Sartoris: an experiment regarding microkernel-based operating systems

Santiago Bazerque sbazerqu@dc.uba.ar
Nicolás de Galarreta nicodega@cvtci.com.ar

Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires.

Final exam, course "Organización de Computadoras II"
Director: Patricia Borensztejn, patricia@dc.uba.ar

Abstract: Sartoris is a minimal, portable microkernel. It provides direct support for the creation and destruction of tasks and threads, and inter-task communication mechanisms (message queues and shared memory). However, it attempts to be as policy-neutral as possible, providing a high degree of freedom to the operating system designer. In this article we discuss design issues and implementation strategies for the x86 family of processors. Finally, we describe a small Sartoris-based operating system, that was created to test the microkernel.

Key words: microkernel, processor architecture, multi-server operating system.

1 Introduction

The idea behind a microkernel-based operating system is to reduce the size and complexity of the kernel by implementing most of the operating system's functionality in the form of servers that run in user-mode. Initially, this idea promised a dramatic increase in flexibility, safety and modularity.

In a monolithic kernel, the operating system's functions are packed in the kernel: scheduling, memory management, file systems, networking, device drivers, and everything else. Traditionally, the μ-kernel approach only requires scheduling, inter-process communication and the implementation of support for several address spaces to be implemented in the kernel. This suffices to allow the implementation of the rest of the subsystems as servers. The advantages of this approach may be summarized as follows:

1. Different application programming interfaces (APIs) or even OS personalities can coexist in one system.
2. The system becomes more flexible and extensible.
3. Server (and driver) malfunctions are isolated as normal application's are.
4. A smaller kernel is easier to maintain and in general less error-prone.
5. The trusted computing base is significantly reduced.

However, first-generation μ-kernels turned out to be rather inefficient (compared to *monolithic kernels*), and they lacked real flexibility. Usually, they had rather large system-call sets and were designed to preserve unix-compatibility, but also allowing the construction of novel features. It was a common belief at that time that the layer of abstraction presented by μ-kernels was either too low or too high.

Recent second-generation μ-kernels address the flexibility and efficiency problems that appeared in earlier ones (a survey of this process is presented in [2]). QNX, L4 and the MIT's Exokernel are examples of second-generation microkernels. They were usually designed from scratch, and have smaller system-call sets than their predecessors. Also, to improve the efficiency of the system, they are usually highly dependent on the underlying hardware architecture.

While Sartoris is similar in some aspects to the microkernels just described (i.e. it makes heavy use of inter-process communication mechanisms, uses the task-thread abstraction to present the processor resource to the rest of the system, supports different program privilege levels and allows the execution of I/O operations outside the kernel), it provides a simpler abstraction of the processor and memory resources than most of them. Firstly, Sartoris performs no I/O at all[1], except for the initial loading of the main system servers. Secondly, since under Sartoris the scheduling is performed by one

[1]Therefore, device drivers are implemented as servers running in user mode.

or more scheduler threads (usually bound to the timer interrupt), all the functions provided by the μ-kernel are non-blocking (blocking is performed at a higher level, usually in the process-server). While these choices didn't yield a particularly small system call set[2], the nature of the resulting kernel is simple and elegant. Functionally, Sartoris behavior is somehow close to the MIT's Exokernel (further described in [4]), which also attempts to separate protection and management issues, leaving the latter in the application domain. Despite this initial coincidence, there are important differences: Sartoris only provides direct support for handling CPUs and main memory (upon request, it grants I/O privileges to programs, but without regarding the *semantics* of those rights), while the Exokernel exposes all the hardware, attempting to protect and isolate each application's actions. Furthermore, the Exokernel design includes software libraries that implement OS policies. Sartoris intention is to be general in that aspect.

2 Motivation and problem description

As was stated in the introduction, microkernel-based operating systems have many interesting features. However, there are some inherent difficulties that must be addressed in their construction. Our experiment consisted in the design of a minimal, architecture-neutral microkernel and it's implementation for the x86 processor. We also implemented a few device drivers and operating system servers that run on top of the microkernel, and a subset of the C library. Using the library, programs can use an API similar to the familiar unix API, with some restrictions due to the simple nature of the servers that were implemented.

Our μ-kernel design goals can be summarized as follows:

> 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.

[2]Sartoris has 22 system calls, quite numerous when compared, for example, with the 7-12 system calls used in different versions of the L4 μ-kernel.

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

The microkernel must provide the security primitives to allow the implementation of a secure environment.

3 Solution outline

As a first step in the design of the microkernel, we tried to reduce the functionality of the μ-kernel to a bare minimum, yet allowing a reasonable implementation of the necessary servers. We concluded that the necessary system calls can be grouped as follows:

Task and thread management. These system calls cover the loading and unloading of tasks into the system, and the creation, destruction and running of threads. They also permit the binding of a hardware interrupt to an interrupt-handling thread.

Memory management. These system calls handle the sharing of memory between tasks. This subsystem should also provide an abstraction of the paging mechanism (this was considered in the design of the microkernel, but has not been implemented yet).

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.

These system calls provide a simple processor abstraction, and they were sufficient for the implementation of all the basic servers. They are also completely policy-independent (note that there isn't even a simple scheduler within the microkernel: just the thread abstraction). Tasks, message queues, shared memory objects and threads have permission data that allows the operating system to restrict the way in which they interact with user programs. The microkernel implements software protection rings that may be used by the operating system to secure it's architecture. Also, a clear interface between the architecture-neutral, *algorithmic* section of the kernel (where all the permissions, shared memory objects, message queues, etc. are

maintained) and the hardware-specific section is defined (a similar approach is described in [3]).

4 Solution description

4.1 The system call set

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

```
/* sartoris system calls */

#ifndef SYSCALL
#define SYSCALL

#include <sartoris/kernel.h>

/* multitasking */
int create_task(int address, struct task *tsk, int *src, int init_size);
int destroy_task(int task_num);

/* threading */
int create_thread(int id, struct thread *thr);
int destroy_thread(int id);
int run_thread(int id);
int set_thread_run_perm(int thread, int perm);
int set_thread_run_mode(int priv, int mode);

/* interrupt handling */
int create_int_handler(int number, int thread, int nesting, int priority);
int destroy_int_handler(int number, int thread);
int ret_from_int(void);

/* message-passing */
int open_port(int port, int mode);
int close_port(int port);
int set_port_perm(int port, int task, int perm);
int set_port_mode(int port, int priv, int mode);
int send_msg(int to_address, int port, void *msg);
int get_msg(int port, void *msg, int *id);
int get_msg_count(int port);
```

```
/* memory sharing */
int share_mem(int target_task, void *addr, int size, int perms);
int claim_mem(int smo_id);
int read_mem(int smo_id, int off, int size, void *dest);
int write_mem(int smo_id, int off, int size, void *src);
int pass_mem(int smo_id, int target_task);

#endif
```

4.2 Tasks and threads

All the processing (even interrupt handling!) in a Sartoris based system is performed in the context of a task, and a thread within that task. A task is loaded into memory upon it's creation, and remains in main memory until it is terminated. No swapping of tasks is directly supported[3].

A task is composed by a flat virtual address space and communication mechanisms (ports and shared memory objects) and is identified by a number between zero and MAX_TSK[4], which is the parameter address passed to the system call create_task(int address, struct task *tsk, int *src, int init_size), where *src points to the beginning of the task image within the calling task's address space, and `*tsk` points to a structure of the type

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

where mem_adr indicates the physical address where this task should be placed in main memory[5], size indicates the size (in processor words) of the task being created, and priv_level indicates it's 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 are currently the only levels

[3]Of course, the operating system could easily implement such functionality.

[4]A constant defined by the implementation

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

that can access the input/output space. Also, the 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. Tasks are destroyed using the similar function destroy_task, which receives the address of the task being destroyed.

A thread is a path of execution within a given task. A task might have zero 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 already exist), ep is the thread's entry point, stack is the initial stack value, and invoke mode is one of the following:

> 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.
>
> PRIV_LEVEL_ONLY: In this mode, the invoking thread must have a privilege level numerically not greater than the invoke_level of the target thread. No per-thread specific permissions are required in this mode, only the described privilege-level restriction is applied.
>
> 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, 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 the mechanisms described above

restrict which threads might be invoked from a given thread-task pair. A thread is able to modify it's invoke_mode using the set_thread_run_mode system call.

4.3 Messaging system

The kernel provides an asynchronous inter-task messaging 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 corresponding to port number port of task to_address. The reception 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 in the queue of the supplied port. Before a port can be used to receive messages, it must be opened using the open_port system call, setting it's protection mode (it works in an analogous fashion to thread's protection modes) 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 unavailable (attempts to send messages to a closed port will fail).

4.4 Memory management

The microkernel must supply a mechanism to perform memory sharing. Therefore, each task owns a set of Shared Memory Objects which allow other tasks to access it's 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[6] at offset addr of the current task

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

address space, that can be accessed by the task[7] 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.

4.5 x86 implementation outline

This section is specific to the x86 processor family, described in [6], [7] and [8].

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 uses BIOS function 0x13 to load the kernel image and the init task image to memory. Then it jumps to the kernel initialization routines. 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.

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 to implement tasks and the TSS descriptors used for multi-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, 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:

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

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 descriptors	MAX_TSK	descriptors for task's Local Descriptor Tables
TSS descriptors	MAX_THR	descriptors for thread's Task State Segments

Interrupt Descriptor Table: The IDT contains the descriptors that define the processor's reaction to exceptions and external or software generated interrupts. 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.
Local Descriptor Tables: Each LDT must be contained in a special system segment in the GDT. 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.

4.6 IA32 low-level functions implementation details

This section briefly describes the behavior of the IA32 implementation of the functions defined in Sartoris' low-level interface.

arch_init_cpu: (invoked from the bootsector)
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.
syscall hooking. All the task gates for the system calls are created in the corresponding GDT positions.
IDT set up. The fist 32 entries of the IDT are filled with interrupt gates[8] 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

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

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 IDT is full with invalid descriptors.
Init service execution. Now the cpu is ready to run the μ-kernel. Using the create_task and create_thread system calls, the operating system init service is created at the exact address to which it was fetched earlier by the bootstrapping code, and executes using the run_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.
create LDT descriptor. The GDT descriptor for this LDT is created with the correct privilege level.

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:
TSS set up. The Task State Segment holds the contents of all the general purpose registers, the base and stack registers for all the privilege levels, the segment selector registers, the eflags register, the LDT selector register, the instruction pointer register, and a few more that are not used under Sartoris.
create TSS descriptor. Once the TSS is in place, a descriptor in the GDT must be created through which the thread may be started and resumed.

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 GDT. 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.

4.7 The test operating system

To test the μ-kernel, a small operating system was implemented. It is composed by

An **init server** that is loaded by the μ-kernel at boot time. It loads the rest of the servers and is destructed by the process server during the first scheduling round.

A **process server** that controls the creation of tasks and performs scheduling.

A **ram-fs server** that implements read-only memory filesystem.

A **console server** that acts as a driver for the keyboard and the VGA adaptor, and implements virtual consoles.

A **DMA server** that administrates direct memory access channels.

A **floppy disk driver server** that uses the DMA server to provide raw access to the floppy disk.

A **filesystem server** that implements a simple filesystem using the floppy disk driver server.

The servers use the messaging and memory sharing functions of the μ-kernel to communicate. The interrupt handlers were implemented as threads using the interrupt-handling support built in the μ-kernel.

5 Conclusions

The μ-kernel provided a suitable environment for the implementation of a simple operating system. Furthermore, building a modular and secure system was very straightforward. While performance was not measured, clearly the increasing amount of context switching and memory sharing that is required by having several servers is a complex issue and solving it requires a careful study of the interactions between the servers and the user programs. In [5], Bershad argues that IPC overhead is, on one hand, improving as hardware and microkenels evolve and on the other, small compared to the overhead introduced by other factors. As a long-term issue, it would be interesting to see the consequences of the incorporation of paging to Sartoris. While porting the microkernel to other hardware architectures has not been attempted yet,

we recently started a port to an x86-like architecture simulated within a unix process. The low-level interface was very successful, and it really simplified the port. However, the simulated architecture is not very different in nature to the original x86 architecture the μ-kernel was designed for. We believe that the aforementioned minimization of the μ-kernel greatly enhanced the extensibility of the system. We modified the test OS several times, adding, removing and modifying servers without great effort. Essentially, every policy was implemented outside the kernel and could be redesigned without risking the introduction of kernel bugs.

References

[1] Abraham Silberschatz, Peter Baer Galvin: *Operating system concepts*, Addison-Wesley Publishing Co., Reading, MA, USA, fourth edition, 1993.

[2] Jochen Liedtke: *Towards Real μ-kernels*, CACM, 39(9), to appear.

[3] See-Mon Tang, David K. Raila, Roy H. Campbell: *A Case for Nano-Kernels*, Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[4] Dawson R. Engler: *The Exokernel Operating System Architecture*, Ph.D. Thesis, MIT, 1998.

[5] Brian N. Bershad: *The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems*, 1992.

[6] Intel Corp.: *Intel Architecture Software Developer's Manual. Volume 1: Basic Architecture*, 1999.

[7] Intel Corp.: *Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference*, 1999.

[8] Intel Corp.: *Intel Architecture Software Developer's Manual. Volume 3: System Programming*, 1999.