Virtual Hardware for Operating Systems Development

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

Abstract

Developing an operating system on bare hardware is difficult due to an inhospitable development environment, long edit-compile-run-debug times, and the need for extra target hardware. This paper contributes general techniques for creating *virtual hardware* for operating systems development. The virtual machine is realized on top of UNIX and is a close approximation of real hardware, including interrupts, time slicing, virtual memory, devices, multiple processors with separately programmable memory management units, and the ability to run application programs natively. Debugging and testing our operating system in such an environment was considerably quicker and easier compared to developing on bare hardware.

1 Introduction

Developing operating systems is difficult on bare hardware. There is a paucity of tools for run time and post mortem debugging, as well as execution profiling. Rapid prototyping requires quick edit-compile-run-debug cycles. Fast turnarounds are a problem when machines take up to several minutes to boot. Incorrect operating system code can also cause the hardware to "hang." In most cases this necessitates a power cycle or a push on a reset button. In addition, native implementations require target hardware to run the operating system on. In fact, the debugging of a native port of an operating system typically requires two machines: one as the target while the other runs a debugger such as GNU's `gdb`. The expense of such setups contributes to the difficulty of developing operating systems directly on bare hardware.

A number of instructional operating systems, such as Nachos[13], its predecessor TOY, and the operating system simulator used at the University of Illinois[7], run as regular UNIX[9] processes. The operating system is simulated within the UNIX process. This removes the impediment of requiring target machines in order to run the system. Students run and debug the instructional system on widely available platforms running a stable operating system with stable development tools. These simulations, however excellent for instructional purposes, are limited. The kernel code is stripped down and simplified. Virtual memory is not simulated closely, thus user applications must be interpreted in order to catch page faults and other exceptions.

This paper contributes general implementation techniques for creating *virtual hardware* on UNIX for operating systems development. The virtual machine is a close approximation to real hardware. Features include:

- interrupts,
- time slicing,
- virtual memory,
- devices,
- multiple processors with separately programmable memory management units, and
- the ability to run application programs natively.

This approach was used to provide a prototyping environment for the development of the *Choices* and *μChoices* object-oriented operating systems[1, 3]. We call the two systems that run on the UNIX virtual

machine *VirtualChoices* and μ*VirtualChoices* respectively. Developing and testing operating system code for *VirtualChoices* and μ*VirtualChoices* resulted in faster edit-compile-run-debug cycles compared to working with the native implementations of the systems. Debugging was also considerably easier with the debugger operating on standard UNIX processes. It also allowed us the use of a memory leak and bounds checker[1] on kernel code. This would have been impossible in a native implementation.

Note that even though the operating system runs on a virtual machine implemented on top of UNIX, the kernel code is not simplified or stripped down for that purpose. Except for a small machine-dependent part, the code is the same as that for our native SPARC and Intel $x86$/Pentium ports of the system. In addition, application code is not interpreted. Applications run natively. The virtual machine generates page faults on application accesses to non-mapped pages. We preferentially develop and test our subsystem prototypes in virtual mode before embedding them in native implementations. Although performance profiling in *VirtualChoices* and μ*VirtualChoices* do not reflect the actual numbers we would get in native mode, performance comparisons made in virtual mode are useful in gaining insight into the system.

2 *Choices* and μ*Choices*

Choices is a full-featured object-oriented operating system. The *Choices* kernel is implemented as a dynamic collection of interacting objects. System resources, policies, and mechanisms are represented by objects organized in class hierarchies[11]. The system architecture consists of a number of subsystem design frameworks[2] that implement generalized designs, design constraints, and a skeletal structure for doing customizations. Key classes within the frameworks can be subclassed to achieve portability, customizations, and optimizations without sacrificing performance[10]. The design frameworks are inherited and customized by each hardware specific implementation of the system providing a high degree of re-use and consistency between implementations.

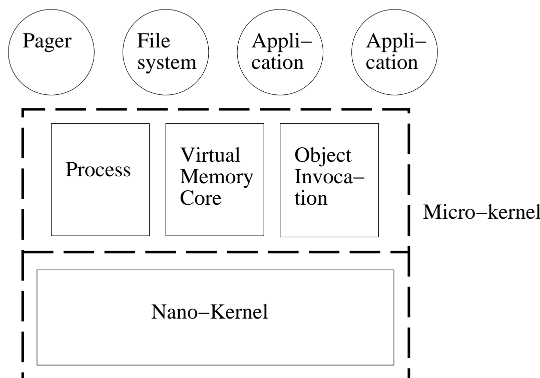

Figure 1: The μ*Choices* micro-kernel is split into high level kernel code and a nano-kernel.

μ*Choices* is a redesign of the original *Choices* system as a micro-kernel based operating system. In μ*Choices*, we have split the micro-kernel into two portions (see figure 1. The machine-dependent *nano-kernel*[12] encapsulates the physical hardware and provides hardware support for the rest of the machine-independent micro-kernel. It provides the micro-kernel with the needed mechanisms for implementing higher-level abstractions, such as processes, timers, and virtual memory. The nano-kernel is not a wrapper around assembler routines. Because μ*Choices* is an object-oriented operating system, the nano-kernel is built as a framework of classes that captures the essential properties of the low-level hardware, presenting a useful interface to the higher levels of the kernel in a machine-independent way. It provides the building blocks for constructing higher level kernel abstractions.

The abstract classes in the μ*Choices* nano-kernel were specialized to create the virtual hardware for μ*VirtualChoices*. The objective was to reuse the entire body of code at the high level micro-kernel, as well as the application level code, unchanged. The remainder of this paper describes the virtual hardware implementation in μ*VirtualChoices*.[2] We developed the system initially on SunOS 4.1, then later on SunOS 5.4 (Solaris 2.4) for Sun SPARC architectures. A Linux port is in progress.

[1] We used `purify`, a trademark of Pure Software, Inc.

[2] μ*VirtualChoices* extended *VirtualChoices*[5] to include multiple processor emulation.

3 Hardware Interface

The *µChoices* nano-kernel hardware interface provides an idealized machine architecture to the higher levels. The basic interface exports:

- one or more processors with maskable interrupts,
- a memory management unit per processor to implement virtual memory, and
- an interface for registering exception handlers.

As the *µChoices* nano-kernel is entirely divorced from the rest of the higher-level operating system, the UNIX virtual hardware is programmed entirely at the nano-kernel level. A separate consequence of our design is that many *different* operating systems may be built on top of the nano-kernel.

The virtual hardware is realized using several basic facilities in UNIX. CPUs are mimicked with the UNIX process facility. The thread of the UNIX process is programmed to follow the thread of control a hardware CPU would in a hardware implementation. Interrupts are emulated with UNIX signals. Interrupts are controlled through the use of UNIX signal masking and signal handling facilities. Virtual memory hardware is implemented with the UNIX memory mapped file facilities. The contents of a UNIX file represent pages of physical memory and are mapped to virtual memory locations on page boundaries. Page protection levels are manipulated with system calls to set the protection levels of memory mappings in a UNIX process. Virtual memory related signals are vectored to the virtual hardware's page fault handler. Hardware device drivers are modeled with non-blocking I/O and signals on file descriptors. These techniques flesh out the hardware support layer in our operating systems. The results are complete implementations of *Choices* and *µChoices* that run in a convenient environment.

4 Design Details

This section describes in detail the design of the virtual hardware in the *µVirtualChoices* nano-kernel. The discussion describes the abstract classes reifying the hardware entities in the nano-kernel, and their realization in the UNIX virtual machine with concrete implementations in the *µVirtualChoices* subclasses.

4.1 CPU

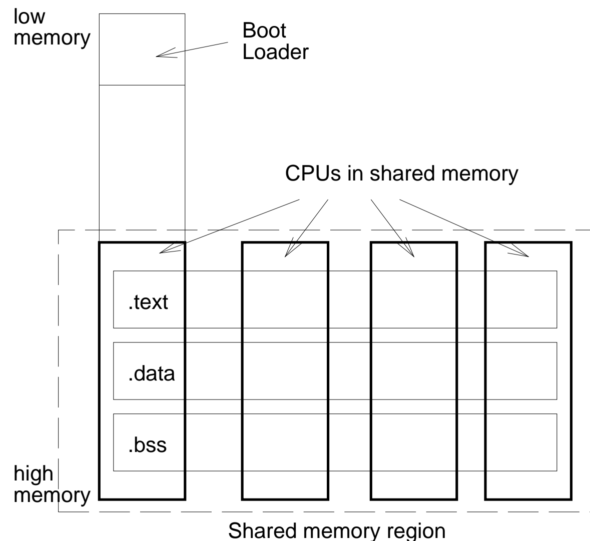

Figure 2: Multiprocessors in *µVirtualChoices* are emulated with heavyweight processes and shared memory.

CPUs in *µVirtualChoices* are programmed with a UNIX process. The thread of control of a UNIX process emulates the execution of a hardware CPU. Context switching and interrupts change the flow of control in a hardware CPU. Their emulation is described later.

The nano-kernel in *µVirtualChoices* emulates a symmetric shared memory multiprocessor. Our initial implementation of multiple processors utilized kernel lightweight threads in Solaris. Time slicing by the UNIX scheduler provided the illusion of asynchronous, parallel execution between the different CPUs. On a physical machine with multiple processors, each of the threads could execute concurrently, thus giving us true parallel execution.

However, we required separately programmable memory management units on a per-CPU basis. Lightweight threads live in a single UNIX virtual memory domain. It was not possible to allocate separate virtual memory maps to each thread since any thread would see a memory object mapped by another thread. To solve this problem, we used heavyweight UNIX processes to emulate the virtual machine CPUs in place of lightweight threads. In order to emulate a symmetric shared memory multiprocessor, the kernel text, data and heap segments of the operating system must be accessible to every CPU. We used a boot loader. The boot loader creates a shared memory region using UNIX shared memory primitives (`mmap`), then loads the text and data pages of the *µVirtualChoices* image into the shared memory region.[3] The boot loader then transfers control to the *µVirtualChoices* image it loaded into the shared memory region. Multiple CPUs are then created using the UNIX system call `fork` to create as many heavyweight processes as there are CPUs. As the text, data and heap space of the kernel is in a UNIX shared memory region, we obtain an emulation of a symmetric shared memory multiprocessor on top of UNIX. In addition, since each CPU is emulated as a separate heavyweight UNIX process, it was possible to create shared memory regions which are mapped on a per-process basis. This forms the basis for separately programmable memory management units per virtual CPU in *µVirtualChoices*.

4.2 Context Switching

The abstract class `ProcessorContext` in the nano-kernel gives higher levels of the micro-kernel a means to implement a process subsystem. Processor contexts store the state of the CPU, including:

- the program counter,
- registers,
- interrupt mask information,
- stack pointers,
- frame pointers, and
- any other information not preserved across function calls that the CPU may need to restore a running task.

The original `ProcessorContext` class exported two methods, namely `checkpoint` and `restore`. The first saves the present context, the other restores a previously saved context. The `checkpoint` method returns 0 on the first call, and will appear to return non-zero on being restored by another thread. We redesigned `ProcessorContext` later to export the `switchTo` method instead. `SwitchTo` takes two `ProcessorContext` arguments. It checkpoints the current processor state in the first context, and restores the state previously saved in the second. Context switching from one process to another in the micro-kernel involves swapping the processor contexts of processes.

The SunOS version of the `ProcessorContext` was realized in the `VCContext` concrete subclass. `Checkpoint` and `Restore` were implemented in SPARC assembler. The Solaris version of the nano-kernel implemented `switchTo` with C library calls to `makecontext`, `getcontext` and `swapcontext` respectively. These calls manipulate user contexts and are used to implement context switching between user-level threads in Solaris. They conveniently mirror the checkpointing and restore functions in the `ProcessorContext` class.

4.3 Traps and Interrupts

While a multiplicity of interrupts may be generated and differ from one machine to another, most operating systems are only interested in a very few at the micro-level. The nano-kernel exports several machine-independent exception types to the machine-independent micro-kernel. These are:

1. `MemoryException`, for memory access violations,

[3] The UNIX `mmap` call establishes a mapping between a UNIX process's virtual memory address space and a memory object. The memory object is represented by an open file descriptor. Equivalent calls to manipulate shared memory without using the file system are `shmget`, `shmat`, and `shmdt`. `Mprotect` may be used to affect the protection level of pages in a process's address space.

Signal	UNIX Meaning	*VirtualChoices* Use
SIGILL	Illegal Instruction	Catch Illegal Instructions
SIGSEGV	Segmentation Violation	Virtual Memory Page Fault
SIGBUS	Bus Error	Virtual Memory Page Fault
SIGUSR1	User Defined/Generated	System Call
SIGFPE	Arithmetic Exception	Catch Floating Point Exceptions
SIGCONT	Continue After Stop	Console To Asynchronous Mode
SIGALRM	Alarm Clock	Timer and Time Slice Interrupt
SIGSTP	Stop From Keyboard	Console To UNIX Mode
SIGIO	I/O Possible on Descriptor	Asynchronous IO
SIGINT	Interrupt Process	Interrupt Current Process
SIGQUIT	Quit Program	Clean Up and Halt Gracefully
SIGABRT	Abort Program	Generate core Dump for Analysis

Table 1: UNIX Signals and their uses in μ*VirtualChoices*.

2. `IllegalInstructionException`, for invalid instructions,
3. `FloatingPointException`, for arithmetic errors,
4. `TimerException`, for timer expirations,
5. `FreeRunningTimerException`, for free running timer expirations,
6. `IOException`, for IO device exceptions.
7. `SystemCallException`, for system calls.

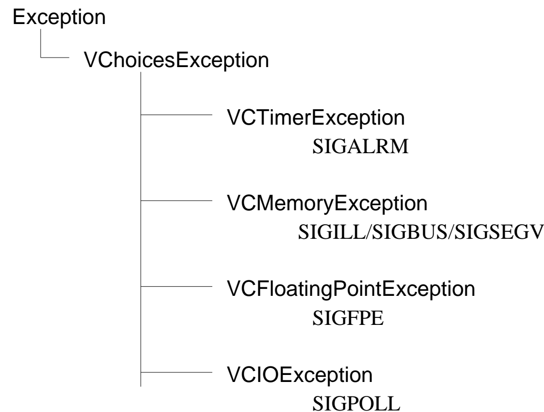

Figure 3: `Exception` class hierarchy for μ*VirtualChoices* and the associated UNIX Signals

In μ*Choices*, the nano-kernel binds an occurence of an interrupt or trap to a `raise` method on an instance of class `Exception`. The `Exception` object maps the hardware interrupt to one of the six types above and calls the micro-kernel registered exception handler, if one exists. As class `Exception` is an abstract class, it is specialized in the μ*VirtualChoices* nano-kernel into concrete subclasses that bind UNIX *signal handlers* (see figure 3). UNIX signals are asynchronous notification of events which are sent to user processes. Processes may register signal handlers. The occurence of a signal interrupts the normal flow of a program and results in the appropriate signal handler being invoked. A signal may be *masked*, in which case it is prevented from delivery to the process.

Table 1 shows the signals μ*VirtualChoices* uses. The first group corresponds to synchronous traps generated by the currently executing instruction. The second group of signals are asynchronous traps and interrupts that may not have been generated by the currently executing instruction.

The nano-kernel idealized machine architecture also provides ways to enable, disable and restore interrupts on a processor. μ*VirtualChoices* implements this with the UNIX signal masking facilities. The `sigsetmask` system call is used to change signal masks and thus change the set of interrupts a process will catch.

4.4 Virtual Memory

The intent of the virtual memory emulation in *μVirtualChoices* is to duplicate, as closely as possible, the semantics of physical memory management units on physical CPUs. The emulation allows the direct reuse of the *μChoices* virtual memory subsystem on the UNIX environment, with no change.

The *μChoices* memory model assumes two virtual memory regions, one for the kernel use and the other for application use. The kernel addresses are found above the application addresses. Kernel region mappings are always active. During the initialization of *μVirtualChoices*, a region for applications is reserved in low memory by memory mapping a UNIX file to reserve virtual memory.

VM mappings for a VM address space is constructed by adding chains of physical pages into an instance of `AddressTranslation`. The collection of pages in an instance of the class represents a virtual memory domain. `AddressTranslation` is an abstract class. Methods on the class allow one to add or remove mappings to physical pages at different addresses, as well as affect the protection levels of the pages in the translation. Although the protocol is specified by the abstract class, the actual implementation is left up to a concrete subclass. Concrete subclasses implement the actual translation in an efficient, machine-dependent manner.

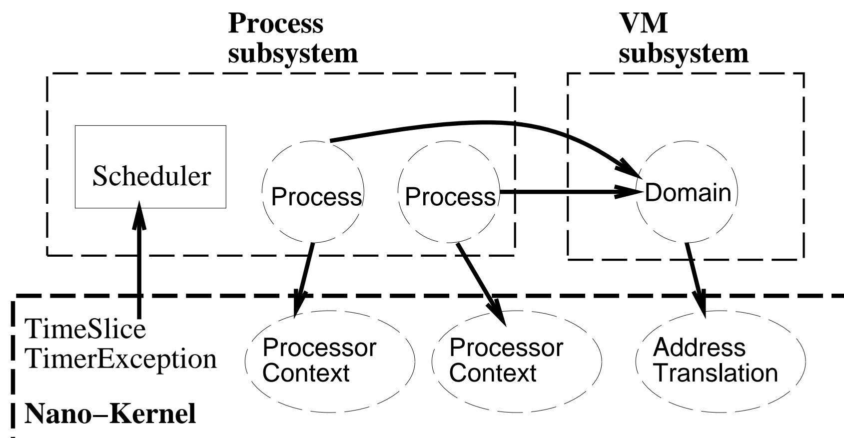

Figure 4: Two processes sharing the same VM domain in *μChoices*.

Figure 4 illustrates an example where two processes share the same virtual memory address space. Each process is managed by the Process subsystem and contains references to nano-kernel provided `ProcessorContext` objects. The virtual memory domain is represented by a `Domain` object, which contains references to an instance of the `AddressTranslation` class.

Class `MMU` implements an abstract protocol for controlling hardware memory management units. Instances of class `MMU` encapsulate the memory management units on the machine. `MMU`s operate on instances of the `AddressTranslation` class. The basic methods exported by the `MMU` interface allow one to enable its operation, activate a given address translation and flush the MMU cache for an address range. A virtual memory domain is mapped when the `activate` method of an instance of class `MMU` is invoked with the translation.

μChoices manages virtual memory by sending the `add`, `remove`, and `changeProtection` messages to instances of the `AddressTranslation` class that is active on the MMU as a result of page faults or virtual memory primitives. The `VCPageTable` concrete subclass of the `AddressTranslation` class implements these methods by maintaining information about the virtual memory environment and by using the UNIX `mmap` system call.[4] Kernel region translation updates are performed directly using the `mmap` call. Application ranged updates are stored and applied using `mmap` if the VCPageTable being updated is the one currently active on the MMU. Virtual memory environments are switched by invalidating the old application range and restoring the application range translation information stored in the new `VCPageTable` using the `mmap` call. Also, each CPU, being emulated as a heavyweight UNIX process, has its own virtual memory map. Thus we obtain a per virtual CPU memory management capabilities. By duplicating the semantics of virtual memory hardware, *μVirtualChoices* directly reuses the entire virtual memory subsystem of *μChoices* in the UNIX environment.

[4] We create virtual memory segments by attaching temporary files with the `mmap` system call.

4.5 I/O and Device Drivers

The *μChoices* nano-kernel `IOException` type represents possible exceptions due to IO devices. I/O interrupts caused by I/O devices cannot easily be cast into a machine-independent mold. Thus each machine-independent exception type may have a ***hardware-dependent vector*** associated with it. In most cases, the hardware vector is known in the processor dependent or machine dependent subclass for that exception type. For example, the SPARC processor uses vector 26 for timer expirations. The vector argument is usually required for the type `IOException`, as different IO devices may raise interrupts on different vectors. The nano-kernel's device driver clients use it in order to register handlers on particular I/O device vectors. Since device drivers are inherently machine-dependent anyway, this does not compromise the machine-independent nature of the rest of the interrupt processing interface.

In *μVirtualChoices*, the vector is a UNIX file descriptor. For example, the *μVirtualChoices* network device is realized as an open, raw UNIX file descriptor that gives access to the hardware network device.[5] The UNIX `ioctl` system call is used to manipulate the file descriptor for asynchronous I/O, and to request the UNIX `SIGPOLL` signal when I/O is ready on the descriptor. Thus by taking advantage of the UNIX facilities for raw synchronous I/O and signals, *μVirtualChoices* supports many of the same interrupt driven devices that a bare hardware implementation does.

5 Experiences Implementing and Using *VirtualChoices* and *μVirtualChoices*

UNIX System/Library Calls	*μVirtualChoices* Use
`fork`	Create a virtual processor
`sigprocmask/signal`	Interrupt control
`mmap/mprotect`	Virtual memory/address translation
`open/read/write/lseek`	Virtual disk driver
`open/send/recv`	Virtual network driver
`getpid`	Virtual CPU identification
`setitimer`	Periodic and time slice timer management
`makecontext/swapcontext`	Process context switching

Table 2: UNIX system and library Call use in *μVirtualChoices*.

Table 2 lists the UNIX system and library calls which are used to implement the *μVirtualChoices* nano-kernel. The code uses no assembler, compared to the approximately 1200 lines of SPARC assembler in the native SPARC version of the nano-kernel. The nano-kernel implementation was quick and easy compared to the native hardware implementations. This was due to well understood system calls in place of cryptic assembler instructions, as well as the availability of UNIX debugging tools. The use of standard UNIX system and library calls eased the porting of the nano-kernel from one UNIX platform to another. We moved from SunOS 4.1 to Solaris 2.4 in a week. The original *VirtualChoices* was also ported to UNIX SVR4.2 on Intel hardware with ease.

In addition to portability, the virtual mode version of our experimental operating system is quicker to start up and debug. Booting a virtual kernel takes several seconds, compared to ten minutes for an install-boot cycle on native platforms. Machine independent operating system code in *μChoices*, including network protocols, the virtual memory subsystem[4], and the object invocation layer[8] were developed with the virtual hardware. The code could then be placed into native versions with little change.

UNIX is a general purpose operating system. It was not designed as a virtual machine. Thus the virtual hardware implementation in *μVirtualChoices* suffers from several minor flaws.

- **Lack of a programmable protection model**

  While the UNIX `mmap/mprotect` system calls can be used to protect kernel code from user applications in the virtual implementation, nothing prevents the application from reusing the same calls.

[5] The Ethernet device is named "`/dev/le0`" on Solaris 2.4.

- **Lack of sufficient page protection states**

  The page protection implementation in UNIX does not support protection levels or adequate reporting of the cause of virtual memory faults. Memory violations in UNIX result in one of three possible signals: `SIGILL`, `SIGBUS` or `SIGSEGV`, for illegal instruction, bus error or segmentation violation respectively. Improved support is needed to hone the fault class, for example, page not present or no execute permission.

- **Lack of support for asynchronous I/O completion**

  The UNIX file model does not support a signal on I/O completion on a file descriptor. The virtual disk driver thus does not implement asynchronous write behavior.

6 Conclusion

The virtual machine concept was pioneered by IBM in its VM/360 and VM/CMS[6] systems. These systems gave the illusion of separate *physical* machines to each user. Each virtual machine ran a separate copy of the operating system. The core system multiplexed between each user's virtual machine. VM/360 and VM/CMS were designed specifically to support virtual machines. In contrast, UNIX was designed as a general purpose time sharing system, not as a virtual machine. The system call and C library interface served as the "virtual machine" on which user applications executed.

Modern features, such as memory mapped files and shared memory regions, have accreted to UNIX over the years. UNIX has become a common workstation operating system. The environment within which an operating system is developed on bare hardware is cumbersome compared to what a UNIX applications developer is accustomed to. We designed the *VirtualChoices* and *μVirtualChoices* prototyping environments to take advantage of the ease of development on top of UNIX. The machine architecture dependent classes in our operating systems were specialized to emulate key features of physical hardware using the UNIX systems interface. The machine independent, higher level subsystems could then be prototyped in a UNIX environment. The emulation provided by the *μVirtualChoices* nano-kernel closely approximates a physical symmetric shared memory multiprocessor. We preferentially prototype our machine independent operating systems code in virtual mode. Our experience shows a tremendous gain in productivity due to lessened edit-compile-run-debug turnarounds and the ease of debugging. The virtual hardware allowed us to use a memory leak and bounds checker on kernel code, something that would not have been possible in a bare hardware implementation.

The virtual hardware allowed us to prototype our operating system without the need for dedicated target hardware. It removed the inconvenience of rebooting production systems and the need to isolate potential damage from experimental software. For small research and educational environments, dedicated target hardware is often not feasible.

The techniques given in this paper (UNIX signals, processes, shared memory facilities, etc.) are applicable beyond the *μChoices* nano-kernel. However, the nano-kernel does provide a convenient encapsulation of the hardware and is ideally retargetable toward a UNIX virtual machine. Since the nano-kernel is completely divorced from higher level code, there is no hindrance toward the reuse of the UNIX mode nano-kernel in other operating systems.

Acknowledgements

We gratefully thank the members of the Systems Research Group at the University of Illinois, Amitabh Dave, Tin Qian, Aamod Sane, and Mohlalefi Sefika, for their ongoing help in our work. UNIX Systems Laboratories also contributed extensively to previous work on *VirtualChoices*: we thank Jishnu Mukerji, Jozef Chou, Tom Vaden and Dennis Weis.

References

[1] 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.

[2] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. *Choices*, Frameworks and Refinement. In Luis-Felipe Cabrera and Vincent Russo, and Marc Shapiro, editor,

Object-Orientation in Operating Systems, pages 9–15, Palo Alto, CA, October 1991. IEEE Computer Society Press.

[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 K. Raila and See-Mong Tan and Roy H. Campbell. Remote Procedure Call Implementations of Microkernel Virtual Memory Services Degrade System Performance. Submitted to USENIX Technical Conference 1996.

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

[6] International Business Machines. *Virtual Machine/System Product.* Endicott, New York, 5th edition, 1986.

[7] Gary Johnston and Roy H. Campbell. "A Multiprocessor Operating System Simulator". Technical Report UIUCDCS-R-88-1460, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, September 1988.

[8] Willy S. Liao, David M. Putzolu, and Roy H. Campbell. Building a Secure, Location Transparent Object Invocation System. In *Fourth International Workshop on Object-Orientation in Operating Systems*, Lund, Sweden, August 1995. IEEE Computer Society.

[9] Dennis M. Ritchie and Kenneth Thompson. The UNIX Time-Sharing System. *AT&T Bell Laboratories Technical Journal*, 57(6):1905, 1975.

[10] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference*, pages 103–114, San Francisco, California, April 1990.

[11] Bjarne Stroustrup. What is object-oriented programming. In *USENIX '87 C++ Workshop.* USENIX Association, November 1987.

[12] See-Mong Tan, David Raila, and Roy H. Campbell. An Object-Oriented *Nano-Kernel* for Operating System Hardware Support. In *Fourth International Workshop on Object-Orientation in Operating Systems*, Lund, Sweden, August 1995. IEEE Computer Society.

[13] 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.