A Family of Operating Systems in a Software Laboratory

TR86-023

May 1986

Frederick Allan Fisher

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

A FAMILY OF OPERATING SYSTEMS IN A SOFTWARE LABORATORY

by
Fredrick Allan Fisher

A Thesis submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Approved by:

[illegible]
Adviser

[illegible]
Reader

[illegible]
Reader

Chapel Hill, 1986

FREDRICK ALLAN FISHER.
A Family of Operating Systems in a Software Laboratory
(under the direction of RICHARD T. SNODGRASS).

It is nearly impossible to provide hands-on experience with operating systems in any introductory operating systems course. This difficulty is due to the size of operating systems and the sensitivity of their position in day to day computer operations. A family of operating systems, designed for classroom use, is shown to be a tool by which operating system design experience may be gained. Two fully-documented systems, which have been used in an operating systems class over the past two years, are presented as the nucleus of such a family. Courses using these operating systems should have the same value for computer science students as laboratory courses have for students in the natural sciences. All the code for both of these operating systems is included, along with suggested experiments and assignments.

Acknowledgements

An undertaking of this sort is never fully the work of one person. The author justly runs a grave risk of being considered less than honest if he fails to give credit to those others who were instrumental in bringing the project to a successful conclusion.

First on the list is my adviser, Rick Snodgrass. Not only did he write the first, one-page version of UniBatch; he also wrote a near-final version of UniBatch's **`ContextHandler`**, and made significant revisions to **`LocalSystem`**. An early version of MultiBatch's **`LowLevelScheduler`** was another of his contributions, as were many of the assignments.

Harder to point to than these contributions, but far more important to me, were his willingness to explain puzzles, his ability to excite interest, and his encouragement throughout the long ordeal which resulted in the production of this thesis.

Dick Morrill, developer of the Integrated Instrumentation Environment for the SoftLab project (mentioned within), also deserves credit. He did much of the early work on MultiBatch, and wrote pseudo-code versions of six of the modules in the system: **`MemoryManager`**, **`DiskManager`**, **`ProcessManager`**, **`MediumScheduler`**, **`HighLevelScheduler`**, and **`Swapper`**. Although the final code for some of these modules bears little resemblance to the original pseudo-code versions, they served as an invaluable starting point; and in fact a few of them remained remarkably stable.

I would also like to thank Rich Hammer for letting me quote so freely from his article, "The Organization of Storage and Procedure Calls on an M-Code Machine." It made the writing of one subsection of my thesis relatively easy.

I should of course point out that none of these individuals can be held responsible in any way for any shortcomings of either my thesis or the code. All final decisions as to what was to be included anywhere in the document were my own, as is the responsibility for any errors.

CONTENTS

LIST OF FIGURES

CHAPTER I.
INTRODUCTION

1.1. THE PROBLEM

Courses in operating systems are generally limited to a passive presentation of material. The theory of operating systems can be taught, and specific operating systems can be chosen as case studies; but there is little opportunity for active development, for hands-on *design* or *modification* of operating systems, especially in a first, one-semester course. This is not simply a matter of teaching approach. The design of a complete operating system is just too large a task for a first course in operating systems, and existing operating systems are generally too vital to the computer community to allow students the privilege of playing with them, and too large and complex to make the playing worthwhile.

This situation is no less a fault for its being intrinsic to the subject matter. The complexities, the innermost workings, the very scope of the design of an operating system–none of these can be truly appreciated or understood by the student who has not actually experienced the design process. Yet the task of operating system design remains too large to ask of a student.

This thesis is concerned with one possible solution to the problem. An overview of the solution is presented in this chapter, as well as a brief review of previous work in the area. Later chapters discuss in more detail the two operating systems that implement the solution. Sections 3.2 (UNIBATCH Overview) and 4.2 (MULTIBATCH Overview) include descriptions of the major data structures, general operation, and salient features of each system. In addition they give a historical perspective on how each operating system was developed. These sections, along with the suggested assignments (sections 3.3 and 4.3), and the Conclusion (Chapter 5) should be of interest to most readers.

Chapter 2 (the machine description), and section one of Chapters 3 and 4 (the machine and machine interface descriptions) are probably of interest only to those who will be reading the code. Students may find these sections useful reference material when trying to understand how interrupt handling and supervisor calls are implemented. Beginning students should realize, however, that this document contains material for their instructors, as well as for students; they should not expect to be able to understand all of it until after they have learned something about operating systems, and spent considerable time studying the code.

1.2. APPROACH

There is no way to make the task of designing an operating system easier, except by specifying the operating system to be so small and simple that the endeavor of designing and building it becomes practically worthless. However, an *environment* can be created, complex in itself, but within which appropriately scaled down activities can still give the flavor and some of the experience of actual operating system design. The situation is analogous to that in the physical sciences. No one would suggest that beginning physics students learn for themselves the entire structure of physics, designing their own experiments to light their way; rather, selected experiments are provided to demonstrate first-hand how certain physical principles are manifested. The student need not discover new principles for himself, but he must *reenact* experiments which demonstrate the existence of those principles.

The creation of the environment within which operating system ''experiments'' are possible is part of the SoftLab project, a group project designed to provide the capability for experimental work in courses on architecture, translators, operating systems, and database management systems, and for research in these areas. The purpose of this present portion of the project is to provide a pair of operating systems–the nucleus of an entire family–which will support experimentation and investigation of operating system design in the manner of a laboratory science. In keeping with the goals of SoftLab, these operating systems are understandable and adaptable; they use only widely available hardware and software; and they are as portable as possible.

These operating systems have been designed to enable tracing the growth, development and relative importance of the different task-oriented units (the scheduler, for example, or the interrupt handler) from the smaller operating system to the larger. They have also been constructed with the idea that additional operating systems, utilizing state of the art techniques, will be added in the future, illustrating ever more difficult problems and complicated mechanisms. The two members of the family developed for this thesis are *UNIBATCH*, a very simple batch, uniprograming system, easily understandable in its entirety; and *MULTIBATCH*, a more elaborate and powerful multiprograming system with spooling, where a detailed understanding of the complete operating system is not necessary for performing experiments and investigations on selected parts.

Since these operating systems are to serve as a theater for experimentation and study, and the source code of the operating systems is to be available for this purpose, it is vital that the code be clear and well-documented. The included experiments and assignments often focus on one segment of an operating system, sometimes with the intent of replacing that segment with another segment designed to accomplish the same task in a different or more efficient manner. The operating systems are therefore modular, so that one section can easily be changed without incurring unwanted side effects throughout the rest of the system. Toward this end, the language chosen for writing the operating systems is Modula-2, a language based largely on Pascal, but with even more support for information hiding and separation of function.

Modula-2 contains strong type-checking facilities, is highly structured, and possesses all the features of a good teaching language which has made Pascal so popular in academic circles--yet at the same time it has low level capabilities necessary for systems programing [16].

Each operating system requires certain hardware features to support it: reserved memory locations, the number and types of registers, and input/output details. Included as part of the description of each operating system, therefore, is a specification of that system's *machine interface;* that is, a description of those hardware features which are visible to the operating system, and which any machine wishing to run the operating system must possess. A part of this interface will consist of a "machine module" which serves as a definition of the machine from the operating system's point of view. This module, called **`LocalSystem`** in both systems presented here, associates variables with reserved memory locations, defines some basic machine data structures, and provides a set of machine instructions which can be used by the rest of the operating system.

Each operating system may be run on a simulated machine having the characteristics described in the machine interface specifications. An overview of these simulated machines is also supplied as part of this document. These machines will be part of an *Integrated Instrumentation Environment* (IIE), also part of the SoftLab project [13]. This environment supports the monitoring of device and CPU utilization, so as to provide results for the experiments and investigations conducted on the operating systems themselves. The IIE is a vital foundation to the project, for without results there is no way to judge the quality of design changes, and experimentation loses most of its value.

Since the purpose of the entire SoftLab project is to provide an environment where experimentation is practical and fruitful, each operating system description is accompanied by a list of suggested experiments and assignments. It was with this in mind that room was deliberately left in the operating systems for improvements. It is hoped that the decision of which features were essential to the operating systems, and which could be omitted in favor of inclusion among the assignments, was made wisely.

Listings for the source code of both operating systems are contained in the appendix. The final versions presented are the work of the author, though some modules are partly the work of others (see Acknowledgements). Both operating systems compile correctly in the form given; unfortunately, the machine on which they were to run has not been completed yet, and so it has proved impossible to completely debug them.

It is expected that students wishing to use this system be competent programers in at least one high level language, that they have some knowledge of machine architecture, and that they be familiar with data structures. There is no need for them to have a solid background in operating systems; that would defeat the purpose. However, this document and the accompanying operating systems are intended to *aid* the student in learning the subject of operating systems, not to teach it outright. It will be most effective, and most understandable, if used as an adjunct to a standard operating systems course.

1.3. PREVIOUS WORK

There are several publications which deal with operating systems in a laboratory environment [1, 5, 6, 7, 8, 9, 11, 12, 15]. This section will briefly discuss ways in which these documents, though successful in their own goals, do not meet the SoftLab objectives; and what their goals actually are. Two texts whose goals seem closest to those of SoftLab will be singled out for more detailed analysis.

Some of the cited publications are textbooks, whose only mention of a software laboratory environment lies in their appendices (Holt [12], Shaw [15]); some others are technical reports (Corbin [7], Corwin [8]). In either case, there simply is not enough space in these publications to do anything more than describe what might be a promising system. Other texts, though devoting a large portion to the subject, nevertheless provide at most a small portion of the relevant code (Brinch Hansen [1], Holt [11]). Some publications suffer in general usefulness by developing a language specifically for the purpose, rather than employing an existing one (Brinch Hansen [5], Halstead [9]); most describe operating systems which are overly simplistic (Brinch Hansen [5], Corbin [7], Corwin [8], Halstead [9], Holt [12], Shaw [15]); and design alternatives are suggested only in the exercises of Comer [6], although Holt [11, 12] presents a number of assignment questions which could be made into good experiments if only the code for the operating systems were included.

Of course, the aims of most of these works differ from one another, and from the aims of SoftLab in general and this project in particular. For example, Brinch Hansen [1] seems most interested in presenting an exposition of Concurrent Pascal, and showing how an operating system can be written in a high level, structured, concurrent language. Holt [11, 12] seems even more interested in demonstrating the value of structured concurrent programming, though an appendix in the earlier work includes a very simple operating system, complete except for references made to procedures discussed earlier in the text. The later work, though similar in intent, provides no code; and the exposition of other topics (such as the Unix operating system) causes it to lose focus. Both of these books, however, raise a number of questions in their exercise sections as to how operating systems could be constructed differently from the expositions given, and are therefore closer to the experimental spirit of SoftLab. Shaw [15] is first and foremost a textbook. Its appendix contains a description of a project to construct a multiprograming operating system (as well as a simulation of the machine on which it will run); but the purpose of this is mostly to apply the concepts learned in the book; no code is given, nor are any alternatives supported.

Given such differences of purpose, it is not to be expected that these publications would fulfill all the stated SoftLab objectives of being understandable, adaptable, reliable, portable, and available. However, two texts have goals which approach the aims of SoftLab closely enough to merit closer scrutiny: Halstead's *A Laboratory Manual for Compiler and Operating System Implementation* [9] and Comer's *Operating System Design, The Xinu Approach* [6].

Halstead's work is obviously directed towards much the same goals as the SoftLab project. However, the approach is to construct a complete compiler and a complete operating system, rather than to experiment with portions of ones already provided. The language used is Pilot, a stripped-down version of Neliac, which in turn is a real-time systems language derived from Algol. The reason given for using such a stripped-down language is that its design will be easy to extend; but the language is so simple that it becomes difficult to read and write–attributes which should be of prime concern in a system devoted to education. Also, although the systems are claimed to be modular by virtue of consisting of subroutines, all variables are global. The operating system is incapable of certain features necessary in any reasonable system: memory management, for example, or multi-level scheduling. Finally, there are no suggestions or support for operating system experimentation, except to implement the system on a machine other than the Univac 1108, since that is the implementation given in the book.

Comer's book is more successful in fulfilling the goals of SoftLab. It is a much larger book than Halstead's, yet deals only with operating system design. The operating system it presents is highly functional, though it assumes only a small amount of memory and is weak on protection. The language used is C, which is easily accessible, though it does not have strong support for modularity and can sometimes be difficult to read. The code itself is fairly clear, however, and the exposition in the text is quite good. In spite of all this, the book is a textbook on operating system design, not a laboratory manual. The XINU system is presented, step by step, with all its code; but it is presented more or less as a finished product. Design alternatives and extensions are suggested in the assignments, but the discussion in later chapters always assumes the original configuration; and perhaps most significant, there is no provision for analyzing the effects any such alterations have on performance.

1.4. SUMMARY

The size of operating systems makes their design and development unsuitable for assignment in operating systems courses. As a result, the absence of any kind of hands-on experience with operating systems is a weakness common to almost all first courses in operating systems.

To help alleviate this problem, a pair of operating systems intended for pedagogical use has been designed, implemented and documented. These operating systems, while preserving a "family" resemblance, illustrate different approaches to operating system design, so that the instructional benefits of studying the different operating systems will be complementary.

Each operating system is easily modifiable to allow for meaningful experimentation with regard to the extension of function and enhancement of performance. The different functions have been compartmentalized into *modules,* to facilitate making selective alterations on operating system functions. For this purpose, and because of its low-level capabilities, the language Modula-2 has been chosen as the language in which to write the operating systems.

The operating systems, by their differences, demonstrate large scale design alternatives. In addition, each operating system comes with a list of experiments and assignments, which demonstrate design alternatives on a smaller scale. In order to monitor the changes in operating system efficiency which occur as the result of executing these assignments, the operating systems may be run in an Integrated Instrumentation Environment.

It is hoped that these operating systems will provide an environment in which students can gain hands-on experience of operating system design, without the laborious effort of building their own operating systems from scratch.

Contained in this document are:

- An overview of the simulated machine on which the operating systems will run (Chapter 2).
- A machine interface specification for each system (section 1 of Chapters 3 and 4).
- An exposition and description of the organization of each system (section 2 of Chapters 3 and 4).
- A set of suggested experiments and assignments for each system (section 3 of Chapters 3 and 4).
- The code for the two operating systems (appendices).

CHAPTER II.

OVERVIEW OF THE MACHINE SIMULATION

MULTIBATCH and UNIBATCH do not run on exactly the same machine, but the variations are slight. The machine simulations used in the SoftLab project are versions of a basic M-Code interpreter, based on Niklaus Wirth's Lilith engine [14]. This interpreter effects a virtual machine called the *M-Code Machine*. The details of its workings which affect most directly the operation of UNIBATCH and MULTIBATCH can be collected into two broad categories: *storage* and *interrupt handling*.

2.1. STORAGE

Most of this section on machine storage is taken from an unpublished SoftLab internal working document called "Organization of Storage and Procedure Calls on an M-Code Machine," by Richard Hammer [10].

2.1.1. Registers

The M-code machine uses the following registers:

PC: program counter.

IR: instruction register.

F: code frame base address: this is the start of the code frame for module 0, the main module of the program.

G: data frame base address: this is the start of the data frame for module 0.

S: stack pointer: this points to the top of the process stack.

H: stack limit address: this is the address which S cannot exceed.

L: local segment address: this points into the process stack and always points to the bottom of the currently-active activation record.

P: process state address: this holds the location where the process state is saved when the process is not running.

M: process interrupt mask: this is a bitmask of sixteen bits, used to disable certain interrupts during the execution of privileged modules.

T: segment table base address: this points to the first address in the segment table (or table of modules) for the currently active process. (In early versions of the M-code machine there will be only one segment table, and this register will always contain 40B (octal), the address of that one segment table.)

For those registers that contain addresses (F, G, S, H, L, P and T), the addresses are always absolute. The content of the PC, however, is relative: the PC contains the number of bytes beyond the start of the current code frame, that is, beyond F.

In addition there is a 16-register expression stack which is used to store the intermediate results of operations, and on which parameters are placed during procedure calls.

2.1.2. Reserved Locations

The first 40B locations in storage are "reserved locations." These locations contain the trap and interrupt vectors, and data needed for bootstraping. These reserved locations are illustrated in Figure 1.

Locations 0-2. Each program module will have in storage a data frame and a code frame. The data frame contains a pointer to the codeframe, an initialization flag which indicates whether that module has had its initialization section executed, a pointer to the string table, and an array of global variables. The code frame contains the M-code instructions for that module.

Module 0 is unique in that the data frame for Module 0 resides in the first three locations in storage, to enable bootstrapping. Address 0 therefore contains the address of the code frame

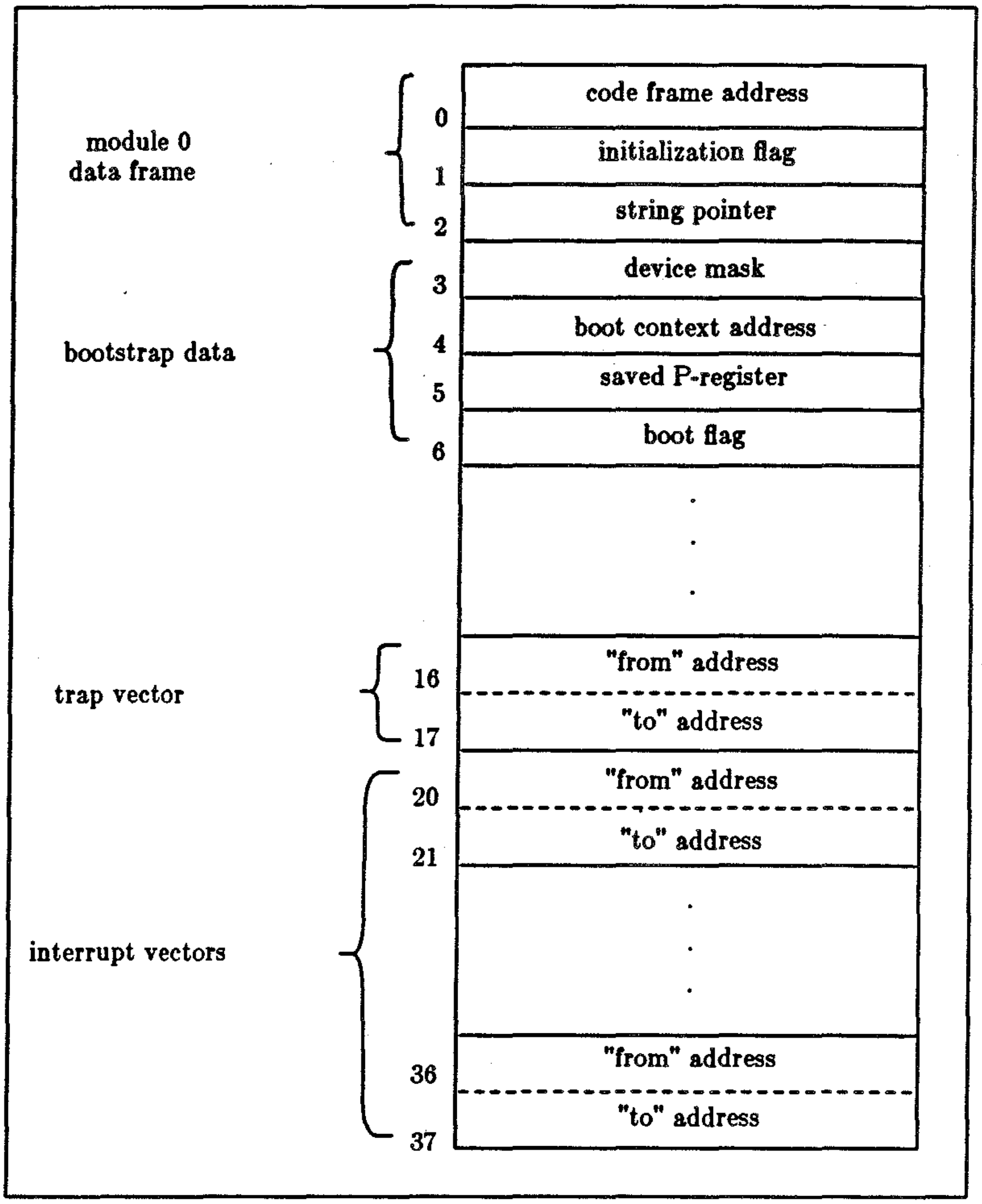

Figure 1. Reserved Locations.

for module 0, address 1 contains the initialization flag, and address 2 contains the string pointer. (Module 0 is also unique in that it has no space for global variables; thus its data frame occupies only three words.)

Locations 3-6. Address 3 contains the device mask, used for masking interrupts. Setting any bit of this mask to 1 disables the corresponding device interrupt. Address 4 contains the address of the *context*, or process state, for the start-up process; that is, the value used to initialize the P register during bootstrapping. Addresses 5 and 6 contain other bootstrapping data: a saved P register and a boot flag.

Locations 16-17. These two words constitute the trap vector, which contains two addresses: the first is the address of the context of the interrupted process, the second is the address of the operating system's trap handling procedure. The first is set whenever an interrupt occurs, the second is set only once during system initialization.

Locations 20-37. These addresses contain eight interrupt vectors. Like the trap vector each interrupt vector consists of two words, a context address and procedure address.

2.1.3. Main Storage

Main storage holds a process's segment table, its data and code, its stack, and its heap.

The segment table contains the addresses of the segments for the modules. Each module has one segment, consisting of a data frame and a code frame. The third address in the segment table, for example, will be the address of the start of the data frame for module 3. In general there will be one segment table for each process. The starting address of the segment table for the currently active process will be held in register T.

The segment table for process 0 will occupy addresses 40-177. By convention, the first location in this segment table contains 0, because the address of the data frame for module 0 is 0.

In general, except for module 0, all modules will have their data and code frames consecutive in memory (thus constituting a segment), and all segments will immediately follow the segment table. This constitutes the process's *static* storage.

At the base of a process's stack is its context, or saved state. Figure 2 shows the contents of the context. The capital letters refer to the values of the registers of the same name. The context holds, in order, the address of the current dataframe, the address of the base of the current activation record, the PC, the process interrupt mask, the stack top, the stack limit, the error code (in the case of a trap), the error trap mask, and the address of the segment table.

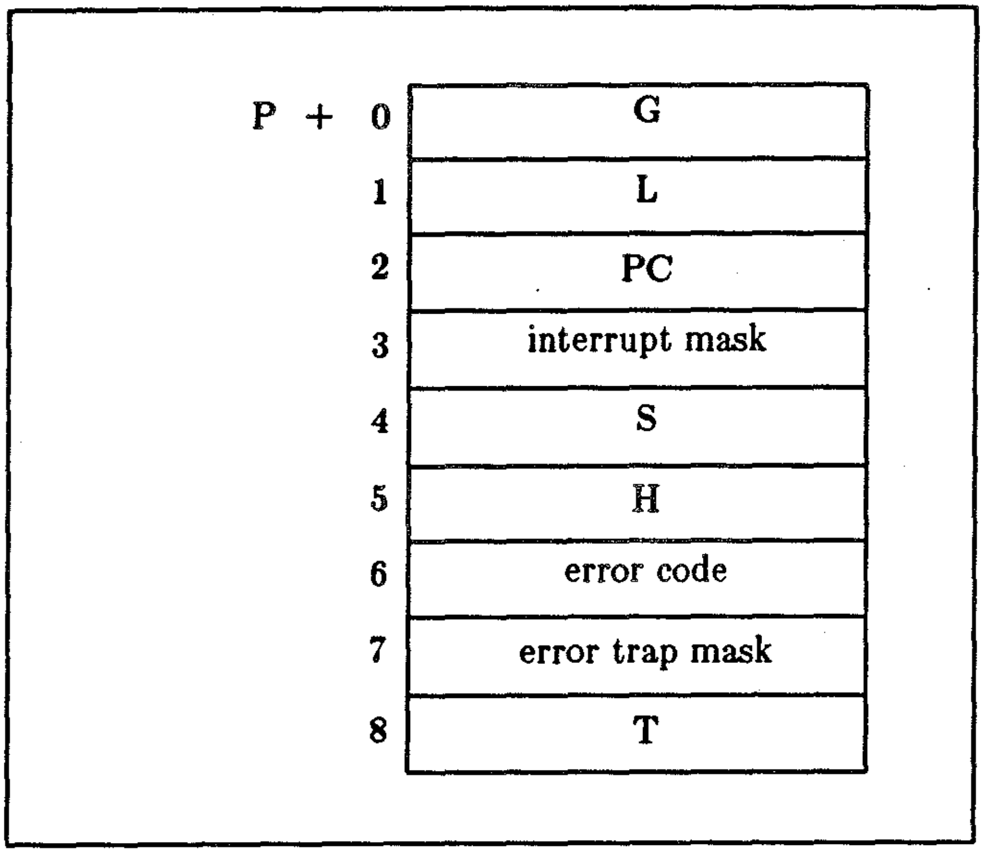

Figure 2. Process Context.

Immediately following the process's context is its run-time stack, holding the activation records, or *dynamic* storage, for all procedures currently in progress. The stack grows upward (from smaller addresses to larger), while the heap, starting at the largest address available to the process, grows downward toward the stack.

Figure 3 shows the overall view of main storage with a single process.

2.2. INTERRUPTS

The M-code Machine architecture contains an interrupt mask register, known as the M register. The M register is set by the module priority (if any) given in the declaration of the module to which the currently executing procedure belongs (if no priority is given in the declaration, then the procedure's priority is the same as that of its calling procedure).

In addition, there is a set of request lines, numbered 8..15, implemented as a BITSET register (ReqLines). Each device will be associated with one request line, or one bit in ReqLines, which will be set to 0 when the device wishes to raise an interrupt. The register REQ will be set to TRUE at the start of each interpretation cycle when at least one of the unmasked request lines is low. A third register, ReqNo, holds the number of the highest bit missing from the set ReqLines; that is, the highest priority line requesting an interrupt. The value of REQ is checked at the beginning of every interpretation cycle; when it is TRUE, the interrupt indicated by ReqNo is handled.

Each device is given one interrupt line. Traps are simply software interrupts, and all traps are grouped together on a single interrupt line (line 7). The trap code is stored in the context (in the error code field).

When an interrupt is processed, the register values are first saved in the process's context, and the address of the context is stored in the interrupt vector. The saved context of the operating system, always stored in reserved location 5, is loaded into the registers, with the value of

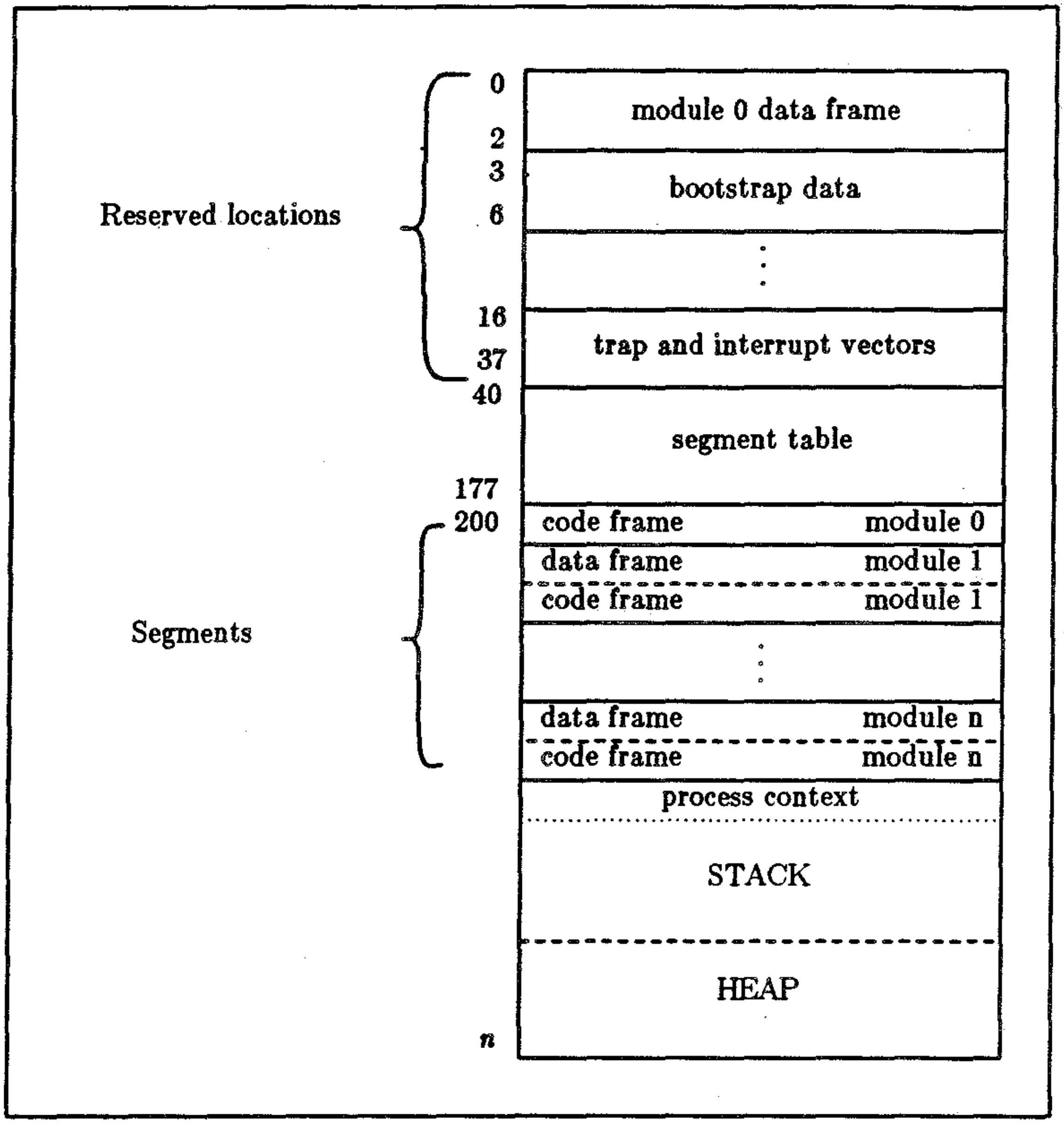

Figure 3. Overall View of Main Storage.

the PC coming from the appropriate interrupt vector. The interrupt handling routine then begins executing. At its conclusion, the context switches back to the context saved in the interrupt vector, and the interrupted process begins where it had left off.

CHAPTER III.

THE UNIBATCH OPERATING SYSTEM

3.1. MACHINE INTERFACE

3.1.1. Introduction

Every operating system, if it is to deserve the name, must be able to operate some particular machine. Certain features of the operating system are therefore bound to reflect features of the machine for which it is written. On the other hand, not *all* machine features will show up. Indeed, the fewer the machine dependent features of the operating system, the more portable it is. Machine dependencies of an operating system can be divided into two basic types: 1) the operating system directly *references* features of the machine, e.g., sets a register value; or 2) the operating system makes *assumptions* about the machine's capabilities, and therefore takes no action on tasks which it expects the machine to handle, e.g., recognizing the occurrence of a device interrupt.

While both UNIBATCH and MULTIBATCH run on an M-Code machine, differences in the operating systems' capabilities demand slight alterations and tailorings of that machine. The machine as used by UNIBATCH shall be known as the *UniBatch Machine*.

Most *direct* references to features of the UniBatch Machine are contained in the module **`LocalSystem`**. These features consist of five dedicated memory locations--two for I/O, one for bootstrapping, one for saving the state of the operating system, and one to hold the address of the interrupt handling routine–and a set of "procedures" which are in fact machine instructions available to the operating system. One of these, `SetBoundsRegister`, directly sets the value of a machine register known as the "Bounds Register". In addition, UniBatch controls two peripherals to the UniBatch Machine. These are a cardreader and a lineprinter and are made explicit in the machine instructions of `LocalSystem`.

The only direct reference to the machine outside of `LocalSystem` is in `ContextHandler`, where the `Context` type is defined to match exactly the process context of Figure 2.

Implicitly assumed by the operating system is the machine's use of the new Bounds Register to protect the operating system from user programs; also assumed is the use by the machine of two values in a process's context, presumably by loading them into registers: a *program counter* (PC), and an interrupt mask containing a *mode bit*; this last is to allow the machine to prevent user programs from having access to most of the machine instructions in `LocalSystem`. In addition, it is assumed that the UniBatch Machine can detect the raising of a trap or interrupt, and call the interrupt handling routine with the appropriate parameters.

Virtually any machine with all of these characteristics should be able to run UniBatch. The M-code machine's CPU needs to be upgraded only in the following ways: 1) a Bounds Register must be added, 2) the interrupt mechanism must be altered to make provision for parameters to the interrupt handling procedure, and 3) it must use the mode bit (placed in the high bit of the interrupt mask) and the Bounds Register to protect the operating system against the user program. This chapter discusses the two modules `LocalSystem` and `ContextHandler`, which form the major interface between UniBatch and the UniBatch Machine; the peripherals controlled by UniBatch; and the method by which UniBatch expects interrupts to be implemented.

3.1.2. Machine Modules

3.1.2.1. `LocalSystem`

Storage

The memory locations 4, 5, and 17 are reserved to be the boot process context address, the saved P register (or operating system context address), and the trap handling procedure address of the trap vector. These meanings are confirmed in **`LocalSystem`**, which uses Modula-2's direct addressing feature to declare them as **`bootcontext`**, **`OScontext`**, and **`interrupthandler`**. In addition, locations 10 and 11 are defined by **`LocalSystem`** to represent the one-byte registers of the card reader and line printer, called **`inputbuffer`** and **`outputbuffer`**. These locations are in reserved memory, but were given no special function by the M-code machine. **`Inputbuffer`** holds the last byte read by the card reader, and **`outputbuffer`** holds the next byte to be written by the line printer.

The presence of a Bounds Register is required explicitly by the presence of the machine instruction **`SetBoundsRegister`**. This is the only register which the operating system actually requires the UniBatch Machine to possess. All user memory references must be above the value in the bounds register to be legal.

Instructions

The procedures **`ContextSwitch`**, **`Read`**, **`Write`**, **`SetBoundsRegister`**, and **`Trap`** can all be implemented as otherwise unused M-code instructions. **`ContextSwitch`** (#246 = **CNTX**) is used to return from an interrupt by restoring the old context; **`Read`** (#240 = **READ**) is used to send a non-blocking request to the card reader to begin reading the next character; **`Write`** (#241 = **WRITE**) is used to send a non-blocking request to the line printer to begin printing the next character; **`SetBoundsRegister`** (#214 = **SBR**) is used during initialization to set the bounds register; and **`Trap`** (#304 = **TRAP**) is used to handle software interrupts–it stores the current context and raises an interrupt signal.

The implementation module of **LocalSystem** is written with empty procedures to permit compilation, both of this module and of any other which imports from it; the linker will have to be altered to substitute the correct M-Code instructions for the named procedures. At a later date, it may be possible to alter the compiler to recognize this special module directly.

3.1.2.2. ContextHandler

Storage

The **Context** record is a data structure which stores the register values representing the working environment and location of a program in order to restart the program after a return from interrupt. This type is defined in **ContextHandler** to match the process context of the M-Code machine. However, of all the fields in the **context**, the operating system regularly accesses only the **PC**. Once, during initialization, it must set all the values of the single user **context**; but the only value which really concerns it at that time is the **interrupt mask**, where setting the high bit to 1 (the mode bit) is a signal that the machine should not let a user program execute any **LocalSystem** instruction but **Trap**. The user context has its mode bit set to 1, while the operating system has its mode bit set to 0.

Procedures

The procedures **NewContext**, **SetInterruptHandler**, **SetPC**, **SVCArgument**, and **SwitchContext** are codable in Modula-2. **NewContext** creates a new context for the user during system initialization; **SetInterruptHandler** stores a pointer to the interrupt handling routine in memory location 17, **interrupthandler**; **SetPC** can change the **PC** in the most recently interrupted context; **SVCArgument** retrieves from a user's stack the argument to a supervisor call which caused the trap; and **SwitchContext** returns to the most recently interrupted context.

3.1.3. Devices

UniBatch can send a start signal to the card reader only by issuing the `Read` instruction, and to the line printer only by issuing the `Write` instruction. These each send a non-blocking request for action to the appropriate device, so that the device begins acting concurrently with the CPU. It is up to the operating system to ensure that at most one request is sent to each device at a single time–most easily by not sending a request until fielding a "device completed" interrupt. Each device handles one byte at a time.

3.1.4. Interrupts

The UniBatch Machine handles interrupts exactly as the M-code machine does, except that there is only one interrupt handler, rather than one for each device and one for traps. Also, the machine must place the interrupted context and the reason for the trap on the expression stack, where they will be picked up as parameters by the interrupt handling procedure.

Regardless of what line the interrupt comes in on, it must be fielded by the procedure whose address is in location 17 (`interrupthandler`), placed there during system initialization by a call to `SetInterruptHandler`. If a trap caused the interrupt, the effect of the machine's trap instruction must first be to place the reason for the trap (which will be the ordinal of a value of type `Exceptioncode`, defined in `LocalSystem`) on the machine's expression stack. If the interrupt was due to a device, then the request line number (8 for the cardreader, 9 for the lineprinter) should go on the expression stack. In either case, after the registers are stored in the context, the address of the context (in the P register) should be placed on the expression stack. Then the interrupt routine should be called. It will use the values on the expression stack as its parameters.

At the close of the interrupt routine, control will be switched back to the context that was passed as a parameter, by a call to `ContextSwitch`.

3.2. OVERVIEW

3.2.1. Introduction

UniBatch is a toy operating system intended to be the first system for which students are given actual code, and on which they may make alterations. It is designed to be a minimal example of a complete operating system, exhibiting all of the major operating system functions in the simplest possible way. As such, it cannot be considered to be useful *as an operating system*; however, the simplicity should promote its effectiveness *as a teaching aid*.

UniBatch processes a stream of batch jobs, one at a time, which have been compiled into M-code, and entered on punched cards, one M-code instruction per card. Any operands will immediately follow the instruction on a separate card, one card for each operand. Since UniBatch provides no file system, all input must be supplied on cards immediately following the job, and all output will go directly to a line printer. The card reader and line printer are the only peripherals which UniBatch supports.

The JCL for the system is extremely simple: each new job is preceded by a card containing the character ''/'', and the code for each job is immediately followed by a card containing the character ''$''. If there is any input data, it will immediately follow the ''$'' card.

This chapter discusses the operation of UniBatch, including the initialization sequence, state maintaining data structures, interrupt handling, and supervisor calls. It goes on to present some salient features of the system as a whole. Then it discusses the organization of UniBatch and the modules which make up the operating system.

3.2.2. Operation of UniBatch

UniBatch, as the operating system, is loaded into the UniBatch Machine in the lowest unreserved locations: its segment table starts at octal location 40, and the code frame of module 0 immediately follows that at location 200. (Remember that the module 0 dataframe, when only one program is loaded, is placed at location 0 rather than immediately before the corresponding code frame.) At location 4 is the address of the operating system's context. The context will immediately follow the end of the last module's code frame.

3.2.2.1. Initialization

At boot time, all appropriate values in the boot context will have been set correctly. When the boot is initiated, control passes to the starting location of the operating system. As with all Modula-2 programs, the initialization sections of all modules in the operating system other than the main module are executed first. These set up the data structures and place the address of the interrupt handling routine in the `interrupthandler` variable. After this, the module `UniBatch` itself begins executing.

UNIBATCH's first action is to save its context; that is, assign the value of `bootcontext` to `OScontext`, the location the M-Code machine calls the saved P register.

Next the cardreader must be started. This is done by calling the `LocalSystem` machine instruction `Read`. When a character has been read, an interrupt will cause the character to be processed and start the card reader going again.

Finally, a user context must be created, and a user program set running. The context is initialized to start at a fixed location, be of a fixed size and have a fixed-size stack. All user processes will use this same context, since only one can run at a time. After the context is created, the operating system requests a user job to be read in, and when this has been accomplished it switches to the user context.

3.2.2.2. Maintaining State: the `Context`

During normal operation, the CPU will constantly be switching between the operating system and the current user process. The mode of the controlling process, and the PC and stack status of the dormant process, must be maintained and accessible at all times. This information is contained in the `Context` data structure.

The `Context` holds the state of a process. It is not kept up to date in the currently running process, but is updated when the process is interrupted. It keeps track of the location of the process in memory, the status of its activation record stack, its mode bit, and the reason for any traps. The `Context` is a hidden data structure, defined and exported by `Context-Handler`; and it is accessed through the pointer type `ContextID`, and exported procedures.

3.2.2.3. Interrupt Handling

When an interrupt occurs, the register values are saved in the `Context` of the current process. The register value holding the address of the `Context`, however, is placed on the machine's expression stack (used, among other things, for passing parameters). Then that register is loaded with the address of the operating system's `Context`, and the values from that `Context` are used to refill the other registers. The value of the `PC` is taken from the stored address of the single interrupt handling routine. The reason for the interrupt is also placed on the machine's expression stack.

Execution of the interrupt handling routine then begins. The type of interrupt and previous context are taken as parameters, the first of which is used to index into a case statement of possible interrupt handling responses. When the interrupt has been handled, the operating system switches back to the former context.

3.2.2.4. Supervisor Calls

Supervisor calls are implemented as a form of interrupt. Each supervisor call first calls an intermediate procedure whose action is simply to call `LocalSystem`'s `Trap` procedure with `SVC` as a parameter; however, the intermediate procedure itself takes as parameters the type of supervisor call and the parameters to it. This places all the necessary information in a known location on the process stack: the top activation record will be the one for `Trap`, and the fourth location above the base of the preceding activation record will be the type of supervisor call. The parameters to the supervisor call will be immediately above that.

The call to `Trap` raises a trap interrupt. The interrupt handler knows from the parameters that the trap was a supervisor call. It then uses **`ContextHandler`**'s **`SVCArguments`** to obtain the type of supervisor call and its arguments. Finally, it takes the appropriate action.

The user may invoke supervisor calls to read from input or write to output.

3.2.2.5. User Processes

Each user job is assigned in turn to the same context. As a result, each job has the same amount of space allotted for its code, and for its stack and heap. It is possible to alter the system to make the values specifiable through JCL extensions.

When a user process obtains the CPU, it keeps it until it is finished, is aborted, or is interrupted. However, all interrupts return the CPU to the same interrupted process when complete, unless they result in termination of the process. Hence, a user process that goes into an infinite loop will hang the machine. No other job is even read in until the current user job is completed or canceled. During I/O requests the operating system merely loops until the appropriate device is free.

3.2.3. Salient Features

UNIBATCH is most notable in the following ways:

(1) It is small. Counting only executable program instructions–that is, ignoring comments and declarations–it consists of less than 200 lines of code. Figure 4 contains a chart of the size of each module of UNIBATCH, with and without comments (but including declarations), and the size of M-code produced.

		line count	without comments	size of M-Code (bytes)
`ContextHandler`	*def*	88	19	320
	mod	141	93	
`IO`	*def*	81	20	567
	mod	152	90	
`InterruptHandler`	*def*	50	11	1308
	mod	176	112	
`Loader`	*def*	34	12	564
	mod	129	77	
`LocalSystem`	*def*	85	28	0
	mod	39	20	
`Scheduler`	*def*	28	8	358
	mod	58	30	
`SVCalls`	*def*	32	6	259
	mod	50	20	
`UniBatch`	*mod*	58	25	258
Total		1201	571	3815

Figure 4. The Size of UNIBATCH.

(2) **It includes most of the major functions of an operating system.** In spite of its small size, it loads jobs into main memory, schedules processes, handles interrupts, processes errors, and provides protection. It does *not* have any provisions for memory management or multiprograming, nor does it include a file system.

(3) **Few of its operating system functions are more than skeletal.** **`Scheduler`** has so little to do that it could logically have been combined with **`Loader`**, and the separation was made purely on functional grounds; all interrupts transfer control to a single procedure; all detected errors result in abortion of the user's program; and protection consists

merely of checking a bounds register to keep the user from compromising the integrity of the operating system.

(1) **It makes use of concurrency in the operation of peripherals.** The "Read" and "Write" instructions in `LocalSystem` merely send a signal to start the card reader and lineprinter. The completion of the devices is signaled by an interrupt.

(2) **It is fully documented.** The definition modules, in particular, have been written so as to permit proper use of all exported procedures without the necessity of looking at the code.

3.2.4. Organization of UniBatch

This section describes the rationale used in dividing UniBatch into modules. It then briefly describes the modules, and the ways in which those modules interact with one another. It also suggests an order in which the modules should be read for maximum comprehension.

3.2.4.1. Division into Modules

UniBatch was not originally conceived as being divided into modules. The first draft of UniBatch, largely in pseudo-code and written well before Modula-2 had been chosen as the language for all SoftLab projects, fit on a single typed page and consisted of eight small procedures, with reference to five other "machine dependent actions." Although modularity was considered from the start to be an important design consideration in SoftLab, the simple use of procedures seemed sufficiently modular for such a small program. As a result, all of the procedures were available to be called by any of the others; and all of the types, constants, and variables used by more than one procedure (thankfully few) were completely global in scope.

With the choice of Modula-2 as the programing language for SoftLab, a new, more formal meaning was attached to the concept of "modularity," embodied in the Modula-2 programing structure of the *module*. This structure was imposed upon a program that had by then grown to be many times its original size. Module boundaries were drawn along lines based primarily upon *functionality*, rather than upon procedure calls or access to data; that is, procedures with similar functions were grouped together, rather than those which called mostly other procedures in the same group, or those which required access to shared data objects. (Procedures which *manipulate* a data object should, of course, be grouped together by the criterion of functionality.) These three criteria often overlap, but the point is that functionality was the criterion chosen.

As a result of this design decision, and because strict modularity was imposed upon a program that had already developed significantly without such restrictions, the import and export lists of most modules are longer than might be expected in what is still a profoundly small program. Fortunately, the smallness of UniBatch is itself a mitigating factor here: it is essentially possible to keep the entire operating system in one's head at once, and the number of interconnections between modules, therefore, by no means prohibits use of the system for its intended purpose. However, it should be noted that changing one module may require more changes in other modules than would initially be expected.

3.2.4.2. Functions of Modules

UniBatch consists of eight modules (seven not counting **`SVCalls`**). A brief statement of the function and character of each follows, in alphabetical order by module name.

`ContextHandler`–Handles all facets of switching between the context of the operating system and the context of the user program.

`InterruptHandler`–Initiates processing of all interrupts, and produces error messages when appropriate.

`IO`–Allows low level input and output: all reading is done from a card reader; all writing is to a line printer.

`Loader`–Loads program instructions into memory.

`LocalSystem`–Defines programer-available machine instructions, and dedicated memory locations. May be thought of as part of the hardware. This module and

ContextHandler together constitute a definition of the machine.

Scheduler--Handles aborts and directs the loader to load the next job.

SVCalls–Allows user access to some operating system functions. Dependent upon, but not part of, the operating system.

UniBatch–Initializes the system and starts it running.

3.2.4.3. Dependencies of Modules

The organization of UNIBATCH can best be seen by looking at its dependency graphs. In these graphs, an arrow points from each module to all modules on which it depends (that is, from which it imports).

Procedural dependencies

Procedural dependencies give the most information on layered structuring, or on which modules are low-level and which are high-level. Low-level modules support high-level ones, but not vice versa. Figure 5 contains the procedural dependency graph for UNIBATCH. Although the graph seems rather tangled, it may be noticed that no arrow points up. In other words, UNIBATCH consists of a hierarchy of four layers: at the bottom is **LocalSystem**, which calls no procedures from other modules; next are **ContextHandler** and **IO**, which import procedures only from **LocalSystem**; higher still are **InterruptHandler**, **Loader**, and **Scheduler**, which import procedures only from modules on the same or lower levels; and at the top is **UniBatch**, which as the main module can import from any module, but which does not export any procedures.

This hierarchy, although accidental in the sense that the modules were not formulated with intent to create such a hierarchy, is not really surprising in terms of functionality. **LocalSystem**, at the bottom, is an extension of the machine instruction set. Above that are **IO** and **ContextHandler**, modules which are intimately connected with hardware. Next

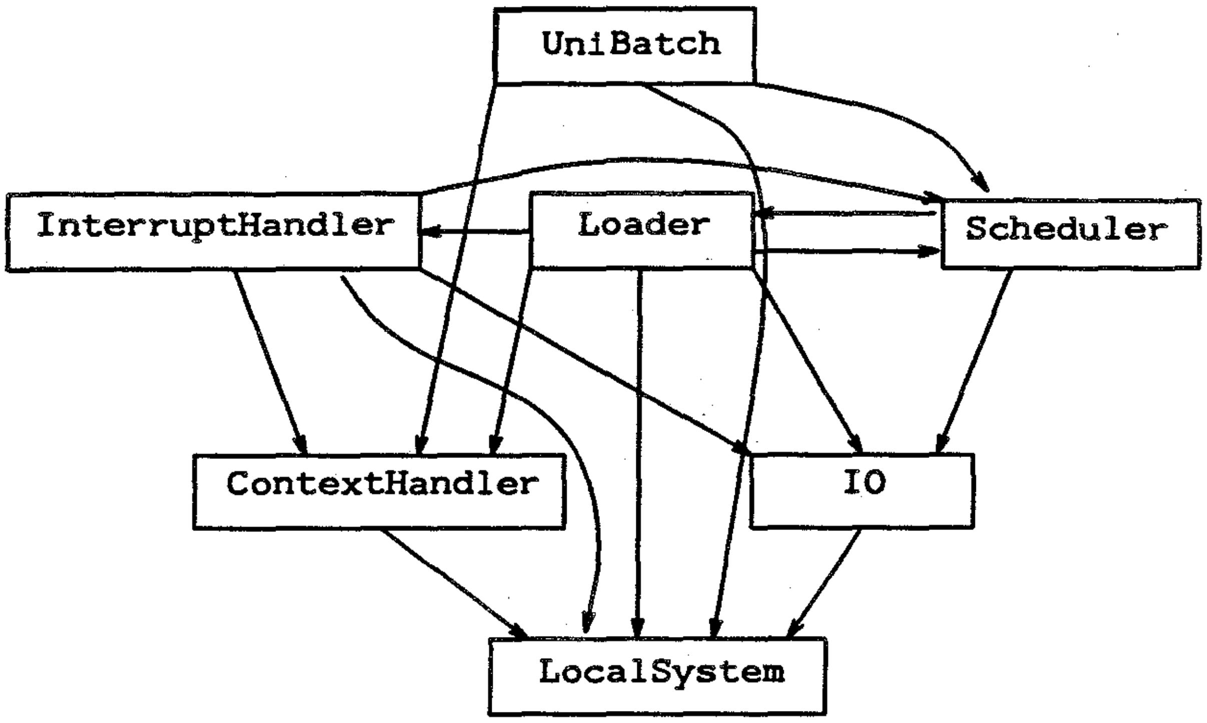

Figure 5. Procedure Dependency Graph for UniBatch.

come the modules dealing with interrupts and primitive processes–**`InterruptHandler`**, **`Loader`**, **`Scheduler`**–and at the top is system initialization, **`UniBatch`** itself.

On the other hand, it could easily be argued that **`IO`** belongs with interrupt handling; or that the **`Scheduler`**, dealing with the high level control of processes, belongs to a level higher than the **`InterruptHandler`** and **`Loader`**. It is only to be expected, given the late stage in development at which UniBatch was broken into modules, that the hierarchy indicated by the dependency graph is not absolutely ideal. In addition, the layering of the hierarchy is weak–**`UniBatch`**, for example, at the top of the hierarchy, directly calls procedures in **`LocalSystem`**, at the bottom. In other words, each layer interfaces not only with the layer directly above and below, but with *all* the layers below. The interface is thus more complicated, and less modular, than would otherwise be the case.

There is no doubt, however, that a reasonable hierarchy does exist, though it may not be the very best one. This knowledge should facilitate understanding the system.

Dependencies on variables

Dependence on variables imported from other modules is fatal to information hiding. The frequent export of variables from modules gives rise to a large number of variables whose scope is global over the entire system, and allows uncontrolled access between modules. Thus frequent dependence of modules on variables imported from other modules demolishes true modularity, leaving only its appearance.

Figure 6 contains the graph of dependencies on variables for UNIBATCH. The modules are

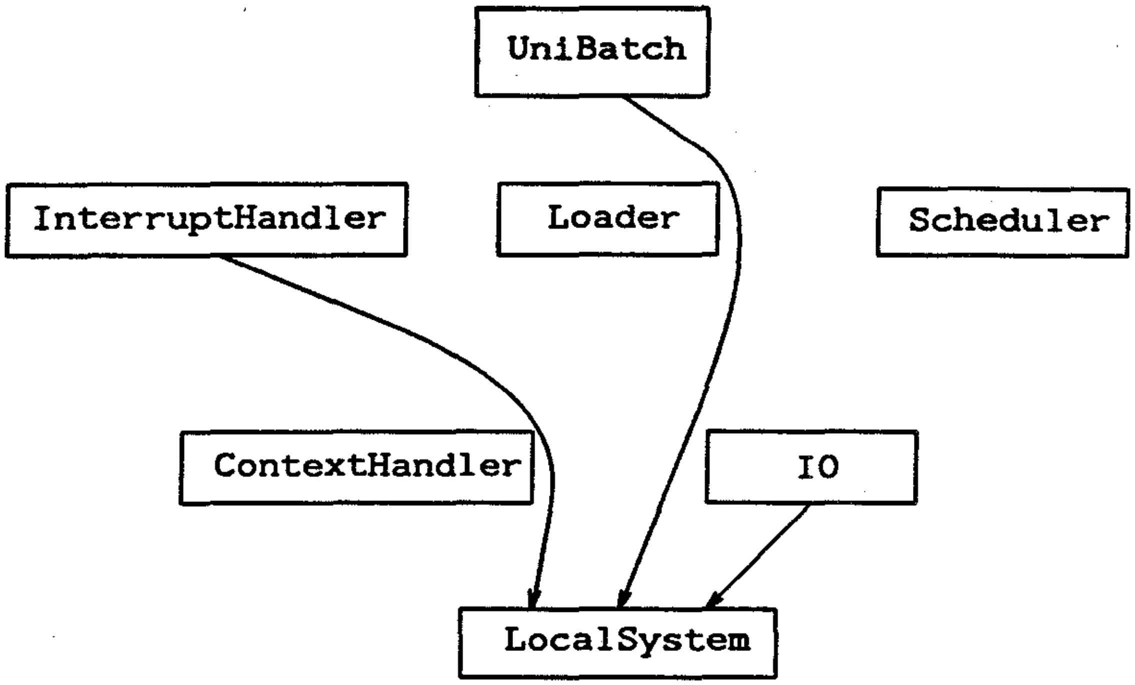

Figure 6. Graph of Dependencies on Variables for UNIBATCH.

arranged in the same hierarchy suggested by the procedural dependency graph. No other hierarchy is suggested by the graph, as only three modules import any variables; and only one, `LocalSystem`, exports any. In fact, all the variables exported are reserved memory locations, which the other procedures need to access. Two of the variables, `currentcontext` and `inputbuffer`, are never modified by the importing procedures; and two others, `OScontext` and `interrupthandler`, are set only once, during system initialization. The final variable, `outputbuffer`, is altered regularly by `IO`–but is never altered anywhere else, not even by `LocalSystem`. No unexpected side effects should occur, therefore, as a result of these few global variables.

Dependencies on constants and types

Dependence on constants and types imported from other modules is not dangerous, but one would expect the dependency graph to resemble the graph of procedural dependencies. Figure 1 contains the graph of dependencies on constants and types for UNIBATCH.

Once again, the modules are presented in the same hierarchy as that suggested by the procedural dependencies graph. For the most part, that hierarchy is supported by the current graph. There are only two exceptions: the arrows pointing upwards from `IO` and `Context-Handler` to `InterruptHandler`. These are the types `Devicecode` and `SVCcode`, and do not represent a serious problem, but they are a further indication that the hierarchy presented is a weak one.

Suggested order of reading

UNIBATCH will be most understandable if read from the bottom of the hierarchy to the top, so that most modules refer only to procedures which have already been encountered. In addition, all of the definition modules should be read before any of the implementation modules. An appropriate order would be:

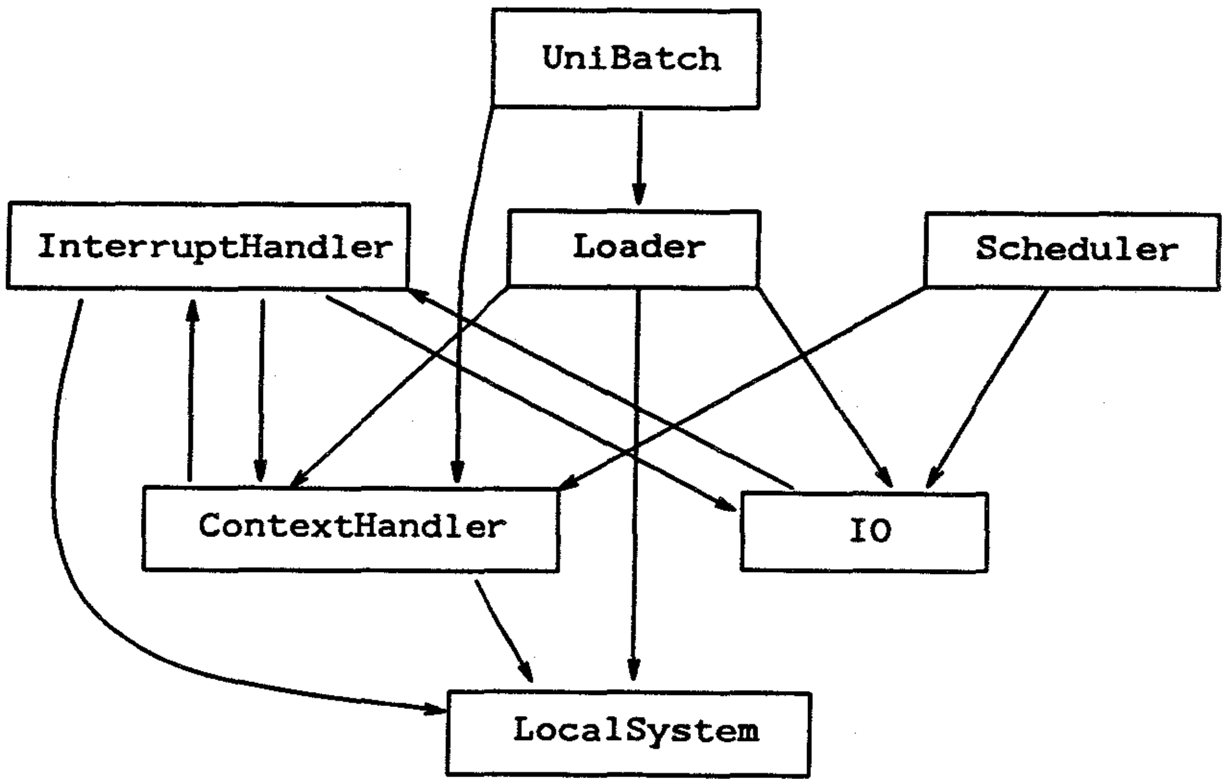

Figure 7. Graph of Dependencies on Constants and Types for UNIBATCH.

(1) **UniBatch**,
(2) **LocalSystem**,
(3) **ContextHandler**,
(4) **IO**,
(5) **InterruptHandler**,
(6) **Loader**,
(7) **Scheduler**,
(8) **UniBatch**.

Note that **UniBatch** is included at the beginning and the end; this is because, as initialization, it comes first; but as it deals only with procedures and types declared in other modules, it is best understood by reading it last. Reading it both times seems the best solution. This same strategy should probably be used on a modular basis: the initialization sections of modules which have them should probably be read quickly before the rest of the module, and again after.

No modules are dependent upon **SVCalls**, and strictly speaking it is not part of the operating system; however, as it has procedural dependencies only on **LocalSystem**, it may be read any time after **LocalSystem**.

3.3. ASSIGNMENTS

For each of the following modifications, be sure to decide how much the following issues apply, and address them suitably:

- What is the overhead of this modification?
- What are the relevant performance measures?
- Are there workloads which
    - a. dramatically increase,
    - b. dramatically decrease, or
    - c. insignificantly alter

  these measures after the modification is installed?
- Is the requested modification a reasonable one to consider?

INTRODUCTORY MODIFICATIONS

(1) Add JCL to the system, making it possible to specify the amount of space in words required for a program's code. If the cardreader reads in more than that number of cards, it should be considered an error.

(2) Add JCL to allow a user to specify the starting location of his program in memory. If that location is below the top of the operating system's stack, or if it causes the top of the user's stack to be beyond the end of memory, it should be considered an error.

(3) Add JCL to allow the specification of the amount of space allotted to a program's stack.

(4) Rewrite **`Loader`** to test only *instructions* for validity (i.e., for being valid M-Code instructions), and to test operands only for valid *format* (i.e., for containing only numerals and having a total numeric value between 0 and 377 octal). You may change the stated format of the input if you wish (perhaps to match that for MultiBatch).

(5) To **`ContextHandler`** add an instruction called **`GetPreviousInstruction`** which returns the instruction most recently executed by the user process. (Be careful not to return an *operand* to the instruction!) You will probably need a new machine instruction, **`LastInstruction`**. Would it be practical to have **`GetPreviousInstruction`** take a parameter and thus return the second to the last instruction, or the fifth to the last? Why or why not? What usefulness might such a procedure have?

ADVANCED MODIFICATIONS

(1) If a **STACKOVERFLOW** error occurs during the running of a user program, cause the operating system to increase the value of the user's stacklimit and re-execute the instruction which caused the error to occur. You will need a new **ContextHandler** routine, **GetPreviousInstruction** (described in exercise #5 above). What happens if the operating system's stack overflows?

(2) Rework UNIBATCH so that it contains no circular dependencies. What happens to the IMPORT and EXPORT lists? Does the code become more or less understandable? Is efficiency affected?

(3) Allow the user to specify an error recovery procedure that would be invoked by **InterruptHandler**, and which would allow the user to recover from certain errors, rather than necessarily having his program abort. You will need at least one new supervisor call, **SetErrorHandler**. What happens if an error occurs during execution of the error handler? How does the system know?

CHAPTER IV.

THE MULTIBATCH OPERATING SYSTEM

4.1. MACHINE INTERFACE

4.1.1. Introduction

MULTIBATCH's machine dependencies are only slightly different than those for UNIBATCH. Without exception, all direct references to the *MultiBatch Machine* are contained in the module **`LocalSystem`**. There are a set of "procedures," which are actually machine instructions available to the operating system, and fourteen dedicated memory locations–two for I/O, one for bootstrapping, one for saving the state of the operating system, and ten for an array of vectored interrupts. In addition, MULTIBATCH controls three peripherals to the MultiBatch Machine. These are a cardreader, a lineprinter, and a disk, and are made explicit in the machine instructions of **`LocalSystem`**. Finally, MULTIBATCH requires a clock, both for regular timed interrupts and for executing time-scheduled events.

MULTIBATCH makes implicit reference to the machine by assuming that the machine checks user memory references to insure that they are within a set of specified bounds. MULTIBATCH also assumes that the machine responds to device interrupts or traps by saving the state of the process and calling the correct interrupt handling procedure, and that it prevents direct user access to the machine instructions in **`LocalSystem`** (except for **`Trap`**) by checking whether the CPU is operating in user mode or supervisor mode.

Virtually any machine with all of these characteristics should be able to run MULTIBATCH. The M-code machine's CPU needs to be upgraded only in the following way: it must use the mode bit (placed in the high bit of the interrupt mask) and *two* bounds values (kept in registers) to protect the operating system from user processes, and user processes from each other. This chapter discusses the two modules **`LocalSystem`** and **`VirtualMachine`**, which form the major interface between MULTIBATCH and the MultiBatch Machine; the peripherals controlled by MULTIBATCH; and the method by which MULTIBATCH expects interrupts to be implemented.

4.1.2. Machine Modules

MULTIBATCH, like UNIBATCH, has two modules which define the machine to the operating system; unlike UNIBATCH, however, the lower level module (**`LocalSystem`**) is completely hidden by the higher level module (**`VirtualMachine`**).

4.1.2.1. **`LocalSystem`**

Storage

The memory locations 4 and 5 are reserved to be the boot process context address, and the saved P register (or operating system context address). The locations 16 and 17 are reserved to be the trap vector, and 20 through 27 are reserved to be interrupt vectors. These meanings are confirmed in **`LocalSystem`**, which uses Modula-2's direct addressing feature to declare them as **`bootcontext`**, **`OScontext`**, and the array **`interruptvector`**. In addition, the locations 10 and 11 are defined by **`LocalSystem`** to represent the one-byte registers of the card reader and line printer, called **`inputbuffer`** and **`outputbuffer`**. These locations are in reserved memory, but were given no special function by the M-code machine. **`Inputbuffer`** holds the last byte read by the card reader, and **`outputbuffer`** holds the next byte to be written by the line printer.

The **`Context`** record is a data structure which stores the register values representing the working environment and location of a program in order to restart the program after a return from interrupt. This type is defined in **`LocalSystem`** to match exactly the process context shown in Figure 2, with the addition of an "upperbound" field at location P + 9. As was mentioned above, the MultiBatch Machine is expected to make certain that user processes keep all memory references within a *pair* of bounds. The value in the "segmenttable" field can serve as the lower bound; a new field is needed only for the upper. This value can be loaded into the same Bounds Register which was added to the UniBatch Machine.

Instructions

The procedures **`ContextSwitch`**, **`DiskRead`**, **`DiskWrite`**, **`Read`**, **`Write`**, and **`Trap`** cannot be coded in Modula-2; however, all can be implemented as otherwise unused M-code instructions. **`ContextSwitch`** (#246 = **`CNTX`**) is used to return from an interrupt by restoring the old context; **`DiskRead`** (#242 = **`DSKR`**) is used to send a non-blocking request to the disk to begin transfering a given disk sector (128 bytes) to a specified memory location; **`DiskWrite`** (#243 = **`DSKW`**) is used to send a non-blocking request to the disk to begin transfering 128 bytes, starting from a specified memory location, to a given sector; **`Read`** (#240 = **`READ`**) is used to send a non-blocking request to the card reader to begin reading the next character; **`Write`** (#241 = **`WRITE`**) is used to send a non-blocking request to the line printer to begin printing the next character; and **`Trap`** (#304 = **`TRAP`**) is used to handle software interrupts–it stores the current context and raises an interrupt signal.

The implementation module of **`LocalSystem`** is written with empty procedures to permit compilation, both of this module and of any other which imports from it; the linker will have to be altered to substitute the correct M-Code instructions for the named procedures. At a later date, it may be possible to alter the compiler to recognize this module directly.

4.1.2.2. **VirtualMachine**

VirtualMachine completely hides **LocalSystem** from the rest of the operating system, and presents itself to the system as the available machine. In part, it acts as a filter for **LocalSystem**, passing only those procedures which need to be called in other parts of the operating system; but it also acts as a modifier and constructor, making the action of some procedures more useful, making variables accessible only through procedures, and developing some procedures and data structures not present in the lower-level module. In this last capacity it is most notable for being a context handler.

Storage

VirtualMachine is not responsible for declaring any storage space. It does, however, make the **Context** of **LocalSystem** into an abstract type, by exporting only the pointer type **ContextID**, which may be operated upon by the procedures **ContextBounds**, **InitOSContext**, **LowerStackLimit**, **NewContext**, **ReturnFromInterrupt**, **SetPC**, **SVCArguments**, **TrapReason**, and **UpdateContext** (described below).

Procedures

All the procedures in **VirtualMachine** are completely codable in standard Modula-2, with the extension of the machine instructions provided by **LocalSystem**. The following procedures are used in context handling: **ContextBounds** returns the high and low bounds of memory to which a process has access; **HighOSBound** returns the highest address occupied by the operating system; **InitOSContext** sets the values for stack limit and upper bound in the operating system's context, and creates a duplicate of that context to serve as the context of the null process; **LowerStackLimit** lowers the stack limit value in response to a demand for more heap space; **NewContext** creates a context for each new process; **ReturnFromInterrupt** returns to a previously interrupted context; **SetPC** can change the PC in an interrupted context; **SVCArguments** retrieves, from a context interrupted by a supervisor call, a pointer to the arguments to that supervisor call; **TrapReason** returns the error code from a given context; and **UpdateContext** updates all the absolute addresses in a context when a process has been relocated in memory.

In its capacity as a filter, **VirtualMachine** passes unchanged from **LocalSystem** the procedures **DiskRead**, **DiskWrite**, and **Trap**. However, the number of exceptions which can be passed as arguments to **Trap** is now restricted. Similarly, access to the interrupt vector array is restricted to the use of the new procedures **SetInterruptHandler**, which sets the address of the interrupt handling routine in a specified vector, and **SwitchContext**, which can change the ''formercontext'' field in the **TRAP** interrupt vector before returning from an interrupt. The two procedures **Read** and **Write** are augmented from **LocalSystem**: **Read** becomes a function which returns the value of the last character read, and **Write** takes as an argument the next character which should be written. This avoids the necessity of exporting the variables **inputbuffer** and **outputbuffer** to the rest of the system.

VirtualMachine also defines the enumerated types **Trapcode**, **Interruptcode**, and the range of valid M-Code instructions, **Mcodeinstruction**.

4.1.3. Devices

MULTIBATCH can send a start signal to the card reader only by issuing the **Read** instruction, and to the line printer only by issuing the **Write** instruction. These each send a non-blocking request for action to the appropriate device, so that the device begins acting concurrently with the CPU. It is up to the operating system to ensure that at most one request is sent to each device at a single time–most easily by not sending a request until fielding a ''device completed'' interrupt. Each device handles one byte at a time.

The disk can be signaled by either of the **DiskRead** or **DiskWrite** instructions, which also send non-blocking requests for action. It is still up to the operating system to ensure that only a single request is sent to the disk at a time; however, the disk handles 128 bytes at once, through direct memory access (DMA); that is, it can fetch those characters from or deposit them to any specified memory location without interrupting the CPU for each character. **DiskManager** is written for a disk with eight cylinders, 128 sectors per cylinder, and 128 bytes per sector; these are all constants which can be changed.

4.1.4. Interrupts

The MultiBatch Machine's handling of interrupts is exactly like that described for the M-code machine. There are four devices. The card reader, the line printer, the disk, and the clock are each given one interrupt line: line 8 to the card reader, line 9 to the line printer, line 10 to the disk, and line 11 to the clock. Traps are simply software interrupts, and all traps are grouped together on a single interrupt line (line 7). The trap code is stored in the context (in the error code field). SVCs are implemented by the operating system as traps.

It should be mentioned here that in the current state of the MultiBatch Machine, traps are handled directly by the M-code interpreter. The interrupt line is never set. This situation must be amended in the final version in order to guarantee proper execution of MULTIBATCH's trap handling mechanism.

At the close of the interrupt routine, **ReturnFromInterrupt** causes control to be switched back to the context that is stored in the interrupt vector of the interrupt that was raised.

4.2. OVERVIEW

4.2.1. Introduction

MULTIBATCH, though simple in comparison to actual operating systems, is by no means the toy that UNIBATCH is. Along with the addition of a disk and a clock, and more flexible interrupt handling, the functions of *spooling* and *multiprograming* have been added to ameliorate the single-mindedness of UNIBATCH.

MULTIBATCH, like UNIBATCH, processes a stream of batch jobs, which have been compiled into M-code and entered on punched cards. As with UNIBATCH, there is no file system, so all input must immediately follow the code. A card reader, a line printer, and a disk are the only peripherals which MULTIBATCH supports. In addition, a clock is used by the system.

Unlike UNIBATCH, MULTIBATCH does not process jobs one at a time, but switches from job to job according to a scheduling discipline. This facility, coupled with that of spooling, tremendously increases the efficiency of CPU and device utilization.

In its current form, MULTIBATCH recognizes no JCL. It is assumed that the punched cards are produced directly by the compiler, and that only correct jobs yield card decks. The code portion of each job consists of only digits and spaces–each card holds one M-code instruction, followed by any operands. The digits of an operand will be separated from those of the instruction or a previous operand by a single blank. The compiler will follow each section of code by a card containing only the end-of-text (etx = cntl-C) character, and each set of input will be followed by a similar card. Jobs with no input will be followed immediately by two such cards.

This chapter discusses the operation of MULTIBATCH, including the initialization sequence, major data structures, interrupt handling, and supervisor calls. It goes on to present some salient features of the system as a whole. Then it discusses the organization of MULTIBATCH and the modules which make up the operating system.

4.2.2. Operation of MultiBatch

MultiBatch, as the operating system, is loaded into the MultiBatch Machine in the lowest unreserved locations: its segment table starts at octal location 40, and the code frame of module 0 immediately follows that at location 200. (Remember that the module 0 dataframe of the first loaded program is placed at location 0 rather than immediately before the corresponding code frame.) At location 4 is the address of the operating system's context. The context itself will immediately follow the end of the last module's code frame. At boot time, all values in the **`bootcontext`** will have been set correctly except for the stack limit and the upper bound.

4.2.2.1. Initialization

As with all Modula-2 programs, the initialization sections of all modules other than the main module are executed before the main module begins. In MultiBatch, this action sets up certain data structures, initializes variables, and connects the interrupt handling routines with the interrupt vectors. However, several data structures depend on dynamic allocation for their initializations, and this cannot take place until after the stack limit and upper bound in the operating systems context have been set, along with the corresponding register values. Therefore, any module with such an initialization sequence must put that sequence in an exported routine which is called explicitly by the main module.

The first job of the **`MultiBatch`** module is evidently to save **`bootcontext`** in **`OScontext`** (as in UniBatch), and set the stack limit and upper bound values in the context and the registers. But MultiBatch does not have direct access to any machine registers, only to the values in a context. If it were to set values in its own context during initialization, those values would be written over at the first interrupt by the values *already in the registers*. The solution is to cause an interrupt and change the values in the operating system's context during that time. On completion of the interrupt the new values, being the values of the interrupted context, are used to fill the registers.

Another task is taken care of during this same interrupt: the creation of the context for the null process. This context is an exact duplicate of the operating system context, and its address is stored in the trap vector before the interrupt finishes, so that, in fact, the operating system henceforth runs as the null process, except during interrupts.

It is now possible to execute the initialization routines of those modules which require dynamic allocation during their initialization. This is done immediately, so that there will be no problem with subsequent instructions which may make calls to procedures in those modules.

Three tasks remain. First, the entire memory space of the operating system must be protected from future memory allocations by allocating a memory block encompassing the entire system. Second, the entry of the null process in the process block table is initialized with the memory block just allocated and the address of the null context (which can still be found in the trap vector). Finally, the cardreader must be started. This is done by calling the **`LocalSys-tem`** machine instruction **`Read`**. Whenever a character has been read, an interrupt will cause the character to be processed and start the card reader going again.

At this point, there is nothing left to do but wait, which is what the null process is best at. A procedure in the main module consisting of an endless loop is called; it will continue to run, except during interrupts, until it is displaced by the first new process, and again whenever no other processes are available.

4.2.2.2. Major Data Structures

During normal operation, the system will be handling a large number of processes, and must keep track of how many there are, which process has the CPU, which will get it next, how much memory space each process controls (and at what locations), how many disk sectors each process controls (and which ones), what the status of each process is, how much free memory and disk space there is, and so on. This information is contained in six major data structures: the **Memoryblock** record, the **Diskblock** record, the **Context**, the **Processblock**, the **Priorityqueue**, and the **deltalist**.

The Memoryblock

All **Memoryblock** records are contained in two linked lists: a FREE list and a USED list. Each **Memoryblock** record contains the following information on one block of memory: 1) the low page of memory included in the block (a page being 128 bytes), 2) the high page, and 3) a pointer to the next **Memoryblock** record in the list. **Memoryblock** records are defined by and hidden within **MemoryManager**; procedures in other modules can only access them through the pointer type **MemoryblockID** and exported procedures.

The two lists of **Memoryblock** records are initialized by a call to **InitMemory** during system initialization. The USED list is set to **NIL**, while the FREE list is set to contain a single record which considers all of memory a single block.

The Diskblock

Diskblock records are all contained in a doubly-linked list. Each **Diskblock** record contains the following information about one block of disk space: 1) the low sector included in the block (a sector also being 128 bytes), 2) the high sector, 3) the status of the disk block (FREE or USED), 4) a pointer to the next **Diskblock** record in the list, and 5) a pointer to the previous record in the list. The methods used by **Diskblock** records and **Memoryblock** records to distinguish between FREE and USED blocks are different for illustrative purposes. **Diskblock** records are defined by and hidden within **DiskManager**; procedures in other modules can only access them through the pointer type **DiskblockID** and exported procedures.

The list of `Diskblock` records is initialized by a call to `InitDisk` during system initialization. The list is set to contain one record per disk cylinder, each of which considers its corresponding cylinder to be a single FREE block. Disk blocks cannot cross cylinder boundaries.

The Context

The `Context` holds the state of a process. It is not kept up to date in the currently running process, but is updated when the process is interrupted. It keeps track of the location of the process in memory, the status of its activation record stack, its mode bit, and the reason for any traps. Technically, the `Context` is a globally available data structure, defined and exported by `LocalSystem`; but in fact it is accessed only through `VirtualMachine`, which exports `ContextID` (a pointer to a `Context` variable) and procedures for manipulating a given context.

Each context is initialized at the time of a process's creation by a call to `VirtualMachine`'s `NewContext`.

The Processblock

The `Processblock` is a record which holds all information about a process. First and foremost, it contains the process's `ContextID`. In addition, it contains the `Memory-blockID` corresponding to the process's physical location in memory, and the `Disk-blockID` corresponding to its physical location on disk, as well as a flag indicating whether it is currently memory-resident. It also keeps track of all other disk blocks allocated for use by the process, how much of the process's input has been read, where its output has been directed, the process's priority, and its running status. All processblocks are stored in an array called the `processblocktable`. Both `processblocktable` and the `Processblock` record are local to `ProcessManager`. Procedures in other modules can only access them through exported procedures and the type **ProcessID**, which is an index into **processblock-table**.

The processblocks in **processblocktable** are initialized by a call to **InitProc-Manager** during system initialization. All processblocks have their status set to TER-MINATED, which is an indication that the **ProcessblockID** for that processblock is currently unused.

The Priorityqueue

The **Priorityqueue** is a list of ProcessIDs kept in order by their priority and, where priorities are equal, on a first-come-first-served basis. Each priority queue is a pointer to an array, indexed by process ID, whose elements are records containing the priority of the indexing process, and the process IDs of the preceding and succeeding elements in the queue. Each priority queue also has a dummy head and tail node, to speed insertion and deletion of records. Priority queues are used to hold the lists of processes with which the scheduler deals. There is one list for ready processes (those which take turns at the CPU), one for eligible processes (those which are prevented from running only by current policy), one for ineligible processes (those currently unable to run, whether blocked, sleeping or suspended), and one for processes waiting to be swapped in (those which were swapped out while ineligible, but whose status has now changed). The array type which constitutes a priority queue is local to **Pro-cessManager**, and can only be accessed through the pointer **Priorityqueue** and exported procedures.

Each priority queue is initialized by the procedure or module which uses it, with a call to **InitPriorityQueue**. A newly initialized list is always empty.

The deltalist

The **`deltalist`** is a linked list of time-scheduled **`Events`**. Each **`Event`** is a record which contains the ID of the process to be acted upon, the difference between the times when the previous **`Event`** and current **`Event`** should be executed (hence "delta" list), and a pointer to the next **`Event`**. There is only one **`deltalist`**, local to **`ProcessManager`**, and it is accessible only by exported procedures. **`Clock`** regularly prompts **`TrapHandler`** to check the list and execute any actions that have come due. Possible actions consist of rescheduling a process, awakening a process, terminating a process, and preempting a process.

4.2.2.3. Interrupt Handling

When an interrupt occurs, the register values are saved in the context of the current process. The register value for the address of the context, however, is saved in the ''formercontext'' field of the appropriate interrupt vector. Then that register value is replaced with the address of the operating system's context, and the values from that context are used to refill the other registers. The value of the PC is taken from the ''interrupthandler'' field of the interrupt vector.

Each device has one routine to handle its interrupts: Disk interrupts are handled by **`TransferComplete`**, in **`DiskManager`**; lineprinter interrupts are handled by **`CallSpoolOut`**, in **`Spooler`**; cardreader interrupts are handled by **`CallSpoolIn`**, also in **`Spooler`**; and clock interrupts are handled by **`ClockInterruptHandler`**, in **`Clock`**. None of these is exported; each is only called as the result of an interrupt. All traps are handled by a single trap handling procedure, **`ProcessTrap`** in **`TrapHandler`**, which likewise is not exported. During the initialization of modules which contain interrupt handling procedures, the ''interrupthandler'' field of each interrupt vector is set to contain the address of the correct interrupt handling procedure by a call to **`VirtualMachine`**'s **`SetInterruptHandler`**.

If the interrupt is a trap, the value of the trap code is stored in the context of the process that was trapped. This value is retrieved at the beginning of the trap handling procedure by a call to **VirtualMachine**'s **TrapReason**, and is used to index into a case statement to execute the proper action. Unlike UNIBATCH, therefore, MULTIBATCH does not require altering the M-Code machine's interrupt mechanism to deal with parameters. Traps can be divided into three types: 1) error traps that are called directly by the machine, such as **STACKOVERFLOW**; 2) other error traps that are caught by the operating system, such as **BADINSTRUCTION**; and 3) non-error traps, such as **INITIALIZE** and **SVC**.

Each interrupt handling routine concludes with a call to **VirtualMachine**'s **ReturnFromInterrupt**. This procedure uses the **LocalSystem** machine instruction **ContextSwitch** to replace the register values with those in the context of the **ContextID** stored in the interrupt vector. The interrupted process then picks up where it left off.

An additional complication occurs in the handling of a disk interrupt. It is often the case that calling for a disk operation requires suspending action on a certain task until the disk operation is completed. To accomodate this situation, **DiskRead** and **DiskWrite** both take as parameters a procedure with one parameter of type **WORD**, and a value to serve as the parameter to that procedure. These values are saved in the queue of disk jobs. When a disk operation completes, causing an interrupt, the interrupt handling procedure retrieves these parameters and calls the procedure. This allows the caller to have some control over what actions are taken when the disk operation completes. Often, the caller does not require any response at the time of disk completion; for these cases, the empty procedure **Null** is exported by **DiskManager**.

This facility is used to a great extent in **Spooler**. Characters are read from the card reader one at a time and stored in a circular memory buffer, divided into a constant number of pages. When any page is filled, its contents are written to the spooling area of the disk. However, no more characters must be written into that page until the disk write has completed. So a procedure is passed to DiskWrite, along with the name of the circular buffer. At disk completion, the procedure is called, recording in the buffer that the page is now free to be written into, and restarting the card reader in the event that it had halted because the buffer was full. Similar actions occur when writing to the line printer.

4.2.2.4. Supervisor Calls

Supervisor calls are implemented as a form of trap. Each supervisor call first calls an intermediate procedure whose action is simply to call `VirtualMachine`'s `Trap` with **SVC** as a parameter; however, the intermediate procedure itself takes as parameters the type of supervisor call and the parameters to it. This places all the information in a known location on the process stack: the top activation record will be the one for `Trap`, and the fourth location above the base of the preceding activation record will hold the variety of supervisor call. The parameters to the supervisor call will be immediately above that.

The call to `Trap` then raises a trap interrupt. The interrupt handler finds from the process's context that the trap was a supervisor call. It then uses `VirtualMachine`'s `SVCArguments` to obtain the type of supervisor call and its arguments. The type of supervisor call is used to index into a case table of possible actions.

The user process may invoke supervisor calls to read from input, write to output, go to sleep, find its upper bound, and allocate and deallocate diskblocks. More calls may easily be added.

4.2.2.5. User Processes

Each user job is assigned a correctly initialized `ProcessID`. The amount of memory space given to the process is equal to the size of its segmenttable and all code and data frames (which is recorded while the job is being loaded), plus a fixed amount for stack and heap space.

If JCL is added to the system, stack and heap size could be made specifiable.

The initialized **ProcessID** is then manipulated by the various levels of the scheduler to determine when the process should be swapped in or out of memory, when it is fit to run, and when it should actually get the CPU. Once a process is given the CPU, no other user process of equal or lesser priority can preempt it. This is because all jobs are batch jobs, and keeping context switching to a minimum will increase throughput. However, additional facilities of time-slicing and maximum permitted CPU time may be added. Of course, any interrupt or trap will give the operating system control of the CPU until the interrupt is handled.

In addition, any call for disk I/O will cause the process to voluntarily relinquish the CPU. At this time, the process will be interrupted by its own supervisor call, and the scheduler will reschedule it with the status **BLOCKED**. The next process in the **readylist** priority queue will then have its context specified as the one with which the registers should be loaded. The **BLOCKED** process will again be rescheduled when the requested I/O has completed. Normal I/O for a user–e.g., reading input–does not cause the process to block since all input is contained in memory, and output goes directly to a memory buffer. However, whenever a page of the output memory buffer fills, it is written to disk, blocking the process. The output is stored in a linked list of disk blocks which can be spooled as soon as it is complete.

There are three basic states in which a process may exist, as shown in Figure 8. Through procedure **ShoulderTap**, called on periodic clock interrupts, **HighLevelScheduler** is prompted to see if any jobs are waiting to be loaded and made into processes. **HighLevel-Scheduler** is thus responsible for bringing jobs into the system and initiating processes. New processes are automatically placed into a *stopped* state. **MediumScheduler** is responsible for deciding which priority queue a process should be on. When it places a process on the **runqueue**, the process becomes *runnable.* Processes placed on any other priority queue remain in a stopped state. **LowLevelScheduler** decides which process on the

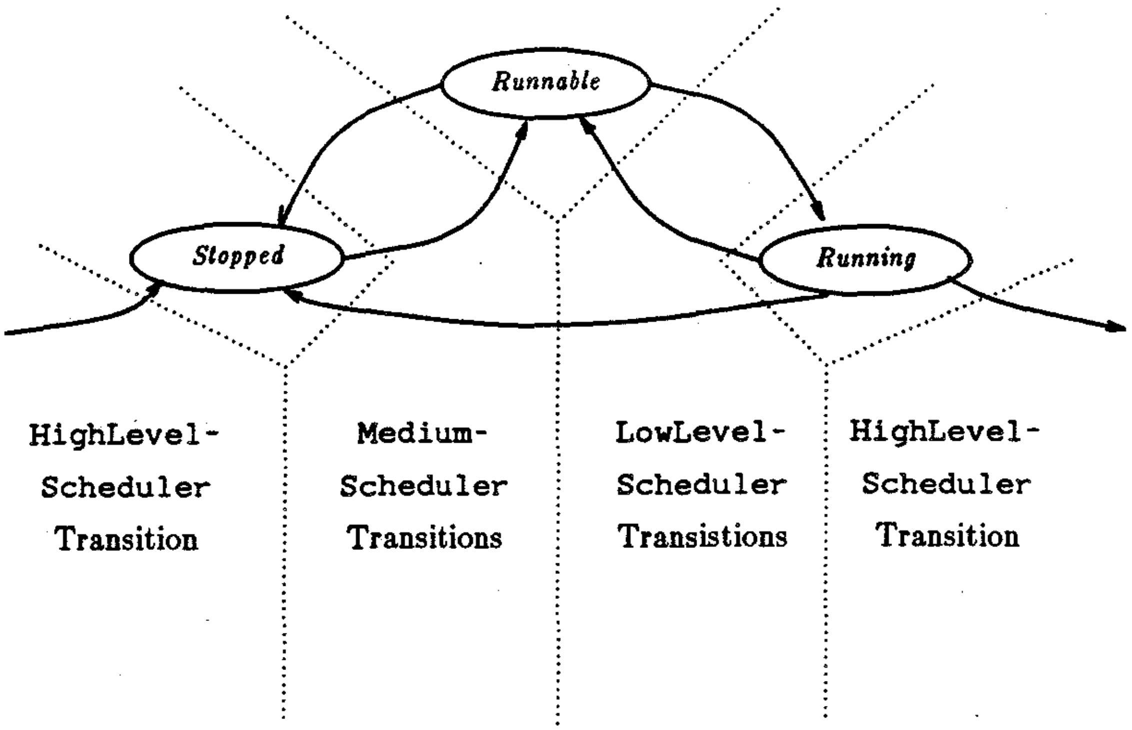
Runnable
Stopped
Running
HighLevel-Scheduler Transition
Medium-Scheduler Transitions
LowLevel-Scheduler Transistions
HighLevel-Scheduler Transition

Figure 8. Process State Transitions in MULTIBATCH.

runqueue should next be set *running*. **LowLevelScheduler** also decides when the running process should yield the CPU. On the occasions when a process goes directly from the running state to a stopped state, as on a block for I/O, **LowLevelScheduler** changes the process's status, and **MediumScheduler** places the process onto the correct priority queue. Finally, **HighLevelScheduler** decides when a process has finished (or made an irrecoverable error), and causes the process to exit from the system. When a user process has finished, the operating system frees all the process's disk and memory space, and spools the linked list of disk blocks which hold the process's output.

ProcessManager declares eight process states, not three: **BLOCKED**, **CURRENT**, **INITIALIZING**, **PENDING**, **READY**, **SLEEPING**, **SUSPENDED**, and **TERMINATED**.

CURRENT is the designation for the *running* process; **READY** means the process is *runnable*. **BLOCKED**, **INITIALIZING**, **PENDING**, **SLEEPING**, and **SUSPENDED** are all *stopped* states. They are distinguished to indicate the reason for the stoppage. **BLOCKED** indicates that the process is awaiting some event (probably an I/O completion); **INITIALIZING** is the state in which **HighLevelScheduler** passes a process to **MediumScheduler**; **PENDING** means that the process may be placed on the run queue as soon as current policy permits; **SLEEPING** indicates that the process is waiting for a time interval to pass; and **SUSPENDED** means that the operating system has indefinitely removed the process from the run queue. **TERMINATED** is the state given to an *unused* process ID, since a process exits the system when it terminates.

4.2.3. Salient Features

MULTIBATCH is most notable in the following ways:

(1) **It is small.** Although MULTIBATCH is about six times the size of UNIBATCH, UNIBATCH was a toy, and MULTIBATCH is not. Compared to real systems, it is still small. Figure 9 contains a chart of the size of each module of MULTIBATCH, with and without comments, and the size of M-code produced.

(2) **It includes nearly all the major functions of an operating system.** In addition to the UNIBATCH functions of loading, scheduling, processing interrupts, handling errors, and providing protection, MULTIBATCH also spools, manages processes, memory and disk space, and has multiprograming.

(3) **Most of its operating system functions are substantially implemented, and easily extendible.** Interrupt handlers are spread throughout the system, one for each device. A new device would require a new interrupt handler, but no existing interrupt handler would have to be altered. The scheduler has grown from doing almost nothing, in UNIBATCH, to

		line count	without comments	size of M-Code (bytes)
Clock	*def*	30	8	258
	mod	62	29	
DiskManager	*def*	97	27	1630
	mod	606	359	
HighLevelScheduler	*def*	38	9	934
	mod	267	173	
Loader	*def*	34	9	504
	mod	108	64	
LocalSystem	*def*	203	92	0
	mod	49	23	
LowLevelScheduler	*def*	71	18	888
	mod	199	116	
MediumScheduler	*def*	60	11	901
	mod	159	93	
MemoryManager	*def*	101	30	908
	mod	241	162	
MultiBatch	*mod*	105	45	455
ProcessManager	*def*	390	76	2867
	mod	764	518	
Spooler	*def*	77	20	2473
	mod	783	492	
Swapper	*def*	45	10	529
	mod	128	69	
SVCalls	*def*	65	15	484
	mod	101	46	
TrapHandler	*def*	13	2	2184
	mod	426	283	
VirtualMachine	*def*	241	52	1154
	mod	314	207	
Total		5777	3058	17797

Figure 9. The Size of MULTIBATCH.

a full three-level scheduler, to which time-slicing and priority scheduling can easily be added. The disk sweep strategy is easily adjustable, as are the strategies for allocating memory and disk blocks. Error checking is still minimal, however, with all detected errors aborting the user's program, and there is still no file system.

(4) **It makes use of concurrency in the operation of peripherals.** The card reader, line printer, and disk all run concurrently with the CPU, signaling their completion by an interrupt.

(5) **It is fully documented.** As with UNIBATCH, particular attention has been given to make the purpose and use of all exported procedures clearly understandable merely from reading the definition modules.

(6) **It is structured.** Though both UNIBATCH and MULTIBATCH are hopefully written in structured code, the overall structure of UNIBATCH was weak and somewhat accidental. That of MULTIBATCH is fully intentional, and stronger as a result.

4.2.4. Organization of MULTIBATCH

This section describes the rationale used in dividing MULTIBATCH into modules; it then briefly describes the modules, and the ways in which the modules interact with one another. It also suggests an order in which the modules should be read for maximum comprehension.

4.2.4.1. Modular Construction

Unlike UNIBATCH, MULTIBATCH was initially conceived as a modular design. This was partly due to its larger size, and partly to its initiation at a later point in the SoftLab project, when several modular languages were already under consideration. Seven modules–`DiskManager`, `MemoryManager`, `ProcessManager`, `Swapper`, and all three levels of the scheduler–were produced in pseudo-code before Modula-2 was actually chosen as the language of SoftLab. At about that time, MULTIBATCH was put on hold while the effort was made to translate UniBatch into Modula-2 and clean up some of its darker corners that were illuminated with the transition.

On returning to MultiBatch, and keeping in mind the lessons learned from UniBatch, it seemed that the first task was to write definition modules for all the modules–both those already existing in a pseudo-code version and those which had yet to be started. This clarified exactly what each module was to be responsible for, and was responsible for some significant changes in the existing pseudo-code modules. As with UniBatch, the modules were divided by the criterion of functionality. **`ProcessManager`** is the least coherent module in this respect, as it incorporates the three separate (though related) functions of process block management, priority queue management, and process event scheduling. These functions are grouped together to allow access to the **`processblocktable`**; putting them in separate modules would have required exporting this private data object.

The next decision was to build MultiBatch from the bottom up. Actually, a dual approach was used: the system was designed top down (as the writing of the definition modules shows), and each module was written top down; but the order in which the modules were constructed was bottom up. The ideal goal was to have a strict hierarchy of modules, where each module imported only from the one directly below it; and to write the modules starting from the lowest one on the hierarchy. As it turned out, it was impossible to make MultiBatch so strictly compartmentalized without being unacceptably artificial about the **`IMPORT`** and **`EXPORT`** clauses; in general, it was deemed a poor idea for Module B to import a procedure which it never used from Module A, just so that it could export it to Module C. However, the ideal of the strict hierarchy did influence the shape of MultiBatch, and as a result it is both more hierarchical and more modular than UniBatch.

4.2.4.2. Functions of Modules

MultiBatch contains twice the number of modules that UniBatch does; where UniBatch has eight (seven not counting SVCalls), MultiBatch has sixteen (fourteen not counting SVCalls and OSStorage). A brief statement of the function of each follows, in alphabetical order by module name.

Clock–Keeps track of time and initiates time-scheduled events. (Not in UniBatch.)

DiskManager–Allocates and deallocates diskblocks, and supports the disk read and write operations. (Not in UniBatch.)

HighLevelScheduler–Initiates and terminates jobs. (In UniBatch as **Scheduler**.)

Loader–Translates characters of input into octal M-code instructions and loads them into memory. (Also in UniBatch.)

LocalSystem–Defines programer-available features of the machine. It contains several machine instructions under the guise of procedures, and defines the Context data type. (Also in UniBatch as **LocalSystem**, though some features are found in **ContextHandler**.)

LowLevelScheduler–Decides which process gets the CPU next. (Not in UniBatch.)

MediumScheduler–Decides which processes should be memory resident, and which should be passed to **LowLevelScheduler**. (Not in UniBatch.)

MemoryManager–Allocates, deallocates, and maintains up-to-date information on main memory blocks. (Not in UniBatch.)

MultiBatch–Initializes the system and starts it running. (In UniBatch as **UniBatch**.)

OSSTORAGE–Provides operating system specific details to allow the vendor-supplied module **STORAGE** to correctly perform dynamic heap allocation; dependent upon, but not part of, the operating system. (Not in UniBatch.)

ProcessManager–1) Creates, maintains, and removes processes; 2) manages priority queue lists of processes; 3) keeps track of time-scheduled events for processes. (Not in UniBatch.)

Spooler–Maximizes device utilization by holding input and output on disk until they can be further processed. (Some features are included in UniBatch module **IO**.)

Swapper–Controls the transfer of non-running processes to and from disk. (Not in UniBatch.)

SVCalls–Allows user access to certain operating system features; dependent upon, but not part of, the operating system. (Also in UniBatch.)

TrapHandler–Responds to the raising of a **TRAP** interrupt. (Included within UniBatch module **InterruptHandler**.)

VirtualMachine–Completely hides **LocalSystem**, presenting a higher level machine to the rest of the operating system. (Some features are present in UniBatch module **ContextHandler**.)

In general, references to the operating system should not be taken to include either **SVCalls** or **OSSTORAGE**, unless specifically stated otherwise.

4.2.4.3. Dependencies of Modules

As with UniBatch, the easiest way to see the overall organization of MultiBatch is to look at its dependency graphs. As before, an arrow will point from each module to all modules on which it depends (that is, from which it imports).

Procedural dependencies

How close was it possible to come to building MultiBatch as a strictly compartmentalized hierarchy, where each module imported only from the one directly below it? In truth, not very close. And yet a careful study of the procedural dependencies for MultiBatch reveals that striving for this ideal goal had a profound effect on the structure of the operating system.

Figure 10 contains the procedural dependency graph for UniBatch. The graph is even more tangled than the graph for UniBatch, but the only reason for this is the greater number of modules. Each module in UniBatch points to an average of 40 percent of the other modules; if this were true in MultiBatch, there would be a total of 73 arcs in the graph. In fact, there are

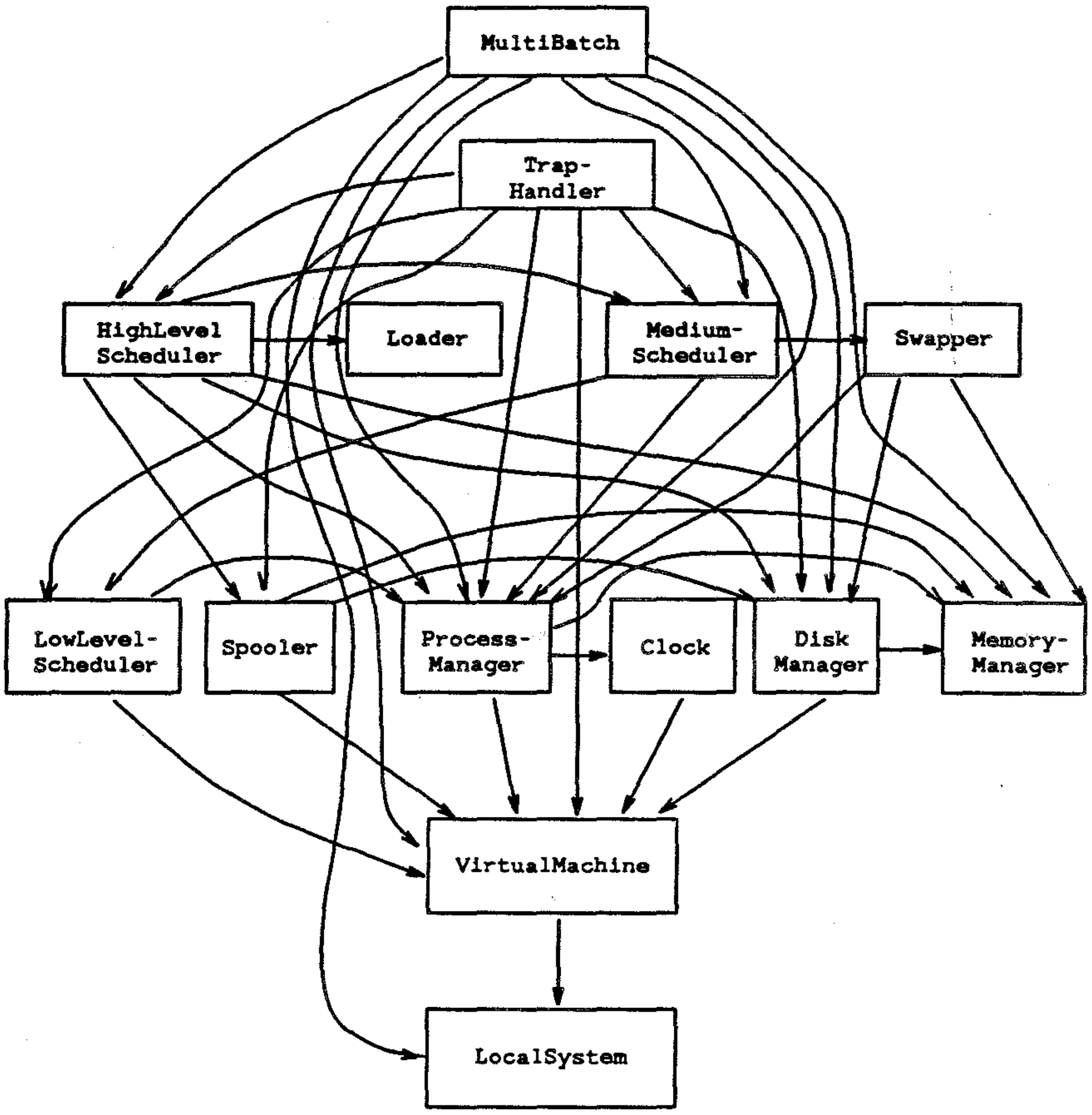

Figure 10. Procedure Dependency Graph for MULTIBATCH.

only 39. As with the graph for UNIBATCH, no arrow points up, thus revealing a hierarchy of six layers–four for normal operation, one for handling traps, and one for system intitialization. Unlike the graph for UNIBATCH, however, no arrow on any given level points left. What this means is that MULTIBATCH contains *no circular dependencies*. In other words, the modules of

MULTIBATCH can be and are arranged in a strict hierarchy with the main **`MultiBatch`** module at the top, **`TrapHandler`** below it, and so forth, going from top to bottom among the levels, and from left to right within a level, with each module placed directly below its predecessor. If arranged in this fashion, the procedure dependency graph *still* has no arrows that point up. So a strict hierarchy is achievable, even if strict compartmentalization is not. UNIBATCH does not share this property.

Compartmentalization has not been completely forgotten. Note that if we allow special exceptions for initialization and interrupt handling, tasks which by their very nature are nonmodular, and remove all arrows due to these actions from Figure 10, we produce the much neater graph of Figure 11. All arrows have now disappeared from **`TrapHandler`**, which is completely involved with the handling of one type of interrupt, and from **`MultiBatch`**, which is nothing but initialization. The remaining arrows are now compartmentalized by level: all modules import only from modules on their own level, or from modules on the level immediately below.

It may be noted that this same trick would have worked to compartmentalize the graph of UNIBATCH–but its only significance in that case would have been a demonstration of the tendency towards order and layering inherent in an operating system. In MULTIBATCH, on the other hand, it reveals an intent to utilize that inherent layering to its fullest. Put another way: **`Scheduler`** in UNIBATCH does not happen to need any exports from **`LocalSystem`** (two levels below it), and so it doesn't break the compartmentalization; but had it needed such an export it would have imported it without hesitation. In MULTIBATCH, with its greater complexity and proliferation of modules, such a need seemed to arise time and again, but in all cases it was deliberately circumvented; and the resulting design is cleaner for it.

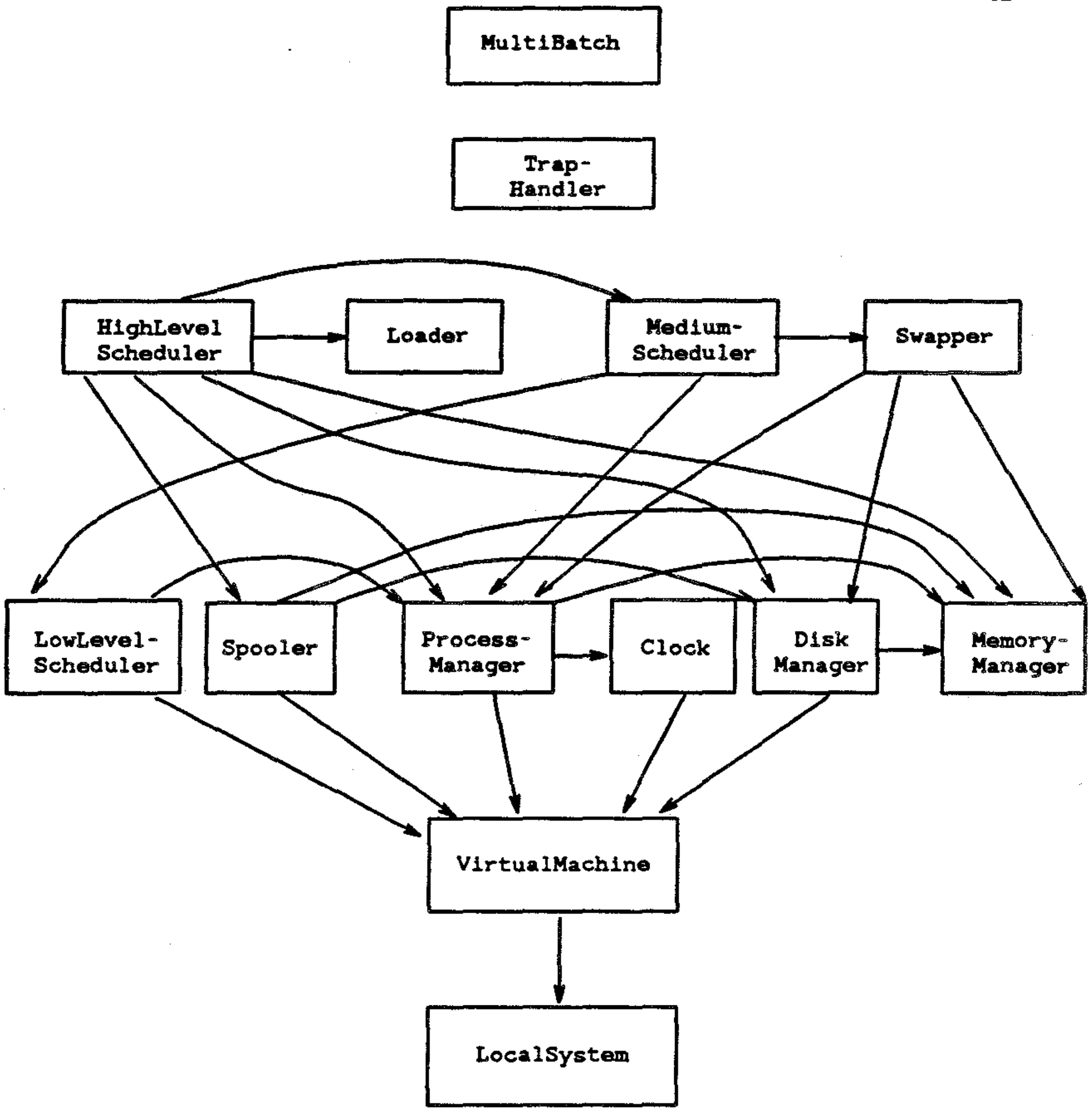

Figure 11. Revised Procedure Dependency Graph for MultiBatch.

Dependencies on variables

As was said in the overview on UniBatch, dependence on variables imported from other modules is fatal to information hiding and true modularity. In MultiBatch, however, this sort of dependence is even less of a problem than in UniBatch. **LocalSystem** is the only module to export variables, as in UniBatch; but unlike UniBatch, MultiBatch has the module `VirtualMachine` which completely hides `LocalSystem` from the rest of the operating system. Any other module needing access to the information in those variables gets it by means of *procedures* exported from `VirtualMachine`. Thus there is no chance of accidentally altering the value of a global variable.

Dependencies on constants and types

It may be noticed that (within limits) hierarchies different from that presented in the last two figures could have been chosen for the modules of MultiBatch. For example, `Loader`, which imports no procedures, could have been placed virtually anywhere in the hierarchy lower than `HighLevelScheduler`; or `Spooler` could have been to the right of `Clock` instead of to the left of `ProcessManager`.

Choices like these were made according to two criteria. One was that the hierarchy should be as reasonable as possible; thus `LocalSystem` and `VirtualMachine` essentially constitute the machine, and are at the bottom of the hierarchy. Constituting the next level are the tasks of managing peripherals, memory, and the CPU itself (including saving the process state). The next level has the tasks of creating and terminating processes, and general high level control of them. (Note that ProcessManager really contains tasks that belong on this level, and other tasks that belong on the last; encompassing them all in one module requires that the module be on the lower of the two levels.) And finally come the two special levels of interrupt handling and initialization.

The other criterion was to hold fast, if possible, to the strict hierarchy in *all* exports and imports, not just those of procedures. It is this criterion which prevents `Spooler` from being placed to the right of `Clock`. Unlike UniBatch, MultiBatch can be arranged in a hierarchy which is completely supported by the exports and imports of types and constants as well as the more vital dependencies of procedures and variables. The dependency graph for constants and types is contained in Figure 12. As with the revised procedural dependency graph, arcs due to initialization or interrupt handling have been omitted.

Dependence on constants and types imported from other modules does reflect the organization of a large program. The ability to select a single hierarchy for the modules of

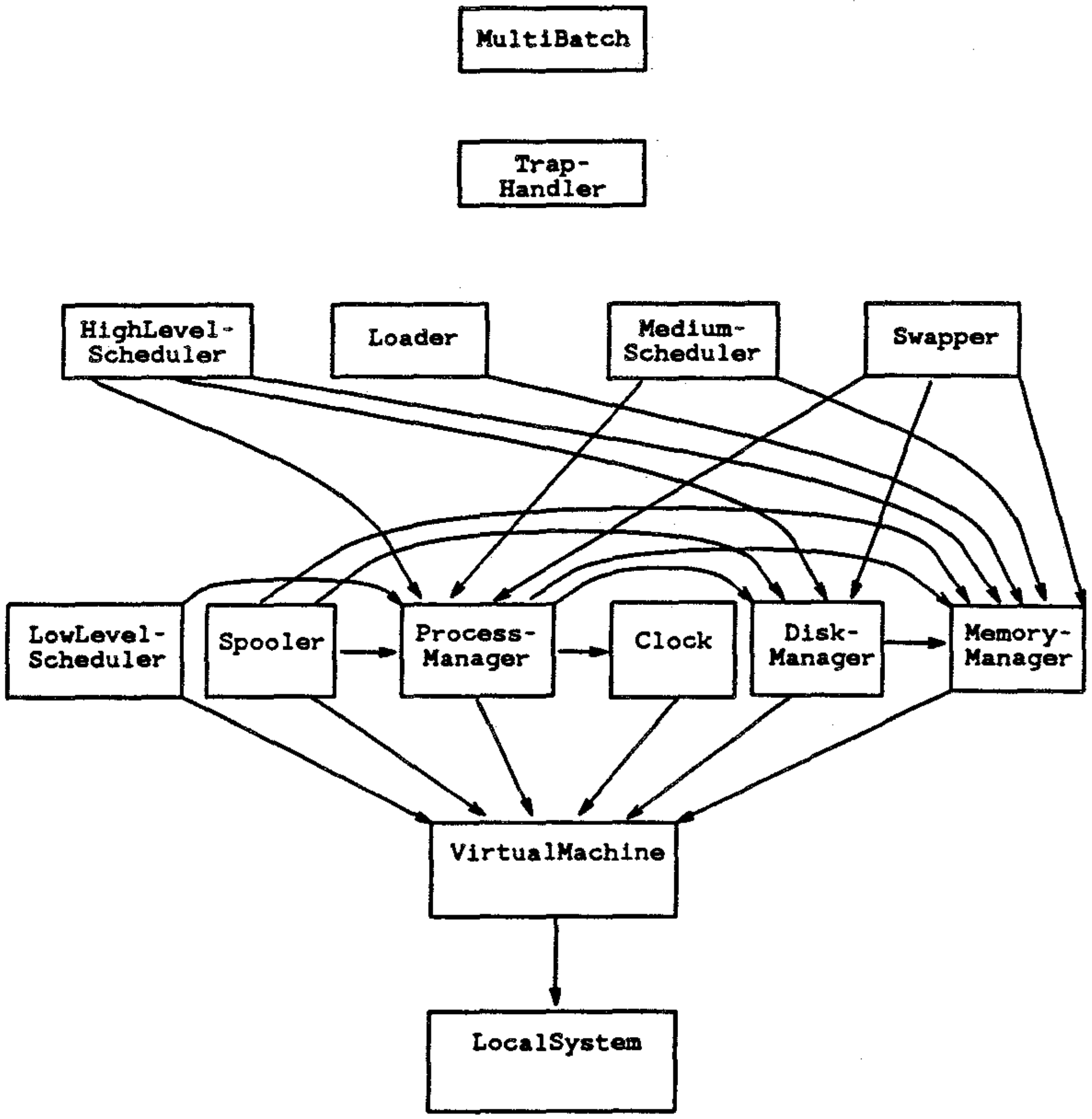

Figure 12. Graph of Dependencies on Constants and Types for MultiBatch.

MULTIBATCH which is supported by *all* dependencies is another indication that, unlike that of UNIBATCH, MULTIBATCH's hierarchy is strong and well-developed.

Suggested order of reading

MULTIBATCH will be most understandable if read from the bottom of the hierarchy to the top, so that each module only refers to procedures, constants, and types which have already been encountered. Note that the strict hierarchy makes this a completely realizable goal, whereas the circular dependencies in UNIBATCH make it impossible. In addition, all definition modules should be read through once before any of the implementation modules.

Although it is possible to provide several hierarchies, all of which are strict in the sense that no module imports from another module higher than itself, it is recommended that the following order of reading be chosen. It is taken from the dependency graphs, where an effort was made to keep closely related modules (such as **`Loader`** and **`HighLevelScheduler`**) together, even when the dependencies would have allowed them to be separated.

(1) **`MultiBatch`**,
(2) **`LocalSystem`**,
(3) **`VirtualMachine`**,
(4) **`MemoryManager`**,
(5) **`DiskManager`**,
(6) **`Clock`**,
(7) **`ProcessManager`**,
(8) **`Spooler`**,
(9) **`LowLevelScheduler`**,
(10) **`Swapper`**,
(11) **`MediumScheduler`**,
(12) **`Loader`**,
(13) **`HighLevelScheduler`**,
(14) **`TrapHandler`**,
(15) **`MultiBatch`**.

As with UNIBATCH, it is suggested that all initialization sequences (including the **`MultiBatch`** module itself) be read both before and after the sections which they initialize.

No modules are dependent upon **`SVCalls`** and **`OSSTORAGE`**, and they are not strictly part of the operating system; however, as they are dependent only upon **`VirtualMachine`**, they may be read any time after **`VirtualMachine`**.

4.3. ASSIGNMENTS

For each of the following modifications, be sure to decide how much the following issues apply, and address them suitably:

- What is the overhead of this modification?
- What are the relevant performance measures?
- Are there workloads which
  - a. dramatically increase,
  - b. dramatically decrease, or
  - c. insignificantly alter

  these measures after the modification is installed?
- Is the requested modification a reasonable one to consider?

INTRODUCTORY MODIFICATIONS

(1) Add the supervisor call `DiskRead`. It should be able to read any specified number of bytes starting from any specified location within any specified sector.

(2) Add the supervisor call `DiskWrite`. It should be able to write any specified number of bytes to any specified location within any specified sector.

(3) Add procedures to keep time statistics on processes. The process block data structure already is built to hold the statistics. Avoid adding any circular dependencies in MULTI-BATCH.

(4) Add JCL to the system so that process priorities may be specified. It should be an error to specify a higher priority than `MAXPRIORITY`. Who should be responsible for setting priority? On what basis? What should the default priority be?

(5) Add JCL to the system so that a user process's stack size may be specified. What should the default stack size be? Should there be any limits to the size which may be specified? If so, specifying a size greater than this limit should be an error. Does it make sense in this system to be able to specify heap size? Why or why not?

(6) Add JCL to the system so that a user process's maximum service limit may be specified. What should the default be? Rewrite `LowLevelScheduler` so that a process is aborted if it runs longer than its maximum service limit. (This assignment assumes that assignment #3 has already been completed.)

(7) Change `SwapOut` in the module `Swapper` to return a boolean indicating success or failure. Alter the code everywhere `SwapOut` is called to test for success and take appropriate action in case of failure.

(8) Add a procedure to `MediumScheduler` to determine the load average (the average length of the ready list) over the past minute, five minutes, and fifteen minutes. Under what circumstances would such a procedure be useful?

(9) Implement in `Loader` the currently unimplemented error checks of `BADINSTRUCTION` (meaning bad format) and `UNDEFINEDINSTRUCTION` (meaning the format is correct, but the instruction is not implemented). Any operand or instruction which is not all digits representing an octal number between 0 and 377 is in bad format. Only instructions can be undefined (there are no undefined operands)–to find out which instructions are undefined, you will need a copy of the CPU interpreter.

(10) Alter the handling of the sleep supervisor call so that the process is not swapped out unless it has asked to sleep for more than 1 second. Is this a reasonable time span? Why or why not? What other factors should the decision depend on?

ADVANCED MODIFICATIONS

(1) As it stands now, if MultiBatch runs out of stack or heap space, it will crash. Rewrite it so that more space will be allocated in this case. What problems would arise in ensuring that the stack remains contiguous? What are the difficulties in dealing with a noncontiguous stack? How much overhead is involved in normal operation, making certain that the stack is not disrupted even during this rare event?

(2) Add protection to the disk, so that no process can write over or read from another's disk space. How much overhead does this add to normal disk reads and disk writes? What if each process specifies public read and write permissions? (This problem assumes that the disk read and write SVCs of Exercises 1 and 2 have been completed.)

(3) Add an overflow list of process IDs so that there is no fixed limit on the number of processes that can run. Those process IDs less than the overflow number should be stored in the array; those greater than the overflow number should be kept in a linked list which must be searched. All procedures which currently depend on indexing only will have to be altered. What effect does increasing and decreasing the size of the overflow value have?

(4) Modify `LowLevelScheduler` to allow each process to run for only a certain length of time (a quantum) before being interrupted. Does this serve any purpose in a batch system? If so, what? Does it alter the system's efficiency when running a queue of I/O bound jobs? Why or why not? How about a queue of compute bound jobs? (This problem assumes that the procedures to collect time statistics, as described in Exercise #3, have been implemented.)

(5) Assuming that time-slicing has been implemented in the scheduler according to the previous problem, modify **`LowLevelScheduler`** so that, rather servicing only the highest priority group of processes, all processes are serviced round-robin. However, the quantum allowed for each process should be **`MAXQUANTUM/priority`**; thus the highest priority processes (priority 1) get a full quantum, and the lowest priority processes (priority 5) get only a fifth of a quantum. (Beware of the null process!) Is this a reasonable way to schedule? Why or why not?

(6) Modify **`MediumScheduler`** to suspend processes when the load average is greater than 15, and not to add new ones if it is greater than 12. Is this useful with the original MULTIBATCH configuration? How about with time-slicing implemented? Are these particular limits appropriate?

(7) Modify **`TrapHandler`** to allow multiple traps to be stacked; that is, to allow traps to be called from within **`TrapHandler`**, without losing the context of the original interrupted context. Can this be done to any depth? Why or why not? Is it useful?

(8) Alter the scheme for allocating memory blocks from First Fit to Best Fit and Worst Fit. Which of the three is best? By how much? Does the type of work load matter? Why or why not?

(9) Change the disk search strategy from SCAN to N-SCAN and C-SCAN. Which of the three is best? Under which work load? By how much?

(10) Assuming that time-slicing has been implemented, adjust the size of the quantum. Does it have a maximum value, in terms of efficiency (throughput)? Are there any other effects? Why or why not? Would it make a difference if MULTIBATCH ran processes in time sharing mode instead of only in batch mode? Why?

(11) Move the constant **`PAGESIZE`** from **`MemoryManager`**'s definition module to its implementation module. Also, eliminate **`VirtualMachine`**'s constant **`BYTESPER-PAGE`**. Remove both from the import lists of all other modules. This will require some major changes, particularly in **`ProcessManager`** and **`Spooler`**, where the circular buffers depend on knowing the size of a page. You will probably want to build a structure inside **`MemoryManager`** that other modules can use to construct circular buffers. In order to keep **`PAGESIZE`** successfully hidden, this data structure must be local to **`MemoryManager`**, and only operable on by procedures that you will also write.

GROUP PROJECTS

(1) Add a second disk to MULTIBATCH, and use it for all spooling. How might this make a part of the spooling process less complicated? Create a configuration that makes it easy to add additional disks, or remove them. (see Comer [6] for a model).

(2) Add a file system to MULTIBATCH. User processes should be able to read from and write to files other than the standard input and standard output. You will need (among other things) supervisor calls for reading to and writing from disk, and opening, closing, creating and removing files.

(3) Add paging capabilities to **`Swapper`**. This will require allocating one memory block per page, rather than one per process. Keep in mind that the PC is currently set relative to the current codeframe, which may be on a different page. Also, dynamic variables are addressed relative to the top of the heap (the process's upper bound), which is also probably on a separate page from the dynamic variable. What will paging do to the simple "Bounds Register" method of protection used in MULTIBATCH? What can be done to replace it?

CHAPTER V.

CONCLUSION

5.1. SUMMARY

UNIBATCH is a toy operating system, which despite its small size permits the introductory study of operating system design. It is respectably modular. Had it been coded in Modula-2 from the start, it could have been more so; however, its size makes slight deficiencies in this area of minor concern. Modules are drawn along functional boundaries, making each module a conceptual unit. This, coupled with Modula-2's import/export mechanism, should make alterations to UNIBATCH fairly straightforward.

MULTIBATCH is a small but authentic operating system, intended to reflect more accurately and completely than UNIBATCH the manner in which actual operating systems are built. Though small, it is large enough that modularity is vital to building or understanding it. Because the interfaces are fairly constricted and always well-defined, it should be possible for students to make changes to the system with relative ease considering its overall size and the necessary interrelations between separate parts.

These operating systems are designed to be used as a teaching tool within the environment of SoftLab. The SoftLab environment will include not only the operating systems, but the simulated machines on which they run and the simulated peripherals which they control. Most importantly, it will include the IIE, or Integrated Instrumentation Environment. The IIE will allow the effects of design alterations to be monitored. This will enable students to obtain performance feedback from their alterations.

5.2. EVALUATION

By far the greatest shortcoming in UniBatch and MultiBatch is that they are untried. Though both systems compile in the form given, the simulated machines on which they were to run were not fully developed when this document was completed. As a result, neither system has ever been tested. Despite intensive desk-checking by the author and many students, there are certain to be some substantial errors uncovered. Substantial debugging will be required, once the hardware simulations become available.

This debugging task will be mitigated by the detailed documentation of UniBatch and MultiBatch. They were written with an eye to maintenance. All variables are commented, as well as virtually all control structures in the code. Each procedure has a block comment stating its purpose and the purpose of its parameters, and Modula-2 itself is as easy to read as Pascal.

Another shortcoming is that both systems illustrate only well-established techniques, and are fairly weak in comparison to real-world operating systems. It would seem that an educational tool should be state of the art and illuminate the leading edge of the discipline; but UniBatch and MultiBatch possess minimal error checking, lack file systems, and have limited protection. On the other hand, these operating systems were not written to be *studied.* They were written to be *manipulated.* It is not the realm of theory, but the realm of practice, in which they should be found valuable. This is the way with laboratory exercises in other disciplines as well. No one would suggest that Physics lab curricula abandon demonstrations of Newtonian mechanics, simply because Einsteinian relativity has proved more viable. There, it is the experimental method itself which is being studied; here it is the practice of operating system *design.*

The greatest strength of these operating systems is their usefulness. Alterations are not merely academic desk exercises. In the framework of SoftLab these systems become runnable operating systems which can be tried, altered, and retried, with performance effects monitored. They can be run *without* a dedicated physical CPU. They may safely be allowed to crash as a learning experience.

In addition, the operating systems and the surrounding SoftLab environment are *accessible*. The operating systems are written in a high level language, commonly available and easily readable. The hardware on which they run is simulated, and easily portable. No special equipment is needed in order to successfully run, test, and manipulate them.

Furthermore, the systems are small but complete. Because they are small, beginning students should not find them beyond their grasp; and because they are complete, students have the opportunity to see how all the parts fit together. Too often students are in the position of studying disembodied portions of larger systems, if they study any actual code at all.

These operating systems are only the first two of a family. Future additions to the family can afford to stress more advanced techniques and greater functionality at the expense of being larger and more complex.

5.3. FUTURE DIRECTIONS

There are many directions which could be taken by the next operating system to be written for the SoftLab project. The next will probably be a process-oriented operating system, similar in functionality to MultiBatch but actually composed of separate processes. These processes would have the status of utility programs, and take their turn in the run queue, just like user processes. This would decrease the CPU time spent in the operating system, and would allow interrupts to be disabled for a shorter time.

In some ways the code for this system might be easier to write and understand than the code for MultiBatch. In any case, there would be a whole new set of problems to solve and code, such as interprocess communication.

Other possible future systems include a multi-processing system, with the ability to run concurrent programs, and a distributed operating system.

Whatever directions are pursued, UNIBATCH and MULTIBATCH should respectably form the nucleus of a family of operating systems designed as a set of tools with which students may gain hands-on experience in system design and implementation.

REFERENCES

1. Brinch Hansen, P. *The Architecture of Concurrent Programs.* Prentice-Hall, Inc., Englewood Cliffs, N.J., (1977).

2. Brinch Hansen, P. The Trio Operating System. *Software-Practice and Experience 10* (1980), 943-948.

3. Brinch Hansen, P. Edison Programs. *Software-Practice and Experience 11* (1981), 397-414.

4. Brinch Hansen, P. Edison: a multiprocessor language. *Software Practice and Experience 11*, 4 (April 1981), 325-361.

5. Brinch Hansen, P. *Programming a Personal Computer.* Prentice-Hall, Inc., Englewood Cliffs, NJ, (1982).

6. Comer, D. *Operating System Design, the Xinu Approach.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, (1984).

7. Corbin, K., Corwin, W., Goodman, R., Hyde, E., Kramer, K., Werme, E, Wulf, W. A **Software Laboratory, Preliminary Report.** 71-104, Computer Science Department, Carnegie-Mellon University, (August 1971).

8. Corwin, W., Wulf, W. **SL230--A Software Laboratory, Intermediate Report.** Computer Science Department, Carnegie-Mellon University, (May 1972).

9. Halstead, M.H. *A Laboratory Manual for Compiler and Operating System Implementation.* American Elsevier, New York, NY, (1974).

10. Hammer, R. **Organization of Storage and Procedure Calls on an M-code Machine.** SoftLab Internal Working Document No. 3, Computer Science Department, University of North Carolina at Chapel Hill, (May 1985).

11. Holt, R. C. *Concurrent Euclid, The Unix System, and Tunis.* Addison-Wesley Publishing Company, Reading, MA, (1983).

12. Holt, R.C., Graham, G.S., Lazowska, E.D., Scott, M.A. *Structured Concurrent Programming with Operating System Applications*. Addison-Wesley Pub. Co., Reading, M.A., (1978).

13. Morrill, D.R. **An Integrated Instrumentation Environment in a Software Laboratory.** Master's Thesis, Computer Science Department, University of North Carolina at Chapel Hill, (expected March 1986).

14. Ohran, R. Lilith and Modula-2. *Byte 9*, 8 (August 1984), 181-192.

15. Shaw, A. *Operating Systems* (The Series in Automatic Computation). Prentice-Hall, Inc., Englewood Cliffs, N.J., (1974).

16. Wirth, N. *Programming in Modula-2, 2nd Edition* (Texts and Monographs in Computer Science). Springer-Verlag, (1983).

APPENDICES

These appendices contain the code for UNIBATCH and MULTIBATCH. Within each system, the modules are arranged in alphabetical order for easy reference. Every page is labeled at the bottom with the module name and operating system name. Page numbering is by module; that is, each module begins a new page 1.

Definition modules begin with a comment giving the function of the module, which provides the information a user would need in order to decide whether the module would be useful to him. Implementation modules begin with a comment stating the policy of the module, which provides user-irrelevant information on design decisions made in implementing the module's function.

Every procedure has a block comment stating the purpose of the procedure, and the functions of all its parameters. These block comments are found in the definition module for exported procedures, and in the implementation module for non-exported procedures.

Some implementation modules have a bracketed number in comments immediately following the name of the module. This is a *module priority* number, and is used to set the process interrupt mask (the **M** register). A process cannot be interrupted by any device unless the device number is greater than the module priority number. (This is also used with Modula-2 coroutines to implement monitors). The compiler currently does not recognize this construct, and so the numbers are commented out. However, this must be changed before the systems will actually run, as these modules depend on not being interrupted in order to function properly. User processes should never be able to declare module priority numbers higher than 6, as

7 or higher would allow them to disable traps or device interrupts.

The import lists of some modules include procedures which have been commented out. This is because the name of that procedure is identical to the name of a procedure declared withing the module, or imported from a third module. Modula-2 does not allow the import of a procedure in the form

```
FROM X IMPORT
      X.procedurename;
```

which would resolve the difficulty. Rather the entire module must be imported, and dereferencing must take place when the procedure is used. The convention used here is:

```
IMPORT X;
FROM X IMPORT
      (* procedurename *);
```

This is simply to reveal in the **IMPORT** statement what procedures are being used from module **X**.

In general, constants are in upper case, variables are in lower case, types begin with a capital letter, and procedures and modules have a capital letter for the beginning of each word (as in **LocalSystem**). Most exceptions are features of Modula-2 (**SYSTEM** is a module, not a constant), but a few exceptions were made to improve readability. Declarations are listed in alphabetical order whenever practical. Declarations of global objects that are used by more than one procedure in a module are located at the beginning of the module. Global variables which are used by only one procedure, but were declared globally so that their values would survive between procedure invocations, are declared immediately before the procedure in which they are used.

APPENDIX A

UNIBATCH

```
DEFINITION MODULE ContextHandler;

(**********************************************************************)
(*                                                                    *)
(*  FUNCTION:   Handle all facets of switching between the context of the  *)
(*              operating system and the context of the user program, including *)
(*              creating new contexts, and accessing and changing features of   *)
(*              a context.  This module and LocalSystem together constitute    *)
(*              a definition of the machine.                           *)
(*                                                                    *)
(*  AUTHORS:    Rick Snodgrass and Rick Fisher                         *)
(*                                                                    *)
(**********************************************************************)

  FROM SYSTEM IMPORT
    (* Types *)
    ADDRESS;

  EXPORT QUALIFIED
    (* Constants *)
    HIGHINSTRUCTION,

    (* Types *)
    ContextID, Mcodeinstruction,

    (* Procedures *)
    NewContext, Reset, SetPC, SwitchContext, SVCArguments;

  CONST
    HIGHINSTRUCTION  =  377B;  (* 377 octal + 1 = 256 = number of M-Code
                                   instructions *)

  TYPE
    ContextID;                 (* pointer to the actual context *)
    Mcodeinstruction   =   [0..HIGHINSTRUCTION];


  PROCEDURE NewContext(codeAndDataframebase, stackbase: ADDRESS;
    stacksize: CARDINAL):   ContextID;
  (*    Create a new context.  The caller must insure that the code frames and   *)
  (*    stacks of the contexts do not overlap.                                  *)
  (*                                                                            *)
  (*    PARAMETERS:   codeAndDataframebase–starting address of the             *)
  (*                      code frame.                                           *)
  (*                  stackbase–starting address of the stack.                  *)
  (*                  stacksize–size of the stack in words.                     *)
  (*    RETURNS  a context ID for the context.                                  *)
```

```
    PROCEDURE Reset(context: ContextID);
    (*      Reset the values in a context when a process is aborted.                    *)
    (*                                                                                  *)
    (*      PARAMETERS:     context–the ID of the context being reset.                  *)


    PROCEDURE SetPC(context: ContextID; PC: ADDRESS);
    (*      Change the PC of a context. The next time there is a context switch to this *)
    (*      context, instruction execution will start from the altered PC.              *)
    (*                                                                                  *)
    (*      PARAMETERS:     context–the ID of the context whose PC is to be             *)
    (*                          changed.                                                *)
    (*                      PC–the address to which the context's PC should             *)
    (*                          be changed. The caller must ensure that the             *)
    (*                          PC is within the code frame of the context.             *)


    PROCEDURE SwitchContext(context: ContextID);
    (*      Return to a previously stored context. The current context is not saved,    *)
    (*      as a context switch should only occur when the current context has          *)
    (*      finished. (The context of the operating system is permanently stored        *)
    (*      in a reserved memory location.)                                             *)
    (*                                                                                  *)


    PROCEDURE SVCArguments(context: ContextID):    ADDRESS;
    (*      Obtain from the previously executing context the arguments of the           *)
    (*      Supervisor Call that caused the context switch. Should be used only         *)
    (*      if the context switch was caused by a SVC.                                  *)
    (*                                                                                  *)
    (*      PARAMETERS:     context–the ID of the previously executing context.         *)
    (*      RETURNS  a pointer to the SVC arguments. If the context switch was          *)
    (*              caused by something other than an SVC, the pointer returned         *)
    (*              will probably be NIL, but it may point to nonsense values.          *)
    (*              Otherwise, the values will be as follows:                           *)
    (*                                                                                  *)
    (*                  WORD[0] is the type of supervisor call.                         *)
    (*                  WORD[1] is the starting address involved in the transfer.       *)
    (*                  WORD[2] is the number of bytes (characters) to be transfered.   *)

END ContextHandler.
```

```
IMPLEMENTATION MODULE ContextHandler;

(*************************************************************************)
(*                                                                       *)
(*  POLICY:        None to speak of.                                     *)
(*                                                                       *)
(*  AUTHORS:       Rick Fisher and Rick Snodgrass                        *)
(*                                                                       *)
(*************************************************************************)

    FROM SYSTEM IMPORT
        (* Types *)
        ADDRESS, WORD,

        (* Procedures *)
        TSIZE;

    FROM LocalSystem IMPORT
        (* Types *)
        Exceptioncode,

        (* Procedures *)
        ContextSwitch;

    FROM InterruptHandler IMPORT
        (* Types *)
        SVCcode;

    CONST
        MAXMODULES     =   98;          (* number of words in initial segment *)
        WORDSINMEMORY  =  40000B;

    TYPE
        Dataframeptr    =  POINTER TO Dataframe;
        Dataframe       =  RECORD
                               codeframe:          (* code base address *)
                                   ADDRESS;
                               initializationflag: (* indicates if module is initialized *)
                                   INTEGER;
                               stringpointer:      (* address of the string table *)
                                   ADDRESS;
                               globals:            (* a variable-sized list of words *)
                                   ARRAY [1..1] OF WORD;
                           END;
        Segmenttableptr =  POINTER TO Segmenttable;
        Segmenttable    =  ARRAY [0 .. MAXMODULES] OF ADDRESS;
        ContextID       =  POINTER TO Contextrecord;
```

```
Contextrecord  =  RECORD
                     dataframe:          (* data base address *)
                         Dataframeptr;
                     currentactivation:  (* base address of current activation record *)
                         ADDRESS;
                     PC:                 (* program counter *)
                         ADDRESS;
                     interruptmask:      (* process interrupt mask *)
                         BITSET;
                     stacktop:           (* pointer to top of stack *)
                         ADDRESS;
                     stacklimit:         (* stack limit address *)
                         ADDRESS;
                     trap:               (* trap responsible for interrupt *)
                         Exceptioncode;
                     errortrapmask:
                         BITSET;
                     segmenttable:       (* segment table address *)
                         Segmenttableptr;
                 END;


(*   Create a new context.   *)
PROCEDURE NewContext(codeAndDataframebase, stackbase: ADDRESS;
    stacksize: CARDINAL): ContextID;

    VAR
        context: ContextID;

BEGIN

    (* locate context at bottom of process stack *)
    context := ContextID(stackbase);

    WITH context↑ DO
        stacklimit := stackbase + stacksize;
        currentactivation := stackbase + TSIZE(Contextrecord);
        stacktop := currentactivation;
        dataframe := codeAndDataframebase;
        PC := ADDRESS(dataframe↑.codeframe↑);
        interruptmask := {15};
    END;
    RETURN context;
END NewContext;
```

```
(*  Reset the context if a user has halted.      *)
PROCEDURE Reset(context: ContextID);
BEGIN
    WITH context↑ DO
        currentactivation := ADDRESS(context) + TSIZE(Contextrecord);
        stacktop := currentactivation;
    END;
END Reset;


(* Set return PC in a context *)
PROCEDURE SetPC(context: ContextID; PC: ADDRESS);
BEGIN
    context↑.PC := PC;
END SetPC;


(* Switch the context *)
PROCEDURE SwitchContext(context: ContextID);
BEGIN
    ContextSwitch( ADDRESS(context) );
END SwitchContext;


(* Get the address of the argument to a Supervisor Call *)
PROCEDURE SVCArguments(context: ContextID): ADDRESS;

    VAR
        argaddress:            (* the address of the parameters *)
            ADDRESS;

BEGIN

    (* first find location of address of previous activation record *)
    argaddress := context↑.currentactivation + 1;

    (* now get the address of its parameters *)
    argaddress := ADDRESS(argaddress↑) + 4;

    (* Make sure it was an SVC trap *)
    IF CARDINAL(argaddress↑) <= CARDINAL(WRITESVC) THEN
        RETURN argaddress;
    ELSE
        RETURN NIL;
    END; (* IF *)

END SVCArguments;

BEGIN
END ContextHandler.
```

```
DEFINITION MODULE InterruptHandler;

(***********************************************************************)
(*                                                                     *)
(*  FUNCTION:   This module handles interrupts and produces error messages.  *)
(*                                                                     *)
(*  AUTHOR:          Rick Fisher                                       *)
(*                                                                     *)
(***********************************************************************)

    FROM LocalSystem IMPORT
        (* Types *)
        Exceptioncode;

    EXPORT QUALIFIED

        (* Types *)
        Devicecode, SVCcode,

        (* Procedures *)
        Error;


    TYPE
        Devicecode =  [CARDREADER .. LINEPRINTER];   (* Subrange of Exceptioncode *)
        SVCcode    =  (READSVC, WRITESVC);



    PROCEDURE Error(errorcode: Exceptioncode);
    (*      Print appropriate error messages.                                   *)
    (*                                                                          *)
    (*      PARAMETERS:    errorcode--the type of the error.  Valid errors are: *)
    (*          BADINSTRUCTION:         illegal characters in instruction.      *)
    (*          BOUNDSVIOLATION:        user attempted to access memory         *)
    (*                                      location below the address in       *)
    (*                                      bounds register.                    *)
    (*          CASEINDEX:              case index out of range.                *)
    (*          ENDofDATA:              attempt to read past end-of-file.       *)
    (*          ILLEGALINSTRUCTION:     illegal instruction.                    *)
    (*          MODEVIOLATION:          user attempted to perform super-        *)
    (*                                      visor instruction.                  *)
    (*          NEWJOB:                 "new job" card found unexpectedly.      *)
    (*          OUTofRANGE:             inaccessible memory location.           *)
    (*          STACKOVERFLOW:          stack overflow.                         *)
    (*          UNDEFINEDINSTRUCTION:   no such M-Code instruction defined.     *)

END InterruptHandler.
```

```
IMPLEMENTATION MODULE InterruptHandler;

(************************************************************************)
(*                                                                      *)
(*  POLICY:     The interrupt handling procedure is invisible to the rest of    *)
(*              the operating system.  All detected errors cause abortion of    *)
(*              the user's program, after a message is printed.  Procedures     *)
(*              in the module cannot be interrupted except by device interrupts. *)
(*                                                                      *)
(*  AUTHOR:     Rick Fisher                                             *)
(*                                                                      *)
(************************************************************************)

    FROM SYSTEM IMPORT
        (* Procedures *)
        ADDRESS;

    FROM LocalSystem IMPORT
        (* Types *)
        Exceptioncode,

        (* Variables *)
        interrupthandler,

        (* Procedures *)
        Trap;

    FROM ContextHandler IMPORT
        (* Types *)
        ContextID, Mcodeinstruction,

        (* Procedures *)
        Reset, SwitchContext, SVCArguments;

    FROM IO IMPORT
        (* Types *)
        Cardbuffer, Linebuffer,

        (* Procedures *)
        DeviceInterrupt, DoWrite, SizedRead, SizedWrite;

    FROM Scheduler IMPORT
        (* Procedures *)
        CleanUp, GetNewJob;
```

```
TYPE
    Processinterrupt=   PROCEDURE(CARDINAL, ContextID);

    (* Subrange of Exceptioncode *)
    Trapcode    =   [BOUNDSVIOLATION .. UNDEFINEDINSTRUCTION];


(* Handle errors *)
PROCEDURE Error(errorcode: Exceptioncode);
BEGIN

    (* Write appropriate error message *)
    CASE errorcode OF
        BADINSTRUCTION:
            DoWrite('ERROR: Illegal characters encountered in instruction.');
    |   ENDofDATA:
            DoWrite('ERROR: Attempt to read past end of File.');
    |   NEWJOB:
            DoWrite('ERROR: Unexpected end of Input.');
            DoWrite('Job Done.');
    |   ILLEGALINSTRUCTION:
            DoWrite('ERROR: Illegal instruction encountered');
    |   OUTofRANGE:
            DoWrite('ERROR: Card contains reference to inaccessible memory location.');
    |   BOUNDSVIOLATION:
            DoWrite('ERROR: Attempt to access illegal memory location.');
    |   CASEINDEX:
            DoWrite('ERROR: Case statment tag is out of range.');
    |   MODEVIOLATION:
            DoWrite('ERROR: Attempt to perform privileged instruction.');
    |   STACKOVERFLOW:
            DoWrite('ERROR: Stack overflow.');
    |   UNDEFINEDINSTRUCTION:
            DoWrite('ERROR: No action has been defined for this M-Code instruction.');
    ELSE
        DoWrite('Undefined error.');
    END; (* CASE *)
END Error;


(* Handle interrupts *)
PROCEDURE ServiceInterrupt(interrupt: CARDINAL; formercontext: ContextID);

    VAR
        instruction:    (* pointer to an M-Code instruction *)
            Mcodeinstruction;
        reason:         (* the CARDINAL parameter translated to Trapcode *)
            Trapcode;
```

```
BEGIN

    (* process interrupt *)
    reason := VAL(Trapcode, interrupt);
    CASE reason OF
        BOUNDSVIOLATION .. MODEVIOLATION,
        STACKOVERFLOW .. UNDEFINEDINSTRUCTION:
            Error(reason);
            CleanUp;
            GetNewJob(formercontext);
    |   Halt:
            CleanUp;
            GetNewJob(formercontext);
            Reset(formercontext);
    |   CARDREADER .. LINEPRINTER:
            DeviceInterrupt(reason);
    |   SVC:
            ProcessSVC(formercontext);
    END; (* CASE *)

    SwitchContext(formercontext);
END ServiceInterrupt;


(* Process supervisor calls *)
PROCEDURE ProcessSVC(formercontext: ContextID);

    VAR
        cardptr:            (* pointer to argument to READ SVC *)
            POINTER TO Cardbuffer;
        lineptr:            (* pointer to argument to WRITE SVC *)
            POINTER TO Linebuffer;
        size:               (* size of argument to SVC *)
            CARDINAL;
        svc:                (* the supervisor call *)
            SVCcode;
        svcargs:            (* the address of the arguments to the SVC *)
            ADDRESS;

BEGIN
    svcargs := SVCArguments(formercontext);
    svc := SVCcode(svcargs↑);
    INC(svcargs);
    IF svc = READSVC THEN
        cardptr := ADDRESS(svcargs↑);
        INC(svcargs);
        size := CARDINAL(svcargs↑);

        SizedRead(cardptr↑, size);
```

```
            (* if job card found in job, abort job and prepare to process next job *)
            IF cardptr↑[0] = '/' THEN
                Error(ENDofDATA);
                GetNewJob(formercontext);
            END; (* IF *)
        ELSE (* svc = WRITESVC *)
            lineptr := ADDRESS(svcargs↑);
            INC(svcargs);
            size := CARDINAL(svcargs↑);

            SizedWrite(lineptr↑, size);
        END; (* IF *)
    END ProcessSVC;


    PROCEDURE SetInterruptHandler(routine: Processinterrupt);
    (*      Store a pointer to the interrupt handling routine.                                    *)
    (*                                                                                            *)
    (*      PARAMETERS:     routine—a procedure variable with the value of the                    *)
    (*                              exception handling routine.                                   *)

    BEGIN
        interrupthandler := PROC(routine);
    END SetInterruptHandler;


BEGIN (* module initialization *)

    (* Set the location of the interrupt handling routine *)
    SetInterruptHandler(ServiceInterrupt);

END InterruptHandler.
```

```
DEFINITION MODULE IO;

(**********************************************************************)
(*                                                                    *)
(*  FUNCTION:  This module allows low level input/output: all reading is done  *)
(*             from a card reader, all writing is to a line printer.  *)
(*                                                                    *)
(*  AUTHOR:    Rick Fisher                                            *)
(*                                                                    *)
(**********************************************************************)

    FROM InterruptHandler IMPORT
        (* Types *)
        Devicecode;

    EXPORT QUALIFIED
        (* Constants *)
        BLANK,

        (* Types *)
        Cardbuffer, Linebuffer,

        (* Procedures *)
        DeviceInterrupt, DoRead, DoWrite, SizedRead, SizedWrite;


    CONST
        BLANK        =  ' ';
        CARDLENGTH   =  80;
        LINELENGTH   =  132;

    TYPE
        Cardbuffer   =  ARRAY [0..CARDLENGTH] OF CHAR;
        Linebuffer   =  ARRAY [0..LINELENGTH] OF CHAR;


    PROCEDURE DeviceInterrupt(device: Devicecode);
    (*      Handle device interrupts.                                 *)
    (*                                                                *)
    (*      PARAMETERS:   device–the type of device responsible for the  *)
    (*                        interrupt. Valid devices are:           *)
    (*                            CARDREADER:    the card reader.     *)
    (*                            LINEPRINTER:   the line printer.    *)
```

```
PROCEDURE DoRead(VAR buffer: ARRAY OF CHAR);
(*      Read the next card in the card reader.  Calls SizedRead.              *)
(*                                                                             *)
(*      PARAMETERS:     buffer–the variable into which the card should be      *)
(*                              read.  If the buffer is not large enough to hold *)
(*                              the entire contents of the card, the rest of the *)
(*                              card will be lost.                             *)


PROCEDURE DoWrite(buffer: ARRAY OF CHAR);
(*      Send a line of output to the line printer.  Will truncate lines longer than *)
(*      132 characters.  Calls SizedWrite.                                     *)
(*                                                                             *)
(*      PARAMETERS:     buffer–the variable holding the characters to be       *)
(*                              written.                                       *)


PROCEDURE SizedRead(VAR buffer: ARRAY OF CHAR; size: CARDINAL);
(*      Read the next card in the card reader.  May be used instead of DoRead  *)
(*      if user already knows the size in bytes of the buffer variable, or if  *)
(*      reading into the first portion of the variable only is desired.        *)
(*                                                                             *)
(*      PARAMETERS:     buffer–the variable into which the card should be      *)
(*                              read.                                          *)
(*                      size–the number of characters to be read (should       *)
(*                              not exceed the size of the buffer!).           *)

PROCEDURE SizedWrite(buffer: ARRAY OF CHAR; size: CARDINAL);
(*      Send a line of output to the line printer.  Will truncate lines longer than *)
(*      132 characters.  May be used instead of DoWrite if user already knows  *)
(*      the size in bytes of the buffer variable, or if only the first portion of that *)
(*      variable is to be written.                                             *)
(*                                                                             *)
(*      PARAMETERS:     buffer–the variable holding the characters to be       *)
(*                              written.                                       *)
(*                      size–the number of characters to be written (should    *)
(*                              not exceed the size of the buffer!).           *)

END IO.
```

```
IMPLEMENTATION MODULE IO;

(**********************************************************************)
(*                                                                    *)
(*  POLICY:      This module reads in blocks of 80 bytes from a card reader or    *)
(*               writes in blocks of 132 bytes to a line printer.       *)
(*                                                                    *)
(*  AUTHOR:      Rick Fisher                                          *)
(*                                                                    *)
(**********************************************************************)

  FROM LocalSystem IMPORT
    (* Variables *)
    inputbuffer, outputbuffer,

    (* Procedures *)
    Read, Write;

  FROM InterruptHandler IMPORT
    (* Types *)
    Devicecode;

  CONST
    NEWLINECHAR  =  12C;

  VAR
    card:            (* holds a card's worth of characters *)
      Cardbuffer;

    cardindex:       (* positions of inputbuffer filled *)
      CARDINAL;
    line:            (* holds a line's worth of characters *)
      Linebuffer;
    lineindex:       (* positions of outputbuffer filled *)
      CARDINAL;
    printerdone,     (* becomes true when cardreader finishes *)
    readerdone:      (* becomes true when lineprinter finishes *)
      BOOLEAN;


  (*  Handle device interrupts.   *)
  PROCEDURE DeviceInterrupt(device: Devicecode);

  BEGIN
    IF device = CARDREADER THEN

      (* read entire card before signaling device completion *)
      IF (inputbuffer = NEWLINECHAR) OR (cardindex = CARDLENGTH) THEN
        card[cardindex] := BLANK;
        cardindex := 0;
        readerdone := TRUE;
        ELSE
            card[cardindex] := inputbuffer;
            INC(cardindex);
            Read;
        END; (* IF inputbuffer *)
    ELSIF device = LINEPRINTER THEN

        (* print entire line before signaling device completion *)
        IF line[lineindex] = NEWLINECHAR THEN
            outputbuffer := NEWLINECHAR;
            lineindex := 0;
            printerdone := TRUE;
        ELSE
            outputbuffer := line[lineindex];
            INC(lineindex);
            Write;
        END; (* IF line[lineindex] *)
    END; (* IF device *)
END DeviceInterrupt;


(*  Read one card.  *)
PROCEDURE DoRead(VAR buffer: ARRAY OF CHAR);

BEGIN
    SizedRead( buffer, HIGH(buffer) );
END DoRead;

(*  Write one line.  *)
PROCEDURE DoWrite(buffer: ARRAY OF CHAR);

BEGIN
    SizedWrite( buffer, HIGH(buffer) );
END DoWrite;


(*  Read a specified number of characters.  *)
PROCEDURE SizedRead(VAR buffer: ARRAY OF CHAR; size: CARDINAL);

    VAR
        i:  CARDINAL;   (* loop index *)

BEGIN
    IF size > CARDLENGTH THEN
        size := CARDLENGTH;
    END;

    (* do nothing until card reader finished *)
    REPEAT
        (* nothing *)
    UNTIL readerdone;
```

```
        (* double-buffer *)
        FOR i := 0 TO size DO
            buffer[i] := card[i];
        END;

        (* begin reading next card *)
        readerdone := FALSE;
        Read;
    END SizedRead;

    (* Write a specified number of characters on one line *)
    PROCEDURE SizedWrite(buffer: ARRAY OF CHAR; size: CARDINAL);

    VAR
        i:        CARDINAL;(* loop index *)

    BEGIN

        (* truncate lines longer than the size of a linebuffer *)
        IF size >= LINELENGTH THEN
            size := LINELENGTH - 1;
        END;

        (* do nothing until line printer is finished *)
        REPEAT
            (* nothing *)
        UNTIL printerdone;

        (* double-buffer *)
        FOR i := 0 TO size DO
            line[i] := buffer[i];
        END;
        line[size+1] := NEWLINECHAR;

        (* begin printing next line *)
        printerdone := FALSE;
        outputbuffer := line[0];
        Write;
    END SizedWrite;

BEGIN (* module initialization *)

    (* both buffers are initially empty *)
    cardindex := 0;
    lineindex := 0;

    (* line printer is initially idle, while card reader will be started by main module *)
    printerdone := TRUE;
    readerdone := FALSE;

END IO.
```

```
DEFINITION MODULE Loader;

(**********************************************************************)
(*                                                                    *)
(*  FUNCTION:   This module loads program instructions into memory.   *)
(*                                                                    *)
(*  AUTHOR:     Rick Fisher                                           *)
(*                                                                    *)
(**********************************************************************)

  FROM ContextHandler IMPORT
    (* Types *)
    ContextID;

  EXPORT QUALIFIED
    (* Constants *)
    USERDATAFRAME, USERSTACKBASE, USERSTACKSIZE,

    (* Procedures *)
    LoadJob;

  CONST
    USERDATAFRAME  =  4000;
    USERSTACKBASE  = 10000;
    USERSTACKSIZE  =  6000;


  PROCEDURE LoadJob(usercontext: ContextID; VAR valid: BOOLEAN);
  (*     Read in and load a job, in preparation for running it.               *)
  (*                                                                          *)
  (*     PARAMETERS:    usercontext–the ID of the user context.               *)
  (*                    valid–TRUE when a job has been successfully loaded.   *)

END Loader.
```

```
IMPLEMENTATION MODULE Loader;

(**********************************************************************)
(*                                                                    *)
(*  POLICY:      Invalid instructions cause the program to be aborted.*)
(*                                                                    *)
(*  AUTHOR:      Rick Fisher                                          *)
(*                                                                    *)
(**********************************************************************)

    FROM SYSTEM IMPORT
        (* Types *)
        WORD, ADDRESS,

        (* Constants *)
        MAXCARD;

    FROM LocalSystem IMPORT
        (* Types *)
        Exceptioncode,

        (* Procedures *)
        SetBoundsRegister;

    FROM ContextHandler IMPORT
        (* Constants *)
        HIGHINSTRUCTION,

        (* Types *)
        ContextID, Mcodeinstruction,

        (* Procedures *)
        SetPC;

    FROM InterruptHandler IMPORT
        (* Procedures *)
        Error;

    FROM IO IMPORT
        (* Constants *)
        BLANK,

        (* Types *)
        Cardbuffer,

        (* Procedures *)
        DoRead, DoWrite;

    FROM Scheduler IMPORT
        (* Procedures *)
        CleanUp;
```

```
(*  Read, echo and load program instructions.   *)
PROCEDURE LoadJob(usercontext: ContextID; VAR valid: BOOLEAN);

VAR
    card:              (* one card read by the card reader *)
        Cardbuffer;
    location:          (* address for next program instruction *)
        ADDRESS;

BEGIN
    valid := TRUE;
    location := USERDATAFRAME;

    (* read and load cards until end of program *)
    DoRead(card);
    WHILE (card[0] # '$') & valid DO

        (* check for unexpected job card before echoing and loading memory *)
        IF card[0] = '/' THEN
            Error(NEWJOB);
            location := USERDATAFRAME;
            valid := FALSE;
        ELSE
            DoWrite(card);
            LoadInstruction(card, location, valid);
            IF valid THEN
                DoRead(card);
            ELSE
                CleanUp;
            END; (* IF valid *)
        END; (* IF card[0] *)
    END; (* WHILE *)

    SetPC( usercontext, ADDRESS(USERDATAFRAME) );
END LoadJob;


(* Load Memory *)
PROCEDURE LoadInstruction(card: Cardbuffer; VAR location: ADDRESS;
    VAR cardvalid: BOOLEAN);

    CONST
        WORDSIZE       =   1;
        MAXADDRESS     =   MAXCARD;
        VAR
            instruction:        (* program instruction *)
                Mcodeinstruction;
            i:                  (* loop index *)
                CARDINAL;

    BEGIN
        i := 0;
        instruction := 0;

        (* translate numerals of instruction into an octal integer *)
        WHILE (card[i] # BLANK) & (card[i] >= '0') & (card[i] <= '7') DO
            instruction := ORD(card[i]) - ORD('0') + instruction * 10B;
            INC(i);
        END; (* WHILE *)

        (* store instruction in memory unless format is bad *)
        IF (card[i] = BLANK) & (instruction <= HIGHINSTRUCTION) THEN
            location↑ := WORD(instruction);
            INC(location, WORDSIZE);
        ELSIF card[i] # BLANK THEN
            Error(BADINSTRUCTION);
            cardvalid := FALSE;
        ELSE
            Error(OUTofRANGE);
            cardvalid := FALSE;
        END  (* IF *)
    END LoadInstruction;

BEGIN (* module initialization *)

    SetBoundsRegister(USERDATAFRAME);

END Loader.
```

```
DEFINITION MODULE LocalSystem;

(***************************************************************************)
(*                                                                         *)
(*  FUNCTION:   This is a pseudo-module containing descriptions of machine *)
(*              dependent features. Because the objects imported from      *)
(*              LocalSystem obey special rules and are implemented directly*)
(*              in M-Code, the module must be known to the linker or compiler, *)
(*              and a definition module is necessary merely for documentation *)
(*              and to allow importing modules to compile properly. Modules *)
(*              making use of features exported from LocalSystem are considered *)
(*              low-level, system-dependent modules, and are therefore     *)
(*              non-portable.                                              *)
(*                                                                         *)
(*  AUTHORS:    Rick Fisher and Rick Snodgrass                             *)
(*                                                                         *)
(***************************************************************************)
   FROM SYSTEM IMPORT
      (* Types *)
      ADDRESS;

   EXPORT QUALIFIED
      (* Types *)
      Exceptioncode,

      (* Variables *)
      currentcontext, OScontext, inputbuffer, interrupthandler, outputbuffer,

      (* Procedures *)
      ContextSwitch, Read, SetBoundsRegister, Write, Trap;


   TYPE
      Exceptioncode =  (BADINSTRUCTION, ENDofDATA, ILLEGALINSTRUCTION,
                        NEWJOB, OUTofRANGE, BOUNDSVIOLATION, CASEINDEX,
                        MODEVIOLATION, CARDREADER, LINEPRINTER,
                        Halt, STACKOVERFLOW, UNDEFINEDINSTRUCTION, SVC);

   VAR
      currentcontext[4]:      (* address of current context *)
         ADDRESS;
      OScontext[5]:           (* address of operating system context *)
         ADDRESS;
      inputbuffer[10],        (* cardreader's one-byte register *)
      outputbuffer[11]:       (* lineprinter's one-byte register *)
         CHAR;

      interrupthandler[17]:   (* interrupt routine for traps *)
         PROC;
```

```
PROCEDURE ContextSwitch(context: ADDRESS);
(*      Return to a previously stored context. Should only be called by            *)
(*      ContextHandler.SwitchContext.                                              *)
(*                                                                                 *)
(*      PARAMETERS:     context–a pointer to the previously stored context.       *)


PROCEDURE Read;
(*      Starts the card reader and returns. The card reader will run concurrently  *)
(*      with the CPU and deposit the next byte read into the reserved memory location *)
(*      ''inputbuffer''                                                            *)


PROCEDURE SetBoundsRegister(location: ADDRESS);
(*      Set the contents of the Bounds Register to be the lower limit at which     *)
(*      a program can be stored in memory. A reference while in User Mode          *)
(*      to an address below that pointed to by the Bounds Register will cause      *)
(*      an OUTofBOUNDS interrupt.                                                  *)
(*                                                                                 *)
(*      PARAMETERS:     location–the lowest address to which the user has         *)
(*                              access.                                            *)


PROCEDURE Trap(reason: CARDINAL);
(*      Store the current context and load the context of the operating system,    *)
(*      then call the interrupt handling routine with ''reason'' as a parameter.   *)
(*      Should only be called through the InterruptHandler or SVCalls.             *)
(*                                                                                 *)
(*      PARAMETERS:     reason–the cause of the trap.                             *)


PROCEDURE Write;
(*      Starts the line printer and returns. The line printer will run concurrently *)
(*      with the CPU and print the byte currently in the reserved memory           *)
(*      location ''outputbuffer''                                                  *)

END LocalSystem.
```

```
IMPLEMENTATION MODULE LocalSystem;

(*************************************************************************)
(*                                                                       *)
(*  POLICY:       None.  At present this module is known to the linker, not the    *)
(*                compiler, so definition and implementation modules must both    *)
(*                exist for the purposes of compilation.                 *)
(*                                                                       *)
(*  AUTHOR:       Rick Fisher                                            *)
(*                                                                       *)
(*************************************************************************)
    FROM SYSTEM IMPORT
        (* Types *)
        ADDRESS;

    PROCEDURE ContextSwitch(to: ADDRESS);                  (*  CNTX = 246   *)
    BEGIN
    END ContextSwitch;


    PROCEDURE Read;                                        (*  READ = 240   *)
    BEGIN
    END Read;


    PROCEDURE SetBoundsRegister(location: ADDRESS);        (*  SBR = 214    *)
    BEGIN
    END SetBoundsRegister;


    PROCEDURE Trap(reason: CARDINAL);                      (*  TRAP = 304   *)
    BEGIN
    END Trap;

    PROCEDURE Write;                                       (*  WRITE = 241  *)
    BEGIN
    END Write;

BEGIN
END LocalSystem.
```

```
DEFINITION MODULE Scheduler;

(**********************************************************************)
(*                                                                    *)
(*  FUNCTION:   This module handles aborts and directs the loader to load the   *)
(*              next job.                                             *)
(*                                                                    *)
(*  AUTHOR:     Rick Fisher                                           *)
(*                                                                    *)
(**********************************************************************)

  FROM ContextHandler IMPORT
    (* Types *)
    ContextID;

  EXPORT QUALIFIED
    (* Procedures *)
    CleanUp, GetNewJob;


  PROCEDURE CleanUp;
  (*      Clean up remaining cards after a program has finished or been aborted.    *)


  PROCEDURE GetNewJob (userContext: ContextID);
  (*      Direct the loader to load the first job with no internal errors.          *)

END Scheduler.
```

```
IMPLEMENTATION MODULE Scheduler;

(*************************************************************************)
(*                                                                       *)
(*  POLICY:      Jobs are aborted if improper JCL or a bad instruction is *)
(*               encountered.                                            *)
(*                                                                       *)
(*  AUTHOR:      Rick Fisher                                             *)
(*                                                                       *)
(*************************************************************************)

    FROM ContextHandler IMPORT
        (* Types *)
        ContextID;

    FROM IO IMPORT
        (* Types *)
        Cardbuffer,

        (* Procedures *)
        DoRead, DoWrite;

    FROM Loader IMPORT
        (* Procedures *)
        LoadJob;


    (* Clean up last job *)
    PROCEDURE CleanUp;

        VAR
            card:           (* one card read by the card reader *)
                Cardbuffer;

    BEGIN

        (* read cards until a ''new job'' card is found *)
        DoRead(card);
        WHILE card[0] # '/' DO
            DoRead(card);
        END;
    END CleanUp;
    (* Find a new job *)
    PROCEDURE GetNewJob (usercontext: ContextID);

        VAR
            jobvalid:    BOOLEAN;

    BEGIN
        jobvalid := FALSE;
        WHILE NOT jobvalid DO
            DoWrite('Job Done.');
            LoadJob(usercontext, jobvalid);
        END;
    END GetNewJob;

BEGIN
END Scheduler.
```

```
DEFINITION MODULE SVCalls;

(*************************************************************************)
(*                                                                       *)
(*  FUNCTION:    This module defines the user available procedures of Read and  *)
(*               Write.                                                  *)
(*                                                                       *)
(*  AUTHOR:      Rick Fisher                                             *)
(*                                                                       *)
(*************************************************************************)

  EXPORT QUALIFIED
    (* Procedures *)
    Read, Write;


    PROCEDURE Read(VAR card: ARRAY OF CHAR);
    (*      Read one card.                                               *)
    (*                                                                   *)
    (*      PARAMETERS:     card–the buffer for array of characters to   *)
    (*                          be read.                                 *)


    PROCEDURE Write(line: ARRAY OF CHAR);
    (*      Write one line..                                             *)
    (*                                                                   *)
    (*      PARAMETERS:     line–the buffer holding the array of characters  *)
    (*                          to be written.                           *)

END SVCalls.
```

```
IMPLEMENTATION MODULE SVCalls;

(************************************************************************)
(*                                                                      *)
(*  POLICY:       None to speak of.                                     *)
(*                                                                      *)
(*  AUTHOR:       Rick Fisher                                           *)
(*                                                                      *)
(************************************************************************)

    FROM LocalSystem IMPORT
        (* Types *)
        Exceptioncode,

        (* Procedures *)
        Trap;

    FROM InterruptHandler IMPORT
        (* Types *)
        SVCcode;

    (*  Supervisor call Read.    *)
    PROCEDURE Read(VAR card: ARRAY OF CHAR);
    BEGIN
        SVCTrap(READSVC, card);
    END Read;

    (*  Supervisor call Write.   *)
    PROCEDURE Write(line: ARRAY OF CHAR);
    BEGIN
        SVCTrap(WRITESVC, line);
    END Write;

    (*      Cause a trap to the correct supervisor call.                    *)
    PROCEDURE SVCTrap(svc: SVCcode; VAR buffer: ARRAY OF CHAR);
    (*                                                                      *)
    (*      PARAMETERS:     svc–the correct supervisor call. May be one of: *)
    (*                          READSVC:    supervisor call to read from the *)
    (*                                      card reader                     *)
    (*                          WRITESVC:   supervisor call to write to the *)
    (*                                      line printer                    *)
    (*                      buffer–the buffer variable designated by the user. *)
    (*                                                                      *)
    (*      Parameters are accessed by UniBatch via ContextHandler.SVCArguments. *)

    BEGIN
        Trap( CARDINAL(SVC) );
    END SVCTrap;

BEGIN
END SVCalls.
```

```
MODULE UniBatch;

(**********************************************************************)
(*                                                                    *)
(*  FUNCTION:   Although this is the main program module, it serves only to    *)
(*              create a user context, get a new job associated with that context,  *)
(*              and switch to that context.                           *)
(*                                                                    *)
(*  AUTHOR:        Rick Fisher                                        *)
(*                                                                    *)
(**********************************************************************)

    FROM SYSTEM IMPORT
        (* Types *)
        ADDRESS;

    FROM LocalSystem IMPORT
        (* Variables *)
        currentcontext, OScontext,

        (* Procedures *)
        Read;

    FROM ContextHandler IMPORT
        (* Types *)
        ContextID,

        (* Procedures *)
        NewContext, SwitchContext;

    FROM Loader IMPORT
        (* Constants *)
        USERDATAFRAME, USERSTACKBASE, USERSTACKSIZE;

    FROM Scheduler IMPORT
        (* Procedures *)
        CleanUp, GetNewJob;

    VAR
        context:    (* the user context *)
            ContextID;

(* Prepare for first batch job; system starts in Supervisor mode at memory location 0 *)
BEGIN (* UniBatch *)

    (* save O/S context before being interrupted *)
    OScontext := currentcontext;

    (* start card reader *)
    Read;
    (* prepare user context and switch to it *)
    context := NewContext( ADDRESS(USERDATAFRAME), ADDRESS(USERSTACKBASE),
        USERSTACKSIZE );
    CleanUp;
    GetNewJob(context);
    SwitchContext(context);

END UniBatch.
```

APPENDIX B

MULTIBATCH

```
DEFINITION MODULE Clock;

(*************************************************************************)
(*                                                                       *)
(*  FUNCTION:    The Clock is responsible for keeping track of time. It does so  *)
(*               by counting the number of interrupts (ticks) caused by the clock *)
(*               device. It also initiates events which have been set to occur   *)
(*               after a certain number of ticks.                                *)
(*                                                                       *)
(*  AUTHOR:      Rick Fisher                                             *)
(*                                                                       *)
(*************************************************************************)

    EXPORT QUALIFIED
        (* Constants *)
        TICKSPERSECOND,

        (* Procedures *)
        TickCount;


    CONST
        TICKSPERSECOND   =   100;     (* The clock device has not yet been written, so
                                         this constant is just a guess. *)


    PROCEDURE TickCount(): CARDINAL;
    (*  RETURNS the time in ticks. The time starts at 0 at system initialization.      *)

END Clock.
```

```
IMPLEMENTATION MODULE Clock(*[11]*);

(************************************************************************)
(*                                                                      *)
(*  POLICY:     The Clock counts one tick on each interrupt. Periodically, it  *)
(*              asks the High Level Scheduler to see if it is a good time to load *)
(*              a new job; more frequently it checks the event list to see if any *)
(*              events are due to be executed. Both of these actions are done  *)
(*              through traps, so as to ensure that as little time as possible is *)
(*              spent in this uninterruptible module.                   *)
(*                                                                      *)
(*  AUTHOR:     Rick Fisher                                             *)
(*                                                                      *)
(************************************************************************)

    FROM SYSTEM IMPORT
        (* Constants *)
        MAXCARD;

    FROM VirtualMachine IMPORT
        (* Types *)
        Interruptcode, OSTraps,

        (* Procedures *)
        SetInterruptHandler, Trap;


    CONST
        NEWJOBFREQUENCY             =  16;
        SCHEDULECHECKFREQUENCY      =   4;

    VAR
        tickcounter: (* number of ticks so far *)
            CARDINAL;


    (*  The number of ticks so far.  *)
    PROCEDURE TickCount(): CARDINAL;

    BEGIN
        RETURN tickcounter;
    END TickCount;
    PROCEDURE ClockInterruptHandler;
    (*      Handle a clock interrupt. Periodically set traps to check for new jobs,      *)
    (*      and to examine the list of time scheduled events.                            *)

    BEGIN
        tickcounter := (tickcounter + 1) MOD MAXCARD;

        IF tickcounter MOD NEWJOBFREQUENCY = 0 THEN
            Trap(SHOULDERTAP);
        ELSIF tickcounter MOD SCHEDULECHECKFREQUENCY = 0 THEN
            Trap(CHECKSCHEDULE);
        END; (* IF *)

    END ClockInterruptHandler;


BEGIN (* module initialization *)
    tickcounter := 0;
    SetInterruptHandler(CLOCK, ClockInterruptHandler);
END Clock.
```

```
DEFINITION MODULE DiskManager;

(**********************************************************************)
(*                                                                    *)
(*  FUNCTION:   The disk manager is responsible for the allocation of diskspace, and  *)
(*              supports the disk read and write operations.          *)
(*                                                                    *)
(*  AUTHOR:     Rick Fisher                                           *)
(*                                                                    *)
(**********************************************************************)

  FROM SYSTEM IMPORT
    (* Types *)
    WORD;

  FROM MemoryManager IMPORT
    (* Types *)
    MemoryblockID;

  EXPORT QUALIFIED
    (* Constants *)
    CYLINDERCOUNT, CYLINDERSIZE, DISKSIZE, NULL, SECTORSIZE,

    (* Types *)
    DiskblockID, Diskcompletion,

    (* Procedures *)
    Allocate, Deallocate, DiskRead, DiskWrite, InitDisk, Null;


  CONST
    CYLINDERSIZE     =  128;                    (* 128 sectors/cylinder *)
    CYLINDERCOUNT    =    8;
    DISKSIZE         =  CYLINDERSIZE * CYLINDERCOUNT;
    NULL             =    0;                    (* for use in procedure Null *)
    SECTORSIZE       =  128;


  TYPE
    DiskblockID;            (* Pointer to the disk block type *)
    Diskcompletion   =  PROCEDURE(WORD);
```

```
PROCEDURE Allocate(size: CARDINAL): DiskblockID;
(*      Allocate a disk block whose size is the smallest SECTORSIZE multiple     *)
(*      equal to or greater than the requested size in words. Return a disk-     *)
(*      block descriptor, or DiskblockID(NIL) if unsuccessful. Blocks are        *)
(*      allocated by a First Fit strategy.                                       *)
(*                                                                               *)
(*      PARAMETERS:     size–the size in words to be allocated.                  *)
(*      RETURNS the ID of the diskblock allocated; if not successful,            *)
(*          DiskblockID(NIL) is returned.                                        *)


PROCEDURE Deallocate(VAR diskblock: DiskblockID);
(*      Free the specified disk block for reuse.                                 *)
(*                                                                               *)
(*      PARAMETERS:     diskblock–the diskblock to be deallocated.               *)


PROCEDURE DiskRead(diskblock: DiskblockID; memoryblock: MemoryblockID;
    notify: Diskcompletion; parameter: WORD);
(*      Read from disk and load into memory, taking the appropriate action       *)
(*      when finished. Reading starts with the named diskblock and continues     *)
(*      until the diskblock is completely read or the memoryblock is filled.     *)
(*                                                                               *)
(*      PARAMETERS:     diskblock–the block on the disk being read.              *)
(*                      memoryblock–the block of memory being written to.        *)
(*                      notify–the procedure which constitutes the correct       *)
(*                          action to take when the Read has completed.          *)
(*                      parameter–the parameter to the notification procedure.   *)


PROCEDURE DiskWrite(memoryblock: MemoryblockID; diskblock: DiskblockID;
    notify: Diskcompletion; parameter: WORD);
(*      Write from memory to disk, taking the appropriate action when            *)
(*      finished. Writing stops when the memoryblock has been completely         *)
(*      written, or when the diskblock is filled.                                *)
(*                                                                               *)
(*      PARAMETERS:     memoryblock–the block of memory being written.           *)
(*                      diskblock–the block of the disk being written to.        *)
(*                      notify–the procedure which constitutes the correct       *)
(*                          action to take when the Write has completed.         *)
(*                      parameter–the parameter to the notification procedure.   *)


PROCEDURE InitDisk;
(*      Order-dependent module initialization. This procedure consists of        *)
(*      initialization statements which should not be executed until after       *)
(*      the initialization for the main module has begun.                        *)
```

```
    PROCEDURE Null(null: WORD);
    (*      This procedure does nothing; it serves as a dummy parameter to DiskRead      *)
    (*      or DiskWrite when no action is demanded upon completion of the               *)
    (*      operation.                                                                   *)
    (*                                                                                   *)
    (*      PARAMETERS:     null–can be any WORD: for example the constant               *)
    (*                          NULL.                                                    *)

END DiskManager.
```

```
IMPLEMENTATION MODULE DiskManager(*[10]*);

(*************************************************************************)
(*                                                                       *)
(*  POLICY:      Blocks are allocated as an arbitrary number of contiguous sectors   *)
(*               of 128 bytes each.  Deallocated blocks are merged with their        *)
(*               neighbors when it is possible.  Blocks are not formed across        *)
(*               cylinder boundaries, so there are always at least as many blocks    *)
(*               on the disk as cylinders.                               *)
(*                                                                       *)
(*  AUTHOR:      Rick Fisher                                             *)
(*                                                                       *)
(*************************************************************************)

  FROM SYSTEM IMPORT
    (* Types *)
    ADDRESS, WORD,

    (* Constants *)
    BYTESPERWORD,

    (* Procedures *)
    ADR, SIZE;

  IMPORT VirtualMachine;
  FROM VirtualMachine IMPORT
    (* Types *)
    Interruptcode,

    (* Procedures *)
    (* DiskRead, DiskWrite, *) ReturnFromInterrupt, SetInterruptHandler;

  FROM MemoryManager IMPORT
    (* Types *)
    MemoryblockID,

    (* Procedures *)
    BlockSize, StartingAddress;

  FROM STORAGE IMPORT
    (* Procedures *)
    ALLOCATE, DEALLOCATE;


  TYPE
    Blockstatus    =   (USED, FREE);
    CylinderID     =   [0 .. CYLINDERCOUNT - 1];
    SectorID       =   [0 .. DISKSIZE - 1];
    DiskblockID   =   POINTER TO Diskblocktype;
    Diskblocktype =   RECORD
                          lowerbound,        (* first sector in block *)
                          upperbound:        (* last sector in block *)
                              SectorID;
                          status:            (* FREE or USED *)
                              Blockstatus;
                          next,              (* next diskblock *)
                          previous:          (* previous diskblock *)
                              DiskblockID;
                      END;
    Diskbuffer    =   POINTER TO ARRAY [0 .. SECTORSIZE - 1] OF WORD;
    Diskoperation =   (READ, WRITE);
    LooseEnds     =   RECORD
                          transferfinished:  (* true for last sector of block transfer *)
                              BOOLEAN;
                          count:             (* number of bytes involved in final transfer *)
                              CARDINAL;
                          buffer:            (* starting address of final transfer *)
                              ADDRESS;
                          notify:            (* procedure to be called at completion of
                                                     transfer *)
                              Diskcompletion;
                          parameter:         (* parameter to the above procedure *)
                              WORD;
                      END;


CONST
    WORDSPERSECTOR =    SECTORSIZE DIV BYTESPERWORD;


VAR
    listheader:       (* pointer to the list of diskblocks *)
        DiskblockID;

    lastbuffer:       (* temporary holding place for last sector of a disk transfer *)
        ARRAY[0 .. WORDSPERSECTOR - 1] OF WORD;


(*   Allocate a disk block.     *)
PROCEDURE Allocate(size: CARDINAL): DiskblockID;

    VAR
        currentblock,         (* block currently under examination in list *)
        newblock:             (* newly allocated block *)
            DiskblockID;
        sectorcount:          (* size in sectors required for disk block *)
            CARDINAL;
```

```
BEGIN

    sectorcount := (size - 1 + SECTORSIZE) DIV SECTORSIZE;

    (* search list of disk blocks for first free block of sufficient size *)
    currentblock := listheader;
    WHILE (currentblock # NIL ) & ( (currentblock↑.status = USED) OR
    (currentblock↑.upperbound - currentblock↑.lowerbound + 1 < sectorcount) ) DO
        currentblock := currentblock↑.next;
    END; (* WHILE *)

    (* allocate new block *)
    WITH currentblock↑ DO
        IF currentblock = NIL THEN
        (* no block was found *)

            newblock := NIL;
        ELSIF upperbound - lowerbound + 1 = sectorcount THEN
        (* block of correct size was found *)

            newblock := currentblock;
            newblock↑.status := USED;
        ELSE
        (* larger block than necessary was found *)

            (* create new block record *)
            ALLOCATE( newblock, SIZE(newblock↑) );
            newblock↑.lowerbound := lowerbound;
            newblock↑.upperbound := lowerbound + sectorcount - 1;
            newblock↑.status := USED;
            newblock↑.previous := previous;
            newblock↑.next := currentblock;

            lowerbound := lowerbound + sectorcount;

            (* insert new block record into list *)
            IF previous = NIL THEN
                listheader := newblock;
            ELSE
                previous↑.next := newblock;
                previous := newblock;
            END; (* IF previous *)
        END; (* IF currentblock *)
    END; (* WITH *)

    RETURN(newblock);

END Allocate;
```

```
(*  Free a disk block for reuse.  *)
PROCEDURE Deallocate(VAR diskblock: DiskblockID);

    VAR
        temp:           (* temporary pointer *)
            DiskblockID;

BEGIN
    WITH diskblock↑ DO
        status := FREE;

        (* combine with next block on cylinder, if free *)
        IF (next # NIL) & (next↑.status = FREE) &
        (upperbound DIV CYLINDERSIZE = next↑.lowerbound DIV CYLINDERSIZE)
        THEN
            upperbound := next↑.upperbound;
            temp:= next;
            next := next↑.next;
            DEALLOCATE( temp, SIZE(temp↑) );
        END; (* IF *)

        (* combine with previous block on cylinder, if free *)
        IF (previous # NIL) & (previous↑.status = FREE) &
        (lowerbound DIV CYLINDERSIZE = previous↑.upperbound DIV CYLINDERSIZE)
        THEN
            previous↑.upperbound := upperbound;
            previous↑.next := next;
            DEALLOCATE( diskblock, SIZE(diskblock↑) );
        END; (* IF *)
    END; (* WITH *)

    diskblock := NIL;
END Deallocate;


(*  Copy data from disk to main memory.    *)
PROCEDURE DiskRead(diskblock: DiskblockID; memoryblock: MemoryblockID;
    notify: Diskcompletion; parameter: WORD);

BEGIN
    DiskCopy(READ, diskblock, memoryblock, notify, parameter);
END DiskRead;


(*  Copy data from main memory to disk.    *)
PROCEDURE DiskWrite(memoryblock: MemoryblockID; diskblock: DiskblockID;
    notify: Diskcompletion; parameter: WORD);

BEGIN
    DiskCopy(WRITE, diskblock, memoryblock, notify, parameter);
END DiskWrite;
```

```
PROCEDURE DiskCopy(operation: Diskoperation; diskblock: DiskblockID;
    memoryblock: MemoryblockID; notify: Diskcompletion; parameter: WORD);
(*      Copy information between disk and memory, a sector at a time.           *)
(*                                                                              *)
(*      PARAMETERS:    operation–either READ or WRITE.                          *)
(*                     diskblock–the ID of the diskblock involved.              *)
(*                     memoryblock–the ID of the memoryblock involved.          *)
(*                     notify–the procedure to be executed at the conclusion    *)
(*                         of the operation.                                    *)
(*                     parameter–the parameter to "notify".                     *)

    VAR
        i:                  (* loop index *)
            CARDINAL;
        buffer:             (* pointer to start of memory block *)
            ADDRESS;
        sector:             (* one sector of the disk block *)
            SectorID;
        size:               (* the size of the memory block *)
            CARDINAL;
        tidyup:             (* to be acted upon when transfer is complete *)
            LooseEnds;
        ptr:                (* initialized to address of "lastbuffer" *)
            ADDRESS;

BEGIN

    (* initialize *)
    tidyup.transferfinished := FALSE;
    tidyup.notify := notify;
    tidyup.parameter := parameter;
    buffer := StartingAddress(memoryblock);
    size := BlockSize(memoryblock);

    (* copy all sectors but the last *)
    FOR sector := diskblock↑.lowerbound TO diskblock↑.lowerbound - 1
    + (size - 1) DIV SECTORSIZE DO
        DiskDriver.CopySector(operation, sector, buffer, tidyup);
        INC(buffer, WORDSPERSECTOR);
    END; (* FOR *)

    (* copy last (partial?) sector of block *)
    sector := diskblock↑.lowerbound + (size - 1) DIV SECTORSIZE;
    tidyup.transferfinished := TRUE;
    tidyup.count := (size - 1) MOD SECTORSIZE + 1;
    sector := diskblock↑.lowerbound + (size - 1) DIV SECTORSIZE;
    IF operation = READ THEN
        tidyup.buffer := buffer;
    ELSE (* operation = WRITE *)
        ptr := ADR(lastbuffer);
        FOR i := 0 TO tidyup.count - 1 DO
            ptr↑ := buffer↑;
            INC(buffer);
            INC(ptr);
        END; (* FOR i *)
    END; (* IF operation *)
    DiskDriver.CopySector(operation, sector, ADR(lastbuffer), tidyup);

END DiskCopy;


(*  Module initialization.    *)
PROCEDURE InitDisk;

BEGIN
(* set up initial list of disk blocks with each cylinder consisting of one free block *)

    ALLOCATE( listheader, SIZE(listheader↑) );

    (* initialize node with the values for the last block on the disk *)
    WITH listheader↑ DO
        next := NIL;
        lowerbound := DISKSIZE - CYLINDERSIZE;
        upperbound := DISKSIZE - 1;
        status := FREE;
    END; (* WITH *)

    (* create rest of list *)
    WHILE listheader↑.lowerbound > 0 DO

        (* create node for previous block *)
        WITH listheader↑ DO
            ALLOCATE( previous, SIZE(previous↑) );
            previous↑.next := listheader;
            listheader := previous;
        END; (* WITH *)

        (* initialize new block *)
        WITH listheader↑ DO
            lowerbound := next↑.lowerbound - CYLINDERSIZE;
            upperbound := next↑.upperbound - CYLINDERSIZE;
            status := FREE;
        END; (* WITH *)
    END; (* WHILE *)

    listheader↑.previous := NIL;
END InitDisk;
```

```
    (*  Take no action on a given diskread or diskwrite.      *)
    PROCEDURE Null(null: WORD);

    BEGIN
    END Null;


    MODULE DiskDriver;
(****************************************************************************)
(*****                                                                     *)
(***** FUNCTION:   This local module does the low level work of actually   *)
(*****             processing transfers of blocks of information between   *)
(*****             the disk and main memory.                               *)
(*****                                                                     *)
(***** POLICY:     Requests for transfers are inserted into a list, ordered *)
(*****             to minimize search time.  Once a request has been       *)
(*****             given to the device, control is returned to the oper-   *)
(*****             ating system until the transfer has completed, at       *)
(*****             which time  an interrupt handler 1) calls the cleanup   *)
(*****             procedure which was passed as a parameter, and          *)
(*****             2) starts acting on a new request.                      *)
(*****                                                                     *)
(****************************************************************************)
(**)
(**)        IMPORT
(**)            (* Constants *)
(**)            CYLINDERCOUNT, CYLINDERSIZE,
(**)
(**)            (* Types *)
(**)            ADDRESS, CylinderID, Diskbuffer, Diskoperation, Interruptcode,
(**)            LooseEnds, SectorID,
(**)
(**)            (* Variables *)
(**)            lastbuffer,
(**)
(**)            (* Modules *)
(**)            VirtualMachine,
(**)
(**)            (* Procedures *)
(**)            ADR, ALLOCATE, DEALLOCATE, (* VirtualMachine.DiskRead,
(**)            VirtualMachine.DiskWrite, *) ReturnFromInterrupt, SetInterruptHandler, SIZE;
(**)
(**)        EXPORT
(**)            (* Procedures *)
(**)            CopySector;
(**)
(**)
```

```
(**)    TYPE
(**)        Requestpointer =   POINTER TO Requestnode;
(**)        Requestnode    =   RECORD
(**)                               buffer:        (* pointer to a sector of the memory
(**)                                                  block involved in transfer *)
(**)                                   Diskbuffer;
(**)                               operation:     (* READ or WRITE *)
(**)                                   Diskoperation;
(**)                               sector:        (* sector to be transfered *)
(**)                                   SectorID;
(**)                               tidyup:        (* information to be acted upon
(**)                                                   when transfer is complete *)
(**)                                   LooseEnds;
(**)                               next:          (* next in list *)
(**)                                   Requestpointer;
(**)                           END;
(**)
(**)    VAR
(**)        currentcylinder,       (* cylinder on which requests are currently being served *)
(**)        i:                     (* loop index *)
(**)            CylinderID;
(**)        direction:             (* scanning direction of read-write head *)
(**)            (INWARD, OUTWARD);
(**)        listheader:            (* pointer to first node in request list *)
(**)            Requestpointer;
(**)        requestqueue:          (* queue of requests for each cylinder *)
(**)            ARRAY CylinderID OF
(**)            RECORD
(**)                head,
(**)                tail:          (* head and tail of queue *)
(**)                    Requestpointer;
(**)            END;
(**)
(**)
(**)    PROCEDURE CopySector(operation: Diskoperation; sector: SectorID;
(**)    buffer: Diskbuffer; tidyup: LooseEnds);
(**)    (*     Create a request node and insert it into the list of requests for        *)
(**)    (*     disk access.  If it is the only request in the list, start acting on     *)
(**)    (*     it (if there are other requests, action will automatically begin         *)
(**)    (*     on the next in line when the previous one is finished).                  *)
(**)    (*                                                                              *)
(**)    (*     PARAMETERS:     operation–READ or WRITE.                                 *)
(**)    (*                     sector–the disk sector involved in the                   *)
(**)    (*                         transfer.                                            *)
(**)    (*                     buffer–the page of memory involved in the                *)
(**)    (*                         transfer.                                            *)
(**)    (*                     tidyup–information to be acted upon when                 *)
(**)    (*                         the entire block has been copied.                    *)
(**)
```

```
(**)          VAR
(**)              request:     (* the request node *)
(**)                  Requestpointer;
(**)
(**)      BEGIN
(**)          ALLOCATE( request, SIZE(request↑) );
(**)          request↑.operation := operation;
(**)          request↑.sector := sector;
(**)          request↑.buffer := buffer;
(**)          request↑.tidyup := tidyup;
(**)
(**)          Insert(request);
(**)
(**)          IF listheader = request THEN
(**)              DiskStart(listheader↑.sector, listheader↑.buffer);
(**)          END; (* IF *)
(**)
(**)      END CopySector;
(**)
(**)
(**)      PROCEDURE DiskStart(sector: SectorID; buffer: ADDRESS);
(**)      (*      Start the disk on its next transfer.                            *)
(**)      (*                                                                      *)
(**)      (*      PARAMETERS:     sector–the disk sector involved in the          *)
(**)      (*                          transfer.                                   *)
(**)      (*                      buffer–the starting of address of memory        *)
(**)      (*                          involved in the transfer.                   *)
(**)
(**)      BEGIN
(**)          WITH listheader↑ DO
(**)              IF operation = READ THEN
(**)                  VirtualMachine.DiskRead( CARDINAL(sector), buffer);
(**)              ELSE
(**)                  VirtualMachine.DiskWrite( buffer, CARDINAL(sector) );
(**)              END; (* IF *)
(**)          END; (* WITH *)
(**)      END DiskStart;
(**)
(**)
```

```
(**)       PROCEDURE Insert(request: Requestpointer);
(**)       (*    Place a request into the list. Arrange the requests to facilitate     *)
(**)       (*    a SCAN, N-SCAN, or C-SCAN search strategy. (For each                  *)
(**)       (*    cylinder on the disk a queue is maintained. The exact choice of       *)
(**)       (*    when and how to link the queues determines which disk-scheduling      *)
(**)       (*    discipline will be followed.)                                         *)
(**)       (*                                                                          *)
(**)       (*    PARAMETERS:     request--the information to be put into the           *)
(**)       (*                        list of requests.                                 *)
(**)
(**)          VAR
(**)             requestcylinder:    (* cylinder of the new request *)
(**)                CylinderID;
(**)
(**)
(**)       BEGIN
(**)
(**)          requestcylinder := request↑.sector DIV CYLINDERSIZE;
(**)
(**)          WITH requestqueue[requestcylinder] DO
(**)             IF tail = NIL THEN
(**)             (* queue is empty *)
(**)
(**)                (* place request at beginning *)
(**)                request↑.next := NIL;
(**)                head := request;
(**)                tail := request;
(**)
(**)                IF listheader = NIL THEN
(**)                (* this is the only request in entire list *)
(**)
(**)                   listheader := request;
(**)                   currentcylinder := requestcylinder;
(**)                END (* IF listheader *);
(**)
(**)             ELSE
(**)                request↑.next := tail↑.next;
(**)                tail↑.next := request;
(**)             END; (* IF tail *)
(**)
(**)             tail := request;
(**)          END; (* WITH *)
(**)
(**)       END Insert;
(**)
(**)
```

```
(**)        PROCEDURE NewScan;
(**)        (*      This procedure links the request queues according to the SCAN       *)
(**)        (*      disk-scheduling discipline when a scan is complete.                 *)
(**)
(**)            VAR
(**)                i,              (* loop index *)
(**)                first,          (* first cylinder of next pass *)
(**)                last,           (* last cylinder of next pass *)
(**)                previous:       (* the last cylinder looked at previously *)
(**)                    CylinderID;
(**)                step:           (* 1 if direction is inward, -1 if outward *)
(**)                    [-1 .. 1];
(**)
(**)        BEGIN
(**)
(**)            (* change directions *)
(**)            IF direction = OUTWARD THEN
(**)                direction := INWARD;
(**)                first := 0;
(**)                last := CYLINDERCOUNT - 1;
(**)                step := 1;
(**)            ELSE
(**)                direction := OUTWARD;
(**)                first := CYLINDERCOUNT - 1;
(**)                last := 0;
(**)                step := -1;
(**)            END;  (* IF *)
(**)
(**)            (* relink queues *)
(**)            i := last;
(**)            WHILE i # first DO
(**)                DEC(i, step);
(**)                previous := CylinderID( INTEGER(i) + step );
(**)                WITH requestqueue[i] DO
(**)                    IF head = NIL THEN
(**)                        head := requestqueue[previous].head;
(**)                    ELSE
(**)                        tail↑.next := requestqueue[previous].head;
(**)                    END; (* IF *)
(**)                END; (* WITH *)
(**)                requestqueue[previous].head := NIL;
(**)            END; (* WHILE *)
(**)
(**)            listheader := requestqueue[first].head;
(**)            requestqueue[first].head := NIL;
(**)
(**)        END NewScan;
(**)
(**)
```

```
(**)        PROCEDURE TransferComplete;
(**)        (*      Handle disk interrupt when a transfer is complete. (Currently        *)
(**)        (*      assumes SCAN disk-scheduling.)                                      *)
(**)
(**)            VAR
(**)                lastrequest:        (* the request just completed *)
(**)                    Requestpointer;
(**)                ptr:                (* initialized to address of ''lastbuffer'' *)
(**)                    ADDRESS;
(**)
(**)        BEGIN
(**)
(**)            (* prepare to start next request *)
(**)            lastrequest := listheader;
(**)            listheader := listheader↑.next;
(**)            IF listheader = NIL THEN
(**)
(**)                (* relink cylinder queues into a new list *)
(**)                NewScan;
(**)            ELSE
(**)                IF lastrequest = requestqueue[currentcylinder].tail THEN
(**)                (* this was last request on cylinder *)
(**)
(**)                    requestqueue[currentcylinder].tail := NIL;
(**)                END; (* IF lastrequest *)
(**)
(**)                (* move to (or stay at) first cylinder that has requests *)
(**)                WHILE requestqueue[currentcylinder].tail = NIL DO
(**)                    IF direction = INWARD THEN
(**)                        INC(currentcylinder);
(**)                    ELSE
(**)                        DEC(currentcylinder);
(**)                    END; (* IF direction *)
(**)                END; (* WHILE *)
(**)            END; (* IF listheader = NIL *)
(**)
(**)            (* start next request *)
(**)            IF listheader # NIL THEN
(**)                DiskStart(listheader↑.sector, listheader↑.buffer);
(**)            END; (* IF listheader # NIL *)
(**)
```

```
(**)            (* tidy up *)
(**)            WITH lastrequest↑.tidyup DO
(**)                IF transferfinished THEN
(**)                    IF lastrequest↑.operation = READ THEN
(**)                        ptr := ADR(lastbuffer);
(**)                        FOR i := 0 TO count - 1 DO
(**)                            buffer↑ := ptr↑;
(**)                            INC(buffer);
(**)                            INC(ptr);
(**)                        END; (* FOR *)
(**)                    END; (* IF READ *)
(**)                    notify(parameter);
(**)                END; (* IF transferfinished *)
(**)            END; (* WITH lastrequest *)
(**)
(**)            DEALLOCATE( lastrequest, SIZE(lastrequest↑) );
(**)            ReturnFromInterrupt(DISK);
(**)        END TransferComplete;
(**)
(**)
(**)    BEGIN(* local module initialization *)
(**)
(**)        SetInterruptHandler(DISK, TransferComplete);
(**)
(**)        (* set up list of requests–list is initially empty *)
(**)        listheader := NIL;
(**)        FOR i := 0 TO CYLINDERCOUNT - 1 DO
(**)            requestqueue[i].head := NIL;
(**)            requestqueue[i].tail := NIL;
(**)        END; (* FOR *)
(**)        direction := INWARD;
(**)
(**)    END DiskDriver;
(*************************************************************************)

BEGIN (* module initialization *)

    (* see procedure InitDisk *)

END DiskManager.
```

```
DEFINITION MODULE HighLevelScheduler;

(**********************************************************************)
(*                                                                    *)
(*  FUNCTION:   The High Level Scheduler is responsible for initiating and terminat-  *)
(*              ing jobs. When it chooses to accept a job into the system, it  *)
(*              creates a new process for that job, and loads the card images  *)
(*              associated with the job.                              *)
(*                                                                    *)
(*  AUTHOR:     Rick Fisher                                           *)
(*                                                                    *)
(**********************************************************************)

    FROM ProcessManager IMPORT
        (* Types *)
        ProcessID;

    EXPORT QUALIFIED
        (* Procedures *)
        InitHLSched, ShoulderTap, Terminate;


    PROCEDURE InitHLSched;
    (*      Order-dependent module initialization. This procedure consists of initial-  *)
    (*      ization statements which should not be executed until after the initialization  *)
    (*      for the main module has begun.                                *)


    PROCEDURE ShoulderTap;
    (*      Check to see if a new process can be loaded; if so, load it.  *)


    PROCEDURE Terminate(process: ProcessID);
    (*      Deallocate the resources of a completed process and attempt to load a  *)
    (*      new one.                                                      *)
    (*                                                                    *)
    (*      PARAMETERS:     process–the process being terminated.        *)

END HighLevelScheduler.
```

```
IMPLEMENTATION MODULE HighLevelScheduler;

(***********************************************************************)
(*                                                                     *)
(*  POLICY:     The High Level Scheduler does no checking of the M-code or *)
(*              program input it gets from the Spooler; it simply loads each *)
(*              page verbatim.                                         *)
(*                                                                     *)
(*  AUTHOR:     Rick Fisher                                            *)
(*                                                                     *)
(***********************************************************************)

   FROM SYSTEM IMPORT
      (* Types *)
      ADDRESS, WORD,

      (* Procedures *)
      SIZE;

   IMPORT MemoryManager;
   FROM MemoryManager IMPORT
      (* Constants *)
      PAGESIZE,

      (* Types *)
      MemoryblockID, Pageptr,

      (* Procedures *)
      (* Allocate, *) BlockID, (* Deallocate, *) StartingAddress;


   IMPORT DiskManager;
   FROM DiskManager IMPORT
      (* Constants *)
      NULL,

      (* Types *)
      DiskblockID,

      (* Procedures *)
      (* Deallocate, *) DiskWrite;

   FROM ProcessManager IMPORT
      (* Types *)
      ProcessID, Disklist,

      (* Procedures *)
      CreateID, DiskUse, Equal, Initialize, MemoryLocation, NullProcess, PermanentLocation,
      Resident;
```

```
FROM Spooler IMPORT
    (* Procedures *)
    DeSpool, PopJobSize;

IMPORT MediumScheduler;
FROM MediumScheduler IMPORT
    (* Procedures *)
    Schedule (*, Terminate *);

FROM Loader IMPORT
    (* Procedures *)
    Load;

FROM STORAGE IMPORT
    (* Procedures *)
    ALLOCATE, DEALLOCATE;


VAR
    codeEnd,            (* last node in code output list *)
    codestart:          (* header node for code output list *)
        Disklist;
    waitingOndisk:      (* TRUE if ''ShoulderTap'' is waiting on disk completion *)
        BOOLEAN;


(*  Module initialization.   *)
PROCEDURE InitHLSched;
BEGIN
    ALLOCATE( codestart, SIZE(codestart↑) );
    codestart↑.next := NIL;
    codeEnd := codestart;
END InitHLSched;



PROCEDURE Resume(null: WORD);
(*      Record completion of disk operation and restart ''ShoulderTap''.          *)
(*                                                                                *)
(*      PARAMETERS:    null–an unused parameter, can be any word.                 *)

BEGIN
    waitingOndisk := FALSE;
    ShoulderTap;
END Resume;


CONST
    PRIORITY    =  1;  (* default priority is higher than the null process *)
    STACKSIZE   =  8;  (* number of pages in default stack *)
```

```
VAR
    codesize,                 (* number of pages in current job's code *)
    i:                        (* loop index *)
        CARDINAL;
    inprogress:               (* TRUE if part of a job has been processed *)
        BOOLEAN;
    inputsize:                (* number of pages in current job's input file *)
        CARDINAL;
    newblock:                 (* the memoryblock to hold the new job *)
        MemoryblockID;
    pageaddress:              (* where the job's next page should be written *)
        ADDRESS;
    pagecount:                (* the number of pages spooled for the job *)
        CARDINAL;
    waitingforprocessID:      (* TRUE if job is loaded, but no ID is assigned *)
        BOOLEAN;


(*  Try to load a new job.   *)
PROCEDURE ShoulderTap;

    VAR
        blockstart:     (* starting address of memory block *)
            ADDRESS;
        newprocess:     (* new process ID *)
            ProcessID;
        page:           (* pointer to the most recently despooled page *)
            Pageptr;

BEGIN

    (* attempt to load only if a previous incaration of ShoulderTap is not waiting
        on a disk completion *)
    IF NOT waitingOndisk THEN

        (* try to load a job *)
        IF NOT waitingforprocessID THEN
        (* current job is not completely loaded *)

            IF NOT inprogress THEN
            (* there is no current job *)

                (* get size of new job *)
                PopJobSize(codesize, inputsize);
                pagecount := codesize + inputsize;
                IF pagecount > 0 THEN
                    inprogress := TRUE;
                END; (* IF pagecount *)
            END; (* IF NOT inprogress *)
            (* try to find memoryblock for a new job *)
            IF ( ADDRESS(newblock) = NIL ) & inprogress THEN

                newblock := MemoryManager.Allocate( (pagecount + STACKSIZE)
                            * PAGESIZE );
                blockstart := StartingAddress(newblock);
                pageaddress := blockstart;
                i := 0;
            END; (* IF newblock *)

            (* load as many pages of the job as possible *)
            IF ADDRESS(newblock) # NIL THEN
                LOOP
                    IF i = pagecount THEN
                    (* job is completely loaded *)

                        inprogress := FALSE;
                        waitingforprocessID := TRUE;
                        EXIT; (* LOOP *)
                    ELSE
                        page := DeSpool();
                    END; (* IF i *)
                    IF page = Pageptr(NIL) THEN
                        EXIT; (* LOOP *)
                    END; (* IF page *)

                    (* write M-code to program's output *)
                    IF i < codesize THEN
                        ALLOCATE( codeEnd↑.next, SIZE(codeEnd↑) );
                        codeEnd := codeEnd↑.next;
                        codeEnd↑.next := NIL;
                        codeEnd↑.diskblockptr := DiskManager.Allocate(PAGESIZE);
                        DiskWrite( BlockID( ADDRESS(page) ), codeEnd↑.diskblockptr,
                            Resume, NULL );
                        waitingOndisk := TRUE;
                    END; (* IF i < codesize *)

                    (* load the page *)
                    Load(page, pageaddress);
                    INC(i);
                    IF i <= codesize THEN
                        EXIT; (* LOOP *)
                    END; (* IF i <= codesize *)
                END; (* LOOP *)
            END; (* IF ADDRESS *)
        END; (* IF NOT waitingforprocessID *)
        (* make completely loaded job into a process *)
        IF waitingforprocessID THEN
            newprocess := CreateID();
            IF NOT Equal( newprocess, NullProcess() ) THEN
                waitingforprocessID := FALSE;
                Initialize(newprocess, newblock, blockstart + codesize * PAGESIZE,
                    blockstart + pagecount * PAGESIZE, STACKSIZE * PAGESIZE,
                    PRIORITY, codestart↑.next, codeEnd);
                codestart↑.next := NIL;
                codeEnd := codestart;
                newblock := MemoryblockID(NIL);
                Schedule(newprocess);
            END; (* IF NOT Equal *)
        END; (* IF waitingforprocessID *)
    END; (* IF NOT waitingOndisk *)
END ShoulderTap;


PROCEDURE Terminate(process: ProcessID);
(*      Deallocate the resources of a completed process and attempt to load       *)
(*      a new process.                                                             *)
(*                                                                                 *)
(*      PARAMETERS:     process–the process being terminated.                     *)

    VAR
        disklist:               (* list of diskblocks used by process *)
            Disklist;
        permanentlocation:      (* disk location of process *)
            DiskblockID;
        tempdisknode:           (* used in deallocating list nodes *)
            Disklist;
        tempdiskblock:          (* used in deallocating a disk block *)
            DiskblockID;
        tempmemblock:           (* used in deallocating a memory block *)
            MemoryblockID;

BEGIN

    (* free memory space *)
    IF Resident(process) THEN
        tempmemblock := MemoryLocation(process);
        MemoryManager.Deallocate(tempmemblock);
    END; (* IF Resident *)
```

```
        (* free disk space *)
        permanentlocation := PermanentLocation(process);
        IF ADDRESS(permanentlocation) # NIL THEN
            tempdiskblock := DiskblockID(permanentlocation);
            DiskManager.Deallocate(tempdiskblock);
        END; (* IF permanentlocation *)
        disklist := DiskUse(process);

        WHILE disklist # Disklist(NIL) DO
            tempdisknode := disklist;
            disklist := disklist↑.next;
            tempdiskblock := DiskblockID(tempdisknode↑.diskblockptr);
            DiskManager.Deallocate(tempdiskblock);
            DEALLOCATE( tempdisknode, SIZE(tempdisknode↑) );
        END; (* WHILE *)

        MediumScheduler.Terminate(process);
        ShoulderTap;
    END Terminate;

BEGIN (* module initialization *)
    inprogress := FALSE;
    pageaddress := NIL;
    newblock := MemoryblockID(NIL);
    pagecount := 0;
    waitingforprocessID := FALSE;
    waitingOndisk := FALSE;

    (* see also procedure InitHLSched *)

END HighLevelScheduler.
```

```
DEFINITION MODULE Loader;

(***********************************************************************)
(*                                                                     *)
(*  FUNCTION:   The Loader takes cards on which M-code instructions are punched,  *)
(*              translates the characters into octal numbers, and loads those     *)
(*              numbers into memory.                                               *)
(*                                                                     *)
(*  AUTHOR:     Rick Fisher                                            *)
(*                                                                     *)
(***********************************************************************)

   FROM SYSTEM IMPORT
      (* Types *)
      ADDRESS;

   FROM MemoryManager IMPORT
      (* Types *)
      Pageptr;

   EXPORT QUALIFIED
      (* Procedures *)
      Load;


   PROCEDURE Load(page: Pageptr; VAR address: ADDRESS);
   (*      Convert the characters on the given page to octal numbers, and load them     *)
   (*      at the given address, updating the address when finished to be the address   *)
   (*      at which the next page should be loaded.                                      *)
   (*                                                                                    *)
   (*      PARAMETERS:     page–the page to be loaded.                                   *)
   (*                      address–the address at which loading is to begin.             *)

END Loader.
```

```
IMPLEMENTATION MODULE Loader;

(*************************************************************************)
(*                                                                       *)
(*  POLICY:      The Loader assumes that cards are correctly spooled, with code  *)
(*               followed by input followed by code followed by input, etc. No   *)
(*               error checking is undertaken.                           *)
(*                                                                       *)
(*  AUTHOR:      Rick Fisher                                             *)
(*                                                                       *)
(*************************************************************************)

    FROM SYSTEM IMPORT
        (* Types *)
        ADDRESS, WORD;

    FROM VirtualMachine IMPORT
        (* Constants *)
        BYTESPERPAGE;

    FROM MemoryManager IMPORT
        (* Constants *)
        PAGESIZE,

        (* Types *)
        Pageindex, Pageptr;

    FROM Spooler IMPORT
        (* Constants *)
        ENDofINPUT, ENDofJOB;


    CONST
        BLANK       =   ' ';
        NEWLINE     =   12C;

    VAR
        code:               (* TRUE if current page is code, FALSE if page is input *)
            BOOLEAN;
        currentvalue:       (* current octal value of characters read so far *)
            CARDINAL;


    (*  Load a page of card images into memory.    *)
    PROCEDURE Load(page: Pageptr; VAR address: ADDRESS);

        VAR
            index:          (* index into the page *)
                Pageindex;
            inputpage:      (* new location for a page of input *)
                Pageptr;
```

```
BEGIN

    (* load input, or translate and load code *)
    IF NOT code THEN
    (* page is input data *)

        inputpage := Pageptr(address);

        (* transfer each character and check for end of input *)
        FOR index := 0 TO BYTESPERPAGE - 1 DO
            inputpage↑[index] := page↑[index];
            IF inputpage↑[index] = ENDofINPUT THEN
                code := TRUE;
            END; (* IF inputpage *)
        END; (* FOR *)

        INC(address, PAGESIZE);
    ELSE
    (* page is code *)

        index := 0;

        (* process each character on the page *)
        LOOP

            (* process all characters in a ''word'' (delimited by white space) *)
            WHILE (page↑[index] # BLANK) & (page↑[index] # NEWLINE) DO
                IF (page↑[index] = ENDofJOB) THEN

                    (* move address to start of next page, unless it is already at the start of a page *)
                    INC(address, PAGESIZE - 1 - (address - 1) MOD PAGESIZE);

                    code := FALSE;
                    EXIT; (* LOOP *)
                ELSE
                    currentvalue := ORD(page↑[index]) - ORD('0') + currentvalue * 10B;
                    INC(index);
                    IF index = PAGESIZE THEN
                        EXIT; (* LOOP *)
                    END; (* IF index *)
                END; (* IF page *)
            END; (* WHILE *)
```

```
                address↑ := WORD(currentvalue);
                currentvalue := 0;
                INC(address);
                INC(index);
                IF index = PAGESIZE THEN
                    EXIT; (* LOOP *)
                END; (* IF index *)
            END; (* LOOP *)
        END; (* IF code *)
    END Load;

BEGIN (* module initialization *)
    code := TRUE;
    currentvalue := 0;
END Loader.
```

```
DEFINITION MODULE LocalSystem;

(************************************************************************)
(*                                                                      *)
(*  FUNCTION:   This is a pseudo-module containing descriptions of machine *)
(*              dependent features.  Because the objects imported from  *)
(*              LocalSystem obey special rules and are implemented directly *)
(*              in M-Code, the module must be known to the linker or compiler, *)
(*              and a definition module is necessary merely for documentation *)
(*              and to allow importing modules to compile properly.  Modules *)
(*              making direct use of variables or procedures exported from *)
(*              LocalSystem are considered low-level, system-dependent modules, *)
(*              and are therefore non-portable.                         *)
(*                                                                      *)
(*  AUTHOR:     Rick Fisher                                             *)
(*                                                                      *)
(************************************************************************)

  FROM SYSTEM IMPORT
    (* Types *)
    ADDRESS, WORD;

  EXPORT QUALIFIED
    (* Constants *)
    BADINSTRUCTION, BOUNDSVIOLATION, CARDREADER, CHECKSCHEDULE,
    CLOCK, DISK, Halt, HIGHMCODEINSTRUCTION, INITIALIZE,
    LINEPRINTER, LOWINTERRUPT, MODEVIOLATION, OUTofRANGE,
    SHOULDERTAP, STACKOVERFLOW, SVC, TRAP, UNDEFINEDINSTRUCTION,
    VALUERANGE, WORDSINMEMORY,

    (* Variables *)
    bootcontext, inputbuffer, interruptvector, OScontext, outputbuffer,

    (* Types *)
    Context, Dataframeptr, Segmenttableptr,

    (* Procedures *)
    ContextSwitch, DiskRead, DiskWrite, Read, Write, Trap;


  CONST
    HIGHMCODEINSTRUCTION =  377B;
    MAXMODULES           =  98; (* number of words in initial segment *)
    WORDSINMEMORY        =  40000B;
    (* Interrupt Codes *)
    TRAP                    =    7;
    CARDREADER              =    8;
    LINEPRINTER             =    9;
    DISK                    =   10;
    CLOCK                   =   11;
    LOWINTERRUPT            =  TRAP;

    (* Trap Codes *)
    STACKOVERFLOW           =    3;
    VALUERANGE              =    4;
    BOUNDSVIOLATION         =    7;
    MODEVIOLATION           =    8;
    UNDEFINEDINSTRUCTION    =    9;
    Halt                    =   10;
    BADINSTRUCTION          =   12;
    CHECKSCHEDULE           =   13;
    INITIALIZE              =   16;
    OUTofRANGE              =   17;
    SHOULDERTAP             =   18;
    SVC                     =   19;

TYPE
    Dataframeptr      =  POINTER TO Dataframe;
    Dataframe         =  RECORD
                             codeframe:          (* code base address *)
                                 ADDRESS;
                             initializationflag: (* indicates if module is initialized *)
                                 INTEGER;
                             stringpointer:      (* address of the string table *)
                                 ADDRESS;
                             globals:            (* a variable-sized list of words *)
                                 ARRAY [1..1] OF WORD;
                         END;
    Segmenttableptr   =  POINTER TO Segmenttable;
    Segmenttable      =  ARRAY [0 .. MAXMODULES] OF ADDRESS;
    Context           =  RECORD
                             dataframe:          (* data base address *)
                                 Dataframeptr;
                             currentactivation:  (* base address of current activation
                                                    record *)
                                 ADDRESS;
                             PC:                 (* program counter *)
                                 ADDRESS;
                             interruptmask:      (* process interrupt mask *)
                                 BITSET;
                             stacktop:           (* pointer to top of stack *)
                                 ADDRESS;
```

```
                                stacklimit:       (* stack limit address *)
                                    ADDRESS;
                                trap:             (* trap responsible for an interrupt *)
                                    CARDINAL;
                                errortrapmask:    (* not used in current machine *)
                                    BITSET;
                                segmenttable:     (* segment table address *)
                                    Segmenttableptr;
                                upperbound:       (* high address available to process *)
                                    ADDRESS;
                            END;
    Vectoredinterrupt   =   RECORD
                                formercontext:    (* pointer to interrupted context *)
                                    ADDRESS;
                                interrupthandler: (* interrupt routine *)
                                    PROC;
                            END;

VAR
    (* predefined locations *)
    bootcontext[4],            (* address of boot context *)
    OScontext[5]:              (* address of operating system context *)
        ADDRESS;
    inputbuffer[10],           (* cardreader's one-byte register *)
    outputbuffer[11]:          (* lineprinter's one-byte register *)
        CHAR;
    interruptvector[16]:       (* set of interrupt vectors *)
        ARRAY [TRAP .. CLOCK] OF Vectoredinterrupt;


PROCEDURE ContextSwitch(context: ADDRESS);
(*      Return to a previously stored context.                                 *)
(*                                                                             *)
(*      PARAMETERS:     context–a pointer to the previously stored context.    *)


PROCEDURE DiskRead(disksector: CARDINAL; memoryaddress: ADDRESS);
(*      Start the disk and return. The disk will run concurrently with the     *)
(*      CPU and transfer the sector starting at the given address to the given *)
(*      memory location. When finished, the disk will cause an interrupt.      *)
(*                                                                             *)
(*      PARAMETERS:     diskaddress–the sector of the disk from which data     *)
(*                          is read.                                           *)
(*                      memoryaddress–the starting address of memory into      *)
(*                          which data is written.                             *)
```

```
PROCEDURE DiskWrite(memoryaddress: ADDRESS; diskaddress: CARDINAL);
(*      Start the disk and return.  The disk will run concurrently with the          *)
(*      CPU and transfer the memory block starting at the given address to           *)
(*      the given disk sector.  When finished, the disk will cause an interrupt.     *)
(*                                                                                   *)
(*      PARAMETERS:     memoryaddress–the starting address of memory                 *)
(*                          from which data is read.                                 *)
(*                      diskaddress–the sector of the disk to which data             *)
(*                          is written.                                              *)


PROCEDURE Read;
(*      Start the card reader and return.  The card reader will run concurrently     *)
(*      with the CPU and deposit the next byte read into the reserved memory         *)
(*      location "inputbuffer".  When finished, the cardreader will cause an         *)
(*      interrupt.                                                                   *)


PROCEDURE Trap(reason: CARDINAL);
(*      Store the current context and load the context of the operating system,      *)
(*      then call the trap handling routine.                                         *)
(*                                                                                   *)
(*      PARAMETERS:     reason–the cause of the trap.  There are three types of      *)
(*                          traps:                                                   *)
(*      Machine-raised error traps:                                                  *)
(*          BOUNDSVIOLATION:            User attempted to access memory              *)
(*                                          location outside the address in          *)
(*                                          the bounds registers.                    *)
(*          MODEVIOLATION:              User attempted to perform super-             *)
(*                                          visor instruction.                       *)
(*          OUTofRANGE:                 Inaccessible memory location.                *)
(*          STACKOVERFLOW:              No room on stack for activation              *)
(*                                          records or dynamic variables.            *)
(*          VALUERANGE:                 Case index, FOR loop index, or               *)
(*                                          array index out of range.                *)
(*                                                                                   *)
(*      Operating system-raised error traps:                                         *)
(*          BADINSTRUCTION:             Illegal characters in instruction.           *)
(*          UNDEFINEDINSTRUCTION:       No such M-Code instruction                   *)
(*                                          defined.                                 *)
(*                                                                                   *)
(*      Other traps:                                                                 *)
(*          CHECKSCHEDULE:              Time to see if any scheduled events          *)
(*                                          should be executed.                      *)
(*          Halt:                       User's program is self-aborting.             *)
(*          INITIALIZE:                 Values must be set in OS context.            *)
(*          SHOULDERTAP:                Time to see if any jobs can be               *)
(*                                          brought into the system and              *)
(*                                          made into processes.                     *)
(*          SVC:                        Supervisor call.                             *)
```

```
    PROCEDURE Write;
    (*    Start the line printer and return. The line printer will run concurrently    *)
    (*    with the CPU and print the byte currently in the reserved memory             *)
    (*    location ''outputbuffer''. When finished, the line printer will cause an     *)
    (*    interrupt.                                                                    *)

END LocalSystem.
```

```
IMPLEMENTATION MODULE LocalSystem;

(*****************************************************************************)
(*                                                                           *)
(*  POLICY:      None.  At present this module is known to the linker, not the   *)
(*               compiler, so definition and implementation modules must both    *)
(*               exist for the purposes of compilation.                          *)
(*                                                                           *)
(*  AUTHOR:      Rick Fisher                                                 *)
(*                                                                           *)
(*****************************************************************************)

   FROM SYSTEM IMPORT
      (* Types *)
      ADDRESS;

   PROCEDURE ContextSwitch(to: ADDRESS);
   BEGIN
      (* CNTX = 246 *)
   END ContextSwitch;

   PROCEDURE DiskRead(disksector: CARDINAL; memoryaddress: ADDRESS);
   BEGIN
      (* DSKR = 242 *)
   END DiskRead;

   PROCEDURE DiskWrite(memoryaddress: ADDRESS; diskaddress: CARDINAL);
   BEGIN
      (* DSKW = 243 *)
   END DiskWrite;

   PROCEDURE Read;
   BEGIN
      (* READ = 240 *)
   END Read;

   PROCEDURE Trap(reason: CARDINAL);
   BEGIN
      (* TRAP = 304 *)
   END Trap;

   PROCEDURE Write;
   BEGIN
      (* WRITE = 241 *)
   END Write;

BEGIN
END LocalSystem.
```

```
DEFINITION MODULE LowLevelScheduler;

(***********************************************************************)
(*                                                                     *)
(*  FUNCTION:   The low level scheduler handles the details of suspension and   *)
(*              termination of processes, and serves as a scheduler for an      *)
(*              individual processor. The ready processes are kept in a priority *)
(*              queue, and given access to the CPU in turn.                     *)
(*                                                                     *)
(*  AUTHOR:     Rick Fisher                                            *)
(*                                                                     *)
(***********************************************************************)


    FROM VirtualMachine IMPORT
        (* Types *)
        ContextID;

    FROM ProcessManager IMPORT
        (* Types *)
        ProcessID;

    EXPORT QUALIFIED
        (* Procedures *)
        AddToReadyQueue, Block, CurrentContext, CurrentProcess, CPULoad, InitLLSched,
        Sleep, Terminate, TimeOut;


    PROCEDURE AddToReadyQueue(process: ProcessID);
    (*      Add the stated process to the ready queue.                           *)
    (*                                                                           *)
    (*      PARAMETERS:     process--the process to be added to the ready queue. *)


    PROCEDURE Block;
    (*      Block the current process and choose a new process for running.     *)


    PROCEDURE CurrentContext(): ContextID;
    (*      RETURNS the context ID of the current process.                      *)


    PROCEDURE CurrentProcess(): ProcessID;
    (*      RETURNS the process ID of the current process.                      *)


    PROCEDURE CPULoad(): CARDINAL;
    (*      RETURNS the number of entries in the ready queue.                   *)
    PROCEDURE InitLLSched;
    (*      Order-dependent module initialization. This procedure consists       *)
    (*      of initialization statements which should not be executed until      *)
    (*      after the the module ProcessManager has been initialized.            *)


    PROCEDURE Sleep(time: CARDINAL);
    (*      Put the current process to sleep for a specified length of time.     *)
    (*                                                                           *)
    (*      PARAMETERS:     time–the number of ticks process should sleep.       *)


    PROCEDURE Terminate(VAR process: ProcessID);
    (*      Terminate a process. Should only be called by MediumScheduler.Terminate. *)
    (*                                                                           *)
    (*      PARAMETERS:     process–the process being terminated.                *)


    PROCEDURE TimeOut;
    (*      Called as a scheduled event, this preempts the current process and   *)
    (*      chooses a new process for running.                                   *)

END LowLevelScheduler.
```

```
IMPLEMENTATION MODULE LowLevelScheduler;

(************************************************************************)
(*                                                                      *)
(*  POLICY:      The low level scheduler makes no decisions regarding the length     *)
(*               of a quantum, or whether priorities should be assigned to           *)
(*               procedures. If priorities are assigned, however, all processes      *)
(*               of the highest priority will be completely serviced round-robin     *)
(*               before any other processes are given the CPU; and a new process     *)
(*               of higher priority than the current process will preempt the        *)
(*               current process.                                                    *)
(*                                                                      *)
(*  AUTHOR:      Rick Fisher                                            *)
(*                                                                      *)
(************************************************************************)


  FROM VirtualMachine IMPORT
    (* Types *)
    ContextID,

    (* Procedures *)
    SwitchContext;

  FROM ProcessManager IMPORT
    (* Types *)
    Actiontype, Priorityqueue, Prioritytype, ProcessID, Statustype,

    (* Procedures *)
    ChangeStatus, Context, Empty, Equal, Insert, InitPriorityQueue, Next, NullProcess,
    Peek, Priority, Remove, Schedule, Status, UnSchedule;


  VAR
    currentprocess:     (* the current process *)
      ProcessID;
    readylist:          (* queue of highest priority processes *)
      Priorityqueue;
    readylistsize:      (* number of entries in the readylist, including the current process *)
      CARDINAL;


  (*  Add the stated process to the ready queue.   *)
  PROCEDURE AddToReadyQueue(process: ProcessID);

    VAR
      next:             (* next process in readylist *)
        ProcessID;
```

```
BEGIN
    ChangeStatus(process, READY);
    IF Priority(process) > Priority(currentprocess) THEN

        (* preempt the running process *)
        Preempt(PENDING);

        (* ensure no rescheduled processes are returned to the readylist *)
        currentprocess := process;

        (* reschedule all other ready processes *)
        WHILE readylistsize > 0 DO
            next := Next(readylist);
            ChangeStatus(next, PENDING);
            Schedule(0, RESCHEDULE, next);
            DEC(readylistsize);
        END; (* WHILE *)

        Insert(process, readylist);
        INC(readylistsize);
        Dispatch;

    ELSIF Priority(process) = Priority(currentprocess) THEN
        Insert(process, readylist);
        INC(readylistsize);
    END; (* IF *)
END AddToReadyQueue;


(*  Block the current process.   *)
PROCEDURE Block;

BEGIN
    Preempt(BLOCKED);
    Dispatch;
END Block;


(*  Return the context ID of the current process.      *)
PROCEDURE CurrentContext(): ContextID;

BEGIN
    RETURN( Context(currentprocess) );
END CurrentContext;
```

```
(*  Return the process ID of the current process.      *)
PROCEDURE CurrentProcess(): ProcessID;

BEGIN
    RETURN(currentprocess);
END CurrentProcess;


(*  Return the number of entries in the queue.   *)
PROCEDURE CPULoad(): CARDINAL;

BEGIN
    RETURN(readylistsize);
END CPULoad;


(*  Select the next process to control the CPU.  *)
PROCEDURE Dispatch;

    VAR
        currentpriority:      (* priority of current process *)
            Prioritytype;
        temp:                 (* temporary variable *)
            ProcessID;

BEGIN

    (* find the first ready process *)
    currentprocess := Next(readylist);
    WHILE (readylistsize > 0) & ( Status(currentprocess) # READY ) DO
        DEC(readylistsize);
        currentprocess := Next(readylist);
    END; (* WHILE *)

    ChangeStatus(currentprocess, CURRENT);
    SwitchContext( CurrentContext() );
END Dispatch;


(*  Module initialization.   *)
PROCEDURE InitLLSched;

BEGIN
    InitPriorityQueue(readylist);
END InitLLSched;
```

```
PROCEDURE Preempt(newstatus: Statustype);
(*    Preempt the current process.                                              *)
(*                                                                              *)
(*    PARAMETERS:    newstatus–the status the process should have after         *)
(*                   being preempted.                                           *)

BEGIN
    IF Equal( NullProcess(), currentprocess ) THEN
        ChangeStatus(currentprocess, READY);
    ELSE
        ChangeStatus(currentprocess, newstatus);
        CASE newstatus OF
            BLOCKED, PENDING, SLEEPING:
                DEC(readylistsize);
                Schedule(0, RESCHEDULE, currentprocess);
        |   READY:
                Insert(currentprocess, readylist);
        ELSE END; (* CASE *)
    END; (* IF *)
END Preempt;


(* Put the current process to sleep.    *)
PROCEDURE Sleep(time: CARDINAL);

BEGIN
    Preempt(SLEEPING);
    Schedule(time, WAKEUP, currentprocess);
    Dispatch;
END Sleep;


(* Terminate a process.    *)
PROCEDURE Terminate(VAR process: ProcessID);

BEGIN
    UnSchedule(process);
    IF Status(process) = READY THEN
        Remove(process, readylist);
        DEC(readylistsize);
    ELSIF Equal(process, currentprocess) THEN
        DEC(readylistsize);
        Dispatch;
    END; (* IF *)
    ChangeStatus(process, TERMINATED);
END Terminate;
```

```
    (*  Preempt the current process and choose a new process to run.    *)
    PROCEDURE TimeOut;

    BEGIN
        Preempt(READY);
        Dispatch;
    END TimeOut;


BEGIN (* module initialization *)

    currentprocess := NullProcess();
    readylistsize := 0;

    (* see also procedure InitLLSched *)

END LowLevelScheduler.
```

```
DEFINITION MODULE MediumScheduler;

(*************************************************************************)
(*                                                                       *)
(*  FUNCTION:    The medium level scheduler is responsible for passing processes   *)
(*               that are ready to run to the low level scheduler, and in deciding  *)
(*               whether the remaining processes should be memory resident.         *)
(*                                                                       *)
(*  AUTHOR:      Rick Fisher                                             *)
(*                                                                       *)
(*************************************************************************)

   FROM ProcessManager IMPORT
      (* Types *)
      ProcessID;

   EXPORT QUALIFIED
      (* Procedures *)
      GetListLengths,InitMedSched, Reschedule, Schedule, Terminate;


   PROCEDURE GetListLengths(VAR eligiblesize, ineligiblesize: CARDINAL);
   (*      Find the lengths of the lists of eligible and ineligible processes.       *)
   (*      A process may be considered eligible if there is nothing intrinsic        *)
   (*      to it which keeps it from running–for example, suspended processes,      *)
   (*      if they are memory resident, would be considered eligible, whereas       *)
   (*      blocked processes are ineligible regardless of whether they are in       *)
   (*      memory or not.  No process is eligible unless it is memory resident.     *)

   (*                                                                            *)
   (*      PARAMETERS:      eligiblesize–the number of eligible processes.           *)
   (*                       ineligiblesize–the number of ineligible processes.       *)


   PROCEDURE InitMedSched;
   (*      Order-dependent module initialization.  This procedure consists of       *)
   (*      initialization statements which should not be executed until after the  *)
   (*      initialization for the main module has begun.                           *)


   PROCEDURE Reschedule(process: ProcessID);
   (*      Reschedule a process because of a change of status (status can change    *)
   (*      if a process becomes blocked or unblocked on an IO request, is suspended *)
   (*      or unsuspended, goes to sleep or wakes up, or uses its entire quantum    *)
   (*      in the CPU).                                                            *)
   (*                                                                            *)
   (*      PARAMETERS:      process–the ID of the process to be rescheduled.         *)
```

```
    PROCEDURE Schedule(process: ProcessID);
    (*      Schedule a new process.                                            *)
    (*                                                                         *)
    (*      PARAMETERS:    process–the ID of the process to be scheduled.      *)


    PROCEDURE Terminate(process: ProcessID);
    (*      Terminate a process, removing it from all lists, etc. Should only be *)
    (*      called by HighLevelScheduler.Terminate.                            *)
    (*                                                                         *)
    (*      PARAMETERS:    process–the process being terminated.               *)

END MediumScheduler.
```

```
IMPLEMENTATION MODULE MediumScheduler;

(************************************************************************)
(*                                                                      *)
(*  POLICY:     Swap out processes which are sleeping; swap in processes which   *)
(*              are ready to run, when memory space is available. Pass the eligible *)
(*              processes (i.e., those which could run if current policy allowed)   *)
(*              to the low level scheduler. Processes are allowed to run if their   *)
(*              priority is as high as that of the CURRENT process.               *)
(*                                                                      *)
(*  AUTHOR:     Rick Fisher                                              *)
(*                                                                      *)
(************************************************************************)

    FROM SYSTEM IMPORT
        (* Types *)
        ADDRESS;

    FROM MemoryManager IMPORT
        (* Types *)
        MemoryblockID;

    FROM ProcessManager IMPORT
        (* Types *)
        Priorityqueue, ProcessID, Statustype,

        (* Procedures *)
        ChangeStatus, ContainedInList, Empty, Equal, InitPriorityQueue, Insert,
        Next, Peek, Priority, Remove, Resident, ListSize, NullProcess, Status,
        UpdateProcessInfo;

    IMPORT LowLevelScheduler;
    FROM LowLevelScheduler IMPORT
        (* Procedures *)
        AddToReadyQueue, CurrentProcess (*, Terminate *);

    FROM Swapper IMPORT
        (* Procedures *)
        SwapIn, SwapOut;


    VAR
        eligiblelist,       (* processes eligible for running *)
        ineligiblelist,     (* processes ineligible for running *)
        swapinlist:         (* processes waiting to be swapped in *)
            Priorityqueue;
```

```
PROCEDURE AllowedToRun(process: ProcessID): BOOLEAN;
(*      Determine if a given process may be added to the run queue. Current         *)
(*      policy allows a process to run if its priority is at least as great as that *)
(*      of the current process.                                                     *)
(*                                                                                  *)
(*      PARAMETERS:     process--the process whose permission to run is being      *)
(*                          tested.                                                 *)

BEGIN
    RETURN Priority(process) >= Priority( CurrentProcess() );
END AllowedToRun;


PROCEDURE CheckEligibleList;
(*      Add to run queue all resident processes which policy allows to be added.    *)

BEGIN
    WHILE AllowedToRun( Peek(eligiblelist) ) DO
        AddToReadyQueue( Next(eligiblelist) );
    END; (* WHILE *)
END CheckEligibleList;


PROCEDURE CheckSwapInList;
(*      Swap in as many processes as possible. This procedure adheres strictly      *)
(*      to the order of processes in the swapin list. For example, if a large process *)
(*      which cannot be swapped in due to size constraints is followed by several   *)
(*      smaller processes which could, then no processes will be swapped in at      *)
(*      that time.                                                                  *)

    VAR
        dummy:    (* receives an unused function-return value *)
            ProcessID;

BEGIN
    WHILE NOT Empty(swapinlist) & SwapIn( Peek(swapinlist) ) DO
        dummy := Next(swapinlist);
    END; (* WHILE *)
END CheckSwapInList;


(*  Find the lengths of the eligible and ineligible lists.    *)
PROCEDURE GetListLengths(VAR eligiblesize, ineligiblesize: CARDINAL);

BEGIN
    eligiblesize := ListSize(eligiblelist);
    ineligiblesize := ListSize(ineligiblelist) + ListSize(swapinlist);
END GetListLengths;
```

```
(* Module initialization. *)
PROCEDURE InitMedSched;

BEGIN
    InitPriorityQueue(eligiblelist);
    InitPriorityQueue(ineligiblelist);
    InitPriorityQueue(swapinlist);
END InitMedSched;


(* Reschedule a process, according to the process's new status. *)
PROCEDURE Reschedule(process: ProcessID);

BEGIN
    CASE Status(process) OF
        BLOCKED:
            Insert(process, ineligiblelist);
    |   PENDING:
            IF ContainedInList(process, ineligiblelist) THEN
                Remove(process, ineligiblelist);
                IF Resident(process) THEN
                    Insert(process, eligiblelist);
                ELSE
                    Insert(process, swapinlist);
                    CheckSwapInList;
                END; (* IF Resident *)
            ELSIF NOT ContainedInList(process, eligiblelist) THEN
                Insert(process, eligiblelist);
            END; (* IF Contained *)
    |   SLEEPING:
            SwapOut(process);
            Insert(process, ineligiblelist);
    |   SUSPENDED:
            SwapOut(process);
            Remove(process, swapinlist);
            IF NOT ContainedInList(process, ineligiblelist) THEN
                Insert(process, ineligiblelist);
            END; (* IF NOT Contained *)
    ELSE END; (* CASE *)

    CheckEligibleList;
END Reschedule;


(* Initial scheduling of a process. *)
PROCEDURE Schedule(process: ProcessID);

BEGIN
    ChangeStatus(process, PENDING);
    Reschedule(process);
END Schedule;
```

```
    (* Remove process from lists. *)
    PROCEDURE Terminate(process: ProcessID);

    BEGIN
        Remove(process, swapinlist);
        Remove(process, ineligiblelist);
        Remove(process, eligiblelist);
        LowLevelScheduler.Terminate(process);
    END Terminate;

BEGIN (* module initialization *)

    (* see procedure InitMedSched *)

END MediumScheduler.
```

```
DEFINITION MODULE MemoryManager;

(***********************************************************************)
(*                                                                     *)
(*  FUNCTION:   The memory manager module contains the data structures and  *)
(*              procedures that allow for the correct management of main    *)
(*              memory blocks. This management encompasses allocation, de-  *)
(*              allocation, and keeping up-to-date information on the size and *)
(*              status of any particular block.                             *)
(*                                                                     *)
(*  AUTHOR:     Rick Fisher                                            *)
(*                                                                     *)
(***********************************************************************)


  FROM SYSTEM IMPORT
     (* Constants *)
     BYTESPERWORD,

     (* Types *)
     ADDRESS;

  FROM VirtualMachine IMPORT
     (* Constants *)
     BYTESPERPAGE;

  EXPORT QUALIFIED
     (* Constants *)
     PAGESIZE,

     (* Types *)
     MemoryblockID, Memoryusage, Pageindex, Page, Pageptr,

     (* Procedures *)
     Allocate, BlockID, BlockSize, Deallocate, InitMemory, StartingAddress;


  CONST
     PAGESIZE          =   BYTESPERPAGE DIV BYTESPERWORD;

  TYPE
     MemoryblockID;        (* pointer to a memory block record *)
     Memoryusage       =   RECORD
                              size:   (* Maximum amount of space, in words *)
                                 CARDINAL;
                              free:   (* Percent free *)
                                 REAL;
                           END;
     Pageindex         =   [0 .. BYTESPERPAGE - 1];
     Page              =   ARRAY Pageindex OF CHAR;
     Pageptr           =   POINTER TO Page;
```

```
PROCEDURE Allocate(size: CARDINAL): MemoryblockID;
(*      Allocate a memory block whose size is the smallest PAGESIZE multi-        *)
(*      ple equal to or greater than the requested size in words. Return a       *)
(*      memory block ID, or MemoryblockID(NIL) if unsuccessful.                  *)
(*                                                                               *)
(*      PARAMETERS:    size–the number of words of memory to be allocated.      *)
(*      RETURNS the ID of the allocated memory block.                            *)


PROCEDURE BlockID(address: ADDRESS): MemoryblockID;
(*      Find the memory block containing the specified address. If the address   *)
(*      is in a free memory block, MemoryblockID(NIL) is returned.               *)
(*                                                                               *)
(*      PARAMETERS:    address–the location being searched for.                  *)
(*      RETURNS the memory block ID of the memory block containing the           *)
(*          given address, or MemoryblockID(NIL) if the address is in a free     *)
(*          block.                                                               *)


PROCEDURE BlockSize(memoryblock: MemoryblockID): CARDINAL;
(*      Find the size in words of a given memory block.                          *)
(*                                                                               *)
(*      PARAMETERS:    memoryblock–the id of the memory block whose size         *)
(*                          is desired.                                          *)
(*      RETURNS the size in words of the memory block.                           *)


PROCEDURE Deallocate(VAR memoryblock: MemoryblockID);
(*      Free the specified memory block for reuse.                               *)
(*                                                                               *)
(*      PARAMETERS:    memoryblock–the pointer to the memoryblock being          *)
(*                          freed. The value of memoryblock on returning         *)
(*                          from this procedure will be MemoryblockID(NIL).      *)


PROCEDURE InitMemory;
(*      Order-dependent module initialization. This procedure consists of        *)
(*      initialization statements which should not be executed until after the   *)
(*      initialization for the main module has begun.                            *)


PROCEDURE StartingAddress(memoryblock: MemoryblockID): ADDRESS;
(*      Get the starting address of a memory block.                              *)
(*                                                                               *)
(*      PARAMETERS:    memoryblock–the ID of the memory block on                 *)
(*                          which information is desired.                        *)
(*      RETURNS the starting address of the memory block, or NIL if there        *)
(*          is no such block.                                                    *)

END MemoryManager.
```

```
IMPLEMENTATION MODULE MemoryManager;

(**********************************************************************)
(*                                                                    *)
(*  POLICY:       Allocate blocks using First Fit.                    *)
(*                                                                    *)
(*  AUTHOR:       Rick Fisher                                         *)
(*                                                                    *)
(**********************************************************************)


    FROM SYSTEM IMPORT
        (* Types *)
        ADDRESS,

        (* Procedures *)
        SIZE;

    FROM VirtualMachine IMPORT
        (* Constants *)
        MEMORYSIZE;

    FROM STORAGE IMPORT
        (* Procedures *)
        ALLOCATE, DEALLOCATE;


    CONST
        HIGHPAGE        =   MEMORYSIZE DIV PAGESIZE - 1;

    TYPE
        PageAddress     =   [0 .. HIGHPAGE];
        MemoryblockID   =   POINTER TO Memoryblock;
        Memoryblock     =   RECORD
                                lowerbound,       (* first page in block *)
                                upperbound:       (* last page in block *)
                                    PageAddress;
                                next:             (* next block in list *)
                                    MemoryblockID;
                            END;

    VAR
        freelist,           (* list of free memory blocks *)
        usedlist:           (* list of used memory blocks *)
            MemoryblockID;
```

```
(*   Allocate a memory block.     *)
PROCEDURE Allocate(size: CARDINAL): MemoryblockID;

    VAR
        current,          (* pointer to current position in a list *)
        newblock,         (* pointer to newly created memory block *)
        previous:         (* pointer to previous position in a list *)
            MemoryblockID;

BEGIN

    (* find a suitable memory block *)
    current := freelist;
    previous := freelist;
    WHILE (current # NIL) & (current↑.upperbound - current↑.lowerbound + 1 < size) DO
        previous := current;
        current := current↑.next;
    END;  (* WHILE *)

    IF (current = NIL) OR (size < 0) THEN
        RETURN NIL;
    ELSE

        (* create a new block *)
        ALLOCATE( newblock, SIZE(newblock↑) );
        WITH newblock↑ DO
            lowerbound := current↑.lowerbound;
            upperbound := current↑.lowerbound + size - 1;
        END; (* WITH *)

        (* adjust free list *)
        IF current↑.upperbound = newblock↑.upperbound THEN
            previous↑.next := current↑.next;
        ELSE
            current↑.lowerbound := current↑.lowerbound + size;
        END; (* IF *)

        (* place new block in used list *)
        current := usedlist;
        previous := usedlist;
        WHILE (current # NIL) & (current↑.lowerbound < newblock↑.upperbound) DO
            previous := current;
            current := current↑.next;
        END; (* WHILE *)
        previous↑.next := newblock;
        newblock↑.next := current;

        RETURN newblock;
    END; (* IF *)
END Allocate;
```

```
(* Find the size of a memory block.      *)
PROCEDURE BlockSize(memoryblock: MemoryblockID): CARDINAL;

BEGIN
    IF memoryblock # NIL THEN
        RETURN (memoryblock↑.upperbound - memoryblock↑.lowerbound + 1) * PAGESIZE;
    ELSE
        RETURN 0;
    END;
END BlockSize;


(* Find the memory block containing the specified address. *)
PROCEDURE BlockID(address: ADDRESS): MemoryblockID;

    VAR
        page:        (* page in which address lies *)
            PageAddress;
        current:     (* current position in list *)
            MemoryblockID;

BEGIN
    page := address DIV PAGESIZE;

    (* search for block *)
    current := usedlist;
    WHILE (current # NIL) & (current↑.upperbound < page) DO
        current := current↑.next;
    END; (* WHILE *)

    IF (current = NIL) OR (current↑.lowerbound > page) THEN
        RETURN NIL;
    ELSE
        RETURN current;
    END; (* IF *)
END BlockID;


(* Deallocate a memory block *)
PROCEDURE Deallocate(VAR memoryblock: MemoryblockID);

    VAR
        current,         (* pointer to current position in a list *)
        previous:        (* pointer to previous position in a list *)
            MemoryblockID;

BEGIN

    current := usedlist;
    previous := usedlist;
```

```
      (* search for the memory block *)
      WHILE (current # NIL) & (current # memoryblock) DO
          previous := current;
          current := current↑.next;
      END;  (* WHILE *)

      IF current = memoryblock THEN

          (* remove block from used list *)
          previous↑.next := current↑.next;

          (* find position in free list *)
          current := freelist;
          previous := freelist;
          WHILE (current # NIL) & (current↑.lowerbound < memoryblock↑.upperbound) DO
              previous := current;
              current := current↑.next;
          END; (* WHILE *)

          (* adjust free list *)
          IF previous = NIL THEN
              memoryblock↑.next := NIL;
              freelist := memoryblock;
          ELSE

              (* coalesce with previous if possible, else insert *)
              IF previous↑.upperbound + 1 = memoryblock↑.lowerbound THEN
                  previous↑.upperbound := memoryblock↑.upperbound;
              ELSE
                  memoryblock↑.next := current;
                  previous↑.next := memoryblock;
                  previous := previous↑.next;
              END; (* IF previous↑.upperbound *)

              (* coalesce with current if possible *)
              IF (current # NIL) & (previous↑.upperbound + 1 = current↑.lowerbound) THEN
                  previous↑.upperbound := current↑.upperbound;
                  previous↑.next := current↑.next;
                  DEALLOCATE( current, SIZE(current↑) );
              END; (* IF current # NIL *)
          END; (* IF previous = NIL *)

          memoryblock := NIL;
      END; (* IF current # NIL *)
  END Deallocate;
```

```
(*  Retrieve information on the status of free memory space.       *)
PROCEDURE GetMemoryInfo(VAR memoryInfo: Memoryusage);

    VAR
        current:            (* pointer to current position in list *)
            MemoryblockID;
        count:              (* number of memory words free *)
            CARDINAL;

BEGIN
    memoryInfo.size := MEMORYSIZE;

    count := 0;
    current := freelist;
    WHILE current # NIL DO
        INC(count, current↑.upperbound - current↑.lowerbound +1);
    END; (* WHILE *)
    memoryInfo.free := FLOAT(count * PAGESIZE * 100)/FLOAT(MEMORYSIZE);
END GetMemoryInfo;


(*  Module  initialization.   *)
PROCEDURE InitMemory;

BEGIN

    (* create lists *)
    ALLOCATE( freelist, SIZE(freelist↑) );
    WITH freelist↑ DO
        lowerbound := 0;
        upperbound := HIGHPAGE;
        next := NIL;
    END;  (* WITH *)
    usedlist := NIL;
END InitMemory;


(*  Return the starting address of a memory block.   *)
PROCEDURE StartingAddress(memoryblock: MemoryblockID): ADDRESS;

BEGIN
    IF memoryblock # NIL THEN
        RETURN memoryblock↑.lowerbound * PAGESIZE;
    ELSE
        RETURN NIL;
    END; (* IF *)
END StartingAddress;
```

```
BEGIN (* module initialization *)
   (* see procedure InitMemory *)
END MemoryManager.
```

```
MODULE MultiBatch;

(***************************************************************************)
(*                                                                         *)
(*  FUNCTION:   This is the main module, responsible for initialization of the  *)
(*              system.  It also includes the procedure which is the basis for  *)
(*              the ''Null Process''–what the CPU runs when no other        *)
(*              processes are available.                                   *)
(*                                                                         *)
(*  AUTHOR:     Rick Fisher                                                *)
(*                                                                         *)
(***************************************************************************)

  FROM LocalSystem IMPORT
    (* Constants *)
    TRAP,

    (* Variables *)
    bootcontext, interruptvector, OScontext,

    (* Procedures *)
    Read;

  FROM VirtualMachine IMPORT
    (* Types *)
    ContextID, SVCcode, OSTraps,

    (* Procedures *)
    HighOSBound, Trap;

  FROM MemoryManager IMPORT
    (* Types *)
    MemoryblockID,

    (* Procedures *)
    Allocate, InitMemory;

  FROM DiskManager IMPORT
    (* Procedures *)
    InitDisk;

  FROM ProcessManager IMPORT
    (* Types *)
    ProcessID,

    (* Procedures *)
    ChangeStatus, InitNullProcess, InitProcManager, NullProcess;

  FROM LowLevelScheduler IMPORT
    (* Procedures *)
    InitLLSched;
    FROM MediumScheduler IMPORT
        (* Procedures *)
        InitMedSched;

    FROM HighLevelScheduler IMPORT
        (* Procedures *)
        InitHLSched;

    IMPORT TrapHandler;
    (* not used, but importing it causes all other modules to be initialized *)


    (*  Occupy the CPU when nothing else is available.       *)
    PROCEDURE NullProcedure;

    BEGIN
        LOOP
        END; (* LOOP *)
    END NullProcedure;


    VAR
        OSmemoryblock:   (* the ID of the memory block containing the operating system *)
            MemoryblockID;


BEGIN (* system initialization *)

    (* save and adjust the context of the operating system *)
    OScontext := bootcontext;

    (* set the context and register values *)
    Trap(INITIALIZE);

    (* order-dependent initialization of other modules *)
    InitMemory;
    InitDisk;
    InitProcManager;
    InitLLSched;
    InitMedSched;
    InitHLSched;

    (* reserve first memory block for the operating system *)
    OSmemoryblock := Allocate( CARDINAL( HighOSBound() ) + 1 );

    (* set up the null process *)
    InitNullProcess( ContextID(interruptvector[TRAP].formercontext), OSmemoryblock );
    (* start card reader *)
    Read;

    (* begin executing null process *)
    NullProcedure;

END MultiBatch. (* should never get here *)
```

```
DEFINITION MODULE OSSTORAGE;

(***********************************************************************)
(*  FUNCTION:   Increase heap size for a process. It is up to the standard       *)
(*              module STORAGE, which exports the ALLOCATE and DEALLO-           *)
(*              CATE routines, to keep track of what heap locations are in       *)
(*              use and which have been freed.                                   *)
(*                                                                               *)
(*  AUTHOR:     jd, Cambridge University Computer Laboratory                     *)
(*              Documented by Rick Fisher                                        *)
(*                                                                               *)
(***********************************************************************)

    FROM SYSTEM IMPORT
        (* Types *)
        ADDRESS;

    EXPORT QUALIFIED
        (* Constants *)
        AddressesPerUnit, WordAlign,

        (* Procedures *)
        HeapAllocate;


    CONST
        AddressesPerUnit  =  1;      (* number of addresses per WORD *)
        WordAlign         =  TRUE; (* if TRUE, all objects allocated from the heap must be
                                        word aligned. *)


    PROCEDURE HeapAllocate (amount: CARDINAL; VAR base: ADDRESS;
        VAR free: CARDINAL): BOOLEAN;
    (*      This procedure incorporates ALL operating system dependencies           *)
    (*      about heap storage allocation.                                          *)
    (*                                                                              *)
    (*      PARAMETERS:     amount–the number of words of contiguous heap           *)
    (*                          storage required.                                   *)
    (*                      base–the current base address of storage already        *)
    (*                          used by the process.  Must be NIL on the            *)
    (*                          process's first call to this procedure.             *)
    (*                          Addresses greater than base contain areas of        *)
    (*                          the heap which have been used by the process.       *)
    (*                      free–the number of free words currently below           *)
    (*                          ''base'', but allotted to the heap.                 *)
    (*                                                                              *)
    (*      On successful return, ''base – free'' will contain the low address of a *)
    (*      contiguous block of free storage, and free (>= amount) will contain its *)
    (*      size in storage units.                                                  *)

END OSSTORAGE.
```

```
IMPLEMENTATION MODULE OSSTORAGE;

(**********************************************************************)
(*                                                                    *)
(*  POLICY:      The heap grows downward (towards the stack).  Each call for more  *)
(*               storage allocates at least the amount of space requested, if      *)
(*               available; otherwise it allocates enough so that, coupled with    *)
(*               what was already free, the total equals the amount requested--if  *)
(*               available.  Otherwise, it fails.                                  *)
(*                                                                    *)
(*  AUTHOR:      Rick Fisher                                          *)
(*                                                                    *)
(**********************************************************************)

    FROM SYSTEM IMPORT
        (* Constants *)
        BYTESPERWORD,

        (* Types *)
        ADDRESS, WORD,

        (* Procedures *)
        SIZE;

    FROM VirtualMachine IMPORT
        (* Types *)
        OSTraps, SVCcode,

        (* Procedures *)
        ContextSize, HighOSBound, Trap;

    FROM MemoryManager IMPORT
        (* Constants *)
        PAGESIZE;

    FROM ProcessManager IMPORT
        (* Procedures *)
        Equal, NullProcess;

    FROM LowLevelScheduler IMPORT
        (* Procedures *)
        CurrentProcess;

    FROM SVCalls IMPORT
        (* Procedures *)
        UpperBound;


    CONST
        MINIMUMINCREMENT = PAGESIZE;
```

```
(*  Allocate some heap space.  *)
PROCEDURE HeapAllocate (amount: CARDINAL; VAR base: ADDRESS;
    VAR free: CARDINAL): BOOLEAN;

BEGIN

    IF (base = NIL) THEN
    (* this is the first allocation for the process *)

        IF Equal( CurrentProcess(), NullProcess() ) THEN
        (* process is operating system–leave room for null context *)

            base := HighOSBound() - ContextSize();

        ELSE
        (* process is user process *)

            base := UpperBound();
        END; (* IF Equal *)
    END; (* IF base *)

    (* allocate space if possible *)
    IF (amount <= MINIMUMINCREMENT) & ( AllocateHeap(MINIMUMINCREMENT) ) THEN
        INC (free, MINIMUMINCREMENT);
    ELSIF AllocateHeap(amount) THEN
        INC (free, amount);
    ELSIF AllocateHeap(amount - free) THEN
        free := amount;
    ELSE
        RETURN FALSE;
    END;

    RETURN TRUE;
END HeapAllocate;


PROCEDURE AllocateHeap(amount: CARDINAL): BOOLEAN;
(*      Increase the amount of user available memory alloted to the heap.  This        *)
(*      is actually a Supervisor Call whose use is restricted to this module.           *)
(*                                                                                      *)
(*      PARAMETERS:     amount–the size in words of the memory to be allocated.         *)
(*      RETURNS TRUE if the allocation is successful, FALSE otherwise.                  *)

    VAR
        success:            (* TRUE if successful *)
            BOOLEAN;

BEGIN
    ThreeParmSVC(HEAPALLOCATESVC, amount, success);
    RETURN success;
END AllocateHeap;
```

```
    PROCEDURE ThreeParmSVC(svc: SVCcode; amount: CARDINAL;
        VAR success: BOOLEAN);
    (*      Fix the parameters to ''AllocateHeap'' on the stack, and cause a       *)
    (*      trap.                                                                   *)

    BEGIN
        Trap(SVC);
    END ThreeParmSVC;

BEGIN
END OSSTORAGE.
```

```
DEFINITION MODULE ProcessManager;

(**************************************************************************)
(*                                                                        *)
(*  FUNCTION:   The process manager module contains three categories of functions   *)
(*              and procedures:  1) to create and remove processes, and update       *)
(*              information on them; 2) to manage lists of processes, organized      *)
(*              as priority queues; and 3) to keep track of events which are         *)
(*              scheduled for processes.                                             *)
(*                                                                        *)
(*  AUTHOR:     Rick Fisher                                               *)
(*                                                                        *)
(**************************************************************************)

   FROM SYSTEM IMPORT
      (* Types *)
      ADDRESS;

   FROM VirtualMachine IMPORT
      (* Types *)
      AllTraps, ContextID;

   FROM MemoryManager IMPORT
      (* Types *)
      MemoryblockID, Pageptr;

   FROM DiskManager IMPORT
      (* Types *)
      DiskblockID;

   EXPORT QUALIFIED
      (* Types *)
      Actiontype, Disklist, Disklistnode, Priorityqueue, Prioritytype, ProcessID, Statustype,
      Updatecode,

      (* Procedures *)
      ChangeStatus, CheckSchedule, ContainedInList, Context, CreateID, DiskUse, Empty,
      Equal, GetNextInput, GetNextOutput, Initialize, InitNullProcess, InitPriorityQueue,
      InitProcManager, Insert, LinkToOutput, ListSize, MemoryLocation, Next, NullProcess,
      OutputList, Peek, PermanentLocation, Priority, ProcessSize, Remove, Resident, Schedule,
      Status, StoreNextInput, StoreNextOutput, TrapReason, UnSchedule, UpdateProcessInfo;


   (**********************************************************************)
   (*                       1. Process Management                        *)
   (**********************************************************************)

   CONST
      MAXPRIORITY   =   5;
```

```
TYPE
    Disklist      =   POINTER TO Disklistnode;
    Disklistnode  =   RECORD
                          diskblockptr:  (* address of first disk block used by process *)
                              DiskblockID;
                          next:          (* next node in list *)
                              Disklist;
                      END;
    Prioritytype  =   [0 .. MAXPRIORITY];
    ProcessID;
    Statustype    =   (BLOCKED, CURRENT, INITIALIZING, READY, PENDING,
                       SLEEPING, SUSPENDED, TERMINATED);
    Updatecode    =   (SWAPIN, SWAPOUT);


PROCEDURE ChangeStatus(process: ProcessID; status: Statustype);
(*      Change the status of a process.                                          *)
(*                                                                               *)
(*      PARAMETERS:     process–the process whose status is being changed.       *)
(*                      status–the new status of the process. Valid status values *)
(*                      are:                                                     *)
(*                          BLOCKED:        Process is awaiting the comple-      *)
(*                                              tion of some event.              *)
(*                          CURRENT:        Process currently has control of     *)
(*                                              the CPU.                         *)
(*                          PENDING:        Process is ready to run, but is      *)
(*                                              temporarily prohibited from      *)
(*                                              doing so by  current policy.     *)
(*                          READY:          Process is ready to run.             *)
(*                          SLEEPING:       Process has voluntary suspended      *)
(*                                              itself for a fixed time.         *)
(*                          SUSPENDED:      Process has been indefinitely        *)
(*                                              suspended by the operating       *)
(*                                              system.                          *)
(*                          TERMINATED:     Process has finished; process        *)
(*                                              ID is available for reuse.       *)


PROCEDURE Context(process: ProcessID): ContextID;
(*      Find the context ID of a process.                                        *)
(*                                                                               *)
(*      PARAMETERS:     process–the ID of the process whose context is desired.  *)
(*      RETURNS the process's context ID.                                        *)
```

```
PROCEDURE CreateID(): ProcessID;
(*      Create a new process ID.                                                    *)
(*                                                                                  *)
(*      RETURNS the ID of the new process.  If no more processes can be             *)
(*           created, returns the ID of the null process.                           *)
(*                                                                                  *)
(*      NOTE:  This process does NOT create a process or reserve a process          *)
(*      ID.  If the process ID returned by this procedure is not initialized (see   *)
(*      Initialize, below), then the same ID could be returned by this procedure    *)
(*      at a later date.                                                            *)


PROCEDURE DiskUse(process: ProcessID): Disklist;
(*      Reveal the disk blocks used by a process.                                   *)
(*                                                                                  *)
(*      PARAMETERS:     process–the process whose disk usage information is         *)
(*                           desired.                                               *)
(*      RETURNS a pointer to the list of disk blocks.                               *)


PROCEDURE Equal(process1, process2: ProcessID): BOOLEAN;
(*      Determine whether two process IDs are equal.                                *)
(*                                                                                  *)
(*      PARAMETERS:     process1, process2–the IDs of the processes in question.    *)


PROCEDURE GetNextInput(process: ProcessID; VAR pageaddress: Pageptr;
    VAR byte: CARDINAL);
(*      Find the location in memory of the next byte to be read from the input     *)
(*      for the given process.                                                      *)
(*                                                                                  *)
(*      PARAMETERS:     process–the process whose input is being read.              *)
(*                  pageaddress–the address of the memory page in                   *)
(*                      which the next input item lies.                             *)
(*                  byte–the index into the given page at which                     *)
(*                      which the next input item lies.                             *)


PROCEDURE GetNextOutput(process: ProcessID; VAR pageaddress: Pageptr;
    VAR byte: CARDINAL; VAR memoryblock: MemoryblockID);
(*      Find the next byte location in memory to be written with the output for the *)
(*      given process.                                                              *)
(*                                                                                  *)
(*      PARAMETERS:     process–the process whose output is being read.             *)
(*                          pageaddress–the address of the memory page in           *)
(*                              which the next output item lies.                    *)
(*                          byte–the index into the given page at which the         *)
(*                              next output item lies.                              *)
(*                          memoryblock–the memoryblock on which the page           *)
(*                              resides.                                            *)
```

```
PROCEDURE Initialize(process: ProcessID; memoryblock: MemoryblockID; input,
    stackbase: ADDRESS; stacksize: CARDINAL; processpriority: Prioritytype;
    codestart, codeEnd: Disklist);
(*     Attach a process ID to an actual process.                                  *)
(*                                                                                *)
(*     PARAMETERS:     process–the ID of the new process.                         *)
(*                     memoryblock–the memory block in which the process          *)
(*                          is stored.                                            *)
(*                     input–the starting address of input for the process.      *)
(*                     stackbase–the first address past the end of the input;    *)
(*                          the caller must ensure that the stack and input       *)
(*                          do not overlap.                                       *)
(*                     stacksize–the size of the stack, in words.                 *)
(*                     processpriority–the priority of the process.              *)
(*                     codestart–the list of diskblocks holding the code (the    *)
(*                          first blocks of output for the process, not the      *)
(*                          disk location for the process).                       *)
(*                     codeEnd–the last in the list of above code blocks.        *)


PROCEDURE InitNullProcess(nullcontext: ContextID; memoryblock: MemoryblockID);
(*     Special initialization for the null process.                               *)
(*                                                                                *)
(*     PARAMETERS:     nullcontext–the duplicate OS context being used for       *)
(*                          the null process's context.                           *)
(*                     memoryblock–the memoryblock holding the code for the      *)
(*                          null process (i.e., for the OS).                      *)


PROCEDURE InitProcManager;
(*     Order-dependent module initialization. This procedure consists of         *)
(*     initialization statements which should not be executed until after the   *)
(*     initialization for module MemoryManager.                                   *)


PROCEDURE LinkToOutput(process: ProcessID; diskblock: DiskblockID);
(*     Link a diskblock to a process's output list.                               *)
(*                                                                                *)
(*     PARAMETERS:     process–the process whose outputlist is to be augmented.  *)
(*                     diskblock–the block to be added.                           *)


PROCEDURE MemoryLocation(process: ProcessID): MemoryblockID;
(*     Find the memory block where the process is stored.                         *)
(*                                                                                *)
(*     PARAMETERS:     process–the process whose location is desired.            *)
(*     RETURNS the ID of the correct memory block (NIL if process is not in      *)
(*          memory).                                                              *)
```

```
PROCEDURE NullProcess(): ProcessID;
(*      RETURNS the process ID for the Null Process.                          *)


PROCEDURE OutputList(process: ProcessID): Disklist;
(*      Finds the list of diskblocks constituting a process's output.         *)
(*                                                                            *)
(*      PARAMETERS:     process–the process whose output is desired.          *)
(*      RETURNS the list of diskblocks.                                       *)


PROCEDURE PermanentLocation(process: ProcessID): DiskblockID;
(*      Find a pointer to the disk block where the process is stored.         *)
(*                                                                            *)
(*      PARAMETERS:     process–the process whose location is desired.        *)
(*      RETURNS a pointer to the correct disk block (NIL if process has never *)
(*          been swapped out of memory).                                      *)


PROCEDURE Priority(process: ProcessID): Prioritytype;
(*      Obtain the priority of a given process.                               *)
(*                                                                            *)
(*      PARAMETERS:     process–the ID of the process whose priority is desired. *)
(*      RETURNS the priority of the process.                                  *)


PROCEDURE ProcessSize(process: ProcessID): CARDINAL;
(*      Finds the number of words in a process.                               *)
(*                                                                            *)
(*      PARAMETERS:     process–the process whose size is desired.            *)
(*      RETURNS the number of words taken by the process.                     *)


PROCEDURE Resident(process: ProcessID): BOOLEAN;
(*      Determine whether a process is resident in main memory.              *)
(*                                                                            *)
(*      PARAMETERS:     process–the ID of the process in question.            *)
(*      RETURNS TRUE if process curently resides in main memory, FALSE        *)
(*          otherwise.                                                        *)


PROCEDURE Status(process: ProcessID): Statustype;
(*      Find the status of a process.                                         *)
(*                                                                            *)
(*      PARAMETERS:     process–the process whose status is desired.          *)
(*      RETURNS the status of the process.                                    *)
```

```
PROCEDURE StoreNextInput(process: ProcessID; pageaddress: Pageptr; byte: CARDINAL);
(*      Store the location in memory of the next byte to be read from input for      *)
(*      the given process.                                                             *)
(*                                                                                     *)
(*      PARAMETERS:     process–the process whose input is being read.                 *)
(*                      pageaddress–the address of the memory page in                  *)
(*                          which the next input item lies.                            *)
(*                      byte–the index into the given page at which                    *)
(*                          which the next input item lies.                            *)


PROCEDURE StoreNextOutput(process: ProcessID; pageaddress: Pageptr; byte: CARDINAL);
(*      Store the next byte location in memory to be written with output from the     *)
(*      given process. If the parameter ''byte'' equals BYTESPERPAGE, 0                *)
(*      will be stored and ''pageaddress'' advanced to the next page.                  *)
(*                                                                                     *)
(*      PARAMETERS:     process–the process whose output is being read.                *)
(*                      pageaddress–the address of the memory page in                  *)
(*                          which the next output item lies.                           *)
(*                      byte–the index into the given page at which the                *)
(*                          next output item lies.                                     *)


PROCEDURE TrapReason(process: ProcessID): AllTraps;
(*      Return the reason for a trap.                                                  *)
(*                                                                                     *)
(*      PARAMETERS:     process–the ID of the process which was running                *)
(*                          when the trap occurred.                                    *)
(*      RETURNS the cause of the trap.                                                 *)


PROCEDURE UpdateProcessInfo(process: ProcessID; reason: Updatecode;
    newlocationptr: ADDRESS);
(*      Update the information stored about a process according to the informa-       *)
(*      tion given.                                                                    *)
(*                                                                                     *)
(*      PARAMETERS:     process–the ID of the process whose information is             *)
(*                          being updated.                                             *)
(*                      reason–may be one of:                                          *)
(*                          SWAPIN:   The process is being swapped in from             *)
(*                              disk.                                                  *)
(*                          SWAPOUT:The process is being swapped out to                *)
(*                              disk.                                                  *)
(*                      newlocationptr–a pointer to the memory block or disk           *)
(*                          block to which the process is being moved.                 *)
```

```
(*************************************************************************)
(*                     2. Priority Queue Management                      *)
(*************************************************************************)

TYPE
    Priorityqueue;


PROCEDURE ContainedInList(process: ProcessID; list: Priorityqueue): BOOLEAN;
(*      Determine if a process is in a particular list.                  *)
(*                                                                       *)
(*      PARAMETERS:     process–the process being sought.                *)
(*                      list–the list in which the process might be.     *)


PROCEDURE Empty(list: Priorityqueue): BOOLEAN;
(*      Reveal if a given list is empty.                                 *)
(*                                                                       *)
(*      PARAMETERS:     list–the list being examined.                    *)
(*      RETURNS TRUE if the list is empty, FALSE otherwise.              *)


PROCEDURE InitPriorityQueue(VAR list: Priorityqueue);
(*      Mandatory initialization for any variable of type Priorityqueue. *)
(*                                                                       *)
(*      PARAMETERS:     list–the variable being initialized.             *)


PROCEDURE Insert(process: ProcessID; VAR list: Priorityqueue);
(*      Place the process ID into the correct location in the given list. No action *)
(*      is taken if the process is already in the list.                  *)
(*                                                                       *)
(*      PARAMETERS:     list–the list of process IDs.                    *)
(*                      process–the process ID to be inserted into the list. *)


PROCEDURE ListSize(list: Priorityqueue): CARDINAL;
(*      Find the number of elements in a list.                           *)
(*                                                                       *)
(*      PARAMETERS:     list–the list whose size is desired.             *)
(*      RETURNS the size of the list.                                    *)


PROCEDURE Next(VAR list: Priorityqueue): ProcessID;
(*      Remove the next process ID from the given list. If the list is empty, the *)
(*      process ID for the null process will be returned.                *)
(*                                                                       *)
(*      PARAMETERS:     list–the list of process IDs.                    *)
(*      RETURNS the process ID of the next process in the list.          *)
```

```
PROCEDURE Peek(list: Priorityqueue): ProcessID;
(*      Obtain the next process ID from the given list.  If the list is empty, the      *)
(*      process ID for the null process will be returned.                               *)
(*      The list is not changed by this procedure.                                      *)
(*                                                                                      *)
(*      PARAMETERS:     list–the list of process IDs.                                   *)
(*      RETURNS the process ID of the next process in the list.                         *)


PROCEDURE Remove(process: ProcessID; VAR list: Priorityqueue);
(*      Remove the given process ID from the list, if it is there; otherwise, do        *)
(*      nothing.  The process does not have to be next on the list.                     *)
(*                                                                                      *)
(*      PARAMETERS:     process–the process to be removed.                              *)
(*                      list–the list from which the process is to be removed.          *)


(************************************************************************************)
(*                              3. Event Scheduling                                 *)
(************************************************************************************)

TYPE
   Actiontype     =  (NULL, RESCHEDULE, TERMINATE, TIMEDPREEMPT, WAKEUP);


PROCEDURE CheckSchedule(VAR act: Actiontype; VAR pid: ProcessID);
(*      Determine if an event should occur, and for which process.  (No action is       *)
(*      actually taken by the Process Manager.)                                         *)
(*                                                                                      *)
(*      PARAMETERS:     act–the action to be taken.  May be one of:                     *)
(*                          NULL:           No action to be taken at this               *)
(*                                              time.                                   *)
(*                          RESCHEDULE:         Send the process to the                 *)
(*                                              Medium Level Scheduler for              *)
(*                                              rescheduling.                           *)
(*                          TERMINATE:      Terminate the current process.              *)
(*                          TIMEDPREEMPT:   Preempt the current process in              *)
(*                                              favor of the next waiting               *)
(*                                              process.                                *)
(*                          WAKEUP:         Wake up a sleeping process.                 *)
(*                      pid–the process whose event is to be scheduled.                 *)
(*      NOTE: Returns only one set of values at a time, though several events can       *)
(*      come due at once.                                                               *)
```

```
    PROCEDURE Schedule(when: CARDINAL; act: Actiontype; pid: ProcessID);
    (*      Schedule an event.                                                    *)
    (*                                                                            *)
    (*      PARAMETERS:     when–the number of ticks before the event should      *)
    (*                          occur.                                            *)
    (*                      act–the action to be taken. May be RESCHEDULE,        *)
    (*                          TERMINATE, TIMEDPREEMPT, or WAKEUP                *)
    (*                          (see Procedure CheckSchedule for descriptions).   *)
    (*                          NULL should not be scheduled.                     *)
    (*                      pid–the process for which the event should occur.     *)

    PROCEDURE UnSchedule(process: ProcessID);
    (*      Remove all events scheduled for a given process.                      *)
    (*                                                                            *)
    (*      PARAMETERS:     process–the process whose event should be unscheduled. *)

END ProcessManager.
```

```
IMPLEMENTATION MODULE ProcessManager;

(************************************************************************)
(*                                                                      *)
(*  POLICY:      Process control blocks are stored in an array, which holds at      *)
(*               most MAXPROCESS processes.  Doubly linked lists of process         *)
(*               IDs are also stored in arrays with header and tail nodes.          *)
(*               The process ID serves as the index into the array.  Lists are      *)
(*               priority queues: setting all priorities equal will result in FIFO *)
(*               queues.  Scheduled events are stored in a delta list which         *)
(*               is singly linked, and which is dynamically allocated since         *)
(*               a process may have more than one scheduled event.  The Process     *)
(*               Manager does not take any action on scheduled events, other        *)
(*               than to record that the event is scheduled.                        *)
(*                                                                      *)
(*  AUTHOR:      Rick Fisher                                            *)
(*                                                                      *)
(************************************************************************)

    FROM SYSTEM IMPORT
        (* Constants *)
        MAXCARD,

        (* Types *)
        ADDRESS,

        (* Procedures *)
        SIZE;

    IMPORT VirtualMachine;
    FROM VirtualMachine IMPORT
        (* Constants *)
        BYTESPERPAGE,

        (* Types *)
        AllTraps, ContextID,

        (* Procedures *)
        ContextBounds, NewContext, (* TrapReason, *) UpdateContext;

    FROM MemoryManager IMPORT
        (* Constants *)
        PAGESIZE,

        (* Types *)
        MemoryblockID, Pageindex, Pageptr,

        (* Procedures *)
        Allocate, StartingAddress;
```

```
FROM DiskManager IMPORT
    (* Types *)
    DiskblockID;

FROM Clock IMPORT
    (* Procedures *)
    TickCount;

FROM STORAGE IMPORT
    (* Procedures *)
    ALLOCATE, DEALLOCATE;

CONST
    BUFFERSIZE      =   2;
    HEAD            =   0;
    MAXPROCESS      =  25;
    TAIL            =  MAXPROCESS + 1;


TYPE
    Bufferindex     =  [0 .. BUFFERSIZE - 1];
    Circularbuffer  =  RECORD
                           buffer:               (* the array of pages and memory-
                                                      blocks *)
                               ARRAY Bufferindex OF
                               RECORD
                                   page:          (* pointer to the array of characters *)
                                       Pageptr;
                                   byte:          (* the index into "page" *)
                                       Pageindex;
                                   memoryblock:   (* ID of the memoryblock containing
                                                              "page" *)
                                       MemoryblockID;
                               END;
                           currentpage:          (* index into "buffer" *)
                               Bufferindex;
                       END;
    ProcessID       =  [0 .. MAXPROCESS];
    Eventptr        =  POINTER TO Event;
    Event           =  RECORD
                           action:       (* RESCHEDULE, TERMINATE,
                                            TIMEDPREEMPT, WAKEUP *)
                               Actiontype;
                           timeleft:     (* time between previous event (or present
                                         moment) and action *)
                               CARDINAL;
                           process:      (* ID of process owning the event *)
                               ProcessID;
                           next:         (* pointer to next event *)
                               Eventptr;
                       END;
```

```
Listpriority        =  [-1 .. MAXPRIORITY + 1];
Listrange           =  [HEAD .. TAIL];
Listelement         =  RECORD
                          priority:       (* process priority *)
                             Listpriority;
                          next,           (* next process in list *)
                          previous:       (* previous process *)
                             Listrange;
                       END;
Locationrecord      =  RECORD
                          resident:       (* TRUE if process is in memory *)
                             BOOLEAN;
                          memory:         (* ID of process's memory block *)
                             MemoryblockID;
                          diskblock:      (* ID of process's disk block *)
                             DiskblockID;
                       END;
Priorityqueue       =  POINTER TO ARRAY Listrange OF Listelement;
Systemusage         =  RECORD
                          quantumsize,    (* maximum time process can run before
                                                clock interrupt *)
                          timecredit,     (* amount of last quantum unused *)
                          servicelimit,   (* maximum time process can run before
                                                being rescheduled *)
                          cputime:        (* time spent in CPU *)
                             CARDINAL;
                       END;
Processblock        =  RECORD
                          context:        (* saved registers etc. *)
                             ContextID;
                          diskuse:        (* list of blocks allocated by the process *)
                             Disklist;
                          location:       (* information on process's location *)
                             Locationrecord;
                          nextinput:      (* address of next input *)
                             RECORD
                                page:     (* page of next input *)
                                   Pageptr;
                                byte:     (* index into "page" *)
                                   Pageindex;
                             END;
                          output:         (* the process's output *)
                             RECORD
                                first,
                                last:     (* diskblocks of output *)
                                   Disklist;
                                current: (* memory buffer for recent output *)
                                   Circularbuffer;
                             END;
```

```
                    priority:        (* process priority *)
                        Prioritytype;
                    status:          (* activity status *)
                        Statustype;
                    systemtime:      (* time statistics on process *)
                        Systemusage;
                END;


VAR
    deltalist:          (* list of scheduled events *)
        Eventptr;
    i,                  (* loop index *)
    j:                  (* loop index *)
        CARDINAL;
    previoustime:       (* tick count last time head of delta list was changed *)
        CARDINAL;
    processtable:       (* the array of process blocks *)
        ARRAY ProcessID OF Processblock;


(*  Change a process's status.  *)
PROCEDURE ChangeStatus(process: ProcessID; status: Statustype);

BEGIN
    processtable[process].status := status;
END ChangeStatus;


(*  Check the delta list.       *)
PROCEDURE CheckSchedule(VAR act: Actiontype; VAR pid: ProcessID);

    VAR
        currenttime:    (* tick count at start of procedure *)
            CARDINAL;
        temp:           (* used for deallocating old list nodes *)
            Eventptr;
        timepassed:     (* time since head of delta list was changed *)
            CARDINAL;

BEGIN
    IF deltalist = NIL THEN
    (* nothing is scheduled *)

        act := NULL;
        pid := NullProcess();
    ELSE
        currenttime := TickCount();

        (* check for wraparound in the tick counter *)
        IF currenttime >= previoustime THEN
            timepassed := currenttime - previoustime;
        ELSE
            timepassed := MAXCARD - previoustime + currenttime + 1;
        END; (* IF *)

        WITH deltalist↑ DO
            IF timeleft > timepassed THEN
            (* nothing happens yet *)

                act := NULL;
                pid := NullProcess();
            ELSE

                (* get information and dispose of node *)
                DEC(timepassed, timeleft);
                act := action;
                pid := process;
                temp := deltalist;
                deltalist := next;
                DEALLOCATE( temp, SIZE(temp↑) );
            END; (* IF timeleft *)
        END; (* WITH *)

        IF deltalist↑.timeleft > timepassed THEN
            DEC(deltalist↑.timeleft, timepassed);
        ELSE
            deltalist↑.timeleft := 0;
        END; (* IF deltalist↑.timeleft *)
        previoustime := currenttime;
    END; (* IF deltalist = NIL *)
END CheckSchedule;


(*  Determine if a process is in a particular list.      *)
PROCEDURE ContainedInList(process: ProcessID; list: Priorityqueue): BOOLEAN;
BEGIN
    RETURN (list↑[HEAD].next = Listrange(process) ) OR (list↑[process].previous # HEAD);
END ContainedInList;


(*  Return a process's context ID.   *)
PROCEDURE Context(process: ProcessID): ContextID;

BEGIN
    RETURN(processtable[process].context);
END Context;
```

```
(*  Create a new process ID.    *)
PROCEDURE CreateID(): ProcessID;

    VAR
        i:   CARDINAL;   (* loop index *)

BEGIN

    (* find an unused process ID *)
    i := 1;        (* 0 is the null process–always ready, never unused *)
    WHILE (i <= MAXPROCESS) & (processtable[i].status # TERMINATED) DO
        INC(i);
    END; (* WHILE *)

    IF i > MAXPROCESS THEN
        RETURN( NullProcess() );
    ELSE
        RETURN(i);
    END; (* IF *)

END CreateID;


(*  Determine a process's disk resource usage.  *)
PROCEDURE DiskUse(process: ProcessID): Disklist;

BEGIN
    RETURN processtable[process].diskuse;
END DiskUse;


(*  Determine if a list is empty. *)
PROCEDURE Empty(list: Priorityqueue): BOOLEAN;

BEGIN
    RETURN(list↑[HEAD].next = TAIL);
END Empty;


(*  Check if processes are the same.     *)
PROCEDURE Equal(process1, process2: ProcessID): BOOLEAN;

BEGIN
    RETURN process1 = process2;
END Equal;
```

```
(*  Get location of next byte of input.     *)
PROCEDURE GetNextInput(process: ProcessID; VAR pageaddress: Pageptr;
    VAR byte: CARDINAL);

BEGIN
    WITH processtable[process] DO
        pageaddress := nextinput.page;
        byte := nextinput.byte;
    END; (* WITH *)
END GetNextInput;


(*  Get the next output location. *)
PROCEDURE GetNextOutput(process: ProcessID; VAR pageaddress: Pageptr;
    VAR byte: CARDINAL; VAR memoryblock: MemoryblockID);

BEGIN
    WITH processtable[process].output.current DO
        pageaddress := buffer[currentpage].page;
        byte := buffer[currentpage].byte;
    END; (* WITH *)
END GetNextOutput;


(*  Initialize a process block.    *)
PROCEDURE Initialize(process: ProcessID; memoryblock: MemoryblockID;
    input, stackbase: ADDRESS; stacksize: CARDINAL; processpriority: Prioritytype;
    codestart, codeEnd: Disklist);

BEGIN
    WITH processtable[process] DO
        context := NewContext( StartingAddress(memoryblock), stackbase, stacksize );

        nextinput.page := input DIV PAGESIZE;
        nextinput.byte := 0;

        WITH output DO
            first := codestart;
            last := codeEnd;
            WITH current DO
                currentpage := 0;
                buffer[currentpage].byte := 0;
            END; (* WITH current *)
        END; (* WITH output *)

        priority := processpriority;
        location.resident := TRUE;
        location.memory := memoryblock;
        status := INITIALIZING;
    END; (* WITH processtable *)
END Initialize;
```

```
(*  Initialize the null process.   *)
PROCEDURE InitNullProcess(nullcontext: ContextID; memoryblock: MemoryblockID);

BEGIN
    WITH processtable[NullProcess()] DO
        context := nullcontext;
        priority  := 0;
        location.resident := TRUE;
        location.memory := memoryblock;
        status := CURRENT;
    END; (* WITH *)
END InitNullProcess;


(*  Module initialization.   *)
PROCEDURE InitProcManager;

BEGIN

    (* Initialize processtable *)
    FOR i := 1 TO MAXPROCESS DO
        WITH processtable[i] DO
            status := TERMINATED;
            WITH output.current DO
                FOR j := 0 TO BUFFERSIZE - 1 DO
                    WITH buffer[j] DO
                        memoryblock := Allocate(PAGESIZE);
                        page := StartingAddress(memoryblock);
                    END; (* WITH buffer *)
                END; (* FOR *)
            END; (* WITH output *)
        END; (* WITH processtable *)
    END; (* FOR *)

    (* Initialize delta list *)
    deltalist := NIL;

END InitProcManager;


(*  Initialize a process queue.   *)
PROCEDURE InitPriorityQueue(VAR list: Priorityqueue);

    VAR
        i:    CARDINAL;    (* loop index *)

BEGIN

    (* set up list with header and tail nodes *)
    ALLOCATE( list, SIZE(list↑) );
    WITH list↑[HEAD] DO
        priority := MAXPRIORITY + 1;
        next := TAIL;
    END; (* WITH *)
    WITH list↑[TAIL] DO
        priority := -1;
        previous := HEAD;
    END; (* WITH *)

    (* indicate each process is not in list *)
    FOR i := 1 TO MAXPROCESS DO
        list↑[i].previous := HEAD;
    END; (* FOR *)

END InitPriorityQueue;


(*  Insert process ID into the list.   *)
PROCEDURE Insert(process: ProcessID; VAR list: Priorityqueue);

    VAR
        priority:           (* the priority of the process *)
            Listpriority;
        current:            (* index of current element in list *)
            Listrange;
        next:               (* the next process after the one being inserted. *)
            ProcessID;

BEGIN
    IF NOT ContainedInList(process, list) THEN

        (* find correct place in list *)
        priority := processtable[process].priority;
        current := TAIL;
        WHILE priority > list↑[current].priority DO
            current := list↑[current].previous;
        END; (* WHILE *)

        (* insert process ID *)
        next := list↑[current].next;
        list↑[process].previous := current;
        list↑[process].next := next;
        list↑[next].previous := process;
        list↑[current].next := process;

    END; (* IF *)
END Insert;
```

```
(* Link a diskblock to a process's output list.   *)
PROCEDURE LinkToOutput(process: ProcessID; diskblock: DiskblockID);

    VAR
        nodeptr:     (* the new node for storing the diskblock *)
            Disklist;

BEGIN

    (* create node *)
    ALLOCATE( nodeptr, SIZE(nodeptr↑) );
    WITH nodeptr↑ DO
        diskblockptr := diskblock;
        next := NIL;
    END; (* WITH *)

    (* add node to end of list *)
    WITH processtable[process].output DO
        last↑.next := nodeptr;
        last := nodeptr;
    END; (* WITH *)
END LinkToOutput;

(* Obtain the number of elements in a list.  *)
PROCEDURE ListSize(list: Priorityqueue): CARDINAL;

    VAR
        count:        (* number of items in list *)
            CARDINAL;
        current:      (* index of current list item *)
            Listrange;

BEGIN
    count := 0;
    current := HEAD;
    WHILE list↑[current].next # TAIL DO
        current := list↑[current].next;
        INC(count);
    END;

    RETURN(count);
END ListSize;
```

```
(*  Return a pointer to the memory block where the process resides. *)
PROCEDURE MemoryLocation(process: ProcessID): MemoryblockID;

BEGIN
    WITH processtable[process].location DO
        IF resident THEN
            RETURN memory;
        ELSE
            RETURN MemoryblockID(NIL);
        END; (* IF *)
    END; (* WITH *)
END MemoryLocation;


(*  Remove the next process ID from the given list.  *)
PROCEDURE Next(VAR list: Priorityqueue): ProcessID;

    VAR
        process:     (* temporary holder for return value *)
            ProcessID;

BEGIN
    IF list↑[HEAD].next # TAIL THEN
        process := list↑[HEAD].next;
        list↑[HEAD].next := list↑[process].next;
        list↑[ list↑[HEAD].next ].previous := HEAD;
    ELSE
        process := NullProcess();
    END; (* IF *)

    RETURN process;
END Next;


(*  Return the Null Process ID. *)
PROCEDURE NullProcess(): ProcessID;

BEGIN
    RETURN(0);
END NullProcess;


(*  Return the list of diskblocks containing the output.   *)
PROCEDURE OutputList(process: ProcessID): Disklist;

BEGIN
    RETURN processtable[process].output.first;
END OutputList;
```

```
(*  Obtain the next process ID from the given list.    *)
PROCEDURE Peek(list: Priorityqueue): ProcessID;

BEGIN
    IF list↑[HEAD].next # TAIL THEN
        RETURN(list↑[HEAD].next);
    ELSE
        RETURN( NullProcess() );
    END; (* IF *)
END Peek;


(*  Return a pointer to the process's disk block. *)
PROCEDURE PermanentLocation(process: ProcessID): DiskblockID;

BEGIN
    RETURN processtable[process].location.diskblock;
END PermanentLocation;


(*  Obtain a process's priority. *)
PROCEDURE Priority(process: ProcessID): Prioritytype;

BEGIN
    RETURN(processtable[process].priority);
END Priority;


(*  Return a process's size in words.    *)
PROCEDURE ProcessSize(process: ProcessID): CARDINAL;

    VAR
        high,               (* high address in process *)
        low:                (* low address in process *)
            ADDRESS;

BEGIN
    ContextBounds(processtable[process].context, low, high);
    RETURN CARDINAL(high - low + 1);
END ProcessSize;


(*  Remove a process from a list.    *)
PROCEDURE Remove(process: ProcessID; VAR list: Priorityqueue);

    VAR
        previous,           (* index of previous process in list *)
        next:               (* index of next process in list *)
            ProcessID;
```

```
BEGIN
    IF ContainedInList(process, list) THEN
        previous := list↑[process].previous;
        next := list↑[process].next;
        list↑[previous].next := next;
        list↑[next].previous := previous;

        (* indicate that process is removed from list *)
        list↑[process].previous := HEAD;
    END; (* IF *)
END Remove;


(*  Determine if a process is memory resident.   *)
PROCEDURE Resident(process: ProcessID): BOOLEAN;

BEGIN
    RETURN(processtable[process].location.resident);
END Resident;


(*  Schedule an event.  *)
PROCEDURE Schedule(when: CARDINAL; act: Actiontype; pid: ProcessID);

    VAR
        current,                (* current item in the delta list *)
        previous:               (* previous item in the delta list *)
            Eventptr;

BEGIN
    previous := deltalist;
    current := deltalist;

    (* find correct place in list *)
    WHILE (current # NIL) & (current↑.timeleft < when) DO
        DEC(when, current↑.timeleft);
        previous := current;
        current := current↑.next;
    END; (* WHILE *)


    (* create node *)
    IF current = deltalist THEN
    (* list is empty, or new event should be placed first *)

        previoustime := TickCount();
        ALLOCATE( deltalist, SIZE(deltalist↑) );
        previous := deltalist;
    ELSE
        ALLOCATE( previous↑.next, SIZE(previous↑) );
        previous := previous↑.next;
    END; (* IF current *)

    (* insert event *)
    WITH previous↑ DO
        action := act;
        timeleft := when;
        process := pid;
        next := current;
    END; (* WITH previous *)

    (* alter next event's waiting time *)
    IF current # NIL THEN
        DEC(current↑.timeleft, when);
    END; (* IF *)
END Schedule;


(*  Obtain a process's status.   *)
PROCEDURE Status(process: ProcessID): Statustype;
BEGIN
    RETURN(processtable[process].status);
END Status;


(*  Store the next input location.     *)
PROCEDURE StoreNextInput(process: ProcessID; pageaddress: Pageptr; byte: CARDINAL);

BEGIN
    WITH processtable[process] DO
        nextinput.page := pageaddress;
        nextinput.byte := byte;
    END; (* WITH *)
END StoreNextInput;


(*  Store the next output location.    *)
PROCEDURE StoreNextOutput(process: ProcessID; pageaddress: Pageptr; byte: CARDINAL);

BEGIN
    WITH processtable[process].output.current DO
        IF byte = BYTESPERPAGE THEN
        (* page has been filled *)

            currentpage := (currentpage + 1) MOD BUFFERSIZE;
            buffer[currentpage].byte := 0;
            ELSE
                buffer[currentpage].byte := byte;
            END; (* IF *)
        END; (* WITH *)
    END StoreNextOutput;


    (*  Find the cause of a trap.      *)
    PROCEDURE TrapReason(process: ProcessID): AllTraps;

    BEGIN
        RETURN VirtualMachine.TrapReason(processtable[process].context);
    END TrapReason;


    (*  Unschedule a process's events.  *)
    PROCEDURE UnSchedule(process: ProcessID);

        VAR
            current,                (* current item in the delta list *)
            previous:               (* previous item in the delta list *)
                Eventptr;

    BEGIN

        IF (deltalist # NIL) & (deltalist↑.next # NIL) THEN
            previous := deltalist;
            current := previous↑.next;

            (* remove events within the list (not the first node) *)
            WHILE current # NIL DO
                IF current↑.process = process THEN
                    previous↑.next := current↑.next;
                    INC(current↑.next↑.timeleft, current↑.timeleft);
                    DEALLOCATE( current, SIZE(current↑) );
                ELSE
                    previous := current;
                END; (* IF *)
                current := previous↑.next;
            END; (* WHILE *)
        END; (* IF *)

        (* remove first node, if appropriate *)
        IF (deltalist # NIL) & (deltalist↑.process = process) THEN
            current := deltalist;
            deltalist := deltalist↑.next;
            INC(deltalist↑.timeleft, current↑.timeleft);
            DEALLOCATE( current, SIZE(current↑) );
        END (* IF *)

    END UnSchedule;
```

```
(*  Update a process control block. *)
PROCEDURE UpdateProcessInfo(process: ProcessID; reason: Updatecode;
    newlocationptr: ADDRESS);

    VAR
        oldstart,      (* starting address of old memoryblock *)
        newstart:      (* starting address of new memoryblock *)
            ADDRESS;
        offset:        (* the amount the process has moved *)
            INTEGER;

BEGIN
    CASE reason OF
        SWAPIN:
            WITH processtable[process] DO

                (* update process block; save old and new starting addreses *)
                WITH location DO
                    oldstart := StartingAddress(memory);
                    resident := TRUE;
                    memory := MemoryblockID(newlocationptr);
                    newstart := StartingAddress(memory);
                END; (* WITH location *)

                (* alter absolute addresses in context to reflect new location *)
                offset := newstart - oldstart;
                UpdateContext(context, offset);

            END; (* WITH processtable *)
    |   SWAPOUT:
            WITH processtable[process].location DO
                resident := FALSE;
                diskblock := DiskblockID(newlocationptr);
            END; (* WITH *)
    END; (* CASE *)
END UpdateProcessInfo;


BEGIN (* module initialization *)

    (* see procedure InitProcManager *)

END ProcessManager.
```

```
DEFINITION MODULE Spooler;

(***********************************************************************)
(*                                                                     *)
(*  FUNCTION:   This module is responsible for taking input from the card reader  *)
(*              and transfering it to the disk until it can be moved to main      *)
(*              memory. It does the reverse with output, sending it to disk       *)
(*              until it can be sent to the line printer.                         *)
(*                                                                     *)
(*  AUTHOR:     Rick Fisher                                            *)
(*                                                                     *)
(***********************************************************************)

    FROM SYSTEM IMPORT
        (* Constants *)
        BYTESPERWORD;

    FROM MemoryManager IMPORT
        (* Types *)
        MemoryblockID, Pageptr;

    FROM DiskManager IMPORT
        (* Types *)
        DiskblockID;

    FROM ProcessManager IMPORT
        (* Types *)
        Disklist;

    EXPORT QUALIFIED
        (* Constants *)
        ENDofINPUT, ENDofJOB, ENDofOUTPUT,

        (* Procedures *)
        DeSpool, EnSpool, PopJobSize;

    CONST
        ENDofINPUT    =  3C;    (* control-C *)
        ENDofJOB      =  3C;
        ENDofOUTPUT   =  3C;


    PROCEDURE DeSpool(): Pageptr;
    (*      Retrieve the next incoming page spooled on the disk, moving it to main   *)
    (*      memory.                                                                   *)
    (*                                                                                *)
    (*      RETURNS a pointer to the page retrieved--Pageptr(NIL) if spool is empty.  *)
    (*                                                                                *)
    (*      NOTES:                                                                    *)
    (*          1) A program's code, input and stack can all be assumed to start     *)
    (*      on page boundaries.                                                       *)
(*          2) The value returned points to a page in a memory buffer of fixed    *)
(*      size. Sometime following the next call to DeSpool, the page will be       *)
(*      overwritten. Therefore, the page should either be completely processed    *)
(*      or copied to a safe location before another call to DeSpool.             *)


PROCEDURE EnSpool(newoutput: Disklist);
(*      Send output of a finished job to spooling area of disk.                   *)
(*                                                                                *)
(*      PARAMETERS:     newoutput–a linked list of all the diskblocks holding     *)
(*                              output from a particular job.                     *)
(*                                                                                *)
(*      NOTES:                                                                    *)
(*          1) The last character of output must be immediately followed          *)
(*      by an ENDofOUTPUT character.                                              *)
(*          2) Each diskblock in the list should be a single sector in size.      *)


PROCEDURE PopJobSize(VAR codesize, inputsize: CARDINAL);
(*      Remove from the queue of job sizes the size in pages of the next          *)
(*      job's code and input.                                                     *)
(*                                                                                *)
(*      PARAMETERS:     codesize–the number of pages in the next job's code.      *)
(*                      inputsize–the number of pages in the next job's input.    *)

END Spooler.
```

```
IMPLEMENTATION MODULE Spooler(*[10]*);

(*************************************************************************)
(*                                                                       *)
(*  POLICY:       The Spooler reserves two cylinders of the disk during system     *)
(*                initialization for its own use.  One of these is used for data   *)
(*                being spooled in, the other for data being spooled out.  Having  *)
(*                all diskblocks for each of these functions on a single cylinder  *)
(*                ensures that disk reads and writes will be processed in the order *)
(*                in which they are requested.                           *)
(*                                                                       *)
(*  AUTHOR:       Rick Fisher                                            *)
(*                                                                       *)
(*************************************************************************)

   FROM SYSTEM IMPORT
      (* Constants *)
      BYTESPERWORD,

      (* Types *)
      ADDRESS, WORD,

      (* Procedures *)
      ADR, SIZE;

   FROM VirtualMachine IMPORT
      (* Constants *)
      BYTESPERPAGE,

      (* Types *)
      Interruptcode,

      (* Procedures *)
      Read, ReturnFromInterrupt, SetInterruptHandler, Write;

   IMPORT MemoryManager;
   FROM MemoryManager IMPORT
      (* Constants *)
      PAGESIZE,

      (* Types *)
      MemoryblockID, Pageindex, Pageptr,

      (* Procedures *)
      (* Allocate, Deallocate, *) StartingAddress;
```

```
IMPORT DiskManager;
FROM DiskManager IMPORT
    (* Constants *)
    CYLINDERSIZE, NULL, SECTORSIZE,

    (* Types *)
    DiskblockID,

    (* Procedures *)
    (* Allocate, Deallocate, *) DiskRead, DiskWrite, Null;

FROM ProcessManager IMPORT
    (* Types *)
    Disklist;

FROM STORAGE IMPORT
    (* Procedures *)
    ALLOCATE, DEALLOCATE;

CONST
    BUFFERSIZE      =   4;

TYPE
    Actiontype      =   (CONSUME, PRODUCE);
    Bufferindex     =   [0 .. BUFFERSIZE];
    Cylinderindex   =   [0 .. CYLINDERSIZE - 1];
    CircularDiskbuf =   RECORD
                            delayingwhileEmpty,     (* TRUE iff consuming procedure
                                                        cannot proceed while buffer is
                                                        empty *)
                            delayingwhileFull,      (* TRUE iff producing procedure
                                                        cannot proceed while buffer is
                                                        full *)
                            full:                   (* TRUE iff buffer is full *)
                                BOOLEAN;
                            next,                   (* next block to be filled *)
                            first:                  (* oldest filled block *)
                                Cylinderindex;
                            spool:                  (* the array of diskblocks *)
                                ARRAY Cylinderindex OF DiskblockID;
                        END;
```

```
        CircularMembuf    =  RECORD
                                 buffer:              (* the array of pages and memory-
                                                          blocks *)
                                     ARRAY [0 .. BUFFERSIZE - 1] OF
                                     RECORD
                                         page:        (* pointer to the array of characters *)
                                             Pageptr;
                                         index:       (* the index into ''page'' *)
                                             Pageindex;
                                         memoryblock: (* ID of memoryblock containing
                                                          ''page'' *)
                                             MemoryblockID;
                                     END;
                                 currentpage,         (* index into ''buffer'' *)
                                 nextIOrequest,       (* next page available for an IO
                                                          request *)
                                 waitingOnIO:         (* page waiting longest for an I/O
                                                          completion *)
                                     Bufferindex;
                                 waitingOndisk:       (* TRUE if all buffer pages await
                                                          disk completion *)
                                     BOOLEAN;
                                 restart:             (* driving producer (consumer) which
                                                          has been stopped because buffer was full
                                                          (empty) *)
                                     PROC;
                             END;
        Bufferptr         =  POINTER TO CircularMembuf;
        Pagecountlist     =  POINTER TO Pagecountrecord;
        Pagecountrecord   =  RECORD
                                 pagecount:  (* page count for a job *)
                                     CARDINAL;
                                 next:       (* rest of list *)
                                     Pagecountlist;
                             END;

    VAR
        endofOutput:         (* pointer to the last of the output blocks *)
            Disklist;
        head:                (* first in list of Pagecountrecords *)
            Pagecountlist;
        inputbuffer:         (* for card images read from card reader *)
            CircularMembuf;
        inspool:             (* disk area reserved for spooled input *)
            CircularDiskbuf;
        nextoutput,          (* pointer to the next output block to be written *)
        output:              (* pointer to the list of output blocks, including those written
                             but not yet deallocated *)
            Disklist;
    outputbuffer:           (* for line images going to line printer *)
        CircularMembuf;
    outspool:               (* disk area reserved for spooled output *)
        CircularDiskbuf;
    tail:                   (* last in list of Pagecountrecords *)
        Pagecountlist;
    transferbuffer:         (* for disk-to-disk transfers, moving output to the output spool *)
        CircularMembuf;
    transfersInprogress:    (* the number of incomplete read requests for the transfer buffer *)
        CARDINAL;
    virtualcardbuffer:      (* for card images being despooled *)
        CircularMembuf;
    waitingforoutput:       (* true if no output in output list or in transfer buffer *)
        BOOLEAN;


PROCEDURE Acknowledge(buffer: WORD);
(*      This procedure is passed to the disk manager as the action to be taken      *)
(*      upon completion of disk operations requested by several spoolling           *)
(*      procedures.                                                                 *)
(*                                                                                  *)
(*      PARAMETERS:     buffer–a pointer to the buffer holding the page             *)
(*                          being read or written (''buffer'' must be of            *)
(*                          type WORD only because the formal parameter             *)
(*                          to which ''Acknowledge'' is the actual takes            *)
(*                          a parameter of type WORD).                              *)

    VAR
        bufferpointer:  Bufferptr;

BEGIN
    bufferpointer := Bufferptr(buffer);
    WITH bufferpointer↑ DO

        (* oldest unready page becomes ready *)
        waitingOnIO := (waitingOnIO + 1) MOD BUFFERSIZE;

        (* if no more pages are waiting, set ''waitingOnIO'' to neutral value *)
        IF waitingOnIO = currentpage THEN
            waitingOnIO := BUFFERSIZE;
        END; (* IF *)

        (* buffer now has free space–the driving procedure can begin again *)
        IF waitingOndisk THEN
            waitingOndisk := FALSE;
            restart;
        END; (* IF *)
    END; (* WITH *)
    (* if appropriate SPOOLING AREA was completely full (empty), restart procedure
        which was waiting *)
    IF ( ADDRESS(buffer) = ADR(inputbuffer) ) & inspool.delayingwhileEmpty THEN
        inspool.delayingwhileEmpty := FALSE;
        virtualcardbuffer.restart;
    ELSIF ( ADDRESS(buffer) = ADR(outputbuffer) ) & outspool.delayingwhileFull THEN
        outspool.delayingwhileFull := FALSE;
        transferbuffer.restart;
    ELSIF ( ADDRESS(buffer) = ADR(transferbuffer) ) & outspool.delayingwhileEmpty
    THEN
        outspool.delayingwhileEmpty := FALSE;
        outputbuffer.restart;
    ELSIF ( ADDRESS(buffer) = ADR(virtualcardbuffer) ) & inspool.delayingwhileFull
    THEN
        inspool.delayingwhileFull := FALSE;
        inputbuffer.restart;
    END; (* IF *)
END Acknowledge;


(*  This is the actual interrupt handler for the cardreader.    *)
PROCEDURE CallSpoolIn;

BEGIN
    SpoolIn;
    ReturnFromInterrupt(CARDREADER);
END CallSpoolIn;


(*  This is the actual interrupt handler for the lineprinter.    *)
PROCEDURE CallSpoolOut;

BEGIN
    SpoolOut;
    ReturnFromInterrupt(LINEPRINTER);
END CallSpoolOut;


(*  Retrieve a page of spooled information, moving it to main memory.   *)
PROCEDURE DeSpool(): Pageptr;

BEGIN

    (* return nil if buffer and spool are both empty *)
    IF inspool.delayingwhileEmpty THEN
        RETURN Pageptr(NIL);
    END;

    (* spool is not empty, so fill buffer as much as possible *)
    FillVirtualCardBuffer;
    WITH virtualcardbuffer DO

        (* return nil if all buffer pages are waiting for a read to complete *)
        IF waitingOndisk THEN
            RETURN Pageptr(NIL);
        END;

        IF waitingOnIO = BUFFERSIZE THEN
        (* the next page to make an IO request will be the only one waiting for completion *)

            nextIOrequest := currentpage;
            waitingOnIO := currentpage;
        ELSIF nextIOrequest = BUFFERSIZE THEN
        (* another IO request can be made when current page advances *)

            nextIOrequest := currentpage;
        END;

        (* free the page last returned by this procedure (if not already free) *)
        IF currentpage # BUFFERSIZE THEN
            currentpage := (currentpage + 1) MOD BUFFERSIZE;
        ELSE
            currentpage := 0;
        END;

        IF currentpage = nextIOrequest THEN
        (* buffer completely empty--spool must be empty too *)

            nextIOrequest := 0;
            waitingOnIO := 0;
            currentpage := BUFFERSIZE;
            inspool.delayingwhileEmpty := TRUE;
            RETURN Pageptr(NIL);

        ELSIF currentpage = waitingOnIO THEN
        (* buffer not empty, but no pages are ready *)

            waitingOndisk := TRUE;
            RETURN Pageptr(NIL);

        ELSE
            RETURN buffer[currentpage].page;

        END; (* IF *)
    END; (* WITH *)
END DeSpool;
```

```
PROCEDURE EmptyTransferBuffer;
(*      Move pages of the transfer buffer to the output spool until the transfer      *)
(*      buffer is empty, or the spool is full.  It is only called when all of the     *)
(*      disk reads requested by FillTransferBuffer have completed.                    *)

    VAR
        alsowaiting:            (* next transfer buffer page to be ''waitingOnIO'' *)
            Bufferindex;
        bufferisfull:           (* true if transfer buffer is full *)
            BOOLEAN;
        currentDiskblock,       (* diskblock to be written to *)
        futureDiskblock:        (* diskblock to be written to subsequently *)
            DiskblockID;
        temp:                   (* for deallocating diskblocks and disklist nodes *)
            Disklist;

BEGIN

    (* deallocate old diskblocks *)
    WHILE output # nextoutput DO
        temp := output;
        output := output↑.next;
        DiskManager.Deallocate(temp↑.diskblockptr);
        DEALLOCATE( temp, SIZE(temp↑) );
    END; (* WHILE *)

    WITH transferbuffer DO
        IF waitingOnIO = nextIOrequest THEN
        (* buffer is empty–no need for a disk block *)

            currentDiskblock := DiskblockID(NIL);

        ELSE
            currentDiskblock := NextDiskBlock(outspool, PRODUCE);
        END;

        bufferisfull := (nextIOrequest = BUFFERSIZE);
        IF ADDRESS(currentDiskblock) = NIL THEN
        (* spool is full or buffer is empty *)

            outspool.delayingwhileFull := bufferisfull;
        ELSE

            (* write from buffer to spool *)
            LOOP

                (* test if another buffer page can also be written to the spool *)
                alsowaiting := (waitingOnIO + 1) MOD BUFFERSIZE;
                IF alsowaiting = nextIOrequest THEN
                    futureDiskblock := DiskblockID(NIL);
                ELSE
                    futureDiskblock := NextDiskBlock(outspool, PRODUCE);
                END; (* IF alsowaiting *)

                (* do not write the last writable page yet *)
                IF ADDRESS(futureDiskblock) = NIL THEN
                    EXIT; (* LOOP *)
                ELSE
                    DiskWrite(buffer[waitingOnIO].memoryblock, currentDiskblock,
                        Null, NULL);
                    currentDiskblock := futureDiskblock;
                    waitingOnIO := alsowaiting;
                END; (* IF futureDiskblock *)
            END; (* LOOP *)

            (* write last page to disk, requesting acknowledgment *)
            DiskWrite( buffer[waitingOnIO].memoryblock, currentDiskblock,
                Acknowledge, ADR(transferbuffer) );
        END; (* IF currentDiskblock *)
    END; (* WITH *)
END EmptyTransferBuffer;


(*  Attach the output of another job to previous output.  *)
PROCEDURE EnSpool(newoutput: Disklist);

BEGIN
    IF newoutput # NIL THEN

        (* attach new output to old output *)
        IF output = NIL THEN
            output := newoutput;
        ELSE
            endofOutput↑.next := newoutput;
        END; (* IF output *)
        IF nextoutput = NIL THEN
            nextoutput := newoutput;
        END; (* IF nextoutput *)

        (* find the new last block of output *)
        endofOutput := newoutput;
        WHILE endofOutput↑.next # NIL DO
            endofOutput := endofOutput↑.next;
        END; (* WHILE *)
        IF waitingforoutput THEN
        (* "FillTransferBuffer" was waiting for new output *)

            waitingforoutput := FALSE;
            FillTransferBuffer;
        END; (* IF waitingforoutput *)

    END; (* WITH *)
END EnSpool;


PROCEDURE FillTransferBuffer;
(*      Move blocks of output to the transfer buffer until the transfer buffer      *)
(*      is full, or the output is finished.                                         *)

    VAR
        bufferisempty:      (* true if buffer is empty *)
            BOOLEAN;

BEGIN
    WITH transferbuffer DO
        bufferisempty := (waitingOnIO = nextIOrequest);
        IF waitingOndisk OR (output = NIL) THEN
        (* can't write to buffer--buffer is full, or no more output *)

            waitingforoutput := bufferisempty;
            EmptyTransferBuffer;
        ELSE

            (* read as many output blocks into the buffer as possible *)
            WHILE (nextIOrequest # waitingOnIO) & (nextoutput # NIL) DO
                DiskRead(nextoutput↑.diskblockptr, buffer[nextIOrequest].memoryblock,
                    TransferComplete, NULL);
                nextIOrequest := (nextIOrequest + 1) MOD BUFFERSIZE;
                nextoutput := nextoutput↑.next;
                INC(transfersInprogress);
            END; (* WHILE *)

            IF nextIOrequest = waitingOnIO THEN
            (* no more pages can be written into at this time *)

                nextIOrequest := BUFFERSIZE;
            END; (* IF nextIOrequest *)
        END; (* IF waitingOndisk *)
    END; (* WITH *)
END FillTransferBuffer;
```

```
PROCEDURE FillVirtualCardBuffer;
(*      Move blocks of input to the virtual card buffer until the virtual card          *)
(*      buffer is full, or the input spool is empty.                                    *)

    VAR
        diskblock:      (* the diskblock being read *)
            DiskblockID;

BEGIN
    WITH virtualcardbuffer DO

        (* fill the buffer as full as possible *)
        LOOP
            IF nextIOrequest # BUFFERSIZE THEN
            (* buffer is not full *)

                diskblock := NextDiskBlock(inspool, CONSUME);
            ELSE
                diskblock := DiskblockID(NIL);
            END;

            IF ADDRESS(diskblock) = NIL THEN
            (* buffer is full or spool is empty *)

                EXIT; (* LOOP *)
            ELSE

                (* read a block and advance next IO request *)
                DiskRead( diskblock, buffer[nextIOrequest].memoryblock, Acknowledge,
                    ADR(virtualcardbuffer) );
                nextIOrequest := (nextIOrequest + 1) MOD BUFFERSIZE;
                IF (nextIOrequest = currentpage) OR (nextIOrequest = waitingOnIO) THEN
                (* no more pages are available for IO *)

                    nextIOrequest := BUFFERSIZE;
                END; (* IF nextIOrequest *)

            END; (* IF diskblock *)
        END; (* LOOP *)
    END; (* WITH *)
END FillVirtualCardBuffer;
```

```
PROCEDURE InitDiskBuffer(VAR buf: CircularDiskbuf);
(*      Initialize a circular disk buffer (spool).  All spools start out in the same     *)
(*      condition–empty, with any consumers from the spool currently delayed.            *)
(*                                                                                        *)
(*      PARAMETERS:     buf–the spool being initialized.                                 *)
(*                                                                                        *)
(*      NOTE: This procedure will allocate all diskblocks for a spool from a             *)
(*          single cylinder if and only if (1) the first available sector on the         *)
(*          disk is the first sector of a cylinder, and (2) every other sector on        *)
(*          that cylinder is also available.  This situation is only guaranteed at      *)
(*          system initialization.                                                       *)

    VAR
        i:   CARDINAL;   (* loop index *)

BEGIN
    WITH buf DO
        full := FALSE;
        delayingwhileEmpty := TRUE;
        delayingwhileFull := FALSE;
        next := 0;
        first := 0;
        FOR i := 0 TO CYLINDERSIZE - 1 DO
            spool[i] := DiskManager.Allocate(SECTORSIZE);
        END;
    END; (* WITH *)
END InitDiskBuffer;


PROCEDURE InitMemBuffer(VAR buf: CircularMembuf; current, next, waiting: Bufferindex;
        start: PROC);
(*      Initialize a circular memory buffer.                                             *)
(*                                                                                        *)
(*      PARAMETERS:     buf–the buffer to be initialized.                                *)
(*                      current–the starting value for ''currentpage''.                  *)
(*                      next–the starting value for ''nextIOrequest''.                   *)
(*                      waiting–the starting value for ''waitingOnIO''.                  *)
(*                      start–the procedure value for ''restart''.                       *)

    VAR i: CARDINAL;   (* loop index *)

BEGIN
    WITH buf DO
        waitingOndisk := FALSE;
        currentpage := current;
        nextIOrequest := next;
        waitingOnIO := waiting;
        restart := start;
        (* associate memory blocks with page buffers *)
        FOR i := 0 TO BUFFERSIZE - 1 DO
            WITH buffer[i] DO
                memoryblock := MemoryManager.Allocate(PAGESIZE);
                page := Pageptr( StartingAddress(memoryblock) );
                index := 0;
            END; (* WITH *)
        END; (* FOR *)
    END; (* WITH *)
END InitMemBuffer;


PROCEDURE NextDiskBlock(spool: CircularDiskbuf; action: Actiontype): DiskblockID;
(*      Identify the next disk block available for the given action.                *)
(*                                                                                  *)
(*      PARAMETERS:     spool–specifies "inspool" or "outspool".                    *)
(*                      action–CONSUME if reading from the spool;                   *)
(*                              PRODUCE if writing to it.                           *)
(*      RETURNS the disk block ID of the desired disk block.  DiskblockID(NIL)      *)
(*          is returned if no block is ready.                                       *)

    VAR
        diskblock:  DiskblockID;   (* ID of the diskblock to be returned *)

BEGIN

    (* assume failure initially *)
    diskblock := DiskblockID(NIL);

    WITH spool DO
        CASE action OF
            CONSUME:
                IF (first # next) OR full THEN

                    (* spool not empty–calling procedure may consume *)
                    diskblock := spool[first];
                    first := (first + 1) MOD CYLINDERSIZE;
                    full := FALSE;
                END; (* IF first *)
        | PRODUCE:
            IF NOT full THEN

                (* calling procedure may produce *)
                diskblock := spool[next];
                next := (next + 1) MOD CYLINDERSIZE;
                IF next = first THEN
                    full := TRUE;
                END; (* IF next*)
            END; (* IF NOT *)
        END; (* CASE *)
    END; (* WITH *)

    RETURN diskblock;

END NextDiskBlock;


(*  Find the size of the next job.      *)
PROCEDURE PopJobSize(VAR codesize, inputsize: CARDINAL);

    VAR
        temp:   (* used for deallocating old list nodes *)
            Pagecountlist;

BEGIN
    IF head = NIL THEN
        codesize := 0;
        inputsize := 0;
    ELSE
        codesize := head↑.pagecount;
        temp := head;
        head := head↑.next;
        DEALLOCATE( temp, SIZE(temp↑) );
        inputsize := head↑.pagecount;
        temp := head;
        head := head↑.next;
        DEALLOCATE( temp, SIZE(temp↑) );
    END; (* IF *)
END PopJobSize;


VAR
    pagecount:                  (* number of pages in a given job *)
        CARDINAL;
```

```
PROCEDURE SpoolIn;
(*      Spool card images from cardreader to disk. This procedure serves       *)
(*      as the body of an interrupt handler for the card reader. The card reader  *)
(*      must be started by the main module during system initialization.      *)

    VAR
        bufferisfull:   (* true when input buffer is full *)
            BOOLEAN;
        diskblock:      (* the next diskblock to be written *)
            DiskblockID;

BEGIN
    WITH inputbuffer DO
        IF currentpage = waitingOnIO THEN
        (* can't read a character–buffer is full *)

            waitingOndisk := TRUE;
        ELSE
        WITH buffer[currentpage] DO

                (* read next character and increment index *)
                page↑[index] := Read();
                IF (page↑[index] # ENDofJOB) & (page↑[index] # ENDofINPUT) THEN
                    index := (index + 1) MOD BYTESPERPAGE;
                ELSE

                    (* code or input finished–reset index and record size in list *)
                    index := 0;
                    IF head = NIL THEN

                        (* start new list of job sizes *)
                        ALLOCATE( head, SIZE(head↑) );
                        tail := head;
                    ELSE

                        (* add new element to list *)
                        ALLOCATE( tail↑.next, SIZE(head↑) );
                        tail := tail↑.next;
                    END;

                    tail↑.next := NIL;
                    tail↑.pagecount := pagecount;
                    pagecount := 0;
                END; (* IF page *)
            END; (* WITH buffer[currentpage] *)
            (* page full? if so, move to next page *)
            IF buffer[currentpage].index = 0 THEN

                IF waitingOnIO = BUFFERSIZE THEN
                (* the next page to make an IO request will be the only one waiting for
                    completion *)

                    nextIOrequest := currentpage;
                    waitingOnIO := currentpage;
                ELSIF nextIOrequest = BUFFERSIZE THEN
                (* another IO request can be made when current page advances *)

                    nextIOrequest := currentpage;
                END; (* IF waitingOnIO *)

                currentpage := (currentpage + 1) MOD BUFFERSIZE;
                INC(pagecount);
            END; (* IF index = 0 *)
        END; (* IF currentpage *)


        (* if buffer has pages filled but not yet written, write one *)
        IF nextIOrequest # BUFFERSIZE THEN
            bufferisfull := (currentpage = nextIOrequest);
            diskblock := NextDiskBlock(inspool, PRODUCE);

            (* write a page if the spool is not full *)
            IF ADDRESS(diskblock) = NIL THEN
                inspool.delayingwhileFull := bufferisfull;
            ELSE
                DiskWrite( buffer[nextIOrequest].memoryblock, diskblock, Acknowledge,
                    ADR(inputbuffer) );
                nextIOrequest := (nextIOrequest + 1) MOD BUFFERSIZE;
                IF nextIOrequest = currentpage THEN
                    nextIOrequest := BUFFERSIZE;
                END; (* IF nextIOrequest = currentpage *)
            END; (* IF diskblock *)
        END; (* IF nextIOrequest # BUFFERSIZE *)
    END; (* WITH inputbuffer *)
END SpoolIn;
```

```
PROCEDURE SpoolOut;
(*      Spool line images from disk to the lineprinter.  This procedure serves     *)
(*      as the body of an interrupt handler for the line printer. The line printer  *)
(*      is started when output is sent to the spooling disk.                        *)

    VAR
        bufferisempty:      (* true when output buffer is empty *)
            BOOLEAN;
        diskblock:          (* the next diskblock to be read *)
            DiskblockID;

BEGIN
    WITH outputbuffer DO
        WITH buffer[currentpage] DO

            IF currentpage = waitingOnIO THEN
            (* can't write a character–buffer is empty *)

                waitingOndisk := TRUE;
            ELSE

                (* write next character and increment index *)
                Write(page↑[index]);
                IF page↑[index] = ENDofOUTPUT THEN
                    index := 0;
                ELSE
                    index := (index + 1) MOD BYTESPERPAGE;
                END; (* IF page *)

                (* page emptied? if so, move to next page *)
                IF index = 0 THEN

                    IF waitingOnIO = BUFFERSIZE THEN
                    (* the next page to make an IO request will be the only one waiting
                        for completion *)

                        nextIOrequest := currentpage;
                        waitingOnIO := currentpage;
                    ELSIF nextIOrequest = BUFFERSIZE THEN
                    (* another IO request can be made when current page advances *)

                        nextIOrequest := currentpage;
                    END; (* IF waitingOnIO *)

                    currentpage := (currentpage + 1) MOD BUFFERSIZE;

                END; (* IF index = 0 *)
```

```
                    (* read another page into buffer, if possible *)
                    IF nextIOrequest # BUFFERSIZE THEN
                        bufferisempty := (currentpage = nextIOrequest);
                        diskblock := NextDiskBlock(outspool, CONSUME);
                        IF ADDRESS(diskblock) = NIL THEN
                            outspool.delayingwhileEmpty := bufferisempty;
                        ELSE
                            DiskRead( diskblock, buffer[nextIOrequest].memoryblock, Acknowledge,
                                    ADR(outputbuffer) );
                            nextIOrequest := (nextIOrequest + 1) MOD BUFFERSIZE;
                            IF nextIOrequest = currentpage THEN
                                nextIOrequest := BUFFERSIZE;
                            END; (* IF nextIOrequest = currentpage *)
                        END; (* IF diskblock *)
                    END; (* IF nextIOrequest # BUFFERSIZE *)
                END; (* IF currentpage *)
            END; (* WITH buffer *)
        END; (* WITH outputbuffer *)
    END SpoolOut;


    PROCEDURE TransferComplete(dummy: WORD);
    (*      Acknowledgement procedure for ''FillTransferBuffer'' Since there          *)
    (*      is no way to tell in advance what order the requested reads into transfer  *)
    (*      buffer will be completed, the most straight-forward course is to count the *)
    (*      reads. When all have completed, it is safe to empty the buffer's pages    *)
    (*      in order.                                                                  *)
    (*                                                                                 *)
    (*      PARAMETERS:     dummy–unused parameter, included for formal reasons.       *)

    BEGIN
        DEC(transfersInprogress);
        IF transfersInprogress = 0 THEN
            EmptyTransferBuffer;
        END; (* IF *)
    END TransferComplete;


BEGIN(* module initialization *)

    SetInterruptHandler(CARDREADER, CallSpoolIn);
    SetInterruptHandler(LINEPRINTER, CallSpoolOut);

    (* initialize memory buffers *)
    InitMemBuffer(inputbuffer, 0, BUFFERSIZE, BUFFERSIZE, SpoolIn);
    InitMemBuffer(outputbuffer, 0, 0, 0, SpoolOut);
    InitMemBuffer(virtualcardbuffer, BUFFERSIZE, 0, 0, FillVirtualCardBuffer);
    InitMemBuffer(transferbuffer, BUFFERSIZE, 0, 0, FillTransferBuffer);
    (* initialize spools *)
    InitDiskBuffer(inspool);
    InitDiskBuffer(outspool);

    (* initialize pagecount list *)
    head := NIL;
    tail := NIL;
    pagecount := 1;

    (* initialize output list *)
    endofOutput := NIL;
    output := NIL;
    waitingforoutput := TRUE;

END Spooler.
```

```
DEFINITION MODULE SVCalls;

(*************************************************************************)
(*                                                                       *)
(*  FUNCTION:   This module defines the procedures and types relating to Supervisor  *)
(*              Calls. It is the only OS module through which a user can get into    *)
(*              the operating system.                                    *)
(*                                                                       *)
(*  AUTHOR:     Rick Fisher                                              *)
(*                                                                       *)
(*************************************************************************)

    FROM SYSTEM IMPORT
        (* Types *)
        ADDRESS;

    EXPORT QUALIFIED
        (* Types *)
        DisksectorID,

        (* Procedures *)
        FreeDiskSector, GetDiskSector, Read, UpperBound, Write;

    TYPE
        DisksectorID;

    PROCEDURE FreeDiskSector(VAR sector: DisksectorID);
    (*      Free a no longer needed disk sector.                             *)
    (*                                                                       *)
    (*      PARAMETERS:     sector–the ID of the disk sector.                *)


    PROCEDURE GetDiskSector(VAR sector: DisksectorID);
    (*      Obtain the sector ID of a free disk block.                       *)
    (*                                                                       *)
    (*      PARAMETERS:     sector–the ID of the disk sector.                *)


    PROCEDURE Read(VAR buffer: ARRAY OF CHAR);
    (*      Read from input.                                                 *)
    (*                                                                       *)
    (*      PARAMETERS:     buffer–the variable to hold the array of characters  *)
    (*                          to be read. The size of the buffer determines the  *)
    (*                          number of characters read.                   *)


    PROCEDURE Sleep(seconds: CARDINAL);
    (*      Halt the execution of a process for the given length of time.    *)
    (*                                                                       *)
    (*      PARAMETERS:     seconds–the number of seconds that should pass before  *)
    (*                          the process again becomes eligible for execution.  *)
    PROCEDURE UpperBound(): ADDRESS;
    (*      Find the upper bound in memory to which the process has access.          *)
    (*                                                                                *)
    (*      RETURNS the address of the upper bound.                                   *)


    PROCEDURE Write(buffer: ARRAY OF CHAR);
    (*      Write to output.  Arrays of more than 128 characters will be truncated    *)
    (*      to that length.                                                           *)
    (*                                                                                *)
    (*      PARAMETERS:     buffer–the array of characters to be written.             *)

END SVCalls.
```

```
IMPLEMENTATION MODULE SVCalls;

(**************************************************************************)
(*                                                                        *)
(*  POLICY:        Limit the user's access to the Operating System to those procedures  *)
(*                 which may be reached through a SVC trap.               *)
(*                                                                        *)
(*  NOTE:          The order of the parameters in the exported procedures must be  *)
(*                      maintained to reflect the order in which the "SVCArguments"  *)
(*                      procedure of "VirtualMachine" expects them (enumerated  *)
(*                      in that procedure's definition module comment).   *)
(*                                                                        *)
(*  AUTHOR:        Rick Fisher                                            *)
(*                                                                        *)
(**************************************************************************)

    FROM SYSTEM IMPORT
        (* Types *)
        ADDRESS, WORD;

    FROM VirtualMachine IMPORT
        (* Types *)
        OSTraps, SVCcode,

        (* Procedures *)
        Trap;


    TYPE
        DisksectorID    =   CARDINAL;


    (*  Free a no longer needed disk sector.      *)
    PROCEDURE FreeDiskSector(VAR sector: DisksectorID);

    BEGIN
        TwoParmSVC1(DISKDISPOSESVC, sector);
    END FreeDiskSector;


    (*  Obtain the sector ID of a free disk block. *)
    PROCEDURE GetDiskSector(VAR sector: DisksectorID);

    BEGIN
        TwoParmSVC1(DISKALLOCATESVC, sector);
    END GetDiskSector;
```

```
(*  Read from input.    *)
PROCEDURE Read(VAR buffer: ARRAY OF CHAR);

BEGIN
    TwoParmSVC2(READSVC, buffer);
END Read;


(*  Sleep a while.  *)
PROCEDURE Sleep(seconds: CARDINAL);

BEGIN
    TwoParmSVC1(SLEEPSVC, seconds);
END Sleep;


(*  Find upper bound.  *)
PROCEDURE UpperBound(): ADDRESS;

    VAR
        address:            (* the upper bound *)
            ADDRESS;

BEGIN
    TwoParmSVC1(UPPERBOUNDSVC, address);
    RETURN address;
END UpperBound;


(*  Write to output. *)
PROCEDURE Write(buffer: ARRAY OF CHAR);

BEGIN
    TwoParmSVC2(WRITESVC, buffer);
END Write;


(*      The following two procedures are identical in form and function, differing    *)
(*      only in the types of parameters they accept. Their purpose is to cause a       *)
(*      trap to the correct supervisor call, at the same time fixing the positions     *)
(*      of the necessary parameters in the activation record stack.                     *)

PROCEDURE TwoParmSVC1(svc: SVCcode; VAR word: WORD);

BEGIN
    Trap(SVC);
END TwoParmSVC1;
```

```
    PROCEDURE TwoParmSVC2(svc: SVCcode; VAR buffer: ARRAY OF CHAR);

    BEGIN
        Trap(SVC);
    END TwoParmSVC2;

BEGIN
END SVCalls.
```

```
DEFINITION MODULE Swapper;

(*************************************************************************)
(*                                                                       *)
(*  FUNCTION:   The swapper copies process images from main storage to disk and  *)
(*              vice-versa, freeing memory space when a process is swapped out.  *)
(*                                                                       *)
(*  AUTHOR:     Rick Fisher                                              *)
(*                                                                       *)
(*************************************************************************)

   FROM DiskManager IMPORT
      (* Types *)
      DiskblockID;

   FROM ProcessManager IMPORT
      (* Types *)
      ProcessID;

   EXPORT QUALIFIED
      (* Procedures *)
      SwapIn, SwapOut;


   PROCEDURE SwapIn(process: ProcessID): BOOLEAN;
   (*      Move the process image from disk to memory, updating process information   *)
   (*      (other than status, which should be set according to the result of SwapIn).  *)
   (*      Processes should only be swapped in if they can become READY or            *)
   (*      PENDING.                                                                    *)
   (*                                                                                  *)
   (*      PARAMETERS:     process–the process being swapped in.                       *)
   (*      RETURNS TRUE if successful, FALSE otherwise.                                *)


   PROCEDURE SwapOut(process: ProcessID);
   (*      Move the process image from memory to disk, updating process information   *)
   (*      (other than status, which should be set before the call).                  *)
   (*                                                                                  *)
   (*      PARAMETERS:     process–the process being swapped out.                      *)

END Swapper.
```

```
IMPLEMENTATION MODULE Swapper;

(****************************************************************************)
(*                                                                          *)
(*  POLICY:      The swapper allocates new memory space for processes being *)
(*               swapped in, reads (or writes) the process to (from) disk, and then *)
(*               alters the Process Control Block to reflect the new location. *)
(*                                                                          *)
(*  AUTHOR:      Rick Fisher                                                *)
(*                                                                          *)
(****************************************************************************)

  FROM SYSTEM IMPORT
    (* Types *)
    ADDRESS, WORD;

  IMPORT DiskManager;
  FROM DiskManager IMPORT
    (* Types *)
    DiskblockID,

    (* Procedures *)
    (* Allocate, Deallocate, *) DiskRead, DiskWrite;

  IMPORT MemoryManager;
  FROM MemoryManager IMPORT
    (* Types *)
    MemoryblockID;

    (* Procedures *)
    (* Allocate, Deallocate *)

  FROM ProcessManager IMPORT
    (* Types *)
    ProcessID, Updatecode, Statustype, Actiontype,

    (* Procedures *)
    ChangeStatus, MemoryLocation, NullProcess, PermanentLocation, ProcessSize,
    Resident, Schedule, UpdateProcessInfo;


  (*  Move the process image from disk to memory.    *)
  PROCEDURE SwapIn(process: ProcessID): BOOLEAN;

    VAR
      currentlocation:    (* location of process before swap *)
        DiskblockID;
      memoryblock:        (* location of process after swap *)
        MemoryblockID;
      size:               (* size of process in words *)
        CARDINAL;
```

```
BEGIN

    IF NOT Resident(process) THEN

        (* allocate new memory block *)
        size := ProcessSize(process);
        memoryblock := MemoryManager.Allocate(size);

        IF ADDRESS(memoryblock) = NIL THEN
            RETURN FALSE;
        ELSE
            UpdateProcessInfo(process, SWAPIN, ADDRESS(memoryblock) );
            currentlocation := DiskblockID( PermanentLocation(process) );
            DiskRead(currentlocation, memoryblock, SwapInComplete, process);
        END; (* IF memoryblock *)

    END; (* IF NOT *)

    RETURN TRUE;
END SwapIn;


PROCEDURE SwapInComplete(process: WORD);
(*      Acknowledge that a process has been swapped in. This procedure is          *)
(*      passed to the DiskManager to acknowledge when a swap-in has completed.     *)
(*                                                                                 *)
(*      PARAMETERS:     process--the process which has been swapped in.            *)

BEGIN
    Schedule( 0, RESCHEDULE, ProcessID(process) );
END SwapInComplete;


(*  Move the process from memory to disk.  *)
PROCEDURE SwapOut(process: ProcessID);

    VAR
        currentlocation:    (* location of process before swap *)
            MemoryblockID;
        diskblock:          (* location of process after swap *)
            DiskblockID;
    BEGIN

        IF Resident(process) THEN
            diskblock := DiskblockID( PermanentLocation(process) );
            IF ADDRESS(diskblock) = NIL THEN
                diskblock := DiskManager.Allocate( ProcessSize(process) );
                IF ADDRESS(diskblock) = NIL THEN
                (* no room on disk for process *)

                    RETURN;
                END; (* IF *)
            END; (* IF *)

            UpdateProcessInfo(process, SWAPOUT, ADDRESS(diskblock) );
            currentlocation := MemoryblockID( MemoryLocation(process) );
            DiskWrite(currentlocation, diskblock, SwapOutComplete, currentlocation);
        END; (* IF *)

    END SwapOut;


    PROCEDURE SwapOutComplete(currentlocation: WORD);
    (*      Acknowledge that a process has been swapped out. This procedure is    *)
    (*      passed to the DiskManager as acknowledgment that a swap out has       *)
    (*      completed.                                                            *)
    (*                                                                            *)
    (*      PARAMETERS:     currentlocation–the memory block to be deallocated.   *)

        VAR
            temp: MemoryblockID;

    BEGIN
        temp := MemoryblockID(currentlocation);
        MemoryManager.Deallocate(temp);
    END SwapOutComplete;

BEGIN
END Swapper.
```

```
DEFINITION MODULE TrapHandler;

(**********************************************************************)
(*                                                                    *)
(*  FUNCTION:    The Trap Handler responds to the raising of any TRAP interrupt.   *)
(*               As it responds only to the interrupt signal, it has no exported   *)
(*               procedures.                                          *)
(*                                                                    *)
(*  AUTHOR:      Rick Fisher                                          *)
(*                                                                    *)
(**********************************************************************)

END TrapHandler.
```

```
IMPLEMENTATION MODULE TrapHandler;

(************************************************************************)
(*                                                                      *)
(*  POLICY:     All error traps cause a message to be printed, and the termination  *)
(*              of the program.                                         *)
(*                                                                      *)
(*  AUTHOR:     Rick Fisher                                             *)
(*                                                                      *)
(************************************************************************)

    FROM SYSTEM IMPORT
        (* Constants *)
        BYTESPERWORD,

        (* Types *)
        ADDRESS, WORD,

        (* Procedures *)
        ADR, SIZE;

    FROM VirtualMachine IMPORT
        (* Constants *)
        BYTESPERPAGE,

        (* Types *)
        AllTraps, Interruptcode, SVCcode,

        (* Procedures *)
        ContextBounds, InitOSContext, LowerStackLimit, SetInterruptHandler, SVCArguments;

    FROM MemoryManager IMPORT
        (* Constants *)
        PAGESIZE,

        (* Types *)
        MemoryblockID, Page, Pageptr;

    FROM DiskManager IMPORT
        (* Constants *)
        SECTORSIZE,

        (* Types *)
        DiskblockID,

        (* Procedures *)
        Allocate, Deallocate, DiskWrite, Null;

    FROM Clock IMPORT
        (* Constants *)
        TICKSPERSECOND;
```

```
FROM ProcessManager IMPORT
    (* Types *)
    Actiontype, ProcessID, Statustype,

    (* Procedures *)
    ChangeStatus, CheckSchedule, Context, Equal, GetNextInput, GetNextOutput,
    LinkToOutput, OutputList, Schedule, StoreNextInput, StoreNextOutput, TrapReason;

FROM Spooler IMPORT
    (* Constants *)
    ENDofINPUT, ENDofOUTPUT,

    (* Procedures *)
    EnSpool;

FROM LowLevelScheduler IMPORT
    (* Procedures *)
    Block, CurrentContext, CurrentProcess, Sleep, TimeOut;

FROM MediumScheduler IMPORT
    (* Procedures *)
    Reschedule;

FROM HighLevelScheduler IMPORT
    (* Procedures *)
    ShoulderTap, Terminate;

FROM STORAGE IMPORT
    (* Procedures *)
    ALLOCATE, DEALLOCATE;


(*      Check the list of scheduled events, and take the appropriate action.          *)
PROCEDURE ExecuteScheduledEvents;

    VAR
        action:         (* the action to be taken *)
            Actiontype;
        process:        (* process on which the action is to be taken *)
            ProcessID;

BEGIN

    REPEAT  (* UNTIL NULL *)
        CheckSchedule(action, process);

        (* take appropriate action *)
        CASE action OF
            NULL:  ;
        |   TIMEDPREEMPT:
                IF Equal( process, CurrentProcess() ) THEN
                    TimeOut;
                END; (* IF *)
        |   RESCHEDULE:
                Reschedule(process);
        |   TERMINATE:
                FlushOutputBuffer(process);
                Terminate(process);
        |   WAKEUP:
                Wakeup(process);
        END; (* CASE *)

    UNTIL action = NULL;
END ExecuteScheduledEvents;


PROCEDURE FlushOutputBuffer(process: ProcessID);
(*      This procedure adds the ENDofOUTPUT character to a process's        *)
(*      current output page, then adds that (partial) page to the list of output *)
(*      blocks for the process, and EnSpools the entire list.               *)
(*                                                                          *)
(*      PARAMETERS:     process-the process whose output is to be flushed. *)

    VAR
        byte:           (* next byte for output on current page *)
            CARDINAL;
        diskblock:      (* the block to which the page is written *)
            DiskblockID;
        memoryblock:    (* the memoryblock in which the page lies *)
            MemoryblockID;
        outputpage:     (* the current output buffer page *)
            Page;
        pageptr:        (* pointer to the current page *)
            Pageptr;

BEGIN

    (* punctuate final output page *)
    GetNextOutput(process, pageptr, byte, memoryblock);
    outputpage := pageptr↑;
    outputpage[byte] := ENDofOUTPUT;

    (* add page to output list *)
    diskblock := Allocate(PAGESIZE);
    DiskWrite(memoryblock, diskblock, Null, NIL);
    LinkToOutput(process, diskblock);

    EnSpool( OutputList(process) );

END FlushOutputBuffer;
```

```
CONST
    MAXARGS         =   2;

TYPE
    Argumentlist    =   ARRAY [0 .. MAXARGS] OF WORD;
    Arglistptr      =   POINTER TO Argumentlist;

(*      Find which SVC caused the trap and act accordingly.                     *)
PROCEDURE ProcessSVC;

    VAR
        address:            (* for allocating space *)
            ADDRESS;
        argptr:             (* address of arguments to SVC call *)
            Arglistptr;
        argument:           (* the arguments to the SVC call *)
            Argumentlist;
        diskblock:          (* for deallocating diskblocks *)
            DiskblockID;
        high,
        low:                (* context bounds *)
            ADDRESS;

BEGIN
    argptr := Arglistptr( SVCArguments( CurrentContext() ) );
    argument := argptr↑;

    (* act according to the correct SVC *)
    CASE SVCcode(argument[0]) OF
        DISKALLOCATESVC:
            argument[1] := WORD( Allocate(SECTORSIZE) );
    |   DISKDISPOSESVC:
            diskblock := DiskblockID(argument[1]);
            Deallocate(diskblock);
    |   HEAPALLOCATESVC:
            address := ADDRESS(argument[2]);
            IF LowerStackLimit( CurrentContext(), CARDINAL(argument[1]) ) THEN
                address↑ := WORD(TRUE);
            ELSE
                address↑ := WORD(FALSE);
            END; (* IF *)
    |   READSVC:
            ReadFromInput( CurrentProcess(), ADDRESS(argument[1]), CARDINAL(argument[2]) );
    |   SLEEPSVC:
            address := ADDRESS(argument[1]);
            Sleep( TICKSPERSECOND * CARDINAL(address↑) );
    |   UPPERBOUNDSVC:
            address := ADDRESS(argument[1]);
            ContextBounds(CurrentContext(), low, high);
            address↑ := WORD(high);
    |   WRITESVC:
            WriteToOutput( CurrentProcess(), ADDRESS(argument[1]), CARDINAL(argument[2]) );
    END; (* CASE *)
END ProcessSVC;


PROCEDURE ProcessTrap;
(*      Choose the action to be taken on each trap. All error traps, and Halt,     *)
(*      are handled directly; CHECKSCHEDULE, INITIALIZE, SHOULDERTAP               *)
(*      and SVC traps are given to other procedures.                               *)

    VAR
        process:            (* the current process *)
            ProcessID;

BEGIN
    process := CurrentProcess();
    CASE TrapReason(process) OF
        BADINSTRUCTION:
            WriteLiteral(process, 'Reported error BADINSTRUCTION currently ');
            WriteLiteral(process, 'inapplicable: SYSTEM ERROR.');
            FlushOutputBuffer(process);
            Terminate(process);
    |   BOUNDSVIOLATION:
            WriteLiteral(process, 'Attempt to access illegal memory location.');
            FlushOutputBuffer(process);
            Terminate(process);
    |   CHECKSCHEDULE:
            ExecuteScheduledEvents;
    |   Halt:
            WriteLiteral(process, 'User initiated abort.');
            FlushOutputBuffer(process);
            Terminate(process);
    |   INITIALIZE:
            InitOSContext;
    |   MODEVIOLATION:
            WriteLiteral(process, 'Attempt to execute privileged instruction.');
            FlushOutputBuffer(process);
            Terminate(process);
    |   OUTofRANGE:
            WriteLiteral(process, 'Attempt to access non-existent memory location.');
            FlushOutputBuffer(process);
            Terminate(process);
    |   SHOULDERTAP:
            ShoulderTap;
    |   STACKOVERFLOW:
            WriteLiteral(process, 'Stack overflow.');
            FlushOutputBuffer(process);
            Terminate(process);
    |   SVC:
            ProcessSVC;
    | UNDEFINEDINSTRUCTION:
        WriteLiteral(process, 'Reported error UNDEFINEDINSTRUCTION currently ');
        WriteLiteral(process, 'inapplicable: SYSTEM ERROR.');
        FlushOutputBuffer(process);
        Terminate(process);
    END;
END ProcessTrap;


PROCEDURE ReadFromInput(process: ProcessID; bufferptr: ADDRESS;
    bytecount: CARDINAL);
(*   Copy the specified number of characters from the input file of the given        *)
(*   process to the memory location starting at the given address. Start with the    *)
(*   first input character not yet read.                                              *)
(*                                                                                    *)
(*   PARAMETERS:   process–the process whose input file is to be read.                *)
(*                 bufferptr–the starting address of the variable                     *)
(*                     into which the input should be read.                           *)
(*                 bytecount–the number of characters to be read.                     *)

    VAR
        buffer:         (* the page pointed to by bufferptr *)
            Page;
        byte:           (* next byte to be read from current page *)
            [0 .. BYTESPERPAGE];
        i,
        j:              (* loop indices *)
            CARDINAL;
        inputpage:      (* the current page of input *)
            Page;
        pageaddress,    (* address of current input page *)
        tempptr:        (* used in type conversions *)
            Pageptr;
        tempaddress:    (* used in type conversions *)
            ADDRESS;

BEGIN

    (* initialize *)
    GetNextInput(process, pageaddress, byte);
    inputpage := pageaddress↑;
    tempptr := Pageptr(bufferptr);
    buffer := tempptr↑;
    i := 0;
    j := 0;
    (* transfer characters until finished or EOF *)
    WHILE (j < bytecount) & (inputpage[byte] # ENDofINPUT) DO
        buffer[i] := inputpage[byte];
        INC(i);
        INC(j);
        INC(byte);

        (* need new page of input? *)
        IF byte = BYTESPERPAGE THEN
            byte := 0;
            tempaddress := ADDRESS(pageaddress);
            INC(tempaddress, PAGESIZE);
            pageaddress := Pageptr(tempaddress);
            inputpage := pageaddress↑;
        END; (* IF byte *)

        (* need new page of buffer? *)
        IF i = BYTESPERPAGE THEN
            i := 0;
            INC(bufferptr, PAGESIZE);
            tempptr := Pageptr(bufferptr);
            buffer := tempptr↑;
        END; (* IF i *)

    END; (* WHILE *)

    (* test for EOF *)
    IF (inputpage[byte] = ENDofINPUT) & (j < bytecount) THEN
        WriteLiteral(process, 'Attempt to read past end of file.');
        FlushOutputBuffer(process);
        Terminate(process);
    ELSE
        StoreNextInput(process, pageaddress, byte);
    END; (* IF inputpage *)
END ReadFromInput;


(*      Respond to completion of a disk write.                                  *)
PROCEDURE Wakeup(process: WORD);
(*                                                                              *)
(*      PARAMETERS:     process--the process ID of the process to be awakened,  *)
(*                          cast as type WORD.                                  *)

BEGIN
    ChangeStatus( ProcessID(process), PENDING );
    Reschedule( ProcessID(process) );
END Wakeup;
```

```
PROCEDURE WriteLiteral(process: ProcessID; string: ARRAY OF CHAR);
(*      Place a literal string into a parameter, so that its address can be used as      *)
(*      a parameter to WriteToOutput. It is assumed that literal strings are              *)
(*      written to user's output only in the case of terminal failure of a program.       *)
(*                                                                                        *)
(*      PARAMETERS:     string–the characters to be written.                              *)

BEGIN
    WriteToOutput( process, ADR(string), SIZE(string) );
END WriteLiteral;


PROCEDURE WriteToOutput(process: ProcessID; bufferptr: ADDRESS;
    bytecount: CARDINAL);
(*      Copy the specified number of characters (maximum 128) to the output file          *)
(*      of the given process from the memory location starting at the given address.      *)
(*      Start immediately after the last output character previously written.             *)
(*                                                                                        *)
(*      PARAMETERS:     process–the process whose output file is to be written.           *)
(*                      bufferptr–the starting address of the variable from which         *)
(*                          the output should be written.                                 *)
(*                      bytecount–the number of characters to be written.                 *)

    VAR
        buffer:             (* the page pointed to by bufferptr *)
            Page;
        byte:               (* next byte of current output page to be written *)
            [0 .. BYTESPERPAGE];
        diskblock:          (* block ID to which output will be written *)
            DiskblockID;
        i:                  (* loop indices *)
            CARDINAL;
        memoryblock,        (* memoryblock associated with current output page *)
        oldmemoryblock:     (* memoryblock associated with previous output page *)
            MemoryblockID;
        outputpage:         (* current memory page receiving output *)
            Page;
        pagewrite:          (* becomes TRUE if output page must be written *)
            BOOLEAN;
        pageaddress,        (* address of current output page *)
        tempptr:            (* used in type conversions *)
            Pageptr;

BEGIN

    (* initialize *)
    pagewrite := FALSE;
    GetNextOutput(process, pageaddress, byte, memoryblock);
```

```
        outputpage := pageaddress↑;
        tempptr := Pageptr(bufferptr);
        buffer := tempptr↑;
        IF bytecount > BYTESPERPAGE THEN
            bytecount := BYTESPERPAGE;
        END; (* IF bytecount *)

        (* transfer characters *)
        FOR i := 0 TO bytecount - 1 DO
            outputpage[byte] := buffer[i];
            INC(byte);

            (* need new output page? *)
            IF byte = BYTESPERPAGE THEN
                pagewrite := TRUE;
                oldmemoryblock := memoryblock;
                StoreNextOutput(process, pageaddress, byte);
                GetNextOutput(process, pageaddress, byte, memoryblock);
            END; (* IF byte *)
        END; (* FOR *)

        StoreNextOutput(process, pageaddress, byte);

        IF pagewrite THEN
            diskblock := Allocate(PAGESIZE);
            DiskWrite(oldmemoryblock, diskblock, Wakeup, process);
            LinkToOutput(process, diskblock);
            Block;
        END; (* IF pagewrite *)
    END WriteToOutput;

BEGIN (* module initialization *)
    SetInterruptHandler(TRAP, ProcessTrap);
END TrapHandler.
```

```
DEFINITION MODULE VirtualMachine;

(************************************************************************)
(*                                                                      *)
(*  FUNCTION:   This module presents a "higher level" machine than that of    *)
(*              LocalSystem. VirtualMachine is a low-level module, dependent  *)
(*              upon LocalSystem; however, modules may import from            *)
(*              VirtualSystem and remain portable. NO OTHER MODULE            *)
(*              SHOULD IMPORT ANYTHING DIRECTLY FROM                          *)
(*              LOCALSYSTEM, as that would diminish the portability of        *)
(*              the entire system. VirtualMachine defines the machine         *)
(*              as visible to the rest of the operating system.               *)
(*                                                                      *)
(*  AUTHOR:     Rick Fisher                                             *)
(*                                                                      *)
(************************************************************************)

    FROM SYSTEM IMPORT
        (* Constants *)
        BYTESPERWORD,

        (* Types *)
        ADDRESS;

    FROM LocalSystem IMPORT
        (* Constants *)
        HIGHMCODEINSTRUCTION, WORDSINMEMORY;

    EXPORT QUALIFIED
        (* Constants *)
        BYTESPERPAGE, BYTESPERSECTOR, HIGHINSTRUCTION, MEMORYSIZE,

        (* Types *)
        AllTraps, ContextID, Interruptcode, Mcodeinstruction, OSTraps, SVCcode,

        (* Procedures *)
        ContextBounds, ContextSize, DiskRead, DiskWrite, HighOSBound,
        InitOSContext, LowerStackLimit, NewContext, Read, ReturnFromInterrupt,
        SetInterruptHandler, SetPC, SVCArguments, SwitchContext, Trap,
        TrapReason, UpdateContext, Write;

    CONST
        BYTESPERPAGE     =  128;
        BYTESPERSECTOR   =  128;
        HIGHINSTRUCTION  =  HIGHMCODEINSTRUCTION; (* 377 octal *)
        MEMORYSIZE       =  WORDSINMEMORY DIV BYTESPERPAGE
                               * BYTESPERWORD;       (* 64K bytes *)
```

```
TYPE
    AllTraps            =  (BADINSTRUCTION, CHECKSCHEDULE,
                            Halt,INITIALIZE, SHOULDERTAP, SVC,
                            UNDEFINEDINSTRUCTION, BOUNDSVIOLATION,
                            MODEVIOLATION, OUTofRANGE,
                            STACKOVERFLOW, VALUERANGE);
    ContextID;             (* pointer to the actual context *)
    Interruptcode       =  (TRAP, CARDREADER, LINEPRINTER, DISK, CLOCK);
    Mcodeinstruction    =  [0 .. HIGHINSTRUCTION];
    OSTraps             =  [BADINSTRUCTION .. UNDEFINEDINSTRUCTION];
        (* all other traps are called directly by the machine *)

    SVCcode             =  (DISKALLOCATESVC, DISKDISPOSESVC, DISKREADSVC,
                            DISKWRITESVC, HEAPALLOCATESVC, READSVC,
                            SLEEPSVC, WRITESVC,UPPERBOUNDSVC);

PROCEDURE ContextBounds(context: ContextID; VAR low, high: ADDRESS);
(*      Find the limits of memory to which the context refers.                *)
(*                                                                            *)
(*      PARAMETERS:     context–the ID of the context in question.            *)
(*                      low–the lowest address of the context.                *)
(*                      high–the highest address of the context.              *)


PROCEDURE ContextSize(): CARDINAL;
(*      RETURNS the size in words of a context.                               *)


PROCEDURE DiskRead(disksector: CARDINAL; memoryaddress: ADDRESS);
(*      Start the disk and return. The disk will run concurrently with        *)
(*      the CPU and transfer the given sector to the given memory location.   *)
(*      If the procedure is called before being notified by a device interrupt *)
(*      that the disk has finished it's last read, the command may be         *)
(*      lost.                                                                 *)
(*                                                                            *)
(*      PARAMETERS:     diskaddress–the sector of the disk from which data is *)
(*                          to be read.                                       *)
(*                      memoryaddress–the starting address of memory into     *)
(*                          which data is to be loaded.                       *)
```

```
PROCEDURE DiskWrite(memoryaddress: ADDRESS; diskaddress: CARDINAL);
(*      Start the disk and return. The disk will run concurrently with the        *)
(*      CPU and transfer the memory block starting at the given address to        *)
(*      the given disk sector. If the procedure is called before being notified by *)
(*      a device interrupt that the disk has finished it's last write, the         *)
(*      command may be lost.                                                      *)
(*                                                                                *)
(*      PARAMETERS:     memoryaddress–the starting address of memory              *)
(*                          from which data is to be fetched.                     *)
(*                      diskaddress–the sector of the disk to which data          *)
(*                          is to be written.                                     *)


PROCEDURE HighOSBound(): ADDRESS;
(*      RETURNS the highest address occupied by the operating system.             *)


PROCEDURE InitOSContext;
(*      Set certain values in the operating system's context. Ineffective if      *)
(*      not called during an INITIALIZE trap. Must be called during system        *)
(*      initialization.                                                           *)


PROCEDURE LowerStackLimit(context: ContextID; amount: CARDINAL): BOOLEAN;
(*      Attempt to lower the context's stack limit by the given amount in words,  *)
(*      thus giving more room for heap space.                                     *)
(*                                                                                *)
(*      PARAMETERS:     context–the ID of the context whose stack limit is        *)
(*                          to be lowered.                                        *)
(*                      amount–the number of words the stack limit is to be       *)
(*                          lowered.                                              *)
(*      RETURNS TRUE if successful, FALSE otherwise.                              *)


PROCEDURE NewContext(processbase, stackbase: ADDRESS;
    stacksize: CARDINAL): ContextID;
(*      Create a new context. The caller must insure that the code frames and     *)
(*      stacks of the contexts do not overlap.                                    *)
(*                                                                                *)
(*      PARAMETERS:     processbase–starting address available in the context.    *)
(*                      stackbase–starting address of the stack.                  *)
(*                      stacksize–size of the stack in words.                     *)
(*      RETURNS  a context ID for the context.                                    *)
```

```
PROCEDURE Read(): CHAR;
(*      Get the value of the character currently in the reserved input buffer       *)
(*      memory location, and start the card reader getting the next character.      *)
(*      Calling this procedure before being notified by a device interrupt          *)
(*      that the card reader has finished may result in lost and/or duplicated      *)
(*      characters.                                                                 *)
(*                                                                                  *)
(*      RETURNS the character most recently read by the card reader.                *)


PROCEDURE ReturnFromInterrupt(type: Interruptcode);
(*      Return to an interrupted process. The context of the current process        *)
(*      is not saved, as this instruction should only be executed when the          *)
(*      current process has finished. (The context of the operating system is       *)
(*      permanently stored in a reserved memory location.)                          *)
(*                                                                                  *)
(*      PARAMETERS:     type–the type of interrupt being concluded.                 *)


PROCEDURE SetInterruptHandler(type: Interruptcode; handler: PROC);
(*      Set the second word of an interrupt vector to the address of the correct    *)
(*      interrupt handling routine.                                                 *)
(*                                                                                  *)
(*      PARAMETERS:     type–the type of interrupt handler being set.               *)
(*                      handler–the address of the handling routine.                *)


PROCEDURE SetPC(context: ContextID; PC: ADDRESS);
(*      Change the PC of a non-running context. The next time there is a            *)
(*      context switch to this context, instruction execution will start from       *)
(*      the altered PC.                                                             *)
(*                                                                                  *)
(*      PARAMETERS:     context–the ID of the context whose PC is to be changed.    *)
(*                      PC–the address to which the context's PC should be          *)
(*                          changed. The caller must ensure that the PC is          *)
(*                          within the code frame of the context.                   *)


PROCEDURE SVCArguments(context: ContextID):     ADDRESS;
(*      Obtain from a previously executing context the arguments of the Supervisor  *)
(*      Call that caused the context switch. Should be used only if the context     *)
(*      switch was caused by a SVC.                                                 *)
(*                                                                                  *)
(*      PARAMETERS:     context–the ID of the previously executing context.         *)
(*      RETURNS  a pointer to the SVC arguments. If the context switch was          *)
(*              caused by something other than an SVC, the pointer returned         *)
(*              will probably be NIL, but it may point to nonsense values.          *)
(*              Otherwise, the values will be as follows:                           *)
(*              WORD[0] is always the ordinal value of the SVC.                     *)
(*              If WORD[0] is DISKALLOCATESVC or DISKDISPOSESVC,                *)
(*                  then WORD[1] is a pointer to the ID of the disk sector      *)
(*                  to be allocated or freed.                                   *)
(*              If WORD[0] is READSVC or WRITESVC, then WORD[1]                 *)
(*                  is the starting address involved in the transfer,           *)
(*                  and WORD[2] is the number of bytes (characters) to be       *)
(*                  transfered.                                                 *)
(*              If WORD[0] is HEAPALLOCATESVC, then WORD[1] is the              *)
(*                  number of words to be allocated, and WORD[2] is a           *)
(*                  pointer to a success flag.                                  *)
(*              IF WORD[0] is SLEEPSVC, then WORD[1] is a pointer to            *)
(*                  the amount of time the process should sleep.                *)
(*              IF WORD[0] is UPPERBOUNDSVC, then WORD[1] is a                  *)
(*                  pointer to the process's uppermost address.                 *)


PROCEDURE SwitchContext(context: ContextID);
(*      Set the return context of the current trap interupt, so that when the  *)
(*      trap concludes, the new context will take control of the CPU.          *)
(*                                                                              *)
(*      PARAMETERS:     context–the ID of the context to be given control of   *)
(*                              the CPU.                                        *)


PROCEDURE Trap(reason: OSTraps);
(*      Force an interrupt, causing the trap handling routine to be called with *)
(*      ''reason'' as a parameter.                                              *)
(*                                                                              *)
(*      PARAMETERS:     reason–the cause of the trap. Valid reasons are:        *)
(*          BADINSTRUCTION:         illegal characters in instruction.          *)
(*          CHECKSCHEDULE:          execute events scheduled for                *)
(*                                      any processes.                          *)
(*          Halt:                   abort process.                              *)
(*          INITIALIZE:             set up initial OS context. Only             *)
(*                                      call during initialization.             *)
(*          SHOULDERTAP:            see if a job can be changed to a new        *)
(*                                      process.                                *)
(*          SVC:                    supervisor call.                            *)
(*          UNDEFINEDINSTRUCTION:   no such M-Code instruction defined.         *)


PROCEDURE TrapReason(context: ContextID): AllTraps;
(*      Find the reason for the given context having been trapped.             *)
(*                                                                              *)
(*      PARAMETERS:     context–the ID of the context which was trapped.       *)
(*      RETURNS the type of trap.                                              *)
```

```
    PROCEDURE UpdateContext(context: ContextID; offset: INTEGER);
    (*    Change all absolute addresses in the context by a specified amount.       *)
    (*                                                                              *)
    (*    PARAMETERS:    context–the ID of the context being changed.               *)
    (*                   offset–the amount by which the addresses should be         *)
    (*                      altered.                                                *)


    PROCEDURE Write(character: CHAR);
    (*    Instruct the line printer to print a character. Calling this procedure    *)
    (*    will overwrite the current contents of the reserved output buffer memory  *)
    (*    location. Calling it without waiting for notification from a device       *)
    (*    interrupt that the line printer has finished will result in lost          *)
    (*    characters.                                                               *)
    (*                                                                              *)
    (*    PARAMETERS:    character–the character to be written.                     *)

END VirtualMachine.
```

```
IMPLEMENTATION MODULE VirtualMachine;

(*********************************************************************)
(*                                                                   *)
(*  POLICY:       Relay requests directly to LocalSystem procedures whenever  *)
(*                possible.                                          *)
(*                                                                   *)
(*  AUTHOR:       Rick Fisher                                        *)
(*                                                                   *)
(*********************************************************************)

    FROM SYSTEM IMPORT
        (* Constants *)
        BYTESPERWORD,

        (* Types *)
        ADDRESS, SIZE,

        (* Procedures *)
        ADR;

    IMPORT LocalSystem;


    CONST
        HIGHSVC     =   WRITESVC;

    TYPE
        ContextID   =   POINTER TO LocalSystem.Context;


    (*  Get the context address boundaries. *)
    PROCEDURE ContextBounds(context: ContextID; VAR low, high: ADDRESS);

    BEGIN
        low := context↑.segmenttable;
        high := context↑.stacklimit;
    END ContextBounds;


    (*  Find the size of a context.    *)
    PROCEDURE ContextSize(): CARDINAL;

        VAR
            context:(* any context *)
                ContextID;

    BEGIN
        RETURN SIZE(context↑);
    END ContextSize;
```

```
(*  Read a sector from disk.      *)
PROCEDURE DiskRead(disksector: CARDINAL; memoryaddress: ADDRESS);

BEGIN
    LocalSystem.DiskRead(disksector, memoryaddress);
END DiskRead;


(*  Write a sector to disk.   *)
PROCEDURE DiskWrite(memoryaddress: ADDRESS; disksector: CARDINAL);

BEGIN
    LocalSystem.DiskWrite(memoryaddress, disksector);
END DiskWrite;


(*  Find the high OS bound.      *)
PROCEDURE HighOSBound(): ADDRESS;

    VAR
        oscontext:          (* operating system context *)
            ContextID;
BEGIN
    oscontext := ContextID(LocalSystem.OScontext);
    RETURN oscontext↑.upperbound;
END HighOSBound;


(*  Initialize the operating system's context.      *)
PROCEDURE InitOSContext();

    VAR
        context,        (* the operating system context *)
        nullcontext:    (* the null process context *)
            ContextID;

BEGIN
    context := ContextID(LocalSystem.OScontext);
    WITH context↑ DO

        (* allow 2K bytes for stack and heap *)
        upperbound := stacktop + 16 * BYTESPERPAGE DIV BYTESPERWORD;
        stacklimit := upperbound - SIZE(context↑);
    END; (* WITH *)

    (* create null context at top of heap identical to OS context *)
    nullcontext := ContextID(context↑.stacklimit + 1);
    nullcontext↑ := LocalSystem.Context(context↑);
    SwitchContext(nullcontext);

END InitOSContext;
```

```
(* Lower the stack limit to make more room for heap.   *)
PROCEDURE LowerStackLimit(context: ContextID; amount: CARDINAL): BOOLEAN;

BEGIN
    WITH context↑ DO
        IF CARDINAL(stacklimit - stacktop) >= amount THEN
            DEC(stacklimit, amount);
            RETURN TRUE;
        ELSE
            RETURN FALSE;
        END; (* IF *)
    END; (* WITH *)
END LowerStackLimit;


(* Create a new context.   *)
PROCEDURE NewContext(processbase, stackbase: ADDRESS;
    stacksize:CARDINAL): ContextID;

    VAR
        context:            (* the context being created *)
            ContextID;
        i:                  (* loop index *)
            CARDINAL;

BEGIN

    (* locate context at bottom of process stack *)
    context := ContextID(stackbase);

    WITH context↑ DO
        INC( stackbase, SIZE(context↑) );
        DEC( stacksize, SIZE(context↑) );
        dataframe := processbase + SIZE(segmenttable↑);
        currentactivation := stacktop;
        PC := ADDRESS(dataframe↑.codeframe↑);
        interruptmask := {15};       (* mode bit *)
        stacktop := stackbase + SIZE(context↑);
        stacklimit := stackbase + stacksize;
        segmenttable := processbase;
        upperbound := stacklimit;

        (* set correct addresses in the segment table *)
        FOR i := 0 TO SIZE(segmenttable↑) - 1 DO
            INC( segmenttable↑[i], processbase);
        END; (* FOR *)
    END; (* WITH *)

    RETURN context;
END NewContext;
```

```
(* Read a character from a card.  *)
PROCEDURE Read(): CHAR;

    VAR
        temp:           (* temporary holder for the input buffer *)
            CHAR;

BEGIN
    temp := LocalSystem.inputbuffer;
    LocalSystem.Read;
    RETURN temp;
END Read;


(* Return to the interrupted process.    *)
PROCEDURE ReturnFromInterrupt(type: Interruptcode);

BEGIN
    LocalSystem.ContextSwitch(LocalSystem.interruptvector[ORD(type) + 7].formercontext);
END ReturnFromInterrupt;


(* Set the interrupt handler.    *)
PROCEDURE SetInterruptHandler(type: Interruptcode; routine: PROC);

BEGIN
    LocalSystem.interruptvector[ ORD(type) + LocalSystem.LOWINTERRUPT ].interrupthandler
        := routine;
END SetInterruptHandler;


(* Set PC in a context.      *)
PROCEDURE SetPC(context: ContextID; PC: ADDRESS);

BEGIN
    context↑.PC := PC;
END SetPC;


(* Get the address of the argument to a Supervisor Call.     *)
PROCEDURE SVCArguments(context: ContextID): ADDRESS;

    VAR
        argaddress:     (* the address of the first argument *)
            ADDRESS;

BEGIN
    (* first find location of address of previous activation record *)
    argaddress := context↑.currentactivation + 1;
    (* now get the address of its parameters *)
    argaddress := ADDRESS(argaddress↑) + 4;

    (* make sure it was an SVC trap *)
    IF CARDINAL(argaddress↑) <= CARDINAL(HIGHSVC) THEN
        RETURN argaddress;
    ELSE
        RETURN NIL;
    END; (* IF *)

END SVCArguments;


(*  Switch the context.  *)
PROCEDURE SwitchContext(context: ContextID);

BEGIN
    LocalSystem.interruptvector[LocalSystem.TRAP].formercontext := ADDRESS(context);
END SwitchContext;


(*  Trap signaler.  *)
PROCEDURE Trap(reason: OSTraps);

BEGIN
    CASE reason OF
        BADINSTRUCTION:
            LocalSystem.Trap(LocalSystem.BADINSTRUCTION);
    |   CHECKSCHEDULE:
            LocalSystem.Trap(LocalSystem.CHECKSCHEDULE);
    |   Halt:
            LocalSystem.Trap(LocalSystem.Halt);
    |   INITIALIZE:
            LocalSystem.Trap(LocalSystem.INITIALIZE);
    |   SHOULDERTAP:
            LocalSystem.Trap(LocalSystem.SHOULDERTAP);
    |   SVC:
            LocalSystem.Trap(LocalSystem.SVC);
    |   UNDEFINEDINSTRUCTION:
            LocalSystem.Trap(LocalSystem.UNDEFINEDINSTRUCTION);
    ELSE END; (* CASE *)
END Trap;
```

```
(*  Find the reason for a trap.   *)
PROCEDURE TrapReason(context: ContextID): AllTraps;

BEGIN
    CASE context↑.trap OF
        LocalSystem.BADINSTRUCTION:
            RETURN BADINSTRUCTION;
    |   LocalSystem.BOUNDSVIOLATION:
            RETURN BOUNDSVIOLATION;
    |   LocalSystem.CHECKSCHEDULE:
            RETURN CHECKSCHEDULE;
    |   LocalSystem.Halt:
            RETURN Halt;
    |   LocalSystem.INITIALIZE:
            RETURN INITIALIZE;
    |   LocalSystem.MODEVIOLATION:
            RETURN MODEVIOLATION;
    |   LocalSystem.OUTofRANGE:
            RETURN OUTofRANGE;
    |   LocalSystem.SHOULDERTAP:
            RETURN SHOULDERTAP;
    |   LocalSystem.STACKOVERFLOW:
            RETURN STACKOVERFLOW;
    |   LocalSystem.SVC:
            RETURN SVC;
    |   LocalSystem.UNDEFINEDINSTRUCTION:
            RETURN UNDEFINEDINSTRUCTION;
    |   LocalSystem.VALUERANGE:
            RETURN VALUERANGE;
    END; (* CASE *)
END TrapReason;


(*  Change the addresses in context by the offset.     *)
PROCEDURE UpdateContext(context: ContextID; offset:   INTEGER);

    VAR
        i:                  (* loop index *)
            CARDINAL;
        datatemp,
        segmenttemp:        (* for type conversion purposes *)
            ADDRESS;
```

```
BEGIN
    WITH context↑ DO
        segmenttemp := ADDRESS(segmenttable);
        datatemp := ADDRESS(dataframe);
        INC(segmenttemp, offset);
        INC(datatemp, offset);
        INC(currentactivation, offset);
        INC(stacktop, offset);
        INC(stacklimit, offset);
        segmenttable := LocalSystem.Segmenttableptr(segmenttemp);
        dataframe := LocalSystem.Dataframeptr(datatemp);

        (* update addresses in segment table *)
        FOR i := 0 TO SIZE(segmenttable↑) DO
            INC( segmenttable↑[i], offset);
        END; (* FOR *)
    END; (* WITH *)

END UpdateContext;


(*  Send a character to the printer.  *)
PROCEDURE Write(character: CHAR);

BEGIN
    LocalSystem.outputbuffer := character;
    LocalSystem.Write;
END Write;


BEGIN
END VirtualMachine.
```