An Object-Oriented *Nano-Kernel* for Operating System Hardware Support

See-Mong Tan David K. Raila Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
Digital Computer Laboratory
1304 W. Springfield
Urbana, IL 61801
{*stan,raila,roy*}*@cs.uiuc.edu*

Abstract

The nano-kernel *in the* μChoices *operating system provides hardware support for the operating system. The nano-kernel is a single, modular subsystem that encapsulates the hardware and presents an idealized machine architecture to the rest of the system. Higher levels of the system that implement policy access the nano-kernel through a single interface. Thus the* μChoices *nano-kernel is fully decoupled from higher level abstractions such as virtual memory or process paradigms. Within the nano-kernel, the hardware is modeled as a collection of abstract classes in a hardware support framework that are subclassed for particular hardware platforms. This architecture provides a highly modular and portable design making the system vastly easier to work with than previous versions of* Choices. *We have implemented a prototype of* μChoices *that runs on UNIX.*

1 Introduction

A prime concern in the design of modern operating systems is the system's portability across different hardware platforms. Operating system implementors have migrated from coding in pure assembler to writing most major parts of an operating system in high level languages such as C. This enhances portability by reducing the amount of machine-dependent assembler that needs rewriting for every new port of the operating system.

Coding in high level languages has indeed made later operating systems code much clearer, more flexible, and easier to maintain. However, significant amounts of machine dependent details remain in even the high level portions of the operating system. Such machine dependencies may appear in many major subsystems of an operating system, such as the file, memory management, and process subsystems.

The *nano-kernel* of the *μChoices*[3] operating system provides hardware support for the rest of the machine-independent micro-kernel. The nano-kernel is a single, modular subsystem that encapsulates the hardware and provides the mechanisms for implementing higher-level abstractions, such as processes, timers and virtual memory. Since all machine-dependencies are captured in a single subsystem that does not contain system policies, porting to another hardware platform involves porting only the nano-kernel. This also means that hardware dependencies are factored out of higher level subsystems, leading to easily portable higher levels. As the nano-kernel is not dependent on the high level design of the rest of the operating system, it is possible for many *different* operating systems to be constructed on top of it.

The low-level machine hardware is modeled in the nano-kernel as a framework composed of abstract classes. These classes are then specialized for particular machines through subclassing. The micro-kernel may access the nano-kernel only through a single hardware support interface. The interface allows us to limit and capture the way the nano-kernel is accessed and used in the rest of the system.

In this paper we describe the design of the nano-kernel in the *μChoices* object-oriented operating system and its first implementation for the *Virtual Choices* machine, an implementation of the operating system that runs on top of UNIX. This version of the nano-kernel presents to the micro-kernel an emulation of a symmetric shared-memory multiprocessor. We describe the overall architecture of *μChoices* in section 2, the internal structure of the nano-kernel in section 3, and the *Virtual Choices* port of the nano-kernel in section 4.

2 The Architecture of *μChoices*

μChoices is a micro-kernel operating system that supports the dynamic composition of an operating system from a set of subsystems. *μChoices* is a redesign of the Choices[2] object-oriented operating system as a micro-kernel that breaks the module interdependencies in the original Choices system and takes advantage of the useful code base and object-oriented design provided by Choices. The goal of *μChoices* is to support modern operating system services, such as user level and gang scheduling, distributed customizable virtual memory, and multi-media in a completely modular architecture while maintaining high perfor-

mance. The model describes:

- an idealized machine architecture (the nano-kernel) at the lowest level. The nano-kernel encapsulates data representations and algorithms associated with the instruction set, and virtual address mechanisms of the computer.
- a micro-kernel interface to the remainder of the operating system. This layer encapsulates the micro-kernel data structures and algorithms and separates them from other subsystems of the operating system.
- intermediate levels of the operating system and the design rules for their construction. The design rules specify the basic synchronization, resource allocation, and consistency requirements for the subsystems that are built upon the micro-kernel as part of the operating system. These subsystems (including client, server, and file system components) are object-oriented frameworks.
- Application interfaces that are used by applications to gain access to operating system services.

μChoices is designed around the ideas of

- objects,
- frameworks,
- independent modules for each subsystem, and
- explicit interfaces for each module.

μChoices is based on a framework for interconnecting different OS subsystems, with these subsystems realized as separate modules. Modules are implemented as independent object-oriented frameworks that interact through well defined interfaces. Within a framework, the subclassing of components provides the ability to customize its various parts to support various implementations and optimizations. *μChoices* may be viewed as a framework for embedding the sub-frameworks for each OS subsystem.

μChoices has *independent* sub-frameworks for each OS subsystem. That is, each OS subsystem is a module realized as a set of abstract classes. Sound software engineering principles are used to decompose the operating system into interacting modules. Modules interact only through well-defined interfaces. Code reuse through inheritance is used within modules internally, and interface reuse through inheritance of abstractions is used between sub-frameworks. This design effectively decouples each module from implementation details specific to other modules. We consider implementation inheritance across module boundaries ill-advised, leading to complicated dependencies and interrelationships between classes in different modules.

Other systems have alluded to the benefits of decoupling interfaces from implementation. Spring[5] uses the Interface Definition Language (IDL) to specify interfaces to objects in the system[6]. One advantage of IDL is that it gives language heterogeneity by hiding implementations behind language independent interface specifications. We have eschewed adding IDL and its associated stub generation tools to *μChoices* choosing instead to implement interfaces programmatically.[1] Baumgartner and Russo[1] report that exporting abstract superclasses make it hard to retroactively change or introduce superclasses. Our experience with modifying the process subsystem in *Choices* in order to support process migration confirm their thesis. Their solution is to introduce a new C++ language construct called the *signature*, which contains only interface definitions.

In *μChoices* *modules* export interfaces to allow other subsystems and clients access to the module. Interfaces in *μChoices* are realized as C++ objects instead of making the abstract superclasses of our frameworks visible. The latter approach, which was taken in *Choices*, caused modules to depend on the structure of the exported abstract superclass. Interfaces are composed to provide simple mechanisms that can be used by clients to implement various policies, analogous to the way a RISC instruction set is designed to be simple and efficient. We chose not to rely on IDL or a non-standard language extension for simplicity and portability.

3 Internal Structure of the Nano-Kernel

The *μChoices* nano-kernel is composed of:

- the *boot* component, responsible for booting and initializing the operating system.
- the *processor* component, responsible for managing physical CPUs.
- the *memory management* component, responsible for managing the MMU, TLB and virtual address mappings.
- the *trap scheduler*, responsible for handling hardware traps and interrupts, scheduling the software handlers for these events, and for providing a machine-independent interface to these services.
- the *boot console* component, responsible for console output at boot time.
- the *debugger* component, responsible for debugger hooks in the kernel.
- the *interface component*, responsible for providing single interface for accessing the hardware support module

The code structure for the nano-kernel organizes the above components into four subdirectories:

1. Framework
2. Processor Dependent
3. Machine Dependent
4. Interface

[1] *μChoices* is written in the C++ language[7].

Class	Methods
Lock	Lock acquire release
CPU	disableInterrupts restoreInterrupts setTrapScheduler
TrapScheduler	setException setPriority scheduleTrap
MMU	enable setMap flushTLB
ProcessorContext	restore checkpoint
Console	write
SystemConfiguration	numberofProcessors pageSize numberOfPhysicalPages

Table 1: Nano-Kernel Internal Framework Classes

The Framework subdirectory holds the basic, abstract classes in the nano-kernel for the *μChoices* idealized machine architecture. These abstract classes in the nano-kernel encapsulate hardware entities such as the processor (class `CPU`) and memory management unit (class `MMU`). Table 1 shows the basic classes and some important public methods for objects in the nano-kernel framework. Class `Lock` in the nano-kernel framework implements a mutual exclusion lock between CPUs. The micro-kernel may request and use locks through the hardware interface. A concrete subclass implements locking and unlocking through appropriate methods provided by the architecture, such as test-and-set instructions. Class `CPU` implements an abstract protocol for handling interrupts and other CPU related mechanisms. Class `MMU` implements an abstract protocol for controlling the memory management unit on the machine. The `ProcessorContext` class gives the higher level micro-kernel the means to implement a process subsystem, with methods to checkpoint and restore the running CPU thread from one context to another. The `Console` implements a C++-like output stream. An instance of the `SystemConfiguration` class allows the querying of system parameters such as the number of processors, the machine page size, and amount of RAM.

The Processor Dependent subdirectory contains implementations of abstract classes in the framework for particular processor architectures. For example, within ProcessorDependent, the subclasses for the SPARC or MIPS processors may be found. The framework requires concrete subclasses for the abstract classes

- CPU,
- MMU,
- AddressTranslation, and
- ProcessorContext.

Method	Comment
newLock	Get mutual exclusion lock
lockAcquire	Acquire the lock
lockRelease	Release the lock
cpuDisableInterrupts	Interrupts off this CPU
cpuRestoreInterrupts	Restore interrupt mask
cpuSetTrapSched	Set CPU to use a trap scheduler
contextCheckpoint	Checkpoint to context
contextRestore	Restore saved context

Table 2: Interface to the Nano-Kernel.

The MachineDependent directory contains subclasses for particular machines implementing device drivers and specializations between machines of the same processor architecture.. The machine dependent subdirectory holds concrete subclasses of the abstract classes for

- Console,
- MemoryAllocator,
- Drivers and Exceptions,
- SystemConfiguration.

Specialized boot code for the machine is found here as well.

The Interface directory holds the definition of the interface that the nano-kernel exports. We allow access to the nano-kernel only through this interface. This architecture maintains module independence and permits us to capture and limit the way the nano-kernel is accessed. A portion of the interface is shown in table 2.

4 The *μVChoices* Implementation of the Nano-Kernel

μVChoices is an implementation of the *μChoices* nano-kernel that emulates a symmetric shared memory multiprocessor on top of the Solaris flavor of UNIX. *μVChoices* provides a convenient and robust prototyping environment for testing and debugging design ideas in *μChoices* similar to *VirtualChoices*[4]. It has a much faster edit-compile-test cycle, no need for dedicated hardware, and is compatible with native implementations of *μChoices*. This section briefly presents the concrete subclasses that *μVChoices* supplies to implement *μChoices*.

The *CPU* subclass in *μVChoices* is programmed with a UNIX process. Additional CPUs are programmed as additional UNIX processes communicating via shared memory and the underlying Solaris filesystem. On a multi-CPU machine true parallel execution can be achieved. Locking is implemented with test-and-set or swap instructions in the SPARC instruction set.

The *MMU* is realized through the virtual memory environment of the Solaris process. The memory

Hardware Interrupt	Signal
Timer expiration	SIGALRM
Debugger trap	SIGTRAP
VM fault	SIGSEGV SIGBUS SIGILL
Floating point fault	SIGFPE
Asynchronous Input/Output	SIGIO

Table 3: A Portion of the Mapping of *μVChoices* Interrupts to Signals.

mapped file facilities are programmed to emulate the manipulation of the AddressTranslation and a Solaris file provides the representation of the pages of physical memory.

Interrupts and traps in *μVChoices* are realized through the use of Unix *signals*. Signals bear a close resemblance to hardware interrupts on physical processors and can be programmed to emulate interrupts. The signals are programmed by registering handlers, catching signals associated with IO, virtual memory, timers, instruction faults, etc., and by manipulating the signal mask to mask and unmask interrupts. Table 3 shows the correspondence between hardware interrupts and Unix signals used.

μVChoices is similar to the Nachos[8] instructional OS in that it runs as a Unix process. *μVChoices* is different in that it not stripped down and simplified, nor do we interpret application code. *μVChoices* also provides multiprocessor support, with physical processors simulated as separate Unix processes. This allows simulated CPUs separate memory management capabilities. Page faults are caught through signal handlers for memory segmentation and bus violations. This provides a more realistic simulation of hardware. *μVChoices* supplies a *complete prototyping environment* for our operating systems development.

5 Conclusion

μChoices is the result of a redesign of the original Choices operating system with the focus on a modular micro-kernel design. Our design is an operating system based on a framework for interconnecting different OS subsystems. Each subsystem is realized as a separate module implemented as an independent object-oriented framework. These modules interact only through well defined interfaces.

The nano-kernel in *μChoices* is the hardware support module that localizes particular machine dependencies and presents an idealized machine architecture to the rest of the operating system. It is entirely divorced from the rest of the OS. Efficient primitives are exported to the rest of the kernel so that higher level operating system abstractions may be constructed. We posit that many different operating systems may be constructed on top of the portable nano-kernel. Portability is enhanced since porting to a new machine means working with a single, small, modular system. The framework defined by the abstract classes and relationships in the nano-kernel guide and simplify the process of targeting new hardware.

The *μVChoices* implementation of *μChoices* provides an emulation of a symmetric shared memory multiprocessor on top of the UNIX operating system. *μVChoices* is a convenient environment for working with *μChoices* and is code compatible in all but the machine and processor subclasses. It provides *μChoices* in an environment rich with tools such as debuggers and profiling tools.

6 Acknowledgments

Amitabh Dave, Willy Liao, David Putzolu, Tin Qian, Aamod Sane, Mohlalefi Sefika, Ellard Roush and Lun Xiao, members of the Systems Research Group at the University of Illinois, also contributed extensively to the design of *μChoices* and the nano-kernel.

References

[1] G. Baumgartner and V. F. Russo. Implementing Signatures for C++. In *USENIX C++ Conference*, Boston, MA, 1994.

[2] Roy H. Campbell and Nayeem Islam. " Choices: A Parallel Object-Oriented Operating System". In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[3] Roy H. Campbell and See-Mong Tan. μChoices: An Object-Oriented Multimedia Operating System. In *Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, Washington, May 1995. IEEE Computer Society.

[4] David Raila and Jishnu Mukerji. A Prototyping Environment for the Choices Operating Systems. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign and Advanced Architecture Department, Unix Systems Laboratories, 1993.

[5] J. Mitchell et al. An Overview of the Spring System. In *Proceedings of Compcon 'Spring 1994*, February 1994.

[6] P. B. Kessler. A Client-Side Stub Interpreter. *ACM SIGPLAN Notices*, 29(8):97, August 1994.

[7] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1986.

[8] W. A. Christopher and S. J. Procter and T. E. Anderson. The Nachos Instructional Operating System. Technical Report UCB//CSD-93-739, University of California, Berkeley, April 1993.