Report No. UIUCDCS-R-88-1460 UILU-ENG-88-1766

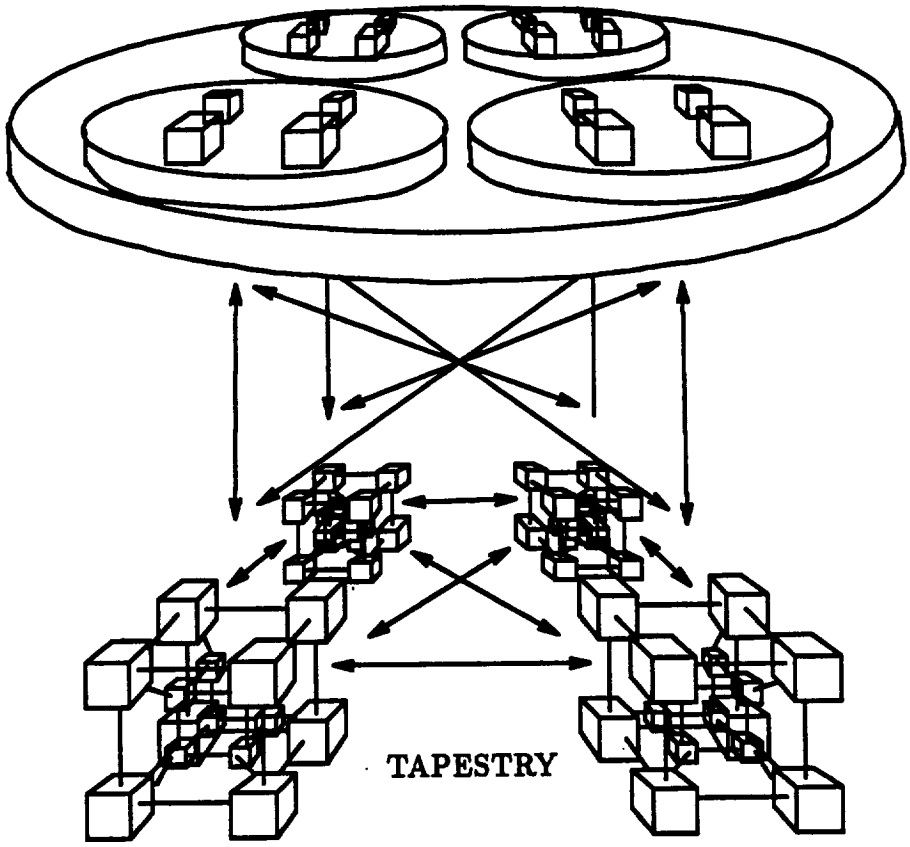
TAPESTRY
Technical Report No. TTR88-5

Principal Investigators Roy Campbell
and Daniel Reed

A Multiprocessor Operating System Simulator

Gary M. Johnston
Roy H. Campbell

September 26, 1988



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS 61801-2987

Contents

A Multiprocessor Operating System Simulator*

Gary M. Johnston
Roy H. Campbell

September 26, 1988

Abstract

This paper describes a multiprocessor operating system simulator that was developed by the authors in the Fall semester of 1987. The simulator was built in response to the need to provide students with an environment in which to build and test operating system concepts as part of the coursework of a third-year undergraduate operating systems course.

Written in C++ [1], the simulator uses the co-routine style *task* package [2] that is distributed with the AT&T C++ Translator to provide a hierarchy of classes that represents a broad range of operating system software and hardware components. The class hierarchy closely follows that of the *Choices* [3] family of operating systems for loosely- and tightly-coupled multiprocessors. During an operating system course, these classes are refined and specialized by students in homework assignments to facilitate experimentation with different aspects of operating system design and policy decisions.

The current implementation runs on the IBM RT PC[1] under 4.3bsd UNIX.[2]

1 Introduction

The principles of low-level operating system design have implications that are difficult to appreciate without the practical experience that is gained from programming multiprocessor systems. However, it is difficult to provide university students with such a learning experience. Hardware resources are too expensive to allow each student single user access to a multiprocessor workstation. Low level parallel processing systems software, as an instructional resource, is usually poorly organized and difficult to understand. In addition, there is little support for the debugging and testing of low-level systems programs on multiprocessors. This paper describes a multiprocessor operating system simulator we have constructed

*This work was supported in part by NSF grant CCR-8-8-09479 and CISE-1-5-30035, by NASA grant NSG1471, and by AT&T ISEP.

[1]RT PC is a trademark of IBM.

[2]UNIX is a trademark of AT&T.

in C++ to overcome these problems. The current implementation is used in the department's instructional laboratory, running on 30 IBM RT PCs which were donated to the university by IBM Corporation.

The simulator is modeled on the Choices multiprocessor operating system family [3] [4] [5]. It includes classes to model both the processes, schedulers, and exception handling mechanisms of Choices and the processors, I/O devices, traps, interrupts, timers, and other hardware components of a typical multiprocessor system like the Encore Multimax.[3]

The simulator was designed for a third year undergraduate course on operating systems that is taught in the Department of Computer Science at the University of Illinois at Urbana-Champaign. The goal of the course is to introduce students to the principles of operating systems and to reinforce those principles with practical experiments and projects involving the design of operating system mechanisms and policies.

Using the simulator, experimentation is conducted within the framework of the class hierarchy and object-oriented programming mechanisms afforded by C++. Many of the practical design exercises involve specializing an abstract class into a concrete class that implements a particular policy or mechanism. Policy exercises include process scheduling, real memory management, page replacement, and disk scheduling. Mechanism exercises include synchronization primitives, I/O queues, paging mechanisms, exception handling schemes, and message passing primitives.

The operating system course benefits from the use of C++ in several ways. The language allows an efficient simulation of the operating system while providing a level of type checking that aids debugging of student programs. Debugging and tracing aids are built into the base classes of the simulator and help the students implement their designs. The class hierarchy organizes the components of the simulation into similar algorithms and data structures. This organization is a useful aid to the student that is learning the system. The class hierarchy enables fairly large simulations of an operating system to be built incrementally by the students.

The remainder of this paper consists of four major sections. Section 2 describes the model of the Choices operating system and class hierarchy supported by the simulator. Section 3 discusses the design and implementation of the simulator. Section 4 describes how the simulator was used, including descriptions of some of the projects. Finally, we summarize our experience with the simulator in section 5.

2 Choices Overview

Choices is a family of operating systems built using a class hierarchical object-oriented approach to systems design and programming. A Choices operating system has been implemented on an Encore Multimax and is being ported to an Intel iPSC/2[4] hypercube [6]. It demonstrates that object-oriented design techniques are both appropriate and beneficial for

[3]Multimax is a trademark of Encore Computer Corporation

[4]iPSC is a trademark of Intel Corporation.

Choices Simulator Classes				
Class	Methods			
Object	–	–	–	–
↑Process	–	–	–	–
↑↑IdleProcess	–	–	–	–
↑ProcessContainer	**add**	**remove**	–	–
↑↑CPU	*add*	*remove*	**interrupt**	**trap**
↑↑FIFOQueue	*add*	*remove*	–	–
↑↑ProcessQueue	*add*	*remove*	–	–
↑Exception	**raise**	**await**	**handle**	–
↑↑InterruptException	↑	↑	*handle*	–
↑↑↑ResetException	↑	↑	*handle*	–
↑↑↑TimerException	↑	↑	*handle*	–
↑↑SoftwareException	*raise*	↑	*handle*	–
↑↑↑IdleException	↑	↑	*handle*	–
↑↑↑TerminateException	↑	↑	*handle*	–
↑↑↑SemaphoreException	↑	↑	*handle*	–
↑Semaphore	**P**	**V**	–	–

Table 1: Choices Simulator Classes.

writing complete operating systems for multiprocessors and networks of multiprocessors.

2.1 Why a Class Hierarchy?

In Choices, the class hierarchy represents the major components of a family of operating system designs. Classes represent the interfaces and implementations of processes, virtual memory, context switching, exception handling, scheduling, and synchronization. They are also used to provide a hardware/software interface by encapsulating machine dependent algorithms and data structures for the hardware entities such as the CPUs, MMUs, interval timers, disks, and networks.

In our experience, an operating system designer may select, refine, and combine classes from the class hierarchy in order to build a custom operating system for a particular hardware environment or a particular application. The resulting operating system is also more easily modified or extended than one based on more "traditional" approaches. The ease of module substitution greatly facilitates prototyping, a great benefit to practical operating systems research and experimentation.

This section presents a brief overview of the Choices project and of the Choices class hierarchy as implemented by the simulator. For more detail, see [3] [4] [5].

2.2 The Choices Class Hierarchy

Legend	
Symbol	Meaning
method	Definition of method.
method	Redefinition of method.
↑	Subclass or inherited method.
–	Undefined method.

Table 2: Class Table Legend.

The major classes of Choices as modeled by the simulator are shown in table 1. The table is explained by the legend shown in table 2. Class *Object* is the root of the hierarchy. Subclasses are used to provide abstract interfaces and concrete implementations for operating system mechanisms. They are used to encapsulate data, policies, and alternative implementations or versions. Subclasses of Object define the basic entities within an operating system. Subclasses of these classes add and/or redefine methods in order to augment, specialize, or provide concrete implementations of these classes.

2.2.1 Processes and ProcessContainers

Class *Process* provides the basic unit of execution within Choices. Process management in the operating system is achieved by moving Processes between *ProcessContainers*. Subclasses of ProcessContainers represent processors and schedulers.

An *IdleProcess* is associated with each simulated processor. It is executed only when there are no other Processes available. Each IdleProcess periodically checks the scheduler and signals the processor when it detects that there are Processes which could be executed.

Other Processes represent "user-level" processes. The behavior is redefined by the simulation designer as necessary. Usually user-level Processes are designed to mimic some type of program behavior in order to provide a "job load" for the simulation.

The *CPU* subclass of ProcessContainer represents processors. Adding a Process to a CPU specifies that a particular process should be executed by a particular processor of the multiprocessor system; that is, the Process is dispatched on the CPU. Removing a Process from a CPU idles the processor, which represents preemption of the Process.

The ProcessContainer class defines methods to *add()* and *remove()* Processes. These methods are specialized by the subclasses CPU, *FIFOQueue*, and *ProcessQueue*. An instance of a CPU is active (the active component is the processor). However, a FIFOQueue and ProcessQueue may contain many Processes although they are passive. The simulator itself runs on a single UNIX process, and the classes of Choices it uses are specialized to emulate a parallel processor. For example a CPU is implemented by tasks in order to emulate the functions of a processor.

Facilities for scheduling and blocking Processes are provided by classes FIFOQueue and ProcessQueue. A FIFOQueue acts as a simple "first-in-first-out" queue of Processes, while a ProcessQueue is associated with a timeslice quantum. When a Process is removed from a

ProcessQueue, the timeslice quantum field of the Process is set to the quantum associated with the ProcessQueue. This field is used by the CPU to determine the maximum amount of time the Process should be allowed to execute before being preempted. The quantum associated with a ProcessQueue may be any value desired, supporting timeslicing. The default quantum is a value which means "run-to-completion." These classes may be refined by other subclasses in order to implement a wide range of policies. FIFOQueues can act as queues of blocked Processes. Other subclasses of ProcessContainer can be defined and substituted to provide whatever sorts of scheduling disciplines the system designer desires.

2.2.2 Exceptions

In Choices, most movement of Processes between ProcessContainers is done by *Exception* handlers. In addition, the only way in which an executing Process can relinquish its CPU is by the raising of an Exception. Relinquishing the CPU may be a voluntary, synchronous action performed by the Process (i.e., a "trap") or an involuntary, asynchronous action caused by an external event (i.e., an "interrupt").

Class Exception itself is an abstract class in that no direct instances of class Exception ever exist. Rather, it provides a base class from which subclasses may be derived in order to provide specialized behavior. An Exception provides the methods *handle()*, *raise()*, and *await()*. The raising of an Exception causes its handler to be invoked (with the possible side-effect of unblocking one or more Processes awaiting the Exception).

There are two abstract subclasses of Exception: *InterruptException* and *SoftwareException*, each of which is further subclassed. An InterruptException is associated with an interrupt vector which, when delivered to a CPU, causes the associated InterruptException to be raised. Thus, InterruptExceptions occur asynchronously with the execution of Processes. A SoftwareException is *not* associated with an interrupt vector. Instead, it is raised directly by an executing Process and acts like a "trap."

InterruptException subclasses include *ResetException* and *TimerException*. Each CPU is associated with an instance of each of these. A ResetException provides the actions to be taken when the CPU is "reset". The TimerException handles the expiration of the per-CPU interval timer.

SoftwareException subclasses include *IdleException*, *SemaphoreException*, and *TerminateException*. An IdleException is a software event that signifies that Processes are available to the CPU for execution. An IdleException is raised by a CPU's IdleProcess when it detects that the CPU's scheduler is non-empty. A TerminateException is raised to remove the Process from the CPU and delete it. A SemaphoreException is raised when a Process attempts to acquire a semaphore which is unavailable. The SemaphoreException removes the Process from the CPU and adds it to the queue of Processes waiting for the semaphore.

2.2.3 Semaphores

A *Semaphore* is the basic synchronization primitive within the simulator. It defines the familiar *P()* and *V()* operations [7] for acquiring and releasing the Semaphore.

3 Implementation

The simulator provides a class hierarchy from which simulated multiprocessor operating systems can be designed and studied, following the Choices model as closely as possible. This section discusses the implementation of the simulator under 4.3bsd UNIX.

3.1 Microscheduling

Like Choices itself, the simulator is written in C++. In order to provide the required simulated concurrency, the simulator was written using the "coroutine-style task package" which accompanies the AT&T C++ Translator [2].

The task package provides user-level coroutine-style tasks, but does not provide for non-voluntary relinquishing of the virtual processor. That is, an executing task does not block unless it explicitly calls a task package procedure (for example, *delay()* or *sleep()*). While this is very useful for system simulation, it is inadequate to emulate a multiprocessor programming environment realistically. A simulated user-level task executing an "infinite loop" will prevent all the other simulated tasks from proceeding. This simple implementation of tasks is inadequate to emulate interrupts or preemptive scheduling policies such as round-robin time-slicing, multi-level feedback queues, or "shortest job first." In addition, we wanted to simulate the nondeterminancy that must be dealt with by programs using or implementing synchronization primitives and executing within a multiprocessor environment. Therefore, the basic task package was augmented with a "microscheduling" sub-system that time-slices between executable tasks preemptively. Note that this involved only *additions* to the task package. The task package itself was *not* modified.

The microscheduling mechanism implements a time-sliced round-robin mechanism underneath the basic task package. This mechanism gives each executable (i.e., non-blocked) task a "microquantum" equal to one virtual clock tick. At the end of the microquantum, the task is delayed for one clock tick, and the next executable task is dispatched. In this manner, executable tasks are preemptively time-multiplexed on the underlying UNIX process.

The 4.3bsd UNIX interval timer and signal mechanisms were used to implement the actual preemption of tasks. At simulator initialization time, an interval timer is armed to deliver a signal to the underlying UNIX process each time it expires. When the signal is received, the signal handler executes in the context of the current task. The signal handler executes a call to the task package to delay the current task by one virtual clock tick, thus relinquishing the underlying UNIX process to execute another runnable task. If there are no more immediately runnable tasks, the virtual clock is incremented (by the task system), allowing tasks which had delayed themselves during the previous clock tick to become "ready" again. When a task that had previously delayed itself via the signal handler becomes ready again, its invocation of the signal handler returns, thus restoring that task's context to that which was in effect when the signal was received. The task then continues execution at the point where it was preempted. Thus, the microscheduling effectively implements a round-robin scheduling policy underneath the existing task package.

The basic task package requires no explicit shared resource access control internally because there is no preemption. Provided that critical sections do not delay, they do not need synchronization because, without preemption, races cannot occur. Once microscheduling has been added, however, this is no longer the case. Within the Choices classes, mutual exclusion primitives are used in order to ensure that critical sections are protected. In order to support these primitives in the simulator, instances of two low-level task classes are distinguished by the microscheduling mechanism and are *not* preempted.[5] Therefore, these classes' methods do not need to use explicit mutual exclusion primitives.

3.2 Class Hierarchies and Layering

The simulator is organized into two major class hierarchies: the augmented task package class hierarchy (including the microscheduling mechanism) and the Choices class hierarchy itself.

The basic task package provides the abstraction of a *task*, which is the primitive unit of execution within a task package application. This hierarchy has been augmented by creating subclasses of the task class in order to provide more specialized behavior as needed by the rest of the simulator. These classes are *CPUManager*, *CPUTimer*, and *ProcessTask*. A CPUManager and a CPUTimer are associated with each simulated CPU. The CPUManager simulates the activity of the CPU. This includes interrupt vector processing, trap processing, and exception handling actions. The CPUTimer simulates a per-CPU interval timer to provide support for preemptive time-slicing of simulated Processes. A ProcessTask is associated with each simulated Process. The CPUManager associated with a CPU allows the ProcessTask to execute (on behalf of the simulated Process) when the Process is dispatched on that CPU.

The Choices simulator class hierarchy provides the classes that form the basis for operating system simulations: Process, ProcessContainer, CPU, Exception (and its subclasses), etc. Table 1 shows this hierarchy. Figure 1 shows the arrangement in terms of layers.

3.3 Class CPU

A CPU contains a number of objects in addition to its CPUManager and CPUTimer. Each CPU has a current Process and an IdleProcess. The current Process is the Process currently being executed by that CPU. Since a CPU is a ProcessContainer, the current Process of a CPU references the Process which the CPU contains (if any). The IdleProcess is executed only when the CPU is otherwise idle (e.g., when there are fewer Processes in the "system" than there are CPUs[6]).

Next, a CPU contains a queue of pending interrupt vectors and a table that maps interrupt vectors to InterruptExceptions. Incoming interrupt vectors and SoftwareExceptions are detected by the CPUManager which then executes the InterruptException handlers.

[5] These classes are *CPUManager* and *CPUTimer*, discussed below.

[6] not including IdleProcesses, of course.

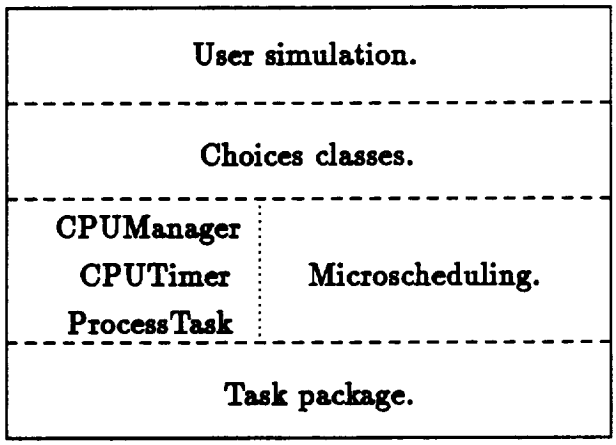


Figure 1: Conceptual Layering in the Choices Simulator.

Each CPU references a ProcessContainer that operates as the "ready queue" or scheduler. When an executable Process is removed from the CPU, it is added to this scheduler. Also, a Process is removed from this scheduler when the CPU requires one. For example, when an executing Process' timeslice expires, it is removed from the CPU and added to this scheduler. Then, another Process is removed from the scheduler and added to the CPU.

In this way, several CPUs may be associated with a particular scheduler. There is no reason why there can't be more than one scheduler in the system, each associated with its own set of CPUs. The simulation designer can change this association dynamically at any time.

There are two groups of operations on a CPU: "private" routines intended for use by "friends" (essentially CPUManagers and Exception handlers) and "public" routines intended for use by the simulation writer.

The private routines include *add()*, and *remove()*, which are redefinitions of the superclass ProcessContainer methods for adding/removing Processes to/from a ProcessContainer. Adding a Process to a CPU is effectively a "dispatch" of the Process, while removing a Process from a CPU corresponds to a "preemption" of the Process.

Two other important private routines are *removeVector()* and *getException()*. These are used by the CPUManager to remove an interrupt vector from the incoming vector queue, and to map a vector to an InterruptException, respectively.

The public operations include the constructor and destructor, routines to get and set the CPU's scheduler ProcessContainer, the *interrupt()* routine which is used to send an interrupt to a CPU, the *trap()* method which is used when a SoftwareException is raised, and the *setException()* routine which is used to associate an interrupt vector with an InterruptException.

When a CPU is created it is empty, i.e., it contains no Process. The Exception table (which maps interrupt vectors to InterruptExceptions) contains two default mappings: a ResetException is associated with the ResetVector, and a TimerException is associated with the TimerVector.

In the implementation, the CPU itself is passive; it is the CPUManager and the CPUTimer which are the active entities, controlling the activities of the CPU. These are discussed next.

3.4 CPUTimer

A CPU's CPUTimer implements timed preemption of Processes. A CPUTimer is a task that sends the TimerVector to the CPU when the time interval expires, unless the CPUTimer is stopped first. If the CPUTimer is stopped before it expires, then the residual time can be retrieved.

In general, when a Process that specifies a timeslice quantum is dispatched, the CPUManager sets the CPUTimer to expire at the appropriate time. If the CPUTimer expires, the TimerVector interrupt triggers the execution of the associated InterruptException's handler (usually a TimerException). If the Process is preempted for some reason other than CPUTimer expiration, the CPUTimer is stopped and the residual is read and stored in a field of the Process for possible use by the scheduler.

3.5 CPUManager Duties

The CPUManager handles asynchronous events in the system like interrupts as well as traps, and invokes the Exception handlers associated with them. The CPUManager is initially "asleep," and the arrival of an interrupt or trap "wakes up" the CPUManager. When a CPU's *interrupt()* method is called, the vector is enqueued on the CPU and its CPUManager is awakened. When a CPU's *trap()* method is called, the SoftwareException is saved on the CPU, the invoking Process is stopped, and the CPUManager is awakened. The general control loop of the CPUManager is shown in figure 2.

3.6 Processes and ProcessTasks

Each Process is implemented by a *ProcessTask* which executes when the Process is dispatched on a CPU. Each Process contains a timeslice quantum and a residual, which is used for preemptive timeslicing. The residual field is set by the CPU when the Process is preempted. This information is intended for use by schedulers. In addition, each Process keeps run-time statistics.

The ProcessTask associated with a Process is the entity which is actually executed. It is ProcessTasks that are multiplexed on the underlying UNIX process by the microscheduling mechanism. The task methods are used by a CPUManager to start and stop the execution of a Process' ProcessTask. In order to provide low-level critical section protection, methods are provided to disable and re-enable the preemption of a ProcessTask by the microscheduling mechanism.

IdleProcess is the subclass of Process that is executed by a CPU when there are no other Processes for it to run. There is one IdleProcess associated with each CPU. The IdleProcess

```
// A CPUManager's work is never done...
for (;;) {
    // Wait for an interrupt.
    sleep();

    // Stop and delete the CPUTimer, if there is one, saving the residual.
    int residual = 0;
    if ( cpu->timer != NULL ) {
        residual = cpu->timer->stop();
        delete cpu->timer;
        cpu->timer = NULL;
    }

    // Handle and reset the pending trap (SoftwareException), if there is one.
    // Otherwise, stop the current Process, if there is one.
    Process * currentProcess = cpu->currentProcess;
    if ( cpu->trap != NULL ) {
        SoftwareException * trap = cpu->trap;
        cpu->trap = NULL;
        trap->handle( cpu );
    } else if ( currentProcess != NULL ) {
            currentProcess->stop();
    }

    // Handle any pending interrupts (InterruptExceptions).
    while ( ( int vector = cpu->removeVector() ) != NoVector ) {
        // Get the corresponding Exception.
        // Call the Exception handler.
        InterruptException * interrupt = cpu->getException( vector );
        interrupt->handle( vector, cpu );
    }

    // Start the current Process, if there is one.
    // Note: The current Process we start here might very well not be
    //       the same one we stopped.
    if ( cpu->currentProcess != NULL ) {
        // Determine how much time the Process will get:
        //     If the current Process is the same as before,
        //     it gets the rest of its timeslice (i.e., the residual).
        //     Otherwise, it gets whatever its scheduler specified.
        int time = (cpu->currentProcess == currentProcess) ?
                    residual :
                    cpu->currentProcess->getQuantum();

        // Start the CPUTimer, unless the Process is marked "run to completion."
        if ( time != RunToCompletion )
            cpu->timer = new CPUTimer( cpu, time );
    }
}
```

Figure 2: Simplified CPUManager control loop.

continually checks the scheduler ProcessContainer of its CPU. When it detects that this scheduler is not empty, it raises an IdleException which causes a Process to be removed from the scheduler and added to the CPU, suspending the IdleProcess until such time as the CPU becomes idle again.

3.7 Exceptions

The Exception subclasses are the major means by which Processes are moved between ProcessContainers in a Choices system itself and in the simulator. Each Exception subclass provides specialized handling. There are two subclasses of Exception, InterruptException, and SoftwareException. Instances of subclasses of InterruptException represent hardware interrupts. When an interrupt is delivered to a CPU, it is mapped by the CPUManager to an InterruptException whose handler is then called. Subclasses of InterruptException include:

ResetException: Associated with the ResetVector. It adds the CPU's IdleProcess to the CPU.

TimerException: Associated with the TimerVector which is sent when the CPU's CPUTimer expires. It removes the current Process from the CPU and adds it to the scheduler ProcessContainer associated with the CPU. It then removes a Process from the scheduler and adds it to the CPU.

A SoftwareException is raised as a direct result of the execution of a Process. SoftwareExceptions are not associated with interrupt vectors; the raise method is invoked directly. SoftwareException subclasses include:

IdleException: Raised when a CPU's IdleProcess detects that the CPU's scheduler has become non-empty. Its handler removes the IdleProcess from the CPU, and then removes a Process from the scheduler and adds it to the CPU.

TerminateException: Raised when the current Process on the CPU is to be terminated. It removes and deletes the current Process from the CPU, and then removes a Process from the scheduler and adds it to the CPU.

SemaphoreException: Raised by a Semaphore when a *P()* operation detects that the requesting Process must block (i.e., the resource is not available). It removes the current Process from the CPU and adds it to the ProcessContainer associated with the Semaphore. It then removes a Process from the CPU's scheduler and adds it to the CPU.

3.8 Semaphores

Each Semaphore contains a count and a FIFOQueue ProcessContainer which holds Processes that have been blocked attempting to acquire the Semaphore. It also references a SemaphoreException that is raised when a Process must block.

The *P()* operation decrements the count. If the count then indicates that the Process must block, a SemaphoreException is raised. The SemaphoreException removes the Process from the CPU and adds it to the queue of blocked Processses.

The *V()* operation increments the count. If there are blocked Processes, one is removed from the queue and added to the scheduler.

4 Experience

The resulting simulator has proven to be very realistic. Several of the race conditions that occurred as bugs in the development of the real Choices operating system were also encountered by students as they developed their own operating system components within the simulator. During the course, the students developed semaphores, messages, supervisor requests, scheduling policies, real storage management, virtual storage management, disk storage management and scheduling for the multiprocessor environment. This section discusses some of these projects and how they were implemented within the environment of the simulator.

4.1 Multiple Concurrent Producers and Consumers

The object of this exercise was to give students experience in designing systems involving producer/consumer relationships among Processes, including deadlock detection and recovery.

Initially, class *Pipe* had to be implemented to support a two-ended stream of *Messages*. Methods were required to perform blocking, non-blocking, and synchronous send operations (*send_block()*, *send()*, and *send_sync()*, respectively), as well as blocking and non-blocking receive operations (*receive_block()* and *receive()*, respectively). Each Message essentially consists of a string of data bytes and an identifier specifying the ultimate destination Process.

In this exercise, Processes are connected by Pipes in a ring, as shown in figure 3. Each Process executes a loop in which it repeatedly choses one of the send or receive operations at random, and performs this operation on one of its two Pipes. For send operations, destinations are chosen at random. For receive operations, if a Message is received on a Pipe whose destination does not specify the receiving Process, it is forwarded on the other Pipe. Since the Processes are arranged in ring, all Messages eventually reach their destinations (unless they are lost or cancelled).

In this situation, deadlocks can and did occur. Students implemented a centralized deadlock detection and recovery mechanism which consisted of a central Pipe *Control* information object and an additional deadlock control Process which would periodically examine the Control information, discovering and breaking deadlock situations. The Pipe class was modified to support this. Each send and receive operation on a Pipe would report its updated state to the Control object, where it could then be used by the deadlock control Process.

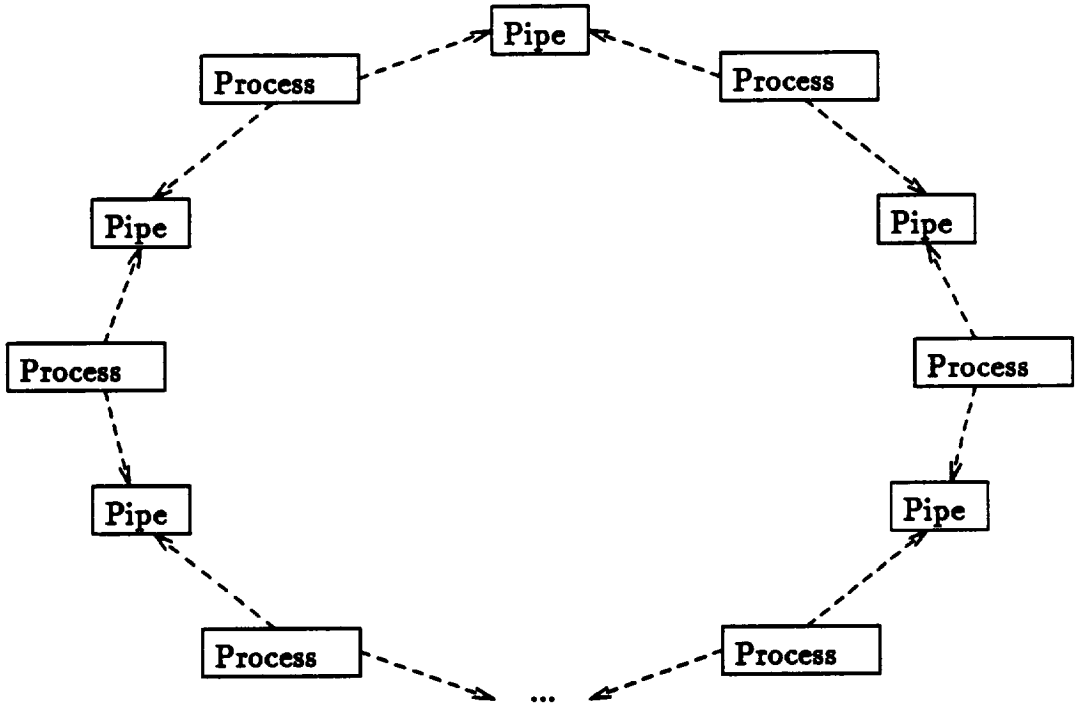


Figure 3: A Ring of Processes Connected by Pipes.

4.2 Real Memory Management

This project involved the implementation of Choices-style real memory management. Two major classes were implemented: *RealMemoryObject* and *RealMemoryManager*.

A RealMemoryObject represents a "segment", or contiguous range, of memory organized in fixed-size pages. The operations supported are *read()* and *write()*. Each operation specifies an *offset* into the RealMemoryObject at which the transfer is to begin, a *length* (in bytes), and a destination/source *buffer* address. Initial reads from unwritten RealMemoryObject locations are read as zeros. The RealMemoryObject maintains a "dirty bit" for each page which has been written. The constructor specifies the range of addresses which the RealMemoryObject will represent.

The other major class required for this project was a RealMemoryManager. A RealMemoryManager represents the physical memory of the simulated machine, so only one instance of this class is created. The RealMemoryManager allocates and deallocates RealMemoryObjects as requested by user Processes. Operations are *allocate()* and *deallocate()*.

The *allocate()* operation specifies a number of bytes, and returns a RealMemoryObject. The RealMemoryManager must find an unallocated range of memory that is at least as large as the request. It then creates a RealMemoryObject to manage the range and returns it.

The *deallocate()* operation specifies a RealMemoryObject to be deleted. The RealMemoryManager deletes the RealMemoryObject, thus freeing the range of memory for possible allocation in future *allocate()* requests.

RealMemoryObject and RealMemoryManager provide simulated system services, and are

not supposed to be directly accessible to the user Processes (although the simulator cannot enforce this). Therefore, the students implemented a subclass of SoftwareException called *SVCException*. This class provides a user program interface to the system. Mechanisms for passing arguments into the "kernel" and for passing results back to the invoking Process were also implemented.

Simulated user Processes were created to randomly allocate and deallocate RealMemoryObjects, and to read and write them randomly. Statistics about memory usage, fragmentation, allocation routine times, etc. were collected. The allocation algorithms commonly known as "first fit," "best fit," and "worst fit" were implemented and analyzed.

4.3 Virtual Memory Management

This project extended the ideas from the previous project in order to provide students with experience in the various aspects of virtual memory management.

The idea of a RealMemoryObject was expanded to represent a Process' virtual address space. This is encompassed by class *MemoryObjectCache*. A MemoryObjectCache maintains the state of each page in the virtual address space it represents. In addition to the "dirty bit" (which was maintained by the RealMemoryObject in the previous project), the MemoryObjectCache must maintain a "referenced bit" and a bit indicating whether or not the page is resident. If the page is non-resident, the location of the page in secondary storage must be stored. A MemoryObjectCache supports the same read and write operations as described for a RealMemoryObject, except that pages may be moved to and from secondary storage.

When a MemoryObjectCache must read or write a page that is marked non-resident, that page must first be retrieved from secondary storage. To facilitate this, an instance of class *PageManager* manages the physical memory of the machine and is responsible for paging to and from secondary storage. The PageManager implements the *pageFault()* method, which is invoked by a MemoryObjectCache when a non-resident page needs to be brought in from secondary storage. The PageManager fetches the specified page from secondary storage and marks it as resident.

Secondary storage is implemented with an instance of class *DiskManager*. The DiskManager responds to the messages *readPage()* and *writePage()*.

Various page replacement algorithms were implemented and studied. These included "least recently used," "not recently used," "first in, first out," and "random." In addition, various disk scheduling strategies were used, including "first come, first served," "linear (or unidirectional) scan," and "circular (bidirectional) scan." Finally, the page access patterns of the Processes were varied in order to simulate different degrees of temporal and spatial locality.

5 Conclusion

In this paper, we have described the use of C++ as a high-level language for describing the system data structures and algorithms introduced in a university undergraduate/graduate course in operating systems. The students used a simulator programmed in C++ that emulated a system based on Choices, an experimental multiprocessor operating system that we are building at the University of Illinois. Class projects and exercises were chosen to give students practice at systems design and programming. These projects and exercises were written in C++ and refined or replaced classes in the simulator.

Most of the students in the course had programmed in C in a previous course on systems programming and machine organization. The transition to C++ was orderly. The students found the additional type checking in C++ an aid; however, many of the diagnostic messages from the compiler required the students to seek help from their teaching assistants. The debugging and tracing aids built into the simulator were found to be very useful as the standard UNIX debugger cannot give accurate diagnostic messages in terms of the names used in C++ programs. This is because the current C++ compiler generates C code which is then compiled by the C compiler. A native C++ compiler would solve many of these problems.

C++ was proved to be an efficient programming language for the simulator. Quite large simulations (both in terms of size and length) could be done on a workstation during the period of time permitted each student in the laboratory.

The use of a class hierarchical object-oriented description of an operating system was instrumental in helping the students understand Choices. The class hierarchies organized the common algorithms and data structures of an operating system and allowed students to infer the properties of the simulator classes from the more abstract classes presented during lecture. Unlike previous operating system courses that we have taught, we were able to present multiprocessor operating system material couched in the general principles of operating system design. The more "traditional" single processor operating system algorithms and data structures could be presented as degenerate cases of the multiprocessor ones.

Currently, a simulator is the only practical approach to providing a large class of students (approximately sixty) with a hands-on environment for multiprocessor operating system design. Many of the problems that are encountered in multiprocessor operating system design — deadlocks, races, unnecessary mutual exclusion and interrupt disabling, etc. — were pointed out in lecture and successfully diagnosed by students during their exercises on the simulator. In this and many other respects, the simulator provided a remarkably accurate emulation of real multiprocessor system software development. The accuracy of that emulation requires better diagnostic and tracing tools than we implemented in the simulator. We believe some form of graphical visualization of the system is needed in order to provide students with a better understanding of the utilization of resources, bottlenecks, and communication flows. However, we do not see this as a drawback to the approach. Rather, it points out a lack of necessary human interfaces and tools for designing complex software. Such software tools would not only be useful in education, but they would have

application in the customization of Choices for particular applications and hardware. We plan to incorporate such tools in the future revisions of the simulator.

References

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

[2] Bjarne Stroustrup & Jonathan E. Shopiro, "A Set of C++ Classes for Co-Routine Style Programming," *Proceedings of the USENIX C++ Workshop* (1987).

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

[4] Vincent Russo, Gary Johnston & Roy Campbell, "Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques," *OOPSLA '88 Conference Proceedings* (forthcoming).

[5] 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.

[6] Roy H. Campbell & Daniel A. Reed, "Tapestry: Unifying Shared and Distributed Memory Parallel Systems," Department of Computer Science, University of Illinois at Urbana-Champaign, Technical Report No. UIUCDCS-R-88-1449, Urbana, Illinois, 1988.

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

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-88-1460	2.	3. Recipient's Accession No.
4. Title and Subtitle A Multiprocessor Operating System Simulator			5. Report Date September 1988 6.
7. Author(s) Gary M. Johnston and Roy H. Campbell			8. Performing Organization Rept. No. R-88-1460
9. Performing Organization Name and Address Department of Computer Science 1304 W. Springfield Ave. 240 Digital Computer Lab Urbana, IL 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No. NSF CCR8-8-09479 CISE 1-5-30035 NASA NSG 1471 AT&T IL SOFT. 1-5-37388
12. Sponsoring Organization Name and Address National Science Foundation Washington, DC 20550 AT&T Information Systems Lincroft, NJ 07738	NASA Langley Research Center Hampton, VA 23665		14.

15. Supplementary Notes

16. Abstracts

This paper describes a multiprocessor operating system simulator that was developed by the authors in the Fall semester of 1987. The simulator was built in response to the need to provide students with an environment in which to build and test operating system concepts as part of the coursework of a third-year undergraduate operating systems course.

Written in C++, the simulator uses the co-routine style task package that is distributed with the AT&T C++ Translator to provide a hierarchy of classes that represents a broad range of operating system software and hardware components. The class hierarchy closely follows that of the Choices family of operating systems for loosely and tightly coupled multiprocessors. During an operating system course, these classes are refined and specialized by students in homework assignments to facilitate experimentation with different aspects of operating system design and policy decisions. The current implementation runs on the IBM RT PC under 4.3bsd UNIX.

17. Key Words and Document Analysis. 17a. Descriptors

computer science education, multiprocessors, operating systems, simulators, class hierarchies, object-oriented design.

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
unlimited	UNCLASSIFIED	21
	20. Security Class (This Page) UNCLASSIFIED	22. Price