C++ and Operating Systems Performance: A Case Study*

Vincent F. Russo, Peter W. Madany, and Roy H. Campbell
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue, Urbana, IL 61801

Abstract

Object-oriented design and programming has many software engineering advantages. Its application to large systems, however, has previously been constrained by performance concerns. The *Choices* operating system, which has over 75,000 lines of code, is object-oriented and programmed in C++. This paper is a case study of the performance of *Choices*.

1 Introduction

Proponents of object-oriented design and programming proclaim advantages including design and code reuse and rapid prototyping. Others hesitate to adopt the approach because of the cost of retraining developers, rumors of poor performance, and the lack of adequate tools for building large, efficient object-oriented systems. The C++ language[14] has been widely promoted as an object-oriented language with minimal overhead. In 1987, members of the *Choices*[4] research team set out to build an object-oriented multiprocessor operating system written in C++. This work represents both an attempt to validate claims made by proponents of object-oriented programming and an attempt to investigate object-oriented operating systems. The results of this ongoing experiment include the *Choices* operating system, which contains about 75,000 lines of C++ code written by a group of approximately 15 graduate students.

In this paper we will measure and analyze both the C++ source code and the performance of *Choices*. We will then compare both to a UNIX™ [2] system which provides similar features and performs similar tasks on the same hardware. The conclusion will show how the evidence from this research supports the claim that, despite the lack of tools for such work, it is possible to build efficient, high-performance object-oriented software in C++ and that such software can compete with commercial software.

2 Overview of *Choices*

Choices is a complete operating system, which runs stand-alone on a particular computer without depending on any proprietary software. Designed with object-oriented

*This work was supported in part by NSF grant CISE-1-5-30035 and by NASA grants NSG1471 and NAG 1-163.

techniques[3] and built using an object-oriented programming language, *Choices* provides a transparent object-oriented application interface to protected system resources. This interface allows dynamic method invocations on the collection of objects which constitutes the kernel of the system.

Choices has been designed as a framework that supports experimentation with modern operating system technology. In particular, *Choices* is multi-threaded for efficient multiprocessor use[11]. It offers low-level support for light-weight context switching from an application process to another that shares the same address space, to a system process, or to an interrupt process. The virtual memory system[12] uses machine-independent page tables, shared paged "segments" that may be simultaneously located in multiple virtual address spaces, and scatter-gather I/O for page fault handling. Each segment can have its own backing storage which may reside on various devices including a disk, a disk partition, a file, or physical memory. Distributed virtual memory[5] supports coherent sharing of data between applications over networks like Ethernet. The file system class hierarchy[7, 8] includes support for disks, partitions, and files and directories conforming to the System V UNIX[15], BSD 4.2 UNIX[9], or MS-DOS™ [10] standards. Programs can use read and write operations to access the files or files can be memory-mapped. Currently, *Choices* has networking support for Ethernet, UDP/IP, and TCP/IP. For users who want to run many existing programs, *Choices* provides a UNIX compatibility library. An X11™ windowing facility is planned.

The *Choices* design has been influenced by, but is not a reimplementation of, systems like UNIX[2] and Mach[1]. Instead, the goal of *Choices* is to further parallel high-performance multiprocessor research by exploiting the customizability of object-oriented systems.

Choices already runs on Encore Multimaxes™ and Apple Macintosh™ IIx's. Ports to the Intel Hypercube™, AT&T 6386 Work Group System™, and Hewlett-Packard Spectrum™ are underway.

3 Results

While it is difficult to evaluate the impact of a programming language or a style of programming on program performance, overall, it is the ease with which it is possible to construct an efficient software system that matters. Knowledge of the code that is produced by the compiler for particular programming constructs, the ability of the programmer, and the use of improved algorithms and data structures can mask the advantages or disadvantages of the language or style. Our evaluation of the source code for *Choices* will therefore be presented on its own without attempting to compare it to UNIX. We will analyze the effect of object-oriented design on code structure and reuse, including measurements of the size and depth of the class hierarchy, the lines of code and the number of methods per class, the number of methods inherited by subclasses, and the number of classes used across *Choices* layers and subsystems.

The calibration of our performance measurements is also complicated by the uniqueness of the design of *Choices* as an object-oriented system. But, like any operating system, *Choices* provides certain basic functions. We have therefore selected to measure the performance of attributes of *Choices* that have a close resemblance to features in UNIX and ignored other features. The measurements were made on an Encore Multimax shared

Function, Proxy, and System Call Overheads	
type	overhead in μs
C function call	3
C++ virtual function call	6
Choices proxy object call	79
Choices system call	81
UNIX system call	98

Table 1: Overhead of Function and System Calls

memory multiprocessor using the 10MHZ NS32332™ processor and a microsecond timer. The UNIX system used is Encore's 4.2 BSD UNIX system which includes optimizations for parallel processing. The reader should be aware that a direct comparison between *Choices* and UNIX may not always be meaningful. For example, we use modern algorithms that could be used to replace the older algorithms in UNIX to improve performance. We also use operating system implementations and features that may have been inappropriate at the time UNIX was designed because of the hardware technology then available. However, comparisons to UNIX can be used to put the *Choices* performance numbers in perspective.

Although our results do not allow one to infer the likely performance of an application running under *Choices*, nor that C++ and object-oriented programming is superior to C and standard coding practices, they do show that there is no inherent loss in performance. We believe that this result, when coupled with the other advantages of building object-oriented systems using object-oriented design, is significant.

The rest of this section includes the measurements of: procedure and method call overhead, system and proxy object call overhead, time-slicing overhead, various context switching times, page faulting handling time, memory allocation and deallocation times, and file read, write, open, create, close, and copy times. Before each measurement, we indicate why the measurement is important and how it was made. After each measurement, we discuss how the results should be interpreted.

3.1 Procedure Call Overhead

The first measurements we present provide the basis for some later comparisons and analysis. Table 1 compares the overhead between a C function call and a C++ virtual function call. The functions called in this table all take no arguments and return only a single integer. The measured overhead includes both the invocation and return of the call. The C++ function call is slower because of the extra indirection caused by the method lookup in the object's virtual function [14] table and the adjustment of the this pointer to account for multiple inheritance.

Table 1 also compares two different approaches implemented in *Choices* for transferring control from user to system protection mode. These mechanism allow user programs to request system services, much in the same way as the system call operates under UNIX. The examples shown implement a function that is similar to the "getpid()" system call in UNIX.

The first approach, which uses a proxy object call [13], allows a user program to invoke an operation on a system object by invoking a similar operation on a proxy object that

Time-slice and Time-slicing Overhead	
System function	overhead in μs
Choices time-slice (system + hardware)	234
Choices minimum time-slice	17
UNIX time-slice (system + hardware)	356

Table 2: Time-slicing Measurements

is in a user read-only region of virtual memory. This is the mechanism by which *Choices* applications request system services. The proxy object method causes a trap into supervisor state and, therefore, *Choices* kernel code. The kernel code validates the call and invokes the appropriate method on the intended system object.

The second approach measured emulates a UNIX system call. It is implemented by trapping into supervisor state and kernel code with a register containing the index of the service requested. This is the mechanism by which a UNIX compatibility library, which is designed to support migration to *Choices* from UNIX, requests *Choices* system services. The last entry in Table 1 shows the overhead for the UNIX getpid() system call, placing the previous two numbers in perspective.

The better performance of the proxy object call is a result of exploiting knowledge of the C++ virtual function calling convention. The implementation of the proxy object call avoids saving unnecessary context during the transfer from non-privileged to privileged execution. Both the two *Choices* schemes and the UNIX system call require a small amount of assembly code to be used in their implementation. The UNIX system call implementation does not contain a significantly different amount of assembly code and the getpid() service imposes about the same amount of overhead in either UNIX or *Choices*.

3.2 Time-slicing Overhead

Both UNIX and *Choices* are timesharing operating systems. Timesharing is implemented by running each program in the ready queue for a quantum of time called a time-slice. Programs are selected from the ready queue using a round robin discipline. Table 2 shows the overhead for implementing a time-slice. The measurements were acquired by time-slicing the execution of a single program that executes a loop and increments a counter 10 million times. For both *Choices* and UNIX, the timings were made on an otherwise idle system (i.e., the ready-queue was otherwise empty). Decreasing the size of the time-slice increases the running time of the program because of additional overhead. The additional overhead is created by the extra time spent trapping into the operating system when the timer expires, deciding which process to execute next, and finally rearming the timer and dispatching the new process. Using an equation relating the elapsed running time to the number of time-slices[13], and the running time with no time-slicing,[1] the measurements can be used to solve a set of simultaneous equations yielding the overhead per time-slice. Under *Choices*, the smallest time-slice that will allow the timer to be set and the user code to be resumed is also shown in Table 2.

The difference between the measurements can be explained by the implementation of

[1] Unlike *Choices*, time-slicing cannot easily be disabled in Encore UNIX.

Process Switching Overheads in μs				
Process type	Domain	System	Application	FP Application
System	same	88	163	176
Application	same	163	221	—
FP Application	same	176	—	244
System	different	—	289	315
Application	different	289	370	—
FP Application	different	315	—	412

Table 3: Process Context Switching Overhead

interrupts in *Choices*. In *Choices*, all interrupts are handled by immediately resuming a system coroutine which decides what to do next[13]. In the case of an empty ready-queue in *Choices*, the minimum overhead is the cost of the interrupt because the interrupted process will be resumed immediately when the system coroutine determines there are no other processes ready to run. The context switch between application and system coroutine and back from system coroutine to application requires a minimal amount of information to be saved and restored. Further, the interrupt handler invoked for the timer interrupt is used to reset the timer. The small minimum allowable time-slice is possible because the interrupt handler delays starting the timer until it has performed all its other book keeping.

To estimate the approximate cost of time-slicing when the ready-queue is not otherwise empty, the overhead above should be added to the cost of switching between processes discussed in Section 3.3.

3.3 Context Switching Overhead

Both UNIX and *Choices* support concurrent processes. Timesharing provides the abstraction that the concurrent processes execute at the same time. Under both Encore UNIX and *Choices* concurrent processes may also execute in parallel on the multiprocessor hardware. Often a task or user application will consist of several communicating concurrent processes. A process may block waiting for synchronization with another process. In this section we examine the overheads associated with concurrent processes.

Choices supports lightweight processes. An abstract Process class defines the notion of a process and subclasses define the behaviors of various specializations. Blocking and time-slicing are implemented using context switching primitives. These save and restore the processor state corresponding to a blocked or suspended process. In *Choices*, the context switching primitives used when a process is blocked or suspended depend on the classes of both that process and of the process that will next be run on the processor. When a system process is blocked and another system process is run, minimal context switching occurs. However, when an application using floating point is blocked and a different application is run, a larger overhead is incurred.

The *Choices* process differs from a UNIX process. Under UNIX, a process executes in its own virtual memory. Context switches between processes have a significant overhead and are "heavy-weight". To alleviate this problem, a separate "light-weight" thread package can be used in some systems. The package is not part of standard UNIX. Other

Virtual Memory Overhead	
system	overhead in seconds
Choices 1024, 4k pages	12.1
UNIX 1024, 4k pages	11.6

Table 4: Allocating and Zero Filling Pages

UNIX derivatives have kernels that include support for "heavy-weight" (normal UNIX) and "light-weight" processes. In *Choices*, it is the context switch that is light- or heavy-weight, not the processes themselves.

Table 3 shows some of the context switching overheads of *Choices* for different classes of process. The measurements were made between two processes using a loop in which each process relinquishes the processor to the other process. Unfortunately, it was impossible to repeat this experiment under UNIX since it does not provide primitives to relinquish the processor directly to another process. Also, since the source code of UNIX on the Encore Multimax was unavailable, we could not instrument UNIX to gather a similar performance measure.

Each *Choices* process executes within a virtual memory called a Domain. Several processes may share the same domain. The time for a context switch from one system process to another in the same domain is 88 microseconds. Such a context switch only requires the saving and storing of CPU registers used by the system code. The time for a context switch between two application processes that do not use floating point running in the same domain is 221 microseconds. The full set of registers used by an application is saved. When floating point is used, the overhead increases to 244 microseconds as a result of saving and restoring the floating point registers. The context switch between floating point application processes in different domains is 412 microseconds. The additional overhead incurred when the domain differs corresponds to flushing the MMU cache and reloading the page tables for the new domain.

3.4 Virtual Memory Overhead

Both *Choices* and Encore UNIX have a paged virtual memory. In a paged virtual memory, fixed sized "pages" of addresses in virtual memory are bound to blocks or "page frames" of physical memory. The contents of virtual memory are stored on a backing store until they are needed by a program. A page fetch transfers a page of data from backing storage to physical memory. On our Encore Multimax, both UNIX and *Choices* have the same overhead of 25 milliseconds for a page fetch. Details of the *Choices* implementation of virtual memory are discussed elsewhere [12]. The overhead is dominated by the access time of data on the disk used for backing storage.

The disk overhead can be eliminated by considering the allocation and filling of new pages in a virtual memory. Here an application references data that is not yet stored on a backing store. The system creates the data being referenced by allocating physical memory to the page being addressed and filling that page with zeros.

In the measurements shown in Table 4, four megabytes of virtual memory are created and zeroed by accessing the first word of each page. In both the *Choices* and UNIX implementations, the page size is 4096 bytes, thus both systems create 1024 pages. The results show that the *Choices* overhead is nearly the same as UNIX, even though the

Memory Allocation times in μs		
Operation	*Choices*	Encore UNIX
Allocate	34	54
Free	39	16
Total	73	70

Table 5: Memory Allocation Measurements

Choices code has yet to be tuned.

3.5 Memory Management Overhead

While dynamic memory management is important in C[6] programming, it is essential to object-oriented programming in C++. In C, the usual memory management operations are malloc and free, whereas in C++, they are new and delete. Using the new and delete operations, we measured the average overhead for creating and deleting objects of various sizes ranging from 32 to 4096 bytes. Table 5 contains the results of our measurements. Both systems use a similar algorithm for memory management, and both systems incur similar overhead. However, the *Choices* allocator adds several important features. These features include alignment, space efficiency, and support for execution on parallel processors. The UNIX allocator aligns every object on an eight-byte boundary; however, it never aligns objects on page boundaries. The *Choices* allocator aligns all objects larger than one page on page boundaries. Such an alignment policy supports more efficient usage of virtual memory and direct-memory-access (DMA) devices. The UNIX allocator stores state information immediately preceding the blocks it allocates. This extra information, combined with the allocation algorithm, results in twice as much storage being reserved for objects whose sizes are powers of two, sizes that are common in computer systems. On average, the UNIX allocator uses 50 percent more space than the *Choices* allocator. The UNIX allocator was written to be run within a single process. The *Choices* allocator was written to support the requests of multiple processes running on multiple processors; therefore, it uses spin-locks to prevent corruption of its internal data structures by simultaneously executing processes. Almost half of the time the new and delete operations take in the *Choices* allocator is spent locking and unlocking these data structures. By optimizing the other aspects of memory allocation, the *Choices* allocator achieves similar performance to the UNIX allocator, even though it supports parallelism.

Since C++ does not support automatic garbage collection of unneeded objects, *Choices* uses reference counts. Because the reference and unreference operations are performed extremely frequently within the *Choices*, we optimized their performance. Currently it takes only 25 microseconds to reference and unreference an object. When an object's reference count reaches zero, the object is automatically deleted.

3.6 File System Performance

Choices provides stream-oriented file systems that conform to operating system standards such as 4.3 BSD UNIX, System V UNIX, and MS-DOS. The file system class hierarchy also supports the construction of customized and experimental file systems. Various instances of file systems can coexist and interoperate in a running *Choices* system. Because

File open, create and close in μs			
Operation	Cached	*Choices*	Encore UNIX
Open existing file	NO	32173 ± 62	28812 ± 241
Open existing file	YES	4163 ± 86	2722 ± 14
Open currently open file	YES	2593 ± 75	2067 ± 86
Create new file	NO	29854 ± 939	25546 ± 1991
Close file	NO	72208 ± 2257	80303 ± 22233

Table 6: File System Operation Measurements

the BSD file system is the most efficient of the systems that currently can be built from our hierarchy, and because it uses the same on-disk data structures as Encore's version of UNIX, we have chosen to measure its performance.

Within a disk-based computer system, disk latencies dominate file operation times. To reduce these delays, file systems use various *caching* techniques, such as the *buffer cache* used in UNIX[2] and the *memory-mapped files* used in *Choices*. A buffer cache allows the file system to keep copies of many of the most recently used disk blocks in physical memory. Since recently accessed blocks are more likely to be reused, the cache can greatly reduce the cost of reading and writing data blocks. The UNIX buffer cache is implemented in software. It uses a least-recently-used buffer replacement algorithm and hashing to map disk block numbers to buffer addresses. In contrast, *Choices* allows the file system to reuse the page replacement algorithms of its virtual memory management system. Instead of using a software mapping, *Choices* uses the virtual memory hardware to map requests for disk blocks to buffer addresses.

Because caching often speeds up file operations by a factor of ten, we measured operations both when the operation generated a cache *miss* and a cache *hit*. To make comparisons more significant, we used the same amount of physical memory, two megabytes, for caching disk blocks in both systems, and we tested the operations in the same order on each system. Also, because disk latencies vary from access to access, we repeated each test several times and report the mean value of each measurement and the 95% confidence interval for the mean.

To use the file system, an application program must first gain access to files via open or create operations. Table 6 contains measurements of the time it takes to open existing files and create new files. The open operation uses both the current directory to convert a file name to an *inumber* and an in-core *inode* to convert the inumber to a reference to an open file object. *Choices* takes slightly longer than UNIX to open files, regardless of whether the disk block describing the file is cached. The reason *Choices* takes longer is that is builds a caching object that it uses to map the file into memory. We expect that this small amount of extra overhead for file opens will be amortized over the entire time the file is open. The create operation is similar to the open operation; we chose to only measure an uncached create, since *Choices* flushes modified directory blocks to the disk after a file is created. Again, the creation time of a caching object accounts for the difference between creating files for *Choices* and for UNIX. We also measured the close operation for a file opened in *read-only* mode. The times are similar for both systems. The reason the close operation takes 70 to 80 milliseconds is that the in-core inode structure must be written back to the disk, even if the file has not been modified.

Read, write, and lseek times in μs			
Operation	Cached	*Choices*	Encore UNIX
Read block direct	NO	26803 ± 420	33002 ± 1275
Read block direct	YES	2524 ± 106	3784 ± 128
Read block indirect	NO	58841 ± 4876	53457 ± 769
Read block indirect	YES	2726 ± 294	4358 ± 219
Write block direct	YES	3752 ± 207	3884 ± 324
Write block indirect	YES	3168 ± 23	4324 ± 306
Lseek	—	111 ± 5	194 ± 6

Table 7: File Operation Measurements

The most important operations on open files are read and write. Measurements of these operations are given in Table 7. Before blocks can be read or written, logical block numbers must be mapped to physical blocks numbers using the data stored in an inode structure. Inodes organize this block mapping information into a variable level tree.[2] We measured I/O operations using both direct blocks and for single-indirect blocks.[3] All the I/O measurements reported in Table 7 are for reads or writes of 8192-byte *aligned* blocks. For cached read and write operations, *Choices* performs better, since it uses virtual memory hardware to map disk block number to buffer addresses. For uncached read and write operations, *Choices* and UNIX perform similarly, since both systems must perform disk I/O and update mapping information.

The lseek operation, which repositions the stream file location pointer, is essential for randomly accessed files. Table 7 also reports the overhead of the lseek operation. *Choices* performs lseeks faster primarily because it provides a more efficient system call mechanism (see section 3.1).

The interactions between various file system operations can often lead to unanticipated results. Therefore, we not only measured the times of individual operations, but we also measured the time of performing a common series of operations: copying an entire file. For this test we chose to copy a one megabyte file; we measured both the time to copy the data blocks from disk-to-disk and from cache-to-cache. Table 8 shows the results of these tests. For disk-to-disk copies, *Choices* performs slightly faster, largely owing to the efficiency of the *Choices* caching mechanism. For cache-to-cache copies, *Choices* takes less than half the time, again owing to the efficiency of the *Choices* caching mechanism. *Choices* also provides a single operation, copy, to copy an entire file. By avoiding the overhead of making many (256) system calls, *Choices* provides a substantially faster file copy mechanism.

4 Code Structure and Reuse

Choices is written as an object-oriented system. It has 281 classes of which 190 are subclassed from class Object and 90 are miscellaneous support classes. Many of the 90

[2] In BSD UNIX, this tree can have 0, 1, or 2 levels of indirection. In System V UNIX, this tree can have up to 3 levels of indirection.

[3] Double and triple-indirect blocks are seldom used.

Copy one megabyte files in seconds			
Block size	Cached	*Choices*	Encore UNIX
8192	NO	8.167 ± 0.02	8.542 ± 0.11
8192	YES	.906 ± 0.02	2.019 ± 0.11
1048576	YES	.562 ± 0.03	—

Table 8: File Copy Measurements

Classes and Methods in Levels of *Choices* Hierarchy			
Level	Classes	Public Methods	Protected Methods
1	1	14	0
2	37	197	104
3	56	386	71
4	46	304	60
5	31	143	98
6	18	122	63
7	2	8	1
All levels	191	1174	397

Table 9: Class Hierarchy Characteristics of *Choices*

classes are introduced to hide machine dependent details. Most of the 190 subclasses are specializations of several abstract classes. Table 9 shows the number of classes that are subclassed from class Object and the depth that they occur within the *Choices* hierarchy. The average depth of a class within the hierarchy is 3.7. Only 10% of the classes have a depth greater than 5. The results support other evidence that inheritance has been used within *Choices* to structure and reuse the code. The performance measurements of *Choices*, as shown in the previous section, indicate that any overheads incurred by method lookup and the object-oriented organization of the system appear to be trivial compared with the use that was actually made of subclassing in the construction of the system.

Table 10 shows the organization of *Choices* code into files of C++ and assembler. Just over half of the files used by *Choices* contain header information. Less than 11% of the files contain machine or processor dependent information. Table 11 shows the organization of *Choices* in terms of the lines of code written in C++ and assembler. Assembler code represents 0.7% of the total number of lines of code, and this is isolated to just ten files. Header lines of code represent 34% of the total number of lines of code. The large number of header files in comparison with the number of lines of code they contain is

Files of C++ and Assembler Code in *Choices*			
Type	".h" files	".c" files	Total files
Entire *Choices* source code	286	232	518
Containing machine dependent code	13	19	32
Containing processor dependent code	10	12	22
Containing assembler code	0	10	10

Table 10: Source File Characteristics of *Choices*

Lines of C++ and Assembler Code in *Choices*			
Type	".h" lines	".c" lines	Total lines
Entire *Choices* source code	27,012	51,768	78,780
Machine dependent code	2,200	5,635	7,835
Processor dependent code	1354	4,434	5,788
Assembler code	0	555	555

Table 11: Code Characteristics of *Choices*

a consequence of trying to organize *Choices* for easier maintenance and less compilation interdependencies.

5 Conclusions

Despite the scarcity of support tools, C++ is a powerful language in which to prototype high-performance systems using an object-oriented approach. The characteristics of object-oriented systems have reduced the effort needed to make major changes in the design and class hierarchies of these prototypes. Classes and inheritance have yielded much code reuse, both within and between components and subsystems. Several developers have been able to work together while avoiding some of the usual pitfalls inherent in large software projects.

Our evidence shows that the benefits of using an object-oriented language for coding system algorithms outweigh the slight overhead of C++ method calls. Many of the *Choices* subsystems outperform and none are significantly slower than their UNIX equivalent, even though the UNIX system used for comparison is optimized for a multiprocessor. Despite the lack of performance analysis tools and the difficulties of making fair comparisons, we encourage others to experiment with building high-performance systems using object-oriented design and programming.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer USENIX*, July 1986.

[2] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[3] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of Object-Oriented Operating System Design. Technical Report UIUCDCS-R-89-1510, University of Illinois at Urbana-Champaign, April 1989.

[4] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, Santa Fe, New Mexico, November 1987.

[5] Gary Johnston and Roy H. Campbell. An Object-Oriented Implementation of Distributed Virtual Memory. In *Proceedings of the USENIX Workshop on Distributed and Multiprocessor Systems*, pages 39–58, September 1989.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[7] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 311–328, Nottingham, UK, July 1989. Cambridge University Press.

[8] Peter W. Madany, Douglas E. Leyens, Vincent F. Russo, and Roy H. Campbell. A C++ Class Hierarchy for Building UNIX-Like File Systems. In *Proceedings of the USENIX C++ Conference*, Denver, Colorado, October 1988.

[9] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[10] Peter Norton. *The Peter Norton Programmer's Guide to the IBM PC*. Microsoft Press, 1985.

[11] Vince Russo, Gary Johnston, and Roy H. Campbell. Process Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '88*, San Diego, CA, September 1988.

[12] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '89*, New Orleans, Louisiana, September 1989.

[13] Vincent F. Russo. *The Design and Implementation of an Object-Oriented Operating System*. PhD thesis, University of Illinois at Urbana-Champaign, April 1990 (in preparation).

[14] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1986.

[15] K. Thompson. Unix implementation. *Bell System Technical Journal*, 57(6):1931–1946, July 1978.