A C++ Class Hierarchy for Building UNIX-Like File Systems*

Peter W. Madany, Douglas E. Leyens
Vincent F. Russo, and Roy H. Campbell

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801

Abstract

Class hierarchical object-oriented programming languages like C++ facilitate the construction of organized libraries of related data structures and algorithms. In operating systems research, it is convenient to build such libraries to support system abstractions. In our Choices [3] parallel operating systems research, we have been experimenting with new and existing file system facilities in an attempt to design an object-oriented file system implementation.

This paper describes a classification of the data structures and algorithms used in UNIX-like file systems and an implemention of them using C++. We present a class hierarchical organization for the System V [8] and 4.2 BSD [4] file systems that reflects the common subcomponents, abstractions, and interfaces that these systems share. Because of the flexibility afforded by designing such systems in an object-oriented language, new specializations of the abstract file system can mix and match components from existing implementations forming hybrid systems.

We conclude by discussing the performance of our system and the influence of C++ on our design and organization.

1 Introduction

This paper describes an experiment in the classification and implementation of data structures and algorithms used in UNIX-like file systems. Our long-term purpose is to provide

*This work was supported in part by NSF grant CISE-1-5-30035, by NASA grant NSG1471, and by AT&T ISEP.

a foundation for further research into object-oriented file systems; however, the immediate goal is to combine the UNIX philosophy of file systems and the Choices philosophy of object-oriented operating system components.

In the following three subsections we will describe Choices, the System V file system, and the 4.2 BSD file system extensions.

1.1 Choices

Choices is a family of operating systems that can be customized to a particular multiprocessor or parallel application [1] [2]. Object-oriented programming and class hierarchies are used to facilitate the building and customization of the family. C++ was adopted as the programming language because it provides an efficient implementation of objects and classes [7].

A Choices system is an object-oriented operating system that uses persistent objects to provide facilities and services to client processes. Choices persistent objects have lifetimes independent of user processes. Many of these subsystem facilities and services would belong in the kernel of a more "traditional" operating system. However, persistent objects allow an application to load only those subsystems that it needs. Persistent objects can provide secure services because Choices uses virtual memory protection mechanisms to restrict access to the objects.

The file system is one of the more important subsystems provided in an operating system. In Choices, we have chosen to implement the file system as a collection of persistent objects; each persistent object implements an independent component of the file system. Using this technique, an application may use a file system composed of many different components, each tailored to improve the performance of the application, to optimize the use of the storage technology, or to provide compatibility with file systems of other operating systems.

Currently, we have completed two different UNIX file systems: the 4.2 BSD file system [4] and the System V [8] file system. The classes of the two file systems are specialized from one, abstract, UNIX-based file system class hierarchy. However, many of the concrete classes realizing the two systems are very different from one another. Further, it is possible to combine file system components from UNIX BSD and System V implementations to produce hybrid systems that combine the features of both. For example, the efficient BSD disk allocation methods and larger block sizes can be combined with the System V directory structure to yield a system with higher throughput without having to rewrite any user level code that relied on a System V record structure for directories. Alternately, individual features such as symbolic links or disk quotas may be added to the System V file system, as needed.

C++ has been an aid in developing our file system implementations. The language was useful because it directly supported the development of the abstract classes that formed the framework for our systems. The abstract interfaces permitted concurrent development and debugging of the various components of each file system. The virtual function feature allowed us to simplify much code. The ease of developing and reusing C++ classes led to

much code reuse, both within a file system and between different file systems.

1.2 System V

The System V [8] file system is the standard file system model found in today's commercially available UNIX systems. Its design is dominated by simplicity.

Basically a user program can view a UNIX file as a sequence of randomly accessible bytes. All files can be accessed via the same standard interface: `read`, `write`, and `lseek`. This interface conceals hardware device dependencies and hides block allocation and block mapping. Because the operating system does not impose record structures on files, the output of most UNIX tools can be the input of others. Nevertheless, any tool can impose a structure on a file. Efficient implementation of random access allows even complex record structures, such as ISAM, to be imposed on specific files when needed.

Disk drives in UNIX systems are divided into logical sub-devices, called partitions, each of which contains one file system. A file system consists of a header for the system called a superblock, information about which disk blocks are available for allocation, and an array of inodes that describe individual files. While file systems cannot span disk partitions, a single directory tree contains all the files on all the file systems. The directory tree hides individual disks and partitions from the user.

The inode is a structure that describes an individual file and and manages access that file. Within a UNIX system, a file can be uniquely identified by specifying its partition and the inode array index number, called the inumber. An inode contains its file's size, reference count, ownership, access rights, timestamps, and the numbers of the blocks which hold the file's data.

Directories are sequences of records that contain (*name*, *inumber*) pairs. Because directories contain inumbers instead of complete inodes, files can appear in more than one directory at a time. Files are only deleted when their reference count reaches zero.

The System V file system's performance is marked by two impressive characteristics: high disk space utilization and low CPU overhead per block transferred. However, there are some deficiencies in both its performance and feature set that have been addressed by the design of the 4.2 BSD file system.

1.3 BSD

The 4.2 BSD file system [4] maintains the same basic interface as System V and adds optimizations and extensions.

The penalty incurred by the System V file system per individual block transferred is small. Its overall throughput is dominated by disk latency. To minimize the disk latency and thereby improve overall throughput, the 4.2 BSD file system increased file block sizes and improved inode and disk block allocation policies.

An 8192 byte block improves throughput almost sixteen times when compared to a 512 byte block. To maintain the high disk space utilization of System V, 4.2 BSD added the

capability to fragment the last block in a file. The improved disk block and inode allocation policies minimize both disk head seek time and rotational latency.

Three of the major extensions provided by the BSD file system are symbolic links, long file names, and per-user disk quotas. Symbolic links allow users to create directory entries which refer to files on different file systems. In System V, file names are restricted to 14 characters because of the fixed-size record structure used for directory entries. The 4.2 BSD file system uses a variable-size record which allows file names to be up to 255 characters long. Disk quotas allow system administrators to restrict individual users to using only a portion of the space in a file system.

The following sections discuss the class and instance hierarchies in our system and are followed by discussion of performance and directions for future work.

2 A Class Hierarchy for File Systems

The use of *class hierarchies* has been proposed as a solution to some of the traditional design and engineering problems in today's software development lifecycle [6] [5] [7]. In particular, class hierarchies support code reuse and the sharing of common interfaces among different implementations. A class in a class hierarchy encapsulates an interface and a possibly empty implementation. The interface, or *signature*, of a class is defined by the set of *methods* or *operations* the class defines for its instances. The implementation of the methods of a class can either be defined by the class itself or can be defined by other classes that are derived from the class through class inheritance. [1] A class in a hierarchy can define or augment an interface, an implementation, or both. Classes that define only an interface and have subclasses that supply implementation are *abstract* classes. Subclasses that define an implementation for a particular interface are termed *concrete* classes. Most classes are neither concrete nor abstract; they often redefine only a portion of an implementation or augment an interface with a few additional methods. A subclass can customize an implementation of a superclass for specific applications and may share all, some, or none of its implementation with its superclass. Class derivation provides a framework for changing specific parts of a system without altering the whole structure.

The following sections describe the majority of the classes in a hierarchy to implement UNIX-like file systems. Figure 2 shows this hierarchy.

[1] The classes that have methods that are inherited are usually termed *parent* or *super* classes. The classes that inherit methods are usually termed *derived* or *sub* classes.

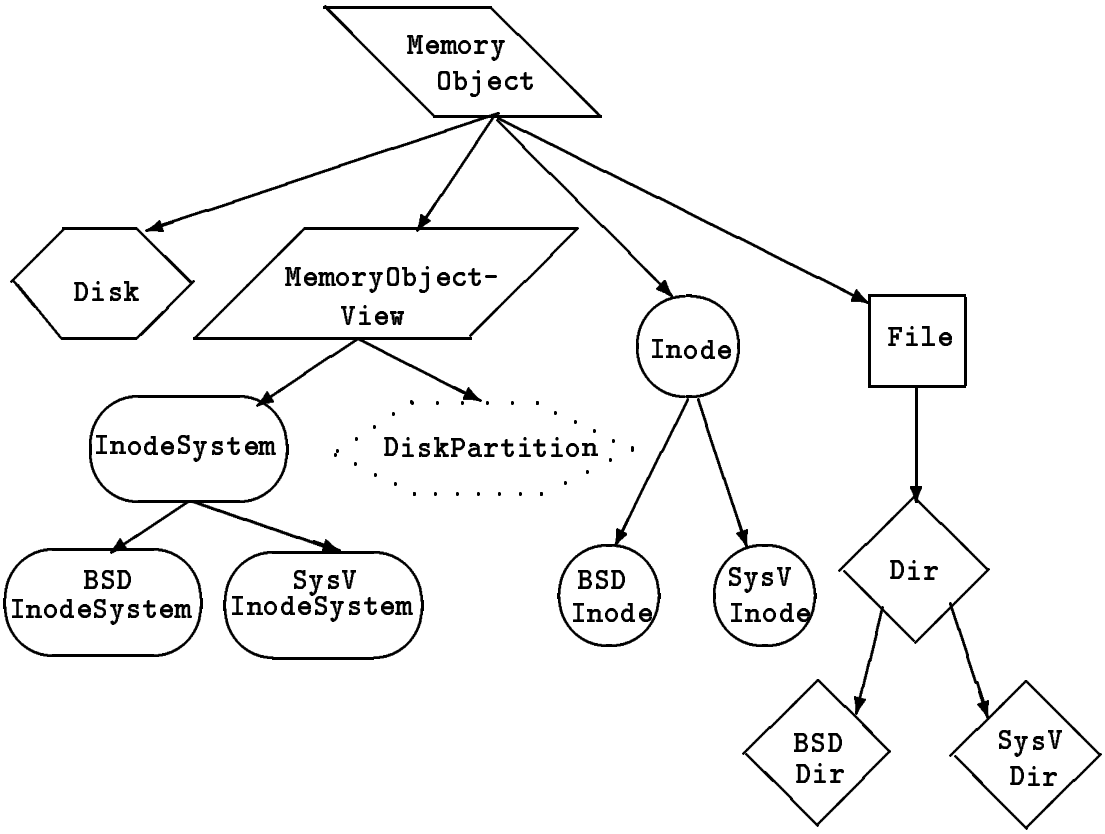

Figure 2: MemoryObject Hierarchy

2.1 MemoryObject

The superclass *MemoryObject* abstracts both the Choices file system and memory management systems. It defines an interface which permits access to a block of data that may either reside on permanent storage or be generated dynamically. The interface uses a read-unit/write-unit protocol. The units used for reading and writing are all the same size within an individual MemoryObject, and this size must be an integer power of two. Subclasses of MemoryObject augment the protocol and provide various implementations of the methods involved.

In Choices, MemoryObjects are most often accessed by mapping them into a process' virtual address space. The system caches portions of the MemoryObject into physical memory and provides the address translation mechanisms necessary for the process to address it with the read/write instructions of the CPU. A MemoryObject can, however, be accessed directly by its `read/write` interface.

Class	Common Public Methods					
MemoryObject	*read*	*write*	*close*	*open*	*create*	*synchronize*
↑ Disk	read	write	–	–	–	–
↑**Inode**	read	write	close	–	–	–
↑↑BSDInode	↑	write	↑	–	–	–
↑↑SVInode	↑	↑	↑	–	–	–
↑**MemoryObjectView**	read	write	–	–	–	–
↑↑DiskPartition	read	write	–	–	–	–
↑↑**InodeSystem**	↑	↑	–	*open*	*create*	*synchronize*
↑↑↑BSDInodeSystem	↑	↑	–	open	create	synchronize
↑↑↑SVInodeSystem	↑	↑	–	open	create	synchronize

Class	Protected Methods				
↑**Inode**	mapUnit	*getDirect*	*getIndirect*	*setDirect*	*setIndirect*
↑↑BSDInode	↑	getDirect	getIndirect	setDirect	setIndirect
↑↑SVInode	↑	getDirect	getIndirect	setDirect	setIndirect

Class	Protected Methods						
↑↑**InodeSystem**	get	put	*free*	*allocate*	readDinode	writeDinode	*getFreeInode*
↑↑↑BSDInodeSystem	↑	↑	free	allocate	↑	↑	getFreeInode
↑↑↑SVInodeSystem	↑	↑	free	allocate	↑	↑	getFreeInode

Legend	
Symbol	Meaning
Boldface	Abstract class.
Italics	Abstract definition of method.
Roman	Concrete class or method.
↑	Subclass or inherited method.
–	Undefined method.

Table 1: MemoryObject Class Hierarchy.

2.2 MemoryObject Subclasses

The following paragraphs discuss individual subclasses of MemoryObject and their particular functionality. Table 1 and Table 2 show the class hierarchy using the format introduced in [3].

The *Disk* subclass of MemoryObject represents the physical disk devices in a system. It provides an abstract interface and access protocol to these disks. It is further subclassed for specific hardware architectures and devices.

It is usually inconvenient or inefficient to copy a MemoryObject into virtual memory. The *MemoryObjectView* subclass of MemoryObject provides a window into another MemoryObject. The size of this window can range up to the size of the MemoryObject being viewed. The window may be offset from the start of the MemoryObject. Its purpose is to restrict access to the MemoryObject under the window. Several MemoryObjectViews may exist for the same MemoryObject.

A *DiskPartition* is simply an instance of MemoryObjectView that windows a sub-range of a Disk. The size and offset of the window is defined by the Disk's hardware partition table.

The *InodeSystem* class is derived from MemoryObjectView and inherits its `read` and `write` methods. One InodeSystem exists per DiskPartition and contains a UNIX file system. The InodeSystem is an abstract class definition that provides the framework for UNIX-like file systems. It contains the code for all methods that have the same implementation in the derived classes. All common methods are implemented in this class to reduce the overall code size and programming effort. The other methods defined here are needed by the subclasses, but since they will be different, they cannot be inherited.

The two major subclasses of InodeSystem implemented are *BSDInodeSystem* and *SVInodeSystem*. Many of the methods of BSDInodeSystem and SVInodeSystem perform identical functions but use different data structures or algorithms. The class InodeSystem contains the code common to both the BSDInodeSystem and the SVInodeSystem. It also provides virtual functions for methods that are implemented differently in these subclasses. For example, inumbers must be mapped to physical blocks by the `readDinode` and `writeDinode` methods. A `mapInumber` method is defined as a virtual function in the InodeSystem. Each subclass implements this method in a different way. However, both the `readDinode` and the `writeDinode` methods can be implemented in the InodeSystem and this implementation can be inherited by the subclasses. The `readDinode` and `writeDinode` methods of BSDInodeSystem and SVInodeSystem use the implementation of `mapInumber` that is appropriate to the subclass of the instance upon which the methods are invoked. Similarly the `get` and `put` methods are inherited but need a variable containing the fragment-to-sector conversion factor to be appropriately initialized by the derived class. Such techniques move general code up into the base class where it can be reused instead of requiring it to be rewritten for each new implementation.

Those methods that are sufficiently different between various subclasses of InodeSystem (types of UNIX file systems) are simply defined as empty virtual functions in InodeSystem and redefined by *all* subclasses. For example, the System V superblock contains a free list for

File Class Hierarchy							
Class	Methods						
↑ File	read	write	seek	close	–	–	–
↑↑ **Directory**	*read*	–	–	–	*put*	*locate*	*remove*
↑↑↑BSDDirectory	read	–	–	–	put	locate	remove
↑↑↑SVDirectory	read	–	–	–	put	locate	remove

Table 2: File Class Hierarchy.

both free data blocks and free disk inodes while the BSD system uses bitmaps. `Allocate`, `free`, and `getUnusedInode` have sufficiently different implementations that they cannot share code, only an interface.

The *Inode* is an abstract class that provides a framework for a UNIX-like in-memory inode object. As in the InodeSystem, common code is moved into the base class and inherited by the BSD and System V derived classes. These methods are mostly private methods used to calculate disk block pointers and and manage internal caches of indirect blocks. There is also a method, `mapUnit`, that maps logical block numbers to physical file system block numbers and can be inherited by both derived classes. The remainder of the class defines the framework to be used by the derived classes.

The *BSDInode* and *SVInode* subclasses implement the Inode framework according to their particular needs. The differences are due to the ways in which data block pointers are stored, and the other fields in the disk inode structure. For example, methods to set and retrieve the direct and indirect pointers are implemented by each subclass. System V has 10 direct pointers, a single, a double, and a triple indirect pointer. Each of these is stored in three bytes in the disk inode and must be converted to and from an integer. BSD, on the other hand, has 12 direct pointers, a single, a double, and a triple indirect pointer. Each of these is stored as a four byte integer requiring no conversion.

From the user's perspective, an important subclass of MemoryObject is *File*. The File class is both a concrete class used for interaction with any UNIX disk file and an abstract class from which the *Directory* class is derived. The unit size for the File class is one byte. The File class adds the concept of a current file location pointer to the MemoryObject interface and adds the `seek` method to position this pointer. The `read` and `write` methods update this file pointer as well. These methods together provide a byte-oriented interface to user level programs. Each instance of File communicates with a corresponding Inode object which reads and writes blocks instead of bytes.

The Directory class is an abstract subclass of File which adds a directory-entry record structure on top of the blocks supplied by the Inode object. It also provides methods to simplify the insertion, retrieval, and removal of directory entries. Since directories in BSD and System V are different, the methods of this class: `read`, `put`, `locate`, and `remove`, must be defined by each subclass and cannot be inherited.

3 An Instance Hierarchy for File Systems

In this section, we describe the instance hierarchy for a complete working file system.

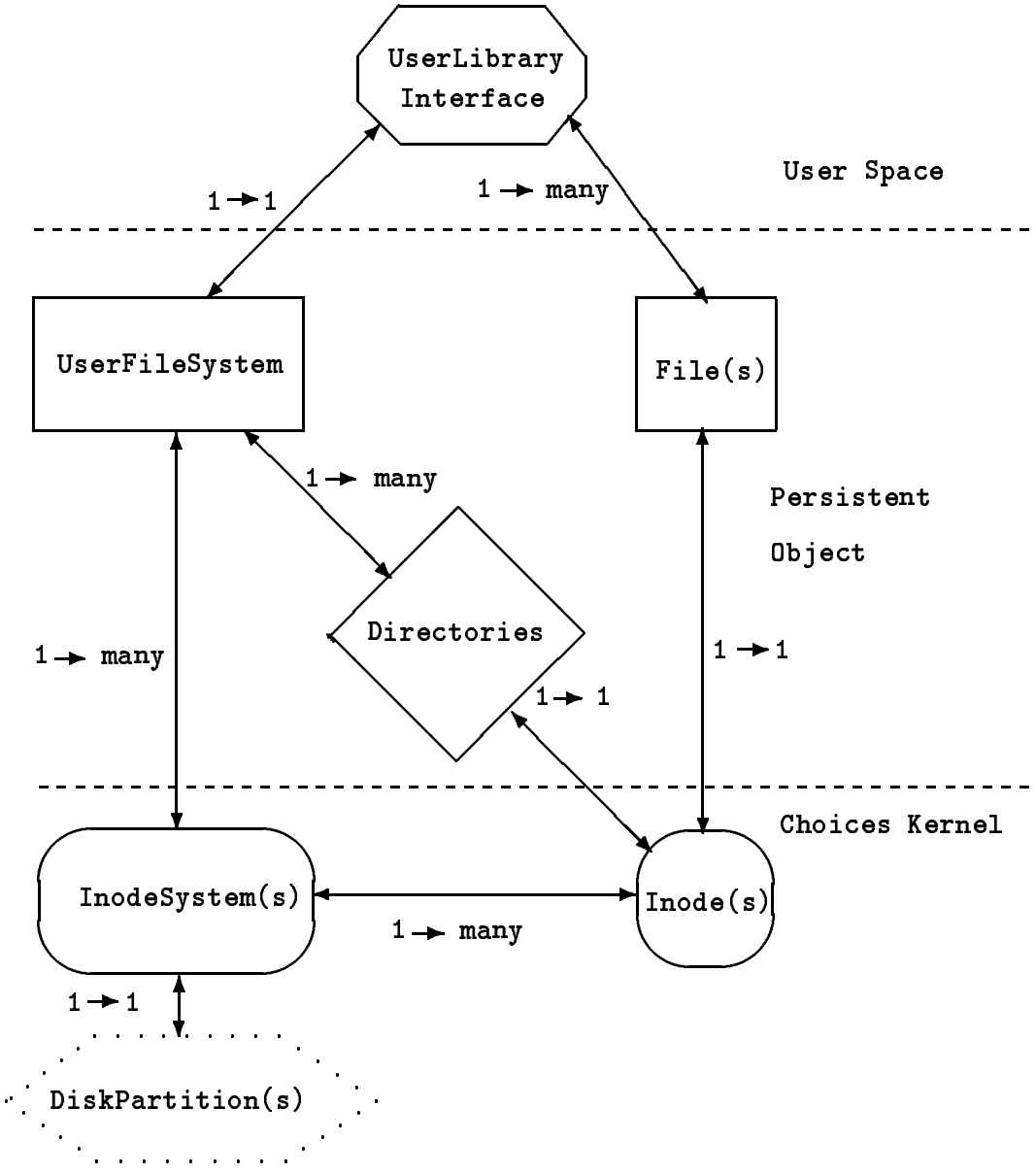

Figure 3: File System Instance Hierarchy

When performing operations on files, user programs must invoke the methods of the User Library Interface. These methods will in turn invoke methods on several other objects in order to perform the requested action. Figure 3 shows the objects involved in these operations and the basic data flow between them. Some sets of objects have a one to one

correspondence; for example, there is one Inode object for each File object. Other sets of objects have a one to many relationship, such as the UserFileSystem which can communicate with several InodeSystem objects.

A user program gains access to all file systems and files via persistent object calls. This can be likened to the system calls used to gain access to a UNIX file system. These calls will be translated into the appropriate method invocations in either the UserLevelFileSystem or a File object. In Figure 3 the set of these calls is referred to as the User Library Interface.

The *UserFileSystem* object views all active file systems as a single tree. A reference is maintained for each InodeSystem in this tree. The UserFileSystem also contains instance variables for pathname resolution that maintain references to the root directory and the current directory. These are needed to correctly implement the operations required of the UserFileSystem.

The public methods of the UserFileSystem are similar to several of the UNIX file system calls including: **`open`**, **`creat`**, **`link`**, **`unlink`**, **`mkdir`**, **`chdir`**, and **`stat`**. These methods operate on and return references to File objects and Directory objects which may in turn be used by the User Library Interface to perform operations on File objects.

The File object corresponds to a UNIX open file table entry and provides a generic interface to open files for user level programs. Its methods support operations similar to the set of UNIX system calls that operate on open files including: **`read`**, **`write`**, **`seek`**, and **`close`**. The File object communicates directly with its corresponding Inode object and maintains a current byte offset for implementing **`seek`** and sequential **`read`** and **`write`**.

An instance of Directory is used to impose the directory record structure on a file. Directory methods include **`read`**, **`put`**, **`locate`**, and **`remove`**. Read returns a directory entry and is used by programs such as **ls**. Directory entries are added and removed from the Directory object's underlying file via the **`put`** and **`remove`** methods. The **`locate`** method finds the inumber of an indicated file name in a directory. All of these methods are invoked by the directory methods of the UserFileSystem. User programs are prevented from executing the **`put`** and **`remove`** methods on a directory. The protection is provided by setting the file access mode as opposed to using protected C++ functions.

An instance of an InodeSystem is used for each active file system. Creation of a new instance is similar to the UNIX **mount** system call. The UserFileSystem communicates with the InodeSystem when requesting operations on new and existing Inodes. The InodeSystem communicates with the *DiskPartition* to read and write disk blocks. It also manages the superblock fields, disk inode allocation, and disk data block allocation. It creates and provides the Inode objects when requested and keeps track of in-memory versions of the corresponding disk inode structures.

The public interface to the InodeSystem includes methods that operate on and return references to Inodes. These are **`open`** and **`create`**. The interface also includes methods to maintain the data blocks of the DiskPartition for use by the Inodes. These methods are **`allocate`**, **`free`**, **`get`** and **`put`**. The **`synchronize`** method is used to write the modified superblock and in-memory versions of disk inodes to the DiskPartition to maintain consistency.

The Inode object contains the UNIX disk inode structure and the methods used to operate on it. These include all information needed to access the file such as size, mode, protection, ownership, and disk block pointers. Once this object is created by the InodeSystem, it may be referenced by a File or Directory object to perform actions on blocks of data stored within the InodeSystem's DiskPartition.

The DiskPartition object maintains the size and starting block location of the partition it represents. It performs disk read and write requests for its corresponding InodeSystem object and checks these requests to ensure that they only access blocks within the range that it manages.

3.1 Choice'ing a File System

Choices supports the concept of customizable operating systems. The file system class hierarchies presented allow a system designer to choose and easily integrate existing, modified, or new concrete components to create new customized file systems.

This mix-and-match approach leads to the following orthogonal "choices" when designing a new file system:

- Fixed or variable-sized file names.
- Per user disk quotas.
- Optimized inode and disk block allocation.
- Large block sizes and fragmented blocks.
- Symbolic links.

Some of these choices involve the selection of a complete specialized subclass, while others simply require creating new concrete subclass with methods from two existing subclasses.

The following section presents performance data measured from our implementation.

4 Performance

The performance of our systems can be characterized in three ways. First, we measured the overhead incurred by all of the Inode and InodeSystem methods as opposed to raw disk reads and writes of the same disk blocks. Second, we checked to make sure that our BSD implementation did not reduce or remove the effectiveness of the BSD optimizations. Third, we observed the effect of altering certain parameters and algorithms used in the Inode class methods.

To calculate the amount of CPU-time needed by all of the Inode and InodeSystem method code, we measured the time to copy a 17 Megabyte file, and then measured the time it took to do a raw disk copy of the same blocks using a simple iterative loop. Both copies were

performed in the kernel. The raw disk copy took 127 seconds, whereas the copy that used the Inode and InodeSystem code took 133 seconds. Therefore, only 5% of the time spent in the kernel while copying a file accounts for all of the block allocation and block mapping code in the Inode and InodeSystem class methods. This corresponds to the System V design goal of low overhead per block transferred.

When designing the interface between the Inode and InodeSystem classes, we took care to ensure that both the BSD block and inode allocation policies were fully supported. We also fully implemented the large block size and the block fragment features of the BSD file system. Therefore, the performance improvements that the BSD optimizations brought to UNIX will also be realized when using the BSD specialized classes under Choices.

After developing the file system code, we measured the effects of altering the block size on the time it took to copy files. Each time the block size was doubled from 512 bytes up to 8192 bytes, the time to copy a file was almost halved. These results confirm those found by the developers of the BSD file system.

Since Choices currently has no disk buffer cache, we added an index block cache to the Inode class. For copies of large files, those between one and sixteen megabytes long, we found the index block cache tripled the speed of file copying operations.

5 Experiences with C++

While building the file system class library, the use of C++ not only enabled but also encouraged an object-oriented programming style. This style in turn helped us to specify object interfaces and to enforce data encapsulation, *which usually allowed us to perform independent development and debugging.* While all the authors contributed as a group to the designs of each class, we were able individually and simultaneously to work on the implementation of the disk class methods, the BSD and System V details, and the user level file and file system methods. Furthermore, by classifying the objects into hierarchies, we were able to achieve both code and design reuse.

The features of C++ that we found most useful were classes, inheritance and virtual functions. A good example is the Inode class, and more specifically its private method called `mapUnit`. The implementation of `mapUnit` needs no information about whether either the Inode object or its containing InodeSystem object conforms to the BSD or System V standard. Once its code has been debugged, it automatically functions equally well for either of Inode's concrete subclasses; in fact, the file containing the code for the abstract class doesn't even need recompilation in order to support additional concrete classes of Inodes. The primary difference between a BSDInode and a SystemVInode is the details of the disk inode representation. In order to allow functions like `mapUnit` to be inherited by concrete subclasses of Inode, virtual functions were defined for disk inode access routines. At run-time, calls to these methods are translated into appropriately redefined concrete subclass methods.

We did find it necessary to make restricted use of friends. Sometimes objects belonging to different classes need more access to information stored in an object of yet another class.

For example, Inodes use the protected InodeSystem methods: **get**, **put**, **allocate**, **free**, and **close**, while the UserLevelFileSystem uses only the public InodeSystem methods: **open**, **create**, and **synchronize**. Even though InodeSystem declares Inode as a friend, an Inode object still never directly accesses any data member of an InodeSystem object. Hence, we do have a suggested enhancement for C++: instead of giving another class access to all the private data and methods of a class via the friend mechanism, it would be useful to make just certain private or protected methods accessible to another class.

In retrospect, the MemoryObject hierarchy suggests the need to use the multiple inheritance feature of C++. Some MemoryObjects, such as InodeSystems, are collections of other MemoryObjects. They should inherit methods **open**, **create**, and **synchronize** from an abstract class MemoryObjectCollection instead of class MemoryObject.

Our systems also benefited from other, somewhat unrelated C++ features such as inline functions and type-checking. When procedure call overhead is eliminated, one no longer has to consider a tradeoff between code modularity and speed.

The **lint**-style type-checking of C++ invariably flags either coding errors or questionable practices, we do not recall it ever getting in the way of code development.

In general, we always felt that the use of C++ provided the same speed and more expressive power than would the use of C.

6 Future Work

We plan to add support for additional existing file systems, including other UNIX file systems such as that of the Ninth Edition UNIX system, and some non-UNIX file systems, such as the MS-DOS file system. Adding MS-DOS classes to the hierarchy will be more challenging, but they will still fit into our existing class hierarchy.

We also plan on implementing experimental file system components to further support our research. In particular, we are developing an object-oriented file system that propagates its object-oriented structure up into the user interface.

7 Conclusions

In Choices, we have used C++ to develop new operating system mechanisms and policies based on object-oriented design. However, C++ may also be used to recode existing systems in an object-oriented manner. In this paper, we discussed the development of a class hierarchy that captures the design of two existing, well-known file systems. Although data encapsulation has been used in the design of these systems, the ease with which we have been able to design a class hierarchy to capture the similarities between the systems also reflects the adherence of the implementation of those systems to the UNIX standard file interfaces.

Our implementation contributes to our understanding of the design of class hierarchical, object-oriented systems in several ways.

- We demonstrated that system programs can be coded as efficiently in C++ as in C.

- We showed that by careful choice of the methods defined and inherited in the class hierarchy, much code and design can be reused even though the implementations may at first sight, appear to be different.
- The class hierarchy we described in this paper defines a family of file systems, and this family provides an insight into new file systems that are not only constructed as object-oriented systems but are also object-oriented in operation.
- The library of file system components that we built allows hybrid file systems to be constructed that use particular components to provide a customized file system.
- The System V and BSD implementations we built are independent of UNIX and could, potentially, be ported to many other systems in addition to Choices.

Throughout the implementation we have been impressed with the ease with which object-oriented design can be expressed in C++ code. This had many major benefits, particularly in code maintenance, debugging, and modification.

To conclude, this paper describes a complete, efficient implementation of 4.2 BSD and System V file systems as a portable package written in C++. Our next step is to build object-oriented file systems for Choices based on our experience of building UNIX-like file systems.

References

[1] Roy H. Campbell, Gary Johnston, Kevin Kenny, Gary Murakami, and Vince Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *Operating Systems Review*, 21(3):9–17, July 1987.

[2] Roy H. Campbell, Gary Johnston, Kevin Kenny, Gary Murakami, and Vince Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). In *Fourth Workshop on Real-Time Operating Systems*, pages 12–18, Cambridge, Mass., July 1987.

[3] Roy H. Campbell, Vince Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, Santa Fe, NM, November 1987.

[4] 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.

[5] Bertrand Meyer. Reusability: the case for object-oriented design. *IEEE Software*, 50–64, March 1987.

[6] Lawrence Snyder. Using types and inheritance in object-oriented programming. *IEEE Transactions on Computers*, March 1981.

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

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