AN OBJECT-ORIENTED OPERATING SYSTEM

BY

VINCENT FRANK RUSSO

B.S., University of Dayton, 1984
M.S., University of Illinois, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois

AN OBJECT-ORIENTED OPERATING SYSTEM

Vincent Frank Russo, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1991
Roy H. Campbell, Advisor

This thesis describes an experiment to use object-oriented programming and design techniques to design and implement an operating system. This experiment uses object-oriented techniques to address problems of operating system portability, maintainability, extensibility and efficiency. The thesis also characterizes an *object-oriented operating system*.

The results of this experiment are presented in two parts. First, after presenting background information and relevant definitions, I characterize an object-oriented operating system. I then proceed to describe the design and implementation of an experimental object-oriented operating system. This presentation maps conventional operating system wisdom into the object-oriented framework supported by the experimental system. In this way, I show that object-oriented techniques can support realistic operating system algorithms and mechanisms, as well as provide software engineering advantages. The presentation of the system stresses ways in which object-oriented techniques support the system's design and implementation.

The experimental system is evaluated in terms of performance, maintainability, portability and extensibility by using examples of how characteristic operating system problems are addressed. I will show that structuring an operating system in an object-oriented fashion and using the capabilities provided by an object-oriented programming language allows the construction of portable, extensible and maintainable operating systems *without sacrificing performance*. Not only is performance not sacrificed, I will even show how such techniques can often help lead to increased performance over conventionally structured systems.

DEDICATION

To Mom, Dad and Lani, for being *very* patient.

ACKNOWLEDGEMENTS

Primarily, I would like to thank the other members of the *Choices* project at the University of Illinois. In particular: Peter Madany, for listening to me ramble and throw out new ideas as we would walk to the gym; Gary Johnston and Gary Murakami for spending those *very* early mornings with me as we developed *Choices* on department machines when no one else was around and we could get downtime; Ralph Johnson for showing me the "object-oriented light"; and Roy Campbell, my advisor, for his time and patience with me as I wrote this thesis.

Others I would like to thank are: Bjorn Helgaas, Panos Kougiouris, Ruth Aydt, and Dave Dykstra for their comments on the structure and portability of *Choices*; and Aamod Sane, for his comments on the virtual memory system.

Finally, I would like to thank Dirk Grunwald for helping me figure out LaTeX and for writing the wonderful previewer I used in writing this thesis.

This research was supported in part by the National Science Foundation, NFS CDA8722836, AT&T Bell Laboratories 1-5-37388 and NASA NAG 1471.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Operating system software should be efficient and flexible. Furthermore, an operating system should survive a lifetime of changes without sacrificing either of these properties. Among others, these changes include advances in hardware technology and changing demands from users. Keeping pace with hardware advances necessitates portability. In the past, such hardware advances have included the introduction of virtual memory and multiprocessors. If an operating system's abstractions and interfaces are common across a variety of hardware, computer users can take advantage of new and faster hardware with minimal program revisions. Operating systems should be extensible enough to meet changing demands from users. In the past, examples of such demands have included support for graphical user interfaces and distributed systems. System designers cannot always predict and stay ahead of such changing demands. However, they should try to design extensible systems that can withstand moderate changes, since a poorly designed system will not withstand even the most minor changes.

Not only are requirements of portability, efficiency and extensibility important on their own, but their interactions can require important tradeoffs. For example, if performance modifications increase module interdependencies, portability may be sacrificed for efficiency. Likewise, extensions may impair performance if existing mechanisms are altered to support or coexist with them. Operating system engineers are increasingly faced with such tradeoffs. They must often choose between keeping operating systems *portable*, *extensible*, *maintainable*, and *efficient*.

Using modern programming methodologies, languages and software engineering techniques may be the best hope to overcome these problems. The *object-oriented* paradigm[Weg87, JF88, Sny86, Mey87, Boo86, HO87, Mey86, GR83] is rapidly gaining attention as a useful tool to solve the kind of problems operating system software suffers. Object-oriented programming is claimed to support the sharing of common interfaces and code, incremental extensibility, and the development of reusable and extensible software by allowing functions to be written that can take many different types of objects as arguments. In this thesis I describe an experiment to use object-oriented programming and design techniques to design and implement an operating system. I evaluate the success of object-oriented techniques at addressing problems of operating system portability, maintainability, extensibility and efficiency. Simultaneously I characterize what constitutes an *object-oriented operating system*.

The results of this experiment are presented in two parts. First, after presenting background information and relevant definitions, I characterize an object-oriented operating system. I then proceed to describe the design and implementation of an experimental object-oriented operating system. This presentation maps conventional operating system wisdom into the object-oriented framework supported by the experimental system. In this way, I show that object-oriented techniques can support realistic operating system algorithms and mechanisms, as well as provide software engineering advantages. The presentation of the system stresses ways in which object-oriented techniques support the system's design and implementation.

The experimental system is evaluated in terms of performance, maintainability, portability and extensibility by using examples of how characteristic operating system problems are addressed. I will show that structuring an operating system in an object-oriented fashion and using the capabilities provided by an object-oriented programming language allows the construction of portable, extensible and maintainable operating systems *without sacrificing performance*. Not only is performance not sacrificed, I will even show how such techniques can often help lead to increased performance over conventionally structured systems.

In detail, the experiment I propose is to design, build, and evaluate an operating system in which the components are organized as a protected, dynamic collection of objects, defined by classes that are structured by inheritance. All components of the operating system, from low level entities like page tables and device registers, to high level abstractions like processes and files, are designed and implemented as objects. Interaction between these components is implemented by sending messages between objects. The object-oriented attributes of these components are maintained dynamically across and within the privileged (operating system) and non-privileged (application) operating modes of the computer system. Since the construction of an entire operating system is beyond the scope necessary to prove this thesis, I will concentrate on specific subsystems of an operating system. In particular I will focus on memory management, process management, and the operating system interface. These subsystems are critical to any operating system and are fundamental enough to show that object-oriented techniques are applicable to low-level operating system algorithms. Work by others will address additional subsystems of an object-oriented operating system[Mad91, Helng, Ley88, Joh91].

The system I present builds on and integrates accepted operating system techniques including: contemporary approaches to virtual memory management and paging; fully reentrant, multithreaded kernels; lightweight concurrent processes; and support for multiprocessors. The mapping of such techniques into the object-oriented paradigm, and the unique insight this gives, is novel and interesting in its own right. The system is distinguished from other work by implementing a protected, multiprogrammed operating system using object-oriented techniques and by providing an *object-oriented application interface* to access system services.

I will attempt to show that operating systems built using object-oriented techniques:

- can be more maintainable than other systems. In particular that they benefit from:
  - increased modularity, which isolates machine dependencies and increases portability.
  - increased interface reuse, which improves documentation and understanding.
  - increased code reuse, which saves programmer time.
- are more extensible than less structured systems. In particular, their modularity can provide a framework that allows objects to be replaced as the need arises, thus altering operating system policies and mechanisms.
- most importantly, perform on a par with other systems.

In short, I will lend credence to the statement:

> "To build better operating systems, we should be building *object-oriented* operating systems".

Note, however, that the prevailing assumption of this thesis is that the object-oriented techniques are presented as a tool and by no means a "silver bullet". The techniques set forth in this thesis will help operating systems engineers design and implement well-structured, portable, and efficient operating systems. These results are not automatic, however, and cannot be achieved without good operating system algorithms and competent system engineering and programming.

The rest of this thesis is arranged as follows. Chapter 2 presents operating system requirements and problems in detail and surveys previous design methodologies and sample systems using each methodology. Chapter 3 presents definitions of object-oriented design and programming relevant to the rest of the thesis. Chapter 4 presents the definition of an *object-oriented operating system* along with the potential advantages of constructing an operating system in such a way. It concludes by introducing the experimental systems. Chapters 5 through 7 present the experimental system in detail. They describe the object-oriented implementation and performance of the various operating system subsystems designed and implemented. In particular, specific concrete examples of the advantages object-oriented techniques have had on the design, implementation and modification of the system are given. Finally, Chapter 8 presents the conclusions and proposed potential directions for future research.

Chapter 2

Operating Systems

2.1 Operating System Characteristics

In general, the purpose of an operating system is to provide programmers with an abstraction that simplifies the programming and management of a computer's resources. These resources include processors, memory, input/output devices and permanent storage devices. An operating system should control and manage resources reliably and efficiently, and often must enforce policies on their use. The abstraction provided by an operating system is usually in the form of a set of primitive operations providing resource access and control. Programmers use these primitive operations, or *primitives*, when writing programs that need to obtain operating system services. The set of these primitives will be termed the operating system's *application interface* and programs using the services of the operating system will be termed *application programs* or simply *applications*. The complexity and richness of application interfaces provided by operating systems varies widely, but the common subset probably includes primitives to support file and device input/output (I/O), process creation/deletion, and memory allocation/deallocation.

Examples of the low level resource management functions that an operating system must perform include moving the head on a disk drive, handling interrupts from devices, or writing packets of data to a network interface. Even where these functions have direct hardware device support, it is the operating system's responsibility to supervise such resource management functions from their initiation to their completion. Operating systems enforce policy decisions including scheduling priorities of application programs, the amount of memory allocated to each

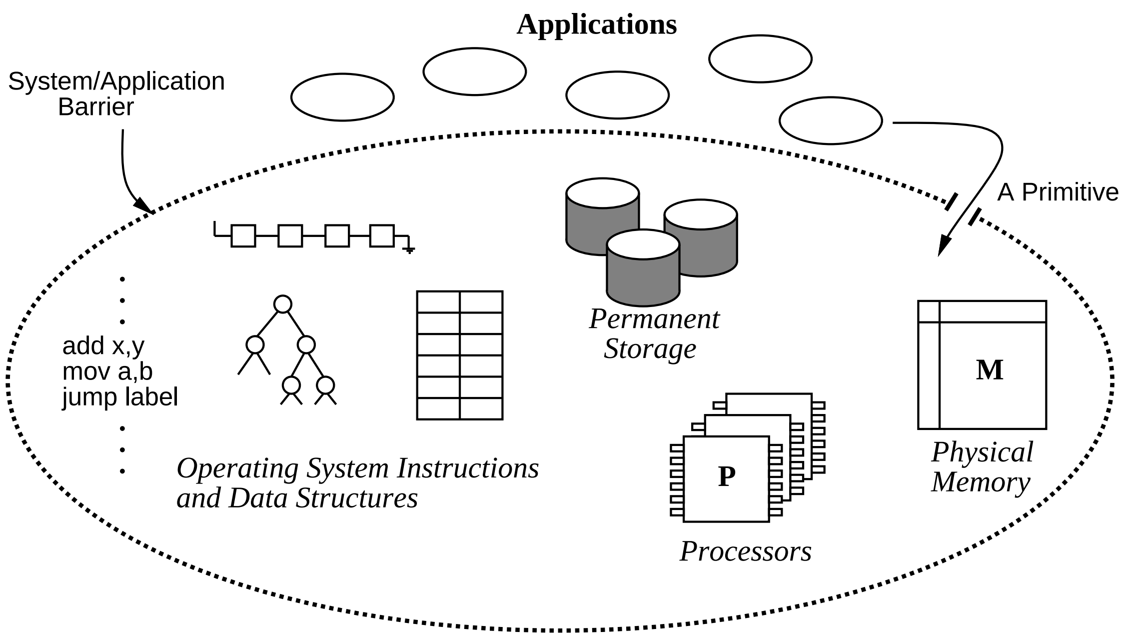

Figure 2.1: Conceptual view of an operating system

application, how long each application shall be given the processor at a time, and the files on secondary storage accessible by an application.

Most operating systems impose a barrier between applications and system functions and data in an attempt to maintain integrity of their data and function, and ensure continuity of operating system services in the presence of potentially malicious or erroneous applications. This barrier is called the *system/application barrier*. The system/application barrier encapsulates the internal components of the system by limiting requests for system services to the operating system primitives. The primitives provide the *only* way to cross the barrier (see Figure 2.1). Depending on the particular computer architecture and the requirements of the operating system, the barrier may be enforced by hardware mechanisms, or be merely a programming convention. If it is simply a programming convention, malicious applications can circumvent the barrier.

The application interface primitives of an operating system allow *delayed binding* of application requests to the operating system functions implementing desired system services. Without such a delayed binding, applications would have to be linked together with the operating system, or include in their data addresses of the system primitive routines. The delayed binding permits the operating system to be changed without modifying the applications.

Most operating systems provide their application interface primitives by an indirection through *entry points* into the operating system. The arguments to such entry points include the operation to be performed and the arguments to that operation. Such entry points decode the desired operation and its arguments, verify the arguments are correct, and call the proper operating system function that implements the service. Conventional operating systems provide a fixed set of services in the form of a predefined set of primitives. In UNIX System V[SVI85] for example, there are about 64 primitives.[1] User applications written to use this interface are independent of its actual implementation and the implementation of the internals of the operating system. New revisions of an operating system may support applications built for older revisions by maintaining backward compatibility with the interface provided by the older revisions of the system. However, as hardware and application requirements change, the interface primitives are likely to need changing as well. UNIX for example, has had numerous additions to its interface from the original version[RT75] to the current versions[SVI85, BSD84].

Avoiding corruption by application programs is just one half of the problem of protecting system data and keeping it consistent. Operating system software must also deal with potential inconsistencies in system data that the operating system software itself might cause. For example, if multiple threads of control are executing the operating system code, they might leave shared data in an inconsistent state unless they synchronize their use of the data.

Other potential inconsistencies can arise from interrupt processing. Interrupts can occur at almost any time during a system's execution. If an operating system is in the middle of updating system critical data and it receives an interrupt, the function to handle that interrupt might need access to the same data. If the interrupted routine had not yet finished updating the data, the interrupt handling function might reference the data in an inconsistent state. Even worse, if the interrupt handling function were to further attempt to update the data, it could compound the problem by further altering already inconsistent data.

The need to prevent potential inconsistencies in shared data caused by concurrent accesses is generally termed guaranteeing *mutual exclusion*. Protecting data from corruption during interrupts is usually addressed on a uniprocessor by disabling interrupts. On a multiprocessor,

[1] In UNIX, primitives are usually termed *system calls*.

this is not enough; mutual exclusion is usually guaranteed by a combination of disabling interrupts to prevent accesses by the same processor and using spin-locks and/or semaphores[Dij68] to prevent access by other processors.

2.2 Types of Operating Systems

The earliest computers had no operating systems by the modern meaning of the term, and had no need of them. Programmers using a computer had complete access to the entire machine. There was no protection mechanism to keep an application program from accessing any device or memory location desired. Applications for such computers ran sequentially. Each would reinitialize the machine for its use, consume its input, perform its computation and produce its output. A program for such a computer is, in a sense, simultaneously the operating system and an application. Operating systems for some *embedded systems* still behave this way. An embedded system is a computer that is contained within a device whose primary purpose is usually not general purpose computation. An example is the navigation computer found on most modern aircraft, or a computer managing call routing in the phone system.

In the absence of an operating system, a programmer has to write functions to manipulate all the devices of the computer. This includes reading data from tapes or disks (if the computer even has any) and generating output to a printer or back to disk or tape. Embedded systems usually have more complex devices, for example, sensors to read and displays to update. These device manipulation functions are difficult to write and must be duplicated in every application that uses the computer. For an embedded system this is usually not a problem as there is often only one "application" ever written. But general purpose computers with multiple application programs should not require reimplementing these functions for every application. If the functions are complex enough to warrant it, libraries of device control routines can be written and used by application programmers to avoid having to reimplement them for every application. These libraries could be considered crude operating systems. This is often the form the operating systems for very early computers took.

If a computer is shared by many users, each with many application programs to run, both computer and programmer time can be utilized more efficiently if each application does not have to be written to perform all the initialization and device management required to make the computer usable each time it begins. The solution to this problem is to create *supervisor programs* that initialize the computer and load and execute applications one at a time. When an application finishes, the supervisor program again takes control and reinitializes the computer for the next application. The supervisor also manages all the device I/O requests for the applications. A supervisor is implemented by localizing common I/O routines, leaving them always resident in the computer's memory, and allowing application programs to call them. These routines define the application interfaces of such operating systems.

Operating systems of this type exhibit most of the characteristics described in the introduction to this chapter. Many of today's personal computer operating systems are designed this way. Most of these systems are, however, *unprotected* operating systems. Unprotected here implies that such systems lack hardware enforcement of their system/application barrier. Without hardware support for protecting the operating system functions and data from malicious or aberrant applications, such systems can be unreliable and easily corrupted. This kind of system is often euphemistically called a *single user* system. While lack of protection may be acceptable for a personal computer, or a computer where the applications are thoroughly debugged and trusted, it is not acceptable on shared computers. Having to reboot and reinitialize the machine every time any application has a bug is unacceptable.

Modern computer architectures provide mechanisms that can be used by an operating system for protection. These mechanisms include *privileged execution modes* for processors and *memory protection* schemes. They prevent non-privileged application programs from accessing the memory storing the operating system data and functions. Operating system functions execute in the most privileged mode, allowing access to system data and functions. Application programs execute in the least privileged mode restricting them from accessing system data and functions. In addition, the least privileged mode prevents the execution of certain processor instructions that might compromise the systems security, for example, a *set privilege mode* instruction. The least privileged mode is also usually denied any direct access to resources. Disallowing applications direct access to system memory and resources enforces the encapsulation of the hardware that the system provides. In such *protected operating systems*, primitives are implemented with special *supervisor call* (SVC), or *trap*, instructions. These instructions raise the privilege level of the processor, and simultaneously jump to an entry point within the operating system's memory. Operating system entry points are functions that verify their arguments and then perform then requested service. Once the system service is complete, the operating system lowers the privilege level of the processor back to that of the application and resumes the application at the instruction following the SVC or trap instruction.

A *multiprogrammed* operating system allows multiple applications to reside in the computer's memory simultaneously. The processor is shared by assigning it to another application while an application is awaiting an I/O completion. This gives more efficient utilization of a computer and its attached devices by overlapping I/O and computation. A *time-shared* operating system is a multiprogrammed operating system in which a timer enforces the sharing of the processor by interrupting executing applications after a certain *quantum* of time if they have not yet blocked on an I/O request. Applications must be protected from one another in a multiprogrammed system. On computers that sequentially execute applications, it was enough to partition the resources of the computer into two parts: those used by the system and those used by the application. Multiprogrammed operating systems must share computer resources (mainly memory) between multiple applications. Support for this is usually provided by allowing multiple address spaces that only the operating system can change between. Each application is assigned its own address space and cannot reference any data or functions in other address spaces. Multiple address spaces can be provided in many ways, including: base/bounds registers, segmentation, and virtual memory[Dei84a, PS85].

As an alternative to building costly high performance processors, the proliferation of low cost microprocessors has allowed system designers to build high performance computing systems out of a large number of small, inexpensive systems. *Distributed operating systems* support such a computing system. Each computer, or *node* in a distributed system is connected to the others by some form of network allowing inter-node communication. A distributed operating system provides an application the abstraction of a single computer with all resources accessible through a uniform, location transparent mechanism[TvR85]. Many distributed systems exchange information between programming entities using *messages*. The entities are distributed across nodes in the system. Each entity has a global identification. Messages are sent to entities in such a way that they are independent of the entity's location. Operating system services are provided by such entities and may reside on arbitrary nodes.

In summary, operating systems range from embedded systems, to simple I/O library packages, to multiprogrammed systems that protect themselves from applications and applications from each other, to distributed systems.

2.3 Operating System Software Engineering

Most operating systems are large bodies of software and require considerable software maintenance. Like any other large software system, they are likely to have bugs that will be discovered and fixed over time. Bug fixes are one just one form of software maintenance, others include adapting software for new purposes, adding new features, and enhancing existing features. Brooks surveys examples of such problems that confronted the IBM OS/360 system in [Bro75]. Current estimates show that software maintenance costs as a percentage of the total software budget have grown from 35-40% in the 1970's to a projected 78-80% in the 1990's[Pre82].

Operating systems do not only suffer from the normal maintenance problems associated with large software systems, they also have some characteristic software engineering problems of their own. The following sections discuss the additional burdens that concerns of portability, efficiency and extensibility place on operating system programmers.

2.3.1 Portability

The decades of the 1970's and 1980's were characterized by rapid advances in computer architecture that lead to an increasing diversity across the spectrum of available computer hardware. To isolate application programmers from this diversity, these decades saw an opposite trend in computer software, namely, the desire to provide common computing environments across differing computer hardware. This need, along with the costs of developing new applications for, and retargeting existing ones to, new operating systems makes it desirable to keep operating systems portable across broad and diverse ranges of computers. For example, the UNIX^{TM} operating system[RT75][2] is available on computers ranging from personal computers costing a few

[2] UNIX will be mentioned extensively in examples throughout the rest of this section. This is mainly because of its current prevalence in the world of operating systems and, therefore, the likely chance that the reader is familiar with it. UNIX was originally developed at Bell Laboratories in Murray Hill, New Jersey. During its evolution many individuals and institutions contributed new features to UNIX. The University of California at Berkeley added paged virtual memory, a new file system and internetworking support to UNIX[LMKQ89], Sun Microsystems, AT&T, and other companies helped develop UNIX into a commercial operating system, and

thousand dollars [App89, IBM88a] to mainframe computers costing millions of dollars[Cra88]. Yet, to facilitate program portability, the UNIX interface is preserved across this range. Obviously, as diversity in hardware increases, keeping an operating system portable becomes an increasingly difficult task. Its intimate coupling to a computer's hardware only compounds this problem. Designing operating systems for portability may make it necessary for particular parts of a system to have many different versions tailored for various architectures. However, since the versions of these parts for different machines all likely perform a similar function, large portions will be identical. If each version is independent, keeping the commonalities in phase as changes are made can be extremely difficult.

2.3.2 Efficiency

Operating system software has an extremely high demand for efficiency placed on it. Purchasers of computer hardware want to get the maximum performance for their dollar. Since operating systems are the primary managers and controllers of a computers resources, inefficiencies can have a tremendous effect on overall system and application performance. Efficiency concerns impact operating software engineering in two ways. First, the most flexible and beneficial software engineering techniques may be inapplicable to operating system construction due to their performance limitations. For example, while interpreted dynamically typed programming languages often provide excellent programming development and debugging environments, the performance of such languages is likely to be insufficient for operating system software. Second, optimizing an operating system for maximum performance on a given architecture may impact its portability to other architectures. For example, using custom context switching instructions or high-performance I/O instructions may bias the implementation towards a particular architecture. What is needed is a mechanism to allow such optimizations to be performed in a localized way and remain isolated from the rest of the system.

2.3.3 Static Extensibility

Operating system requirements for extensibility come in many forms. Static extensions are those implemented by modifying the operating system source code and rebuilding the system. Often

---

recently, Carnegie Mellon University has begun to reimplement much of the UNIX internals with its Mach[A+86] project.

such extensions are the result of hardware advances and the corresponding change in demands that they cause programmers to place on the operating system. Such extensions can lead to major modifications of existing operating systems and often necessitate entirely new operating systems being designed and implemented. For example, the introduction of virtual memory has allowed applications to be written with memory requirements that far exceed the amount of physical memory of the computer. This requires operating systems to manage the sharing of physical memory not only between the address spaces of different applications, but within different parts of the address space of the same application. Simply partitioning the available memory into pieces large enough to hold each application is no longer a possible solution. Another example of how hardware advances have stressed operating systems is the way in which multiprocessors have stimulated the development of applications requiring concurrently executing threads of control sharing common data. This requires operating systems that can cope with shared memory and maintain data integrity in the presence of concurrent accesses.

UNIX, for example, has undergone considerable changes to deal with shared memory and multiprocessors. Multiple solutions have been adopted to support shared memory. One approach is to extend UNIX to allow specification of regions of a process's address space to be shared between a parent and child process across a *fork*.[3] This is the approach adopted in the UMAX[Enc86] and Mach[R+87] versions of UNIX. Another approach is to allow a programmer to specify a region of a program's memory to be shared with other programs that make similar requests. This solution is chosen as the basis for the UNIX System V shared memory standard[SVI85]. The first solution follows the UNIX philosophy of passing resources (previously just open files but now memory as well) by process creation. The second mechanism is somewhat more flexible and allows easy implementation of shared code libraries. However, the interface of UNIX is now inconsistent between UMAX, Mach and System V. Application portability has been sacrificed to allow shared memory.

Multiprocessor architectures likewise affected UNIX. For an application program to take full advantage of the concurrency made possible by multiprocessors, it is necessary to have

[3] The fork primitive is the UNIX process creation primitive. The semantics of fork are to create a second, or *child*, process with an exact copy of the creating process's memory. The second process has no further access to its parent process's memory and communicates with its parent process only through the UNIX I/O system. In UNIX, new programs are executed using the *exec* primitive. The exec primitive replaces the contents of the memory of the invoking process with the data and code of an entirely new program and directs the process to jump to that programs beginning.

multiple processes simultaneously executing on multiple processors. This means that multiple processes may simultaneously require services from the operating system. The assumption in the original UNIX implementation was that only one process would be executing the operating system's code at a time. Therefore, when UNIX processes executed system code by invoking an operating system primitive, they were guaranteed exclusive access to system data by simply disabling interrupts and not relinquishing the processor to another process until the critical data was updated or accessed. On a multiprocessor this is not adequate. A process executing on another processor may simultaneously need access to the same data. One solution to the problem is to adopt a master/slave system in which a select, or master, processor executes all of the operating system while the other processors execute only applications. When an application needs system services, it has to wait for the master processor to execute its request[Bac86]. This approach suffers from low utilization of the available processors when processes are generating a lot of requests to the operating system. In the worst case, all the application processors are blocked waiting for something to do, while the master processor is busy processing many requests. A master/slave approach does not scale up well to many processors since the master processor becomes a bottleneck[BB84]. A more desirable solution is to rewrite the UNIX kernel to allow multiple processors to execute system code. When necessary, exclusive access to system data can be guaranteed by explicitly coded mutual exclusion primitives such as semaphores and spin-locks[JAvdG86]. This solution has the disadvantage of requiring almost the entire UNIX kernel to be reimplemented or modified in some way, but has been adopted by many current vendors of multiprocessor UNIX systems[Enc89, Seq85b]. Like shared memory interfaces, parallel programming interfaces for UNIX[Enc88, Seq85a, Doe87, SUN88, CD88] also lack standardization and further aggravate the application portability problem.

2.3.4 Dynamic Extensibility

Some extensions to an operating system are dynamic, in the sense that they should not involve redesigning or maybe even recompiling the system. These include extensions such as increasing the number of memory or disk storage devices attached to the computer, adding new devices and device drivers, adding new interfaces to these or existing devices, and changing operating system policies. An increase in the amount of memory or disk storage might necessitate reevaluation of resource management/allocation policies and corresponding changes in the parameters governing their enforcement. Adding a new device to a computer, and driver software for it to the operating system, should have little if any impact of the rest of the system. Some computer hardware, for example a network controller, even allow dynamic additions of new devices without interruption of service[McM81]. Operating systems for such computers should be able to provide the same capability, namely, the ability to add drivers for, and interfaces to, these devices without even rebooting the computer.

Another form of dynamic extensibility arises from needs to alter policies on resource allocation and management. An operating system should be able to adapt to changing policy requirements without mechanisms having to be changed and, ideally, without even having to reboot the computer. Policy changes should be as simple as changing a few system parameters or replacing one policy unit with another.

2.3.5 Software Engineering Problems with Extensibility

Both static and dynamic extensions to an operating system place increasingly higher demands on its design and implementation. Operating systems need to be constructed in a way that they remain extensible enough to cope with changing requirements, while keeping the cost of maintenance as small as possible. Often the original implementation of an operating system has trouble supporting many desired extensions. This can be the result of either bad original design decisions or poor implementations of those decisions. Likewise decisions that were appropriate ten or twenty years ago may now be out of date. The problem can be aggravated by poor documentation about the original design and implementation and by the potential lack of availability of the original authors for consultation. With the rapid advances in computer architectures currently underway, these situations are not likely to improve soon. It is possible that UNIX is one system rapidly evolving past the point that the original design can support all the desired additions. This would explain the current trend to build UNIX compatible systems based on newer design and implementation technologies [LS87, A+86, RAN88, OCD+88]. This is not the fault of the original designers of UNIX as they had a very specific goal in mind and were constrained by contemporary software engineering techniques. It is rather a fault of the computing community pushing the original UNIX design too far. Portability considerations make it desirable to preserve the UNIX interface; this allows programmers to use familiar programs and tools. This in no way, however, means that the internals or implementation of

UNIX need be preserved. Whether the interface of UNIX is good is not the issue here. Instead, the point is that the internal structure of most UNIX implementations must be rethought and redesigned. The goal should be to use modern software engineering techniques to redesign and reimplement UNIX, other systems, and *new operating systems* in such a way as to minimize problems with constructing maintainable, portable, efficient and extensible operating system software. This thesis will show that object-oriented design and programming are such flexible and efficient techniques.

2.3.6 Maintenance for Different Types of Operating Systems

The operating system software engineering problems discussed so far affect each type of operating system discussed in Section 2.2 differently. Portability is usually not a concern for embedded systems since they are usually designed and constructed for a specific piece of hardware. Likewise, personal computer operating systems are often designed for a specific architecture, such as the Apple Macintosh operating system[App88] or the IBM-PC operating system[Nor85]. However for the reasons stated above, multiprogrammed systems with a rich set of existing applications need to be constructed in such a way as to remain portable across hardware variations. Extensibility for almost any form of operating system is important for the reasons discussed above. Maintenance is a concern for any large body of software as well. Even embedded systems will have maintenance costs associated with them. Efficiency is important for any type of operating system. Thus, software engineering issues relevant to operating system construction are orthogonal to the particular type of operating system although more complex operating systems (protected, multiprogrammed, distributed ones for example) may be more affected by virtue of their complexity.

2.4 Operating System Design Techniques

Many design approaches have been applied to structuring operating systems to address the kinds of software engineering problems discussed in Section 2.3. Most attack operating system problems by decomposing the system into smaller pieces, or *modules*, with well defined interfaces. Webster's dictionary defines a module as: "any of a set of units ...designed to be arranged or joined in a variety of ways"[McK79]. Modularization is a major accepted technique to decompose and structure large software systems[Par72]. The key to a successful modularization technique is to determine the correct granularity of these modules and to provide efficient data exchange between modules.

The following sections survey a few common and historical approaches to the structuring and modularization of operating systems. The techniques are roughly presented in chronological order of their use. These techniques have evolved in response to advances in hardware technology and improvements in software engineering techniques. Each section concentrates on a particular design philosophy and identifies its problems in order to motivate the design approach put forth in this thesis.

2.4.1 The Single Uninterruptable Monitor Approach

One of the earliest operating system structuring techniques is that of a single *uninterruptable* monitor program with a single thread of control. This type of system, in effect, dispatches or "calls" application programs in much the way that it calls functions within itself. An application programs "returns" to the system under one of three conditions: an *interrupt* from a device requiring service, a service request from the program to the operating system, or the program's termination. Once the call to an application program returns, the operating system services the interrupt or request (or deletes the terminating program) and then resumes another (or possible the same) program.

An operating system constructed as a monitor guarantees mutually exclusive accesses to system information since there is only one thread of control allowed to execute the operating system code at a time. While this thread of control is executing in the operating system, interrupts are disabled, prohibiting all but explicit changes of control flow. This makes implementation easier since the implementor can ignore such problems as mutual exclusion and concurrent access to system data structures. The main problem of such a design is the lack of scalability. The frequency of calls to the operating system from interrupts and application service requests is proportional to the number of application programs and devices in the system. As the number of calls to the system increases, the time during which interrupts are disabled increases because the rising amount of time spent executing in the system routines. This in turn increases the number of interrupts that can be lost, or held pending for a long time, thereby decreasing I/O device throughput. A system structured to allow only one thread of control accessing system data must, on multiprocessor architectures, serialize simultaneous attempts to enter the system as the result of interrupts. This results in a decrease in potential concurrent execution. Decreases in potential concurrency are also seen as application programs must wait for each other to enter and exit the system.

The monitor technique protects system data from interference, but at the price of scalability and performance. From a software engineering point of view, this approach has many problems as well. Such a system imposes no guidelines on how to structure its internals. Since all of the functionality of the operating system is placed within a single module, maintenance is severely impacted. The system internals are not divided into functions or sub-units that can be separately developed and maintained. This reduces portability and extensibility. These problems could be solved, however, with a good approach to further decomposing the monitor.

2.4.2 The Kernel Approach

The *kernel*[4] model of structuring an operating system is an attempt to remedy some of the performance problems of the uninterruptable monitor approach, in particular, those of scalability and device under-utilization. It also attempts to further decompose the components of an operating system. This model treats all computational entities in a computer as *processes*, or threads of execution. Examples are application program processes, interrupt handler processes, and device driver processes. The kernel is primarily an interprocess communication module. In the kernel model, an operating system is a set of concurrently executing system processes that request services from this kernel. Applications are likewise viewed as sets of concurrently executing processes that request services from the operating system processes via the kernel. The kernel provides a minimal set of routines that perform the basic functions of interprocess communication, process management, and interrupt processing. Higher level operating system functions are built around the kernel by using processes. The kernel is responsible for scheduling processes and directing interrupts to the proper system processes. Interrupts are disabled while executing within the kernel, but since most of the operating system functions are moved out of the kernel and into system processes, interrupts are enabled more often, thus improving device performance. A kernel should be capable of processing multiple requests concurrently as

[4] Some designers have used the term nucleus.

long as the processes are programmed to use mutual exclusion to access system data. Therefore this approach is applicable to multiprocessor architectures.

A minimal kernel needs only to manage interprocess communication and direct interrupts to the proper processes. Larger kernels may also create and delete processes, provide memory management, implement the application interface primitives, and supply a wide variety of other services. Kernels become more difficult to implement and maintain as they get larger.

The cooperating, concurrently executing process model is the most valuable contribution of the kernel model. The extra concurrency provided by this model improves on the monitor approach. The idea of decomposing an operating system into a set of communicating and cooperating processes increases modularity thus aiding portability and extensibility. The problem of identifying which processes should handle which operating system functions and further decomposing those processes, as well as the kernel itself, still remains. However, the kernel model remains the basis for the construction of most modern operating systems[A+86, Che84, RAN88, Mul87, OCD+88].

2.4.3 Level Structured (Layered) Operating Systems

Layered systems, most notably THE[Dij68] and later VENUS[Lis72], attempt to decompose an operating system by structuring it in small, easily understood, layers or levels. The processes or functions of the system are separated into layers that provide successive abstractions of the operating system. These layers are ordered by increasing level of functionality, and each layer depends only on the previous layer in this ordering. Usually, the hardware is the lowest layer, and the application interface is the highest layer.

Many early layered system divided an operating system into layers of processes performing system functions. Habermann, Flon and Cooprider[HFC76] argue that this makes it difficult to separate the logical activities of processes from the processes themselves. They argue for a *functional hierarchy* of layers in the system. In their design, layers are built to reflect the *functions* in the system. Various processes in a system invoke these functions, but processes are independent of individual layers. The lowest layer corresponds to the hardware instruction set of the processor. Functions in higher layers can use functions from lower layers. Concurrent processes within the operating system can access functions at different layers within the hierarchy.

Layering aids in implementation, debugging and testing of the system. Layers enhance portability; if lower layers hide the hardware, only these layers need to be changed when retargeting the operating system to new architectures. An implementor can ignore the implementation details of lower layers but still use their functionality when designing and debugging higher layers. This improves maintenance. For example, some layered systems use as their lowest layer the concept of an *abstract machine* representing an idealized computer architecture. This reduces the portability problem to reimplementing the abstract machine for the available computer hardware. Abstract machines also can represent real computer hardware as in the VM operating system[MD74]. This allows multiple virtual computers to be simulated on a single physical computer by supporting multiple concurrent copies of the lowest layer each sharing the physical computer. Operating system software can be developed and debugged on any of the virtual computers and, when ready, be run directly on the physical computer without any changes.

The major difficulty with building layered operating system kernels is determining the layer in which a process or function belongs. Since each layer may only rely on the processes or functions provided by lower layers, careful planning is necessary. For example, in virtual memory systems, the disk device drivers should be provided by a lower level than the virtual memory paging mechanism since the memory paging mechanism must use the disk as a backing store. But, the memory that the disk drivers use for I/O buffers must be coordinated with the virtual memory management. Such circular dependencies are the most difficult problem in defining the layers of an operating system.

Another problem with layered systems can be performance. If a layered system is structured in such a way that a layer has access to only the layer *directly* beneath it, performance can suffer as requests must traverse several layers to achieve a low level service. It is more desirable to allow a layer to access the functionality in any of the layers beneath it.

Perhaps the biggest drawback to layered systems is that the granularity of the abstractions it provides (the layers) are too coarse. However, layering is orthogonal to many other structuring techniques. When further decomposed into servers, in the message-passing approach as discussed in the next section, or into the objects in object-based system as discussed in the following section, layers can substantially aid the documentation and high level understanding of a system.

2.4.4 Message-Passing Operating Systems

Message-passing operating systems are systems based on the kernel idea. They attempt to further decompose an operating system's structure. Message-passing systems use explicit *send* and *receive* operations to exchange information (messages) between concurrently executing processes. Each of these processes (or often sets of processes) is usually viewed as a *server* providing functionality to other *client* processes. References to servers are obtained from *name servers*, which convert symbolic service names to references to servers implementing the service.

Each message from a client includes a request to the server and arguments specific to that request. In a message-passing system, *all* communication and computation is achieved by explicit message exchanges between clients and servers. Messages are sent to servers and replies are sent back. This message exchange may be synchronous, in which case the sender does not continue executing until the reply is received, or asynchronous, in which case the sender continues to execute and awaits the reply whenever desired.

Processes executing on behalf of application programs are often just consumers of services and may not provide any of their own. In message-passing systems the kernel is usually viewed as a server as well. It can be composed of many processes; each executes on behalf of the system to perform system management functions, for example, handling interrupts and creating or deleting new processes. Processes desiring a service from the kernel send a message and (optionally) await a reply just as they would do if requesting a service from another process.

Message-passing systems come in two forms, those that consider all message receivers to be processes or servers directly, such as the V system[Che88, Che84], and those that consider message receivers to be message ports read by servers, such as Accent[RR81] and Mach[TR87, R^+87, Ras86]. In a message-passing system using ports, server processes poll selected ports when ready to receive a message. In the other type of system, messages are sent directly to a target process and are received the next time that process executes an "anonymous" receive primitive. The receive is anonymous in the sense that the process simply receives the next message queued for it. In a port-based system, the server process could selectively chose which port to receive the next message from, giving more flexibility in assigning priorities to messages.

Message-passing systems work well in distributed environments. The only support needed is to provide server identities that can be used independently of location and message send and receives that can span machine boundaries. In this way, processes on one machine request services on another machine in exactly the same way they request services on the machine they are executing on: by sending a message and awaiting a reply.

Decomposing the operating system into a set of servers increases both portability and maintenance. Portability is improved since only servers relying on machine specific details need to be retargeted for new architectures. Maintenance is assisted by the decomposition of system functions imposed by servers. One problem with such systems is that a message send/receive is usually much more expensive that a normal procedure call. Since any message send can potentially cross a machine boundary, arguments must often be copied rather than being referenced off the stack of the sender of the message. Likewise, the synchronization between the sender and the receiver imposes additional overheads. For example, a context switch may be incurred from the sending to the receiving process. These performance problems often cause the model to become inefficient for low levels of abstraction. Programmers then have to revert to traditional programming methodologies. This results in two kinds of entities in a message-passing system: the high level servers described by the model and modules of low level routines invoked with traditional procedure calls. This lack of consistency in paradigm is undesirable from a maintenance point of view. For example, a server providing memory allocation may need to update the page tables for the current process. For portability and consistency, it would be desirable to make each page table another server and have the memory allocation server send a message to the page table server to update memory mappings. Only the page table server would need to be reimplemented when retargeting the operating system to a new machine. The problem is that the memory allocation server may have to make repeated requests to the page table server. Since they are fundamental parts of the kernel, the page table server and the memory allocation server are likely to occupy the same address space. The expense of multiple message send/receives between them will be higher than just invoking functions directly to update the page tables.

Many system designers have gone to great extent to minimize message sending costs. Shared memory can minimize argument copying costs[Y^+87]. Having one process execute on the behalf of multiple servers can reduce context switching costs. Since a high level abstraction likely makes multiple requests on a low level abstraction, requests to low level abstractions are usually more common than those to high level abstractions. Therefore, rather than starting with a complex scheme for handling high level abstractions that will not easily and efficiently handle the low level cases, it would be desirable to start with a simple scheme for handling low level abstractions that is efficient and will scale up to higher level abstractions. This allows a single paradigm to be used throughout the construction of the system.

2.4.5 Object-Based Approach

Object-based approaches to operating system design replace the kernel model of communicating, concurrently executing processes, with a collection of communicating, cooperating *objects*.[5] Each object in the collection represents a particular logical entity of the system. Objects can represent processes, memory ranges, communication channels, devices, and many other operating system abstractions. Each object provides a set of operations available to other objects in the system. These operations define the behavior of the object and the interface provided to other objects. In this model, objects invoke such operations by sending messages to other objects. These messages are conceptually similar to the messages in the message-passing approach. The main difference is that sending a message to an object is always synchronous and no explicit receive is needed.

This encapsulation of behavior in object-based systems closely parallels the software engineering concepts of modular programming and data encapsulation. An object can send a message to any other object as long as it has a reference to that object. Like the servers in message-passing systems, objects can reside on different nodes in a distributed system. Examples of object-based operating systems include: HYDRA[W+74], Eden[LLA+81], Emerald[JLHB87], CLOUDS[Spa86], CEDAR[SZBH86], the Intel iAPX432 architecture[Int81], Amoeba [Mul87, TM81, TvR85], and CHORUS[RAN88, Mar88, BMR85].

Object-based approaches are more *data-driven* than message-passing approaches to operating system construction. They separate the abstractions of a system into different modules (objects), each with a well defined function and interface. Objects address other objects in a system by means of a *reference* or *capability* to the object. The reference can define what permissions the invoking object has with respect to the object on which it is operating. These references can be as simple as direct pointers to other objects, in effect providing no permission enforce-

[5] A more complete definition will be given in Chapter 3, but for now it suffices to say that an object is a data encapsulation consisting of a set of state variables and a set of operations to access and modify those variables.

ment, or as complex as capabilities in full protected capability-based systems[KL87, MB80]. In a capability-based system, only trusted objects can modify and distribute capabilities. To maintain security, capabilities can only be updated by trusted objects. Some systems provide hardware support for this protection, while others rely on indirection through a trusted manager of capabilities.

Object-based approaches address many of the operating system problems discussed in the introduction. Sets of objects can be used to abstract the hardware and thus increase portability. Objects also represent a small enough encapsulation to improve maintenance and documentation.

The main potential problem of object-based systems is, like message-passing systems, one of efficiency. Efficiency in an object-based system is a function of the expense, or "weight", of objects and the implementation of message sends between objects. Object-based systems span a spectrum of implementations. At one end of this spectrum are systems like CLOUDS[Spa86] and Elmwood[MLC+87] that are, in a sense, remote procedure call (RPC)[BN84], object-based, message-passing systems. Such systems use the object/message send paradigm to structure the servers of message-passing systems. Objects encapsulate servers and object messages structure the messages exchanged between clients and servers by automatically providing the opcode and defining the types of parameters. The object interface gives structure to the interface that the server presents to its clients and defines the messages sent between the client and server. However, such systems suffer the same performance penalties as message-passing systems.

At the other end of the spectrum are systems like Smalltalk[Gol84] that, rather than using an explicit send/receive paradigm to implement message sending, transfer control between objects by explicitly weaving threads of control from object to object. In such systems, objects are usually passive. Message sending is implemented with traditional procedure calls. The thread of control of the invoking process enters the object to perform the operation. This reduces synchronization and context switching costs and involves no added expense of argument copying. The one problem with such systems is that, without the underlying message send/receive paradigm, they are difficult to extend to distributed cases. Section 8.8 discusses this further and proposes a solution.

Systems based on multiple monitors [Hoa74, Bri85, Bri73] for data encapsulation, like Pilot[LR79], have a type of object-based design at this end of the spectrum as well. Rather than a single monitor like the approach discussed in Section 2.4.1, these systems use multiple monitors, each encapsulating some system data. Like objects, monitors provide a set of operations (monitor procedures) to access an otherwise inaccessible set of data. Invoking a monitor procedure is done in a similar way to invoking a normal procedure. The unique contribution of monitors is the enforcement of mutually exclusive access to the data. A process invoking an operation on a monitor is not allowed to enter the monitor until any other process in the monitor exits it. Local operations are provided by monitors for a process to exit the monitor until a specified condition occurs and for another process to signal that condition's occurrence. Campbell and Kolstad[CK79] propose a similar object-based scheme but provide a more generalized mutual exclusion mechanism by using path expressions[CH74].

Object-based systems address the software engineering problems for operating systems by decomposing its functionality into small modules (objects) with well defined interfaces. If objects and message sending can be made efficient enough, performance will not be a problem. A disadvantage of object-based systems is that objects, like servers in message-passing systems, are often designed to implement general purpose algorithms in order to provide the desired interface under a large variety of applications. The general purpose nature of such objects may cause performance to be sacrificed for generality. Obviously, a new object can be implemented to perform a specialized or optimized service. However, it is difficult to program the commonalities this object may have with other, more general, objects in a way that when additional features are added to the general objects, they are also automatically made to the specialized object. What is needed is a way to allow efficient specialization of objects while allowing sharing of common features between the original and the specialized object. Some of this problem can be alleviated by allowing one object to *delegate* an operation to another object. There is, however, no convenient language or system enforced manner to structure such delegation. The burden is place entirely on the programmer.

2.5 Summary

Although they represent many important ideas, the techniques presented so far all fail to address fully the operating system software engineering problems discussed in Section 2.3. The single uninterruptable monitor approach suffers from the obvious disadvantage of placing all the functionality with a single module. Although additional structuring techniques could be applied to further decompose the monitor, it is only suitable for small systems since it does not scale up well due to competition to enter the system.

The kernel approach addresses the structuring problem by breaking an operating system into a set of cooperating, communicating, concurrent processes. This approach helps by separating an operating system into a set of modules (the kernel proper and the processes) but fails to aid in specifying the internals of the kernel itself, the functions the processes should perform, and the information that should be exchanged between communicating processes. In theory, only the kernel should need to be changed in order to move the operating system to new architectures. In practice, however, kernels are much more complex than the simple model describes. Issues such as memory and device management usually pervade the design of kernels and decrease their portability substantially. Likewise, the functions the processes should perform and the information that should be exchanged between communicating processes is not specified. Kernels do, however, solve the mutual exclusion bottleneck present with the uninterruptable monitor approach.

Layered, message-passing and object-based systems attempt to provide further structure to kernels. Layered systems structure operating systems by separating the processes or functions of the system into layers that provide successive abstractions of the operating system. The major difficulty with building layered operating system kernels is determining the layer to which a process or function belongs. Since each layer may only rely on the processes or functions provided by lower layers, careful planning is necessary. Layering is, however, a useful high level structuring technique that can be used orthogonally to other techniques.

Message-passing solutions use explicit send and receive primitives between processes and servers to achieve communication and computation. In such a system, even kernel services are obtained by sending messages and awaiting replies. The overhead of message send and receives can, however, be prohibitive and force designers to abandon the paradigm at frequently used interfaces.

Object-based architectures organize the system as a network of cooperating objects rather than layers of processes or functions. Processes are then just one type of object; memory, devices, and communication channels are represented by other objects. Objects can solve the circular dependency problems in layered systems since object communication topologies can be arbitrary. If objects and message sending can be made inexpensive enough, object-based systems offer desirable software engineering advantages in terms of encapsulation and modularity. A mechanism to allow specialization in a flexible manner is still, however, lacking.

The object-oriented solution proposed by this thesis builds on the lightweight object-based approach and improves it. It uses object-oriented programming to add a classification mechanism as well as dynamic binding of operations to objects. It likewise provides mechanisms to specialize objects while maintaining commonalities between similar objects. The resultant systems provides for *very* lightweight and flexible abstractions and encapsulations that can be used uniformly throughout the system. Before the presentation of this solution can be fully detailed, a discussion of the object-oriented programming paradigm is necessary and will be undertaken in the next chapter.

Chapter 3

The Object-Oriented Paradigm

> *"No doubt about it – a new technique called 'object-oriented programming' has become all the rage in the world of software."*

This quote is from an article by Lee Gomes under the headline *"Programmers do it without a definition"* in the San Jose Mercury News, Feb. 7, 1990.[1] The quote is very true but, unfortunately, what is perhaps more true is the headline. This chapter will attempt to define object-oriented programming sufficiently for the presentation of the work in the rest of the thesis.

3.1 Principles

The following sections discuss four essential principles of the object-oriented paradigm relevant to the construction of object-oriented software in general, and to this thesis in particular. These principles are: *data encapsulation*, *data abstraction*, *polymorphism* and *inheritance*.

3.1.1 Data Encapsulation

Normally in a computer program, separate invocations of functions have no way of affecting each other without saving information in state variables preserved externally from one function invocation to the next. This makes it difficult to build software modules since state variables either need to be referenced globally or passed explicitly as arguments to each invocation of

[1] Many thanks to Mike Powell for giving me this amusing article.

a function. Making such variables globally accessible prevents a software module's boundaries from being enforced since any arbitrary function can refer to this data as well. *Data encapsulation* techniques[AR84] are used to increase program modularity, maintainability and reliability by localizing data together with functions that operate on them and restricting access to the data exclusively to these functions. Data encapsulation techniques allow state to be preserved between invocations of functions. This state is stored in the encapsulated data.

Own data in Algol[Nau63] and *static* data in C[KR78] limits access to a particular function. *Packages* in ADA[Uni81], *named common* in FORTRAN[CDG70], *block scoping* in languages like Pascal[JW89], and *static file scope* in C limit access to a set of functions. The latter is often more desirable since modularity considerations make it important to allow a set of functions to share a common set of data that is hidden from other functions. The enqueue and dequeue operations on a queue, the push and pop operations on a stack, and the add and lookup operations on a database are all examples of sets of operations that share common (localized) data. Encapsulation techniques can provide more than just data hiding. *Monitors*[Hoa74, Bri85], for example, are a form of data encapsulation that provides for synchronization and mutual exclusion of accesses to the encapsulated data as well.

The *object* is the unit of data encapsulation in the object-oriented paradigm. An object is a simple encapsulation consisting of a set of variables and a set of operations used to alter and access them. The variables will be referred to as the object's *instance variables*. An individual operation in the operation set will be termed a *message* the object *accepts*. Invoking an operation on an object will be described as *sending a message* to the object. The entire set of messages accepted by an object will be termed its *signature*.[2]

Messages used in the context of the object-oriented paradigm should be contrasted with messages in the message-passing systems discussed in Section 2.4.4. The phrase "send a message to an object" is only a statement of how the object-oriented paradigm should be conceptually viewed. It does not imply the implementation of message sends is actually achieved by creating messages that are exchanged via explicit send and receive primitives between an object and the sender of the message. What sending a message in the object-oriented paradigm implies is that a *method* (the code performing the operation the message describes) is looked up based on the object and *invoked*. Sending a message to an object, therefore, results in a *method lookup*

[2] The term *protocol* is also often used.

to find the proper method and then a *method invocation* to call the method. In most object-oriented languages, method invocation is achieved with traditional procedure calls. In other words, a *message* is part of an object's interface, while a *method* defines the implementation of the message.

Like other data encapsulation techniques, objects preserve state between message sends that invoke their methods. This state is preserved in an object's instance variables. Ideally, an object's methods provide the only means by which other objects in the system can access its state and perform operations upon it. The difference between objects and other data encapsulation techniques is mainly one of perspective. In object-based data encapsulation schemes, each object is identified by some name, or *reference*, to which messages are sent. In other data encapsulation schemes, the functions are often the "first class" entities. In these schemes, data analogous to an object's instance variables are explicitly passed to a function or referenced in an enclosing scope. Programming with objects is, therefore, a more *data driven* encapsulation approach than function based encapsulation schemes.

3.1.2 Data Abstraction

Building *models* to reduce problems to less complex ones is a common problem solving technique. Models use *abstractions* of entities being manipulated. These abstractions allow irrelevant details to be ignored and focus to be placed on the essence of a problem. Abstractions also allow the details of unstructured entities to be hidden by a framework through which they can be manipulated easily.

Computer programs can be viewed as models. The values operated on by statements in a computer program are abstractions of the entities being modeled by the computation. In the object-oriented paradigm, all values are objects. Many objects represent similar abstractions and, therefore, share similar or identical behavior. In order to express the commonalities of identical objects, the object-oriented paradigm introduces the concept of a *class*. A class is a "template" to create a kind of object. It defines the instance variables, signature, and implementation of the messages in the signature (methods), for *instantiations*, or *instances* of the class. A class is, therefore, a *data abstraction* that simultaneously specifies the interface and encapsulates the implementation of each of its instances. [3] Classes are similar to *clusters* in CLU[LAB+81] and *object records* in Path Pascal[CM78].

The methods of a class can reference the instance variables that the class defines. These instance variables are bound at runtime to those of a particular instance of the class.[4] This runtime binding of class methods to the data they operate on gives a class a "cookie cutter" like behavior. Late binding is also what distinguishes object-oriented programming from other abstractions and encapsulation techniques. When programming with classes, functions (methods) are bound at runtime to the data (instance variables) they operate on. With other encapsulation schemes, data are usually compile time bound to functions. This means that there can be only one instance of such an encapsulation.

Expressions within a program generate values which are usually assigned to variables or used in other expressions. *Type checking* is used to determine whether values generated by expressions are used in the proper context[Set89]. Most object-oriented languages rely solely on *signature equivalence* for type checking. Signature equivalence means that the signature (or a subset of the signature) of an object used at a location in the program matches the signature required. A *type safe* operation is one where a message being sent to an object is in that object's signature.

There are two forms of type checking: *static* and *dynamic*[Set89]. *Statically typed* languages require compile time determination from the programs source of whether a value conforms to the type implied where it is used. For example, that + is applied to a pair of integers, reals, or perhaps a combination of the two. Usually, static typing is assured by requiring the programmer to specify a type for each variable and function result and having the language specify the types which result from the application of built in operators and functions. In object-oriented programming, these types are specified as signatures which any object assigned to the variable must have. In many object-oriented languages, this signature is specified with a class. The signature of this class then implicitly defines the type of the variable. Other object-oriented languages allow signatures to be specified independent of classes[JGZ88].

[3] As will be seen in the next section, this is not strictly true. Inheritance allows classes to delegate some or all of this responsibility to other classes.

[4] In most languages names like self or this usually refer to this object within the context of the implementation of a particular method.

In *Dynamically typed* languages, runtime checking is used to check a value's conformance to a particular type within a particular context. Dynamic typing, in the context of object-oriented programming, requires that variables are typeless and can reference objects of any type. Dynamic and static typing have software engineering and performance tradeoffs. The explicit type information given for variables in a statically typed language can aid in both the documentation and compilation of the program, while dynamically typed languages may lead to code that is more flexible and easier to reuse. Dynamically typed languages are usually less efficient since they lack type information to guide compilation. Type inferencing systems[Gra89] can improve the efficiency of dynamically typed languages by allowing types to be determined at compile time and corresponding optimizations to be made.

Orthogonal to static and dynamic typing is whether a language is *strongly* or *weakly* typed[Set89]. Strongly typed languages allow only type safe assignments to be made – the value being used *always* conforms to the type required. Weakly typed languages either do not check types or allow the checking to be overridden.

3.1.3 Inheritance and Subclassing

Classes arise naturally from the similar behavior of groups of objects. Likewise, different classes may have many messages and methods in common. *Class inheritance*, first introduced in SIMULA[BDMN73], is a technique to describe this commonality. Inheritance allows a set of classes to share parts of a common interface (signature) and perhaps parts of a common implementation (methods and instance variables). *Subclassing* is the best known inheritance mechanism. With subclassing, part or all of the signature, methods and instance variables of a class can be inherited from ancestor classes. Single inheritance allows one ancestor class. Multiple inheritance allows for several ancestor classes. Inheritance relations between classes form directed acyclic graphs and, therefore, are usually termed *class hierarchies*. The child classes of a given class in a hierarchy are usually termed its subclasses and are said to be *derived* from the class. The ancestors of a class are usually termed its *parent* or *super* classes. Delegation[Lie86], where objects forward messages to other objects, is another way to inherit behavior. Subclassing is the more commonly used sharing technique and is concentrated on in this thesis.[5]

Inheritance makes customization and incremental refinement possible in a structured manner by allowing a class designer to address the problem put best as:

> *"The class I need is* almost *like the one I have, except for..."*.

The implementor of a new class has the option of augmenting the inherited signature with additional messages and/or redefining the implementation of any inherited methods. Methods corresponding to added messages can likewise either be specified by the class or left to subclasses of the new class to be defined. Classes that define only a signature and leave the implementation to other classes to define are usually termed *abstract* classes. Classes that define an implementation for a particular signature are termed *concrete* classes. Abstract classes define a data abstraction, concrete classes define or refine implementations of this abstraction.[6] In practice, many abstract classes fall somewhere between purely abstract and concrete; they often provide methods for some, but not all the messages in their signatures.

The separation of interface from implementation provided by using abstract and concrete classes is important when designing for portability. Abstract classes can define interfaces that are implemented by different concrete classes for different applications. The separation also makes the system easier to understand since abstractions can be designed separate from their implementations.

3.1.4 Polymorphism

Sending a message to an object makes certain assumptions about the object. In particular, it assumes that the object's signature contains the message being sent. Violations of this requirement can be detected at compile-time in statically typed languages and at runtime in dynamically typed languages. For example, an object referenced as a **File**[7] will be expected

[5] There is even argument that delegation and subclassing are simply different views of the same principles[Ste87].

[6] Because of this, often when describing object-oriented programs, designers use the name of an abstract class when they mean that class or any concrete classes implementing the interface.

[7] A convention used throughout the rest of this thesis is that class names will have their first letter capitalized, while message names will not and will have their names in sans serif font. Another convention will be to use class names in **bold** face type. Plural nouns will be used to indicate collections of instances of a class. The indefinite article will be used to indicate a single instance that belongs to the collection. For example, rather than using

to accept read, write and seek messages. Likewise, a **Directory** would be expected to accept addFile, removeFile and lookup messages, and a **BitmappedScreen** setPixel and clearPixel messages.

Since the actual parameters to a method may be of many different types, the method may behave differently depending on the arguments. This is implemented in object-oriented programming by the delayed binding of messages to the methods that implement them. In particular, when a message is sent to an object, the actual method invoked is not determined until runtime. It is determined by the object's class. This runtime method lookup underlies the abstract/concrete class structuring techniques discussed in the last section.

The set of messages sent to an object in a given context is usually only a subset of the messages defined by the object's class. This is especially true of objects received as parameters to a method. For example, a method with a formal parameter to be used as a file from which to read data might only send read and seek messages to the actual parameter. The actual parameter object may have many more messages defined, such as write, close, etc. Likewise, different implementations of file objects may exist in the form of instances of different classes. Any of these would be acceptable arguments to the method in question. A method that can accept arguments of many different types and behave differently is said to be *polymorphic* with respect to those arguments.

The polymorphism discussed so far is more specifically a form of *inclusion* or *bounded polymorphism*[DT88]. Inclusion polymorphism restricts polymorphism to objects that share a common representation or signature. *Signature-based* inclusion polymorphism is prevalent in object-oriented languages. For this reason, unless otherwise specified, the rest of this thesis will use unqualified polymorphism to mean signature-based inclusion polymorphism. In signature based inclusion polymorphism, if the target object has the desired signature as a subset of its messages then it can be used in a given context.

Other forms of polymorphism include: *functional polymorphism* and *operator overloading*. Function polymorphism is where the type of the arguments to a function are irrelevant (the identity function is the classic example). This is also termed *parametric polymorphism* in [CW85], although Danforth[DT88] distinguishes parametric polymorphism as the case where

"instances of class **Widget**", "**Widgets**" will be used when referring to objects instantiated from the **Widget** class. Likewise, "a **Widgit**" will be used to indicate the singular case.

types are *explicitly* provided to a function, and functional polymorphism as the case where they are *implicitly* determined. Operator overloading is the classic form of polymorphism seen in traditional programming languages. The way in which the built in $+$ operator can operate on integer/integer, real/real, integer/real or real/integer arguments in most programming languages is an example of operator overloading.

Polymorphism of formal parameters has restriction. Each formal parameter of a method has an implicit signature, S, that consists of the set of messages that will be sent to the parameter within the method and within any method to which the argument is further passed. The argument used to satisfy a polymorphic parameter must satisfy the requirement that it accepts the messages defined by that parameter's S. In statically typed object-oriented languages, the signature of every argument is explicitly specified as one having all the messages in S as a subset of its messages. In a dynamically typed language, any object having the messages in S as a subset of its messages can be used as an argument, but the checking is deferred until runtime.

Polymorphism allows for a large degree of flexibility in design and reconfiguration of object-oriented software. It is crucial to designing reusable code. Polymorphism allows the addition of new components and the replacement of existing components since new classes implementing old signatures can be used with existing code that expects those signatures. For example, if an object having a file-like signature (**read**, **write**, and **seek** operations) is needed in a certain application, an instance of any class defining these methods would suffice. The actual object could be a local file, remote file, disk drive, or tape drive. As long as it has the desired signature, the particular kind of object is unimportant and irrelevant to the code using it.

Because object-oriented languages use signature equivalence for type checking, different restrictions can be placed on the polymorphism of the parameters of methods. Dynamically typed languages implement polymorphism in its pure form as discussed so far. This is because any actual argument object will be checked for type conformance at runtime. Statically typed languages come in two forms: those that specify formal parameter types as classes, and those that specify formal parameter types as signatures. Statically typed languages that specify formal parameter types as signatures can implement polymorphism in its pure form. Since in a statically typed language the type of all values can be determined, the signatures can be compared at compile time to check for conformance of the actual parameter to the formal

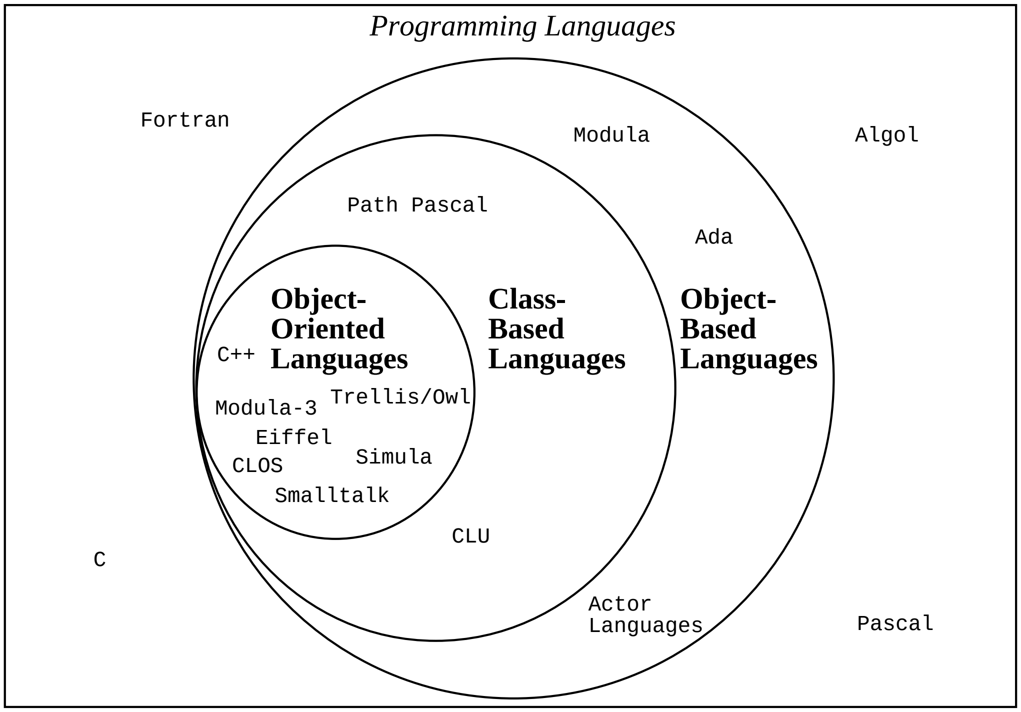

Figure 3.1: Types and examples of programming languages

parameters specified type. Statically typed languages that specify parameter types indirectly with a class implement a restricted form of polymorphism. Instances of a specified class or any subclass of that class are the only acceptable arguments, even if instances of other classes might have the signature desired. This behavior will be termed *inheritance polymorphism*.

3.2 Definitions

The following sections add two definitions necessary for the presentation of the work in this thesis. These are definitions of an *object-oriented language* and of an *object-oriented system*.

3.2.1 Object-Oriented Language

Wegner[Weg87] defines an *object-oriented programming language* as an object-based language that supports both classes and inheritance (see Figure 3.1). This definition is in some ways too simple as it does not explicitly mention the need to support polymorphism, which is essential to achieve the benefits of using object-oriented programming. By not using one feature (virtual functions), C++[Str86], a common object-oriented language, would still meet this definition of object-oriented. However, it would lack polymorphism since there would be no runtime binding of messages to methods. A better definition is, therefore, that an object-oriented language fully supports programming in the object-oriented paradigm by providing objects as the unit of data encapsulation, classes as the unit of data abstraction, inclusion polymorphism, and some form of sharing of interfaces and method implementations, either by delegation or inheritance.

Wegner cites ADA, Modula, CLU and Actor languages[Agh86] as examples of object-*based* languages (but not object-oriented), C++ and Smalltalk[GR83] as examples of object-oriented languages.[8] Other languages fitting the definition of object-oriented include Eiffel[Mey88], Trellis/Owl[SCB+86], SIMULA[BDMN73], CLOS[DG87] and Modula 3[CDG+89].

3.2.2 Object-Oriented System

A software system will be defined to be a body of software that is constructed to provide service to other software that relies upon it. A software system must provide an interface to other software that will use the services it provides. Usually this interface takes the form of a small set of subroutines or functions that, when invoked, request services from the system and/or return results. Software systems are distinguished from simple programming libraries by preserving state between invocations of their interface functions, and by interface functions affecting each other's behavior. Operating systems are perhaps the best examples of software systems. Systems providing windowing capabilities on a bit-mapped screen or a database with access/update routines are other examples.

An *object-oriented software system* will be defined by two characteristics. First, an object-oriented software system should be constructed using object-oriented techniques. This means it should be implemented as a dynamic collection of objects, where each object is an instance of some class representing a logical entity the system is modeling. Inheritance and polymorphism should be used to organize the classes and their interrelations. In particular, polymorphism and inheritance should be used to facilitate code sharing, interface sharing, code reuse, and reconfiguration of the system. To simplify using inheritance and polymorphism, an object-

[8] Many object-oriented languages implement the paradigm with varying degrees of completeness. C++ for example, allows accesses to an object's instance variables by functions other than the object's methods. This violates the encapsulation dimension of object-oriented programming.

oriented system should be implemented in an object-oriented language. Second, an object-oriented system should provide its services by exposing a select set of constituent objects to external (dynamic) message sends. Taking the windowing system as an example, representative objects might be a window, a pointing device, a title bar, or a scroll bar.

A object-oriented system needs to provide a mechanism for applications to obtain references to service objects and to send messages to them. Sending messages to objects replaces the interface functions of a traditionally structured software system. Such a system should also allow additions to the system to be made by inheritance from existing classes and it should support the polymorphism achieved by the late binding of message sends to methods. In short, an object-oriented system is open and extensible within its object-oriented framework.

The Smalltalk environment[Gol84] is perhaps the archetypal object-oriented system. It is a collection of objects that provide a complete programming environment, it is written in an object-oriented language (Smalltalk itself), and it allows new application software written in the environment to have access to any other object already in the environment. The limitation of Smalltalk is that all applications that exist in the environment must be written in Smalltalk.

3.3 Summary

In summary, four important features of the object-oriented paradigm are:

- *Data Encapsulation* to increase program modularity, maintainability and reliability by localizing data together with functions that operate on them. Data encapsulation in the object-oriented paradigm is focused on an *object* identified by a name, or reference, to which messages are sent.

- *Data Abstraction* to identify the types in a program. Data abstraction in object-oriented programming is directed at identifying *classes* to create replicated instances of objects with similar behaviors.

- *Inheritance* to allow code sharing (reuse) between classes and the enforcement of common interfaces.

- *Polymorphism* to provide code reuse by allowing methods to be written that take objects of many different types as arguments.

An object-oriented language is a language that provides support for these four features. An object-oriented system is a software system designed and constructed using the object-oriented principles and implemented in an object-oriented language, and that provides an interface to its components (its objects and classes) in an object-oriented fashion (sending messages to them).

Using objects as an encapsulation technique is simply good programming methodology. Likewise, using classes as an abstraction technique is good design. The object-oriented principles of inheritance and polymorphism make programming in the paradigm worthwhile. Combined, they provide for *sharing of common interfaces*, *code reuse*, *customization*, and *optimization*. They can help address the problems of operating system *portability*, *maintainability*, and *extensibility* discussed in Section 2.3. This is the concentration of the next chapter. It defines an *object-oriented operating system* and outlines the benefits of constructing such systems. The remaining chapters evaluate these benefits in light of the experimental system mentioned in Chapter 1.

Chapter 4

Object-Oriented Operating Systems

The previous chapters have concentrated on the necessary background definitions and principles for the presentation of the rest of this thesis. This chapter turns to my primary concern, namely, the application of the object-oriented paradigm to operating systems. First, I give the definition of an *object-oriented operating system* assumed in the rest of this thesis. I will use this definition to characterize the experimental system proposed in Chapter 1 and help distinguish this work from others. Next, I present guidelines for, and the benefits of, applying the object-oriented principles discussed in Section 3.1 to solve the operating system design and construction problems discussed in Section 2.3. Finally, I conclude with a short introduction to the experimental system itself. In the following chapters I will present this system in detail, analyze its performance, and discuss its success at addressing the problems facing operating system software set forth in the introduction.

4.1 Definition of an Object-Oriented Operating System

First, and foremost, an object-oriented operating system is an object-oriented software system as defined in Section 3.2.2. Like any object-oriented system, *all* entities in an object-oriented operating system are represented by objects that are instances of representative classes. This includes encapsulations of hardware devices, traditional operating system entities such as processes and files, system data structures managing resource allocation and management, system policy modules, and, in particular, low level operating system data structures such as page tables and device control registers. Examples of candidate classes for such objects include:

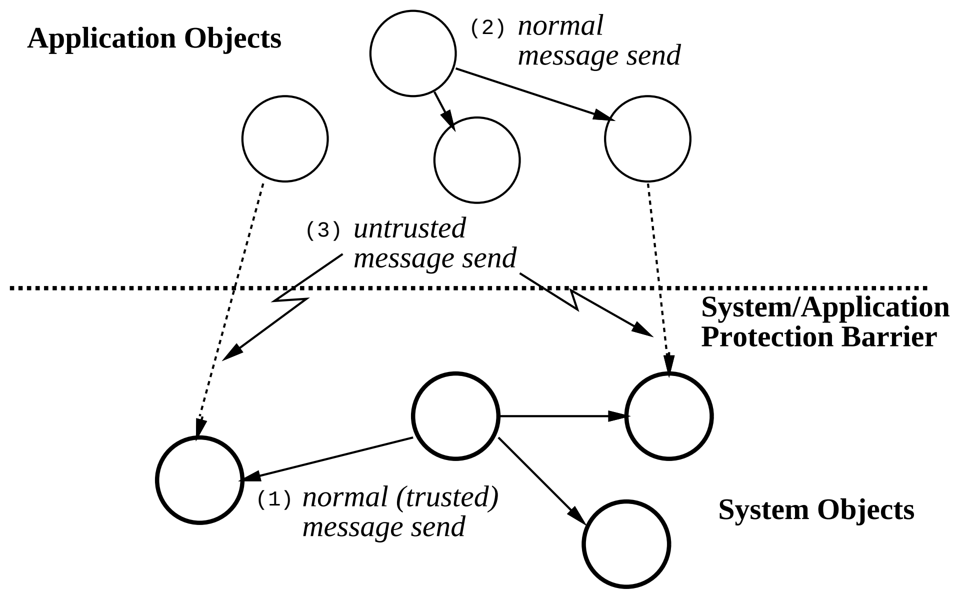

Figure 4.1: Method invocation in an object-oriented operating system

Processor (to represent a physical processor), **Process** (to represent a logical thread of control), **PageTable** (to represent a hardware page table), **VirtualMemoryRange** (to represent a range of contiguous memory), **Scheduler** (to represent a process scheduler), **File** (to represent a permanent storage file), **DeviceRegister** (to represent a physical device register), and **PhysicalMemoryFrame** (to represent a physical memory page frame).

Operating systems provide a set of application interface primitives to allow delayed binding of application requests to the operating system functions that implement desired services and to insure integrity of system data and functions. The main feature distinguishing an object-oriented operating system from traditional operating systems is that, like any object-oriented system, it provides its primitives as message sends to system objects. This implies that sending a message in an object-oriented operating system has one of three characteristics (see Figure 4.1). All message sends within the system itself (on the system side of the system/application barrier) behave as described in Chapter 3, the corresponding method is invoked with a normal procedure call. All message sends within an application behave likewise. The interesting case arises when an application sends a message to a system object. The invocation of the corresponding method must cross the system/application barrier to provide the protection and dynamic binding to system services characteristic of the barrier. Unprotected object-oriented operating systems can implement this with an extra indirection on the method invocation associated with a message send to a system object. A protected object-oriented operating system must implement method invocation on system objects in such a way that privilege levels can be crossed in a transparent manner (Chapter 5 describes several ways to implement this).

An object-oriented operating system needs a naming mechanism to provide applications with references or capabilities (see Section 2.4.5) to system objects. This is similar to how references to servers must be obtained in message-passing systems. One solution is to provide every application with a predefined reference to a *name server* object that maps symbolic names to references. Queries to this object return references to other objects, that in turn provide specific system services. The name server defines the set of all system services available to an application in the form of a set of objects. The classes of these objects, in turn, define the application interface of the operating system for this application. This interface can be extended or reduced dynamically by adding objects to, and removing object from, an application's name server.

Object-oriented operating systems may impose static or dynamic typing on system object references. Typing can be integral to the implementation of the name servers. In a dynamically typed object-oriented system, queries to a name server require only the name of the object and return a reference to the object if it exists no matter what the type of the object. Type checking is deferred until messages are sent to the object. Statically typed systems do not allow messages to be sent to untyped references. The type of the object being referenced must be confirmed before message sends are allowed. Support for static typing can be achieved in two ways. First, name servers can provide untyped references to objects, but the references are not usable until they are *narrowed* to a particular type. After narrowing, a new reference is constructed that may be used as the target of a message send with no further checks. If the object is not of the proper type, the narrow operation should fail. Second, untyped references can be prohibited. The type of the object being looked up can be included as an argument to the name server query operation. The name server will only return a reference if an object of the given name exists *and* it is of the type requested. In this way, once a reference is obtained, it can immediately be treated as a typed reference and message sends can proceed normally. This is how the experimental system described in this thesis implements its interface.

4.2 Advantages of Object-Oriented Operating Systems

The previous section defined an object-oriented operating system, this section concentrates on the advantages of designing and implementing object-oriented operating systems. It focuses on how such systems address the problems of operating system portability, efficiency, maintainability and extensibility discussed in the introduction. During the presentation of the experimental system proposed in Chapter 1 in the following chapters, concrete examples of the techniques discussed in the following sections will be given to support these claims.

4.2.1 Portability Advantages

When trying to design portable operating system software, it is important to isolate dependencies on particular devices or architectures in modules that are separate from, and opaque to, *architecture independent* portions of the system. These *architecture dependent* modules provide an interface to the entity that they are abstracting and encapsulating. They allow programmers to design and implement the rest of the system in such a way as to only rely on using this interface, while remaining unaware of its exact implementation. Ideally, such modules can be reimplemented when the software is retargeted for a new architecture without any modifications to the rest of the system. This allows architecture independent portions of the software to be reused as the system is retargeted to new architectures. If this is to be true, each architecture dependent module created as the software is retargeted for a new architecture must have the same interface as the modules for other architectures. Modules representing hardware abstractions create a platform upon which the rest of the operating system can be constructed much like the virtual machines discussed in Section 2.4.3. In particular, they can be used to represent hardware entities that might differ in detail from architecture to architecture, but still perform a similar function. Examples are page tables, processor registers and device controller registers.

Object-oriented programming provides a convenient framework within which to support such needs. Abstract classes can be used to define the signatures (interfaces) of such modules while concrete classes can implement these signatures for various architectures. Creating abstract classes to define interfaces to architecture dependent entities allows operating system algorithms and data structures to be constructed without detailed knowledge of the specific hardware being used. Polymorphism allows code that relies on the interface to use instances of any of the concrete classes implementing the signature as target objects. If the abstract interface is properly defined, only a new concrete class needs to be created to encapsulate architecture dependencies when retargeting the operating system to new computer hardware.

As a specific application of this technique, consider the page tables used by the dynamic address translation hardware of many architectures. A **PageTable** class can be constructed to abstract a hardware page table. Its signature should define the essential operations on a page table. For example, likely messages for such a class to accept are addTranslation(virtualAddress, physicalAddress), removeTranslation(virtualAddress) and getPageSize(). Each address space can use an instance of a concrete class implementing this signature to represent its virtual memory mappings. This form of abstraction allows such details as where particular bits in a page table mapping entry lie, or whether the page table is one, two or $\mathcal{N}$ leveled, to be isolated from other objects which reference and manipulate page tables. Concrete classes can implement the **PageTable** signature for a particular hardware's page tables (for example **VAXPageTable** or **68000PageTable**).

As another example, consider the interface provided by a device driver for a mass storage disk device. An abstract **MassStorageDevice** class with read, write and seek messages might be sufficient. This signature can be implemented for various kinds of hardware disk devices by concrete classes. The complexity of the concrete implementations of the interface will depend on the particular hardware device. Intelligent disk controllers with firmware device drivers will need only minimal classes to encapsulate their firmware defined operations. Hardware supporting only direct access to a disk controller's registers may need a complex class that manages disk head seeks, data transfer initiations, and interrupt responses to such transfers. This complexity can, however, be completely hidden in the concrete class. System engineers who specialize in writing and designing such software need only be told of the abstract interface needed. Their particular expertise and knowledge can be applied directly to creating a concrete class implementing the signature. They are freed from being forced to understand the client software using their classes.

4.2.2 Code Sharing Advantages

As discussed in the last section, most operating systems contain machine dependent modules that are reimplemented for each target architecture. Traditionally structured system usually specify such modules in terms of a set of operations (functions) to be provided. However, without inheritance, similar implementations of this interface must be independent and have no way to share common code. For example, the Mach operating system's **pmap** system[R+87] is such a module definition. By their very nature of being machine dependent, the eighteen operations that the pmap system defines need to be reimplemented, or at least modified, for each architecture to which Mach is retargeted. Implementations for similar architectures have no convenient way to share code. Even interface sharing is only by convention and not enforced by any compiler or language support.

The code sharing and reuse possibilities facilitated by class inheritance can easily solve this problem. In particular, concrete subclasses implementing abstract signatures need not be totally unrelated. Often a new device or processor architecture will be only a little different from an existing one. Similar architectures should be expected to result in similar implementations. Commonalities can be abstracted into a new class and the differences can be represented by subclasses of this class.

Returning to the page table example in the previous section, consider two architectures that both use similar page tables and have the same size pages. It is likely that the placement of a few bits in a page table entry may differ between these architectures, but the majority of algorithms doing page table management will likely be identical. Programming this commonality in the object-oriented paradigm is simple. A new abstract class meeting the **PageTable** signature can be created to encapsulate all the commonality for handling the similar architectures. Subclasses of this class can each encapsulate specifics about a particular architecture. All these classes still have the **PageTable** signature and, therefore, instances of any of the (concrete) subclasses can be used anywhere an instance of such a representative class is expected. The common code between both architectures is shared through the superclass. As an added benefit, any changes made to the superclass, for example a bug fix or performance improvement, will automatically get inherited by the concrete subclasses. This allows all the classes to take advantage of the change without the programmer having to make it multiple times. Inheritance also aids extensibility since, if a third architecture is introduced with much the same commonalities, only a third subclass of the abstract superclass needs to be constructed.

Another reason that code sharing can benefit operating systems is that there are often similar data structures implementing various operating system policies and mechanisms between versions of an operating system, or within a version. An example of this situation can be found in various versions of the UNIX operating system. The UNIX file system is localized around the concept of an *i-node*[Bac86], which maps logical blocks of the file onto physical disk blocks and localizes other information about the file such as access protection bits and the file's size. The two major version of UNIX, System V[SVI85] and BSD[BSD84], both implement i-nodes in subtly different ways while maintaining an almost identical behavior. Support for such a situation can be provide by constructing an abstract class that defines the interface and localizes any common behavior (and if possible the representation) of the data structure, and using concrete subclasses to localize the differences[MLRC88, Mad91]. For example, in a class hierarchy of UNIX file systems representations, the commonalities and signature can be abstracted into a **UNIXInode** class, with **SystemVInode** and **BSDInode** subclasses encapsulating the peculiarities of both systems respectively.

The basic principle motivating when code sharing can be used in any object-oriented application is that similar architectures and data structure should beget similar implementations. Inheritance allows grouping of the commonalities while localizing the differences.

4.2.3 Separation of Policy from Mechanism

When trying to design extensible operating system software, it is important to separate policy decisions from the mechanisms that implement them[PS85]. Just as architecture dependencies can be isolated inside modules, policy decisions with characteristic interfaces can be identified and modularized. This allows different policies to be enforced as needs change by replacing a particular policy module.

In the same way that inheritance and polymorphism support portability by allowing easy definition of interfaces that are implemented in many ways, they likewise support isolation of policy decisions. Policy decisions with characteristic interfaces can be represented with abstract classes to define the interface. Concrete classes can then implement the interface for specific policys.

Process scheduling in a multiprogrammed operating system is a good example of an application of this technique. The mechanism involved is quite simple: idle processors need a way to get the next process to run from a set of processes ready to execute. The policy decision involved is the ordering of the set of processes ready to run, i.e., when a processor becomes idle,

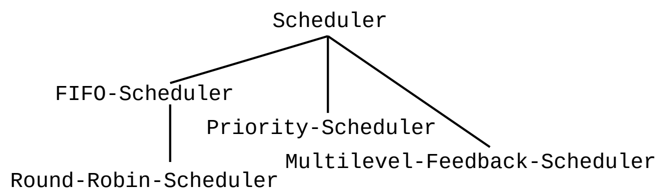

Figure 4.2: A sample class hierarchy

what is the process that will be assigned to the processor next. Figure 4.2 shows a sample application of object-oriented programming techniques to the problem in the form of a hierarchy of classes derived from an abstract **Scheduler** class. The **Scheduler** superclass encapsulates any commonalities, such as the queue head and tail, and defines the signature that an abstract **Scheduler** presents. For example, the signature will likely contain an `enqueue` message for adding a process and a `dequeue` message for removing the next process. Each subclass provides the same signature as the parent **Scheduler** class but implements its policy in different ways by defining different orderings on processes enqueued and dequeued. For example, the **FIFO-Scheduler** class may dequeue processes in first-in-first-out order, and the **Priority-Scheduler** may dequeue processes in an order derived from their priority. Instances of any of these classes can be used where a **Scheduler** is required, for example as the system's ready queue. A FIFO scheduling discipline could be imposed on processes by using an instance of the **FIFO-Scheduler** class. By replacing that object with an instance of the **Priority-Scheduler** class, the scheduling policy of the system could be modified to be priority based. Note that this can be achieved by only replacing a single object, i.e., by changing a single line of code in the implementation. It could also be done dynamically by changing a single reference at runtime.

As another example of separating policy from mechanism, consider page replacement algorithms in a virtual memory based operating system. If an abstract **VirtualMemoryManager** class is constructed to represent the locus of information about page placement and replacement decisions, concrete classes implementing the signature can provide different strategies while leaving code using the interface unaltered. For example, a **FIFO-Manager** class might define the replacement algorithm to be first-in-first-out, while a **NUR-Manager** class might define it to be a not-used-recently[Dei84a] policy. Other common operating system policy decisions that benefit from this approach include disk head scheduling, serial line character processing and file system access methods.

4.2.4 Optimization by Subclassing

The examples in Section 4.2.2 show how inheritance allows specialization while at the same time achieving the advantages of code sharing. Inheritance and polymorphism combine to support another important advantage, namely, structured optimization.

Performance is essential to an operating system. Operating system algorithms are constantly refined and optimized. It is important to keep any optimizations from hindering the maintenance and evolution of the code. Object-oriented programming gives a good framework within which to support such optimizations. Abstract classes can define interfaces which, for performance reasons, can be implemented in different way by concrete classes that impose different constraints on their use.

As an example, consider an operating system constructed of multiple processes all concurrently executing. When exchanging control between processes, the state of the currently executing process must be saved, and the state of the new process restored on the processor. A logical way to design this in an object-oriented operating system would be to develop a **Process** class that has save and restore messages to support the state saving and restoring. However, consider the case described in Section 2.4.2 where some processes execute on the behalf of the operating system itself and some on behalf of application programs written to use the system. If processes executing on the system's behalf are more "lightweight" than the application processes, this is to say they have less state and resources associated with them, the implementation of the save and restore methods can be optimized. A traditional way to implement this is to store a flag with each process that specifies the kind of logical process it represents. The save and restore operations can discriminate upon the flag to decide the amount of information to save and restore. Object-oriented programming provides a simpler and more efficient solution. The **Process** class can be made abstract with **SystemProcess** and **ApplicationProcess** concrete subclasses. The **SystemProcess** subclass's methods can save and restore only the state a system process uses, while the **ApplicationProcess** subclass's methods save and restore the additional state application processes need.

Using subclassing to encode special case testing like this avoids both reserving the extra space in the object for the flag, and testing the flag upon every invocation of a method that relys on its value. Instead, the value of this flag is implicitly encoded in the object's class. Message sends automatically cause the proper method to be invoked. Thus, object-oriented programming accomplishes the desired result with less expense in both time and space.

This is just one example of a more general result in well designed object-oriented systems: the elimination of discriminating on an explicitly coded type identifier. The binding of a particular message to the method implementing it is based on a object's class and performed at runtime. Often this delayed binding can even be more efficient than explicitly coding the type codes and discriminating on them[RK88].

4.2.5 Trading Portability for Efficiency

Inheritance and polymorphism can also combine to provide an interesting solution to the problem of trading portability for efficiency. An inefficient but portable class can be used during the initial phases of retargeting an operating system for a new architecture, and later replaced with a machine specific class that improves efficiency but decreases portability. For example, consider a **MemoryBlock** class with messages to zero-fill (bzero) and copy (bcopy) the block of memory. The corresponding methods can be implemented in a portable way in most object-oriented languages with simple loops. However, without a sophisticated compiler that performs such optimizations as loop unrolling and specialized code sequence recognization, such code will likely be less efficient than hand tailored assembly code. An instance of this class could, however, be used initially in any retargeting effort of the operating system in order to minimize the work needed to get the system running. Once the system works, it can be tuned by replacing instances of the portable class with instances of a new class that implements the signature to perform the copy and zero-fill operations in the way most efficient for the particular target architecture, for example, with architecture specific instructions.

4.2.6 Component Testing

With respect to maintainability, object-oriented operating systems benefit from all the advantages of any object-oriented system. In particular, the narrow interfaces to data defined by objects makes testing and debugging of modules easier[Fie89, PG90]. Object-based encapsulation schemes ease debugging and testing by allowing the construction of simple test cases to exercise an object's interface. Object-oriented programming makes it possible to reuse even test cases. A test case can be constructed to exercise the interface described by an abstract class and multiple concrete classes can be plugged in and evaluated/debugged. As new classes implementing the abstract signature are implemented, they can be tested and debugged without creating new test cases.

4.2.7 Support for Adaptable Interfaces

One way to extend an operating system is to add new components to the application interface. Object-oriented operating systems are especially well suited for this. The application interface of an object-oriented operating system is defined by the set of objects made available to an application by its name server and the classes of those objects. The set of available objects can change dynamically as the needs of an application change. This can be accomplished by modifying, adding or removing bindings of names to objects in an application's name server. Different applications can even be assigned different name servers, allowing different applications to have different interfaces to the operating system.

The application interface of an object-oriented operating system is actually defined by the *classes* of the objects accessible through a name server. A potential service can be defined by the signature of an abstract class. A specific service can be provided by an operating system object that is an instance of a concrete class implementing the signature. Polymorphism allows an application compiled to use the abstract interface to remain independent of the specific class of an object implementing that interface. Therefore, both new classes, and new instances, can be added to the system and bound to name servers. Applications may access objects that are instances of classes that were not even in existence at the time the application was compiled if these classes implement existing interfaces in new ways. This can occur statically by recompiling the operating system with new classes and objects or, ideally, dynamically with a tool that allows new classes to be loaded into the operating system and new instances of those classes to be created and bound into name servers at runtime.

4.2.8 Mutual Exclusion and Synchronization

Object-oriented and object-based programming techniques have many advantages when applied to parallel and concurrent systems. Since the object-oriented principle of encapsulation allows only the methods of an object access to the state of the object, an object can control the exclusivity of accesses to its state through its methods. Thus an object can provide a "safe" interface to its instance variables. This idea is similar to Hoare's monitors[Hoa74], except that monitors guarantee mutually exclusive access only and preclude potentially concurrent access. Concurrent accesses to an object's data might be desirable in a multiprocessor or multiprogrammed operating system to increase performance. In object-oriented programming, the implementor of a class is free to code arbitrary restrictions on the ordering and mutual exclusivity of methods using semaphores, locks, or other similar techniques.

For example, consider an atomically incremented counter. This can be implemented in a non-object-oriented fashion in two ways. First the burden of mutual exclusion can be placed on the users of the count. For example:

```
P( mutex );
counter = counter + 1;
V( mutex );
```

Or, the burden can be placed in a special function that localizes the synchronization, and the code using the count be required to call special functions to update the counter. For example:

```
– assuming call by reference semantics...
IncrementAtomicCount( counter : integer )
{

    P( mutex );
    counter = counter + 1;
    V( mutex );
}

– A sample usage
integer : counter;

IncrementAtomicCount( counter );
```

The problem with the first solution is that programmers cannot always be relied upon to implement the mutual exclusion correctly. Similarly, the second solution suffers from the problem that programmers cannot be relied upon to always call the IncrementAtomicCount method and not just increment the counter themselves. Another problem with both solutions is that existing functions that take count arguments will have to be modified if they are to be used in parallel cases, or non-parallel as well as parallel cases will have to pay the mutual exclusion costs by compiling only one version of a function and using one of the above two solutions.

The object-oriented solution to such a problem is quite simple. If an abstract **Count** class with increment and value messages is introduced, one concrete class can implement the signature for parallel applications, and one for non-parallel applications. The non-parallel version can implement increment as a simple non-atomic add one operation, while the parallel version can implement the needed mutual exclusion much like the second case above. The advantage of this technique is that polymorphism allows generic code that was written and compiled to only rely on the interface provided by the abstract **Count** class to be used both efficiently (without the added cost of the mutual exclusion) in non-parallel cases, and safely (with the added cost for mutual exclusion) in parallel cases.

A technique like Campbell's path expressions[CH74] applied to an object-oriented languages could simplify the problem of reliably coding mutual exclusion between and within an set of messages. Complex requirements like:

> "The initialize message must be sent before the read and write messages, that can be sent in any order, but must eventually be followed by a send of the cleanup message, after which no read or write messages may be sent until a subsequent send of initialize."

can be easily specified and implemented by synchronization at method entrance and exit. This type of ordering is very difficult to enforce in traditional programming paradigms. Usually, one is forced to explicitly code complex locking protocols directly into the invoking routines.

4.2.9 Efficiency

With all the benefits of using object-oriented techniques discussed so far, the natural question to address next is the cost (in terms of efficiency) of using object-oriented programming techniques to construct an operating system. Obviously, the answer to this question depends heavily on the particular object-oriented language chosen. Of the languages discussed in Section 3.2.1, Smalltalk and CLOS are unlikely to be useful for operating system construction for performance and/or runtime support reasons. Work by Johnson[JGZ88] to develop a compiled Smalltalk capable of producing high quality stand-alone code should change this in the future. C++, Eiffel, and Trellis/Owl are probably all reasonable choices. Each is a statically typed object-oriented language allowing the production of efficient code. Modern statically typed object-oriented languages manage to provide object-oriented features with little extra cost over normal procedure calls [Ros87]. Static typing should not be underestimated as a valuable software engineering advantage to the language as well. It allows better documentation by specifying applicable argument types explicitly.

Another cost associated with an object-oriented operating system is the overhead of accessing system objects across the system/application barrier. Querying name servers may be expensive since it may involve parsing names, but this is only a start up cost since it is performed only to obtain a reference to a system object. Also, this cost is not much different than the similar costs of looking up files or devices through the file system of a conventional operating system. Once the reference is obtained, any performance expense reduces to the cost associated with actually using the reference as the target of a message send. Mechanisms to implement this efficiently will be discussed in detail in Chapter 5, but this obviously involves crossing the system/application barrier. The cost of crossing the system/application barrier is likely to dominate any other messaging cost, and is also incurred in a conventional operating system when invoking an operating system primitive.

4.3 Are any Existing Operating Systems Object-Oriented?

The question "Are there already any object-oriented operating systems?" should be asked. If such systems exist, they could be evaluated in terms of the operating system problems discussed in Section 2.3 to see how well the proposed advantages of object-oriented operating systems are realized in practice.[1]

Conventional operating systems such as UNIX and VMS[Dei84b] fail to provide any notion of a object-oriented services and are designed and implemented in traditional manners.

Message-passing systems like Mach[TR87] and V[Che84] provide a messaging interface to ports or processes (servers). These systems provide a kind of object in the form of their servers, but take no advantages of language supported classes or inheritance to structure and reuse parts of servers. The messages in a message-passing system are not structured or typed. A server does not necessarily define the set of messages it may be sent. Message-passing systems do, however, provide a form of polymorphism in that the binding of client messages to servers is dynamic. Since the receiver of a message may forward the message to another receiver, inheritance could be simulated. This is similar to the way in which inheritance is implemented in interpreted object-oriented languages and suffers similar performance penalties. In particular, the deeper the simulated inheritance hierarchy, the more the message forwarding and, therefore, the worse the performance.

Non-language supported object-based systems like those discussed in Section 2.4.5 have explicit objects. The binding of references to objects can be dynamic, giving a polymorphic behavior as with message-passing systems. However, without language support for inheritance, the burden of code and interface reuse is placed directly on the programmer in the form of programming conventions. Another disadvantage of non-language supported object-based operating systems is that such systems tend to implement the method invocations corresponding to message sends with some form of remote procedure calls [JLHB87]. This makes message sending expensive and, for the reasons discussed in Section 2.4.5, possibly prohibitive for implementing very low level operating system abstractions. Implementing all abstractions as objects is desirable from a software engineering point of view. Object-based/object-oriented languages gain their advantages by providing small, light weight objects with method invocations built on ordinary procedure calls. If an operating system is implemented with such a language, the notion of an object has to be extended to encompass objects outside the scope of a single program. Section 8.8 discusses approaches to extend the light weight object model across a distributed

[1] Not to mention that if such systems exist, the experimental system proposed in this thesis may be unnecessary or could be compared to them.

system. This allows both low level system abstractions and high level remote objects to be efficiently implemented. Thus, distributed systems are seamlessly integrated.

Operating systems that provide an object-oriented interface to applications but are not themselves internally constructed as object-oriented systems (like the interfaces to the Mach operating system provided by the NeXT computer system interface[NEX]) allow the benefits of object-oriented techniques to be realized by application programmers, but deny them to operating system programmers and designers. The true value of an object-oriented operating system is obtained in the software engineering benefits realized by applying object-oriented techniques to the construction of the operating system software itself.

The Xerox Cedar system[SZBH86], is an object-oriented system and meets the requirements closest. It, however, was not a protected operating system and failed to implement many modern operating system features.

The SOS system[Sha88] implements many of the object-oriented techniques discussed so far but concentrates on high level objects and does not apply the object-oriented principles to the lowest levels of the system.

The definition of an object-oriented system presented is believed to exclude existing systems. The experimental system presented in this thesis is believed to be the first system meeting all the criteria proposed. It is a protected, multiprogrammed operating system implemented completely as an *object-oriented software system* and providing an *object-oriented application interface.*

4.4 *Choices*: the Experimental Prototype

In the introduction to this thesis I described an experiment in which I would design and implement critical portions of an object-oriented operating system in an attempt to measure the success of object-oriented techniques at solving problems of operating system extensibility, maintainability, portability and performance. This chapter so far has defined the characteristic of such an object-oriented operating system and has presented some of the potential benefits of constructing one. The rest of this thesis will focus on the experiment itself and how well object-oriented techniques live up to the proposed benefits. The experimental system is called *Choices*[CRJ87]. Chapters 5 through 7 will present the design, implementation and performance details of *Choices*. But first, in the remainder of this chapter I will introduce *Choices* and discuss its implementation.

4.4.1 Attributes of *Choices*

Even if object-oriented techniques live up to the benefits claimed so far for simple systems, they are relatively useless if they cannot be used to construct systems of the complexity of modern protected, multiprogrammed operating systems. Therefore, *Choices* was built to see if these techniques scale up and work for such complex, hardware dependent systems. *Choices* in its current form runs on the Encore Multimax[Enc89] symmetric multiprocessor, the Apple Macintosh IIx personal computer, and the AT&T WGS–386 computer. In particular, *Choices* provides:

- Full multiprogramming support for tasks composed of multiple processes sharing a common memory. Any number of such tasks can execute simultaneously.
- A hardware enforced system/application barrier supporting message sends to system objects.
- Complete support of virtual memory with a choice of page placement and replacement policies.
- Support for multiple arbitrary backing stores and arbitrary associations of virtual memory ranges to those backing stores.
- A fully reentrant kernel capable of parallel execution on multiprocessors.
- Light weight context switching of concurrently executing processes capable of arbitrary memory sharing with each other.
- Access to a variety of industry standard disk file formats.

Implementing a system of this complexity is, perhaps, beyond the capability of one person. The entire *Choices* operating system is actually a product of multiple researchers at the University of Illinois. I am primarily responsible for the design and implementation of the subsystems of *Choices* described later in this thesis. These include: the virtual memory management system, the process management and scheduling system, and the object-oriented interface. These three subsystems provide enough examples to support the claims for applying the object-oriented paradigm to operating system design and implementation presented in this thesis. Other major implementation contributions to *Choices* include: work by Peter Madany and Doug Leyens[MCRL88] on the support for multiple file system types; work by Gary Johnston, Aamod Sane and Ken McGregor on the support for distributed virtual memory; more work by Gary Johnston on some of the overall "glue" holding things together; and work by Bjorn Helgaas, Panagiotis Kougiouris, Dave Dykstra and Ruth Aydt on identifying portability problems in the design and implementation by retargeting *Choices* for additional architectures. Roy Campbell as the manager of the *Choices* project and contributed to the design in many ways.

4.4.2 The *Choices* Implementation Language

As discussed in Section 4.2.9, the programming language used to implement an object-oriented operating system can drastically affect its performance. The implementation language chosen for *Choices* was C++. This is mainly due to the advantages of statically typed object-oriented languages discussed in Section 4.2.9, along with the added advantage that, although they violate the pure object-oriented paradigm, C++ allows certain low-level programming techniques necessary for the easy and efficient implementation of an operating system. In particular, the language allows the programmer to specify an object's representation in memory, to place objects at a specific address, and to predetermine the size of an object. Specifying an object's representation in memory is necessary to allow classes to represent hardware defined entities such as device or processor control registers and device command/control messages. It is also necessary in order to allow data structures specified by certain standards, such as the representation of a file on disk or the placement of fields in a network protocol packet, to be encapsulated within objects that are instances of representative classes. The ability to specify a new object's location in memory is necessary to allow addressing hardware specified entities as objects once representative classes have been designed. Again, this includes entities like device registers or hardware defined data structures that are often at a fixed location in memory. Finally, the ability to precisely determine the size of an object is useful to optimize memory allocation/deallocation for frequently instantiated classes.

C++ does not always faithfully implement the object-oriented paradigm. It can, however, be used as an object-oriented language and allows an examination of the advantages of object-oriented programming applied to operating systems. C++ supports objects, classes, inheritance and polymorphism. However, every value in a C++ program is not an object. This is a concession to simplicity of code generation and optimization since, in particular, primitive types such as integers, floating point numbers and characters are not objects that are instances of representative classes. Having such primitive types built in to the language allows the compiler to generate traditional code for operations on such types. Specifically, since no method lookup is done for such operations straight one-to-one mappings to machine code exist and can be expanded in line in the generated code. Another violation of the pure object-oriented programming is that C++ allows direct accesses to instance variables although this mechanism does not have to be used.

Perhaps the biggest drawback of C++ is that it implements inclusion polymorphism only in the form of inheritance polymorphism (see Section 3.1.4).[2] This is mostly a concession to efficiency and, along with static typing, allows C++ to implement a very fast method invocation scheme (see Section 5.1.1 for the detail of C++ method invocation). However, it forces the programmer to constrain the type hierarchy to the class hierarchy. I.e., a concrete class implementing a desired signature *must* be a subclass of the abstract class defining the signature.

4.5 Summary

An object-oriented operating system applies the object-oriented paradigm uniformly. It has internal system components that are represented by objects, structured by inheritance, and made available at the applications interface by supporting message sends that cross the system/application barrier in a protected manner.

Numerous advantages exist for constructing object-oriented operating systems. These include:

- Increased portability.
- Increased code and interface sharing and reuse.

[2]To get even this behavior, methods used in argument classes must be implemented as C++ *virtual functions*[Str86].

- Support for easy separation of policy from mechanism.
- Optimizations through specialization.
- Support for a structured way to trade portability for efficiency when necessary.
- Support for component testing.
- Support for adaptable interfaces.
- Efficient and localized support for mutual exclusion and synchronization.
- Support for efficient system construction.

The following chapters present the important parts of the design of *Choices*. Special emphasis is placed on how object-oriented design principles have aided both its design and implementation and how well the above benefits have been realized in practice. Specifically, the topics of memory management, process management, and application interfaces are discussed in detail. The memory management and process management chapters also illustrate how well modern operating system techniques can be mapped into the object-oriented framework provided by *Choices*.

Chapter 5

Application Interfaces

5.1 Overview

The definition of an object-oriented operating system set forth in Chapter 4 requires *Choices* to provide a mechanism that allows application objects to send messages to select system objects, while maintaining the system/application barrier described in Section 2.1 to protect other system objects.

Most modern, protected operating systems enforce their system/application barrier by placing all operating system data and functions in memory that is made inaccessible to applications by the memory management system. Special instructions are used to cross over this protection barrier and access system functions. In an object-oriented operating system, objects encapsulate system data. Messages can only be sent to objects that are accessible. However, accessibility (or the lack there of) cannot be enforced without some form of language or compiler support for separate logical address spaces, or hardware support for object capabilities[NW77, MB80]. Language and compiler implemented protection schemes are not available for C++. Only traditional protection mechanisms (virtual memory and privileged/non-privileged execution modes) are available on the hardware platforms chosen for developing *Choices*. Therefore, the application interface of *Choices* is designed to only rely on conventional hardware protection mechanisms to limit object accessibility. All system objects are placed in memory made inaccessible to untrusted applications by the *Choices* virtual memory management system. A mechanism that allows messages to be sent to system objects in a protected manner is provided to cross the system/application barrier. The implementation of this mechanism uses a *proxy* *object* technique[Sha86]. *Choices* allows ordinary C++ programs, compiled to access ordinary C++ objects, access to system objects without changes to the compiler or the application. At the same time, it allows detection and interception of incorrect accesses to system objects.

Support for the object-oriented interface of *Choices* centers around the **ObjectProxy** class. **ObjectProxys** are indirect references to system objects. Although system objects are normally inaccessible to applications, **ObjectProxys** *are* addressable by application programs. Messages can be sent to an **ObjectProxy** using the normal C++ method lookup and invocation scheme. To an application, an **ObjectProxy** appears to be the system object itself. Messages sent to the **ObjectProxy** provide the same results as if they were sent to the system object itself. In reality, messages sent to an **ObjectProxy** are transparently forwarded to the actual system object they represent. This occurs after entering the operating system in a controlled manner (i.e. raising the privilege level of the processor via an SVC like operation) and verifying that the operation being performed is valid.

ObjectProxys are initially obtained from *name servers* that convert symbolic names into system object references. The abstract **NameServer** class defines the interface of classes that perform this function. **NameServer** defines the bind message to insert a mapping of a symbolic name to a system object and the lookup message to convert a symbolic name into a proxy of the object it names (if that object exists). Only trusted system routines can add new bindings to a **NameServer**. Concrete classes implementing the **NameServer** signature are free to implement these operations in any way desired. Each *Choices* application is assigned a **NameServer**. Since the only system objects accessible to an application are those for which **ObjectProxys** can be obtained, a *Choices* **NameServer** completely defines the application interface for the applications to which it is assigned.

Before the implementation of **ObjectProxys** can be explained in detail, it is necessary to briefly discuss how C++ implements object method invocations.[1]

[1] More information about the C++ language can be found in [Str86].

5.1.1 C++ Method Invocation

C++ associates with each object a reference to a list of its methods.[2] The class of an object determines the contents of the list. Multiple instances of a class can share the list. A class reserves an entry in the list for each of its methods. A subclass uses a copy of the superclass's list and replaces the entries corresponding to redefined method with the addresses of the new methods. If the subclass defines new methods, it extends the list.

Sending a message in C++ proceeds by indexing into the list to look up the desired method and invoking the method with a normal procedure call. In order to allow the method access to the object's state, a reference to the object is passed along as an extra (implicit) parameter. The table is indexed by a fixed offset per message.[3] Since C++ is a statically typed language, the offsets are assigned at compile time.

This form of method lookup and invocation allows C++ to provide the object-oriented feature of delayed (runtime) binding of messages to object methods with the cost of only a pointer dereference, a table index, and a procedure call. Versions of C++ supporting multiple inheritance have slightly additional costs for method invocation. In particular, an adjustment of the reference to the actual object data is required. The C++ compiler used for *Choices* supports multiple inheritance and, therefore, imposes this penalty. However, Stroustrup[Str89] claims that this cost is not a constant cost for all method invocations and can be eliminated if multiple inheritance is not used.

For example, consider the C++ class declaration in Figure 5.1. The C++ compiler creates a method lookup table for such a class with the addresses of the buy and sell methods. Figure 5.2 shows the actual method lookup table generated by the C++ compiler used to compile *Choices* on the Encore Multimax (g++ version 1.37.1[Tie90]). It consists of an array of entries, each with three fields. The first two fields (the pairs of zeros) are used to support multiple inheritance. They determine the adjustment to be added to the implicit reference to the object data when a method is invoked. These fields are irrelevant to the rest of this discussion. The

[2] It is assumed by the *Choices* implementation that all the methods of a class that have instances available as application accessible system objects are *virtual functions* in C++ parlance. Likewise, the data (instance variables) of a proxied class are assumed to be *only* accessed through these methods.

[3] While small variations exists, the mechanism discussed in this section is, in general, true for all C++ compilers I have seen to date. While it is not the only mechanism available to a C++ compiler writer, its use has become so wide spread that it is rapidly becoming the "standard" way to implement method invocation for C++.

```
class Widgit {
     int value;
     int cost;

     virtual int buy( int number );
     virtual int sell( int number );
};
```

Figure 5.1: A sample C++ class declaration

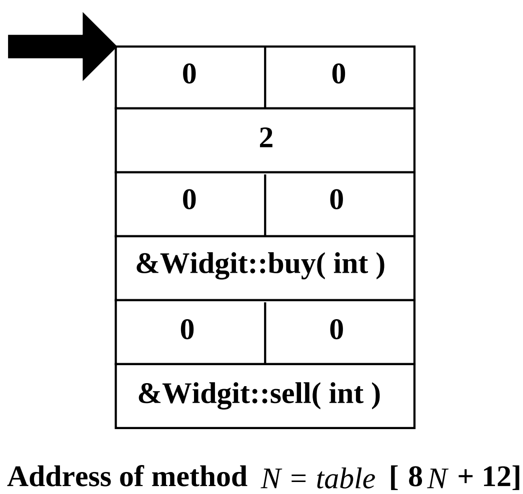

Figure 5.2: The method table for the **Widgit** class

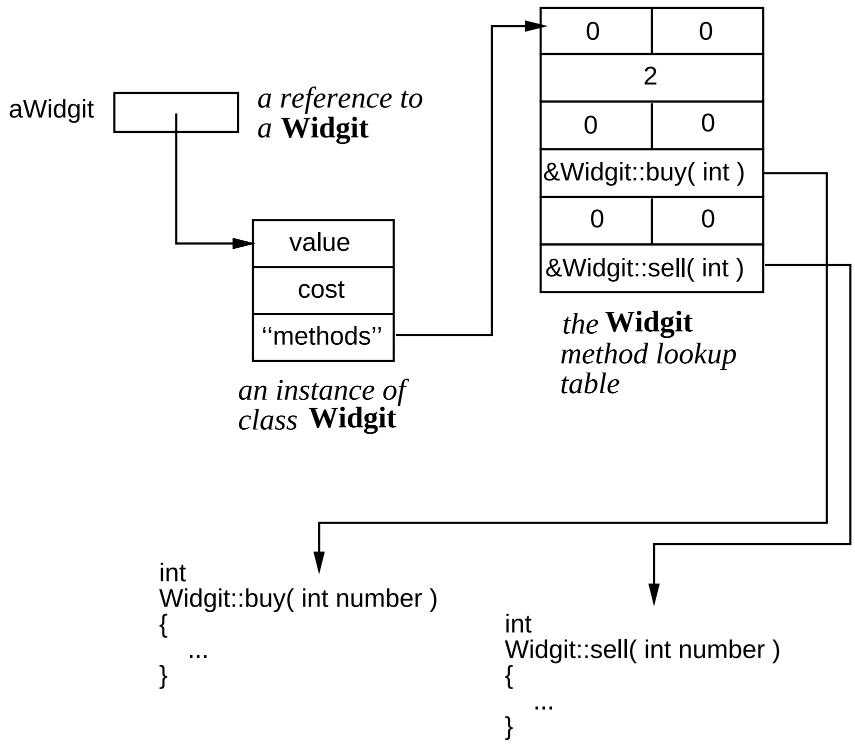

Figure 5.3: C++ method invocation

third field is the address of a method. One exception is that instead of the address of a method, the first entry in the table has as its third field the number of entries following it in the table. This is the number of methods the class has defined.

Each instance of a class references the class's method lookup table. (see Figure 5.3). If, for example, the buy message is sent to a **Widget**, the address of the method table is fetched from the object and the address of the buy method is found in the second entry in this table. The function at this address is then called with the arguments to the buy message and the implicit parameter representing the object itself as parameters. Once the function implementing the method has completed, control and results are returned to the instruction following the message send using the normal procedure return mechanism.

5.1.2 Proxied Method Invocation

An **ObjectProxy** behaves as an indirection to a system object. An **ObjectProxy** has two instance variables: a reference to a system object (realObject), and a flag indicating if the **Ob-**

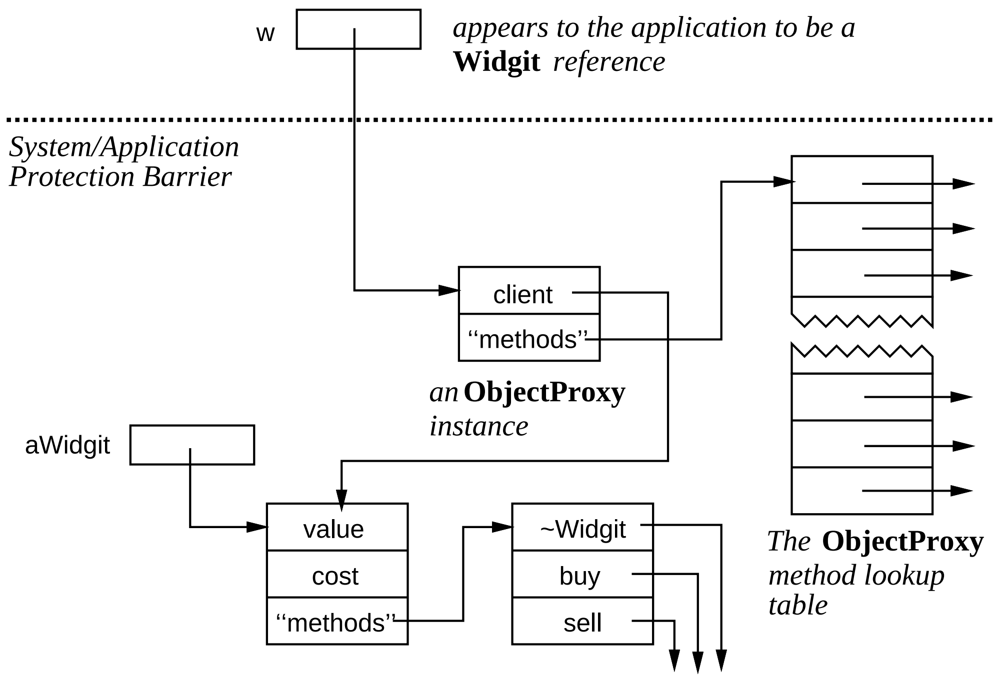

Figure 5.4: Using **ObjectProxys** to access system objects

jectProxy is valid (validFlag). What distinguishes **ObjectProxys** from other system objects is that they and their method lookup tables are kept in memory that is readable, but not writeable, by applications. This lets applications send messages to an **ObjectProxy** without being able to modify it. Each method of the **ObjectProxy** class provides a controlled entry into the operating system. The code implementing these methods, like the **ObjectProxys** themselves, is kept in memory readable (executable) but not writeable by application programs. Thus, application programs can fetch the address of the method table from an **ObjectProxy**, fetch the address of a method from this table, and branch and execute the code; but they cannot modify the **ObjectProxy**, its method table, or its methods in any way (see Figure 5.4). To the application programmer and C++ compiler, it appears that programs can perform normal C++ method invocations on system objects represented by **Objectproxys**. For example, a method that was compiled to send the print message to an object passed in as an argument will work if the actual parameter is either a system object (**ObjectProxy**) or a normal application object. A set of **ObjectProxys** can be kept in memory shared between a set of applications. These applications can then share access to the corresponding system objects.

All methods of **ObjectProxy** cause entry into the privileged execution mode in order to allow the real system object the **ObjectProxy** represents to access other system objects and data. This is accomplished with the help of an architecture specific trap instruction that saves the application state, raises the privilege level, and branches to a trap handler within the operating system. The trap handler decides which method was being invoked either by which trap was taken, or by arguments passed to the trap handler by the **ObjectProxy** method implementation. The **ObjectProxy** the message was sent to can be determined from the saved application state. Since the **ObjectProxy** contains a reference to the real object, the trap handler can obtain that as well. Once the real system object and the desired message are determined, the trap handler copies any arguments out of the saved application state and sends the message to the system object with those arguments. All this occurs in the privileged operation mode so that, in effect, the operation is invoked on the object as if it were done by operating system code being executed by the process. The target system object, therefore, has access to other system objects and data without further checks. Once the method returns, results are placed back in the saved application state and the trap handler returns back to the application after resuming non-privileged execution.

If an application passes an **ObjectProxy** as an argument to a message being sent to another **ObjectProxy**, that argument is transparently usable by the system object that receives the message. Messages may be sent to this **ObjectProxy** from within the operating system. For efficiency, this technique depends on using as the privileged instruction to trap into the operating system, the equivalent of a *change mode to privileged execution* instruction. If the processor is already executing in privileged mode, such an instruction is a no-op. If the processor is executing in non-privileged mode, it causes the desired trap into the operating system. **ObjectProxy** methods are implemented so that if the trap is not taken, the **ObjectProxy** is dereferenced to fetch the reference to the target system object, and a second message send is performed on that object. This occurs with the only overhead being an effective no-op instruction plus the cost of an additional method lookup and invocation. Most compiled object-oriented languages which support dynamic method binding are likely to implement something similar to the C++ table scheme. Therefore, if *Choices* were reimplemented in another language, **ObjectProxys** similar to the C++ implementation should be possible. To clarify some of the issues introduced above,

```
        .word    methodNumber
Entry:
        bicpsrw  U-Bit
        movd     realObjectOffset(r0),r0
        pushd    methodTableOffset(r0)
        movd     ((8*methodNumber)+12)(0(sp)),0(sp)
        ret      $0
```

Figure 5.5: A sample **ObjectProxy** method

the actual implementation of **ObjectProxys** for the NS32332 processor architecture[Nat86] is given in the next section.

5.1.2.1 An Actual Implementation of ObjectProxys

On the NS32332, the `bicpsrw` instruction is used to trap into the operating system. This instruction mnemonic literally means "**bi**t **c**lear in **p**rocessor **s**tatus **r**egister".[4] The argument to this instruction specifies which bits to clear. Clearing the *U*-bit (the 8'th bit position in the word) will attempt to switch the processor to the "system" protection level (the highest privilege mode). When the *U*-bit is *set*, the processor executes in the "user" protection level (the lowest privilege mode). Clearing the *U*-bit is itself a privileged operation. If the processor is already executing in the system protection level, clearing the *U*-bit has no effect, otherwise it causes a *privileged instruction trap* to occur. This is exactly the behavior needed to implement **ObjectProxys**.

The code implementing a sample **ObjectProxy** method is shown in Figure 5.5. The word *before* the entry point to the method stub is used by the trap handling code to determine which method was being invoked when the trap was taken. The `bicpsrw` instruction is the first instruction executed. It attempts to clear the *U*-bit. If the processor was already executing in the system privilege level, this instruction will have no effect and execution will continue with the next instruction. If the processor was executing in the user privilege level (executing

[4] The trailing **w** indicates the instruction operates on a *word* of data (2 bytes on this architecture), this word being the processor status register (psr) itself.

application code) then a trap will occur and the processor will invoke a trap handler. What happens in both these cases is discussed in the following sections.

No Trap Taken: If no trap is taken, the instruction following the bicpsrw will be reached. To understand what happens next, a quick description of the argument passing/procedure call mechanism used on the NS32332 by the C++ compiler used for *Choices* is needed. The first two arguments to any procedure are passed in r0 and r1 (general purpose registers zero and one) respectively. The C++ compiler passed the implicit reference to the object as the first parameter. Therefore, r0 contains this reference and r1 contains the first explicit argument to the message. The remaining arguments are pushed onto the stack. The jump-to-subroutine, jsr, (absolute address operand) and branch-to-subroutine, bsr, (relative address operand) instructions are used to effect the actual transfer of control. They both push the return address (the address of the next instruction) on the top of the stack and branch to the entry point of the function being called. Therefore, upon entry to the code stub above, r0 contains the first argument (the address of the **ObjectProxy** itself) and r1 contains the second argument (actually the first argument to the message in the C++ source code). The rest of the arguments and the return address are on the stack. The code in Figure 5.5 converts r0 into a pointer to the real system object, looks up the method for this object, and jumps to the method's entry point without disturbing either r1 or the contents of the stack. The one side effect is that the r0 register is altered to contain the address of the real system object rather than the **ObjectProxy** so that the method called will have access to the real object's data.

In detail, the first instruction after the bicpsrw instruction dereferences the **ObjectProxy** to obtain the value of the realObject instance variable. This reference replaces the **ObjectProxy** reference in r0, which is no longer needed. The next instruction dereferences the reference to the real object and obtains a reference to that object's method table, which is pushed onto the stack. The following instruction indexes into that table to find the method address (see Figure 5.2) and replaces the top of the stack with the address of the proper method to call. The final instruction simultaneously pops that address off the stack and branches to the instruction at the address (the entry point of the method). No *net* stack change occurs after this code sequence, so the return from the actual object's method will return to the *invoker* of this code (the location where the message was sent to the **ObjectProxy** in the first place) directly.

Since the stack and registers are provided to the method exactly as needed, any arguments are properly passed on without copying or interpreting them in any way. Likewise, values returned by the actual method will be passed back to the original invoker through the normal procedure return mechanism.

Confirmation that the **ObjectProxy** involved was not forged by an application when it passed it as an argument can be performed in a manner similar to that described in the next section. This is currently not done in *Choices*. If a system object sends a message to another system object, it is the sending object's responsibility to guarantee the validity of the **ObjectProxy**. The value of making this check the default behavior on every message sent to an **ObjectProxy** from within the operating system is still being investigated.

Trap Taken: If a trap is taken by executing the bicpsrw instruction, then the result is more complicated. The *Choices* trap and interrupt handling mechanism will be discussed in more detail in Section 7.7, but for this discussion it suffices to say that the mechanism saves the state of the executing application in a save area and sends the handle message to a trap handling object within the operating system. For the NS32332 implementation of *Choices* all privilege instruction traps are directed to the same trap handling object. The handle method of this object first determines why the trap was taken. If it is the result of attempting to clear the *U*-bit, the trap is assumed to be the result of invoking a method on an **ObjectProxy**, otherwise other trap handling code is executed. Since clearing the *U*-bit is a privileged operation, no application should execute this instruction during normal processing. Executing the instruction with random or incorrect register/stack contents in an attempt to provide a forged **ObjectProxy** and corrupt or crash the operating system will be caught by one of the checks discussed below.

Since the state of the application is saved, the trap handling object's handle method has access to the registers of the processor at the time of the trap. Therefore, the r0 register can be fetched to find a reference to the **ObjectProxy** to which the message was sent. With this reference, the actual system object can be found by accessing the realObject instance variable. The **ObjectProxy** is validated by first checking if it falls within the range of memory assigned to **ObjectProxys** for the invoking application. This test assures that the application does not maliciously supply a pointer to an arbitrarily created **ObjectProxy** *imposter* in an attempt to gain access to an otherwise inaccessible system object. The next test is to check the alignment of the **ObjectProxy** to prevent a malicious program from returning a random pointer into the **ObjectProxy** memory space. The final test verifies that the **ObjectProxy** is active (by checking the value of the validFlag instance variable) to prevent a malicious application program from supplying the address of an inactive **ObjectProxy**. Combined, these three tests guarantee that the **ObjectProxy** reference in the saved r0 register is valid.

In the NS32332 implementation, the message that was sent can be found by dereferencing a pointer into the code stream obtained by adding a constant negative offset to the address of the bicpsrw instruction (see Figure 5.5). This number can be compared to the number of methods the object has by looking in the object's method table (see Figure 5.2). The trap mechanism places the address of the bicpsrw instruction in the applications saved context. This address can be checked against a table of valid addresses to guarantee that a malicious application has not tried to supply an invalid method number by executing a random bicpsrw instruction with an erroneous value at a negative offset from the instruction's address.

Once the method number is obtained, the address of the object's method table and the address of the proper method can be obtained in a similar way as in the non-trapping case. The final step is to call this method with the same context as originally intended. This is accomplished by copying the arguments on the top of the application's stack onto the system's stack, loading the processors registers with the values saved at trap time, and branching to the method. In practice, a fixed number of arguments is copied regardless of the method. This number (five) was determined by analyzing the source code of *Choices* to find the maximum number of arguments any message ever takes. If the message being sent does not take this many arguments then no harm occurs since, in C++, the sender of a message pops the arguments off the stack and not the method invoked. It should be noted that it is the responsibility of the target system object's implementation to ensure that the arguments are correct for the method being invoked.

When the method returns, the current contents of the processor registers replace the values in the save area from the trap so that when the trap handler returns to the application, it will appear as if the actual object message send returned, and any results will appear in the proper registers. Likewise, the arguments copied onto the system stack are removed. Finally, the trap handler resumes the application at the point immediately following where the trap occurred and, therefore, the application continues execution at the point following where it sent the message.

5.2 Performance of the ObjectProxy Mechanism

The performance of the **ObjectProxy** mechanism will be evaluated by comparing it against "normal" C++ message sends and against traditional operating system service mechanisms. In order to give a basis for comparison, Table 5.1 first lists the overheads of C++ message sends (virtual function calls) versus traditional function calls for various numbers of arguments. These measurements, along with the others, were made on a 6 processor Encore Multimax with NS32332 processors each with a 15 megahertz clock. The measured overheads includes both the invocation and return from the call. In these and all the tests that follow, the functions (method) ignore their arguments. The arguments are only passed to give a measure of the cost of passing arguments, not the cost of using them. As can be seen in the Table 5.1, C++ message sends impose an additional overhead of 3 to $4\mu s$ over traditional function calls. This overhead is a result of the extra indirection caused by the method lookup in the object's method table and the overhead multiple inheritance (See Section 5.1.1) introduces. To put the additional 3 to $4\mu s$ in perspective, deferencing a pointer to a global variable costs $0.6\mu s$, an integer add of two variables in memory costs $1.60\mu s$, and an integer multiply of two variables in memory costs $6.80\mu s$. Any reasonable amount of computation performed by a method is likely to overwhelm the additional cost of the message send.

The effect of the compiler argument passing strategy can also be seen in Table 5.1. Since the first two arguments are passed in registers, the cost of function calls would be expected to rise dramatically at 3 arguments over two arguments due to the accessing of the stack to push the third argument. This is confirmed by the results. In the case of a message send, the jump is seen at 2 arguments rather than 3. This is because, as discussed in Section 5.1.1, the C++ compiler passes a reference to the object itself as an additional implicit argument to each method, thus, increasing the number of arguments by one.

Table 5.2 compares the *Choices* **ObjectProxy** mechanism to a traditional system service approach using SVC-like instructions. To put these numbers in perspective, the UNIX getpid() system call (that takes no arguments and returns a single integer), takes $98\mu s$ to complete on

Args	Message Send	Function Call
0	5.67 μs	1.93 μs
1	6.19 μs	2.47 μs
2	7.00 μs	2.40 μs
3	7.19 μs	3.55 μs
4	8.02 μs	4.61 μs
5	9.05 μs	4.62 μs

Table 5.1: Overhead of C++ message sends versus traditional function calls

Args	**ObjectProxy**	System Call
0	86.9 μs	96.3 μs
1	87.4 μs	96.1 μs
2	87.8 μs	96.7 μs
3	89.1 μs	98.5 μs
4	88.1 μs	99.9 μs
5	92.8 μs	100 μs

Table 5.2: Overhead of **ObjectProxys** versus traditional system calls

the same hardware. The *Choices* system call is implemented by loading a register with the desired service number and trapping into supervisor state. Arguments are accessed directly out of the saved application context without copying since every system call is custom written for the function it performs. The UNIX getpid() operation is supported by the UNIX kernel in a similar way.

The better performance of the **ObjectProxy** message send is a result of exploiting knowledge of the C++ method invocation convention. In particular, registers volatile across a message send do not need to be saved upon the trap into the operating system supporting the **ObjectProxy** method implementations. The implementations of UNIX system calls (and the corresponding *Choices* mechanism) need to save a significantly larger amount of state and impose approximately the same amount of overhead in either UNIX or *Choices*.

Since a fixed number of arguments are copied from the application context to the system context during an **ObjectProxy** invocation, the cost would be expected to be immune to the number of arguments to the message. This is not quite true since the caller must still push additional arguments on the stack. This explains the small rising expense of additional arguments to proxied message sends.

Table 5.3 measures the overheads of performing some of the same operations from within the operating system code itself. The interesting result in this table is the difference between

Args	**ObjectProxy**	Message Send	Function Call
0	10.8 μs	5.67 μs	2.07 μs
1	11.1 μs	5.91 μs	2.27 μs
2	12.3 μs	7.13 μs	2.33 μs
3	13.2 μs	8.06 μs	3.55 μs
4	13.6 μs	8.49 μs	4.39 μs
5	14.4 μs	9.45 μs	4.85 μs

Table 5.3: Overhead of **ObjectProxys** used from within the system

an ordinary message send and a message send to an **ObjectProxy** passed in as an argument from an application program. This difference (approximately $5\mu s$) reflects the extra message send incurred by using the **ObjectProxy**. As would be expected, the cost is independent of the number of arguments since no arguments are copied.

These results indicate that a reasonably robust object-oriented interface can be provided by an object-oriented operating system without significant degradation of performance for an application's requests for system services and without significant penalty for the system using the proxy mechanism as well.

5.3 Summary

Choices uses hardware memory protection to protect operating system data and functions from erroneous or malicious application programs. At the same time, the definition of an object-oriented operating system set forth in Chapter 4 requires *Choices* to provide a mechanism that allows application objects to send messages to select system objects, while maintaining a system/application barrier to protect other system objects.

The **ObjectProxy** mechanism allows messages to be sent to system objects without modifications to the C++ compiler or the application program itself. Code previously compiled to send messages to non-system objects will work equally well with system objects (**ObjectProxys**). The performance of the *Choices* **ObjectProxy** mechanism is comparable to the costs of traditional mechanisms for implementing operating system primitives. Therefore, the *Choices* application interface shows that the advantages of adaptable interfaces described in Section 4.2.7 can be achieved with costs comparable to traditional application interfaces.

Chapter 6

Memory Management

6.1 Overview

Most modern computer architectures provide hardware support for *virtual memory* [ADU71, BS88, Dei84a, PS85] to allow operating systems to support efficient and secure sharing of physical memory between multiple applications, to allow an application's address space to be larger than the physical memory present in the computer, to provide artificial data contiguity, and to protect operating system data and functions from an application. *Choices* uses such hardware support to implement its memory management system.

The goals of the *Choices* memory management system are to provide a portable virtual memory system that is compatible with a wide range of architectures, to provide a flexible and extensible model of virtual memory, and to do both efficiently. *Choices* extends and refines existing algorithms, such as those in UNIX[LMKQ89], Mach[R$^+$87], SunOS[GMS87], and VAX/VMS[LL82], to implement its object-oriented approach to the management of memory and secondary storage.

The memory management system of *Choices* provides:

- virtual memories that are composed of independent logical memories mapped into the address space.
- shared logical memories both within and between virtual memory address spaces.
- logical memories mapped into multiple, arbitrary virtual memory address ranges.

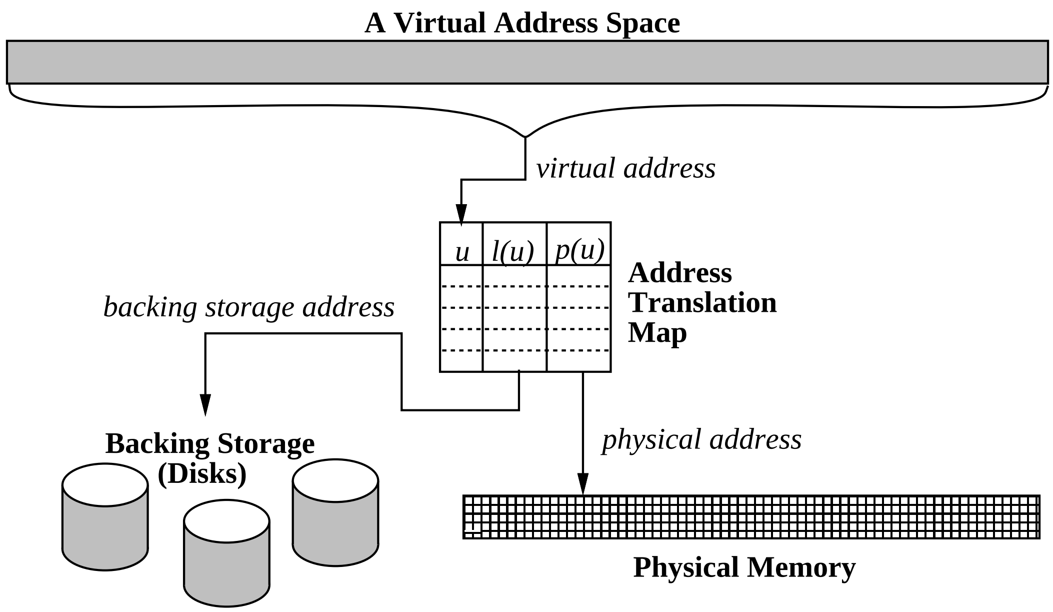

Figure 6.1: Traditional virtual memory management

- independent backing storage for each logical memory.
- a choice of different backing storage policies.
- alternate logical memory access through a file-like read/write access protocol.
- a physical memory representation scheme that provides an abstraction for exploiting scatter gather direct memory access hardware.
- a choice of local page placement and replacement algorithms for individual logical memories.
- a framework of abstractions and reusable software that can be used to build experimental virtual memory systems.

Choices departs from other systems by mapping such algorithms into the object-oriented framework it provides. This allows greater flexibility and customizability.

Conceptually, traditional virtual memory systems associate with each virtual address space a table mapping each unit of virtual memory, u, to a backing storage address $l(u)$ and a physical memory address $p(u)$ (see Figure 6.1). This table forms an inventory of virtual memory units specifying where they are to be found in physical memory or on backing storage. This table is actually implemented in most operating systems as two separate entities. The portion of the table mapping units to physical addresses is directly implemented in the hardware dynamic address translation tables, and the backing storage mapping is implemented in operating system data structures.

In an object-oriented operating system like *Choices*, it is natural to use objects to manage the mapping of addresses to backing storage and physical memory respectively. Many virtual memory implementations use a single backing store for an entire virtual address space (or even for all virtual address spaces). However, the *Choices* virtual memory scheme is similar to those used in MACH[R+87] and SunOS[GMS87], in which a virtual address space is divided into multiple regions with each region having its own backing store and physical memory map. Each region is treated as a *logical memory* that is cached in physical memory[1], and objects are assigned to manage each region's physical memory and backing storage management. A logical memory is basically a collection of related data aggregated together. A disk file is the best example, although an entire disk would also qualify.[2] Other examples include the code of an executing application, the data of an executing application, the control stack of a process, or an arbitrary collection of related objects.

Dividing the virtual address space into logical regions allows the *Choices* memory placement and replacement polices to take advantage of the characteristics of a logical range of memory. For example, references to memory holding a stack will exhibit strong spatial locality. Since each region has its own object representing its mappings to physical memory (see Figure 6.2), the replacement algorithm can simply discard memory furthest from the top of the stack yielding performance equal to a least recently used memory replacement algorithm without requiring the collection of access pattern information. Similarly, the division of the virtual address space into regions also improves locality of information about data placement on backing storage since the

[1] The concept of a *logical memory* is almost identical to a *segment* in architectures supporting segmentation. A segment would be ideal, but since *Choices* should run on traditional non-segmented (paged) architectures, the logical memory concept is the best that can be achieved. The main difference between a *Choices* logical memory and a segment is that addressing past the end of a logical memory does not cause an error as it would in a segmented architecture, but rather it enters the next logical memory in the address space.

[2] Since a logical memory occupies a set of virtual memory addresses, an entire disk might be too large to be mapped directly to a virtual memory range limited by the size of the hardware supported address size. For example, a five gigabyte disk could not be mapped into an address space with 32-bit addresses. Section 6.4.1 details an alternative solution.

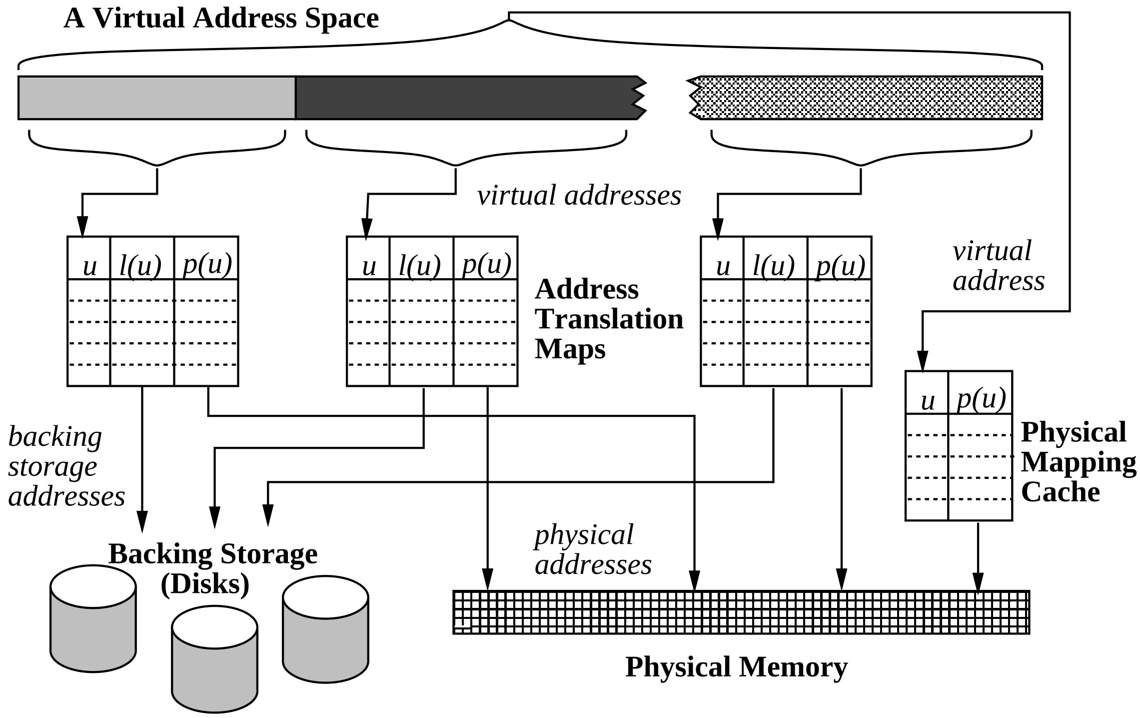

Figure 6.2: Virtual memory management in *Choices*

object representing each region's backing storage can manage its logical-to-permanent storage mappings in different ways.

Choices spreads the virtual to physical mappings across multiple objects. Unfortunately little, if any, computer hardware supports dynamic address translation tables in such a format. This problem is solved by introducing another object that represents the actual hardware dynamic address translation tables for a virtual address space (see Figure 6.2). One of these objects exists per virtual address space. This object maintains a cache of currently active virtual to physical mappings in the hardware tables. Non-cached mappings can be reconstructed on demand from the architecture-independent information kept by the rest of the virtual memory management system objects. This solution is similar to the pmap system in MACH[R$^+$87].

6.1.1 Overview of the *Choices* Memory Management Classes

The object-oriented framework constructed for the *Choices* storage and memory management system roughly divides the system into five sets of classes:

1. The classes that manage dynamic address translation hardware consist of **AddressTranslation**, which represents a single hardware dynamic address translation table, and **AddressTranslator**, which represents the actual hardware dynamic address translation unit (MMU) of an individual processor. Instances of **AddressTranslation** implement the $u \rightarrow p(u)$ physical mapping *cache* in Figure 6.2.

2. The classes that manage physical memory management consist of **Store**, **PhysicallyAddressableUnit**, and **PhysicalMemoryChain**. **PhysicallyAddressableUnits** each represent a single unit of physical memory (a page or segment frame). A **Store** represents the collection of all physical memory as **PhysicallyAddressableUnits** and manages their allocation and deallocation. A **PhysicalMemoryChain** represents a collection of **PhysicallyAddressableUnits** composing an arbitrary virtual address range as resident in physical memory, or a range of physical memory addresses for the I/O system.

3. Access to backing storage is implemented by the **MemoryObject** class. Each **MemoryObject** provides access to logically contiguous blocks of data stored on secondary storage. **MemoryObjects** represent backing storage of the logical memories discussed

earlier. **MemoryObjects** implement the $u \to l(u)$ mappings to backing storage in the address translation maps in Figure 6.2.

4. The transfer of data between physical memory and backing storage (**MemoryObjects**) is managed by **MemoryObjectCaches**. **MemoryObjectCaches** represent the $u \to p(u)$ mappings to physical memory in the address translation maps shown in Figure 6.2.

5. A complete virtual address space is represented by a **Domain**. Each **Domain** divides a virtual memory into multiple regions, each mapped to an individual **MemoryObject**. Each **Domain** has an associated **AddressTranslation** to effect its mappings in hardware.

During normal execution, the mappings maintained by an **AddressTranslation** contain the information necessary for the dynamic address translation hardware to convert virtual addresses to physical addresses. When an instruction references an address for which a mapping is not present, a fault occurs. At this point the current **Domain** is consulted to determine within which **MemoryObject** and at what offset within that **MemoryObject** the missing data falls. That **MemoryObject** is then sent a message to bring that data into physical memory. The **MemoryObject** passes the request on to its corresponding **MemoryObjectCache**. The **MemoryObjectCache** allocates new physical memory in the form of **PhysicallyAddressableUnits** and reads the data from the **MemoryObject**. Once the data is brought into physical memory, the **Domain** updates the current **AddressTranslation** with the new mapping and restarts or resumes the faulting instruction.

When physical memory is full, **MemoryObjectCaches** return some of their physical memory resident data back to the backing storage provided by their corresponding **MemoryObjects**. The proper **AddressTranslations** are updated to reflect the fact that the data is no longer present in physical memory so that a fault will occur if removed data is referenced again in the future.

The following sections explain in detail each class in the *Choices* virtual memory and backing storage management system and the messages they define to implement the system. Emphasis is placed on how object-oriented techniques have benefited the design and implementation of the memory management system.

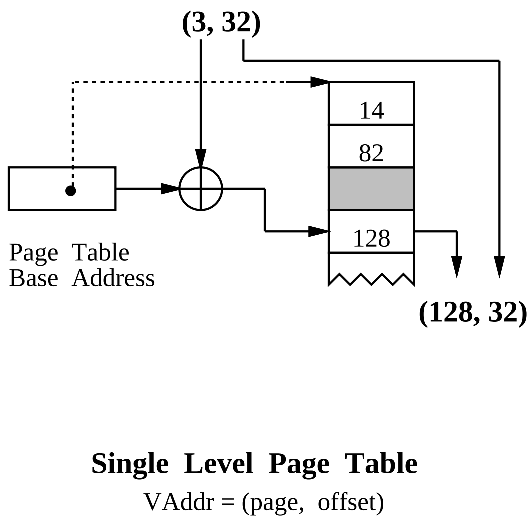

Figure 6.3: A single level page table

6.2 Dynamic Address Translation

Dynamic address translation hardware is the key mechanism that makes virtual memory practical. Across the spectrum of today's hardware, many different dynamic address translation mechanisms are employed. Examples of dynamic address translation hardware schemes include *simple page tables*, *multi-level page tables*, *translation lookaside buffers*, and *inverted page tables*.

Single level page tables (see Figure 6.3) decompose a virtual address into a pair (P_v, O) by dividing the bits in the address into two halves. The page size determines the number of bits dedicated to the page table index (P_v). The page table contains physical memory page frame addresses. The physical page number returned from the page table, P_p, is concatenated with the offset (O) from the original virtual address to form the destination physical address, (P_p, O). The problem with single level page tables is that for a large address space, and even moderately sized pages, the size of a page table can be quite large. The Digital Equipment Corporation VAX-11[LL82] computer line uses a single level page table scheme, but tries to reduce the memory occupied by the page table. The address space is divided up into regions. One region is set aside for the system and a single system page table (that resides in *physical* memory) maps this region. The *virtual* memory represented by this region contains the page

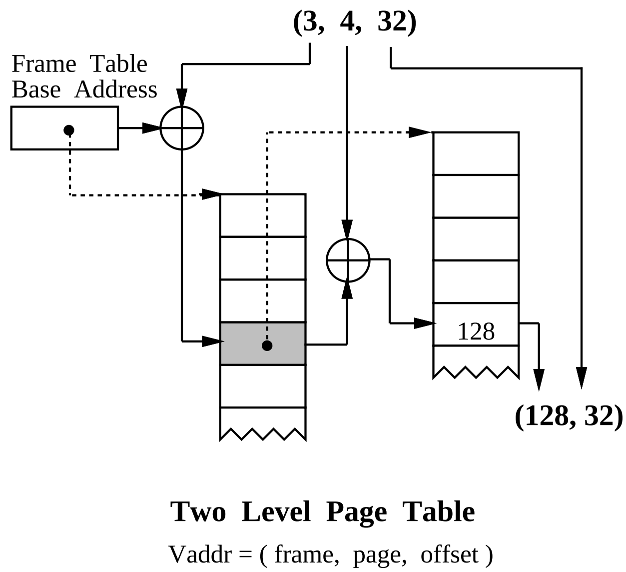

Figure 6.4: A two level page table

tables for the other regions of memory. Since the other region's page tables are in virtual memory, they can be sparse, and may even be paged.

Multi-level page tables reduce the amount of physical memory occupied by page tables as well. Multi-level page tables extend the single level page table approach by dividing an address into a tuple $(L_1, L_2, L_3, ..., L_n, O)$. Each L_i, $i = 1, 2, 3, ..., n-1$, is used in turn to index into a corresponding table and return the next level page table until L_n is reached and an actual physical page frame address is found in the highest page table. This is perhaps the most common dynamic address translation scheme. It reduces memory usage since marking a low level page table entry invalid invalidates all higher level page tables and, therefore, all virtual memory addresses that can be reached by indexing through that low level table entry. Figure 6.4 shows a sample two-level page table. Two-level page tables are employed by the Intel iAPX386[Int87] and the National Semiconductor 32000[Nat86] architectures. The Motorola 68000[Mot89] and Motorola 88000[Mot88] architectures employ multilevel page table schemes where the number of levels (1-4) is a parameter of the memory management unit.

Inverted page tables (see Figure 6.5) decrease the amount of memory committed to page tables to a constant that is a function of the virtual memory address space size. They split an address into a (P_v, O) pair, but hash P_v into a table that has one entry per *physical* page in the

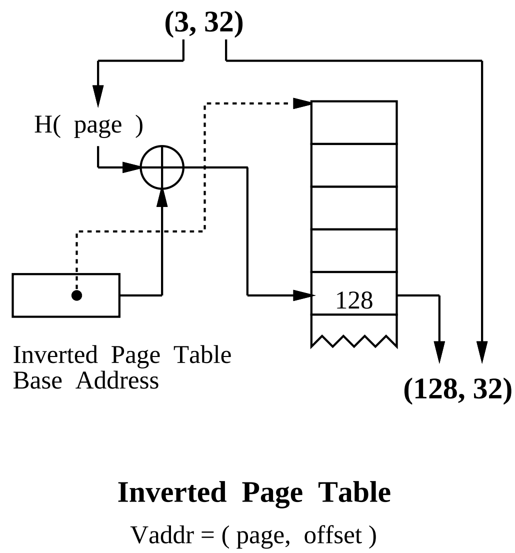

Figure 6.5: An inverted page table

system. The entry that matches P_v determines P_p. Inverted page tables are usually used on segmented memory systems such as the IBM 801[CM87], the RT/PC[IBM86] and the Hewlett Packard Precision Architecture[Hew87] where a single large address space exists and auxiliary mechanisms are used to partition that address space up between applications. One inverted page table is used to map the entire single address space.

The most physical memory efficient dynamic address translation mechanism uses a *translation lookaside buffer* (see Figure 6.6). Translation lookaside buffers decrease the physical memory used for tables to zero. Like page table based schemes they divide an address into a (P_v, O) tuple, but in parallel search a hardware table of virtual to physical page mappings. This table is kept in extra fast memory outside the normal address space (physical memory) of the processor. It is usually small and filled by software via special instructions when a P_v match is not found.

A hardware managed translation lookaside buffer is also used as a cache to speed up dynamic address translation with most page table based schemes [Int87, Nat86, Mot89, Mot88, LL82]. The translation lookaside buffer, in this case, holds a cache of recently accessed virtual addresses and their corresponding physical addresses obtained from the page tables. Each virtual address

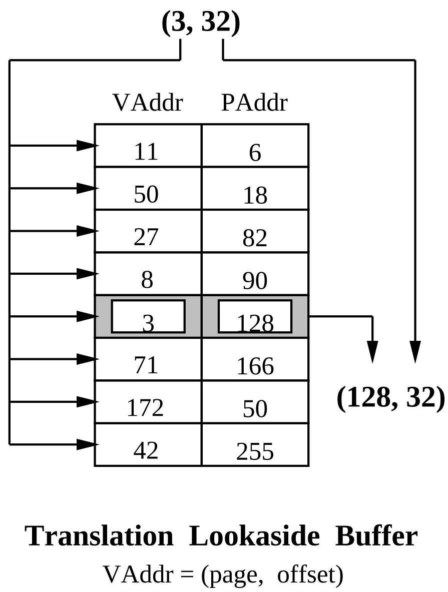

Figure 6.6: A translation lookaside buffer

generated is first searched for in the lookaside buffer. If no match is found, the page tables are searched normally, and the lookaside buffer is updated with the values found in the page table.

6.2.1 Dynamic Address Translation in *Choices*

In order to keep the system maintainable and capitalize on the portability advantages of object-oriented programming described in Section 4.2.1, one of the major design guidelines of the *Choices* memory management system is to encapsulate the hardware dependencies in a few well chosen classes. The aim is to do this without sacrificing efficiency. In *Choices*, the abstract **AddressTranslation** and **AddressTranslator** classes encapsulate the hardware function of mapping virtual addresses generated by running programs into physical addresses. The *Choices* dynamic address translation sub-system is defined by these two abstract classes. These classes present an *architecture-independent interface* to the rest of the memory management system. Each **AddressTranslation** encapsulates the hardware-dependent representation of dynamic address translation for a single virtual address space and defines messages to manage these mappings. An **AddressTranslator** encapsulates the architecture-dependent memory management unit (MMU) that actually performs dynamic address translation. Concrete classes implementing the **AddressTranslation** and **AddressTranslator** signatures implement their methods for specific architectures.

The messages defined by the **AddressTranslation** class include:

- addMapping: to add a virtual-to-physical translation at a given protection level.
- removeMapping: to invalidate a mapping for a virtual address range.
- changeProtection: to change the protection of a given range of virtual addresses.
- activateOn: to indicate that an **AddressTranslation** is being used by a particular **AddressTranslator**
- deactivateOn: to indicate that an **AddressTranslation** is no longer being used by a particular **AddressTranslator**.
- and syncUsageInformation: to copy referenced and modified information about a particular address to the corresponding **PhysicallyAddressableUnit**.

The main **AddressTranslator** messages are:

- determineFaultCode: which is used during translation fault processing to determine what caused the fault.
- and flushTranslation: which flushes any translation cache mappings for a given virtual address. This message is sent primarily by algorithms responsible for keeping shared memory consistent as it is moved between physical memory and backing storage.

The addMapping message of **AddressTranslation** is sent during missing memory fault processing (see Section 6.6.2). When a request for a non-resident memory address occurs, the rest of the *Choices* memory management system uses architecture-independent algorithms and information to retrieve the data, and update the appropriate **AddressTranslation** by sending it the addMapping message. When architecture-independent page or segment replacement algorithms (see Section 6.6.3) are invoked to swap out virtual memory pages or segments from main memory, they send the removeMapping message to invalidate previously active mappings. The implementation of the removeMapping method in turn sends flushTranslation to the current **AddressTranslator** to invalidate any cached dynamic address translation mappings.

The **AddressTranslation** class's changeProtection message is used to support the implementation of special-purpose virtual memory techniques that require the alteration of the physical protection of a region of memory. For example, *copy-on-write* requires the ability to change the access allowed to a region of memory from *read-write* to *read-only* and back again.

The addMapping, and changeProtection messages take architecture-independent protection level arguments that are used by **AddressTranslation** objects to determine the accesses the dynamic translation hardware should allow to a range of virtual addresses. The values of these protection level arguments are elements of enumerated type including: ReadOnly, ReadWrite, NoAccess, ExecuteOnly etc. These values are converted by an **AddressTranslation** into the actual values the specific architecture defines for enforcing the corresponding logical protection on a range of virtual memory. The implementation of **AddressTranslation** may weaken, but never strengthen, a protection level value if the particular architecture cannot enforce the given value. For example, on the NS32332 version of *Choices*, ExecuteOnly is actually enforced as ReadOnly.

The activateOn and deactivateOn messages of **AddressTranslation** are used to support consistency algorithms in cases where an **AddressTranslation** is concurrently shared by multiple processors. These messages are sent to indicate that the **AddressTranslation** is currently being made active or inactive on a particular **AddressTranslator** (and therefore its corresponding processor). When mappings in an **AddressTranslation** are modified, this information is used to determine which **AddressTranslators** are actively referencing the **AddressTranslation**. The flushTranslation message is then sent to these **AddressTranslators** in order to invalidate any hardware cached dynamic address translation mappings.

As discussed in Section 6.2, hardware dynamic address translation tables on most architectures consume physical memory. Following the example of the *pmap* system in Mach[R$^+$87], architecture-dependent representations of the virtual-to-physical memory mappings kept by **AddressTranslations** may be discarded at any time. This allows **AddressTranslations** to act as caches of recently referenced virtual-to-physical mappings for the address spaces they represent. The total set of active mappings kept by all **AddressTranslations** can be stored in a *limited* amount of physical memory shared between them.

Keeping a subset of active mappings in the hardware tables is possible because missing mappings can be reconstructed on demand from architecture-independent information kept by the rest of the *Choices* memory management system. An **AddressTranslation** may discard mappings at any time in order to allow the memory they were occupying to be reused by another **AddressTranslation**. In order for such a "cached mapping" system to be more space efficient than a traditional page table or similar approach, the amount of information kept in the architecture-independent form must be smaller than the amount the traditional approach would consume. Examples of how this can be achieved include: using a logical page size greater than the physical page size; storing only the base page and total length of a contiguously mapped region; and coalescing contiguous unmapped regions into a single (architecture-independent) descriptor. *Choices* **MemoryObjectCaches** (see Section 6.5) can use any of these approaches.

Besides facilitating the portability of *Choices*, the **AddressTranslation** class hierarchy exemplifies the ability of object-oriented techniques to support code reuse across similar hardware versions. The *Choices* virtual memory system was first implemented on the National Semiconductor NS32332 [Nat86] processor. This architecture uses four-kilobyte pages and a two-level page table to implement dynamic address translation. *Choices* represents this by the **NS32332Translation** class. *Choices* has since been retargeted to run on the Intel 80386 [Int87] processor. The 80386 and NS32332 architectures both use four kilobyte pages and a two-level page table. At the level of dynamic address translation, they differ only in the placement of a few bits in their respective page table entries.

The initial retargeting to the 80386 was implemented by a person who was not involved with the initial design of *Choices*. This effort included the **i386Translation** class as the concrete class implementing the **AddressTranslation** signature for the 80386 architecture. Then later, *Choices* was retargeted for the MC68030[Mot89], which also can use two level page tables with four kilobyte pages, and the **MC68030Translation** class was constructed[Helng]. After a close comparison, the **NS32332Translation**, **80386Translation** and **MC68030Translation** implementations were all combined to allow substantial code sharing.[3]

A new **TwoLevelPageTable** class was introduced to define a generic two-level page table parameterized by the page size. The implementation of this class uses the auxiliary **PageTableEntry** class to represent an individual entry. Subclasses of **PageTableEntry** specify the representation of page table entries for the NS32332 (**NS32332PTE**), 80386 (**i386PTE**)

[3]In retrospect, sharing should have been expected since, as discussed in Section 4.2.2 similar architectures permit common algorithms and data structures to be reused.

and the MC68030 (**MC68030PTE**). The superclass **TwoLevelPageTable** localizes common features of the three architectures. The new **NS32332Translation**, **i386Translation**, and **MC68030Translation** classes are subclasses of **TwoLevelPageTable** and reuse most of the original code via inheritance. The differences between the architectures is expressed by using instances of the different page table entry classes. Future support for other architectures with two level page tables should also be simplified by inheriting from the **TwoLevelPageTable** class. All that is needed is to add a new subclass. The page size parameter of the **TwoLevelPageTable** class trivializes retargeting to a system with two level page tables and a page size other than four kilobytes. There is no inherent reason why a **MultiLevelPageTable** class could not be constructed with **TwoLevelPageTable** as a subclass to further increase potential code sharing when targeting architectures that support higher leveled page tables.

6.3 Physical Memory Management

Efficient management of physical memory is essential to a virtual memory system. Physical memory must be shared between virtual address spaces since there is usually far less physical memory than the sum of the sizes of all the virtual address spaces. The *Choices* physical memory management classes allocate and deallocate physical memory to support the rest of the memory management system. These classes include those that maintain the status of physical memory blocks, manage the allocation and deallocation of physical memory, and manage the aggregate blocks of physical memory into lists for simple manipulation. These classes are all architecture-independent. They are parameterized only by the size of a convenient physical memory allocation unit – the page size for most paged systems.

6.3.1 PhysicallyAddressableUnit

In *Choices*, an instance of the **PhysicallyAddressableUnit** class represents a block of *contiguous* physical memory. It is used to localize usage information about the block of memory represented. The address message returns the physical address of the block. Each **PhysicallyAddressableUnit** maintains information to indicate the **MemoryObjectCache** for which it is holding data, and which unit of that **MemoryObjectCache's** data it holds. The currentCache and unitNumber message return these values. The size message is sent to a **Phys-**

icallyAddressableUnit to determine the length of the block in bytes. The addIOReference, removeIOReference and currentIOReferences messages are used to maintain and access information about whether the **PhysicallyAddressableUnit** is currently in use by the I/O system. The referenced, setReferenced, modified, and setModified messages access and update various attributes of the unit when it is actively holding data. This information is updated during memory placement and replacement. In particular, when removing a mapping from an **AddressTranslation**, the reference and modification information for the corresponding block of physical memory is transferred to the corresponding **PhysicallyAddressableUnits** by sending setReferenced and/or setModified.

A block of memory may be simultaneously accessed by several processors. Therefore, each **PhysicallyAddressableUnit** maintains a set of tuples consisting of a reference to an **AddressTranslation** currently mapping virtual addresses to the physical memory the **PhysicallyAddressableUnit** manages, and the starting virtual address of this mapping. This information allows the actual hardware translation tables to be consulted in order to guarantee that the referenced/modified status of a **PhysicallyAddressableUnit** is up to date. Sending the **syncUsageInformation** message to all the **AddressTranslations** in the set causes this to occur.

In a system heavily utilizing shared memory, the number of address spaces actively referencing a particular physical memory range may be quite high and, therefore, the number of **AddressTranslations** referencing a given **PhysicallyAddressableUnit** may be high as well. Keeping track of all the **AddressTranslations** referencing each **PhysicallyAddressableUnit** may seem prohibitively expensive. This problem is solved in *Choices* because an **AddressTranslation** is only a cache of active mappings, not a complete set of virtual to physical mappings. Each **PhysicallyAddressableUnit** maintains only a fixed number of references to **AddressTranslations**. If it is necessary to add another reference (usually the result of a missing memory fault being processed), and all references are currently in use, the least recently added reference is replaced. Before it is replaced, however, the **AddressTranslation** currently referenced is made to invalidate its mapping to the memory the **PhysicallyAddressableUnit** represents. This is accomplished by sending it the removeMapping message with the associated virtual address as an argument. The upper bound on the number of references a **PhysicallyAddressableUnit** needs to maintain is the number of physical processors in the system. This is because, in the worst case, a different address space may be active on each processor and each address space may be actively referencing a given physical memory unit. In practice, however, this is probably much larger than actually needed. Since the current amount of sharing in *Choices* is actually quite low, using three references has been hueristically chosen to be acceptable.

6.3.2 Store

An instance of the **Store** class represents a collection of all of the allocatable **PhysicallyAddressableUnits** in the system. The allocate and free messages are sent to a **Store** to manage the assignment of physical memory to operating system subsystems including the virtual memory management and I/O subsystems. The allocate message is sent to request a number of *bytes* of physical memory from the free physical memory pool. The request is satisfied by assigning the minimal number of **PhysicallyAddressableUnits** needed to hold that number of bytes. In the event that the allocate method cannot satisfy the request, it blocks. When the memory replacement algorithm frees sufficient **PhysicallyAddressableUnits** by swapping out the contents of infrequently accessed virtual memory locations to their backing stores, it unblocks and fulfills unsatisfied requests (see Section 6.6.3).

6.3.3 PhysicalMemoryChain

Since a virtual address range may be mapped to multiple, perhaps discontiguous, physical address ranges, individual **PhysicallyAddressableUnits** are not always convenient data structures with which to manipulate arbitrary virtual address ranges. The **PhysicalMemoryChain** class defines an arbitrarily connected collection of **PhysicallyAddressableUnits**. **PhysicalMemoryChains** consist of a linked list of **PhysicallyAddressableUnits**, an aggregate size, and a byte offset into the first unit (see Figure 6.7). These three entities are sufficient to describe an arbitrary virtual address region by its physical memory addresses. **PhysicalMemoryChains** are the primary form in which memory regions are passed between objects in the *Choices* virtual memory system. They describe memory regions for I/O and physical addresses for **AddressTranslations**.

A **PhysicalMemoryChain** construction function builds the list of **PhysicallyAddressableUnits** from arguments that specify a starting virtual address, a length, and the virtual

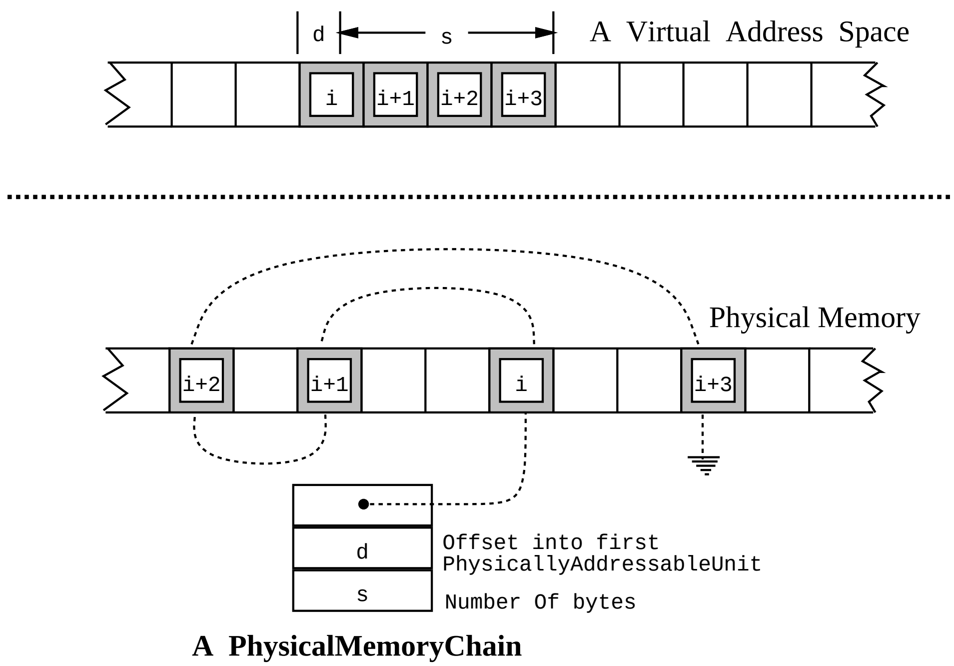

Figure 6.7: Graphic representation of a **PhysicalMemoryChain**

memory in which the address range is valid. As it builds the list, the function invokes other virtual memory system methods to make sure that the virtual memory specified in the arguments is resident in physical memory (see Section 6.6.2). Also, reference counts kept in the component **PhysicallyAddressableUnits** are incremented (by sending addIOReference). Destroying a **PhysicalMemoryChain** likewise decrements the reference counts of the component **PhysicallyAddressableUnits**. The memory replacement algorithm will not move data held in the memory managed by a **PhysicallyAddressableUnit** to its backing store in an attempt to reclaim physical memory if the reference count of the **PhysicallyAddressableUnit** is non-zero.

Reliable I/O is assured since the physical memory required for the I/O must first be included in a **PhysicalMemoryChain** and then passed to the I/O subsystem. The physical memory will be unavailable to the replacement algorithm until the I/O completes and the **PhysicalMemoryChain** is deleted. This is a good example of the utility of the object-oriented technique of encapsulation within objects when applied to system programming. The **PhysicallyAddressableUnit** reference counts do not need explicitly updated by the I/O system. Creating and destroying **PhysicalMemoryChains** automatically manages these reference counts and therefore automatically locks the corresponding logical memory resident in physical memory.

6.4 Backing Storage Management

The **MemoryObject** class defines an abstract signature for accessing the backing storage of a collection of logically related data. In *Choices* a **MemoryObject** is viewed as a sequence of identically sized indexed storage units. A unit size of one byte models a byte stream, while a unit size of 512, 1024, 2048, or 4096 bytes can be used to represent blocks on a disk device.[4] Instances of concrete classes implementing the **MemoryObject** signature represent the backing storage for data corresponding to program instruction segments, stacks, disks, heaps, data spaces, and files. They appear throughout the operating system and encapsulate physical disk drivers, disk partitions, files, and even the instruction and data sections of an executable program file. In this way they span the storage hierarchy representing everything from low level disk devices to high level files. Current **MemoryObject** subclasses have been implemented to represent physical disks, disk partitions, sub-ranges of other **MemoryObjects**, Berkeley UNIX inodes, System V UNIX inodes, MS-DOS files and other new and experimental file system structures [MCRL88, MLRC88, Mad91].

The **MemoryObject** numberOfUnits and unitSize messages return the number and the size of the units respectively. Subclasses of **MemoryObject** may choose to make the number of units fixed, or may allow it to grow. The **setNumberOfUnits** message exists to allow subclasses to let the number of units to be decreased or increased. Sending **setNumberOfUnits** with a value less than the current number of units will truncate the data the **MemoryObject** represents and decommit any storage used by the freed units. Sending setNumberOfUnits with a value greater than the current number of units will grow the **MemoryObject** to that size. Reading the data in the growth area will initially return a data full of zeros. Concrete classes that implement fixed size **MemoryObjects** can choose to ignore sends of setNumberOfUnits and return an error. The offsetToUnit and unitToOffset messages convert byte offsets into unit indices and vice versa. The message read and write take as arguments, an index, the number

[4] Although not necessary, but for the sake of implementation simplicity and efficiency, the size of a unit was chosen to always be an integer power of two.

of units to be read or written, and a **PhysicalMemoryChain**. They cause the corresponding units within the **MemoryObject** to be accessed. The **PhysicalMemoryChain** provides the locations of physical memory blocks that are to be used in the read or write operation. Reading and writing to regions described with physical addresses provides a virtual memory mapping-independent mechanism for I/O. This is important since most I/O devices perform I/O using physical memory addresses, not virtual memory addresses. Concrete **MemoryObject** classes are free to make writing a unit greater than the current number of units an error, or to grow the **MemoryObject** out to that size.

Finally, the **MemoryObject** class accepts the makeResidentInPhysicalMemory message to guarantee that a set of units within it are buffered in physical memory. The implementation of this message will be discussed in Section 6.6.2.

6.4.1 MemoryObjectViews

A **MemoryObject** represents a logical memory backing store. Actual physical backing storage usually takes the form of disks or drums. Partitioning multiple address spaces into multiple logical regions each managed by a **MemoryObject** will result in many more logical backing stores than physical disks (also represent by **MemoryObjects**) on an average computer. This situation is handled in *Choices* by introducing the **MemoryObjectView** subclass of **MemoryObject**. A **MemoryObjectView**, as its name implies, represents a subrange of another **MemoryObject**. With **MemoryObjectViews**, large **MemoryObjects** can be partitioned up into smaller **MemoryObjects** used to store logical data. Subclasses of **MemoryObjectView** manage mapping to contiguous or discontiguous ranges of underlying **MemoryObjects**. For example, a disk in *Choices* is represented by a **MemoryObject** that is similar to a traditional device driver in that it can read and write the physical disk. Such a **MemoryObject** is then broken up into partitions by creating instances of **MemoryObjectPartition**. **MemoryObjectPartition** is a subclass of **MemoryObjectView** that manages a contiguous region of an underlying **MemoryObject**. Each of these partitions are then further broken up into files by using instances of subclasses of **MemoryObjectView** which handle discontiguous regions of the underlying **MemoryObject**. Examples include **BSDInode**, which recognizes the format BSD UNIX files take, or **MSDOSFile**, which recognizes the format of MSDOS files.

6.4.2 Alternate Implementations of read and write

Many alternatives exist for the type of the source and destination arguments for the **MemoryObject** read and write message.[5] One possibility is to use a starting *virtual* address and a length. However, since sending read and write to most **MemoryObjects** eventually results in traversing layers of **MemoryObjectViews** until a **MemoryObject** representing a physical disk is arrived at, and most physical disks expect to perform I/O to physical memory locations, this requires a virtual to physical memory translation of the argument to be performed by the read or write methods of the low level physical disk driver **MemoryObject**. This would increase the dependencies in the memory management system since **MemoryObjects**, which should only deal with backing storage, would need to reference objects managing virtual memory. Such dependencies are reminiscent of the cyclical dependencies in layered systems. This scheme also restricts the implementations and uses of **MemoryObjects** as well. An instance would need to have available a description of the virtual address space in which the argument data is coming from or going to. For example, consider that it is convenient to place the lowest level **MemoryObjects** that drive devices like the disk in system (protected) memory and execute their methods in a privileged mode. If address arguments were virtual address ranges, the abstract **MemoryObject** protocol would require read and write to have an argument that specifies the virtual memory within which to perform the translation. Additionally it would be necessary for the virtual memory system to support the access of arbitrary addresses in arbitrary virtual address spaces.

An alternate implementation that solves the previous solution's problems is to use a starting *physical* address and a length rather than a virtual address. This suffers from the problem that even though a range of virtual memory may be stored contiguously on backing storage to promote locality and efficiency of accesses, it may not be contiguous in physical memory. Passing a single physical address and a length cannot take advantage of the potential contiguity on backing storage since a virtual address range would have to be decomposed into its component physical ranges and each of those read/written individually.

PhysicalMemoryChains solve both these problems since, although a **PhysicalMemoryChain** represents a contiguous logical entity in virtual memory, it may include disjoint

[5] In fact each of the following possibilities was tried first before the development of the design presented here.

blocks of physical memory. Therefore, a virtual address range's logical contiguity can be taken advantage of allowing it to be read from, or written to, its backing store in a single request. Also the use of scatter/gather direct memory access hardware techniques like those supported by IBM/370 architecture device channels[IBM88b] (which were designed specifically for such applications) is facilitated by this locality and the information in **PhysicalMemoryChains**.

6.5 Buffering MemoryObjects in Physical Memory

Applications usually access the logical data of a **MemoryObject** by mapping it directly to an address range within a virtual address space. The data of such a **MemoryObject** can then be accessed by the processor's instructions. Examples of such **MemoryObjects** include those that represent the instructions or data of an executing application program. **MemoryObject** data can be accessed either through the virtual memory system in this way, or it can be accessed indirectly off backing storage by copying portions of it into application buffer memory. This is accomplished by sending read and write to the **MemoryObject** directly. Such indirect access of a **MemoryObject's** data usually suffers from access latency and limited device throughput. Memory must also be (redundantly) committed to buffer the data before processing it.

The class **MemoryObjectCache** provides an abstract signature for mapping a **MemoryObject's** data into physical memory using the functions provided by the physical memory management classes (see Figure 6.8). Concrete classes implementing the **MemoryObjectCache** signature provide specific schemes or dynamic *caching* techniques to map all, part, or none of the data. Each **MemoryObject** bound to a virtual address range in *any* number of virtual address spaces has a *unique* **MemoryObjectCache** responsible for caching its logical data in physical memory.

Mapping a **MemoryObject** to a virtual address range does not not change the functionality of **MemoryObjects**, instead, it simplifies access to their data. Rather than first having to be explicitly copied into buffers, mapping a **MemoryObject's** data to a virtual address range allows data to be transparently accessed by the instructions of the computer. The virtual memory system takes care of moving data to and from backing storage. Also, since a **MemoryObject** can be bound to different address ranges within different virtual address spaces, but is cached in physical memory only once (by its **MemoryObjectCache**), its data is never

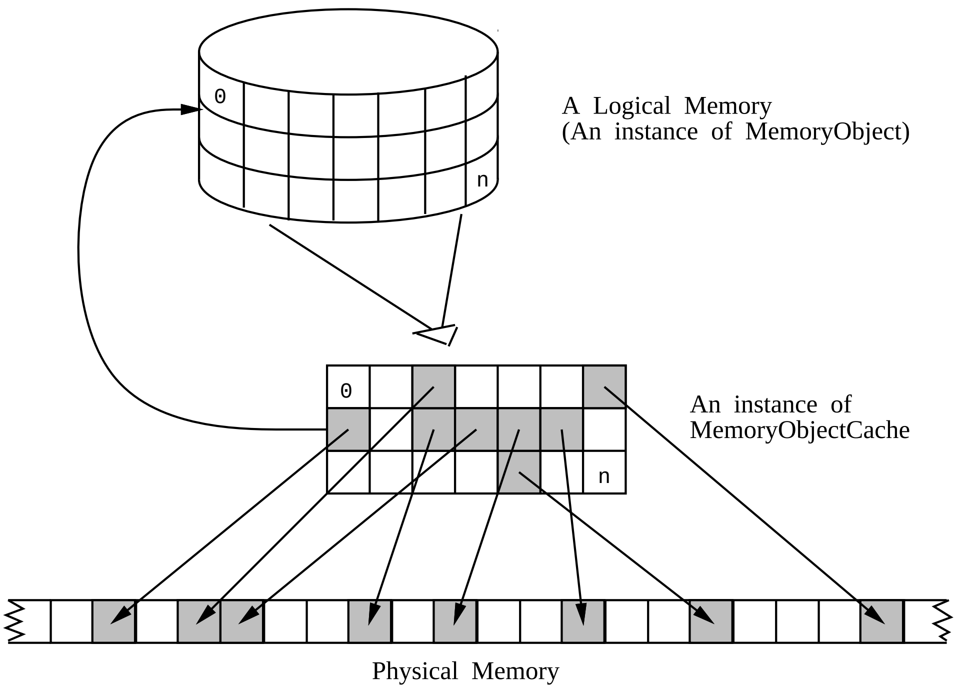

Figure 6.8: Graphic representation of a **MemoryObjectCache**

redundantly buffered. This avoids memory waste as well as the cost of keeping multiple buffers coherent.

Sending the cache message to a **MemoryObjectCache** causes a region of the data of the **MemoryObjectCache's** associated **MemoryObject** to be made resident in physical memory. The cache message has a starting unit and number of units as arguments and returns a **PhysicalMemoryChain** describing the physical memory caching this region. If the region is not resident, the cache method first sends the allocate message to a **Store** to obtain a **PhysicalMemoryChain** describing a region of sufficient length. Next, it sends the read message to the corresponding **MemoryObject** with this **PhysicalMemoryChain** as an argument (see Section 6.4).

Sending the selectUnitForRemoval message to a **MemoryObjectCache** causes it to select one of the currently resident data units for return to backing storage. Concrete classes implementing the **MemoryObjectCache** signature use this to affect the policy they implement for caching logical data in physical memory. Sending the release message to a **MemoryObject-**

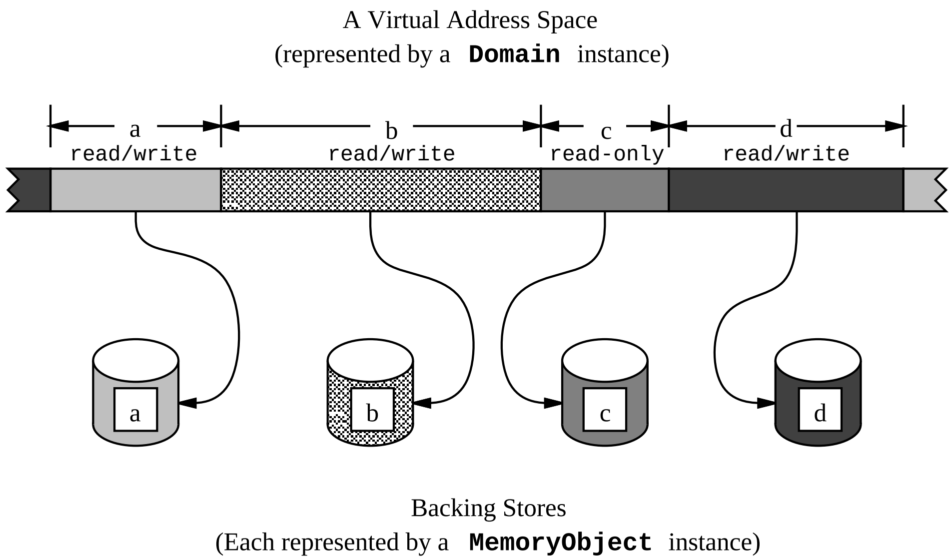

Figure 6.9: Abstract relationship between **Domains** and **MemoryObjects**

Cache along with a unit number as an argument causes the corresponding unit to be actually be returned to its backing storage.

6.6 Virtual Memory

The previous sections describe the manipulation of the dynamic address translation hardware, the management of backing storage, and the caching of the data encapsulated by a **Memory-Object** into physical memory. This section introduces the **Domain** class, which ties together the previous classes and provides the primary virtual memory abstraction. This section also details memory fault processing and memory replacement processing.

6.6.1 Domains

Instances of **Domain** are the top level abstraction of the virtual memory system. A **Domain** maintains a collection of **MemoryObjects** and associated access rights together with a map of these into a complete virtual address space (see Figure 6.9). Each **MemoryObject** maps a range of addresses in the address space. Therefore, using the mechanisms discussed so far, **Domains** provide for multiple virtual address spaces mapped to multiple logical memories, each represented by a **MemoryObject**. Each **Domain** also has an associated **AddressTransla-**

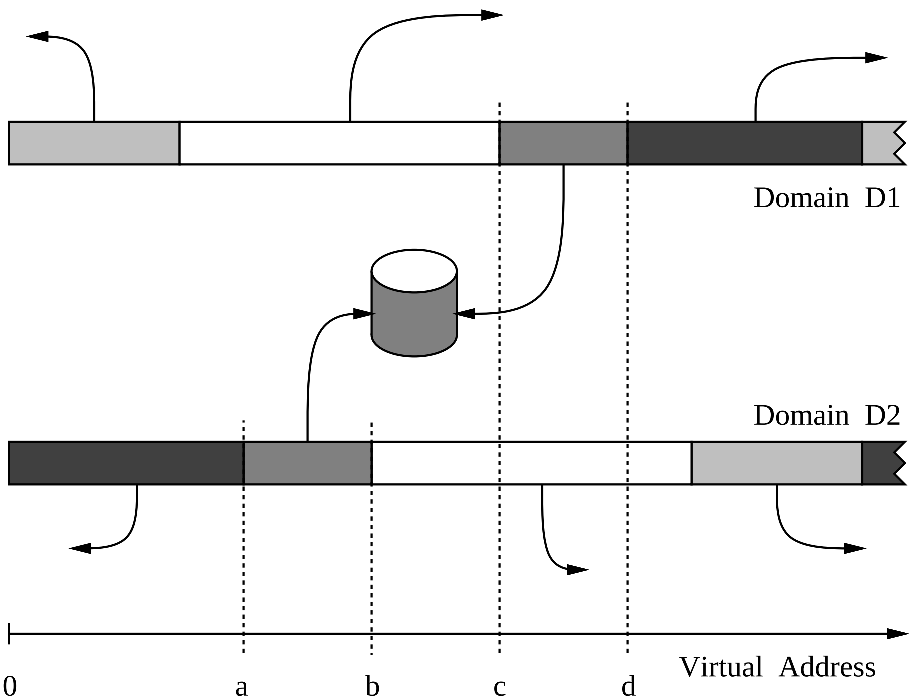

Figure 6.10: A shared **MemoryObject**

tion that it uses to affect its virtual memory mappings. Consistency of a **MemoryObject** shared by multiple **Domains** is ensured since each **MemoryObject** has a single **MemoryObjectCache** (i.e., the virtual memory of all the sharing **Domains** will be mapped to the same physical addresses). **MemoryObjects** can also map to different regions of different (or the same) **Domain**.

The three most important operations in the **Domain** signature are add, remove and fixFault. Adding a **MemoryObject** to a **Domain** binds a virtual memory range to the data managed by the **MemoryObject**. The add message can either bind a **MemoryObject** to a specified virtual memory subrange or allocate a range of memory sufficient to map it. Removing a **MemoryObject** from a **Domain** causes any mappings to that **MemoryObject's** data to be invalidated (see Section 6.6.4). The fixFault message is sent to process a missing memory fault as described in Section 6.6.2.

Sharing a **MemoryObject** between different **Domains** (see Figure 6.10) is simple in *Choices*. Because a **MemoryObjectCache** is independent of any virtual memory address, a **MemoryObject's** data can be mapped into different ranges of virtual addresses in different address spaces.[6] Since **Domains** assign access rights, the data can also have different access rights in different address spaces.

In *Choices* the operating system objects are partitioned among a set of **MemoryObjects** that are in every **Domain** but mapped as only accessible while executing in the privileged mode of the processor. Examples of such **MemoryObjects** are those containing the objects necessary to process address translation errors (**Domains** and **AddressTranslations**), the objects used to fetch missing data from permanent storage (**MemoryObjects** corresponding to physical devices), and the objects necessary for context switching (see Chapter 7). The data of these **MemoryObjects** is locked resident in physical memory by creating **PhysicalMemoryChains** describing all of the memory, and never deleting these **PhysicalMemoryChains**.

A **MemoryObject's** data can be shared by different **Domains** residing on different nodes of a distributed system. Each node sharing the logical memory has a local **MemoryObject** and **MemoryObjectCache** for the data and uses a cache coherence protocol to keep them consistent. Such a system is implemented in [Joh91].

6.6.2 Missing Memory Fault Processing

Data must be moved into physical memory before it can be accessed by the processor. When a program accesses a virtual address that generates a missing virtual memory error, the handler for that error first sends the determineFaultCode message to the current **AddressTranslator** to determine the condition that caused the fault. The fixFault message is then sent to the current **Domain** using the faulting virtual address and the fault condition as arguments. The implementation of the **Domain** fixFault method then attempts to correct the faulting condition (usually by bringing missing virtual memory into physical memory), or raises another exception (see Section 7.7) if the fault cannot be repaired.

In order to keep the **Domain's** mappings of virtual address data consistent in the presence of simultaneous faults on a multiprocessor, fixFault currently locks the **Domain** to permit only one fault per **Domain** to proceed at a time. This restriction is actually a result of the current implementation of **Domain** and can be corrected by keeping multiple locks, one per range of addresses, per **Domain**.

[6] Obviously this only makes sense if the **MemoryObject** contains position independent data. Files and disks are good examples.

The fixFault method next converts the faulting address (passed as an argument) into a **MemoryObject** reference and an offset within the data of that **MemoryObject**. The fixFault method ensures that the corresponding logical memory block is cached in physical memory by sending the makeResidentInPhysicalMemory message to the resultant **MemoryObject**. The makeResidentInPhysicalMemory message takes a range of MemoryObject units as an argument, uses the corresponding **MemoryObjectCache** to obtain the corresponding **PhysicallyAddressableUnits** by sending the cache message to the **MemoryObjectCache**.

The cache method first locks the **MemoryObjectCache** to prevent inconsistencies introduced by multiple invocations of **cache** on multiple processors. Again, the amount of locking here is implementation dependent. For example, a single lock could protect all the units of a **MemoryObjectCache**, or the units could be divided up into groups with one lock per group. Next, the cache method determines if the data required are already in a set of PhysicallyAddressableUnits. If not, new **PhysicallyAddressableUnits** are allocated from the **Store** and the data are transferred into them by reading the corresponding **MemoryObject**.

Once the proper **PhysicallyAddressableUnits** have been determined, their references to active **AddressTranslations**, along with the inverse mappings, need updated. This is accomplished by locking the **PhysicallyAddressableUnit**, setting an **AddressTranslation**/virtual address tuple (an existing one is replaced if necessary as described in Section 6.3.1), sending addMapping to the **AddressTranslation** to set the mapping in the other direction (and to actually reflect the mapping in hardware), and unlocking the **PhysicallyAddressableUnit**. The information indicating for which **MemoryObjectCache** the **PhysicallyAddressableUnit** it is holding data is also updated. Once all this has been done for all **PhysicallyAddressableUnits** containing the data, the lock on the **MemoryObjectCache** data is released and the **cache** method returns. Finally, the **Domain** lock is released and the fixFault method returns. At this point the faulting instruction can be restarted or resumed.

6.6.3 Memory Replacement

Periodically, or when a **Store** runs of of memory, a "page-out" algorithm runs to maintain adequate free memory for future **Store** allocations. *Choices* can support both global (**PhysicallyAddressableUnit** based) and more localized (**MemoryObjectCache** based) virtual memory replacement algorithms.

A global replacement algorithm incrementally scans all the **PhysicallyAddressableUnits** in the system and extracts usage information on which to base its page replacement decisions. Policies such as global *working set*[Den68] or global *least recently used*[BS88] can be implemented with such a scheme.

A **MemoryObjectCache** specific, replacement scheme scans all active **MemoryObjectCaches** rather than **PhysicallyAddressableUnits**. This allows the **MemoryObjectCaches** to impose localized memory replacement since **MemoryObjectCache** subclasses can specialize the selectUnitForRemoval message to apply specific replacement policys to the physical memory allocated to the **MemoryObjectCache**

When using either algorithm, the memory replacement mechanism reaches a point where it has the unit number within a **MemoryObjectCache** that needs to be removed from memory. In the global case this information is found by sending the currentCache message and unitNumber messages to the **PhysicallyAddressableUnits** chosen to be reused. In the local algorithm, this information is known automatically, since the **MemoryObjectCache** itself is choosing to do the replacement of its own units.

In either case, memory replacement proceeds by sending the release message to the **MemoryObjectCaches** to free physical memory. The **MemoryObjectCache** first locks its internal data structures for consistency in the presence of possible concurrent invocations of cache for memory fault processing (see Section 6.6.2). Next, all **AddressTranslations** actively referencing any **PhysicallyAddressableUnits** need to have such mappings invalidated. This proceeds by first locking the **PhysicallyAddressableUnits**, following each **AddressTranslation** pointer and sending removeMapping to the corresponding **AddressTranslation**. The removeMapping method will transfer any referenced or modified information kept by the hardware mappings back to the **PhysicallyAddressableUnit**.

Once this is complete, the **PhysicallyAddressableUnit** can be unlocked. After this has been done for all **PhysicallyAddressableUnits** holding the data being removed, the release method can inspect each **PhysicallyAddressableUnit** and return any modified data contained in them back to its backing storage.

Selected units that have not been modified can just be returned to the **Store** and marked non-resident in the **MemoryObjectCache's** tables. Whether the unit has been written or not can be determined by sending the modified message to the corresponding **PhysicallyAd-**

dressableUnits. The modified method indicates whether a physical memory block has been modified since it was last read from the **MemoryObjectCache's** corresponding **MemoryObject**. This information is guaranteed to be up to date since there will no longer be any **AddressTranslations** mapping to the **PhysicallyAddressableUnits** and no new ones can be added since the **MemoryObjectCache** lock on the logical units is still held.

Modified units must be returned to their backing storage. To achieve this, the release method builds **PhysicalMemoryChains** to represent all of the **PhysicallyAddressableUnits** containing modified data. The release method then writes all of the **MemoryObjectCache's** units that have been modified to its **MemoryObject**, ensuring the consistency of the data on backing storage.

6.6.4 Removing a MemoryObject from a Domain

Sending the remove message to a **Domain** deletes the mapping between a virtual memory range and a logical memory. First, the remove method locks the **Domain** and removes all references to the corresponding **MemoryObject** in the **Domain**. This keeps new missing memory faults from attempting to bring new data in from the **MemoryObject**. The **Domain** is then unlocked. Next, the **Domain** sends removeMapping to its corresponding **AddressTranslation** to invalidate all hardware physical address translation mappings for the corresponding virtual address range. This also updates reference and modification information in those **PhysicallyAddressableUnits** that correspond to the physical memory blocks storing resident data (see Section 6.3.1). Finally, if the **MemoryObjectCache** corresponding to the **MemoryObject** is not referenced by any other **Domain**, then it is deleted. Deleting the **MemoryObjectCache** will force any modified resident data to the backing store and return any physical memory to the **Store** it was allocated from by sending the **Store** the deallocate message.

6.7 Performance

The performance of a memory management system is limited by the speed of the backing storage devices. On the Encore Multimax version of *Choices*, a transfer of one page (4K) of data to or from a physical disk takes on the average 23.6 ms.[7] Disk seek and rotational latency can as much as double this time. Under *Choices* on the Multimax, processing a missing memory fault takes as a minimum 25.1 ms from the time the process traps into the operating system until it is resumed. This indicates that the minimum overhead the *Choices* memory management system imposes for processing missing memory faults is approximately 1.5 ms. Sending the determineFaultCode to the current **AddressTranslator** takes approximately 215 μs. Looking up which **MemoryObject** is managing a particular virtual address takes approximately 250 μs. Sending the addMapping message to an **AddressTranslation** takes approximately 300 μs. The remaining time is spent in the makeResidentInPhysicalMemory method of **MemoryObject**.

A more useful measurement of the *Choices* memory management system is how well it compares to other systems. Without the source to the UNIX operating system, it is impossible to get numbers as detailed as those above. However, it is possible to run a program with poor paging performance on both UNIX and *Choices* and compare the results. The test chosen was to zero fill four megabytes of memory in a tight loop. Running under the UNIX operating system on the Encore Multimax, this test takes 11.6 seconds to execute. Under *Choices* the same program takes 12.1 seconds to execute. With 4 kilobyte pages, zero filling 4 megabytes of memory causes 1024 page faults. The difference between the UNIX results and the *Choices* results is 0.5 seconds. Dividing this additional overhead by the number of page faults indicates that *Choices* imposes approximately 488 μs over UNIX for processing a page fault. As currently implemented, most of the *Choices* memory management algorithms are rather simplistic and inefficient. It should not be hard to improve them.

6.8 Summary

The *Choices* memory management system provides a portable virtual memory system that is compatible with a wide range of architectures. It provides both a flexible and extensible model of virtual memory.

Reducing the architecture dependencies to just two classes facilitates portability across a wide range of architectures. Likewise, as discussed in Section 6.2.1, concrete classes implementing the **AddressTranslation** and **AddressTranslator** signatures for similar architectures can

[7] The data in this paragraph were collected by Aamod Sane and Taed Nelson.

share a great deal of code through inheritance. The address translation classes are not the only classes that benefit from code sharing. **MemoryObject** classes for similar backing storage representations of data can share significant code[Mad91].

The **MemoryObject** class signature is an interface for storing data to and retrieving data from backing storage. Numerous policies for the management of backing storage can be implemented by concrete classes implementing the **MemoryObject** signature. Likewise, the **MemoryObjectCache** signature defines an interface for caching portions of a **MemoryObject's** data in physical memory. Different policies for the management of this caching can be implemented by concrete **MemoryObjectCache** classes.

The encapsulation provide by object-oriented programming also allows the simplification of many memory management issues. In particular, **PhysicalMemoryChains** simplify the temporary locking of virtual memory addresses to physical memory addresses while I/O is being performed. Creating and destroying **PhysicalMemoryChains** automatically manages reference counting physical memory ranges. The memory replacement algorithms have the proper information to detect that a range of physical addresses is being used in an I/O request without the explicit coding of locking primitives. Creating a **PhysicalMemoryChain**, which is necessary to do the I/O operation in the first place, manages the locking.

In summary, the design and implementation of the *Choices* memory management system shows that many of the claims made in Chapter 4 for the construction of object-oriented operating systems are easily realizable.

Chapter 7

Process Management

7.1 Overview

The concept of a *process*[1] is fundamental to all modern kernel-based operating systems. A process represents a program in execution[Dei84a]. An *individual control path* through a program in execution is perhaps a more precise definition since a program may have multiple concurrent execution paths.

In traditional systems, an individual process follows a control path between a program's various functions and procedures. As discussed in Chapter 3, an object-oriented system is characterized by messages being sent to objects in order to perform computation. These message sends cause object methods to be invoked. Therefore, with respect to object-oriented systems, a process follows a control path between object methods. In both object-oriented and traditional systems, each process has a *current execution point* (the address of the instruction it is currently executing). The current execution point of a process in a traditional systems is always within a particular procedure or function. In an object-oriented system the current execution point is always within a particular method. Message sends cause the current execution point to move from method to method of various objects (see Figure 7.1).

Details of process management for an operating system are very low-level and architecture specific. The *Choices* process management system is an excellent demonstration of how object-oriented programming can support such low-level details without sacrificing performance. The

[1]Some systems use the term *task* or *agent*.

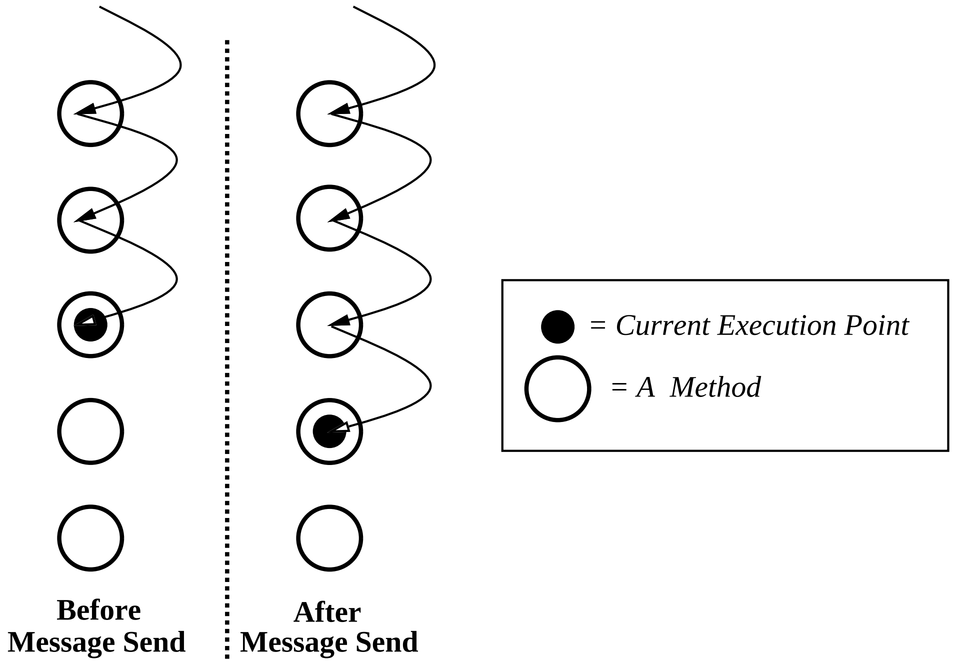

Figure 7.1: Flow of control from method to method

goal of the *Choices* process management system is to support an efficient implementation of a model of an application composed of a potentially large number of parallel processes that can share portions (or all) of their address spaces. In this way, *Choices* processes are similar to the *processes* of the V System[Che88], *threads* in Mach[T$^+$87] and *lightweight processes* in the extensions to UNIX described in [MS87, Seq85a, Enc86].

As with most operating systems, *Choices* characterizes a process by an address space describing the memory it can access and the state of the processor that is executing it. This state is usually referred to as the process's *context*. Every computer has one or more processors. In *Choices*, like any multiprogrammed system, processors need to be shared between processes. While a process is active on a processor, its context is reflected in that processor. When the process is not active on a processor, its context must be saved. A process is said to be *executing* when it is active on a processor. In order to transfer the processor between various processes, an operating system must implement primitives to allow it to switch the physical processor between the contexts of different processes. This is usually termed *context switching*. *Choices* implements its process model using the object-oriented paradigm. All processes are represented as objects (instances of the **Process** class) and their execution is manipulated by sending messages to those objects. In particular, context switching is implemented by sending messages to **Processes**.

In *Choices*, a context switch occurs when one process relinquishes the processor to another. This usually happens as the result of a process's control flow reaching a method of a synchronization or scheduling object. Examples of such objects include those implementing semaphores, monitors, or time-sharing schedulers. Figure 7.2 shows an example of context switching in *Choices* graphically. The small solid circle represents the execution point of the physical processor. Solid arrows represent message sends causing the flow of control to move from method to method. Dashed arrows represent a control flow change that is caused by some form of process exception (a trap or interrupt). The shaded circles represent methods of synchronization objects that perform process scheduling or synchronization functions. Invocations of such methods eventually return like any other method invocation, except that there is no guarantee that the processor has not been assigned to another process and back in the mean time. Likewise, there is no guarantee that the process resumes execution on the same processor that it was executing on when it first sent the message to the synchronization or scheduling object.

In *Choices*, multiple processes can share data by sharing portions of their address spaces. An application in *Choices* consists of an address space and one or more processes to follow the flows of control through the application's program. A single application's processes all execute in its (shared) address space. Multiple applications can cooperate and share data by sharing portions of their address spaces. This allows processes within and across applications to exchange data by placing values in the shared memory for other processes to retrieve.[2] Programmers can use multiple processes to program concurrency and parallelism in their programs. If context switching is expensive, the cost of using multiple cooperating processes may be prohibitive. *Choices* uses object-oriented techniques to optimize this expense.

The normal flow of control of a process can be altered by one of two events. First, an *interrupt*[PS85] may occur. Interrupts usually occur as the result of devices requiring attention from the operating system[BS88]. This may be the completion of an earlier I/O request, or an

[2] Such accesses may require mutual exclusion. This must be guaranteed by higher level synchronization primitives.

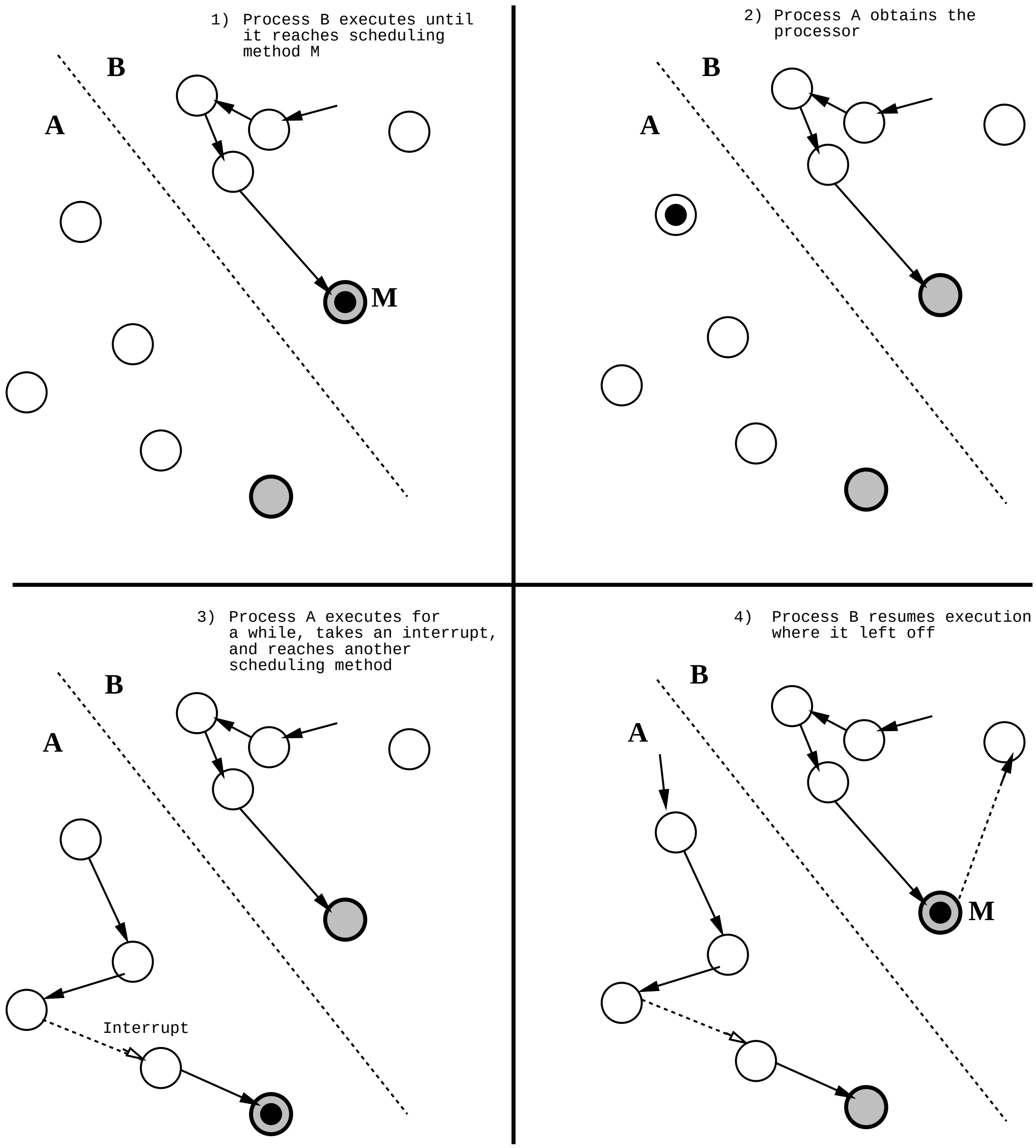

Figure 7.2: Overview of *Choices* process management

unexpected event such as a power failure or device failure. Second, the process may execute an instruction that, for various reasons, may not be able to be completed. Examples include:

- attempting to divide-by-zero.
- executing a floating point operation where the result overflows the floating point representation.
- executing an illegal or privileged instruction.
- executing an instruction that causes dynamic address translation to fail.
- executing an SVC or trap instruction.

Such conditions will be referred to as program *traps*. In *Choices*, both traps and interrupts are generically termed *exception conditions* or *exceptions*. When an exception occurs, a message is sent to an *exception handling object* to correct the condition and then resume the process. If the exception is fatal and the process cannot be resumed, it is terminated. A useful exception management architecture for an operating system must be efficient enough to allow rapid correction or processing of an exception condition or, if that is not possible, graceful termination of the process responsible for generating the exception.

7.2 The *Choices* Process Management Classes

The *Choices* process management and scheduling system is divided into four major abstract classes.

1. The **Process** class represents a process and its context.
2. The **Processor** class represents a physical processor.
3. The **ProcessContainer** class represents a repository of **Processes**.
4. The **Exception** class provide an encapsulation of trap and interrupt handlers.

In *Choices*, a unique **Process** object exists to represent each process.[3] The **Process** class defines the giveProcessorTo message to implement context switching. This message is sent

[3]The distinction between the term process with a capital P (**Process**) and a without (process) is that the former will be used when referring to an instance of a class with the **Process** signature while the latter will be used to refer to the logical entity that it represents.

from within synchronization and scheduling objects to the current **Process** in order to effect processor sharing.[4] The giveProcessorTo message takes a single argument: the next **Process** to run.

Primitives for scheduling and blocking processes in *Choices* are built using instances of classes in the **ProcessContainer** hierarchy. A **ProcessContainer**, as the name implies, is a repository of **Processes**. Scheduling decisions involve transferring **Processes** between **ProcessContainers** and switching the processor to the contexts of processes that are removed from **ProcessContainers**.

The **Exception** class encapsulate the handlers for interrupts and traps. When a trap or interrupt occurs, low level architecture specific code is invoked by the hardware interrupt mechanism to save some of the currently executing process's context and send the handle message to an **Exception** object corresponding to the trap or interrupt.

Eventually, either involuntarily as the result of an exception or voluntarily, a process invokes giveProcessorTo to affect a context switch to another process. For example, consider implementing a semaphore[Dij68]. When a semaphore P operation is performed, if the semaphore is busy, the currently executing process needs to be blocked and another process run. In *Choices*, this occurs by arranging to have the current process placed in a queue of processes blocked on the semaphore, choosing another process to run from the queue of ready to run processes, and sending the giveProcessorTo message to the current **Process** with the new **Process** to run as an argument. Both the queue of processes blocked on the semaphore, and the queue of processes ready to run are represented by **ProcessContainers**. When a V operation is performed on the semaphore, a blocked process can be removed from the semaphore's blocked process queue and added to the queue of ready to run processes.

The *Choices* process management classes are presented in detail in the following sections. Special attention is paid to how object-oriented programming and design have benefited its design and construction as well as supported optimizations of the model in a clean and modular manner.

[4] The thisProcess() function exists to obtain a reference to the current **Process**.

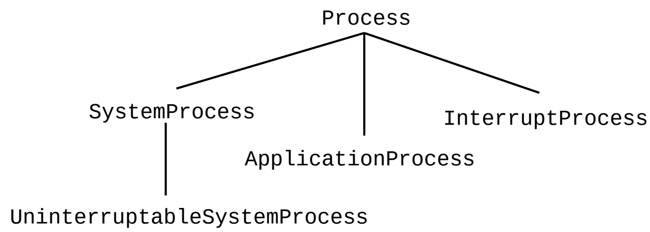

Figure 7.3: Some subclasses of the **Process** class

7.3 The Process Class

Subclasses of **Process** (see Figure 7.3) represent different kinds of processes. Each subclass reflects the requirements of the kind of process it represents. For example **InterruptProcesses** represent processes that manage device interrupts, **ApplicationProcesses** represent application program processes, **SystemProcesses** represent operating system management processes, and **UninterruptableSystemProcesses** represent very high priority system processes.

Each **Process** forms a repository for a process's context while the process is not active. Therefore, it stores the information necessary to resume the process's execution on a (possibly different) processor. Likewise, a **Process** is the object to which messages are sent to alter a process's context. For example, a **Process** can be sent messages to disable or enable interrupts, or set scheduling parameters or priorities. For efficiency, the amount of information kept per-process, and the context switching effort between two processes, is minimized. The context switching overhead in *Choices* is based on the kind of process relinquishing the processor and the kind of process being given the processor.

When a new process is created, the corresponding **Process** is parameterized by a **Domain**, a stack, an initial execution address, and arguments to the procedure at this address. The **Domain** argument defines the virtual address space within which addresses generated by the process are resolved (see Section 6.6.1). Usually the executable code, initialized data, uninitialized data, and stack are represented as separate **MemoryObjects** within this **Domain**. The initial execution address (specified as a method to initially invoke[5]) is a location within the address space defined by this **Domain**. It represents the initial value of the process's *program*

[5] Returning from this method terminates the execution of the process.

counter register.[6] The stack argument is a block of memory within the process's address space used by the process as its execution stack.

Each process in *Choices* shares memory with other processes through the memory management mechanisms described in Chapter 6. If the **Domain** argument used in the construction of a new **Process** is the same **Domain** as the process creating the new process, the new process will share all the memory of the creating process.[7] If it is an entirely disjoint **Domain**, the new process shares no memory with the creating process.[8] Or, if it is a **Domain** that the creating process has constructed containing only some of the same **MemoryObjects** as its own **Domain** and some other **MemoryObjects**, the new process shares only a portion of the creating process's address space. This latter case, and that once a process is executing it can modify its own **Domain** by adding or removing **MemoryObjects**, allows arbitrary shared/private regions to be set up between cooperating processes.

The context of a process takes a different form on every computer architecture. Usually it consists of a set of register contents, a program counter and a stack pointer. In order to increase portability by localizing architecture dependencies in as few places as possible, the state of a process in *Choices* is actually split between two objects, a **Process** and a **ProcessorContext**. **Processes** encapsulate all of the processor architecture *independent* information about a process. This information consists mainly of scheduling information and a reference to the process's **Domain**. The processor architecture *dependent* context of a process is kept in an associated **ProcessorContext**. **ProcessorContext** defines messages to be sent during context switching to save (the checkpoint message) and restore (the restore message) the processor architecture dependent context of a process. Subclasses of **ProcessorContext** represent the saved context of a process for specific architectures. Subclasses of these classes further refine the **ProcessorContext** checkpoint and restore methods to handle specific kinds of **Processes** on these architectures (see Figure 7.4).

The **Process** class defines messages for context switching, manipulating scheduling parameters, and enabling and disabling interrupts (the becomeUninterruptable and becomeInterruptable

[6]Various architectures give this register different names but the concept remains the same. It is the address of the current execution point.

[7]This form of sharing is similar to how a parent and child process share memory across the UNIX *vfork* primitive[BSD84]. The difference being that the two processes share *all* the memory. This allows the new process access even to the creating process's stack.

[8]This form of process creation is similar to the UNIX *exec* primitive.

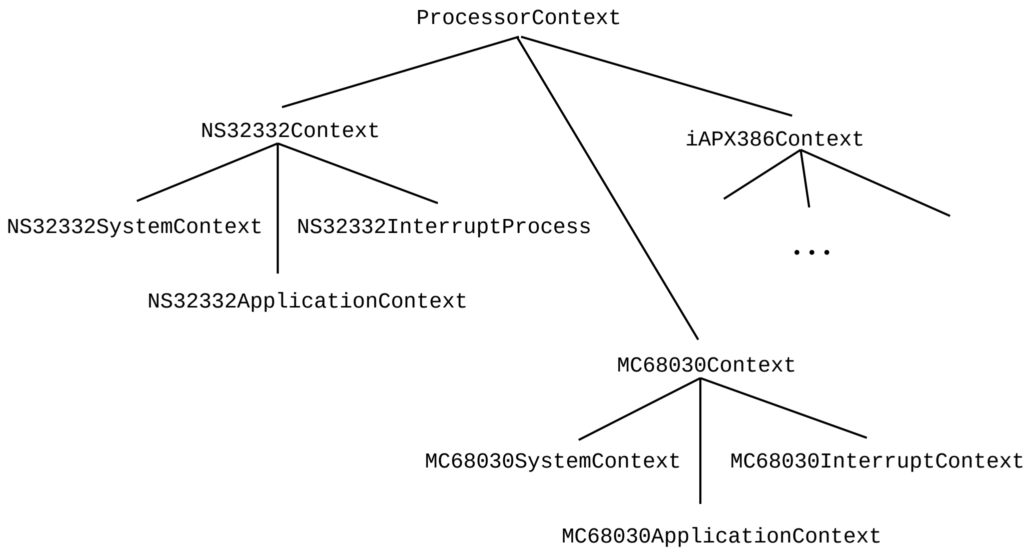

Figure 7.4: Sample subclasses of **ProcessorContext**

messages). The becomeUninterruptable and becomeInterruptable messages are actually forwarded by a **Process** to its corresponding **ProcessorContext** object since they are architecture dependent operations.

7.3.1 *Choices* Application Processes

Since the operating system must protect itself from application programs, processes executing on the behalf of applications do not execute in the privileged mode of the processor. This prevents application programs from accessing system objects. Chapter 5 discussed the way *Choices* allows application programs access to select system objects in a controlled manner.

Application processes do, however, need to execute system code when interrupts or traps occur. *Choices* supports this by implementing an application process as a pair of coroutines[Con65, Knu73]: the *system coroutine* and the *application coroutine*. These two coroutines have separate control stacks. The system coroutine runs in the protected mode of the processor with its stack in system protected memory. The application (or an application support library) manages the application coroutine's control stack. All *Choices* processes share this model but some (the system processes) have a non-existent application coroutine.

The application coroutine is the part of a process that executes the code of application programs and, therefore, is completely untrusted by the operating system. It executed in

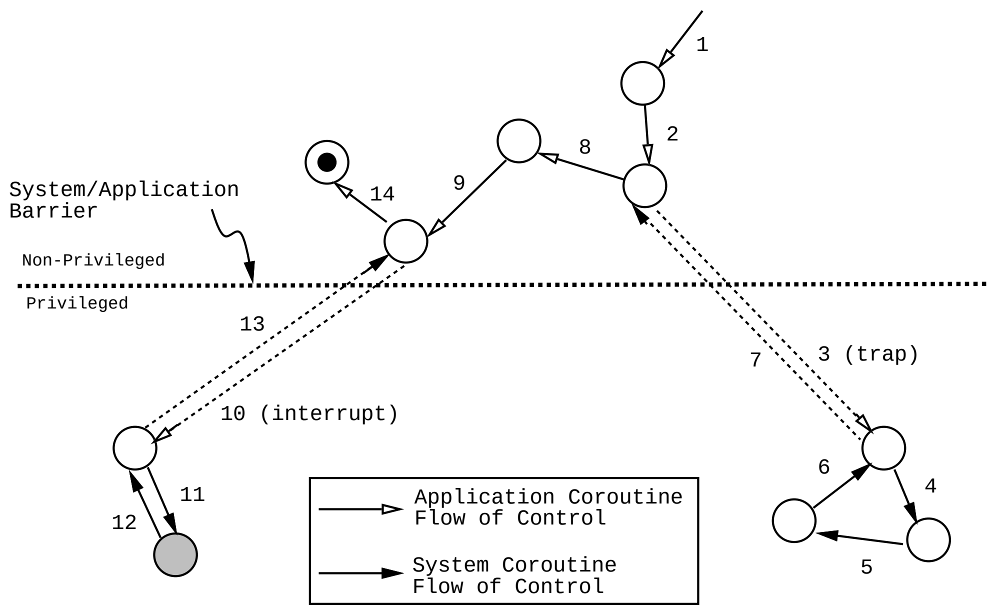

Figure 7.5: Application process coroutines

the non-privileged mode of the processor. The application coroutine never executes operating system code directly. Only the system coroutine does. For example, when a process crosses the system/application barrier by using the **ObjectProxy** mechanism described in Section 5.1.2, it traps into the operating system. The privilege level of the processor is raised and the system coroutine is resumed (see Figure 7.5). Resumption of the system coroutine can also occur as the result of any other trap or an interrupt. The *Choices* exception handling mechanism discussed in Section 7.7 is used to handle voluntary traps (usually **ObjectProxy** invocations), involuntary traps (program execution exceptions), and interrupts by resuming the system coroutine when any of these exceptions occur.

7.4 The Processor Class

A physical processor in *Choices* is represented by an instance of the **Processor** class. Multiprocessors are handled by having multiple instances of **Processor**, one per physical processor. An executing process can use the thisProcessor() function to obtain a reference to the **Processor** object managing the physical processor on which it is currently executing. **Processor** is an abstract class subclassed for each physical processor type to which *Choices* is targeted. Besides process scheduling, it provides other processor specific functions such as finding the handler for a particular trap or interrupt.

7.5 The ProcessContainer Class and Scheduling

ProcessContainer is an abstract class defining a signature for the storing and retrieving of **Processes**. This signature includes the messages add (for inserting **Processes** into the container), remove (for removing **Processes** from the container), and isEmpty (for testing whether the container contains any **Processes** or not). Subclasses of **ProcessContainer** impose queuing disciplines on the processes that they contain by implementing the add and remove methods to, for example, add **Processes** and remove them in FIFO or priority order.

Process scheduling in *Choices* follows the traditional running/ready/blocked model[Dei84a], but supports it within the *Choices* object-oriented framework. All queues of processes are represented as **ProcessContainers**.

7.5.1 The Per-Processor Container

In the running/ready/blocked model of process scheduling, all processes that could execute as soon as a processor is free are kept in the *ready-queue*. In *Choices*, the ready-queue is a **ProcessContainer**. Multiple ready-queues are supported and each processor may be assigned a different (or shared) queue. Having multiple ready-queues allows the partitioning of processes among groups of processors in a multiprocessing system. Each **Processor** references a **ProcessContainer** as its ready queue. This reference is stored in an instance variable named idleContainer since it refers to the **ProcessContainer** from which a **Process** is removed when the processor is idle.[9]

Sending the getNextReadyProcess message to a **Processor** returns the next process from that **Processor's** ready-queue. Scheduling and blocking objects send this message to select another process to run if they are not relinquishing the processor to a predetermined process. A typical example is when a process blocks on an I/O request.

[9]An idle processor results when the process currently executing on the processor invokes a method on a scheduling object that results in the processor being relinquished. Examples of this case include invoking the P operation on a busy semaphore, or blocking for an I/O completion.

The getNextReadyProcess method removes a process from the **Processor's** idleContainer. If the idleContainer is empty, getNextReadyProcess returns the **Processor's** idleProcess, which is a process that is always ready to execute. A **Processor's** idleProcess is always ready to execute since it just loops with interrupts enabled until another process is ready to run (the idleContainer is no longer empty). When another process is ready to run, the idleProcess relinquishes the processor to that process by sending itself giveProcessorTo.

7.5.2 The Per-Process Container

Since *Choices* supports multiple ready-queues, a mechanism is needed to decide in which queue a newly ready process belongs. This is solved by each **Process** maintaining a reference to the ready-queue to which it will be added when it is ready to execute. The reference is stored in an instance variable named readyContainer. A process's readyContainer is initially set to its creating process's value.

Sending a **Process** the ready message adds the **Process** to its readyContainer. After a **Process** is constructed and initialized, it is sent the ready message to allow it to run as soon as a processor becomes available. The ready message is also sent to unblock a blocked process after an I/O event completes or when a busy semaphore becomes free.

7.5.3 Scheduling Ready Processes

A traditional symmetric multiprogrammed system has a single ready-queue. This can be achieved in *Choices* by having all **Process** readyContainers and all **Processor** idleContainers reference the same **ProcessContainer** (see Figure 7.6). This balances the processes evenly over the processors by assigning each idle processor a process from the pool of all ready processes. Such a **ProcessContainer** must provide mutual exclusion on its internal data structures since it will require concurrent accesses by multiple processors. As the number of processors increases, the number of invocations of add and remove will increase proportionally. The mutual exclusion overheads may, therefore, lead to a bottleneck. In order to decrease contention, the ready-queue function may be distributed between multiple **ProcessContainers** each assigned to a subset of the available processors. **Processes** can then be migrated between these **ProcessContainers** when processors are idle or overloaded in order to balance the load.

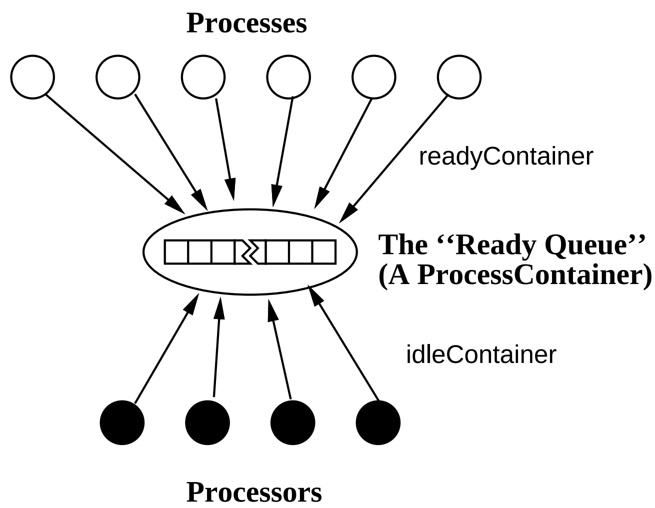

Figure 7.6: A traditional multiprogrammed system

Another use of multiple ready-queues is to simultaneously support both real-time and time-shared processes. One **ProcessContainer** could be dedicated to real-time processes and a different one to non-real-time processes. The readyContainer of all real-time processes could reference the real-time ready queue, and that of the timeshared processes reference the timeshared ready-queue (see Figure 7.7). The processors could be partitioned by setting the idleContainer of some processors to the real-time ready-queue, and the remaining processor's idleContainers to the timeshared ready-queue. In this way, real-time processors would never be assigned to timeshared processes. They would always be available for real-time processes only.

7.5.4 Deadlock and Race Avoidance

When a **Process** is added to a **ProcessContainer**, it can potentially be removed immediately and run on another processor. Therefore, if a process adds *itself* to a container, it may begin simultaneously executing on two processors. Even though this will only be the case for the very short period of time until the first processor begins running another process, it could be disastrous as both processors will execute with the same stack, overwriting each other's values. Therefore, a process is constrained to never add the **Process** representing itself to a **ProcessContainer**.

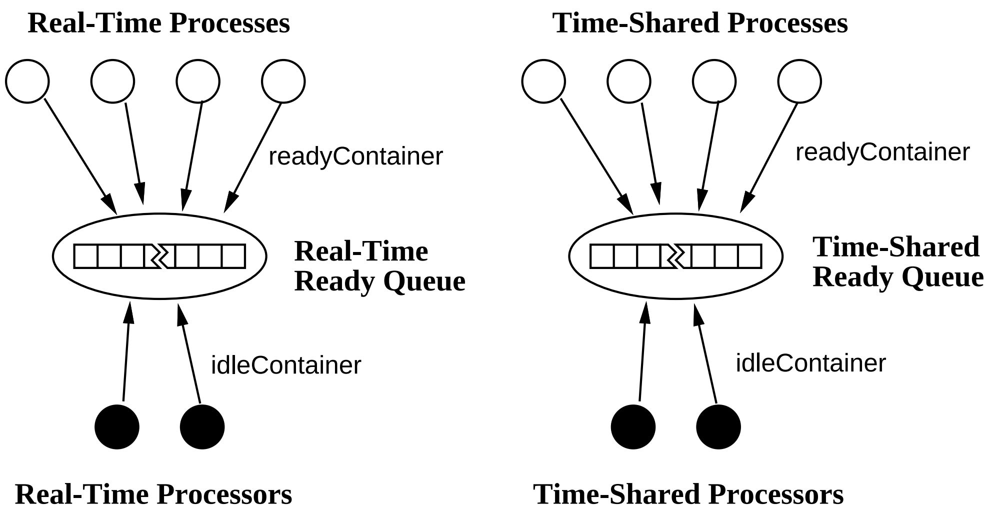

Figure 7.7: Processors partitioned between real-time and time-shared processes

Choices avoids the race condition by using a **ContextSwitchResponsibility**. A **ContextSwitchResponsibility** object to delegate the function of adding a running **Process** to a **ProcessContainer** to the process selected to run next. When a process relinquishes the processor it sends the giveProcessorTo message to the corresponding **Process**. Before doing this, however, it assigns a **ContextSwitchResponsibility** to the process being given the processor. The responsibility instance variable of a **Process** holds a reference to this **ContextSwitchResponsibility**. The **ContextSwitchResponsibility** class is abstract and defines a single message, perform. This message is sent to the responsibility of a process just after the process being given the processor is resumed, but before it returns to the point where it relinquished the processor. The argument of the perform method is the **Process** that just released the processor. Sending the perform message to a **Process's** responsibility can be thought of as part of the "cost" of giving the processor to that process. The **DefaultResponsibility** class is used when the process relinquishing the processor has no special needs. The perform method of **DefaultResponsibility** simply sends the ready message to the argument process so that it can execute again when a processor becomes idle. Other classes implementing the **ContextSwitchResponsibility** signature usually redefine perform to place the **Process** in a blocked or wait queue (see Section 7.8.2 for an example).

A possibility of indefinite postponement is avoided in the *Choices* process management system by disabling interrupts between the point just before a **Process** is removed from a **Processor's** idleContainer and the point that the currently executing process relinquishes the processor. The indefinite postponement avoided is the delay of running of the removed process due to the possibility of an interrupt occurring before the processor is relinquished.

7.6 Implementation of giveProcessorTo

The giveProcessorTo method saves the context of the current **Process** (the **Process** receiving the message) and restores the context of the process represented by the argument. The argument to giveProcessorTo is determined by a scheduling or synchronization object. The giveProcessorTo message is the only *mechanism* with which to affect a context switch. The *policy* of which processes to run is determined by other objects, such as an exception handling object or a semaphore. Eventually, a **Process** that was sent giveProcessorTo will itself be an argument to a giveProcessorTo message send. At this time, that process will resume execution and will appear to have finally returned from the earlier invocation of giveProcessorTo.

The **Process** giveProcessorTo message is the interface **Process** presents for context switching. Most of the work of context switching is actually done by two separate **Process** messages (save and restore) and two **ProcessorContext** messages (checkpoint and restore).

The full implementation of the giveProcessorTo method of **Process** is shown in Figure 7.8. The method first sends the save message to the current **Process** to store any architecture independent information about the process prior to relinquishing the processor. Once that is completed, the checkpoint message is sent to the **Process's** corresponding **ProcessorContext**. This stores the architecture dependent context of the **Process** in the **ProcessorContext** object. The way the checkpoint method of a **ProcessorContext** is actually implemented parallels the setjmp/longjmp mechanism of UNIX[SVI85]. Sends of the checkpoint message appear to return twice. The first time checkpoint returns is when the context is saved. The value returned is a null reference. The second time checkpoint returns is when the context is resumed from its saved state by another process. The value returned this time is a reference to the **Process** that gave the resumed process the processor.

```
Process::giveProcessorTo( anotherProcess )
{
    // save the architecture independent
    // information about this process.
    save();

    // take a checkpoint of the current context.
    processWhoResumedUs = context.checkpoint();
    if( processWhoResumedUs == NULL ) {

        // you reach here when the process
        // first takes a checkpoint.
        currentProcess = anotherProcess;
        anotherProcess.context.restore();
    }

    // you reach here when the process is
    // resumed by another process
    restore( processWhoResumedUs );
}
```

Figure 7.8: The implementation of the giveProcessorTo method

Since checkpoint and restore have such abnormal behavior, and since they need access to the entire state of the processor, they are implemented in assembly language in *Choices*. Figures 7.9 through 7.11 give the implementation of checkpoint and restore for a system process's context[10] on a few common processors.[11]

The first time checkpoint returns, the restore message is sent to the **ProcessorContext** of the **Process** to be run next. That process then appears to return from its own earlier send of the checkpoint message a second time. This is the point where the physical flow of control is actually transferred between processes.

By the second time the send of checkpoint returns, the architecture dependent context of the new process has been restored. The giveProcessorTo method then sends the restore method to

[10] The checkpoint and restore methods of subclasses of **ProcessorContext** for other kinds of processes only differ in the amount of state saved.

[11] I implemented the NS32332 version. Subsequently, Dave Dykstra implemented the 80386 version and Bjorn Helgaas implemented the MC68030 version.

```
// Upon entry to a method, the calling convention on the 32332 has put a pointer to
// the ProcessorContext object itself in r0, and the first argument to
// the method in r1. The return address will be on the top of the stack,
// followed by any additional arguments to the method. Registers r0, r1
// and r2 are assumed not to be preserved across a procedure call.
// The result of a function is returned in r0.
```

```
NS32332SystemContext::checkpoint()
{
    // first, save the general registers
    movd    r3, R3offset(r0)
    movd    r4, R4offset(r0)
    movd    r5, R5offset(r0)
    movd    r6, R6offset(r0)
    movd    r7, R7offset(r0)

    // next, save the frame pointer
    movd    0(fp), FPoffset(r0)

    // next, save the stack pointer (actually
    // pretend the return address has been
    // popped off the stack)
    addr    4(sp), SPoffset(r0)

    // restart at the address on
    // the top of the stack
    movd    0(sp), PCoffset(r0)


    // return 0 when first called
    movqd   $0, r0

    // return to the caller
    // the first time
    ret     $0
}
```

```
NS32332SystemContext::restore( oldProcess )
{
    // first, restore the general registers
    movd    R3offset(r0), r3
    movd    R4offset(r0), r4
    movd    R5offset(r0), r5
    movd    R6offset(r0), r6
    movd    R7offset(r0), r7

    // next, restore the frame pointer
    lprd    fp, FPoffset(r0)

    // next, restore the stack pointer
    lprd    sp, SPoffset(r0)

    // save the restart address in r2
    movd    PCoffset(r0), r2

    // return the first argument
    // (the process that called giveProcessorTo)
    movd    r1, r0

    // ``return'' from checkpoint the
    // second time
    jump    0(r2)
}
```

Figure 7.9: The checkpoint and restore methods for the National Semiconduction NS32332

```
// Upon entry to a method, the calling convention on the 80386 has put the return address on the
// top of the stack, followed by a pointer to ProcessorContext object itself.
// Any additional arguments to the method will follow on the stack. Registers %eax
// and %edx are assumed not to be preserved across a procedure call.
// The result of a function is returned in %eax.
```

```
i386SystemContext::checkpoint()
{
    mov     4(%esp), %eax

    mov     %edi, EDIoffset(%eax)
    mov     %esi, ESIoffset(%eax)
    mov     %ebx, EBXoffset(%eax)
    mov     %ebp, EBPoffset(%eax)
    mov     %esp, ESPoffset(%eax)
    addl    $4, ESPoffset(%eax)
    mov     (%esp), PCoffset(%eax)
    mov     $0, %eax
    ret
}
```

```
i386SystemContext::restore( oldProcess )
{
    mov     4(%esp), %edx
    mov     8(%esp), %eax
    mov     EDIoffset(%edx), %edi
    mov     ESIoffset(%edx), %esi
    mov     EBXoffset(%edx), %ebx
    mov     EBPoffset(%edx), %ebp
    mov     ESPoffset(%edx), %esp

    mov     PCoffset(%edx), %edx

    jmp     *%edx
}
```

Figure 7.10: The checkpoint and restore methods for the Intel i80386

```
// Upon entry to a method, the calling convention on the MC68030 has put the return address on the
// top of the stack, followed by a pointer to ProcessorContext object itself.
// Any additional arguments to the method will follow on the stack. Registers d0, d1, a0
// and a1 are assumed not to be preserved across a procedure call.
// The result of a function is returned in d0.

MC68030SystemContext::checkpoint()            MC68030SystemContext::restore( oldProcess )
{                                             {
    moveml  d0-d7/a0-a6, a0@(RXoffset)            moveml  a0@(RXoffset), d0-d7/a0-a6
                                                  movel   sp@(8), d0
    lea     sp@(4), a1
    movel   a1, a0@(SPoffset)                     movel   a0@(SPoffset), sp
    movel   sp@, a0@(PCoffset)                    movel   a0@(PCoffset), a0

    clrl    d0
    rts                                           jmp     a0@
}                                             }
```

Figure 7.11: The checkpoint and restore methods for the Motorola MC68030

the current **Process** passing along the reference to the **Process** that relinquished the processor. The **Process** restore method reinstates any architecture independent context of the **Process** and sends the perform message to the **Process's** responsibility. Finally, the giveProcessorTo method returns and the process resumes execution.

When a new **Process** is created, its initial context is stored in a form suitable for the **Process** and **ProcessorContext** restore methods to use. This allows new **Processes** to begin execution at their entry point the first time they are the argument to a send of the giveProcessorTo message.

The giveProcessorTo message must be invoked with interrupts disabled. This prevents partially saved contexts occurring as the result of a context save being interrupted. Disabling interrupts is usually guaranteed by meeting the condition that interrupts also be disabled when a **Process** is removed from a **Processor's** idleContainer. Thus, the normal chain of events for a synchronization object that wishes to transfer the physical flow of control to a new process is to:

1. Disable interrupts if they are not already disabled. This is accomplished by sending the current **Process** the becomeUninterruptable message.

2. Choose the process to run next. This might be determined by the object itself, as in the case of invoking a V operation on a semaphore that has blocked processes, or by sending the getNextReadyProcess message to the current **Processor**.

3. Set the responsibility of the process to run next.

4. Send the giveProcessorTo message to the **Process** object representing the current process. The **Process** to run next is used as an argument.

5. Once control returns as the result of being given the processor back, re-enable interrupts if they were enabled to start with by sending the current **Process** the becomeInterruptable message.

7.6.1 Optimizing Context Switching

Many operating systems are designed to support *lightweight processes*. The goal of these systems is to minimize the cost of using multiple processes by minimizing the cost of context switching between them. The performance of using multiple processes in an application could be prohibitive if the expense of synchronizing and switching between them is too large. Likewise, interrupt and real-time processing requires that the overhead of context switching between processes be minimized. In reality, what this motivates is the need for *lightweight context switching* between processes. The "weight" of the process itself is often not at issue, rather the expense of context switching between processes.

Lightweight context switching is usually implemented by having processes share as much state as possible with each other. This reduces the time to switch between them as common state does not need saving or restoration. It also has the added advantage of reducing the amount of memory dedicated to storing per-process information. In *Choices*, the most commonly shared state is a process's address space.

Lightweight context switching is addressed in *Choices* by having subclasses of **Process** and **ProcessorContext** redefine their context switching methods in a way corresponding to the kind of process. For example, the *Choices* system code as implemented in the current prototype uses no floating point arithmetic. Therefore, it does not use the floating point registers. However, applications might require the floating point registers. Saving these registers when a system or interrupt process relinquishes the processor and restoring them upon its restart would be wasteful since their values are irrelevant. Therefore, only the particular subclass of **ProcessorContext** for **ApplicationProcesses** redefines checkpoint and restore to save and restore floating point registers.

This exemplifies an advantage of polymorphism when applied to operating systems. Important primitives like giveProcessorTo can be transparently optimized by redefining methods that they in turn rely on. Another advantage applies to easing operating system portability. When initially retargeting *Choices* for a new architecture, it is easiest to implement the checkpoint and restore methods in the parent **ProcessorContext** class for that architecture in such a way as to save and restore the entire context of the processor. The subclasses for various kinds of processes can then inherit these methods. Optimizations like the one in the previous paragraph can be added later by redefining the methods in the subclasses. This is a specific case of specialization by subclassing. The most general case can be implemented first in one class. Later, this class can be subclassed to implement optimizations that increase performance.

7.6.2 Memory Management Issues and Context Switching

Saving and restoring registers is not the only cost when context switching between two processes. Changing active address spaces is also costly and can be the most expensive part of a context switch. However, when two processes that share an address space (**Domain**) exchange the processor, no memory management related context switching costs are incurred.

The restore method of **Process** manages the changing of **Domains**. Context switching overhead in *Choices* is lowest between processes that share a common **Domain**. Within the restore method, if the **Domain** of the process being given the processor matches the **Domain** of the invoking process, then no memory context switching overhead is introduced. If the two processes have different **Domains**, the **AddressTranslation** of the **Domain** of the invoking process must be deactivated on the processor's corresponding **AddressTranslator** and the **AddressTranslation** of the **Domain** of the other process must be activated. The expense of altering the currently active **AddressTranslation** varies from architecture to architecture but usually involves flushing the contents of various translation caches and registers and reloading them for the new process.

Changing active **AddressTranslations** is not the only cost when switching from one **Domain** to another. Inactive **MemoryObjects** can have their working set of data swapped out to backing store in order to keep physical memory available. For this reason, little or none of the data necessary for the execution of a process being resumed may be resident in physical memory. All a process's working set must be swapped back in to allow the process to execute. While it does not increase the cost of the context switch itself, the work done fixing the address translation faults that will occur once the process begins referencing non-resident locations in its working set will increase the time the newly resumed process will spend in the system before it can do any useful work.

The restart time of a process due to reloading non-resident memory can be reduced by locking the data of critical **MemoryObjects** resident in physical memory. This is achieved by creating a **PhysicalMemoryChain** representing all the units in such a **MemoryObjects** and never deleting that **PhysicalMemoryChain** (see Section 6.3.3). This optimization is used sparingly since it decreases the amount of physical memory in the system available for data from other **MemoryObjects**.

Choices system objects are addressable from within any **Domain**. An interrupt or system process, or the system coroutine of an application process can, therefore, execute in any **Domain**. This is why the code for checkpoint and restore in Figures 7.9 through 7.11 can reload stack pointers and reference saved contexts of other processes without having to alter the active **Domain**.

7.6.3 Context Switching Performance

Table 7.1 shows the context switching overheads between different kinds of processes. These numbers were acquired on a version of *Choices* running on a six processor Encore Multimax with NS32332[Nat86] processors running at 15MHz. The measurements were made with two processes executing a loop in which each process relinquishes the processor to the other process by sending itself the giveProcessorTo message with the other process as an argument. In all these cases, the processes were executing in the same virtual address space (**Domain**). Both application processes that use floating point and those that do not were measured. The time

	System	Application	FP Application
System	88 μs	163 μs	176 μs
Application	163 μs	221 μs	233 μs (estimated)
FP Application	176 μs	233 μs (estimated)	244 μs

Table 7.1: System and application process context switch times

	System	Application	FP Application
System	88 μs	289 μs	315 μs
Application	289 μs	370 μs	391 μs (estimated)
FP Application	315 μs	391 μs (estimated)	412 μs

Table 7.2: Effect of different virtual address spaces

for a context switch from one system process to another is $88\mu s$. Such a context switch only requires the saving and restoring of processor registers used by the system code, i.e., not the floating point registers, application stack pointer, etc. The time for a context switch between two application processes that do not use floating point is $221\mu s$. When floating point is used, the overhead increases to $244\mu s$ as a result of saving and restoring the floating point registers.

Table 7.2 repeats these measurements but this time the second process is given a different **Domain** to execute in. The additional overhead incurred when the **Domains** differ derives from flushing the MMU cache and reloading the page tables for the new **Domain**. Note that the time to switch between system processes remains unchanged. Because system processes can execute in any **Domain**, the *Choices* context switching code is optimized to not alter the active **Domain** whenever two system processes exchange the processor. This is accomplished by redefining the **SystemProcess** restore method *not* to activate a new **Domain** when it is resumed. The currently active **Domain** is used instead. The context switch between two floating point application processes in different domains is the largest of all context switches at $412\mu s$.

7.7 Exceptions

When an exception occurs, the normal flow of control of a process is suspended and an exception handler is invoked. Virtually all architectures automatically save some of the processor context and enter privileged execution mode when an interrupt or trap occurs. The saved context is usually pushed on an interrupt stack or saved in special registers. Once this context is saved, control passes to an operating system entry point. There may either be a common entry point for all exceptions (in which case some identification of which exception occurred is supplied), or the hardware may consult a table mapping exceptions to entry points an use entry point per exception. When the handler at the entry point returns, a *return-from-trap* or *return-from-interrupt* instruction returns the processor to its pre-exception state. In particular, if the process was running in non-privileged mode before the exception, it will be placed back in non-privileged mode.

Exception management in *Choices* is encapsulated by the abstract **Exception** class. Both interrupts and traps are managed by instances of **Exception** subclasses. The **Exception** class defines the handle message to correct an exception condition. When *Choices* is initialized, all hardware traps and interrupts are mapped to **Exceptions**. When a trap or interrupt occurs, the handle message is sent to the corresponding **Exception**. If the exception condition cannot be repaired or handled, the currently executing process's execution is terminated.

When a trap or interrupt occurs in *Choices*, the system coroutine of the currently executing process is resumed at an operating system entry point. If a process's system coroutine was already executing when the exception occurred, the exception causes the process to perform a branch directly to the entry point. At such an entry point, architecture dependent code is invoked to translate the condition to the corresponding **Exception** object (see Figure 7.12).

The *Choices* exception handling mechanism assumes that sending the handle message to an **Exception** will always return and the trapping or interrupted process will be resumed where it left off. Because of this assumption, the architecture dependent code that handles the exception only needs to save any process context that the method invocation mechanism will not save across the send of the handle message (the method volatile context). In particular, since method invocation is implemented in C++ with procedure calls, this context consists of the set of registers that are not saved across a procedure call. The send of the handle message has no other net effect on the rest of the processes context, so upon its return, only the method volatile context needs to be restored before the process is resumed.

In general, the exception condition is handled by the system coroutine of the current process. For example, a page fault is repaired without resuming another process. This is implemented by having the handle method of the particular subclass simply repair the exceptional condition and return. If it cannot be handled by the system coroutine of the process active when the

```
OperatingSystemEntryPoint( condition )
{
        save method invocation volatile context

        convert condition into an  Exception

        Exception->raise();

        restore method invocation volatile context

        return-from-exception;
}
```

Figure 7.12: Pseudo-Code for an operating system entry point

exception occurred, the handle method sends giveProcessorTo to relinquish the processor to another process (usually an **InterruptProcess**) that handles the exception condition. When the original process is resumed, it returns back to the handle method that was originally invoked and then returns from there.

Exceptions used to handle interrupts implement handle by making the interrupted process ready to run again and then signalling the exception condition to another process using a semaphore. On a multiprocessor, this allows the interrupt to be processed on one processor while the interrupted process continues to execute on another processor.

7.7.1 Types of Exceptions

The **AwaitedInterruptException** subclass of **Exception** defines a new message, await. The await message is sent to an **AwaitedInterruptException** to block the current process's execution until the interrupt occurs, at which time it can be resumed. The **Process** is blocked by placing it in a **ProcessContainer** associated with the **AwaitedInterruptException**. If the interrupt occurs before the **AwaitedInterruptException** is sent the await message, its occurrence is logged so that the next invocation of await will return immediately. Otherwise, if there was a process waiting then it is resumed. The resumption occurs by sending giveProcessorTo to the current process with the awaiting **Process** as the argument. A **DefaultResponsibility** is assigned to the new process so that the ready message will be sent to the interrupted process and it can run on another processor if one is available.

In addition to interrupts that have processes awaiting them, unawaited interrupts are supported by *Choices*. For example, a time-slice interrupt is handled by an instance of the **TimeSliceInterrupt** class. When a time-slice interrupt occurs, another process is chosen from the current **Processor's** idleContainer. The chosen process is assigned a **DefaultResponsibility** so that the interrupt process can be run as soon as there is a free processor. The current Process is then told to give the processor to the new Process by sending giveProcessorTo. If no other processes are ready to execute, the **TimeSliceInterrupt** handle method simply returns and the interrupted process is resumed.

7.8 Process Concurrency

If multiple processes are to cooperate then they must often synchronize their activities. Process synchronization problems generally fall into one of two categories. The first arises when multiple processes simultaneously attempt to access a common object. If the accesses must be performed atomically (i.e. one must be fully completed before any other is begin) then they are said to be *mutually exclusive*[PS85]. The instructions that require mutually exclusive access to the common object are termed a *critical section*. Mutual exclusion requires any process attempting to enter a critical section to be blocked while another process is executing the critical section.

The second process synchronization problem arises when one set of processes is producing data that another set requires. This is termed *producer-consumer* synchronization[PS85]. Producer-consumer synchronization may require the consumers to be blocked until the producers have generated more data, or the producers to be blocked until the consumers have processed already produced data.

Choices provides spin-locks (implemented by the **Lock** class) and busy-wait loops (implemented by the **BusyWait** class) for mutual exclusion and semaphores (implemented by the **Semaphore** class) for both mutual exclusion and synchronization.

	acquire	release
Lock	9.38 μs	3.35 μs
BusyWait	5.27 μs	0.437 μs

Table 7.3: Cost of the **Lock** and **BusyWait** methods on the Multimax

7.8.1 Locks and BusyWaits

Locks are provided for lightweight mutual exclusion. The implementation of **Lock** assumes that the processor will not be relinquished while the **Lock** is held. The acquire message is sent to a **Lock** to enter a critical section. The corresponding method simply disables interrupts, and uses a test-and-set loop to wait for the **Lock** to be free. The release message is sent to a **Lock** to indicate that the **Lock** is free. The corresponding method releases the lock and re-enables interrupts if they were enabled when the **Lock** was first acquired. Since there can only be one processor competing for a **Lock** at a time on a uniprocessor computer, versions of *Choices* for such computers are free to implement the **Lock** acquire method solely as disable interrupts and release as re-enable interrupts (if there were enabled when acquire was sent).

BusyWaits are simplified versions of **Locks** that, for efficiency, assume that the enabling and disabling of interrupts is handled by the code using the **BusyWait** rather than by the **BusyWait** itself. A **BusyWait**, therefore simply implements a test-and-set loop. **BusyWaits** are useful for mutual exclusion cases where interrupts are known to already be disabled.

Table 7.3 gives the costs of the acquire and release methods on the Encore Multimax for both the **Lock** and **BusyWait** classes. It should be noted that the C++ compiler in line expands the acquire and release methods for both classes. This explains why the cost of sending the release message to a **BusyWait** is less than the minimum message send costs reported in Chapter 5.

7.8.2 Semaphores

A semaphore is implemented by the **Semaphore** class and its P and V methods.[12] The definition of the **Semaphore** class is shown in Figure 7.13. The implementation of the **Semaphore** class's methods are given in Figures 7.14 through 7.16.

[12] The names of these methods violate the rule of method names not being capitalized. They were chosen to reflect the corresponding operations on a Dijkstra semaphore.

```
class Semaphore : public Object {
protected:
	BusyWait mutex;
	int count;
	ProcessContainer * queue;
	SemaphoreResponsibility mySemaphoreResponsibility;
public:
	Semaphore( int initialCount );
	~Semaphore();

	virtual void P();
	virtual void V();
};
```

Figure 7.13: The *Choices* **Semaphore** class

As shown in Figure 7.14, the implementation of the P method first disables interrupts then acquires mutually exclusive access to the semaphore count by sending the acquire message to the mutex instance variable (a **BusyWait**). It then decrements the count and tests to see if the invoking process must block. If the count is greater than or equal to zero then the invoking process can continue. This is accomplished by sending release to the mutex instance variable in order to release mutually exclusive access on the semaphore count and then re-enabling interrupts and returning.

If the count went negative, then the invoking process must be blocked. The invoking process cannot add itself to the semaphore's wait queue for the same reasons discussed in Section 7.5.4. It must be added by the process being given the processor. To accomplish this, first getNextReadyProcess is sent to the current **Processor** in order to get another process to run. Then, that **Process's** responsibility is set to the mySemaphoreResponsibility **Semaphore** instance variable. The mySemaphoreResponsibility variable is an instance of the **SemaphoreResponsibility** class. **SemaphoreResponsibility** redefines perform to place the blocking process in the semaphore's wait queue (see Figure 7.15). Finally, the processor is relinquished and the next process is run by sending giveProcessorTo to the current **Process**.

```
Semaphore::P()
{
    int wasInterruptable = thisProcess()->becomeUninterruptable();
    mutex.acquire();
    count = count - 1;
    if( count < 0 ) {
        Process * nextProcess = thisProcessor()->getNextReadyProcess();
        nextProcess->setResponsibility( mySemaphoreResponsibility );
        thisProcess()->giveProcessorTo( nextProcess );
    }
    else {
        mutex.release();
    }
    if( wasInterruptable ) thisProcess()->becomeInterruptable();
}
```

Figure 7.14: Implementation of the **Semaphore** P method

```
class SemaphoreResponsibility : public Responsibility {
    Semaphore * mySemaphore;

    SemaphoreResponsibility( Semaphore * sem ) { mySemaphore = sem; };
    void perform( Process * oldProcess );
};

SemaphoreResponsibility::perform( Process * oldProcess )
{
    // mySemaphore is an instance variable set when the
    //  SemaphoreResponsibility is created. It references
    // the associated semaphore.
    mySemaphore->queue->add( oldProcess );
    mySemaphore->mutex.release();
}
```

Figure 7.15: Implementation of the **SemaphoreResponsibility** class

```
Semaphore::V()
{
    int wasInterruptable = thisProcess()->becomeUninterruptable();
    mutex.release();
    count = count + 1;
    if( count <= 0 ) {
        Process * waiter = queue->remove();
        mutex.release();
        waiter->ready();
    }
    else {
        mutex.release();
    }
    if( wasInterruptable ) process->becomeInterruptable();
}
```

Figure 7.16: Implementation of the **Semaphore** V method

The V method is implemented by similarly acquiring mutually exclusive access to the semaphore count, incrementing the count, and testing if the count is still negative or zero. If the count is positive, then no other processes are blocked and the invoking process can be removed by releasing the mutual exclusion on the count, re-enabling interrupts, and returning. If there is a process waiting, then it is removed from the **Semaphore's** block queue and sent the ready message.

7.8.3 Alternate Semaphore Implementations

The **GraciousSemaphore** subclass of **Semaphore** implements a form of semaphore that causes the current process to immediately relinquish the processor when the V messages is sent the **GraciousSemaphore** and there are blocked processes. The executing process suspends itself and resumes a waiting process by sending itself giveProcessorTo with the **Process** corresponding to one of the waiting processes as an argument. The implementation of the **GraciousSemaphore** V method is given in Figure 7.17.

```
GraciousSemaphore::V()
{
    int wasInterruptable = thisProcess()->becomeUninterruptable();
    mutex.release();
    count++;
    if( count <= 0 ) {
        Process * waiter = queue->remove();
        mutex.release();
        thisProcess()->giveProcessorTo( waiter );
    }
    else {
        mutex.release();
    }
    if( wasInterruptable ) thisProcess()->becomeInterruptable();
}
```

Figure 7.17: The **GraciousSemaphore** V method

P	V
65.7 μs	65.4 μs

Table 7.4: Cost of the **Semaphore** methods on the Multimax

7.8.4 Semaphore Performance

Table 7.4 gives the cost of sending the P and V messages to a **Semaphore** on the Encore Multimax. The data were acquired by first sending the P message to a **Semaphore** with a positive count, and then sending the V message. Therefore, these numbers reflect the cost of acquiring a free **Semaphore** and releasing a **Semaphore** on which no other processes are blocked. If a **Semaphore** is not free when the P message is sent to it, then the time before the P method returns depends on the length of time until another process sends the V message to the **Semaphore**. If another process were blocked awaiting the **Semaphore** when the V message was sent, then the time would increase by the amount of time necessary to dequeue the waiting process and send it the ready message.

To give an estimate of the overhead of using semaphores for synchronization, a test was run where two system processes looped exchanging the processor by alternately sending the P and V messages to a pair of semaphores. The first process sent the V message to one semaphore then the P message to another, while the second process did the opposite. **GraciousSemaphores** were used for this test to affect immediate context switches when the semaphore V operations were performed. The result is that four context switches occur for each iteration of the loop. A single iteration of the loop took 659 μs. Using the data in Table 7.1, a context switch between a pair of system processes takes 88 μs. Subtracting off the cost of the four context switches and halving, therefore, leaves a minimum overhead of approximately 154 μs from the time a process blocks on a semaphore as the result of a P, until the process is restarted.

7.9 Summary

The *Choices* process management provides an object-oriented interpretation and implementation of process scheduling, context switching, and exception handling. *Choices* attempts to let the operating system optimize context switching based on the requirements of the processes themselves. This is accomplished by using an abstract class to define a process and then subclassing that class to implement the process abstraction with various performance enhancements. Exception handling in *Choices* is likewise mapped into the object-oriented paradigm. An object represents each exception which can occur. That object is sent a message whenever an exception occurs.

Object-oriented interpretations of process management and exception handling have resulted in opportunities for both performance tuning and portability advantages in *Choices*. Portability is increased by localizing processor dependencies in the **ProcessorContext** objects. To quickly get *Choices* "up and running" on a new architecture, an initial **ProcessorContext** class for a new processor architecture can implement a full context save/restore with its checkpoint and restore methods. Later, once the system is stable and running, this class can be subclassed to implement the performance optimizations made possible by taking advantage of the requirements of the particular kind of process the **ProcessorContext** represents. Even once the new classes are implemented, inheritance still allows them to share common code through the superclass.

The *Choices* giveProcessorTo primitive provides the mechanism to affect context switching between processes. Policy decisions are implemented by different scheduling and synchronization objects, for example, **Semaphores** or **AwaitedInterruptExceptions**. In many cases, *Choices* uses inheritance to alter policy decisions. For example, the **GraciousSemaphore** class redefines the V method of **Semaphore** to implement a different policy when a process blocked on a semaphore is resumed. Even a **Semaphore** may have different policies. The implementation of **Semaphore** only relies on the interface **ProcessContainer** provides. Therefore, various classes implementing the **ProcessContainer** signature can be used to alter the behavior of a **Semaphore**. For example, the **ProcessContainer** implementing the queue of blocked processes may implemented a FIFO or priority-based scheme.

Process context switching is a very low level architecture dependent operation. The examples in Figures 7.9 through 7.11 show that the object-oriented paradigm can successfully handle such low level details as context switching, while at the same time presenting abstract, reusable interfaces.

In summary, *Choices* provides a flexible, efficient, and object-oriented interpretation of process context switching, scheduling, and exception handling.

Chapter 8

Conclusions

As discussed in Chapter 3, object-oriented programming techniques have been reported to provide many software engineering benefits. The goal of this thesis is to evaluate the success of object-oriented techniques at addressing problems of operating system portability, maintainability, extensibility and efficiency as discussed in the first two chapters. This goal is met by detailing an experiment to design and implement an operating system using object-oriented techniques throughout. The result of this experiment is the *Choices* architecture for operating system design and construction. *Choices* provides an extensible model of the internal framework of an operating system that does not penalize performance. It includes a hierarchy of software classes that can be specialized to build operating systems for particular applications or architectures. The hierarchy contains *abstract* primitive classes to define interfaces and abstractions, and *concrete* subclasses to define objects with a particular desired behavior or that implement a particular algorithm. *Choices* is successful at using object-oriented programming techniques to implement common operating system algorithms and data structures while remaining efficient. In this conclusion, by using examples from *Choices*, I evaluate how well in practice the proposed benefits of constructing object-oriented operating systems put forth in Chapter 4 were achieved.

8.1 Portability

This thesis illustrates in numerous places how object-oriented programming techniques can help increase operating system portability. Abstract classes such as the **AddressTranslation** and **AddressTranslator** classes described in Section 6.2 and the **ProcessorContext** class described in Section 7.3 all successfully hide the underlying details of a particular machine architecture. Concrete classes implementing these abstract signatures are a convenient way to encapsulate architectural dependencies. They simultaneously preserve the interface and isolate architecture independent code from the specifics of any particular computer architecture.

Higher level machine dependencies are also easily encapsulated within abstract classes. For example, *Choices* successfully uses subclasses of the **MemoryObject** class described in Section 6.4 to encapsulate hardware dependencies of various permanent storage disks and disk interfaces used by the various computers to which *Choices* has been targeted.

8.2 Code and Interface Sharing and Reuse

Examples of code sharing/reuse and interface sharing abound in *Choices*. Often it is just an interface that is shared. For example, the interface defined by the **ProcessorContext** class is implemented for various processor scheduling algorithms as described in [Ley88]. Often much code is shared as well. For example, the concrete versions of **AddressTranslation** for various architectures substantial amounts of code via an abstract superclass. Likewise, subclasses of **ProcessorContext** specific to different kinds of process on different architectures all share code through the superclass for that architecture (see Section 7.3).

Another example of interface sharing is described in Section 6.4. The interface provided by the **MemoryObject** class is inherited by a large number of classes including classes to represent: Berkeley and System V UNIX inodes[MLRC88], MS-DOS files [MCRL88], disks from the Encore Multimax and AT&T WGS-386 computers, partitions of disks, and subranges of other **MemoryObjects**. The Berkeley and System V UNIX inode classes, likewise, share substantial code through a **UnixInode** superclass.

8.3 Separation of Policy From Mechanism

Choices demonstrates how policy and mechanism can be separated cleanly by object-oriented programming. The **ProcessContainer** class is a good example of defining a policy interface with a signature and implementing that signature for numerous policies with concrete classes.

Once the signature for **ProcessContainer** was defined, many subclasses were implemented by an individual mostly unaware of the rest of the *Choices* process management system[Ley88].

Two other good examples of the separation of policy from mechanism can be found in the virtual memory system. First, the selectUnitForRemoval method of the **MemoryObjectCache** class is redefined to implement different memory replacement policies (see Section 6.5). Second, **MemoryObject** classes can redefine the policy governing layout and placement of logical data on permanent storage by redefining the read and write methods.

8.4 Optimization Through Specialization

The best example of using subclassing to allow optimizations by specialization occurs in the *Choices* process management system. The subclassing of **Process** and **ProcessorContext** in order to optimize context switching as described in Section 7.3 yields significant performance increases as summarized in Section 7.6.3.

8.5 Trading Portability for Efficiency

The example of building a heavyweight **ProcessorContext** for a particular architecture then subclassing it later for performance increases, as described in Section 7.6.1 supports the proposition that portability can be an initial goal, but that subclassing can later be used to optimize the system without impacting existing code.

8.6 Adaptable Interfaces

An object-oriented operating system as defined in Chapter 4 requires an object-oriented interface to system services. The **NameServer** and **ObjectProxy** mechanisms described in Chapter 5 provide this for *Choices*. Together, they provide a very flexible application interface that is customizable to individual processes. The performance of this mechanism is shown in Section 5.2 to be comparable to the interfaces provided by traditional operating systems.

8.7 Efficiency

The overall efficiency of *Choices* detailed throughout Chapters 5, 6, and 7 lends credence to the ability of object-oriented programming techniques to support efficient operating system software development. With few tools and little implementation experience, we were able to develop a system which performs within a few percent of UNIX on the same hardware. Some features of *Choices* are actually faster than UNIX. Much of this performance is owed to the efficient of C++. However, since *Choices* does not rely on any features specific to C++, using any efficiently compiled object-oriented language with the characteristics discussed in Section 4.4.2 should yield similar results.

8.8 Future Work

The most promising direction to take this research next is towards distributed systems. The uniformity of accessing data and performing computation using object message sending should be an invaluable aid for building distributed systems. In conventional distributed systems, data within an application's address space are usually accessed with absolute addresses or by dereferencing pointers to memory, while remote objects are accessed by sending a message to a remote entity. Local data within an application's address space could likewise be accessed by messaging, but this is usually inefficient and harder to program. Object-oriented operating systems easily support the distributed system model since the encapsulation provided by objects allows method invocation to be implemented as local procedure invocations or remote procedure calls (RPC's)[BN84]. All that is needed is to extend object identities to be valid across nodes in a distributed system. Doing this has the advantage of presenting a single model of computation: sending messages to object. Objects within an application access each other this way and objects access objects in other address spaces, or even in address spaces on different computers the same way.

The implementation of such a scheme is similar to the "stubs" used in traditional RPC systems. A reference to an object that is actually remote will, in reality, reference a local *surrogate* object which acts as the stub for the remote object. When a message is sent to a surrogate, its methods marshall the arguments together and send them to a remote surrogate which in turn unmarshalls them and sends the message to the actual target object. One advantage of such a system is that the inter-surrogate protocol can be specialized for different classes of objects[Sha86]. Generic RPC might work for most cases, but certain surrogates might use a protocol highly optimized for the kinds of methods they are forwarding.

One problem with extending the object-oriented model to encompass distributed systems is the latency that might be introduced while crossing machine and address space boundaries. One of the assumptions of object-oriented programming is that message sends are relatively cheap compared to the operation performed. This is especially true in statically typed object oriented languages since often the message send can be converted directly to a procedure call or to a procedure call after a single indirection. If this assumption is no longer valid, questions about the efficiency of distributed object-oriented systems might arise. However, this cost is already familiar to designers of distributed systems. Remotely accessed entities are usually of sufficient "weight" so as not to matter. Some of this performance loss might be mitigated by using specialized surrogates as well. A specialized surrogate could cache some information about the remote object locally in order to reduce the communication to the remote object and, therefore, increase performance.

8.9 Summary

In summary, constructing object-oriented operating systems is a successful technique, both from a software engineering standpoint and an efficiency standpoint. A set of operating system components, maintained as a class hierarchy, provides both an abstract and a practical classification scheme for existing algorithms as well as revealing many new algorithms in its own right. Using a set of operating system algorithms and data structures organized within an object-oriented framework provides useful guidelines to the developer, and reduces any subsequent efforts to re-target or modify the system. It facilitates both customization and optimization of the system. This thesis demonstrates all of the benefits can be achieved without significant performance sacrifices.

BIBLIOGRAPHY

[A+86] Mike Accetta et al. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX Conference*, pages 93–111, June 1986.

[ADU71] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *Journal of the ACM*, 18(1):80–93, January 1971.

[Agh86] G. Agha. *A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[App88] Apple Computer, Inc., Cupertino, California. *Macintosh System Software User's Guide: Version 6.0*, 1988.

[App89] Apple Computer, Inc., Cupertino, California. *A/UX Programmers Reference*, 1989.

[AR84] William F. Appelbe and A. P. Ravn. Encapsulation Constructs in Systems Programming Languages. *ACM Transactions on Programming Languages and Systems*, 6(2):129–158, April 1984.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, Englewood Cliffs, New Jersey, 1986.

[BB84] M. J. Bach and S. J. Burhoff. Multiprocessor UNIX Operating Systems. *AT&T Bell Laboratories Technical Journal*, 63:1733–1749, October 1984.

[BDMN73] G. Birtwistle, O. Dahl, B. Mhyrtag, and K. Nygaard. *Simula Begin*. Auerbach Press, 1973.

[BMR85] J. S. Banino, G. Morisset, and M. Rozier. Controlling Distributed Processing with Chorus Activity Messages. In *Proc. of the Eighteenth Annual Hawaii International Conference on Systems Science*, pages 257–265, 1985.

[BN84] Andrew Birrell and Bruce Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[Boo86] Grady Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, pages 211–221, February 1986.

[Bri73] P. Brinch Hansen. *Operating System Principles.* Prentice Hall, Englewood Cliffs, New Jersey, 1973.

[Bri85] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1985.

[Bro75] Fredrick P. Brooks Jr. *The Mythical Man-Month.* Addison-Wesley, Reading, MA, 1975.

[BS88] Lubomir Bic and Alan C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[BSD84] 4.2 Berkeley Software Distribution, University of California, Berkeley. *UNIX Programmer's Manual: Reference Guide*, March 1984.

[CD88] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU–CS–8–154, Computer Science Department, Carnegie-Mellon University, June 1988.

[CDG70] P. H. Cress, P. H. Dirksen, and J. W. Graham. *Fortran IV With WATFOR and WATFIV.* Prentice Hall, Englewood Cliffs, New Jersey, 1970.

[CDG+89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 Report (revised). Technical report, Digital Equiptment Co., 1989.

[CH74] R. H. Campbell and A. N. Habermann. The Specification of Process Synchronization by Path Expressions. In G. Goos and J. Hartmanis, editors, *Operating Systems,* *International Symposium, Rocquencourt*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102. Springer-Verlag, New York, April 1974.

[Che84] David R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–42, April 1984.

[Che88] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[CK79] R. H. Campbell and R. B. Kolstad. Practical Applications of Path Expressions to Systems Programming. In *ACM79*, pages 81–87, Detroit, 1979.

[CM78] Roy H. Campbell and T. J. Miller. A Path Pascal Language. Technical Report UIUCDCS–R–78–919, Department of Computer Science, University of Illinois at Urbana-Champaign, April 1978.

[CM87] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 109–110, November 1987.

[Con65] M. Conway. A Multiprocessor System Design. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 139–146, 1965.

[Cra88] Cray Computer Co. *UNICOS Primer*, 1988.

[CRJ87] Roy Campbell, Vincent Russo, and Gary Johnston. The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, 1987. Also Technical Report No. UIUCDCS–R–87–1388, Department of Computer Science, University of Illinois at Urbana-Champaign.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Dei84a] Harvey M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley Publishing Company, Reading, Massachusetts, Reading, Massachusetts, 1984.

[Dei84b] Harvey M. Deitel. Case Study: VAX. In *Introduction to Operating Systems*, pages 505–533. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[Den68] Peter J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[DG87] L. G. DeMichel and R. P. Gabriel. The Common Lisp Object System. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '87)*, 1987.

[Dij68] Edsger W. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, pages 341–346, May 1968.

[Doe87] Thomas W. Doeppner. Threads: A System for the Support of Concurrent Programming. Technical Report CS-87-11, Department of Computer Science Brown University, Providence, RI 02912, June 1987.

[DT88] Scott Danforth and Chris Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[Enc86] Encore Computer Corporation, Marlboro, Massachusetts. *UMAX 4.2 Programmer's Reference Manual*, 1986.

[Enc88] Encore Computer Corporation, Marlboro, Massachusetts. *Encore Parallel Threads Manual*, 1988.

[Enc89] Encore Computer Corporation, Marlboro, Massachusetts. *Multimax Technical Summary*, 1989.

[Fie89] S.P. Fiedler. Object-Oriented Unit Testing. *HP Journal*, 36(4), April 1989.

[GMS87] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual Memory Architecture in SunOS. In *Proceedings of the Summer 1987 USENIX Conference*, pages 81–84, 1987.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[GR83] Adele Goldberg and David Robison. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Gra89] Justin O. Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1989.

[Helng] Bjorn A. Helgaas. Porting the Choices Object-Oriented Operating System. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1990 (fotrhcoming).

[Hew87] Hewlett Packard, Cupertino, California. *Precision Architecture and Instruction Reference Manual*, 1987.

[HFC76] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, pages 266–272, May 1976.

[HO87] Daniel C. Halbert and Patrick D. O'Brien. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software*, pages 71–79, September 1987.

[Hoa74] C. A. R. Hoare. Monitors, An Operating System Structuring Concept. *Communications of the ACM*, October 1974.

[IBM86] International Business Machines Corporation. *IBM RT PC Hardware Technical Reference*, September 1986.

[IBM88a] International Business Machines. *IBM AIX Technical Reference Volume 1 for PS/2*, 1988.

[IBM88b] International Business Machines, Poughkeepsie, New York. *IBM Enterprise Systems Architecture/370: Principles of Operation*, 1988.

[Int81] Intel Corporation, Santa Clara, California. *System 432/600 System Reference Manual*, 1981.

[Int87] Intel Corporation, Santa Clara, California. *80386 System Software Writer's Guide*, 1987.

[JAvdG86] M. D. Janssens, J. K. Annot, and A. J. van de Goor. Adapting UNIX for a Multiprocessor Environment. *Communications of the ACM*, 29(9):895–901, September 1986.

[JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *The Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 18–26, 1988.

[JLHB87] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 62–74, November 1987.

[Joh91] Gary M. Johnston. *Data and Process Migration in Distributed Virtual Memory Systems.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, (forthcoming) 1991.

[JW89] Kathleen Jensen and Niklaus Wirth. *PASCAL: User Manual and Report.* Springer-Verlag, New York, 1989.

[KL87] R. Kain and L. Landwehr. On Access Checking in Capability-Based Systems. In *IEEE Transactions on Software Engineering*, pages 202–207, February 1987.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

[KR78] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[LAB+81] Barbara H. Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, and Alan Snyder. *CLU Reference Manual, Lecture Notes in Computer Science Vol. 114.* Springer-Verlag, New York, 1981.

[Ley88] Douglas E. Leyens. A Choices Implementation of the Universal Scheduling System. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1988.

[Lie86] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 214–223, 1986.

[Lis72] Barbara H. Liskov. The Design of the Venus Operating System. *Communications of the ACM*, 15:144–149, March 1972.

[LL82] Henry Levy and Peter H. Lipman. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer*, pages 35–41, March 1982.

[LLA+81] E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler, and S. Vestal. The Architecture of the Eden System. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 148–159, 1981.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[LR79] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 43–44, December 1979.

[LS87] L. K. Loucks and C. H. Sauer. Advanced Interactive Executive (AIX) Operating System Overview. *IBM Systems Journal*, 26(4):326–346, 1987.

[Mad91] Peter W. Madany. *An Object-Oriented Approach towards a General Model of File Systems.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, (forthcoming) 1991.

[Mar88] J. L. Martins. The Design of the CHORUS Inter-process Communication Facility. *IFIP*, pages 61–72, 1988.

[MB80] G. Myers and B. Buckingham. A Hardware Implementation of Capability-Based Addressing. In *ACM Operating Systems Review*, pages 13–25, October 1980.

[McK79] Jean L. McKechnic, editor. *Webster's New Twentieth Century Dictionary.* Simon & Schuester, 1979.

[McM81] Lee E. McMahon. An Experimental Software Organization for a Laboratory Data Switch. In *Proceedings of the 1981 IEEE International Conference on Communications*, volume 2, pages 24.4.1–24.4.4, 1981.

[MCRL88] Peter W. Madany, Roy Campbell, Vincent Russo, and Douglas E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. Department of Computer Science, University of Illinois at Urbana-Champaign, 1988. To be presented at ECOOP '89.

[MD74] S. E. Madnick and J. J. Donovan. Virtual Machine/370 (VM/370). In *Operating Systems*, pages 549–563. McGraw-Hill, New York, 1974.

[Mey86] Bertrand Meyer. Genericity Versus Inheritance. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 391–405, 1986.

[Mey87] Bertrand Meyer. Reusability: The Case for Object-Oriented Design. *IEEE Software*, pages 50–64, March 1987.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[MLC+87] J. M. Mellor-Crummey, T. J. LeBlanc, L. A. Crowl, N. M. Gafter, and P. C. Dibble. Elmwood – An Object-Oriented Multiprocessor Operating System. Technical Report TR–226, Computer Science Department, University of Rochester, September 1987.

[MLRC88] Peter Madany, Douglas Leyens, Vincent Russo, and Roy Campbell. A C++Class Hierarchy for Building UNIX-Like File Systems. In *Proceedings of the USENIX C++ Conference*, October 1988. Also Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign.

[Mot88] Motorola, Inc. *MC88200 Cache/Memory Management Unit User's Manual*, 1988.

[Mot89] Motorola, Inc. *MC68030 Enhanced 32-Bit Microprocessor User's Manual*, 1989.

[MS87] Paul R. McJones and Garret F. Swart. Evolving the UNIX System Interface to Support Multithreaded Programs. Technical report, Digital Equiptment Corporation, September 1987.

[Mul87] S. J. Mullender, editor. *The Amoeba Distributed Operating System: Selected Papers 1984–1987*. CWI Tract 41, Center for Mathematics and Computer Science, 1987.

[Nat86] National Semiconductor Corporation, Santa Clara, California. *Series 32000 Databook*, 1986.

[Nau63] P. Nauer. Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 6(1):1–17, January 1963.

[NEX] *The NeXT Computer Applications Programing Interface.*

[Nor85] Peter Norton. *The Peter Norton Programmer's Guide to the IBM PC.* Microsoft Press, 1985.

[NW77] R. M. Needham and R. D. H. Walker. The Cambridge CAP Computer and its protection system. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 1–10, 1977.

[OCD+88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–35, February 1988.

[Par72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems Into Modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.

[PG90] D.E. Perry and G.E.Kaiser. Adequate Testing and Object-Oriented Programming. *The Journal of Object-Oriented Programming*, 2(5):13–19, Jan/Feb 1990.

[Pre82] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New York, 1982.

[PS85] James L. Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1985.

[R+87] Richard Rashid et al. Machine–Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[RAN88] M. Rozier, V. Abrossimov, and W Neuhauser. CHORUS-V3 Kernel Specification and Interface, Draft. Technical Report CS/TN-87-25.10, CHORUS Systems, February 1988.

[Ras86] Richard F. Rashid. From RIG to Accent to Mach: The Evolution of a Network Operating System. Technical report, Computer Science Department, Carnegie-Mellon University, 1986.

[RK88] Vincent F. Russo and Simon M. Kaplan. A C++ Interpreter for Scheme. In *Proceedings of the USENIX C++ Conference*, Denver, Co, October 1988.

[Ros87] John R. Rose. Fast Dispatch Mechanisms for Stock Hardware. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 27–35, 1987.

[RR81] Richard F. Rashid and George G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the ACM Symposium on Operating System Principles*, December 1981.

[RT75] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *AT&T Bell Laboratories Technical Journal*, 57(6):1905, 1930, 1975.

[SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 9–16, 1986.

[Seq85a] Sequent Computer Systems. *Balance 8000 Guide to Parallel Programming*, July 1985.

[Seq85b] Sequent Computer Systems. *Balance 8000 System Technical Summary*, 1985.

[Set89] Ravi Sethi. *Programming Languages: Concepts and Constructs.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[Sha86] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th. International Conference on Distributed Computer Systems*, May 1986.

[Sha88] Marc Shapiro. The Design of a Distributed Object-Oriented Operating System for Office Applications. In *Proc. Esprit Technical Week 1988*, Brussels (Belgium), November 1988.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 38–45, 1986.

[Spa86] Eugene H. Spafford. Kernel Structures for a Distributed Operating System. Technical Report GIT–ICS–86/16, School of Information and Computer Science, Georgia Institute of Technology, 1986.

[Ste87] Lynn A. Stein. Delegation is Inheritance. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 138–146, 1987.

[Str86] Bjarne Stroustrup. *The C++Programming Language.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Str89] Bjarne Stroustrup, 1989. Personal Communication.

[SUN88] SUN Microsystems. *SUN OS Reference Manual*, May 1988.

[SVI85] AT&T Customer Information Center, Indianapolis, IN. *System V Interface Definition*, 1985.

[SZBH86] Daniel Swinehart, Polle Zellweger, Richard Bach, and Robert Hagmann. A Structural View of the CEDAR Programming Environment. *ACM Transactions on Programming Languages and Systems*, 8:419–490, October 1986.

[T+87] Avadis Tevanian et al. Mach Threads and the UNIX Kernel: The Battle for Control. In *Proceedings of the USENIX Conference*, pages 185–197, June 1987.

[Tie90] Michael Tiemann. *User's Guide to GNU C++*. The Free Software Foundation, Inc, March 1990.

[TM81] A. S. Tanenbaum and S. J. Mullender. An Overview of the Amoeba Distributed Operating System. *ACM Operating Systems Review*, pages 51–56, July 1981.

[TR87] A. Tevanian and R. F. Rashid. MACH: A Basis for Future UNIX Development. Technical Report CMU-CS-87-139, Computer Science Department, Carnegie-Mellon University, 1987.

[TvR85] Andrew S. Tanenbaum and Robbert van Renesse. Distributed Operating System. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[Uni81] United States Department of Defense, editor. *The Programming Language Ada: Reference Manual.* Springer-Verlag, New York, 1981.

[W+74] William A. Wulf et al. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, pages 337–345, June 1974.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 168–182, 1987.

[Y+87] Michael Young et al. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 63–76, 1987.

VITA

Vincent Frank Russo was born on the 8th of August, 1962 in Pittsburgh, Pennsylvania. Shortly thereafter, his family moved to Dayton, Ohio. He attended grade school and high school in Dayton. After graduating from Carroll High School in Dayton in 1980, Vincent enrolled at the University of Dayton. In 1984, he received a Bachlor of Science degree in Computer Science/Physics from the University of Dayton.

In August of 1984, Vincent begun graduate work at the University of Illinois at Champaign-Urbana in the Department of Computer Science. He received his Masters Degree in 1986 and in 1990 finshed work on his Ph.D. The Ph.D. degree was awarded in January 1991. His thesis was entitled "An Object-Oriented Operating System".

Vincent is currently an Assistant Professor at Purdue University in West Lafayette, Indiana.