Report No. UIUCDCS–R–88–1415 UILU–ENG–88–1720

Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques

by

Vincent Russo, Gary Johnston, Roy Campbell

July 26, 1991

REPORT NO. UIUCDCS–R–88–1415

Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques

by
Vincent Russo, Gary Johnston, Roy Campbell

July 26, 1991

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. University Ave., Urbana, IL 61801–2987

This work was supported in part by NASA under grant number NSG1471 and by AT&T METRONET.

Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques[1]

Vincent Russo, Gary Johnston, Roy Campbell

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. University Ave., Urbana, IL 61801–2987

Abstract

The programming of the interrupt handling mechanisms, process switching primitives, scheduling mechanisms, and synchronization primitives of an operating system for a multiprocessor require both efficient code in order to support the needs of high-performance or real-time applications and careful organization to facilitate maintenance. Although many advantages have been claimed for object-oriented class hierarchical languages and their corresponding design methodologies, the application of these techniques to the design of the primitives within an operating system has not been widely demonstrated.

To investigate the role of class hierarchical design in systems programming, the authors have constructed the *Choices* multiprocessor operating system architecture using the C ++ programming language. During the implementation, it was found that many operating system design concerns can be represented advantageously using a class hierarchical approach, including: the separation of mechanism and policy; the organization of an operating system into layers, each of which represents an abstract machine; and the notions of process and exception management. In this paper, we discuss an implementation of the low-level primitives of this system and outline the strategy by which we developed our solution.

1 Introduction

The *Choices*[2] [1] [2] operating system architecture is organized as a class hierarchical solution to the design problems of operating systems. Applications for which Choices has been designed include numerical computations, embedded flight control and ground-based monitoring systems, and controllers for high-speed circuit and packet switched networks. The research was motivated by the difficulties of building multiprocessor operating systems for specialized high-performance, real-time applications on large collections of heterogeneous shared memory and networked multiprocessors. For example, current operating systems

[1]This work was supported in part by NASA under grant number NSG1471 and by AT&T METRONET.

[2]*Class Hierarchical Open Interface for Custom Embedded Systems.*

cannot be easily extended to manage dynamic load balancing, reconfiguration, process migration, and heterogeneous CPU resource management. Similarly, the conventional operating system provides applications with a "kernel" that offers a predefined selection of system services; this kernel cannot be easily extended to provide specialized services for particular concurrent applications on particular parallel hardware. Choices uses objects and class hierarchies to organize and facilitate solutions to both example problems. An operating system implemented with the Choices architecture currently runs on the Encore Multimax^{TM}.

Choices provides a hierarchy of classes from which the operating system designer may choose or specialize components in order to build a custom system, an *instance* of a Choices operating system. Classes within the hierarchy may be customized for specific hardware environments or the needs of a particular dedicated application. Thus, the Choices architecture supports the concept of a *family* operating systems. The hierarchy contains *abstract* primitive hardware-independent classes and system service primitive classes that respond to application requests. Concrete subclasses of these classes define objects in particular Choices implementations. Thus, not only can an operating system be constructed in order to effectively exploit a particular hardware architecture, but it can also be designed to provide specific applications with specialized support, avoiding any overheads associated with features designed into a "general purpose" operating system that the application neither requires nor uses.

A major motivation for this research was to determine if the class hierarchical object-oriented approach could be used for the design and implementation of complete operating systems. The C ++ programming language [3] was used exclusively in the implementation of Choices. Therefore, this research is also an investigation of the appropriateness of implementing complete operating systems in a language which supports object-oriented programming and class hierarchies, as opposed to either adding features to an existing language or using a more complex programming language that already has system programming features (e.g., Mesa or Ada^{TM}).

Many operating systems are structured using a layered approach similar to that of the THE system [4]. Each layer refines and enhances the services and functionality of the layer upon which it is built. In general, however, layers do not shared a common internal organization. In Choices we have built a class hierarchical implementation of these layers using an object-oriented methodology. We believe that this imposes a discipline within (and between) layers which aids debugging, facilitates extensive code reuse, and eases modification.

Choices design goals include support for:

- high-performance real-time and multiprocessing applications,
- custom systems development, including customization for particular hardware architectures and applications, allowing for the implementation of the "smallest" operating system required for a given application,[3]

[3]General purpose operating systems often employ delayed bindings to provide flexibility. Examples include communication schemes, file systems, access to other system services, and additional kernel code to handle different applications requirements and different architectures and configurations. Choices can allow several

- operating systems research, facilitated by easy customization and module substitution,
- efficient variable-grained parallelism,
- exploitation of virtual memory techniques for efficient interprocess communication via shared memory and shared object access,
- error recovery in parallel systems, and
- real-time interrupt handling, including both global system interrupts and processor specific interrupts.

Some of the issues addressed in the design of Choices include:

- the problems associated with a parallel operating system kernel which allows multiple processors to execute concurrently within the kernel,
- the avoidance of the inclusion of specific communication schemes into the lowest levels of the operating system which would restrict the possibility of specializing the kernel at higher levels in order to exploit the communication patterns of the application or the communication mechanisms supported by the underlying hardware,[4]
- the reduction of process context switching overheads,
- the avoidance of excessive cache and cache flushing overheads in cached-based multiprocessors,
- the avoidance of designing another distributed operating system, concentrating instead on the design of a *parallel* operating system.[5]

In this paper, we discuss the classes within Choices that support process dispatching and execution, exception handling, process context switching, scheduling, and synchronization. The classes provide the primitives from which higher-level layers of the operating system or application-specific systems are constructed.

applications to coexist within the same computing system, each running its own customized Choices operating system. Any communication required between the applications is supported by common Choices primitives and operations on shared persistent objects.

[4]Message-oriented kernels like the V Kernel [5], Accent [6], Amoeba [7], and MICROS [8] build specific communication schemes into the lowest levels of the kernel. For example, some systems implement a few ways of providing "virtual" messages like "fetch on access." However, these systems are not easy to adapt to support other approaches such as "send process on read" or "remote procedure call on execute."

[5]Many distributed multiprocessor Unix systems (Unix United [9], Locus [10], Mach [11], RFS [12], NFS [13], Encore Multimax Unix (UMAX^{TM}) [14], and Sequent Balance^{TM} 8000 Unix) [15] still impose Unix limitations on the parallelism and performance of applications.

1.1 Related Research

The organization of an operating system is a difficult task. Many different approaches to structuring operating systems have evolved in response to advances in hardware technology and improvements in software engineering techniques. Approaches to the structuring of operating systems [16] include a monolithic, kernel, process hierarchy, functional hierarchy, and object-oriented capability structure.

Many simple microprocessor and some early batch operating systems are structured using the *monolithic* approach: as a large program that invokes user programs as "subroutines." These applications "return" as a result of an interrupt, a request for an operating system service in the form of a system call, or the termination of the program.

Many current operating systems like UNIX^{TM} are organized using a *kernel* approach in which application programs execute on an "abstract CPU" that hides details of the real CPU such as its concurrent use by other applications and its I/O hardware interface. Kernel's often introduce the notion of a concurrent process to support the abstraction that permits the sharing of a CPU among multiple applications. Typically, the software of the kernel is minimal; the majority of operating system software executes outside of the kernel exploiting the abstractions provided by the kernel.

Even though the software in a kernel should, by definition, be kept to a minimum, it often is large and can benefit from structuring. Layered systems, most notably THE [4] and Venus [17], structure an operating system as a series of layers, or levels. Each layer is built as a collection of concurrent processes and software modules that exploit the abstractions and enhancements provided by the previous layer. By separating the various solutions required to build an operating system, layering offers improved maintainability and portability.

However, it is not always easy to organize the processes into layers. Instead, a system may be organized as a *functional hierarchy* [18]. Concurrent processes within the operating system may access functions at different levels within the hierarchy.

The *object-oriented capability* approach to structuring an operating system considers it as a collection of objects that includes applications and system services. The interaction of the objects is organized as a network of capabilities. One example of such an object-oriented operating system is the Intel iAPX 432 system [19].

To summarize, operating systems have been organized by hiding machine dependencies, by providing a machine independent kernel that hides the details of sharing of a CPU among concurrent processes, by structuring system support into layers of processes, into a functional hierarchy, or by representing the mechanisms, services, and abstractions of an operating system as objects. Each approach has merits in terms of simplicity or coping with complexity. Each approach may sometimes lead to implementations that could have been derived by a different approach. In the next section, we shall distinguish these previous approaches with our own class hierarchical, object-oriented approach.

1.2 Structuring an Operating System Using Class Hierarchies

In Choices, we exploit class hierarchies and inheritance to create operating system classes that are customizable for particular hardware and applications. We believe that the careful use of class hierarchies to build an operating system results in a major improvement in the organization and design of the system. Not only do classes permit the expression of the standard operating system design techniques, but they also represent and organize major design concerns that could not be expressed easily within existing design approaches. In this section, we describe the design methodology that is permitted by class hierarchies.

Both the monolithic and kernel approaches to organizing operating systems are suitable for small operating systems but do not scale up well because of the lack of any internal structure. The operating system kernel approach has the desirable property of separating an operating system into a set of cooperating processes communicating through the kernel. The cooperating process model is very important because it structures asynchronism in an intuitive manner. Layering the contents of a kernel is a practical solution to structuring large operating systems that has been applied to many of today's major operating systems [11] [20] [21] [22].

The major difficulty with building layered operating system kernels is determining the layer in which processes or functions should be implemented and structuring the internals of a particular layer. 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 designing the layers of an operating system. The object-oriented capability approach helps to reduce the problems caused by such cyclic dependencies since capabilities can be used to represent arbitrary networks. However, such arbitrary networks lack structure (unless the objects are somehow organized into layers.)

The use of *class hierarchies* to structure operating systems is presented here as an orthogonal issue to the layering of a system. The hierarchies are used as a means to structure the internals of a layer or kernel. Within Choices, object-oriented programming and inheritance are used to build a kernel from layers that are *collections of objects*; some objects may provide system functions and others may define or contain concurrent processes. A process defined within one layer may access objects in other layers. The class inheritance mechanism and upward type coercions allow references to these objects in a structured manner. Class hierarchies and object-oriented programming also have the software engineering advantages that include code reuse and modular programming.

The class hierarchical approach encourages three methods of *code reuse*. The first method is gained from the use of a language with class and inheritance provided that care is used in the construction of the subclasses. By building class hierarchies in which methods are specialized incrementally through subclassing, much code can be shared between classes.

The second form of code reuse is particular to layered operating systems. Common system functions and utilities are often repeated within different layers. For example, queues, lists, hash tables, mutual exclusion primitives, semaphores, processes, and schedulers may be used in several layers including the application layer. Classes allow the reuse of the definition and implementation of abstract algorithms and data structures throughout the layers of a system. This form of code reuse is enhanced by defining generic classes to represent the abstract algorithms and data structures of useful operating system concepts, subclassing these classes to apply them to the specific needs of an operating system, and then instantiating these classes to produce objects within a specific layer.

The third method of reuse is a refinement of the first method applied to a family of operating systems. An abstraction of some operating system component within the family may be defined as a class with many different subclasses each implementing a "version" of that class for a particular hardware or a particular application's requirements. The different versions of the class can all share a large portion of the implementation through the parent class if they are similar. This form of reuse simplifies the customization of the family of operating systems for a target machine and application.

The class hierarchical approach also supports *customization*. Customization and modification of a family of systems is guided and aided by subclassing and by the structure induced by the class hierarchy. The class hierarchy provides the systems designer with a conceptual view of how the components of an operating system function. It classifies components of a system with respect to their function; by learning the function of a parent class, the possible function of a subclass can be inferred [23] [24]. In the class hierarchy for Choices, only the top few classes need to be mastered to achieve a good overall view of the system. In addition, subclassing permits the behavior of a specific part of a Choices operating system to be modified without changing the rest of the system.

All *machine dependencies*, operating system *mechanisms* (e.g., page table management), operating system *policies* (e.g., schedulers) and *design decisions* are *encapsulated* within classes in Choices. Design decision are implemented as a potential set of subclasses. An *abstract* class specifies a general behavior and protocol (its methods) that may be used on instances of the class and its concrete subclasses. A *concrete* class refines the implementation of an abstract class. Abstract classes are used in our system design to specify operating system abstractions. Concrete subclasses are used to specify particular versions, policies or mechanisms that implement the abstraction. Wherever possible, the class hierarchy is constructed so that similar sub-hierarchies can be specialized from a common ancestor hierarchy. (Not only do classes support reuse, but the structure induced by a hierarchy may also be reused, including code reuse of the classes in the hierarchy.) Overall, we have constrained the "fan out" to be small (2–7) to encourage code reuse.

In later parts of this paper we will discuss the use of this methodology to design process management and exception handling within Choices.

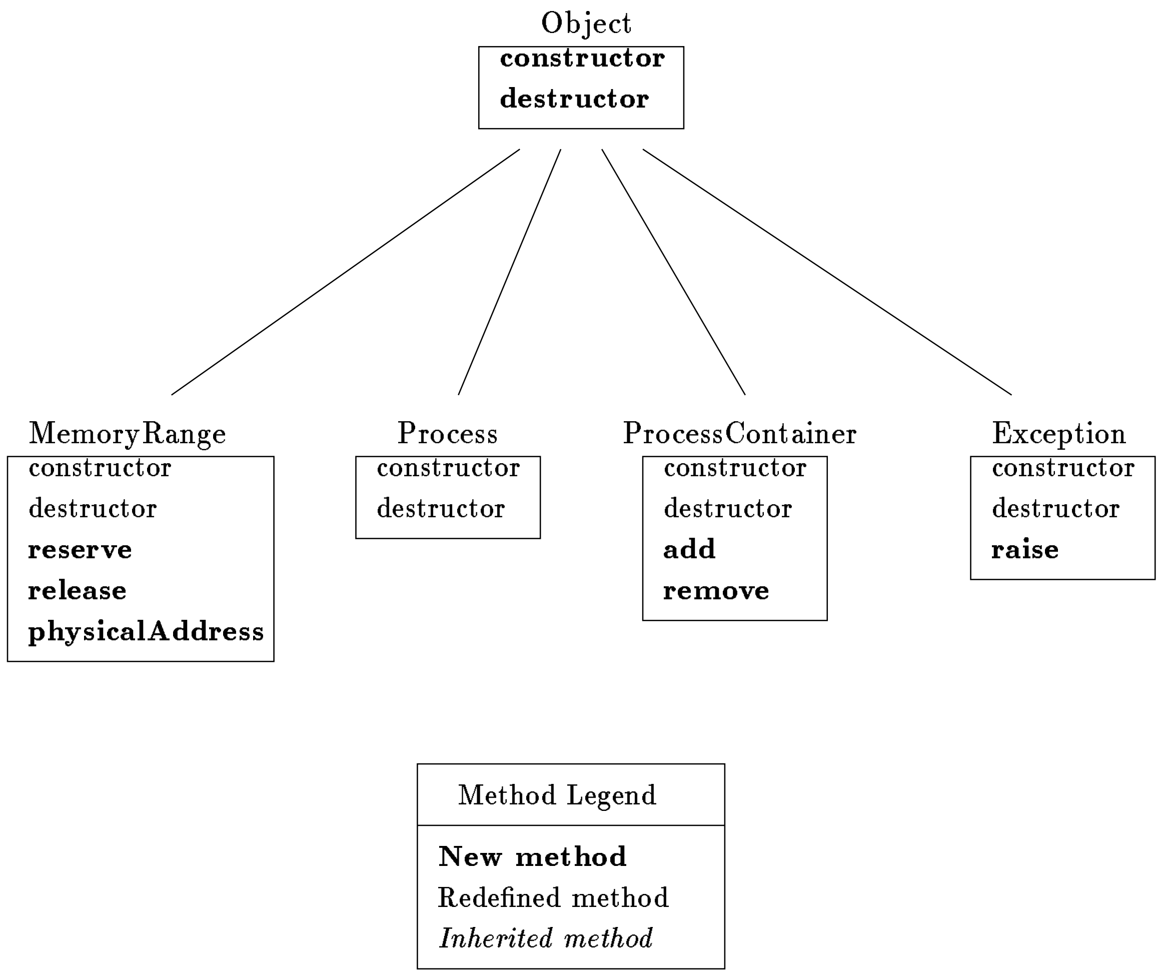

Figure 1: Major Base Classes in the Choices Hierarchy.

2 The Choices Class Hierarchy

Before going into the details of the process and exception classes, a brief overview of the Choices class hierarchy is appropriate. Some of the major classes in the first level of the Choices class hierarchy are shown in Figure 1. Each subclass redefines and adds methods defined for class *Object*. Class *MemoryRange* provides the base for storage management in a Choices operating system. Instances of class *Process* are the basic units of execution in a Choices system. A Process is represented by the information necessary to execute it. This is usually the processor state information (i.e., machine registers) and information about the virtual memory in which it expects to execute. Processes are scheduled and executed within a Choices system by being added to and removed from *ProcessContainers*. Class ProcessContainer is specialized to provide for Process execution and scheduling. Class *Exception* provides the basis for exception handling, including traps and interrupts. The raising of an Exception usually causes Exception-specific movement of Processes between

ProcessContainers.

To aid in portability, objects in this design are grouped into layers. The *Germ* is the lowest layer. It is a set of objects that encapsulates the major hardware dependencies and provides an "idealized" hardware architecture to the rest of the layers in the system.[6] The Germ provides the *mechanisms* for managing and maintaining the physical resources of the computer. Objects defined for use in the Germ for specific architectures are instances of subclasses of generic classes that define interfaces to the hardware memory management, the hardware exception, and physical processor mechanisms. Intermediate layers in the system include memory management, exception management, scheduling and naming. The *Kernel* is the highest layer in the system. It defines the interface provided by the system to applications. A complete operating system consists of Germ objects appropriate for the particular hardware of the system, objects belonging to the middle layers, and Kernel objects appropriate for the applications that are supported by the operating system. Individual applications that run on top of the new system can further augment the class hierarchy with their own classes.

All operating system components, including support for parallelism and synchronization, are implemented using C ++. The language is efficient and portable. It implements object-oriented programming and class hierarchy semantics with minimal runtime overhead and thus is ideal for operating system programming. There are, of course, a few small assembly language routines whose functionality could not be implemented directly in C ++. It is easy to interface C ++ to assembler in order to achieve things impossible in the language itself (for example, loading stack pointers and memory management unit registers). However, even these assembly routines are mostly machine-dependent implementations of methods of low-level classes, thus preserving the object-oriented implementation even to the lowest levels of the class hierarchy. C ++ was chosen because it supports class hierarchical object-oriented design while imposing negligible run-time overhead.

In the following sections, we will describe in more detail some of the classes which implement the functions of interrupt handling, process dispatching, context switching, scheduling, and synchronization.

3 Processes

Choices supports the concept of a computation that is composed of a potentially large number of lightweight, independent parallel processes similar to those described in [5] [11] and [25]. An application may use multiple communicating processes to achieve concurrency and parallelism. A single one of these processes is represented by an instance of the Choices *Process* class. Each instance of the Process class represents an independent flow of control that can share memory through the memory management mechanisms described below. In order for such an abstraction to be useful and to allow efficient process migration, the

[6]The "virtual" machine provided to the higher layers of the system by the Germ objects is not a virtual copy of the actual hardware as in the IBM VM/370 [22] operating system, but rather an idealized architecture.

amount of information kept on a per-process basis and the context switching effort between two processes is minimized.

3.1 Memory Management

A complete discussion of the Choices memory management system is beyond the scope of this paper.[7] A general description of the way memory is managed is, however, necessary in order to discuss process management and context switching in Choices.

Memory management in Choices is implemented by a hierarchy of classes with the abstract *MemoryRange* class as root. The classes in this hierarchy support virtual memory, the sharing of memory, and memory protection. An instance of a MemoryRange class represents a contiguous range of memory addresses, as the name implies. MemoryRange is subclassed to represent the different kinds of memory in a system. The most important example is class *Space*. A Space is used to represent a range of virtual memory addressable by a Process. The class *SpaceList* is provided for the aggregation of Spaces. It is subclassed into the *Domain* class which represents the complete view of virtual memory that may be accessed by a Process. The list of Spaces in the Domain of a process is consulted during memory management decisions. Sharing of memory between processes can be accomplished either through shared Domains or through different Domains which list common Spaces.

3.2 Process Implementation and Context Switching

Each Choices Process references a Domain that specifies its virtual memory. Usually, the executable code, initialized data, uninitialized data, and stack are represented as Spaces within this Domain. The constructor for a process is parameterized by an initial Domain, initial processor state (for example, program counter or stack pointer), and arguments to the process. Methods for Processes alter their Domains, manipulate scheduling parameters, and handle preemption and dispatching.

The state of a process is recorded by storing the processor state and register contents within a Process object. A small supervisory stack is maintained by each Process object in order to handle preemption. The **dispatch** method of the Process class is used by the process switching code in Choices to reload a CPU's registers with copies that are stored within the Process object. Context switching overhead is lowest between Processes which execute within a common Domain since if the Domain of the Process being dispatched matches the Domain in which the processor is currently executing, no memory context switching is necessary.

Interrupt and real-time processing require the ability to switch between processes with minimum context switching overhead. Since an executing process accesses a stack, code, and data represented by the various Spaces contained within its Domain, fast context switching can be achieved by locking the memory of a Space as resident. Locking memory to be resident within a Space causes the corresponding virtual addresses to be validated and the associated real memory to be locked as resident in physical memory. In addition, a Space

[7] A more complete description is contained in [1].

may be locked as addressable by all Processes. A context switch to a process that addresses only resident pages in resident virtual memory incurs only register loading overhead.[8]

Locking can optimize the performance of interrupt handlers and real-time processes as desired. Such processes may still be protected from other applications by running the processes in the privileged state of the processor and setting the memory protection of the globally shared Spaces to exclude access in non-privileged mode. Thus, even though a Space may be locked as addressable by all Domains, it can remain protected from unprivileged processes. The Kernel and Germ memory of a Choices system are implemented as sets of such Spaces.

3.3 ProcessContainers

Primitives for scheduling, blocking and dispatching processes in Choices are built by using instances of the *ProcessContainer* class and its subclasses. A ProcessContainer, as the name implies, is a container of Processes. Scheduling and dispatching algorithms in Choices involve transferring Processes between ProcessContainers. Figure 2 shows some of the classes in Choices which implement Processes and ProcessContainers.

Subclasses of ProcessContainer impose queueing disciplines on the Processes that they contain. Some subclasses are defined to only contain a single Process. The ProcessContainer class is abstract and defines the operations **add** (for inserting Processes into the container), **remove** (for removing Processes from the container), and **isEmpty** (for testing whether the container is empty or not). Subclasses can redefine these methods, for example, to add and remove Processes in FIFO, LIFO, or priority order.

ProcessContainers represent an operating system abstraction that may be refined to implement queues of processes (e.g., "run queues" and "ready queues"), and may be used to store processes that await an event or are blocked on a Semaphore operation. Scheduling in Choices is discussed in more detail in a later section.

3.4 Exception Handling

Low level exceptions are introduced in Choices by the abstract *Exception* class and refined by its subclasses. The Exception class defines the method **raise** to intercept the exception condition. The Exception class has subclasses *HardwareException* and *SoftwareException*. Figure 3 shows the base classes of Choices which implement exception handling. The raise method for a HardwareException is called directly as a result of a hardware trap or interrupt. HardwareExceptions are used to encapsulate the hardware exception mechanism of the underlying architecture. The raise method of a SoftwareException is called voluntarily by an executing process. Exceptions are the primary objects in Choices whose methods cause Processes to be moved between ProcessContainers. Raising an Exception is the only way for a process to suspend its own execution.

[8]Plus MMU cache flushing overhead, if the old and new Domains differ.

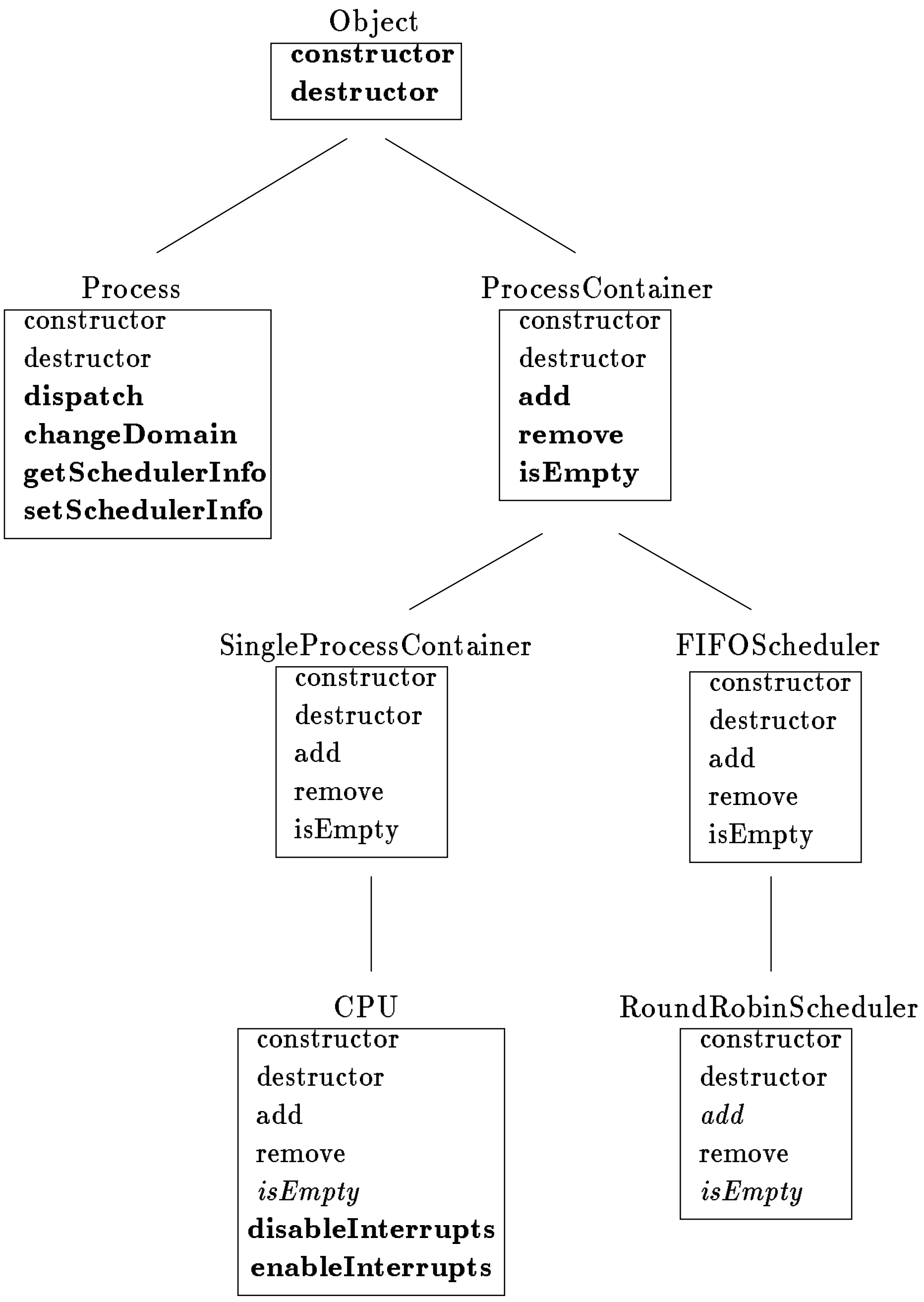

Figure 2: Process and ProcessContainer Base Classes.

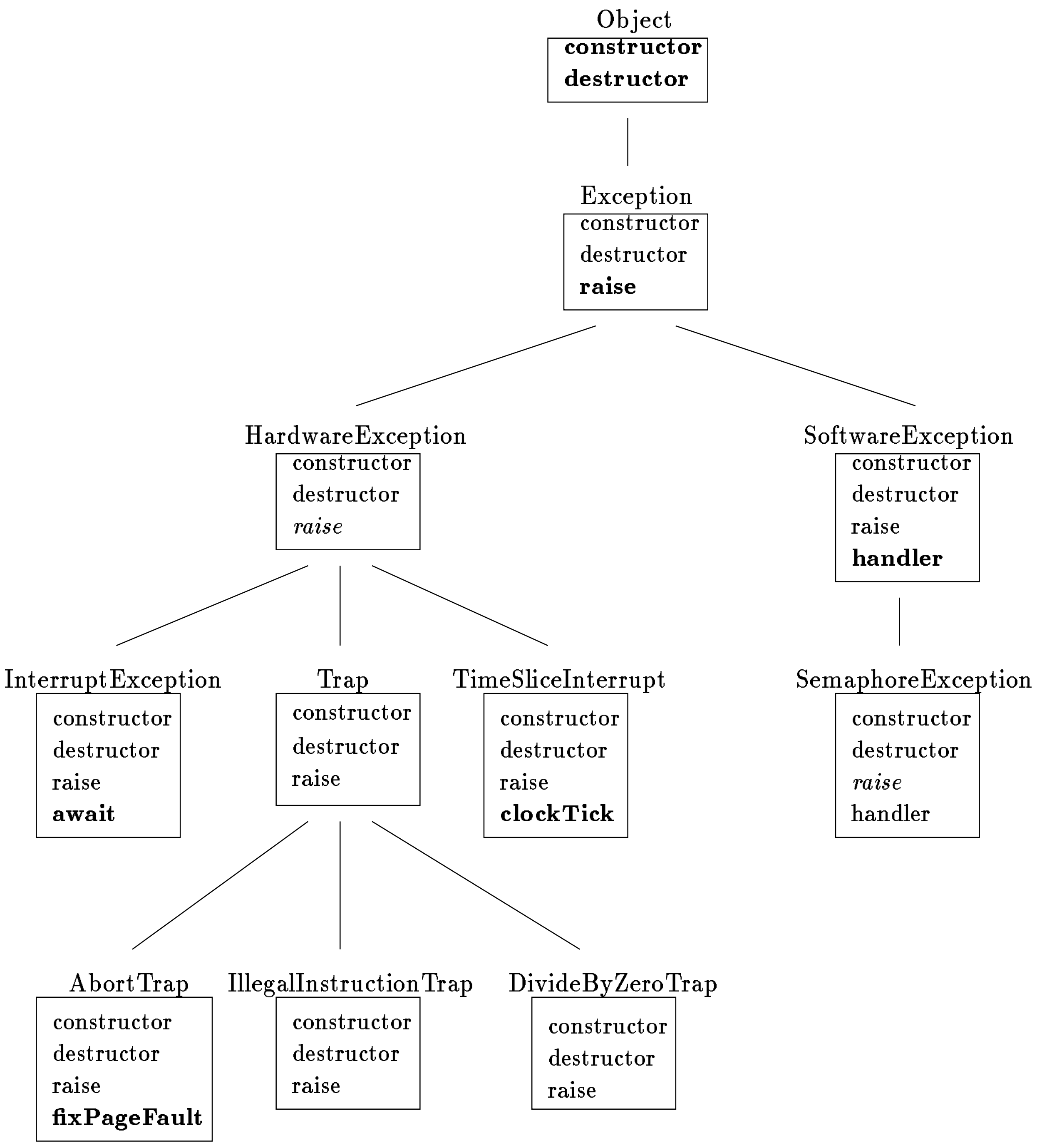

Figure 3: Exception Handling Base Classes.

3.5 The CPU ProcessContainer Subclass

A special subclass of ProcessContainer, *CPU*, represents an actual physical processor. Multiprocessing fits naturally into this model since a system can consist of more than one instance of the CPU class, each corresponding to an actual processor. The CPU class redefines the add method to dispatch and execute a Process on the processor it represents. The remove method of the CPU class is used by Exceptions to implement CPU preemption. It returns the Process for which the CPU has most recently saved a context. To effect the transfer of the CPU from one process to another, the context of the executing process must first be saved. This is accomplished by raising an exception. When an Exception is raised, the context of the currently executing process is saved on its per-process supervisory stack. These supervisory stacks exist one per Process object and need only be large enough to hold a single process context. Once the state is saved on this stack, the raise method invokes the Exception handler. The Exception handler is executed independent of the state of any particular process by switching to a per-*processor* supervisory stack.[9] Exception handlers usually execute code which removes the Process from the CPU, stores it in a ProcessContainer, and adds another Process to the CPU. Not all classes of Exceptions behave this way; some define the raise method to save only a minimum amount of context because, after processing the exception, it will immediately resume the process. The CPU class defines methods to install Exception objects as the handlers of hardware Exceptions.

Exceptions can be raised synchronously by a Process voluntarily invoking an Exception object's raise method, or asynchronously by the raise method of an Exception being "invoked" via an interface to the hardware exception mechanism. Hardware exceptions are discussed in more detail in the following sections. Combined uses of ProcessContainers and Exceptions, most notably in the implementation of semaphores, are discussed later.

3.6 Hardware Exceptions

The HardwareException class has several major subclasses. The *Trap* class provides a mechanism for handling traps that a process may generate as a direct result of its execution. This includes machine traps (for example, divide-by-zero and illegal instruction), virtual memory access and protection errors (for example, page faults of various types), and explicit program traps (for example, a "system call" via an "SVC" instruction). The Trap handler services the exception condition within the context of the faulting Process and then resumes, or terminates that Process.

Interrupts occur asynchronously and, in general, have nothing to do with the currently executing process. The *InterruptException* subclass of HardwareException defines a new method, **await**. The await method is invoked by a Process to block its execution until the

[9]This stack is created by the CPU class during its initialization. If the exception handling code was executed within the context of any particular Process (i.e. if a Process attempted to remove itself from the CPU and added itself to another ProcessContainer), a race would exist where the Process could be removed from the second ProcessContainer and added to another CPU thus resulting in the Process being executed by two processors at the same time.

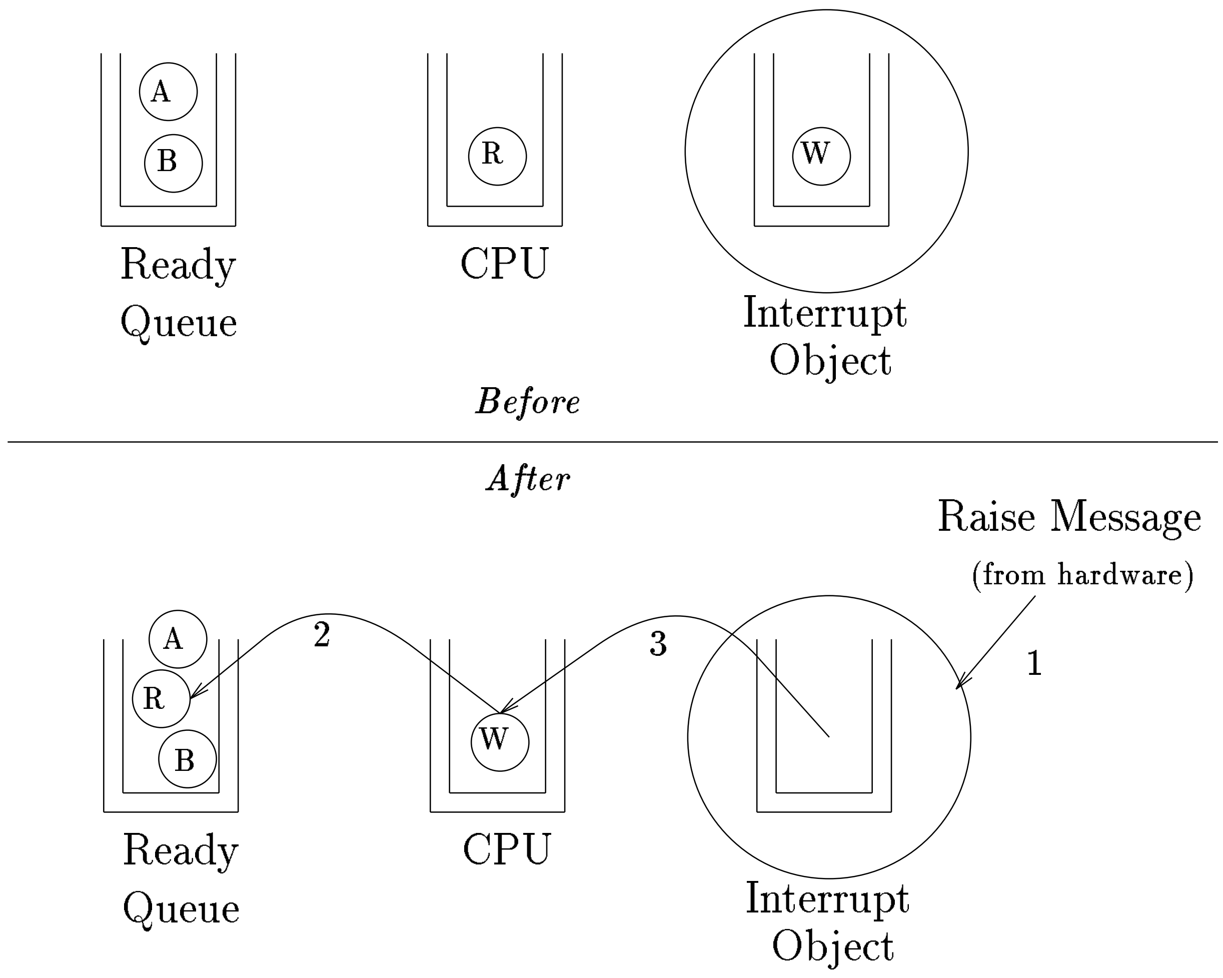

Figure 4: Interrupt Processing.

interrupt occurs, at which time it can be resumed. InterruptExceptions *must* be awaited if they are not to be missed. The raise method of the InterruptException class saves the context of the interrupted process, adds it to the system's "ready queue" ProcessContainer and resumes the Process awaiting the occurrence of the interrupt. Figure 4 shows the sequence of events in more detail. Before the interrupt occurs, Process **R** is running on the processor and Process **W** is awaiting the interrupt's occurrence. The InterruptException object contains as an instance variable a ProcessContainer to hold the Process awaiting the interrupt. In the figure, Process **R** is removed from the CPU object and added to the ready queue ProcessContainer. Finally Process **W** is removed from the interrupt objects ProcessContainer and added to the CPU object.

In addition to the synchronous hardware interrupts, described above, that must have a process awaiting them described above, Choices provides asynchronous interrupts. For example, a time-slice interrupt is handled by an instance of the *TimeSliceInterrupt* class. TimeSliceInterrupts are not awaited. When one occurs the running process is pre-empted, removed from the CPU, and placed on the ready queue ProcessContainer. Another process is then chosen from the ready queue ProcessContainer and added to the CPU.

The above example highlights an advantage of the class hierarchical design method. Instances of ProcessContainer and its subclasses can occur many different places within a system and in many different layers. The abstract ProcessContainer class defines an interface that is known throughout the system. In a strictly layered system, if the function of a scheduler were defined at level L_n, any level lower than L_n would have to define it's own methods of keeping lists of processes. With a class hierarchy, however, once the interface of ProcessContainer is defined, subclasses and instances can be created and used anywhere throughout the system.[10] Comprehension and maintenance is facilitated if all subclasses of ProcessContainer behave, in the abstract sense, alike.

4 Semaphores

In Choices, a semaphore [4] is implemented by the *Semaphore* class and its methods, **P** and **V**, defined abstractly as follows:

```
P(Semaphore):
   count := count - 1;
   if ( count < 0 ) {
      Block on the queue of Processes waiting on the Semaphore.
   }

V(Semaphore):
   count := count + 1;
   if ( count <= 0 ) {
      Wakeup one of the waiting Processes.
   }
```

A constructor is also provided by the Semaphore class to set the initial value of the semaphore.

In order to maximize parallelism in a shared-memory multiprocessor environment, a semaphore must not only provide mutual exclusion for the execution of its methods, but it must also ensure that its methods are completed quickly, without interruption. For example, disabling interrupts will provide mutual exclusion on a single processor system, but not on a multiprocessor shared memory system. A test-and-set operation on a lock may be used to guarantee that a method is executed in mutual exclusion on a multiprocessor. However, unless interrupts are disabled, test-and-set operations can lead to many wasted CPU instruction cycles if a process is suspended by an interrupt or time-slice expiration while it has possession of the lock. Therefore, the correct implementation for a multiprocessor test-and-set spin lock should first disable interrupts to prevent the process attempting the lock being preempted, and then attempt to acquire the lock with a test-and-set operation. When

[10]This same effect cannot easily be achieved with languages that do not support upward coercion of types through the class hierarchy.

the process has completed its critical section, the lock should be released and interrupts reenabled.

A queue of suspended processes is associated with each semaphore. A mechanism is required to transfer a process from the CPU to this queue when the process requests a blocking P method on the semaphore. Correspondingly, a mechanism must exist to move a blocked (enqueued) process from this queue into the system ready queue when another process executes a V method on the semaphore.

A new subclass of SoftwareException, *SemaphoreException*, is used to handle the blocking of processes requesting a P method on a busy Semaphore. An instance of a SemaphoreException, a ProcessContainer, and an integer count variable (along with a test-and-set spin lock to guarantee its atomic update) are the only instance variables of the Semaphore class. The ProcessContainer is used to hold the Processes waiting on the Semaphore.[11]

When the P method of a Semaphore blocks a process it places it on the queue of processes awaiting the Semaphore. This requires removing the process from the CPU and adding it to the Semaphore object's ProcessContainer. As previously discussed, the convention for removing a Process from the CPU (that is, to suspended it) is to raise an Exception. The raise method of a SemaphoreException is used by the Semaphore's P method to achieve this. The handler for the SemaphoreException chooses another Process to run by removing a different Process from the system ready queue ProcessContainer and adding the result to the CPU.[12] If necessary, the V method of a Semaphore "wakes up" a blocked process by removing it from the Semaphore's ProcessContainer and adding it to the system ready queue.

5 Schedulers

Different subclasses of ProcessContainer are used to implement different scheduling disciplines or policies. The operating system designer that implements a new scheduling policy creates a new subclass of ProcessContainer (or, more likely, subclasses an existing scheduler). The new scheduler redefines the add and remove methods in order to provide the desired behavior. The Process selected by the scheduler for removal is determined by its scheduling policy.

The scheduler interacts with both Processes, and CPUs. Each CPU has an associated scheduler from which a Process may be removed for execution when the CPU becomes idle.

[11]The queueing behavior an individual Semaphore can be modified by changing the type of ProcessContainer storing the queued Processes. Currently, all Semaphores in Choices use a subclass of ProcessContainer that imposes FIFO ordering on adds and removes.

[12]After adding the blocking process to the semaphores ProcessContainer, the handler must release the atomic access lock on the semaphores count variable on behalf of the semaphore. The lock was acquired prior to decrementing and testing the value of the count variable. This must be done at this point to avoid the race of releasing it any earlier and having another process test the count before the blocking process has been enqueued. The SemaphoreException therefore requires access to both the semaphores ProcessContainer and lock. These are passed to the Exception in its constructor which is called by the Semaphore classes constructor.

In the Encore Multimax Choices implementation many CPUs are associated with the same scheduler; but there may be more than one scheduler within the system. This allows the system configurer to group CPUs within a Choices system, associating each group with a different scheduler, thus allowing the partitioning of CPUs according to the scheduling policies that apply to particular application(s). This assignment need not be static, and the system can be repartitioned as necessary.

Each Process has an instance of a *SchedulerInformation* class associated with it that is maintained by the scheduler; it is left undisturbed by the rest of the system. In order to provide time-sliced scheduling, a time-slice or quantum is associated with each process (which may be a value that represents "run to completion"). The quantum of a process may be set by the scheduler's remove method. When a Process is added to a CPU for execution, a timer is initiated which will cause a TimeSliceInterrupt Exception to be raised when the time-slice expires. When a Process is removed from a CPU, the amount of unused time is recorded in another field of the Process object. A value of zero indicates that the Process used the entire quantum. When the Process is added to a scheduler, the scheduler can examine this information and use it for future scheduling decisions or for updating the SchedulerInformation it maintains for the Process.

As an example, consider a system in which a single, centralized scheduler exists; that is, all Processes and CPUs are associated with the same scheduler. In addition, suppose this scheduler imposes a time-sliced scheduling discipline on the system. We begin with the execution of an Exception handler which has removed a Process from the CPU and added it to some other ProcessContainer. At this point the Exception handler removes a Process from the scheduler associated with the CPU on which it is executing and adds it to the CPU. When the Process is added to the CPU, its time-slice quantum is examined. If this quantum is not "run-to-completion," a timer is armed which will raise a TimeSliceInterrupt Exception when the quantum expires. When the current Exception handler completes its work the Process it has added to the CPU is dispatched and begins execution. Assuming that no other Exception is raised on this CPU, the timer will raise the TimeSliceInterrupt Exception at the end of the time-slice. The handler for this Exception removes the current Process from the CPU and adds it to the Process' scheduler. At this point we have come full circle and the Exception handler removes another Process from the CPU's scheduler for execution.

Current Choices schedulers include a *FIFOScheduler* for run-to-completion scheduling of Processes and a *RoundRobinScheduler* which provides simple time-slicing. Other schedulers can be built either by deriving specialized subclasses from existing scheduler classes or by creating wholly new ProcessContainer subclasses. An example of the latter is the *MultiLevelFeedbackQueue*, which contains several RoundRobinSchedulers. These schedulers represent the different priority levels within the queue and each provide for a different time-slice quantum, if desired. A MultiLevelFeedbackQueue maintains the dynamic priority level of the process. A Process added to the queue has its priority updated to be either the next lower level (if it used up its entire time-slice quantum) or the highest priority level (if it relinquished the CPU voluntarily, perhaps to perform I/O). The Process is then placed on the

Preliminary Choices Performance Data Encore Multimax 32032 (0.75 MIP)		
Operation	Encore 4.2 BSD Unix	Choices
System Call Overhead	173μsec	39μsec
Process Creation	26.3msec	3.8msec
Context Switch	—	536μsec
Shared Memory Example[13]	0.032sec	0.022sec

Table 1: Performance Data.

internal queue associated with its new priority. Removing a Process from a MultiLevelFeedbackQueue involves removing a Process from the highest level internal queue which is not empty.

The ProcessContainer is a powerful abstraction that may be used to provide encapsulation of physical CPUs and temporary storage of processes.

6 Summary

A Choices kernel currently runs on a 10 processor Encore Multimax that supports the classes and concepts discussed in this paper.

Of particular concern during the development of the system is whether or not the class hierarchical approach can support the construction of *entire* operating systems. In this paper we discuss how this approach can promote the structuring of levels within an operating system, encourage reuse, encapsulate decisions and policies, and permit alternate implementation.

C ++ was chosen as an implementation language because it supports class hierarchies and inheritance while imposing negligible performance overhead at run-time. A software monitor is being used to evaluate the performance of Choices on an Encore Multimax with NS32032 processors. Although it is difficult to provide a meaningful performance measurement of an operating system, we have obtained results that are encouraging. These are shown in Table 1. The "system call" overhead (including a trap and change to supervisor state) compares favorably with UNIX and is only about four times the overhead of a normal procedure call. The Process creation time includes creation of new virtual memory Spaces for the Process. Further tuning will improve these figures.

Current effort is devoted towards improvement and further implementation of communication and persistent object support. Future plans include an object-oriented file system, an advanced interface compiler, and tools for configuring Choices systems. Once Choices is stable, the code will be placed in the public domain to promote research into customized operating systems.

[13]The example creates four processes on independent processors, three sum a ten column array and the fourth sums the three resulting sums. The Multimax multitasking library package was used under UMAX.

References

[1] Roy Campbell, Vincent Russo & Gary Johnston, "The Design of a Multiprocessor Operating System," *Proceedings of the USENIX C++ Workshop* (1987).

[2] Roy H. Campbell, Gary M. Johnston & Vincent F. Russo, "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)," *Operating Systems Review* 21 (July 1987), 9–17.

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

[4] Edsger W. Dijkstra, "The Structure of the THE-Multiprogramming System," *Communications of the ACM* 11 (May 1968), 341–346.

[5] David R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software* (April 1984).

[6] Richard F. Rashid & George G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *Proceedings of the Eighth Symposium on Operating Systems Principles* (December 1981).

[7] Andrew S. Tanenbaum & Sape J. Mullender, "An Overview of the Amoeba Distributed Operating System," *Operating Systems Review* (July 1981).

[8] L. D. Wittie & A. Van Tilborg, "MICROS - A Distributed Operating System for MICRONET - A Reconfigurable Network Computer," in *Tutorial: Microcomputer Networks*, H. A. Freeman and K. J. Thurber, ed., IEEE Press, 1981, 138–147.

[9] D. R. Brownbridge, L. F. Marshall & B. Randell, "The Newcastle Connection, or UNIXes of the World Unite!," *Software - Practice and Experience* (1982).

[10] G. Popek, B. Walker & others, "LOCUS: A Network Transparent High Reliability Distributed System," *Proceedings of the Eight Symposium on Operating Systems Principles* 15 (December 1981).

[11] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian & Michael Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition* (June 1986).

[12] Andrew P. Rifkin & others, "RFS Architectural Overview," *USENIX Conference Proceedings* (1986).

[13] Dan Walsh & others, "Overview of the Sun Network File System," *USENIX Conference Proceedings* (January 1985).

[14] Encore, *Encore Multimax Technical Summary*, Encore Computing Corporation, 1986.

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

[16] Lubomir Bic & Alan C. Shaw, *The Logical Design of Operating Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1988, Second Edition.

[17] Barbara H. Liskov, "The Design of the Venus Operating System," *Communications of the ACM* 15 (March 1972), 144–149.

[18] A. N. Habermann, L. Flon & L. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems," *Communications of the ACM* 19 (May 1976), 266–272.

[19] Intel, *System 432/600 System Reference Manual*, Intel Corporation, 1981.

[20] D. M. Ritchie & K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM* 17 (July 1974), 365–375.

[21] Harvey M. Deitel, "Case Study: VAX," in *An Introduction to Operating Systems*, Addison-Wesley, Reading, MA, 1984, 505–533.

[22] S. E. Madnick & J. J. Donovan, "Virtual Machine/370 (VM/370)," in *Operating Systems*, McGraw-Hill, New York, 1974, 549–563.

[23] Daniel C. Halbert & Patrick D. O'Brien, "Using Types and Inheritance in Object-Oriented Programming," *IEEE Software* (September 1987).

[24] Walter F. Tichy, "What Can Software Engineers Learn from Artificial Intelligence?," *IEEE Computer* (November 1987).

[25] Paul R. McJones & Garret F. Swart, *Evolving the UNIX System Interface to Support Multithreaded Programs*, Systems Research Center, Digital Equipment Corporation, Palo Alto, California, September 28, 1987.

Contents