniquely by a number between zero and MAX_TSK[2], which is the parameter address passed to the system call create_task(int address, struct task *tsk), `*tsk` points to a structure of the type

```
struct task {
    int mem_adr;
    int size;
    int priv_level;
};
```

When Paging is disabled, mem_adr indicates the physical address where

[2] A constant defined by the implementation

this task should be placed in main memory[3], but if Paging is enabled, this field will be the desired task Base Address, and it must be greater than MIN_TASK_OFFSET defined on kernel.h. Size indicates the size (in words) of the task being created, and priv_level indicates its privilege level. This number might be used by the operating system to restrict the ability to send messages to ports and to run specific threads using privilege levels. Zero is the most privileged level, and levels zero and one have a special meaning to the microkernel, because they are the only levels that can access the input/output space.

It should be noted that System Calls to create and destroy tasks, threads and interrupt handlers are restricted to tasks running in privilege level zero. The system call ret_from_int is restricted to levels zero and one.[4] Tasks are destroyed using the similar function destroy_task, which receives the address of the task being destroyed.

A thread is a processor state associated with a given task. The maximum amount of concurrent threads is determined by the constant MAX_THR. A task might have zero, one, or more threads. Threads are created using the system call create_thread(int id, struct thread *thr). The parameter id is an integer that uniquely identifies each thread, and must not exceed MAX_THR-1. The structure thread is defined as

```
struct thread {
    int task_num;
    int invoke_mode;
    int invoke_level;
    int ep;
    int stack;
};
```

where task_num is the task that defines the context in which this thread is to be created (which must have been already created), ep is this thread's entry point, stack is the initial stack value, and invoke mode is one of the following:

[3]The organization of the tasks in physical memory is therefore under absolute control of the operating system.

[4]Notice that the numerical privilege levels used by the microkerenl are independant of those defined by the underlying processor architecture. For example, threads running with mircrokerenel privlege 0 in the x86 implementation of Sartoris are running with a processor privilege level of 1, since level 0 is being used by the microkernel itself.

PERM_REQ: In this mode, it is necessary (additionally to the privilege level constraints) to have specific authorization (obtained through the function set_thread_run_perm) to run this thread.[5]

PRIV_LEVEL_ONLY: In this mode only the usual privilege level constraints are applied to the function run_thread applied to this thread[6].

DISABLED: The thread is disabled, and it can't be invoked by anyone.

The value of the field invoke_level indicates the numerically higher privilege level that can invoke this thread.

Threads are destroyed using the destroy_thread system call[7], and may be started (or resumed) by interrupt requests signaled to the processor, software generated interrupts, exceptions, traps and the already mentioned run_thread system call.

As a non-restrictive policy, the run_thread system call can be used by threads running at every privilege level, but there are restrictions regarding which threads might be invoked. The target task protection mode must not be DISABLED, the privilege level of the invoking task must be numerically lower or equal than the invoke_level of the target task, and if the protection mode of the target task is equal to PERM_REQ, the invoking task must have been authorized by the target task through the function set_thread_run_perm. A thread can modify it's invoke_mode using the set_thread_run_mode system call at any time.

1.4 Interrupt handlers

Interrupt handlers are created using the create_int_handler system call.

Upon completion of interrupt service, a nesting interrupt handler **must** invoke ret_from_int system call, to return from the interrupt. When invoked, sartoris will switch to the first thread on the interrupt stack (if any).

Non-nesting interrupts will not be pushed onto Sartoris interrupt stack and the only way out of them is invoking run_thread.

[5] Right now, on implementation for Sartoris 2, this functionallity is no longer supported. Perhaps on a future version it will be supported again.

[6] see the description of the function run_thread below

[7] It may be worth noting that the currently running thread can't destroy itself.

1.5 Messaging system

The kernel provides an asynchronous inter-task message system through the send_msg, get_msg and get_msg_count system calls. The messages have a fixed size defined by the implementation, and the kernel guarantees that messages arrive in FIFO order, but each message queue has a maximum capacity which is also implementation-defined. When a message is sent using the function int send_msg(int to_address, int port, void *msg), the contents of the message pointed by msg are copied to the queue for port number port of task to_address. The receiving of messages is done in an analogous fashion using the function int get_msg(int port, void *msg, int *id), which removes the fist message in the supplied port's queue and copies it's contents to the address msg. The variable id is used to return the id of the sender task. Both functions return 0 on success. The function int get_msg_count(int port) returns the amount of messages waiting in the queue of the supplied port. Before a port can be usde to receive messages, it must be opened using the open_port system call, setting it's protection mode (see below) and the numerically higher privilege level that can send messages to this port. When a port is closed using the close_port system call, the associated message queue is flushed and the port becomes available to be used again. The amount of simultaneous ports a task can open at any given time is defined by the constant MAX_TSK_OPEN_PORTS located in the header file /include/sartoris/message.h.

In order to provide a secure messaging system, Sartoris defines the following port protection modes:

PERM_REQ: When a port's protection mode is set to PERM_REQ, a task can send messages to this port if it has been authorized through the set_port_perm system call, and it's privilege level is numerically lower or equal than the port's privilege level.

PRIV_LEVEL_ONLY: The privilege level of the task sending the message must be numerically lower or equal than the port's privilege level.

DISABLED: If a task wishes to deny access to a port, this is the protection mode it ought to have. The protection mode for a port can be modified using the set_port_mode system call. Permissions can be granted (or removed) to individual tasks using the set_port_perm system call.

Creation and deletion of ports is done with the create_port, delete_port and delete_task_ports functions. It's important to note that the whole msg_container has to be initialized with the init_msg function before using any other function. Messages on ports are managed with the empty, enqueue and dequeue functions. All these functions are implemented in mk/message.c.

1.6 Memory management

The address space of each task is, as was mentioned earlier, defined by a set of page tables. Even though the details of the paging subsystem vary from processor to processor (i.e. levels of indirection, exact format of a page table entry, etc.), the microkernel strives to present a simple abstraction of the paging mechanism. Again, paging operations should be directed by a process (or several processes) running on top of the microkernel, and using appropriate system calls to build and destroy the page mappings. The same function, int page_in(int task, void *linear, void *physical, int level, int attrib), is used to insert page tables and individual page entries into the page table structure of a task. The parameter level is used to distinguish between the former and the latter. For example, the x86 implementation of Sartoris treats level 0 as page directory, level 1 as page table and level 2 as individual page entry. Of course, this depends upon the levels of indirection implemented by the processor. The parameter task indicates the id of the task whose page tables are to be modified, the linear and physical pointers indicate the mapping whose creation is being requested. Finally, the parameter attrib supports the following attributes:

PGATT_CACHE_DIS: the page should be marked as cache-disabled.

PGATT_WRITE_ENA: the page should be marked as read-write.

PGATT_WRITE_THR: the page should be marked as write-through (i.e. don't cache writes).

Which attributes are supported at each level depends upon the platform. Bear in mind that the page levels have to be inserted in the correct order, since the microkernel uses no intermediate structures in the construction of the page tables. For example, if the paging system has a page-directory, page-tables structure (as in the x86 processors), the directory must be inserted first, then the relevant page tables and finally the individual page

entries. Destruction of mappings functions symetrically through the system call int page_out(int task, void *linear, int level). Removing a page table will disable all its page table entries as well, and removing the page directory will disable the entire address space of the task.

The functions int flush_tlb(void) and int get_page_fault(struct page_fault *pf) flush any translation lookaside buffers the processor may have, and retrieve information about the last page fault.

kernel internals: In current implementations, the microkernel has its own set of page tables to access the low memory area where its own code and static data structures reside, that is injected into every task's page directory upon insertion. The page mangling functions described above take the appropiate measures to prevent user programs from disrupting the microkernel address space. To access user memory, the kernel uses the same page tables as the user processes do. Hence, the presence in memory of user data needed by system calls cannot be guaranteed by the microkernel. This is specially delicate when the microkernel must access a lock in order to perform the system call. Usually, critical sections are implemented in single processor kernels by disabling the interrupts. However, a page fault may well occur with interruptions disabled, thus disrupting the locking mechanism. The solution to this problem is twofold: for small structures received from user space, the structure is copied to the kernel stack before any locking is performed, and therefore page faults, if present, can be resolved without harm. For larger memory access (for example, in the implementations of the functions read_mem and write_mem), each time the microkernel must access a new page from user space, the presence of the relevant page in memory is checked and, if missing, a page fault is issued by the microkernel (not by the processor) after releasing any pertinent locks and saving the necessary information to allow the system call to continue normally and securely after the page fault has been resolved. On version 2.0 of Sartoris, dynamic memory management is introduced on the system, and while the kernel can check for mapping of its static data and code, it won't be able to check if a process accessed one of it's dynamic memory tables.

The microkernel must supply a mechanism to perform memory sharing. Therefore, each task can access a set of Shared Memory Objects which allow other tasks to access its address space in a controlled way. However, there is no memory aliasing, and access is limited to reading from and writing to the shared sections using system calls that copy memory contents from one task address space to another.

The system call int share_mem(int target_task, void *addr, int size, int perms) creates a SMO of size size words[8] at offset addr of the current task's address space, that can be accessed by the task[9] target_task. The parameter perms indicates if access is granted for reading, writing, or both. The function returns an id number that identifies the SMO just created, or -1 in case of failure. SMOs can be destroyed using the system call int claim_mem(int smo_id) and the target task of an SMO can pass it over to another task using the system call int pass_mem(int smo_id, int target_task), which changes the target task to the supplied parameter.

The system call int read_mem(int smo_id, int off, int size, void *dest) copies the size words at offset off of the SMO identified by smo_id to the address dest. Conversely, the system call int write_mem(int smo_id, int off, int size, void *src) copies size words from address src from the current task's address space to offset off of the SMO identified by smo_id. Of course, in both cases the current task must be the target task of the SMOs, and have the right permission.

All the memory-sharing system calls excepting share_mem return 0 in case of success and -1 otherwise.

kernel internals: Memory sharing functions are the only functions that need to access the address space of a task distinct from the currently executing one. This must be resolved by each platform implementation, providing functions arch_cpy_from_task and arch_cpy_to_task that perform inter-task address translation. See section 1.7 for relevant details.

1.7 Dynamic Memory

From version 2, Sartoris implements dynamic memory management. When enabled, memory for the microkernel structures will be partially allocated[10]. When a system call is issued to the microkernel, requesting the creation of a structure on micorkernel memory[11] the microkernel will rise a Page Fault interrupt. The OS, will be able to tell if the interrupt was rised because

[8] The actual unit in which size is expressed is defined by the implementation.

[9] What is meant here is that it can be accessed by any thread of the corresponding task.

[10] Initially, only a predefined ammount of space will be preallocated for each type of structure.

[11] i.e. Sending a message, creating an SMO, creating a Task/Thread or opening a port.

of a sartoris memory need, by invoking get_page_fault syscall, and testing for the value 0xFFFFFFFF on task_id member of the structure sent as a parameter. When servicing a sartoris Dynamic Memory interrupt, the OS on top of Sartoris, *must* issue a grant_page_mk syscall, passing the physical address of the page granted to Sartoris. If the physical address is NULL (0), Sartoris will consider no page is available for him, and the system call which generated the need for dynamic memory will fail.
Whenever Sartoris considers it will no longer require an allocated page, it will also rise a Page Fault interrupt, although this time the operating system will be sent a task_id of 0xFFFFFFFE upon a call to get_page_fault. The physical address of the page being freed by the microkernel, will be sent on the member linear of the structure.
While servicing a Dynamic Memory page fault (i.e. with task_id 0xFFFFFFFF or 0xFFFFFFFE) the Operating System, is free to continue its normal operation, however, when the page fault has task_id on 0xFFFFFFFF, the OS wont be allowed to run the thread which originated the need for Dynamic Memory allocation, and every call to a syscall which requires Dynamic Memory will fail.

1.8 Low-level interface

The Sartoris microkernel is divided in an architecture neutral section, which is written in the C programming language, and handles the kernel data structures, and an architecture specific section, which handles the processor data structures and its low level management. The interface that defines the low-level functions is located in include/sartoris/cpu-arch.h

```
/* This are the arch-dependent functions needed to support the kernel */

#ifndef CPUARCH
#define CPUARCH

#include <sartoris/kernel.h>

void arch_init_cpu(void);

int arch_create_task(int task_num, struct task* tsk);
int arch_destroy_task(int task_num);
```

```
int arch_create_thread(int id, struct thread* thr);
int arch_destroy_thread(int id, struct thread* thr);
int arch_run_thread(int id);

int arch_page_in(int task, void *linear, void *physical, int level, int attrib);
int arch_page_out(int task, void *linear, int level);
int arch_flush_tlb(void);

void arch_mem_cpy_words(int *src, int *dst, int len);
void arch_mem_cpy_bytes(char *src, char *dst, int len);

int arch_cpy_to_task(int task, char* src, char* dst, int len);
int arch_cpy_from_task(int task, char* src, char* dst, int len);

void arch_dump_cpu(void);

#ifndef HAVE_INL_CLI
int arch_cli(void);
#endif

#ifndef HAVE_INL_STI
void arch_sti(int x);
#endif

#ifndef HAVE_INL_GET_PAGE_FAULT
void *arch_get_page_fault(void);
#endif

#ifndef HAVE_INL_ISSUE_PAGE_FAULT
void arch_issue_page_fault(void);
#endif

#ifndef MAKE_KRN_PTR
# define MAKE_KRN_PTR(x) ((void*)x)
#endif

#ifndef MAKE_KRN_SHARED_PTR
# define MAKE_KRN_SHARED_PTR(t, x) ((void*)x)
#endif
```

```
int arch_cli(void);
void arch_sti(int);

#endif
```

The architecture specific functions are called from the proper system calls. While they may access the rest of the kernel's data, they should define their own architecture specific data structures and reserve the necessary memory regions to hold them.

Chapter 2

The x86 implementation

2.1 System organization overview

Initializing the processor and loading the microkernel and the init-process to main memory is the very fist task that the IA32 specific section of Sartoris must overcome. Once the bootstrapping has concluded, the kernel initialization function must create the system tables and execute the init task, which will start the operating system.

The main purpose of the architecture specific functions in Sartoris is the management of tasks, threads, paging and locking. In the IA32 architecture, the creation and destruction of tasks updating the descriptors in the Global Descriptor Table and one Local Descriptor Table, so that the user segments have the correct privilege levels and point to the task's memory areas. To perform the multiplexing of the CPU into several threads, the IA32 architecture defines a data structure which contains all the information required to create, suspend and resume a thread, the Task State Segment.

The IA32 kernel code is located in arch/i386/kernel/.

2.2 Bootstrapping

In the PC architecture, bootstrapping begins after the BIOS loads the first 512-byte sector of the boot drive to offset 0x7c00, and executes it in real mode. The boot sector assembly source is in arch/i386/boot/boot.s. It uses

BIOS function 0x13 to load the raw area immediately following the boot sector in the boot media to memory, extracting the kernel image and the init task image. Then it jumps to the kernel initialization routines. The boot media and the kernel size are hardcoded in the boot sector for now, so they have to be manually set.

In order to run the kernel, the boot sector must also enable the 20'th address line in the bus, which is done through the keyboard controller and change the processor executing mode to protected mode. Temporary IDT and GDT tables are set up before the switch to protected mode.

2.3 Locking

The ability to enable and disable interrupts is used to implement critical sections that protect the integrity of microkernel data structures. The locking is as fine-grained as possible, without over-complicating the implementation. Some operations, like for example page-table mangling, are completely atomic, since implementing finer locking would require contemplating a too complex scenario. As was explained in 1.5, page faults are the only traps that are allowed to occur during critical sections. Hence the special measures taken to access user data from system calls.

2.4 IA32 system tables

2.4.1 Global Descriptor Table

The GDT contains the descriptors that are shared among all the tasks in the system. Some descriptors, in particular the LDT descriptors used for tasking and the TSS descriptors used for threading, must reside in the GDT. Some other descriptors that are shared among all the tasks are also in the GDT. The variables MAX_SCA, GDT_LDTS, MAX_TSK and MAX_THR are used to statically reserve entries in the GDT for the maximum possible amount of system calls, tasks and threads. The descriptor layout in the GDT is:

descriptor group	**how many?**	**details**
system descriptors	4	dummy, kernel code, kernel data, high memory area
syscalls	MAX_SCA	call gates for the system calls
ldt segment descriptors	GDT_LDTS	ldt segment descriptors
task state	1	global TSS descriptor
LDT descriptors	MAX_TSK	descriptors for task's Local Descriptor Tables
TSS descriptors	MAX_THR	descriptors for thread's Task State Segments

The very first descriptor of the GDT is a null descriptor, required by the processor architecture[1]. The kernel sees the memory as a flat address space. The second descriptor is the kernel code descriptor, which defines a 32 bits execute-read segment with base zero and limit set to 16 megabytes (this is all the maximum amount of memory currently supported), and Descriptor Privilege Level set to zero (most privileged). The third descriptor is the kernel data descriptor, like the second one but with data-read-write type. The fourth descriptor is has a Descriptor Privilege Level of two, and is a data-read-write type descriptor which gives the service level tasks access to the high memory area[2], since numerous input output devices are configured to use buffers in this area (ie the standard VGA adapter). The next MAX_SCA entries are reserved for the system calls' call gates. A call gate allows the microkernel to offer it's services in a controlled way. Each gate has a privilege level (some calls may be accessed from service tasks only), an entry point into the kernel code, and a parameter count. When a thread performs a far call to one of these gates, control is transfered to the kernel and the right amount of parameters are copied to the kernel privilege zero stack. The following descriptors are used to hold the LDTs and the TSS, which will be discussed later in this section.

[1] Loading a zero offset into an unused segment selector register is licit if the corresponding descriptor is null; a protection fault is only generated if a memory access through the null descriptor is attempted.

[2] region between the physical addresses 0xa0000 and 0x100000.

2.4.2 Interrupt Descriptor Table

The Interrupt Descriptor Table contains the descriptors that define the processor's reaction to exceptions and external or software generated interrupt. The first 32 entries are reserved for the processor's exceptions, while the rest may be used to handle external interrupts or operating system services invoked through an int instruction.

Sartoris uses an interrupt stack. Any given thread can be *attached* to an interrupt, and can be declared as either nesting or non-nesting[3].

When an interrupt is signaled, Sartoris will perform a thread-switch[4] to the handler. If the handler was marked as nesting, it will be pushed onto the interrupt stack.

Upon completion of interrupt service, a nesting interrupt handler **must** invoke ret_from_int system call, to return from the interrupt. When invoked, sartoris will switch to the first thread on the interrupt stack (if any).

Non-nesting interrupts will not be pushed onto the interrupt stack and the only way out of them is invoking run_thread. In any case threads **cannot** use iret or ret instructions to return execution.

2.4.3 Local Descriptor Tables

Every task has it's own linear address defined by two descriptors in it's Local Descriptor Table: an execute-read type descriptor for it's code and a read-write descriptor for it's data and stacks. There are only four LDT segment descriptors on the Global Descriptor Table, when thread-switching, sartoris will modify ldt descriptors to point to the switching task ldt, using first slot for privilege 0 descriptors, second slot for privilege 1 and so on up to privilege 3. any other privileges will map to ring3 on the processor.

2.4.4 Page Tables

To define an address space in the IA32 architecture a 4 Kb page directory is needed. This page directory indicates the location in physical memory of the page table corresponding to each chunk of linear space addresses. Then each

[3]Using the nesting parameter on create_int_handler system call.
[4]Thread switching is implmented through software.

4 Kb page table is divided into 1024 4-byte entries that indicate the location in physical memory of a particular page, plus attributes (read, write, cache behavior, etc.). The microkernel has no intermediate structures for holding page mappings, it uses the page tables directly.

2.5 IA32 function implementation details

This section describes the behavior of the IA32 implementation of the architecture dependent functions.

arch_init_cpu:

PIC reprogramming. The init functions reprograms the programmable interrupt controllers so that the interrupts from the master controller go to the offsets 32-39 and interrupts from the slave controller go to offsets 40-47 of the IDT. The slave PIC is cascaded through the second interrupt request line of the master. All the interrupts are disabled though the PICs interrupt masks.

GDT set up. The dummy, kernel code, kernel data and high memory area descriptors of the GDT are created. All the other descriptors are invalidated. The Global Descriptor Table is loaded using the lgdt instruction, which using a virtual descriptor composed by a linear base and a limit located in main memory loads the GDTR register.

syscall hooking. All the task gates for the system calls are created in the corresponding GDT positions. This is done through the hook_syscall function, with the correct entry point, privilege level and parameter count for each call, as described in subsection 2.4.1.

IDT set up. The fist 32 entries of the IDT are filled with interrupt gates[5] that point to routines that will dump the cpu registers and information about the currently running task and thread and halt the machine. These handlers should be replaced by the operating system exception handling threads, but for operating system development and to show some diagnostic in case the operating system dies very early in the boot process these default handlers are useful. The rest of the

[5] An interrupt gate is very similar to a call gate, but the processor handles the interrupt enable flag differently.

IDT is full with invalid descriptors. Finally, the IDT is loaded in a way analogous to the GDT, using the lidt instruction.

Init service execution Now the cpu is ready to run the microkernel. Using the create_task and create_thread system calls, the operating system init service is created in the exact address to which it was fetched earlier by the bootstrapping code (currently, at the three megabyte mark), and executes using the reun_thread system call. This is the last action the microkernel will take on it's own initiative.

arch_create_task:

LDT set up. An execute-read segment and a read-write segment are created in the task's local descriptor table first and second descriptors, with the privilege level corresponding to the task being created and base and limit according to the corresponding syscall parameters.

arch_destroy_task:

invalidate LDT descriptor. The task's LDT descriptor is invalidated, preventing any future access to or execution from the task's address space.

arch_create_thread:

Thread State set up. Sartoris Thread State structure holds the contents of general purpose registers wich must be preserved on cdecl convention, the base and stack register, the segment selector registers, the eflags register, the LDT selector register and the instruction pointer register. In a new thread's state, all the general purpose registers are set to zero. The base and stack pointer are set to the stack value supplied by the call. The cs register is loaded with a selector for the code segment in the task's LDT with the correct Requested Privilege Level, while cs, ds and ss are loaded with the second descriptor in the LDT, the data segment. The fs and gs registers are loaded with null descriptors, but in service task's threads fs is loaded with a selector for the high memory area descriptor located in the fourth entry of the GDT. The eflags register is loaded with IO Privilege set to 2 (only service tasks can access the IO space), and interrupts disabled

if this is a privilege 0 thread. The LDT selector register is loaded with the task's LDT selector, which is the one corresponding to its privilege level. Finally, the instruction pointer register is loaded with the entry point supplied for the newly created thread.

arch_run_thread:

do task switch. A task switch to the target thread is initiated by performing a far jump to offset zero of the thread's TSS descriptor in the IDT. No nesting of tasks is produced.

arch_cli:

disable interrupts. The interrupt enable bit of the eflags register is saved and then cleared. The function returns the original value.

arch_sti:

enable interrupts. The previous value of the interrupt enable bit is examined and interrupts are re-enabled only if they weren't disabled before the call to the previous arch_cli.

Chapter 3

Final thoughts

We consider that the microkernel concept provides a natural and adequate layer of abstraction between the management of the central processor, the memory management unit and the interrupt controller and the rest of the operating system. Sartoris, being a very thin microkernel (even the scheduling is performed outside the kernel), imposes a few performance constraints on the operating system; however, we don't believe this restrictions would hinder the overall performance of the system for all but a few very demanding scenarios. The benefits of a microkernel architecture seem to outweight the small performance hit.

On the implementation front, some features of the IA32 architecture, like processor hardware support for threading using Task State Segments, interrupt handling threads through task gates in the IDT, thread-nesting support through TSS nesting, and call gates to provide controlled access to operating system services, really simplified the implementation of the microkernel. The suspension and re-activation of threads is almost reduced to the execution single machine instruction.